

WORDWARE GAME DEVELOPER'S LIBRARY



Programming Multiplayer Games



Companion
CD-ROM
Included



**Andrew Mulholland
and Teijo Hakala**

Programming Multiplayer Games

**Andrew Mulholland
and Teijo Hakala**

Wordware Publishing, Inc.

Library of Congress Cataloging-in-Publication Data

Mulholland, Andrew.

Programming multiplayer games / by Andrew Mulholland and Teijo Hakala.
p. cm.

Includes index.

ISBN 1-55622-076-6 (pbk.; companion cd-rom)

1. Computer games--Programming. I. Hakala, Teijo. II. Title.

QA76.76.C672M855 2004

794.8'1711--dc22

2003027637

CIP

© 2004, Wordware Publishing, Inc.

All Rights Reserved

2320 Los Rios Boulevard

Plano, Texas 75074

No part of this book may be reproduced in any form or by any means
without permission in writing from Wordware Publishing, Inc.

Printed in the United States of America

ISBN 1-55622-076-6

10 9 8 7 6 5 4 3 2 1

0403

UNIX is a registered trademark of The Open Group in the United States and other countries.

All brand names and product names mentioned in this book are trademarks or service marks of their respective companies. Any omission or misuse (of any kind) of service marks or trademarks should not be regarded as intent to infringe on the property of others. The publisher recognizes and respects all marks used by companies, manufacturers, and developers as a means to distinguish their products.

This book is sold as is, without warranty of any kind, either express or implied, respecting the contents of this book and any disks or programs that may accompany it, including but not limited to implied warranties for the book's quality, performance, merchantability, or fitness for any particular purpose. Neither Wordware Publishing, Inc. nor its dealers or distributors shall be liable to the purchaser or any other person or entity with respect to any liability, loss, or damage caused or alleged to have been caused directly or indirectly by this book.

All inquiries for volume purchases of this book should be addressed to Wordware Publishing, Inc., at the above address. Telephone inquiries may be made by calling:

(972) 423-0090

Contents

About the Authors. xiv
Introduction xv

Part I — Theory

Chapter 1 Introduction to Windows Programming. 3
Introduction 3
Windows Messaging System 3
Creating a Window 4
Sending Information to Your Window 8
Static Link Libraries 9
 Creating a Static Link Library 10
 Using a Static Link Library. 13
Summary 14

Chapter 2 Using Databases 15
Introduction 15
What Is MySQL?. 15
Installing MySQL 16
SQL Statements 19
Data Definition Language 19
 Creating and Dropping Databases 19
 Creating a Database 20
 Dropping a Database. 21
 Column (Field) Types in MySQL 22
 Adding, Modifying, and Dropping Tables 24
 Creating Tables. 24
 Modifying Tables. 26
 Dropping (Removing) Tables 28
Data Manipulation Language (DML) 29
 Inserting Data. 30
 Modifying Data 32
 Removing (Deleting) Data 34
 Using Select Statements 35
Relational Databases. 40
Data Import Methods 44
 Importing from a Text File 44

Importing from a Native Source	45
Backing Up and Restoring Data.	47
Backing up a Database to a File	47
Restoring a Backed-Up Database	49
MySQL C++ Interface	50
Example 1 — Connecting and Retrieving Data from MySQL	50
Example 2 — Updating Data in MySQL from an Application	53
Summary	55
Chapter 3 Creating Web-Based Server Interfaces	57
Introduction	57
Setting Up an Apache 1.3.x Web Server	57
Installing PHP4 for Apache 1.3.x	60
Using PHP: Hypertext Preprocessor.	63
The Basics.	64
Example 1 — index.php	64
Variables.	65
Example 2 — index2.php	65
Operators and Loops	67
Example 3 — index3.php	67
Conditional Statements.	68
Example 4 — index4.php	68
Arrays	70
Functions	70
Example 5 — index5.php	70
User Input.	71
Example 6a — input.php.	71
Example 6b — output.php.	72
The “Command” System	73
Example 7a — core.php	74
Example 7b — welcome.php	74
Example 8a — core.php	76
Example 8b — welcome.php	76
Example 8c — page1.php	77
Example 8d — page2.php	77
Accessing MySQL	78
MySQL Example 1 — Connecting and Disconnecting	78
MySQL Example 2 — Storing and Retrieving Data	79
MySQL Example 3 — Updating and Removing Data.	84
Using FastTemplate	90
Multiple Templates	93
Converting the Command Parser Example to FastTemplate	95
Summary	101
Chapter 4 Introduction to TCP/IP.	103
Introduction.	103

What Is a Protocol?	103
OSI Model.	104
OSI Model Layers	104
Internet Protocol	106
Introduction to the Transport Layer	108
Transmission Control Protocol	108
User Datagram Protocol	109
Ports	109
Introduction to Sockets	110
Socket Types	111
Address	112
Platforms	112
History of WinSock	113
Summary	113
Chapter 5 Basic Sockets Programming	115
Introduction	115
WinSock Initialization	115
WSAStartup Function (Win32)	115
WSACleanup Function (Win32).	116
WSAEnumProtocols Function (Win32)	117
WinSock Initialization Function	117
Error Handling	120
WSAGetLastError Function (Win32)	120
Sockets Data Types	121
Platform-specific Data Types	121
Address Structures	121
IPv4 Address Structure	121
IPv6 Address Structure	122
Generic Address Structure.	123
Basic Sockets Functions	124
socket Function (Unix, Win32)	124
bind Function (Unix, Win32)	125
connect Function (Unix, Win32)	126
listen Function (Unix, Win32)	127
accept Function (Unix, Win32)	128
close Function (Unix)/closesocket Function (Win32)	129
Input/Output Functions	129
send Function (Unix, Win32)	129
recv Function (Unix, Win32)	131
sendto Function (Unix, Win32)	132
recvfrom Function (Unix, Win32).	133
Address Data Conversion Functions	133
inet_aton Function (Unix, Win32)	134
Client/Server Programming	134
Server Methods	134

Clients	135
Byte Ordering.	136
Creating a Server.	136
TCP.	137
UDP	138
Simple Echo TCP Server	139
main Function.	142
InitSockets Function	142
ServerProcess Function	145
Simple Echo UDP Server	146
InitSockets Function	148
ServerProcess Function	149
Creating a Client	150
TCP.	150
UDP	151
Simple Echo TCP Client	151
main Function.	153
InitSockets Function	154
ClientProcess Function.	155
Simple Echo UDP Client	157
InitSockets Function	159
ClientProcess Function.	159
Running the Simple Echo Application.	160
Summary	161
Chapter 6 I/O Operations	163
Introduction.	163
Detecting Network Events	163
select (Unix, Win32).	163
Macros	164
WSAAsyncSelect (Win32)	165
WSAEventSelect (Win32).	165
WSAWaitForMultipleEvents (Win32)	166
Event Object.	166
Multithreading	167
What Is Multithreading?	167
CreateThread (Win32)	168
pthread_create (Unix).	169
I/O Strategy.	169
Blocking I/O	169
Non-blocking I/O	170
Signal-driven I/O	170
Multiplexing I/O	171
I/O Control	172
ioctl (Unix)/ioctlsocket (Win32)	172
setsockopt/getsockopt (Unix, Win32)	173

shutdown (Unix, Win32)	174
Broadcasting	175
Searching for Servers	175
Broadcast Function	176
Summary	177

Part II — Tutorials

Tutorial 1 Using 2DLIB	181
Introduction	181
Configuring Visual Studio	181
Creating a Skeleton Project	182
Creating the Workspace	182
Adding the Static Libraries	183
Adding the Source File	183
Creating a Basic Windowed Application with 2DLIB	184
The WinMain Function	184
The Windows Procedure	185
The Complete Code	187
Using the 2DLIB Graphics Routines	190
2D Positions on the Screen	190
Use of Colors	190
Plotting a Single Pixel	191
Drawing a Line	191
Drawing a Rectangle/Filled Rectangle	191
Drawing a Triangle/Filled Triangle	191
Graphic Loading Functions	192
Graphics Display (Blitting) Function	192
Keyboard Input Method	193
2DLIB Example 1 — Moving Primitives with the Cursor Keys	195
Complete Code Listing for Example 1	197
2DLIB Example 2 — Loading and Rotating Graphics	200
Complete Code Listing for Example 2	201
Summary	203
Tutorial 2 Creating Your Network Library	205
Introduction	205
Why Create a Network Library of Our Own?	206
Planning the Structure	206
Planning the Functionality	207
Identifying Hosts	207
Sending Data to Hosts	208
Pinging — Calculating Network Latency	209
Timing Out	209
Building the Library	209
Windows	210

Unix/Linux	211
Creating Independent Code	212
Creating Definitions for Data Types	212
Log System	213
StartLogConsole Function	214
dreamConsole Constructor	215
dreamConsole Destructor	215
println Function	215
StartLog Function	216
LogString Function	217
StopLog Function	218
Getting Started	219
Setting Up Source and Header Files	219
dreamSock.h File	220
dreamMessage Class	225
dreamClient Class	226
dreamServer Class	228
Global Setup Functions	229
dreamSock_Initialize	229
dreamSock_InitializeWinSock	230
dreamSock_Shutdown	231
Global Socket Functions	232
dreamSock_Socket Function	238
dreamSock_SetNonBlocking Function	239
dreamSock_SetBroadcasting Function	239
dreamSock_StringToSockaddr Function.	240
dreamSock_OpenUDPSocket Function	241
dreamSock_CloseSocket Function.	243
dreamSock_GetPacket Function.	243
dreamSock_SendPacket Function	244
dreamSock_Broadcast Function	245
dreamSock_GetCurrentSystemTime Function	246
dreamSock_Linux_GetCurrentSystemTime Function	246
dreamSock_Win_GetCurrentSystemTime Function	247
Retrieving Error Values.	248
Summary of Global Functions	248
Creating dreamSock Network Library	248
dreamMessage Class Member Variables.	249
dreamMessage Class Functionality	249
Init Function	253
Clear Function	253
GetNewPoint Function.	254
AddSequences Function	255
Write Function	255
WriteByte Function.	255
WriteShort Function	256

WriteLong Function	256
WriteFloat Function	256
WriteString Function	257
BeginReading Function.	257
Read Function.	257
ReadByte Function	258
ReadShort Function	258
ReadLong Function	259
ReadFloat Function	259
ReadString Function	259
dreamMessage Summary	260
dreamClient Class Member Variables	260
dreamClient Class Functionality	262
dreamClient Constructor.	268
dreamClient Destructor	268
Initialize Function.	268
Uninitialize Function	269
Reset Function	270
DumpBuffer Function	270
System Messages vs. User Messages.	270
SendConnect Function	271
SendDisconnect Function	272
SendPing Function	272
ParsePacket Function.	273
GetPacket Function.	275
SendPacket Function (Internal Message).	276
SendPacket Function (External Message)	278
dreamClient Summary	279
dreamServer Class Member Variables	280
dreamServer Class Functionality.	280
dreamServer Constructor	289
dreamServer Destructor	289
Initialize Function.	290
Uninitialize Function	290
SendAddClient Function	290
SendRemoveClient Function.	292
SendPing Function	293
AddClient Function	293
RemoveClient Function	295
ParsePacket Function.	297
CheckForTimeout Function	299
GetPacket Function.	301
SendPackets Function	303
dreamServer Summary.	304
Summary	304

Tutorial 3	Creating a Basic Network Application with dreamSock	305
Introduction		305
Planning the Functionality		306
Catching Exceptions		306
Creating a Basic Client Application		306
signin.h File		307
CSignin Class		308
network.h File		309
main.h File		309
common.h File		310
main.cpp File		310
Global Variables		316
CreateAccountDialogProc Function		316
WinMain Function		319
signin.cpp File — CSignin Class Methods		323
CSignin Constructor		329
CSignin Destructor		329
ReadPackets Function		329
AddClient Function		333
RemoveClient Function		334
RemoveClients Function		335
SendSignIn Function		336
SendKeepAlive Function		336
Connect Function		337
Disconnect Function		337
RunNetwork Function		337
Creating a Basic Server Application		338
signin.h File		338
CSigninServer Class		339
network.h File		340
common.h File		340
main.cpp File		341
Global Variables		347
WindowProc Function		347
WinMain Function		347
daemonInit Function		351
keyPress Function		352
main Function		353
signin.cpp File — CSigninServer Class Methods		355
CSigninServer Constructor		361
CSigninServer Destructor		361
InitNetwork Function		362
ShutdownNetwork Function		362
ReadPackets Function		362
SendExitNotification Function		366

AddClient Function	366
RemoveClient Function	367
RemoveClients Function	368
RunNetwork Function	369
Summary	369
Tutorial 4 Creating the Game Lobby	371
Introduction	371
Creating the Lobby Client Application	371
Creating the Dialogs	372
Lobby Dialog	372
Create Game Dialog	375
Create View Players Dialog	376
Join Game Dialog	376
Lobby System Code	377
Lobby Client Code	377
lobby.h File	377
network.h File	379
main.h File	379
main.cpp File	380
CreateViewPlayersDialogProc Function	380
CreateGameDialogProc Function	380
JoinGameDialogProc Function	382
LoginDialogProc Function	382
LobbyDialogProc Function	384
WinMain Function	386
lobby.cpp File — CLobby Class Methods	389
RefreshPlayerList Function	398
ReadPackets Function	399
RequestGameData Function	401
SendChat Function	402
SendCreateGame Function	402
SendRemoveGame Function	402
SendStartGame Function	403
Connect Function	403
Disconnect Function	403
RunNetwork Function	404
Unimplemented Functions	405
Lobby Server Code	406
lobby.h File	406
network.h File	407
main.cpp File	407
lobby.cpp File — CLobbyServer Class Methods	408
ReadPackets Function	418
Unimplemented Functions	425
Summary	425

Tutorial 5	Creating Your Online Game	427
Introduction		427
Designing the Functionality		428
Frame Time		428
Compressing Messages		428
Dead Reckoning		429
Frame History		430
Handling Messages		430
Game Server Code		430
server.h File		431
network.h File		436
main.cpp File		436
network.cpp File — CArmyWarServer Class Part 1		437
InitNetwork Function		449
ReadPackets Function		449
SendCommand Function		451
ReadDeltaMoveCommand Function		452
BuildMoveCommand Function		453
BuildDeltaMoveCommand Function		454
server.cpp File — CArmyWarServer Class Part 2		458
GenerateRandomMap Function		467
CalculateVelocity Function		468
CalculateHeading Function		469
CalculateBulletVelocity Function		470
MovePlayer Function		472
CheckFlagCollisions Function		475
Frame Function		477
lobby.cpp File		478
AddGame Function		478
RemoveGame Function		479
RemoveGames Function		481
Summary of Server Code		481
Game Client Code		481
client.h File		482
network.h File		487
common.h File		488
main.cpp File		488
VectorLength and VectorSubtract Functions		488
ApplicationProc Function		489
Dialog Procedures		489
Main Loop		489
network.cpp File		490
StartConnection Function		502
SendCommand Function		502
SendStartGame Function		504
SendRequestNonDeltaFrame Function		504

Connect Function	504
Disconnect Function	505
ReadMoveCommand Function	505
ReadDeltaMoveCommand Function	506
BuildDeltaMoveCommand Function	508
RunNetwork Function	509
client.cpp File	511
CArmyWar Constructor and Destructor Functions	525
InitializeEngine Function	526
Shutdown Function	527
DrawMap Function	528
Frame Function	530
CheckVictory Function	531
KillPlayer Function	532
GetClientPointer Function	532
CheckKeys Function	532
CheckPredictionError Function	533
CheckBulletPredictionError Function	534
CalculateVelocity Function	535
CalculateHeading Function	536
PredictMovement Function	537
MoveObjects Function	539
lobby.cpp File	541
RefreshGameList Function	541
RefreshJoinedPlayersList Function	541
Other Unimplemented Functions	542
Summary	542
Index	543

About the Authors



Andrew Mulholland

Andrew Mulholland has a BSc (Hons) in Computer Games Technology and is a partner in a games development company based in Scotland called Hunted Cow Studios (www.hunted-cow.com). The company's current project is an online gaming web site called CowPlay.com, which currently offers free multiplayer games.

Teijo Hakala is a software engineer from Jyväskylä, Finland, who specializes in network programming, game programming, and optimization. He also has wide work experience with computer technology.



Teijo Hakala

Introduction

With Internet technology developing rapidly and the use of broadband Internet services increasingly common, Internet computer gaming has become ever more popular, while documentation on how to develop Internet games remains inadequate. *Programming Multiplayer Games* provides in-depth coverage of all the major topics associated with online game programming, as well as giving the programmer easy to follow, step-by-step tutorials on how to create a fully functional network library, back-end MySQL database, and a complete, working online game.

The book contains two main parts. The first explains practical theory on how to utilize MySQL, PHP4, sockets, and basic Windows programming. The second part consists of five extensive tutorials, leading you through the stages of creating a working online game, which you can both learn from and expand upon.

After reading this book, you will have a solid knowledge of online game programming and you will also be able to start making your own online games. Also note that the companion CD contains all the source code from the book and a ready-to-use version of the network library you will create in the tutorial section.

We hope you enjoy reading and learning from this book as much as we have enjoyed writing it!

This page intentionally left blank.

Theory

The theory section of this book is full of practical information that will help you understand how to make functional online games. We recommend that you read through this section thoroughly before attempting the tutorial section, as there is a lot of knowledge that will benefit you here.

This section first covers the basics of dialog-based Windows programming, which we will utilize in the tutorial section to create our login and lobby system for the sample online game. Then we cover how to use MySQL and PHP to create a back-end database for your game, allowing you to interact with game data directly from a web browser. We also give an introduction to TCP/IP and sockets, followed by how to get started with sockets programming. Finally, we learn about different ways to send data and how to modify the behavior of our sockets.

This page intentionally left blank.

Chapter 1

Introduction to Windows Programming

Introduction

The most essential knowledge anyone can have is the basics. If you already know how to create dialog-based Windows applications, you can skip this chapter, but if you do not, this chapter will give you a quick and easy introduction so that you will find the rest of this book more accessible.

Windows Messaging System

Windows controls everything through the use of its messaging system. This is a fundamental idea to grasp if you wish to create any Windows-based applications. Tasks to be processed by the operating system are stored in a queue. For example, when a user clicks a button in a window, a message is added to the queue and is then sent to the appropriate window to inform it that the button has been pressed.

When the operating system creates a window, the window continually checks for messages being sent to it. If it receives a relevant message, it will react accordingly; otherwise, it will send it back to the operating system to be reprocessed.

Each window created is assigned a unique handle that is used to control and determine which messages are relevant to that window. In code, this is defined as the `HWND`, or window handle.

The main reason behind this system is to allow for multitasking and multithreading. This means that the operating system can run more than one application in one instance, even though a processor can only handle one task at a time.

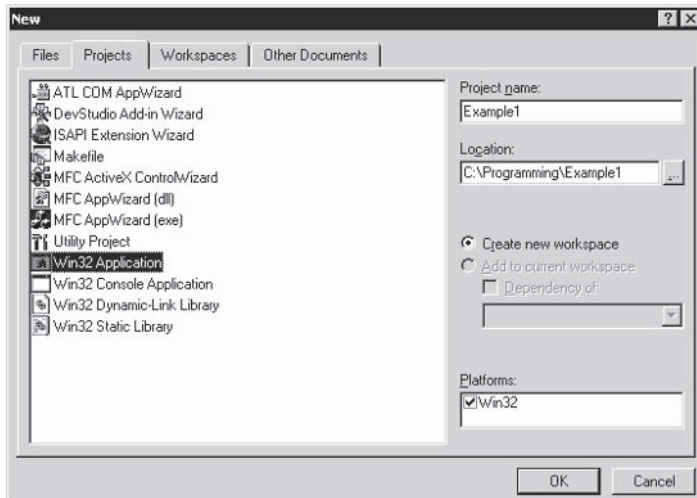
There is a lot more to windows than this, but this should give you a reasonable overview of how the system works.

Creating a Window

Load up Microsoft Visual Studio and select File, New...

The following dialog box is now visible in the center of the screen.

Figure 1-1



Select the Projects tab at the top of the dialog and then choose the Win32 Application option on the main display. Select the location for your project, enter your project's name, and click OK.

Next, select the type of project you wish to create. Leave it on the default option (An empty project) and click the Finish button. A project information box is now visible; simply click OK in this box.

Now we are working with the Visual Studio main interface. Currently the ClassView is active, but we are interested in the FileView, so select this tab.

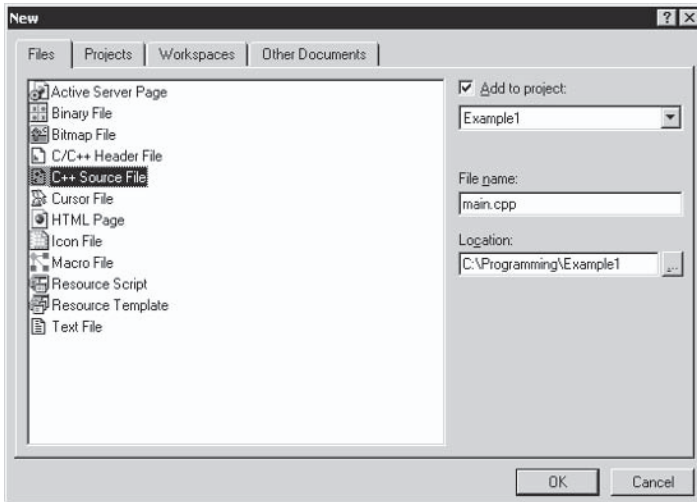
Figure 1-2



The FileView is a list of all the C and C++ source and header files that are active in your project. Currently we do not have any files in our project, so we need to add our main C++ source file.

Select File, New... as you did before, but this time we will be using the Files tab instead of the Projects tab. The following dialog will be visible.

Figure 1-3



Select the C++ Source File option as shown in Figure 1-3 and type in the filename as `main.cpp`. Now click the OK button to add this empty file to your project.

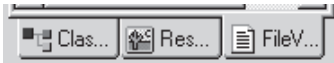
You now have your main source file in your project and it is visible in the Visual Studio editor.

There are two main items required in a standard Windows program: the entry point to your program, which is named `WinMain`, and the Windows callback procedure, commonly named `WndProc`, which is used to keep your Windows application up to date.

For what we require though, it is best to take the dialog approach, making it even simpler to design and code. First, we need to add our dialog, so click File, New... again, but this time you want to add a resource script. Type in the filename as `resource` and click OK.

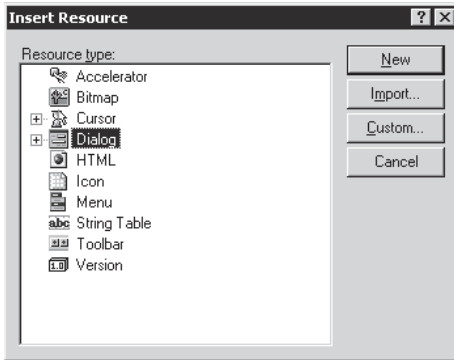
Once this is done, you will notice another tab has appeared between the ClassView and FileView tabs. This tab is called the ResourceView; it allows you to visually create and edit dialogs for use within your program.

Figure 1-4



Once you select the ResourceView tab, you will be presented with the resource editor. Right-click on resource.rc in the main view and then left-click on the Insert option. You will then be presented with the following dialog box.

Figure 1-5



Select the Dialog option and click the New button. Now you will see a sample dialog box in front of you. For now, we will not do much to it except change the name of the title bar and its identifier, which I explain in the following code.

Double-click on the sample dialog box that Visual Studio created. Now a dialog properties box can be seen. All we are interested in here is the ID, which will probably be set to `IDD_DIALOG1`, and the Caption, which should be Dialog. Let's change the ID to `IDD_CLIENT` and the Caption to Window Example.

It's time to go back and do some code now. We have our dialog template that we can call from our code, so let's do it. Here is the code required to make your dialog window appear on the screen. The OK button on the dialog can be pressed but will have no action, whereas the Cancel button will close the dialog.

```
// Simple Windows Code
#include <windows.h>
#include "resource.h"

LRESULT CALLBACK
ClientDlgProc(HWND DialogWindow, UINT Message, WPARAM wParam, LPARAM lParam)
{
    // Process Messages
    switch(Message)
    {
        case WM_INITDIALOG:
            return FALSE;

        case WM_COMMAND:
```

```
        switch(wParam)
        {
        case IDCANCEL:
            EndDialog(DialogWindow, FALSE);
            return FALSE;
        default:
            break;
        }
        break;

    default:
        break;
    }
    return FALSE;
}

int APIENTRY
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine,
        int nCmdShow)
{
    DialogBox((HINSTANCE) hInstance, MAKEINTRESOURCE(IDD_CLIENT), NULL,
              (DLGPROC) ClientDlgProc);
    return 0;
}
```

NOTE If you get an error that tells you it can't find `afxres.h`, you need to install MFC support for Visual Studio, which comes with the Visual Studio package.

If you have never seen Windows code before, the above code may look complex and a little confusing. Welcome to the world of Windows! Well, it's not that bad, honest.

Let's start with the `WinMain` function. This is simply the point at which Windows starts executing your code. Do not worry about the variables that are passed in; they are controlled by Windows and are beyond the scope of this book.

The main issue here is the `DialogBox` function and the `ClientDlgProc` callback procedure that creates our dialog window on the screen. The first parameter is the instance of the application that you simply take from the first parameter of the `WinMain` function. Next is the identifier that we set when we created the template for our dialog. The third parameter is of no interest to us so we set it to `NULL`, but the final one is. This is a pointer to the update function for the dialog. Each dialog you create requires this update function (basically the same idea as a Windows procedure). In this update function is where you set the actions for buttons and other useful tools. So we set this update function to our callback function for the dialog (`ClientDlgProc`).

For example, the identifier for the Cancel button is IDCANCEL. As you can see in the code, there is a case statement for the Cancel button so when it is clicked, it will close the dialog window. Other buttons can be easily added to the template using the toolbox on the template editor. Just remember that each button must contain a unique ID so you can reference it from within your code.

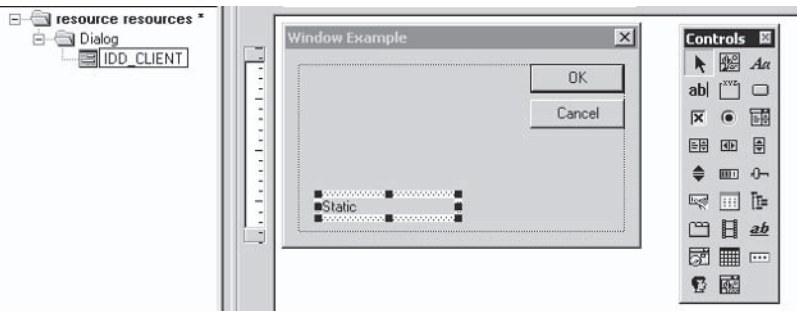
Sending Information to Your Window

In addition to being useful for debugging, being able to update information to a window is essential knowledge and can be used in many situations, such as displaying how many players are connected to the game server.

First you have to add a static text string to the dialog window. To do this you need to go back to the template editor by selecting the ResourceView tab as before. Then you simply double-click on the `IDD_CLIENT` text as seen in Figure 1-6 to bring up your dialog in the main area.

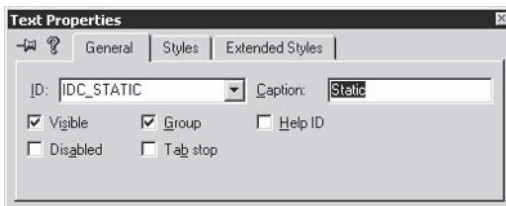
Next, select the *Aa* button from the Controls toolbox and place it somewhere on your dialog as seen in the figure. Now double-click on the text you added to the dialog box to display its properties.

Figure 1-6



The following dialog will now be visible on the screen. All we need to change here is the ID. Change the text `IDC_STATIC` to `IDC_SERVERSTATUS`. This will give it more meaning when it comes to adding it into the code.

Figure 1-7



Now that we have some text, we want to be able to set it to a value from within our code. For example, if we want the text to read “Server Online,” add this line of code after the line that contains “case WM_INITDIALOG”:

```
SendDlgItemMessage(DialogWindow, IDC_SERVERSTATUS,  
WM_SETTEXT, NULL, (long) "Server Online");
```

Then when the dialog box is initialized, Windows will send a message to the dialog box to tell it to update the IDC_SERVERSTATUS text with the string you supplied in the function. In this case, it would update the text from “Static” to “Server Online.”

The first parameter is the handle to the dialog, which is the first variable that is passed into the dialog update procedure. The second is the identifier for what you want to change or update. Next comes the command that you wish to send. There are many commands and the best way to figure out how they work is just to experiment with them. For now we are using WM_SETTEXT, which tells Windows to change a static text string in a dialog box. The fourth variable is not used for the WM_SETTEXT command, so we simply set it to NULL. The final variable is used to declare the string that we want to update the static text with, in this case “Server Online.” Also note that the string must be typecast to a long.

TIP Try experimenting with editable text. It works on the same principles, and you simply send a WM_GETTEXT message to retrieve what the user entered.

Static Link Libraries

Later when we create our online tutorial game, we will be using static link libraries to encapsulate all our network and graphics code, which makes it easier to reuse for future projects. As well as the reusability factor, static link libraries also protect your source code while allowing others to use the functionality of your code.

When creating a static link library, a WinMain function or a Windows update function (WndProc) are not required. This is because we are not actually creating a program but rather just a collection of functions that we can use from within our programs. The library is created using standard C/C++ source and header files, but the output is a library rather than a Windows executable.

When you compile your library, the compiler will generate a library file with the extension .lib. To use the functions in your library, you are required to include it in your project and include the header file that

was used to create the library (which contains all the external variables, types, and function prototypes).

The easiest way to use your library is to create Lib and Include folders for your library and include those directories in the Visual Studio directory settings that are explained in the sections to follow.

Creating a Static Link Library

When creating a static link library, the best thing to do is to create a directory structure on your hard drive for the library to be stored in. From our experience, we recommend the structure shown in Figure 1-8.

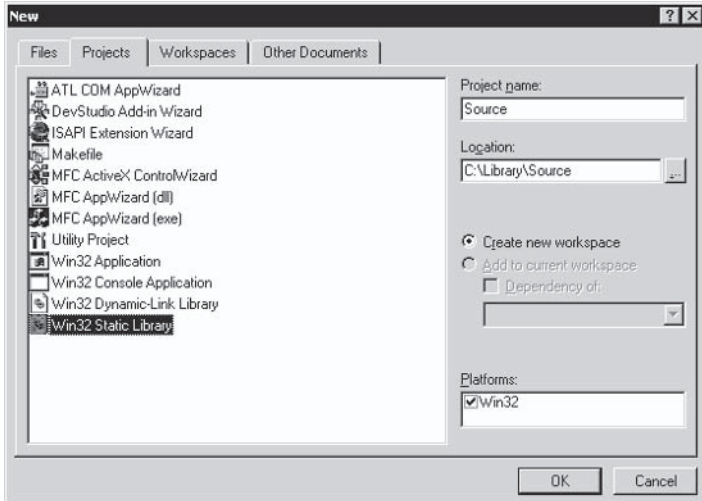
Figure 1-8



- **Examples** — All example programs that display how to use your library are stored in this folder. This is probably one of the most useful things that can accompany your library since the source code is not visible to any other programmer using it.
- **Formats** — This is where you store any file formats specific to your static link library (i.e., your own 3D model format).
- **Include** — This stores the entire collection of C/C++ header files that are needed to use your library. This one of the directories that you must set up in Visual Studio to make your library work.
- **Lib** — This is where you actually store your complete library file. It is a good idea to include both the release and debug versions of your library here so that it is easier to debug programs created with it. This is the other directory required by Visual Studio.
- **Source** — All source code related to your library must be kept in this folder.
- **Tools** — Any programming tools that are used alongside your library are stored here (such as file format converters).

Now that we have our structure, we need to create a static link library project. This is done by selecting File, New... in Visual Studio. The following dialog is now visible.

Figure 1-9



Select the Win32 Static Library option and enter a name for your library. Next, select the location and press the OK button.

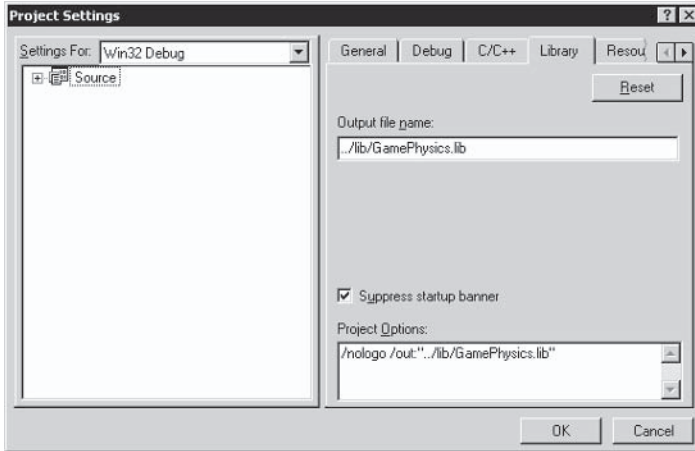
In the dialog that appears, leave both the Pre-Compiled header and MFC support options unchecked as we will not be using either in this book, and then click Finish. A project information box is now visible; simply click OK.

Next, we need to add our source and header files to the project as you did when creating the window. Remember, this time we do not require the WinMain or update procedure, just functions that we wish to reuse. Let's call the source file library.cpp and the header file library.h for this example.

If you now press F7, it will build your library and put it in the Release or Debug folder depending on your project configuration. The library will have the same name as your project. For example, if your project is named "GamePhysics" your library will be "gamephysics.lib."

It is a good idea to make Visual Studio automatically copy your header file and library to the correct directories in your structure to assure that you are always using the latest version. Selecting Project, Settings from the main menu makes the following dialog visible.

Figure 1-10



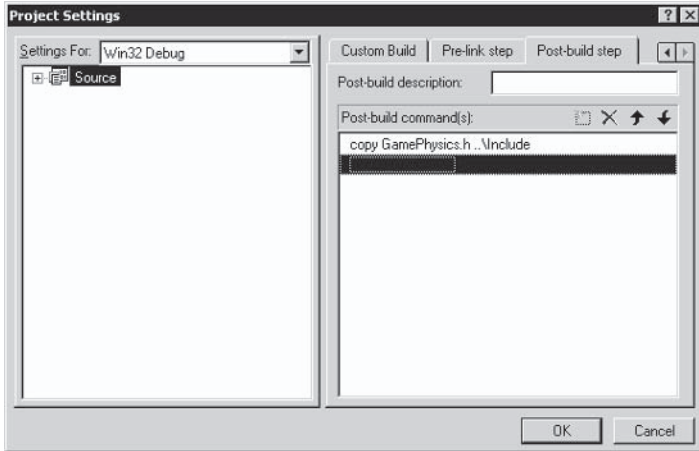
Set the Output file name for your library and choose where you want it to go relative to the directory your source files are in. This must be done for both release and debug configurations. In the previous dialog the Debug settings are currently active.

TIP It is a good idea to have both debug and release versions of your library. Call your release version the correct name (i.e., "GamePhysics.lib") and add a suffix to the debug version (i.e., "GamePhysicsDebug.lib") to distinguish them easily.

Now the library file is created in the correct directory in our structure when the project is built. We also want to copy our header file to the Include directory of our structure.

As can be seen in Figure 1-11, you simply add a post-build command that tells the compiler to copy the header file(s) to the Include directory.

Figure 1-11



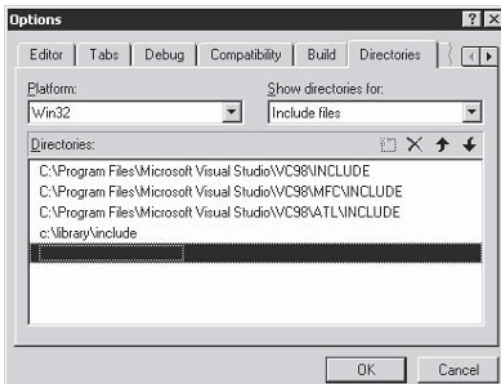
We have now covered the basics on how to create a static link library. The most important thing to remember is to prototype all functions and extern all global variables you wish to be accessible outside of your library. In the next section, we discuss how to set up Visual Studio to find your library.

Using a Static Link Library

Once you have built your library and it is in the correct directories, you need to tell Visual Studio where to look for it. Select Tools, Options... from the main Visual Studio menu to open the Options dialog box.

If the Directories tab is unselected, select it now to display a list of directories as shown in Figure 1-12. Only the top three should be visible unless you have previously added other static libraries.

Figure 1-12



You must now add the Include directory of your library for the Include files and the Lib directory for the Library files. You select which one you wish to add by changing the top-right drop-down box.

Now Visual Studio is able to find and recognize your static link library. To use it in a program you have written, you must first load your project into Visual Studio, then select Project, Settings...

Next, select the Link tab and add your library name before "kernel32.lib." Also, remember to include the header file for your library in your main code.

Summary

This chapter discussed the basics to creating dialog-based applications in Visual Studio. The best thing to do is experiment by creating dialogs, adding buttons, and making the buttons set strings to different values when you press them. Also, learn to use editable text as this is highly useful and allows the user to give feedback to your program.

If you are interested in learning more, there are entire books written on Windows programming and there is also an excellent resource available on the Internet (<http://msdn.microsoft.com>), but the quick introduction provided here will give you enough knowledge to understand the concepts used in this book.

TIP *Windows 98 Programming from the Ground Up* by Herbert Schildt (ISBN: 0-07-882306-4) provides an excellent way of understanding Windows programming.

Chapter 2

Using Databases

Introduction

This chapter covers how to create a stable and fast database system for your game server.

Although it is possible to store player information on the player's local machine, it makes much more sense to store the data on the server for many reasons. The most important reason is to prevent players from running “hacks” on their computers to change their character data in the game or even back the data up to another file. If the data is stored on the server, it makes it impossible for the player to run a local game hack that will modify the character data. Another reason for this is to allow players to play from different locations and machines without having to copy their character data to another computer.

In this chapter, we cover how to install and use MySQL and then look at how to access a database from within a C++ application (aka a game).

What Is MySQL?

MySQL is an open source relational database management system. Its purpose is to store and allow easy and fast access to data.

Installing MySQL

Installing MySQL is relatively painless. Firstly, you need the installation program for MySQL, which is available on the CD that accompanies this book. Alternatively, you can download the latest version from <http://www.mysql.com>.

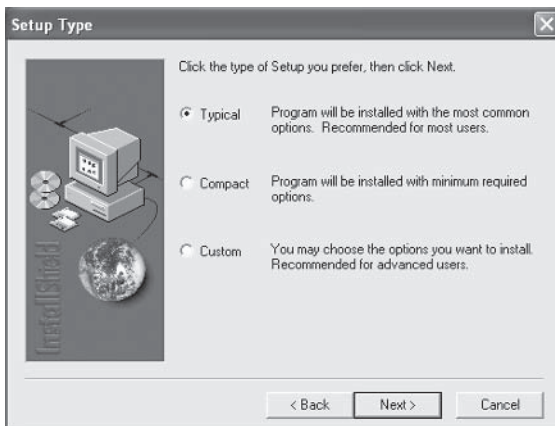
NOTE If you choose to install MySQL in a folder other than the default (C:\MYSQL) or you wish to start MySQL on NT/Win2000 as a service, you need to create a file named MY.CNF in the root of your C:\ drive with the following information in it (or append the following information to either \Windows\my.ini or \winnt\my.ini, depending on whether you are using Windows 98 or 2000/XP, respectively):

```
[mysqld]
basedir=E:/installation-path/
datadir=E:/data-path/
```

After you have installed MySQL, the installation directory will contain a file named my-example.cnf. You can use this as a template to create your own my.cnf file.

When you start the installation, you will be asked which type of setup you would prefer: Typical, Compact, or Custom. Here we select the Typical option, as it will install all the components we require in order to work with MySQL.

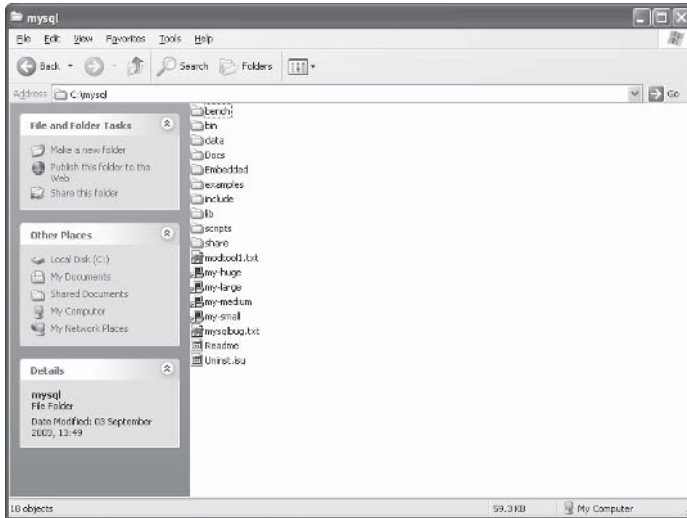
Figure 2-1
Selecting
the Typical
install
option



After selecting Typical, simply press the Next button to automatically complete the installation process.

Now that we have MySQL installed, first we will browse the directory so we can see what it has installed for us. If we open up the directory to which we installed MySQL to (typically `c:\mysql`), we can see the following directory structure.

Figure 2-2
The MySQL
directory
structure

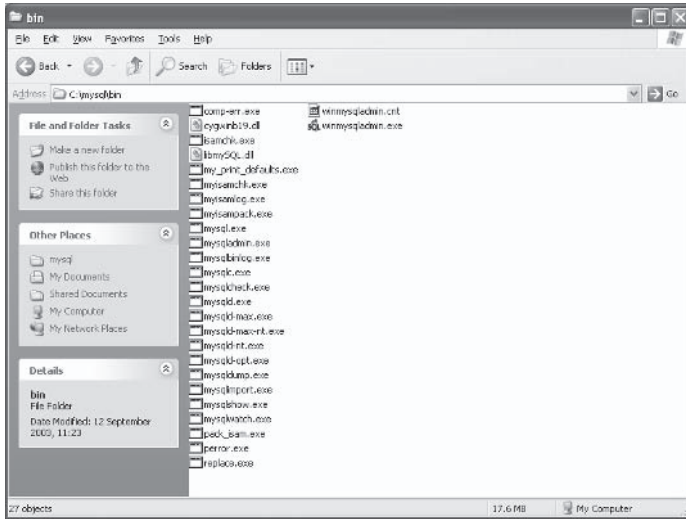


All we really have use for here is the `bin` directory, which contains the MySQL server and client executables, and the `docs` directory, which contains the HTML version of the MySQL manual.

NOTE An Adobe PDF version of the MySQL manual is supplied on the companion CD. We find this easier to read, but you will also require the free program Adobe Acrobat Reader to view this manual, which is also on the CD and available to download at <http://www.adobe.com/products/acrobat/readermain.html>.

Let's now take a look in the `bin` directory and see what is of use to us there. The contents can be seen in the following image.

Figure 2-3
The contents
of the bin
directory



As you can see, there are many executables in this directory; some are daemons (i.e., the MySQL server) and some are console-based clients for accessing the MySQL server.

NOTE A daemon is simply a program or process that sits idly in the background until it is invoked to perform its task.

The executable `mysql.exe` is a console-based client, which is used to interact with the MySQL server using the SQL language. We cover SQL in great depth later in this chapter.

If we try to run the console client (`mysql.exe`) now, the following screen will appear for a couple of seconds and then disappear.

Figure 2-4
A blank
window?!



Why? The reason the window appears and promptly disappears is because there is currently no MySQL server to connect to — i.e., there is no MySQL daemon running for the client to interact with.

So the obvious thing to do now is to run a MySQL server so we can access it via the client. The best way to do this is to run the `winmysqladmin.exe` file located in the `bin` directory. The first time this is run it will ask for a username and password. This isn't really important for testing, so just enter something like `admin` for the username and the password. After that, you will see a small traffic light icon appear in your system tray with the green light indicating the MySQL server is running. Note that MySQL will now automatically run every time Windows is booted up.

SQL Statements

Now that we have the MySQL server running, it is time to load up the MySQL console client. This is done the same way as we loaded it before (i.e., run the `mysql.exe` executable that is located in the `c:\mysql\bin\` directory).

In SQL (structured query language), there are two types of statements we can execute. These are DDL (data definition language) and DML (data manipulation language). DDL is used to affect the structure of the database, such as adding databases, tables, etc., whereas DML is used to add and modify data in an existing database and retrieve information. The following sections cover these statements.

Data Definition Language

First we take a look at DDL, as we require a database to work with before we can manipulate any data within it.

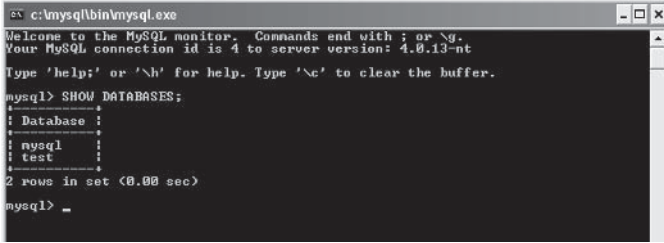
Creating and Dropping Databases

First, let's see what databases already exist in the MySQL server. We can view this information by using the `SHOW` command. Enter the following (except the "`mysql>`" part) into the MySQL console client, followed by a Return:

```
mysql> SHOW DATABASES;
```

When you press Return, the following should be visible in the console display:

Figure 2-5
Dialog box



```

c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 4 to server version: 4.0.13-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> SHOW DATABASES;
+-----+
| Database |
+-----+
| mysql   |
| test    |
+-----+
2 rows in set (0.00 sec)

mysql>
  
```

As you can see, there are two databases already created in the MySQL server. The `mysql` database contains administration information for the MySQL server and should not be modified. The other database, `test`, is exactly what it is called — a test for the MySQL server. Again it is not a good idea to remove it.

Creating a Database

So how do we add our own database to the MySQL server? By using the `CREATE` command. To create a database called `mydata`, we would use the following syntax.

```
mysql> CREATE DATABASE mydata;
```

Notice that the semicolon is added after every command in the SQL language. When we press Return after entering this command, the console informs us that the query was okay. This can be seen in Figure 2-6.

NOTE The following length and character restrictions are imposed on the names of databases, tables, columns, and aliases.

Table 2-1 Naming restrictions

Identifier	Max Length	Valid Characters
Database	64	All valid directory name characters except "." and "/".
Table	64	All valid directory name characters except "." and "/".
Column	64	All are valid
Alias	255	All are valid

Figure 2-6
Creating a
database

```

c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 5 to server version: 4.0.13-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> CREATE DATABASE mydata;
Query OK, 1 row affected (0.00 sec)

mysql> _

```

Now that we have created the database, we can ensure it is on the server by again using the SHOW command as follows:

```
mysql> SHOW DATABASES;
```

When we press Return after entering this command, we can see that our database has been added to the list (note that the list is in alphabetical order, not the order in which they were created).

Figure 2-7
The mydata
database
has been
added to
the list.

```

c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 6 to server version: 4.0.13-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> SHOW DATABASES;
+-----+
| Database |
+-----+
| mydata   |
| mysql    |
| test     |
+-----+
3 rows in set (0.00 sec)

mysql> _

```

Dropping a Database

Now we will remove a database from the server. Note that when we do this all data (if any) will be lost. To remove a database, we “drop” it from the server by using the DROP command. So to drop our new mydata database we would use the following command:

```
mysql> DROP DATABASE mydata;
```

When we execute this command by pressing Return, the query will be reported as OK, as shown in the following screenshot.

Figure 2-8
Dropping a
database

```

c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 7 to server version: 4.0.13-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> DROP DATABASE mydata;
Query OK, 0 rows affected (0.03 sec)

mysql> _

```

Let’s again list the databases using the SHOW command:

```
mysql> SHOW DATABASES;
```

Notice that our mydata database is no longer visible in the list.

Figure 2-9
Database
listing after
the DROP
command

```

c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 8 to server version: 4.0.13-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> SHOW DATABASES;
+-----+
| Database |
+-----+
| mysql   |
| test    |
+-----+
2 rows in set (0.00 sec)

mysql> _

```

Column (Field) Types in MySQL

Before we discuss the creation of tables within databases, now is a good time to discuss the different column types we can have in tables.

Each column in a table must be assigned a type that represents the kind of information that field is going to hold. Table 2-2 lists the available types you can use.

Table 2-2 MySQL column types

Type	Description
TINYINT	A very small integer. Signed range is -128 to 127. Unsigned range is 0 to 255.
SMALLINT	A small integer. Signed range is -32768 to 32767. Unsigned range is 0 to 65535.
MEDIUMINT	A medium-sized integer. Signed range is -8388608 to 8388607. Unsigned range is 0 to 16777215.
INT	A normal-sized integer. Signed range is -2147483648 to 2147483647. Unsigned range is 0 to 4294967295.
BIGINT	A large integer. Signed range is -9223372036854775808 to 9223372036854775807. Unsigned range is 0 to 18446744073709551615.
FLOAT	A small, single-precision floating-point number that cannot be unsigned. Signed range is -3.402823466E+38 to -1.175494351E-38, 0, and 1.175494351E-38 to 3.402823466E+38.
DOUBLE	A double-precision floating-point number that cannot be unsigned. Signed range is -1.7976931348623157E+308 to -2.2250738585072014E-308, 0, and 2.2250738585072014E-308 to 1.7976931348623157E+308.

Type	Description
DECIMAL	An unpacked floating-point number that cannot be unsigned. Works like a char column in that the number is stored as a string (i.e., each number uses one character in the string).
DATE	A date. Range is 1000-01-01 to 9999-12-31 and is in the format YYYY-MM-DD.
TIME	A time. Range is -838:59:59 to 838:59:59 and is in the format HH:MM:SS.
DATETIME	A combination of date and time. Range is 1000-01-01 00:00:00 to 9999-12-31 23:59:59 and is in the format YYYY-MM-DD HH:MM:SS.
YEAR[(2 4)]	A year in two- or four-digit format (default is four-digit). Range is 1901 to 2155 and also 0000.
TIMESTAMP	A timestamp. Range is 1970-01-01 00:00:00 to sometime in the year 2037 in the format of YYYYMMDDHHMMSS.
CHAR(length)	A fixed-length string that is always right-padded with spaces to the specified length when stored. The range is 1 to 255 characters depending on the length specified.
VARCHAR	A variable-length string.
TINYBLOB/ TINYTEXT	A tiny binary object with a maximum length of 255 characters. *See Note below table.
BLOB/TEXT	A binary object with a maximum length of 65,535 characters. *See Note below table.
MEDIUMBLOB/ MEDIUMTEXT	A medium binary object with a maximum length of 16,777,215 characters. *See Note below table.
LOBLOB/ LONGTEXT	A large binary object with a maximum length of 429,496,295 characters. *See Note below table.
ENUM('val1','val2'...)	An enumeration. A list of string values of which only one can be selected. Maximum of 65,535 distinct values.
SET('val1','val2'...)	A set. A string object that can have zero or more values, each of which must be chosen from the list (i.e., 'val1', 'val2', etc.). Maximum of 64 members.

NOTE The only difference between the BLOB and TEXT types is that for sorting and comparisons, a BLOB is case sensitive whereas the TEXT type is not.

Adding, Modifying, and Dropping Tables

The following sections discuss how to manipulate tables in databases.

Creating Tables

Now that we know the possible types for the columns in our tables, let's look at how we actually go about creating a table.

Let's say we wish to create a table to hold some user details within a database. In fact, we will be using a similar table later in the tutorial section. We want to store the user's title, first name, last name, age, email address, and the date they were added to the database. This will require the following columns:

- Title
- Firstname
- Surname
- Age
- Email
- DateAdded

Before we get into how to actually add the information, let's first think about how we are going to store it. Or, more to the point, what types we require for each of the columns.

For the title, firstname, and surname, we can use the `TEXT` type as it contains plenty of characters to allow for all possibilities. For age, an unsigned `TINYINT` would be an obvious choice as these are numerical and no one has ever been known to live past 255. For email address, we can again use a `TEXT` type, as it will give us substantial storage space for the address. And finally, for the date that the user was added to the table, we can use a `TIMESTAMP`. This is very useful in that the time and date can be retrieved automatically into the database; this is discussed later in the chapter.

So now that we know what types we want for our columns, we first need to create a database to add the table into. This goes back to what we learned in the previous section. Let's create a database called `myinfo` with the following command:

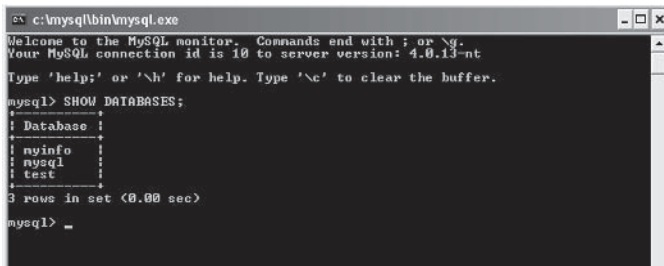
```
mysql> CREATE DATABASE myinfo;
```

When we execute this command, the console should report that the query was okay. We can now check that our database has been created by using the following command:

```
mysql> SHOW DATABASES;
```

When we execute this command, the following should be visible in the console:

Figure 2-10
The myinfo database is now visible in the console after using the SHOW DATABASES command.



```
c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands and with ; or \g.
Your MySQL connection id is 10 to server version: 4.0.13-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> SHOW DATABASES;
+-----+
| Database |
+-----+
| myinfo   |
| mysql   |
| test    |
+-----+
3 rows in set (0.00 sec)

mysql> -
```

Now we need to tell MySQL that we wish to perform actions on the myinfo database. This is accomplished by using the USE command:

```
mysql> USE myinfo;
```

After executing this command, any DDL (data definition language) and DML (data manipulation language) statements that are executed will affect the database in use, which in this case is our myinfo database.

With the database set up and ready to accept commands, we can create our table (which we will call userinfo) with the following statement:

```
mysql> CREATE TABLE userinfo (
  -> id INT auto_increment,
  -> title TEXT,
  -> firstname TEXT,
  -> surname TEXT,
  -> age TINYINT,
  -> email TEXT,
  -> dateadded TIMESTAMP,
  -> PRIMARY KEY(id));
```

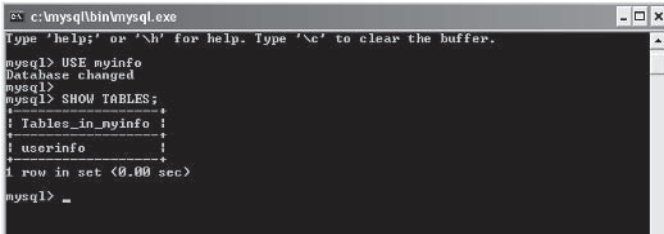
Let's break this up a little so we can see what is going on. First, we declare that we wish to create a table by entering CREATE TABLE. Next we specify the name we wish to call the table, in this case userinfo. Then within parentheses we list all of the column names and types we require. Note how we have added an extra field named id. This makes it easier to handle data in a relational way, as we discuss later in this chapter. Finally, note the addition of the primary key as the last parameter. This is used to determine how the table is optimized within the database. Again, we discuss the use of keys later in this chapter.

We can now see if our table was created successfully by executing the following command:

```
mysql> SHOW TABLES;
```

When this is executed, the following output should be visible in the console:

Figure 2-11
Here the
userinfo table
can be seen as
part of our
database.



```

c:\mysql\bin\mysql.exe
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE myinfo
Database changed
mysql> SHOW TABLES;
+-----+
| Tables_in_myinfo |
+-----+
| userinfo          |
+-----+
1 row in set (0.00 sec)

mysql> _

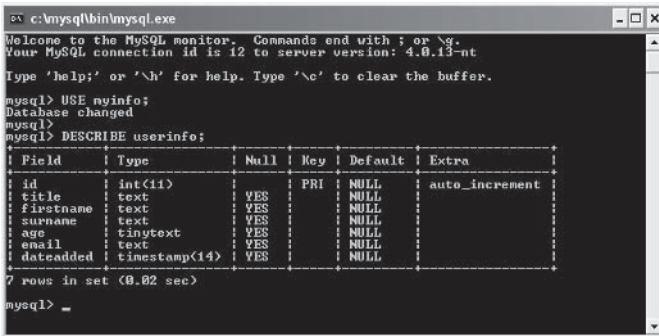
```

Note that you can also view the columns in a table by using the following command:

```
mysql> DESCRIBE userinfo;
```

When you execute this command, the console will display all the details for each of the columns in the userinfo table, as shown in Figure 2-12.

Figure 2-12
Describing
the userinfo
table



```

c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 12 to server version: 4.0.13-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE myinfo;
Database changed
mysql> DESCRIBE userinfo;
+-----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id    | int(11) | YES | PRI | NULL | auto_increment |
| title | text   | YES |     | NULL |                |
| first | text   | YES |     | NULL |                |
| name  | text   | YES |     | NULL |                |
| sur   | text   | YES |     | NULL |                |
| name  | text   | YES |     | NULL |                |
| age   | tinytext | YES |     | NULL |                |
| email | text   | YES |     | NULL |                |
| date  | timestamp(14) | YES |     | NULL |                |
| added |         |         |         |         |                |
+-----+-----+-----+-----+-----+-----+
7 rows in set (0.02 sec)

mysql> _

```

This information can be useful to ensure the table was created as you envisioned and to recap the columns a table contains at a later date.

Modifying Tables

Now let's see how to modify a table. Modifying a table can range from simply changing the type of one of the columns to adding a complete new column or removing an existing column.

Let's first look at how we can change the name of an existing column. In our userinfo table we have a column called firstname, but let's now change this to read forename instead.

To make this change we need to use the following syntax:

```
mysql> ALTER TABLE userinfo CHANGE firstname forename TEXT;
```

Note how we also must supply the data type for the column as well as its old and new names. Here is how this should look in the MySQL console client:

Figure 2-13
Modifying a
column name

```
c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 13 to server version: 4.0.13-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE myinfo;
Database changed
mysql>
mysql> ALTER TABLE userinfo CHANGE firstname forename TEXT;
Query OK, 0 rows affected (0.03 sec)
Records: 0 Duplicates: 0 Warnings: 0

mysql> _
```

If we now describe the userinfo table with the following command:

```
mysql> DESCRIBE userinfo;
```

we can see that the firstname column has been renamed to forename:

Figure 2-14
Description of
the updated
userinfo table

```
c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 14 to server version: 4.0.13-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE myinfo;
Database changed
mysql>
mysql> DESCRIBE userinfo;
+----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+----+-----+-----+-----+-----+-----+
| id | int(11) | YES | PRI | NULL | auto_increment |
| title | text | YES | | NULL | |
| forename | text | YES | | NULL | |
| surname | text | YES | | NULL | |
| age | tinytext | YES | | NULL | |
| email | text | YES | | NULL | |
| dateadded | timestamp(14) | YES | | NULL | |
+----+-----+-----+-----+-----+-----+
7 rows in set (0.00 sec)

mysql> _
```

We can also change the data types of columns in tables. To change the age column from a TINYINT to an INT, we would use the following command:

```
mysql> ALTER TABLE userinfo MODIFY age INT;
```

After executing this command and then using the DESCRIBE command on this table, we can see the type has changed to INT.

Figure 2-15
Now the age
column is of
type INT
rather than
TINYINT.

```
c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 15 to server version: 4.0.13-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE myinfo;
Database changed
mysql>
mysql> DESCRIBE userinfo;
+----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+----+-----+-----+-----+-----+-----+
| id | int(11) | YES | PRI | NULL | auto_increment |
| title | text | YES | | NULL | |
| forename | text | YES | | NULL | |
| surname | text | YES | | NULL | |
| age | int(11) | YES | | NULL | |
| email | text | YES | | NULL | |
| dateadded | timestamp(14) | YES | | NULL | |
+----+-----+-----+-----+-----+-----+
7 rows in set (0.00 sec)

mysql> _
```

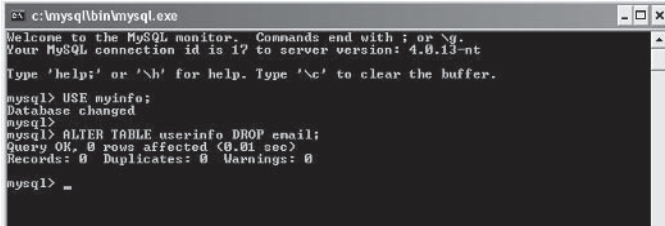
Finally, it is good to know how to remove fields from a table when they are no longer required. Let's say we no longer require the email field in our userinfo table. What we want to do is drop the field from our table,

just as we did earlier in the chapter when we dropped the database. Here is the syntax for removing the email field:

```
mysql> ALTER TABLE userinfo DROP email;
```

Here is how this looks in the MySQL console:

Figure 2-16
Dropping a
field from a
table



```
c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 17 to server version: 4.0.13-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE myinfo;
Database changed
mysql>
mysql> ALTER TABLE userinfo DROP email;
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

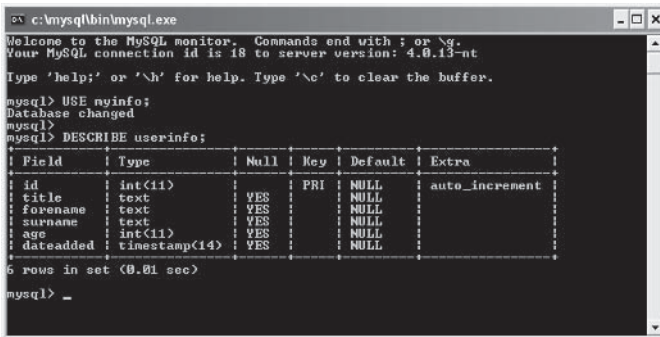
mysql> _
```

Once this command is executed, if we describe the table with the following command:

```
mysql> DESCRIBE userinfo;
```

we can see that the email field has been removed from our userinfo table.

Figure 2-17
As you can
see, the email
field has now
been
removed.



```
c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 18 to server version: 4.0.13-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE myinfo;
Database changed
mysql>
mysql> DESCRIBE userinfo;
+----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+----+-----+-----+-----+-----+-----+
| id | int(11) | YES | PRI | NULL | auto_increment |
| title | text | YES | | NULL | |
| forename | text | YES | | NULL | |
| surname | text | YES | | NULL | |
| age | int(11) | YES | | NULL | |
| dateadded | timestamp(14) | YES | | NULL | |
+----+-----+-----+-----+-----+-----+
6 rows in set (0.01 sec)

mysql> _
```

Dropping (Removing) Tables

Removing tables from a database is very simple, but without careful use can have disastrous effects. The main thing to note when you drop a table is that you also lose all the data contained within the table. Therefore, it is always wise to back up a database before executing any drop commands. We will look at backing up a database later in this chapter.

Let's now look at how we drop the userinfo table from our myinfo database. To do this, we need to execute the following command in the MySQL console client:

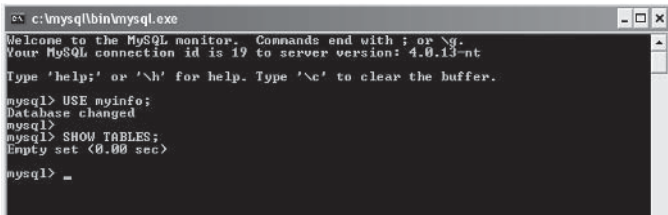
```
mysql> DROP TABLE userinfo;
```

Once this command is executed, we can verify that the table has been removed by using the `SHOW` command to see what tables are currently in our `myinfo` database:

```
mysql> SHOW TABLES;
```

As you can see from Figure 2-18, the table no longer exists in the database.

Figure 2-18
After dropping the `userinfo` table, we have an empty database.



```
c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 19 to server version: 4.0.13-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE myinfo;
Database changed
mysql> SHOW TABLES;
Empty set (0.00 sec)

mysql> _
```

Data Manipulation Language (DML)

In this section, we look at how to add, modify, and remove data from tables in the database using DML (data manipulation language). Without this knowledge, we would not really have any use for a database; it would simply be a static entity with no purpose.

Let's first create a database and table to work with in this section, using the DDL we learned in the previous section. Our database will be called `dmlexample`, so let's create that with the following statement:

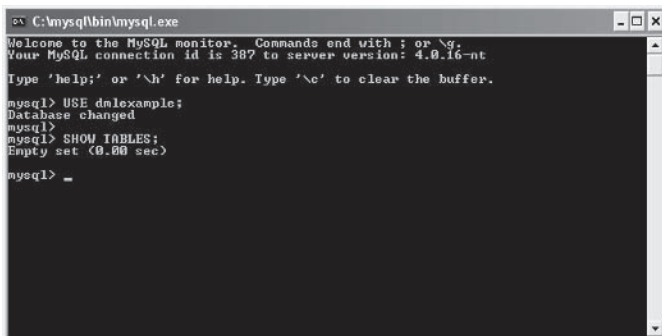
```
mysql> CREATE DATABASE dmlexample;
```

We then need to specify that we wish to use the new database by executing the following statement:

```
mysql> USE dmlexample;
```

The console should now inform us that the database has changed, as seen in Figure 2-19.

Figure 2-19
Creating the `dmlexample` database



```
C:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 307 to server version: 4.0.16-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE dmlexample;
Database changed
mysql> SHOW TABLES;
Empty set (0.00 sec)

mysql> _
```

Now we need to create a table called `sampletable`. This table will contain the following fields:

- Username
- Password
- Age
- Email
- DateCreated

This is accomplished using the knowledge we learned in the section on DDL statements. First we need to specify that we wish to use the `dmlexample` database with the following command:

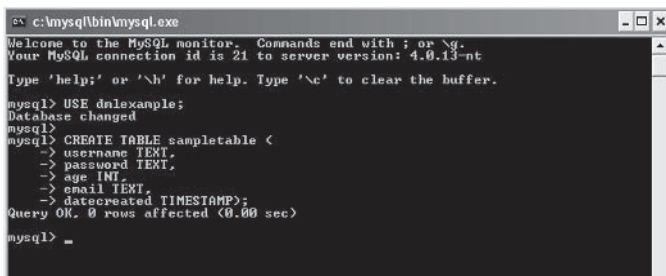
```
mysql> USE dmlexample;
```

Once this is done, we can create our table with the following DDL statement:

```
mysql> CREATE TABLE sampletable (  
-> username TEXT,  
-> password TEXT,  
-> age INT,  
-> email TEXT,  
-> datecreated TIMESTAMP);
```

Figure 2-20 shows how this should look in the MySQL console client:

Figure 2-20
Creating our
sample table



```
c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 21 to server version: 4.0.13-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE dmlexample;
Database changed
mysql>
mysql> CREATE TABLE sampletable (<
-> username TEXT,
-> password TEXT,
-> age INT,
-> email TEXT,
-> datecreated TIMESTAMP);
Query OK, 0 rows affected (0.00 sec)

mysql> _
```

Inserting Data

To add rows to the table, we use the `INSERT` command. Here is how we would add a single row to our `sampletable`:

```
mysql> INSERT INTO sampletable VALUES ('andrew', 'qwerty', 20,  
                                         'andrew@huntedcow.com', NULL);
```

Figure 2-21 shows how this looks when we enter it into the MySQL console. Notice how the feedback from the console tells us that one row has been affected. This means we have added one row to our `sampletable` table.

Figure 2-21
Inserting a
single row of
data

```

c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 25 to server version: 4.0.13-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE dmlexample;
Database changed
mysql>
mysql> INSERT INTO sampletable VALUES ('andrew', 'querty', 20, 'andrew@huntedcov
.com', NULL);
Query OK, 1 row affected (0.00 sec)

mysql> _

```

We can then use the `SELECT` command to view the data in the table. We go into more detail about this command later in this chapter as it is very important, but for now we just use it without further explanation. Let's use `SELECT` to see what data is in our `dmlexample` table:

```
mysql> SELECT * FROM sampletable;
```

When we execute this statement, the following will be visible in the MySQL console:

Figure 2-22
Viewing the
new row in
the table

```

c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 26 to server version: 4.0.13-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE dmlexample;
Database changed
mysql>
mysql> SELECT * FROM sampletable;
+-----+-----+-----+-----+-----+
| username | password | age | email | datecreated |
+-----+-----+-----+-----+-----+
| andrew | querty | 20 | andrew@huntedcov.com | 20031008005157 |
+-----+-----+-----+-----+-----+
1 row in set (0.02 sec)

mysql> _

```

NOTE Notice how the `datecreated` field reflects the time and date when we added the row to the table. This is because we specified `NULL` when we added the row; doing this will make a `TIMESTAMP` field grab the current date and time from the system by default.

It is also possible to add several rows of data in a single command. Let's try this now by adding another three rows to our table in a single `INSERT` command, as shown below:

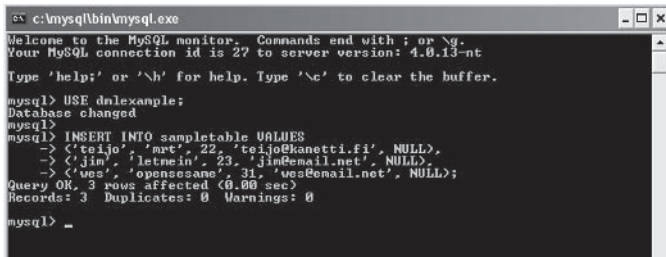
```

mysql> INSERT INTO sampletable VALUES
-> ('teijo', 'mrt', 21, 'teijo@kanetti.fi', NULL),
-> ('jim', 'letmein', 23, 'jim@email.net', NULL),
-> ('wes', 'opensesame', 31, 'wes@email.net', NULL);

```


When we execute this command, the following can be seen in the MySQL console client:

Figure 2-23
Inserting
multiple rows
in a single
statement



```

c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 27 to server version: 4.0.13-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE dmlexample;
Database changed
mysql>
mysql> INSERT INTO sampletable VALUES
-> <'teijo', 'mrt', 22, 'teijo@kanetti.fi', NULL>,
-> <'jim', 'letmein', 23, 'jim@email.net', NULL>,
-> <'wes', 'opensesame', 31, 'wes@email.net', NULL>;
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> _

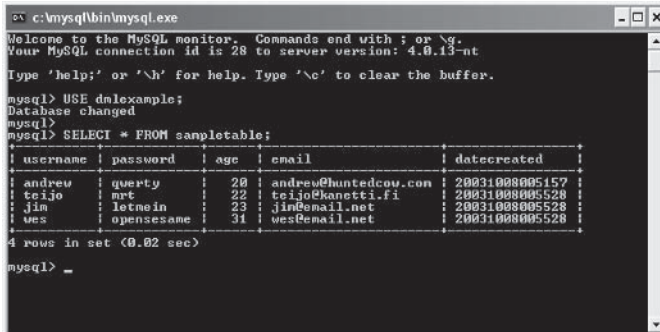
```

As you can see, this time the feedback from the console indicates that three rows have been affected; thus, we have added three rows to our table. We can verify this by again using the SELECT command:

```
mysql> SELECT * FROM sampletable;
```

When this is executed, you will see that the table contains four rows (or records, if you like) of information. Figure 2-24 shows the MySQL console after the SELECT statement has been executed.

Figure 2-24
Now we have
four rows in
the table.



```

c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 28 to server version: 4.0.13-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE dmlexample;
Database changed
mysql>
mysql> SELECT * FROM sampletable;
+-----+-----+-----+-----+
| username | password | age | email | datecreated |
+-----+-----+-----+-----+
| andrew | querty | 20 | andrew@huntedcow.com | 20031008005157 |
| teijo | mrt | 22 | teijo@kanetti.fi | 20031008005528 |
| jim | letmein | 23 | jim@email.net | 20031008005528 |
| wes | opensesame | 31 | wes@email.net | 20031008005528 |
+-----+-----+-----+-----+
4 rows in set (0.02 sec)

mysql> _

```

Modifying Data

To modify existing data in a table we use the UPDATE command. First let's try to change all the passwords in all the rows in the table to changeme. This can be accomplished with the following statement:

```
mysql> UPDATE sampletable SET password = 'changeme';
```

When we execute this statement, the console will inform us that four rows have been affected as we have changed the password for *every* row in that table. We can see the effect on the table by using the SELECT command as follows:

```
mysql> SELECT * FROM sampletable;
```

Here is a screenshot of this command being executed in the console:

Figure 2-25
The password field has been updated in all of the rows.

```

c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 30 to server version: 4.0.13-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE dmlexample;
Database changed
mysql>
mysql> SELECT * FROM sampletable;
+-----+-----+-----+-----+-----+
| username | password | age | email | datecreated |
+-----+-----+-----+-----+-----+
| andrew | changeme | 20 | andrew@huntedcow.com | 20031008005706 |
| teijo | changeme | 22 | teijo@kanetti.fi | 20031008005706 |
| jim | changeme | 23 | jim@email.net | 20031008005706 |
| wes | changeme | 31 | wes@email.net | 20031008005706 |
+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)

mysql> _

```

But what if you only want to update a single row? Let's say we wish to change the password for Teijo from changeme back to mrt. We would use the following statement to do this:

```
mysql> UPDATE sampletable SET password = 'mrt' WHERE username = 'teijo';
```

When we execute this command in the console, it informs us that one row has been affected. This is because it will only update the password field if the username field is equal to teijo. If we use the SELECT command on the table now, we can see that only Teijo's password has changed, as shown in Figure 2-26.

Figure 2-26
Updating only a single row

```

c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 31 to server version: 4.0.13-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE dmlexample;
Database changed
mysql>
mysql> UPDATE sampletable SET password = 'mrt' WHERE username = 'teijo';
Query OK, 1 row affected (0.02 sec)
Rows matched: 1 Changed: 1 Warnings: 0

mysql> SELECT * FROM sampletable;
+-----+-----+-----+-----+-----+
| username | password | age | email | datecreated |
+-----+-----+-----+-----+-----+
| andrew | changeme | 20 | andrew@huntedcow.com | 20031008005706 |
| teijo | mrt | 22 | teijo@kanetti.fi | 20031008005854 |
| jim | changeme | 23 | jim@email.net | 20031008005706 |
| wes | changeme | 31 | wes@email.net | 20031008005706 |
+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)

mysql> _

```

We can also use this technique to update only certain fields. For example, we could change all the passwords of the people who are 30 years old or younger. Here is the command to do this:

```
mysql> UPDATE sampletable SET password = 'young' WHERE age <= 30;
```

When we execute this command, it will inform us that three rows have been affected, as three of the four records in our table have an age field that is 30 or less. If we then use the SELECT command we can see the following output in the console:

Figure 2-27
Conditional
updates

```
c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 34 to server version: 4.0.13-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE dmlexample;
Database changed
mysql>
mysql> SELECI * FROM sampletable;
+-----+-----+-----+-----+-----+
| username | password | age | email | datecreated |
+-----+-----+-----+-----+-----+
| andrew | young | 20 | andrew@huntedcow.com | 20031008010014 |
| teijo | young | 22 | teijo@kanetti.fi | 20031008010014 |
| jim | young | 23 | jim@email.net | 20031008010014 |
| wes | changene | 31 | wes@email.net | 20031008005706 |
+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)

mysql> _
```

TIP A useful idea is to update a `TIMESTAMP` field with `NULL`. This will retrieve the latest time from the system the database is running on. A practical use would be to note the last time a player logged in.

Removing (Deleting) Data

Removing data from a table is done very similarly to updating data. Let's look at how to delete a single row of data from the database using the following statement:

```
mysql> DELETE FROM sampletable WHERE username = 'teijo';
```

When we execute this command, the MySQL console client will inform us that one row was affected. If we now use the `SELECT` command on the table, the following can be seen in the console:

Figure 2-28
Deleting a
single row

```
c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 35 to server version: 4.0.13-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE dmlexample;
Database changed
mysql>
mysql> DELETE FROM sampletable WHERE username = 'teijo';
Query OK, 1 row affected (0.00 sec)

mysql> SELECI * FROM sampletable;
+-----+-----+-----+-----+-----+
| username | password | age | email | datecreated |
+-----+-----+-----+-----+-----+
| andrew | young | 20 | andrew@huntedcow.com | 20031008010014 |
| jim | young | 23 | jim@email.net | 20031008010014 |
| wes | changene | 31 | wes@email.net | 20031008005706 |
+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)

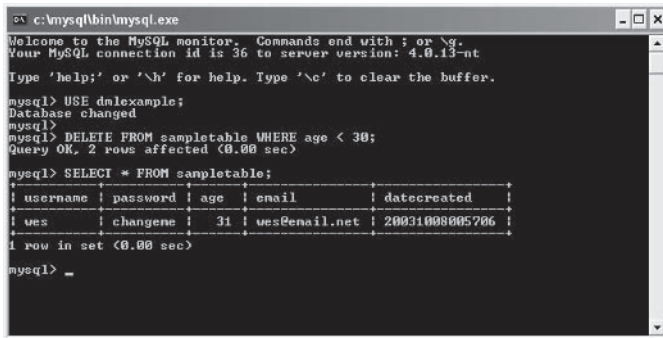
mysql> _
```

Again, as with the update statements, we can specify conditions to allow us to delete, for example, everyone who is younger than 30. Let's do this now with the following statement:

```
mysql> DELETE FROM sampletable WHERE age < 30;
```

When we execute this statement, the client will inform us that two rows have been affected, or in this case deleted. If we now use the `SELECT` command on our table, we will see that only one row is left in the table.

Figure 2-29
Conditional
deleting



```

c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 36 to server version: 4.0.13-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE dmlexample;
Database changed
mysql>
mysql> DELETE FROM sampletable WHERE age < 30;
Query OK, 2 rows affected (0.00 sec)

mysql> SELECT * FROM sampletable;
+-----+-----+-----+-----+-----+
| username | password | age | email | datecreated |
+-----+-----+-----+-----+-----+
| wes | changeme | 31 | wes@email.net | 20031008005706 |
+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> _

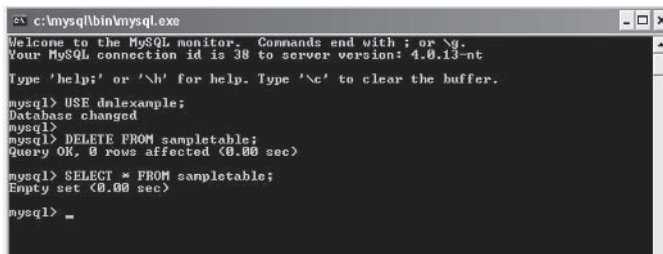
```

Finally, it is also possible to delete all the rows from a table in a single statement. All we need to do for this is not specify any condition, as we did when we updated all the password fields to `changeme` earlier. Here is the statement we require to delete all the rows in a table (i.e., empty the table):

```
mysql> DELETE FROM sampletable;
```

After executing this, if we select all the information in the table using the `SELECT` command, the following will be shown in the console:

Figure 2-30
Deleting all
the data from
a table



```

c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 38 to server version: 4.0.13-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE dmlexample;
Database changed
mysql>
mysql> DELETE FROM sampletable;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM sampletable;
Empty set (0.00 sec)

mysql> _

```

As you can see in Figure 2-30, the table now contains no information as it has all been deleted.

Using **SELECT** Statements

So far we have simply used the following command to show all the data in our `sampletable` table:

```
mysql> SELECT * FROM sampletable;
```

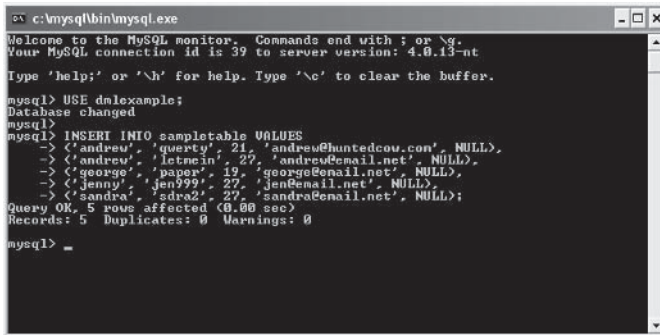
What this actually does is it fetches all the fields from the `sampletable` and returns them. The `*` is a wildcard which basically means it represents anything (or in this case, any field).

Before we discuss the `SELECT` statement further, let's add some data to experiment with to our `sampletable` table. Use the following statement to insert some data:

```
mysql> INSERT INTO sampletable VALUES
-> ('andrew', 'qwerty', 21, 'andrew@huntedcow.com', NULL),
-> ('andrew', 'letmein', 27, 'andrew@email.net', NULL),
-> ('george', 'paper', 19, 'george@email.net', NULL),
-> ('jenny', 'jen999', 27, 'jen@email.net', NULL),
-> ('sandra', 'sdra2', 27, 'sandra@email.net', NULL);
```

Figure 2-31 shows how this should look when we enter it into the console and execute it.

Figure 2-31
Inserting our
new data
into the
`sampletable`
table



```
c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 39 to server version: 4.0.13-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE dmlexample;
Database changed
mysql>
mysql> INSERT INTO sampletable VALUES
-> ('andrew', 'qwerty', 21, 'andrew@huntedcow.com', NULL),
-> ('andrew', 'letmein', 27, 'andrew@email.net', NULL),
-> ('george', 'paper', 19, 'george@email.net', NULL),
-> ('jenny', 'jen999', 27, 'jen@email.net', NULL),
-> ('sandra', 'sdra2', 27, 'sandra@email.net', NULL);
Query OK, 5 rows affected (0.00 sec)
Records: 5  Duplicates: 0  Warnings: 0

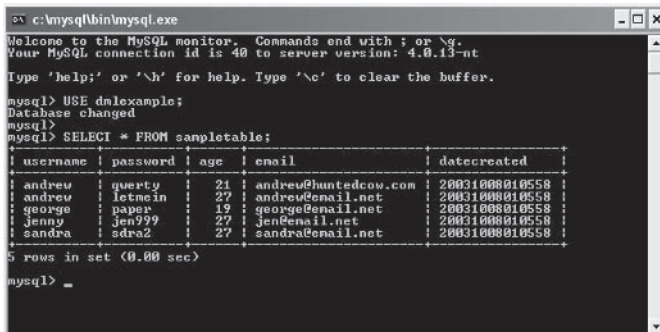
mysql> _
```

Now that we have added our data into the table, if we use the `SELECT` statement with the wildcard (`*`) as we were doing before, it will retrieve and display all of the information from the table to the console. Let's try this with the following statement:

```
mysql> SELECT * FROM sampletable;
```

Here is a screenshot of the output from the console:

Figure 2-32
Using the
wildcard with
a `SELECT`
statement



```
c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 40 to server version: 4.0.13-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE dmlexample;
Database changed
mysql>
mysql> SELECT * FROM sampletable;
+-----+-----+-----+-----+-----+
| username | password | age | email | datecreated |
+-----+-----+-----+-----+-----+
| andrew | qwerty | 21 | andrew@huntedcow.com | 20031008010558 |
| andrew | letmein | 27 | andrew@email.net | 20031008010558 |
| george | paper | 19 | george@email.net | 20031008010558 |
| jenny | jen999 | 27 | jen@email.net | 20031008010558 |
| sandra | sdra2 | 27 | sandra@email.net | 20031008010558 |
+-----+-----+-----+-----+-----+
5 rows in set (0.00 sec)

mysql> _
```

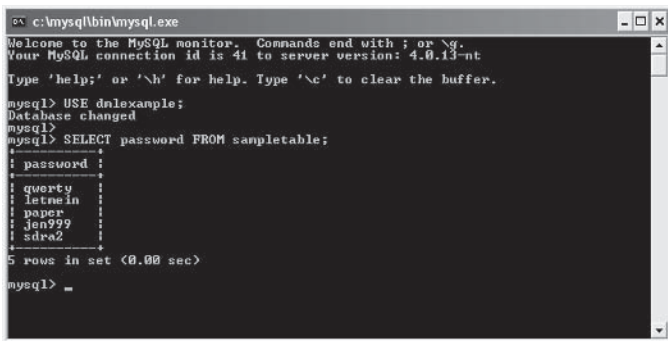
As you can see, the statement has retrieved *all* of the information from the table, that is, all of the rows, and all of the columns contained in each of the rows.

Let's say all we want to retrieve is the password field. To get all of the passwords from the `sampletable` table, we would use this statement:

```
mysql> SELECT password FROM sampletable;
```

When we execute this statement, we can expect the following output from the console:

Figure 2-33
Retrieving
only a single
column



```
c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 41 to server version: 4.0.13-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE dnlexample;
Database changed
mysql> SELECT password FROM sampletable;
+-----+
| password |
+-----+
| querty   |
| letme in |
| paper    |
| jen999   |
| sdra2    |
+-----+
5 rows in set (0.00 sec)

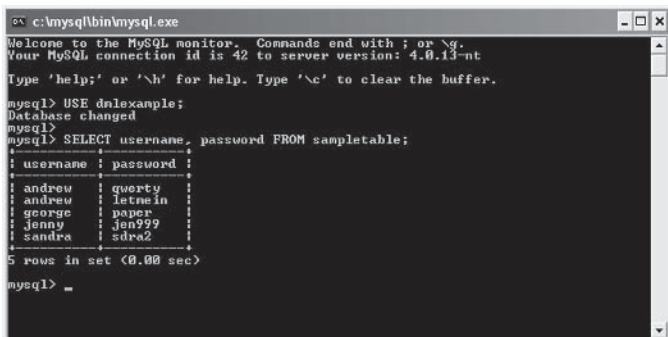
mysql> _
```

Notice how we simply replace the wildcard (*) with the column we wish to retrieve. We can also retrieve multiple columns by using a comma to delimit them. To select both the username column and password column only, use this statement:

```
mysql> SELECT username, password FROM sampletable;
```

When we execute this statement, we can see in the console that only the username and password fields have been selected from the table, as shown in Figure 2-34.

Figure 2-34
Retrieving
multiple col-
umns



```
c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 42 to server version: 4.0.13-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE dnlexample;
Database changed
mysql> SELECT username, password FROM sampletable;
+-----+-----+
| username | password |
+-----+-----+
| andrew   | querty   |
| andrew   | letme in |
| george   | paper    |
| jenny    | jen999   |
| sandra   | sdra2    |
+-----+-----+
5 rows in set (0.00 sec)

mysql> _
```

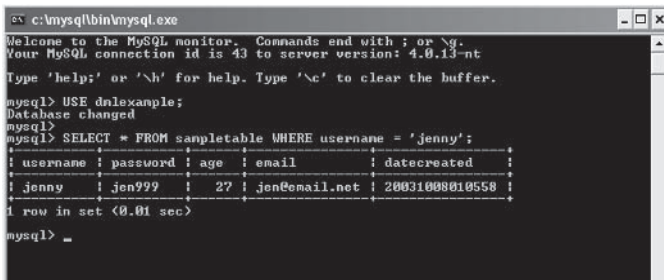
We know how to retrieve individual fields from the tables, but how do we retrieve a single row? We can easily apply a condition to a SELECT statement, just as we did when we were updating the table and deleting

from the table. Using a conditional SELECT statement, let's only display Jenny's information from the database. Here is the statement to do this:

```
mysql> SELECT * FROM sampletable WHERE username = 'jenny';
```

When we execute this statement, only Jenny's details will be displayed in the MySQL client console, as shown in Figure 2-35.

Figure 2-35
Selecting a
single row



```
c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 43 to server version: 4.0.13-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE dnlexample;
Database changed
mysql>
mysql> SELECT * FROM sampletable WHERE username = 'jenny';
+-----+-----+-----+-----+-----+
| username | password | age | email | datecreated |
+-----+-----+-----+-----+-----+
| jenny    | jen999   | 27 | jen@email.net | 20031008010558 |
+-----+-----+-----+-----+-----+
1 row in set (0.01 sec)

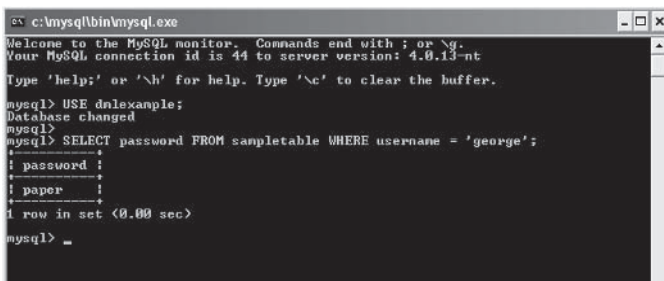
mysql> =
```

We can also select specified fields, such as finding the password that related to a username. Here is how we would retrieve the password that belongs to George:

```
mysql> SELECT password FROM sampletable WHERE username = 'george';
```

When we execute this statement, we can see that only a single field is displayed — George's password, as shown in Figure 2-36.

Figure 2-36
Selecting a
single row
with specified
columns



```
c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 44 to server version: 4.0.13-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE dnlexample;
Database changed
mysql>
mysql> SELECT password FROM sampletable WHERE username = 'george';
+-----+
| password |
+-----+
| paper    |
+-----+
1 row in set (0.00 sec)

mysql> =
```

In our sample data, there are two rows with the username andrew. If we use a conditional statement to get the password for Andrew, we will in fact get two passwords, one for each andrew entry in the database. Here is the statement that will give us that result:

```
mysql> SELECT password FROM sampletable WHERE username = 'andrew';
```

When we execute this statement, we can see that we have two passwords showing in the console. Here is a screenshot of this result:

Figure 2-37
The two-
password
problem!

```

c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 45 to server version: 4.0.13-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE dnlexample;
Database changed
mysql>
mysql> SELECT password FROM sampletable WHERE username = 'andreu';
+-----+
| password |
+-----+
| qwerty  |
| letmein |
+-----+
2 rows in set (0.00 sec)

mysql> _

```

Later in this chapter we discuss a way around this problem with the use of relational databases and keys, but let's not go into that just yet.

Instead, let's have a look at how the LIKE command can help us find needed information. Using LIKE is ideal for finding strings in databases, especially if you only have a part of the complete string (i.e., for a search engine). For example, let's say we wish to find someone in the database who has a name starting with the letter "J." To accomplish this, we would use the following statement:

```
mysql> SELECT * FROM sampletable WHERE username LIKE 'j%';
```

When we execute this statement, we can expect the following output from the MySQL console:

Figure 2-38
Using LIKE
with a SELECT
statement

```

c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 46 to server version: 4.0.13-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE dnlexample;
Database changed
mysql>
mysql> SELECT * FROM sampletable WHERE username LIKE 'j%';
+-----+-----+-----+-----+-----+
| username | password | age | email | datecreated |
+-----+-----+-----+-----+-----+
| jenny    | jen777  | 27 | jen@email.net | 20031008010558 |
+-----+-----+-----+-----+-----+
1 row in set (0.02 sec)

mysql> _

```

Notice here how Jenny was retrieved, as her username is the only one to start with a "J." The % represents a wildcard when used with LIKE, so if we used the following statement instead:

```
mysql> SELECT * FROM sampletable WHERE username LIKE '%j%';
```

it would mean that the letter "J" could appear anywhere in the string. Also note that you can have more than a single character, such as:

```
mysql> SELECT * FROM sampletable WHERE username LIKE '%nny';
```

This would retrieve all of the people who have names that end with the text "nny."

Relational Databases

We have been looking mainly at how to create database structures and do simple data manipulation within them. However, there are a lot of ideas and theories that make databases even more useful to us.

Let's now look at what sort of structure we would want for a relational database. Think of a database that related players in a game to one another, for example to determine who was a friend of each player and, conversely, who was an enemy of each player.

Let's first create a table to store the data for each of the players, with the addition of a *primary key*, which optimizes the database for searches on that particular column. Note also that every row of data in the primary key must be unique to one another. Here is the statement required to create our gamedata database and our playerdata table:

```
mysql> CREATE DATABASE gamedata;

mysql> USE gamedata;

mysql> CREATE TABLE playerdata (
  -> username CHAR(255) UNIQUE NOT NULL,
  -> password CHAR(255),
  -> age INT,
  -> datecreated TIMESTAMP,
  -> PRIMARY KEY(username));
```

Notice here how we set the username column to be UNIQUE and also NOT NULL. In simple terms, this means that it must contain a value and that value must not be the same as any other username in any other record in the table. Note also that we have set the primary key of the table to be the username field, as we will be mainly searching on this field.

Additionally, we need some way of storing a player's friends and enemies. This is done by means of a *link table*. A link table is really just a normal database table, but its main purpose is to relate data in some way or another to conserve space and also optimize the way the database accesses the information.

To create two link tables, one for relating friends and one for relating enemies to each other, use these statements:

```
mysql> CREATE TABLE relatefriends (
  -> player CHAR(255),
  -> friend CHAR(255));
```

And also...

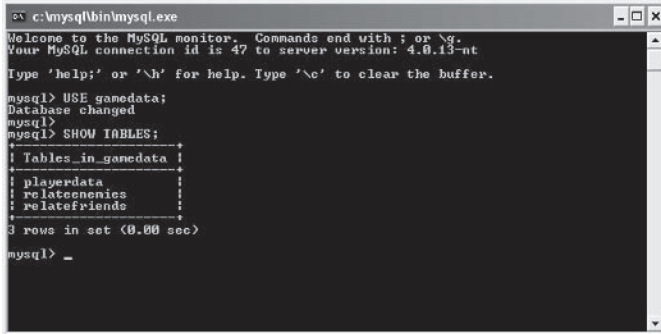
```
mysql> CREATE TABLE relateenemies (
  -> player CHAR(255),
  -> enemy CHAR(255));
```

If we show the tables in the database with the following command:

```
mysql> SHOW TABLES;
```

we can see from Figure 2-39 that our database now contains three different tables, our playerdata table and the two link tables.

Figure 2-39
Our three
tables in the
gamedata
database



```
c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 47 to server version: 4.0.13-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE gamedata;
Database changed
mysql>
mysql> SHOW TABLES;
+-----+
| Tables_in_gamedata |
+-----+
| playerdata          |
| relatecnerics       |
| relatefriends       |
+-----+
3 rows in set (0.00 sec)

mysql> _
```

Let's add some sample data to the playerdata table so we can experiment with the link tables and understand how to use them effectively. Here is the statement to add our sample data into the playerdata table:

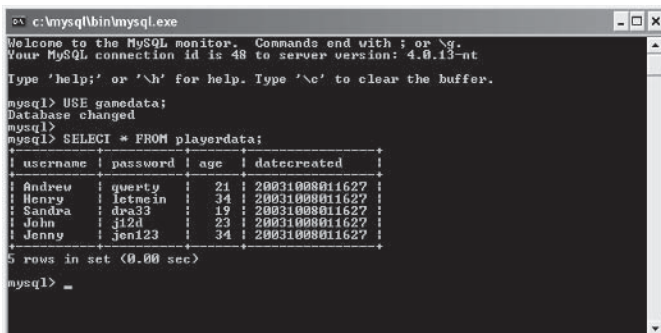
```
mysql> INSERT INTO playerdata VALUES
-> ('Andrew', 'qwerty', 20, NULL),
-> ('Henry', 'letmein', 34, NULL),
-> ('Sandra', 'dra33', 19, NULL),
-> ('John', 'j12d', 23, NULL),
-> ('Jenny', 'jen123', 34, NULL);
```

If we select all the information from the playerdata table now using the following command:

```
mysql> SELECT * FROM playerdata
```

we can see in Figure 2-40 that all of our data is now in the playerdata table.

Figure 2-40
Our data in
the playerdata
table



```
c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 48 to server version: 4.0.13-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE gamedata;
Database changed
mysql>
mysql> SELECT * FROM playerdata;
+-----+-----+-----+-----+
| username | password | age | datecreated |
+-----+-----+-----+-----+
| Andrew  | qwerty   | 21  | 20031008011627 |
| Henry   | letmein  | 34  | 20031008011627 |
| Sandra  | dra33    | 19  | 20031008011627 |
| John    | j12d     | 23  | 20031008011627 |
| Jenny   | jen123   | 34  | 20031008011627 |
+-----+-----+-----+-----+
5 rows in set (0.00 sec)

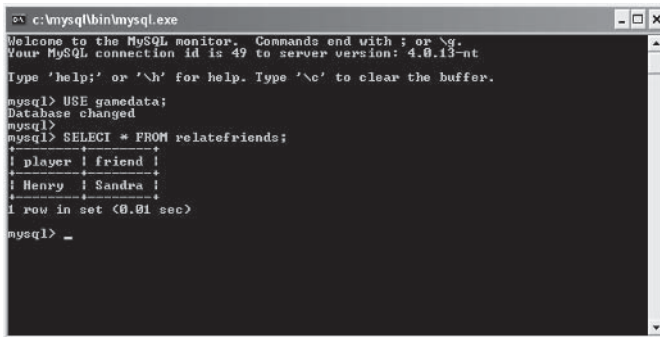
mysql> _
```

Now that we have some sample data, let's create some relations between the players in the database. First, add to the `relatefriends` link table the fact that Henry is friends with Sandra. Here is the statement required to add this to the link table:

```
mysql> INSERT INTO relatefriends VALUES
-> ('Henry', 'Sandra');
```

If we now show all of the data from the `relatefriends` link table, the following will be visible in the MySQL console:

Figure 2-41
Our data in
the friends
table



```
c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 49 to server version: 4.0.13-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE gamedata;
Database changed
mysql>
mysql> SELECT * FROM relatefriends;
+-----+-----+
| player | friend |
+-----+-----+
| Henry  | Sandra |
+-----+-----+
1 row in set (0.01 sec)

mysql> _
```

Let's add some more sample data into both the `relatefriends` and `relateenemies` link tables and then see how we can manipulate the data. Here are the two statements required to add the sample data:

```
mysql> INSERT INTO relatefriends VALUES
-> ('Andrew', 'Henry'),
-> ('Andrew', 'John'),
-> ('Andrew', 'Jenny'),
-> ('Sandra', 'Jenny');
```

and also...

```
mysql> INSERT INTO relateenemies VALUES
-> ('Andrew', 'Sandra'),
-> ('Henry', 'Jenny'),
-> ('Henry', 'John');
```

Let's see if we can find out who Andrew is friends with by using the following statement:

```
mysql> SELECT friend FROM relatefriends WHERE player = 'Andrew';
```

When we execute this statement, the console displays a list of all of the players with which Andrew is friends, as shown in Figure 2-42.

Figure 2-42
Finding out a
player's friend
list

```

c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 50 to server version: 4.0.13-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE gamedata;
Database changed
mysql>
mysql> SELECT friend FROM relatefriends WHERE player = 'Andrew';
+-----+
| friend |
+-----+
| Henry  |
| John   |
| Jenny  |
+-----+
3 rows in set (0.00 sec)

mysql> _

```

When we start implementing databases into Java in the next chapter, we could use this data to find out more information about each of Andrew's friends.

Again, we can do exactly the same with the relateenemies link table. For example, we could find out all of Henry's enemies with the following statement:

```
mysql> SELECT enemy FROM relateenemies WHERE player = 'Henry'
```

When we execute this statement, the following console output can be expected:

Figure 2-43
Finding out a
player's
enemy list

```

c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 51 to server version: 4.0.13-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE gamedata;
Database changed
mysql>
mysql> SELECT enemy FROM relateenemies WHERE player = 'Henry';
+-----+
| enemy |
+-----+
| Jenny |
| John  |
+-----+
2 rows in set (0.00 sec)

mysql> _

```

If we then wanted to find out more information about Henry's enemy who has a username of Jenny, we would use the following statement:

```
mysql> SELECT * FROM playerdata WHERE username = 'Jenny';
```

Here is a screenshot of our expected console output:

Figure 2-44
Finding more
data about an
enemy

```

c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 52 to server version: 4.0.13-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE gamedata;
Database changed
mysql>
mysql> SELECT * FROM playerdata WHERE username = 'Jenny';
+-----+-----+-----+-----+
| username | password | age | datecreated |
+-----+-----+-----+-----+
| Jenny    | jen123  | 34 | 20031008011627 |
+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> _

```

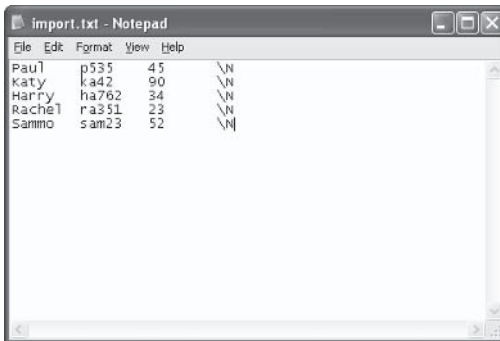
Data Import Methods

Here we learn how to import data into our tables.

Importing from a Text File

Let's now create a text file that contains several records to be added to our playerdata table. We do this simply by denoting each column with a tab and each row by a new line. Figure 2-45 shows five lines of data to be added to the database in Windows Notepad:

Figure 2-45
The data to
import in
Windows
Notepad



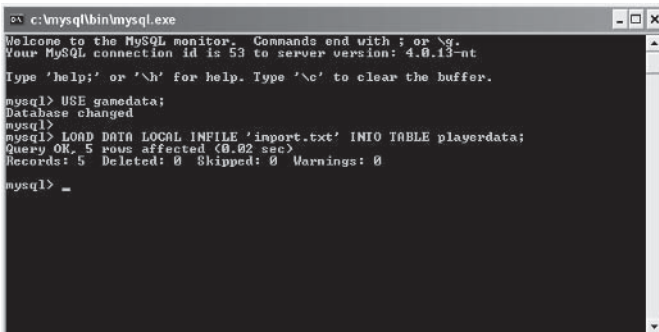
Note how we use \N to specify a field that contains NULL and also that an extra tab is required after each row of data to signify the end of that row. We have saved this file in the MySQL bin directory (i.e., c:\mysql\bin) with the filename import.txt.

Now go to the MySQL console client and enter the following:

```
mysql> LOAD DATA LOCAL INFILE 'import.txt' INTO TABLE playerdata;
```

The console will inform us that five rows have been affected, or in this case added to our database. This can be seen in the following screenshot of the console.

Figure 2-46
Importing
data from a
text file



If we now select all the information from the playerdata table, we can see that our five rows of data have been imported correctly into the database. Here is a screenshot of the client that shows our imported data in the table.

Figure 2-47
The imported
data in our
playerdata
table

```

c:\mysql\bin\mysql.exe
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 56 to server version: 4.0.13-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE gamedata;
Database changed
mysql>
mysql> SELECI * FROM playerdata;
+-----+-----+-----+-----+
| username | password | age | datecreated |
+-----+-----+-----+-----+
| Paul | p535 | 45 | 20031008012732 |
| Katy | ka42 | 90 | 20031008012732 |
| Harry | ha762 | 34 | 20031008012732 |
| Rachel | ra351 | 23 | 20031008012732 |
| Sammo | san23 | 52 | 20031008012732 |
+-----+-----+-----+-----+
5 rows in set (0.00 sec)

mysql> _

```

Importing from a Native Source

Another method for importing data is using the Microsoft Excel spreadsheet program or basically any other application that can export data as tab-delimited data.

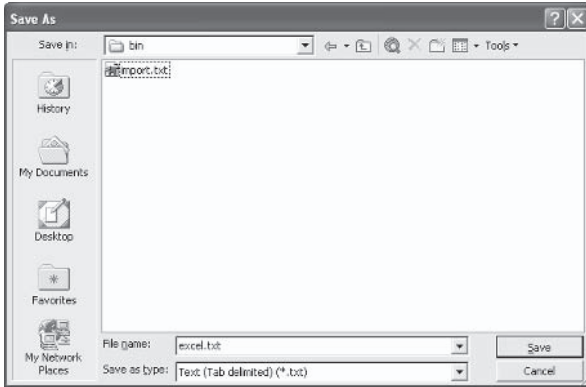
For this example, however, we will use Microsoft Excel. Let's now enter another five rows of data we wish to add to our playerdata table into Excel. Once this is done, it should look roughly the same as Figure 2-48.

Figure 2-48
Entering the
data into
Microsoft Excel

	A	B	C	D	E	F	G	H
1	Ralph	ral45	34	\N				
2	Jean	je26	18	\N				
3	Ian	ia152	20	\N				
4	Laz	la146	23	\N				
5	Peter	pet123	56	\N				
6								
7								
8								
9								
10								
11								
12								
13								
14								
15								
16								
17								
18								
19								

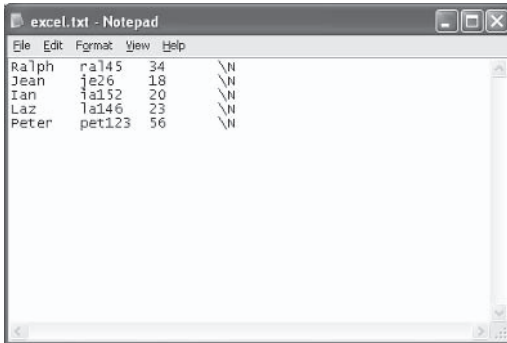
Once our data is entered, we need to save it in a format MySQL can understand. In this case we will use tab-delimited values and save them in a text file called excel.txt in the MySQL bin directory.

Figure 2-49
Saving as a
tab-delimited
text file



Now the process is the same as importing a text file as we did in the last section. In fact, if you open up the text file in Windows Notepad you will see the file format is identical to what we created in the previous section. Figure 2-50 shows how the file looks when we open it up in Notepad.

Figure 2-50
The excel.txt
file in Note-
pad



Backing Up and Restoring Data

Let's turn our attention to how we can export and import a database in MySQL. This is a relatively simple process but is extremely important for backing up data.

Backing up a Database to a File

When we back up a database from MySQL, it is written to a text file and is simply a list of the SQL statements that are required to recreate the database.

Let's export the gamedata database that we created earlier to a text file called gamedata.txt. First we need to open up a command line by clicking Run on the Start button in Windows. When the Run dialog appears, type in command and press the OK button.

NOTE On Window 2000/XP, instead of typing in command, you need to type in cmd.

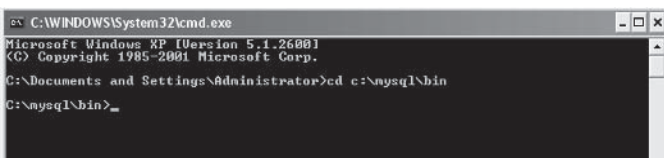
A command-line window will appear. Next you need to go to the bin directory of MySQL using the following command:

```
cd C:\mysql\bin
```

Note that you may have to change the above line if you modified the default MySQL installation directory.

Figure 2-51 shows how this should look.

Figure 2-51
The command-line window (MS-DOS)



Now that we are in the correct directory, we will use a utility called `mysqldump`, which exports a specified database to a file of our choice. Here is how we would export our gamedata database to a text file called gamedata.txt:

```
mysqldump gamedata > gamedata.txt
```

If we open up the text file (which is now located the `mysql\bin\` directory), we can see that it contains many SQL statements and comments added by the `mysqldump` utility. Here is a listing of our exported database text file:

```
# MySQL dump 8.16  
#
```



```
# Host: localhost    Database: gamedata
#-----
# Server version    3.23.47

#
# Table structure for table 'playerdata'
#

CREATE TABLE playerdata (
  username char(255) NOT NULL default '',
  password char(255) default NULL,
  age int(11) default NULL,
  datecreated timestamp(14) NOT NULL,
  PRIMARY KEY (username),
  UNIQUE KEY username (username)
) TYPE=MyISAM;

#
# Dumping data for table 'playerdata'
#

INSERT INTO playerdata VALUES ('Andrew','qwerty',20,20020209203741);
INSERT INTO playerdata VALUES ('Henry','letmein',34,20020209203741);
INSERT INTO playerdata VALUES ('Sandra','dra33',19,20020209203741);
INSERT INTO playerdata VALUES ('John','j12d',23,20020209203741);
INSERT INTO playerdata VALUES ('Jenny','jen123',34,20020209203741);

#
# Table structure for table 'relateenemies'
#

CREATE TABLE relateenemies (
  player char(255) default NULL,
  enemy char(255) default NULL
) TYPE=MyISAM;

#
# Dumping data for table 'relateenemies'
#

INSERT INTO relateenemies VALUES ('Andrew','Sandra');
INSERT INTO relateenemies VALUES ('Henry','Jenny');
INSERT INTO relateenemies VALUES ('Henry','John');

#
# Table structure for table 'relatefriends'
#

CREATE TABLE relatefriends (
  player char(255) default NULL,
  friend char(255) default NULL
) TYPE=MyISAM;

#
```

```
# Dumping data for table 'relatefriends'  
#  
  
INSERT INTO relatefriends VALUES ('Henry','Sandra');  
INSERT INTO relatefriends VALUES ('Andrew','Henry');  
INSERT INTO relatefriends VALUES ('Andrew','John');  
INSERT INTO relatefriends VALUES ('Andrew','Jenny');  
INSERT INTO relatefriends VALUES ('Sandra','Jenny');
```

Restoring a Backed-Up Database

Now it's time to look at how we would go about restoring a database that has been backed up. First, let's drop our gamedata database from MySQL using the following statement:

```
mysql> DROP DATABASE gamedata;
```

Now that our gamedata database has been removed, we need to create a new, empty database to import our data into. So let's do this with the following statement:

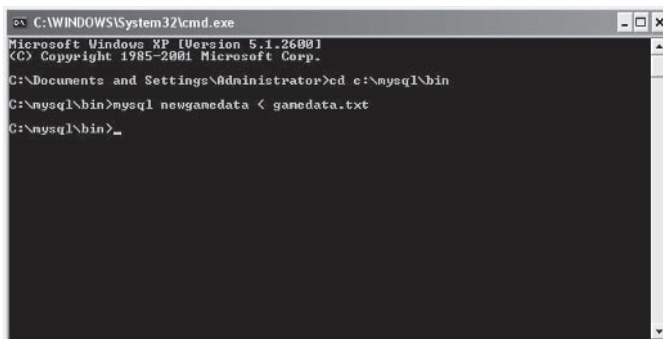
```
mysql> CREATE DATABASE newgamedata;
```

Next, we need to open up a command-line window again (by using the Run dialog and entering command or cmd). Change to the mysql\bin\ directory as we did previously when we were exporting the data and then type in the following command to import the data from our gamedata.txt text file into our newgamedata database:

```
mysql newgamedata < gamedata.txt
```

Figure 2-52 shows a screenshot of the command-line window:

Figure 2-52
Importing a
text file into
MySQL



MySQL C++ Interface

The C++ interface for MySQL, MySQL++, gives the database system a practical use from a game developer's point of view. It allows us to access a MySQL database via the IP address in which the database is stored. This means we can easily retrieve and update data from our database directly within our server applications.

First, you need to copy the MySQL++ libraries. They are on the companion CD in a file called `mysql++-1.7.1-win32-vc++.zip`. Extract the libraries with a utility such as WinZip (a trial version is available on the CD and at <http://www.winzip.com>). You must then include the lib and include directories in Visual Studio 6.

In addition, you must include the lib and include directories from the MySQL directory. Setting up and using static libraries (such as MySQL++) is explained in detail in Chapter 1. Once the library has been set up, we can use it to write an application to retrieve data from a database that is stored in MySQL.

Let's create a simple console application so we can understand the basics of how to access, retrieve, and update information in a database from a C++ application.

Remember to include the `mysql++.lib` in your project settings (see Chapter 1 for more detail on how to do this). You must also copy the `libmysql.dll` from the MySQL++ libraries example directory into your applications directory (or the windows/system directory) in order for your application to execute correctly.

Example 1 — Connecting and Retrieving Data from MySQL

Here is the C++ code we require to create a connection to our `gamedata` database and list all the information from our `playerdata` table. After the code listing, each segment of the code is explained.

```
#include <iostream>
#include <iomanip>
#include <mysql++>

int main(void)
{
    // -> Create a connection to the database
    Connection con("gamedata", "127.0.0.1");

    // -> Create a query object that is bound to our connection
    Query query = con.query();

    // -> Assign the query to that object
```

```

query << "SELECT * FROM playerdata";

// -> Store the results from the query
Result res = query.store();

// -> Display the results to the console

// -> Show the field headings
cout.setf(ios::left);
cout << setw(10) << "username"
    << setw(10) << "password"
    << setw(10) << "age" << endl;

Result::iterator i;
Row row;
// The Result class has a read-only random access iterator
for (i = res.begin(); i != res.end(); i++)
{
    row = *i;
    cout << setw(10) << row["username"]
        << setw(10) << row["password"]
        << setw(10) << row["age"] << endl;
}

return 1;
}

```

In the code, we first create a connection to the server on which the database is stored. We use the following code segment to achieve this:

```
Connection con("gamedata", "127.0.0.1");
```

Connection is simply a class whose constructor takes in the parameters to establish a connection to a MySQL database. The first parameter is the name of the database you wish to connect to. The second is the IP address of the server the database is located on. Notice here that the IP address is 127.0.0.1; this is a special IP address that represents the local machine, i.e., the machine that the C++ application is running on.

Next, we create a query object to allow us to pass queries into the connection we have established with the database. This is done with the following code segment:

```
Query query = con.query();
```

We can now use any standard MySQL query that we have used in the MySQL console with this query variable. We process a query using the following code:

```
query << "SELECT * FROM playerdata";
```

This code does the same as selecting all the information in the playerdata table in the MySQL console.

Next, we store the results from the query in a Result class, which contains a random access iterator for cycling through all the records of data that the query returned. Here is how we assign the query results into the Result class:

```
Result res = query.store();
```

Next, we print the field headings to the screen. This is not essential, but it makes the output data easier to understand.

Now that we have the results in the Result class, we can use the iterator to cycle through all the records that the query returned. We declare the iterator as `i` in this code:

```
Result::iterator i;
```

Then we also want to create a Row class, which will hold each record of data as we cycle through the records with the iterator. We create the Row class as follows:

```
Row row;
```

Finally, we cycle through the data, outputting each record to the screen on a new line. We declare which field we wish to print from the current result by accessing the correct part of the Row class as follows:

```
row["fieldname"];
```

Here is the code we use to cycle through each record contained in the Result class:

```
for (i = res.begin(); i != res.end(); i++)
{
    row = *i;
}
```

Each time through the loop, the current record is assigned to the Row class so we can access individual fields from each record. Therefore, we can print each record using the following code within the for loop:

```
for (i = res.begin(); i != res.end(); i++)
{
    row = *i;
    cout << setw(10) << row["username"]
        << setw(10) << row["password"]
        << setw(10) << row["age"] << endl;
}
```

Here is the expected output from our C++ application:

Figure 2-53

The screenshot shows a Windows console window titled "C:\Documents and Settings\Administrator\Desktop\mysql\example1\Debug\mysql\example1...". The console output displays a table with three columns: 'username', 'password', and 'age'. The data rows are: Paul (p235, 45), Katy (ka42, 90), Harry (ha762, 34), Rachel (ra351, 23), and Sammo (sam23, 52). The prompt 'Press any key to continue...' is visible at the bottom of the console.

username	password	age
Paul	p235	45
Katy	ka42	90
Harry	ha762	34
Rachel	ra351	23
Sammo	sam23	52

Example 2 — Updating Data in MySQL from an Application

In this example, we will modify a field of data from our playerdata table in the gamedata database. First we will display the field, then change the value, and display the field again so we can check that the data has been updated.

We are going to change the password of the player with the username Katy. Here is the code for the C++ console application we require to do this. After the code, we explain how it works.

```
#include <iostream>
#include <iomanip>
#include <mysql++>

int main(void)
{
    // -> Create a connection to the database
    Connection con("gamedata","127.0.0.1");

    // -> Create a query object that is bound to our connection
    Query query = con.query();

    //// DISPLAY BEFORE UPDATE

    // -> Assign the query to that object
    query << "SELECT * FROM playerdata WHERE username = 'Katy'";

    // -> Store the results from the query
    Result res = query.store();

    // -> Display the results to the console

    cout << "Before Update" << endl;
    cout << "-----" << endl;
```

```

// -> Show the field headings
cout.setf(ios::left);
cout << setw(10) << "username"
    << setw(10) << "password"
    << setw(10) << "age" << endl;

Result::iterator i;
Row row;
// The Result class has a read-only random access iterator
for (i = res.begin(); i != res.end(); i++)
{
    row = *i;
    cout << setw(10) << row["username"]
        << setw(10) << row["password"]
        << setw(10) << row["age"] << endl;
}

//// UPDATE THE INFORMATION

// Send an execute and update query in MySQL
query << "UPDATE playerdata SET password = 'qwerty' WHERE
        username = 'Katy'";
query.execute();

//// DISPLAY AFTER UPDATE

// -> Assign the query to that object
query << "SELECT * FROM playerdata WHERE username = 'Katy'";

// -> Store the results from the query
res = query.store();

// -> Display the results to the console

cout << "Before Update" << endl;
cout << "-----" << endl;
// -> Show the field headings
cout.setf(ios::left);
cout << setw(10) << "username"
    << setw(10) << "password"
    << setw(10) << "age" << endl;

// The Result class has a read-only random access iterator
for (i = res.begin(); i != res.end(); i++)
{
    row = *i;
    cout << setw(10) << row["username"]
        << setw(10) << row["password"]
        << setw(10) << row["age"] << endl;
}

return 1;
}

```

The only major difference between this code and the code in Example 1 is the small segment in the middle that updates the field in the `playerdata` table. The code before and after that simply displays the record from the table we are modifying (in the same way as the last example).

Let's look at the middle segment to see how it works.

```
query << "UPDATE playerdata SET password = 'qwerty' WHERE username = 'Katy'";
```

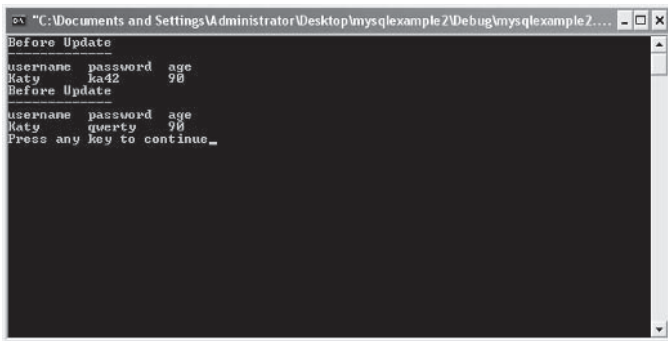
First, we set the query to what we want it to be, just as we would enter it in the MySQL console. Once we have our query set, we then need to execute the query using the following command:

```
query.execute();
```

We did not require this command before as the `store` command that we used to store the results in the `Result` class automatically executes the query.

When we then execute the application, we see the values of the record before and after we update the data. Here is a snapshot of the output from our application:

Figure 2-54



```
"C:\Documents and Settings\Administrator\Desktop\mysqlxample2\Debug\mysqlxample2...
Before Update
username password age
Katy      ka42      98
Before Update
username password age
Katy      qwerty    98
Press any key to continue_
```

As you can see, the password field has been changed successfully from `ka42` to `qwerty` as we intended.

Summary

In this chapter, you learned how to create and use a MySQL database from both the MySQL console and a C++ application. In the next chapter, we move on to learning how to create web-based interfaces for our game servers using the PHP language.

This page intentionally left blank.

Chapter 3

Creating Web-Based Server Interfaces

Introduction

In this chapter we are going to look at the PHP4 language and then learn how to use it to create web-based interfaces for our game servers.

First, however, we will look at how to set up an Apache web server in Windows XP (so we can test our code locally), and then we will install and provide the PHP4 functionality within it. After this, we will look at the foundations of the PHP4 language, moving from accessing a game database to developing a complete web interface for a game server.

Setting Up an Apache 1.3.x Web Server

The first step is to get the Apache package, either by downloading it from <http://www.apache.org> or installing it directly from the companion CD.

The Apache package on the CD has a file named `apache_1.3.28-win32-x86-no_src.msi`. Start the installation by double-clicking this file.

You should then be presented with the following installation splash screen.

Figure 3-1
Installing
Apache 1.3.x



Click Next in the splash screen, read and accept the license, then click Next twice, so that the following dialog is visible.

Figure 3-2
Setting up the
server informa-
tion

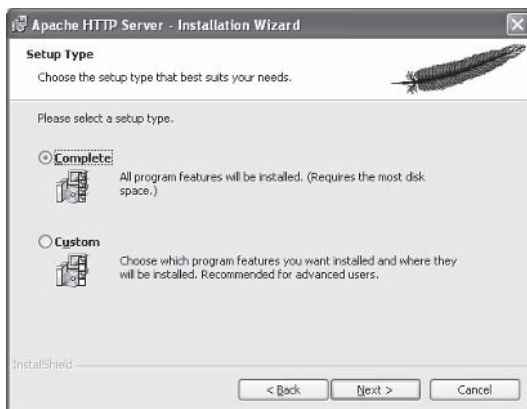


In this dialog, since it is simply a testing server, we can enter 127.0.0.1 (the localhost) in both the Network Domain and the Server Name fields.

Then set the Administrator's Email Address field to something like `andrew@127.0.0.1`, as this again is irrelevant here. Next, ensure the Run as a service for All Users radio button is selected and click Next to continue.

You will then be asked if you wish to install the complete package or perform a custom installation.

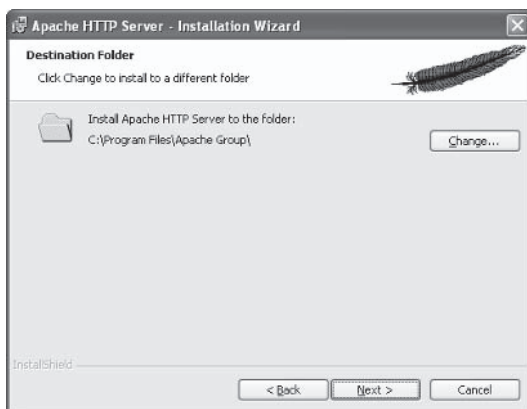
Figure 3-3
Installation
type



Leave this as a Complete install, unless you have reason to do otherwise, and click Next to proceed.

Next, you will be asked to select the location to which Apache should be installed, which will have a subfolder to hold the web site data. The dialog for this is shown in Figure 3-4.

Figure 3-4
Installation
folder

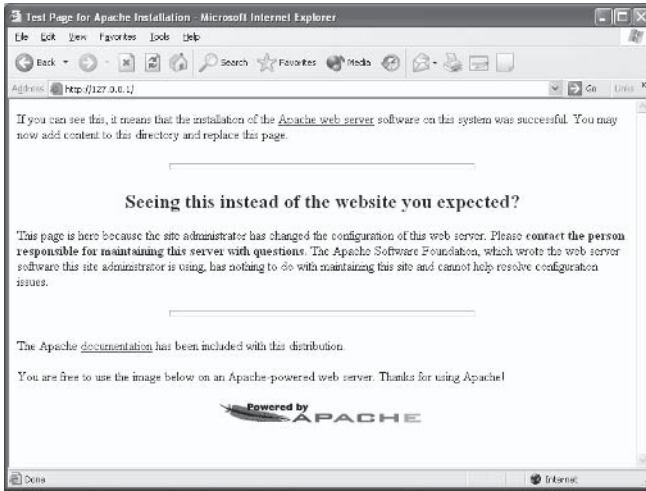


As can be seen in the dialog, the default installation folder is C:\Program Files\Apache Group, which again should be left alone unless you have reason to change it. After accepting the installation folder by clicking Next, you will be prompted to start the installation process. Do this now by clicking the Install button. Apache will install and start running automatically after the installation is complete and also each time Windows starts up.

You can test to see if your installation was successful by entering `http://127.0.0.1/` in your web browser (such as Internet Explorer).

When you do this, you should see a web page displayed as follows:

Figure 3-5
Testing the
installation



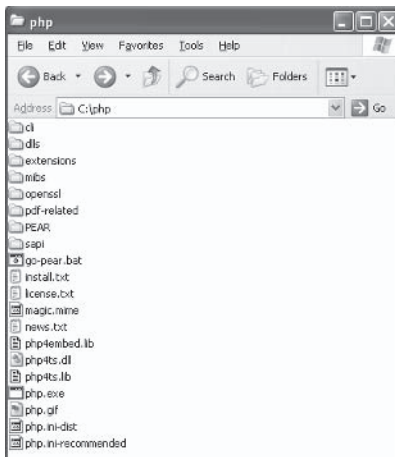
Installing PHP4 for Apache 1.3.x

Once our local testing web server is set up, we can add PHP support to it, allowing pages ending with the extension .php and .php3 to be first parsed by PHP4 before being sent to the browser.

To install PHP4, first grab the PHP4 package php-4.3.4-Win32.zip from either the PHP web site (www.php.net) or the companion CD.

Once you have this zip file, extract it to the folder C:\php, using a utility such as WinZip (the shareware version is available on the CD as well as a download from www.winzip.com). When the files are extracted, the directory structure should look as follows:

Figure 3-6
The extracted
PHP zip file



We then want to copy two dynamic link libraries (DLLs) into the Windows system folder (usually C:\windows\system or C:\winnt\system). Copy the following two files, located within the C:\php and C:\php\sapi folders respectively, into the Windows system folder now: php4ts.dll and php4apache.dll.

Now we want to copy the standard PHP configuration file, php.ini-dist (located in the C:\php folder), into the Windows directory (i.e., C:\windows or C:\winnt). Once there, rename the file to php.ini and open it in Notepad.

We need to make one minor adjustment to the php.ini file to specify the correct temporary folder in which to store session data (we'll learn about this later in the chapter, but basically it's a very easy and safe way of using cookies). So, once the file is open in Notepad, perform a search for the string "/tmp" until you find the following line:

```
session.save_path = /tmp
```

Once this is found, change it to read as follows:

```
session.save_path = c:\temp
```

Then ensure that you create a folder on your C:\ drive called temp. Save the changes to the php.ini file and close Notepad.

The final step is to inform Apache about PHP by editing the Apache configuration file. To do this, first open the configuration file called httpd.conf in Notepad, which is located in the C:\Program Files\Apache Group\Apache\conf folder (providing you have installed Apache to the default directory). Once the file is open in Notepad, add the following two lines to it:

```
LoadModule php4_module c:/php/sapi/php4apache.dll  
AddType application/x-httpd-php .php
```

After this change is made, we need to restart the Apache web server in order to load in the PHP4 module. To restart the server, go into the Start menu and then into the Programs submenu. Select the Apache HTTP Server folder, then Control Apache Server, and then Restart.

After this is done, we need to make sure the PHP integration was successful.

The main web folder is located at the following location: C:\Program Files\Apache Group\Apache\htdocs. So any file placed within this folder would be accessible from the following URL: http://127.0.0.1/.

Therefore, if we created a subfolder within the htdocs folder, such as mysite, the entire path would look as follows: C:\Program Files\Apache Group\Apache\htdocs\mysite.

We could then access this from the URL http://127.0.0.1/mysite/.

As you can see, this is a useful way to organize your different web projects.

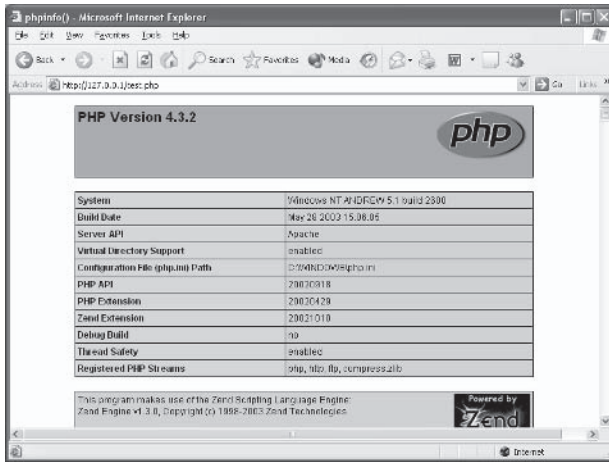
Anyway, back to the point. We need to test PHP, so to do this we can write a very simple PHP script that displays all the information about the PHP installation. So let's now create a file called `test.php` that will be placed within the `htdocs` folder. Once created, open this file in Note-pad, then enter the following into it.

test.php

```
<?php
    print phpinfo();
?>
```

If you then access the URL `http://127.0.0.1/test.php`, you should see the following page visible in the browser:

Figure 3-7
A successful
PHP installa-
tion



If when you access the URL, it does not look like Figure 3-7, but instead looks similar to Figure 3-8....

Figure 3-8
A failed PHP
installation



check that you have:

- Followed the previous installation instructions precisely
- Named the file `test.php` and not `test.php.txt`
- Restarted the Apache server

Now that you have successfully installed Apache and PHP4, we are going to look at the basics of the PHP language.

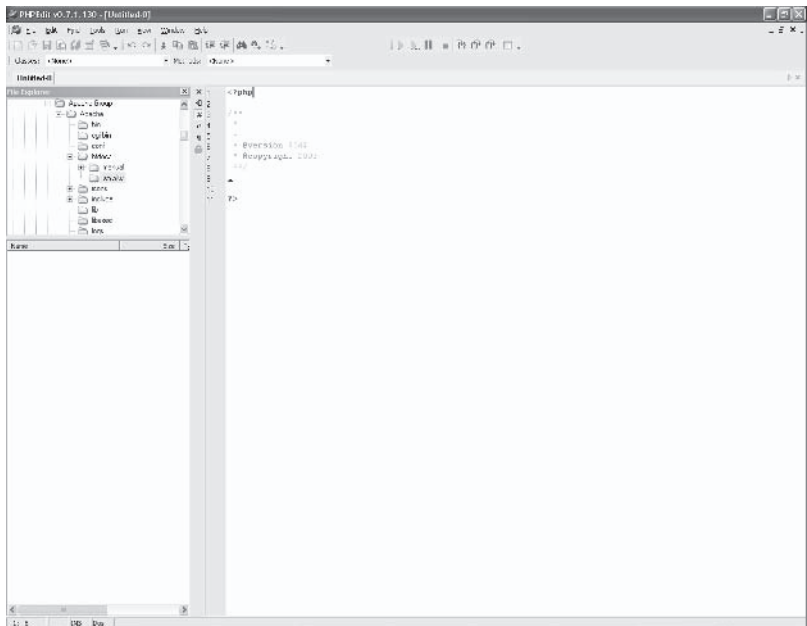
Using PHP: Hypertext Preprocessor

Now that PHP is installed successfully, the next step is to grab a decent editor, as Notepad doesn't quite suffice for coding PHP. The best free IDE (integrated development environment) in our opinion is PHPEdit (<http://www.phpedit.net>); however, if you have some money to burn, you may wish to take a look at Dreamweaver MX (<http://www.macromedia.com>).

Which IDE you use is really personal preference, so experiment with different ones and see which you prefer.

To get started, let's create a folder in the htdocs folder called testsite, then, assuming you're using our editor of choice (PHPEdit), browse to the folder using the File Explorer window. When this is done, depending on how you lay out the IDE, it should look something similar to the following.

Figure 3-9
PHPEdit IDE



The Basics

In the previous version of PHP (version 3), PHP files had the file extension `.php3`, but since the fourth release all PHP files should have the extension `.php`. This indicates to the Apache web server that the file should be parsed by the preprocessor before it is output to the client's web browser.

Just as `index.html` and `index.htm` are recognized as default pages that should be loaded when someone types in a URL, so can `index.php` be used as the default page.

All PHP code with a `.php` file extension needs to be placed within special tags. Before you write any PHP code, you must first tell the preprocessor that you are going to do so. This is indicated by specifying the following tag:

```
<?php
```

After you have finished writing PHP code, you close the tag with the following:

```
?>
```

Within the PHP tags, you can also add comments to your code in exactly the same way as C++:

```
// single line comment
```

```
/* multi  
   line  
   comment */
```

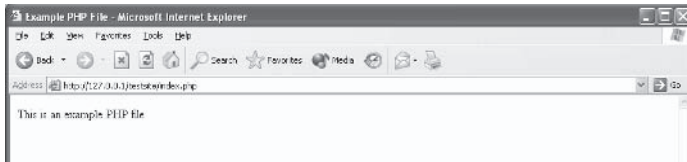
Let's try an example. Save a file within the IDE called `index.php` to the `testsite` folder.

Example 1 — `index.php`

```
<html>  
  <head>  
    <title>Example PHP File</title>  
  </head>  
  <body>  
    This is an example PHP file  
  </body>  
</html>
```

When we call the example in the web browser (`http://127.0.0.1/testsite/index.php`), we should see the output shown in Figure 3-10.

Figure 3-10
Example 1
output



Wait! That's just a standard HTML file with no PHP, yet it's in a PHP file! Well, basically anything outside of the PHP tags is interpreted as standard HTML and is output to the browser as normal. This is very useful as we can turn PHP on and off as we require — even hundreds of times within the same source file. Let's look at a more exciting example that uses variables.

Variables

Although PHP does have types of variables (such as `INT`, `FLOAT`, etc.), they are all handled internally so there is little need to worry about the actual type of your variables. Note, however, that it is possible to directly set and get the types. We will look into this in more detail later.

As well as types not being overly important, there is also no need to declare a variable; you can just initialize it and use it directly. Let's expand upon our previous example to show how to assign and print variables to the screen.

Example 2 — `index2.php`

```
<?php

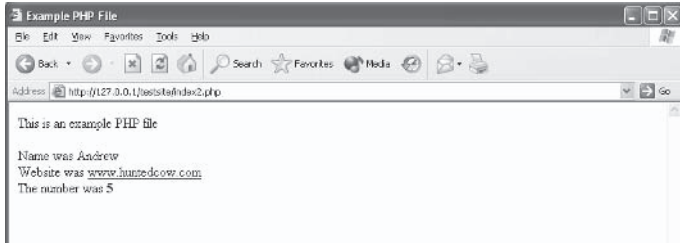
$name = "Andrew";
$website = "www.huntedcow.com";
$number = 5;

?>

<html>
  <head>
    <title>Example PHP File</title>
  </head>
  <body>
    This is an example PHP file
    <br><br>
    <?php
      print "Name was $name";
      print "<br>Website was <a href=\"http://\".$website.\">
        $website</a>";
      print "<br>The number was <b>$number</b>";
    ?>
  </body>
</html>
```

When we load Example 2 in the browser (<http://127.0.0.1/testsite/index2.php>), it should look somewhat like the following:

Figure 3-11
Example 2
output



At the top of the file, we start with the following block of code:

```
<?php
    $name = "Andrew";
    $website = "www.huntedcow.com";
    $number = 5;
?>
```

All we are doing here is entering a PHP block and assigning three variables. Note that variables are defined by the use of the “\$” sign and we use the “” to denote strings, whereas we just assign numbers directly. We then turn PHP off using the `?>` tag and continue by printing out HTML as we did in the previous example.

Then, we enter PHP again and print out the `$name` variable using the following line of code:

```
print "Name was $name";
```

Notice in this line how we actually have the `$name` variable within another string that we wish to print. Before the string is output to the browser, the value of `$name` is replaced with the actual value, which in this example is “Andrew.”

The next line creates a web link using HTML and places the `$website` variable within the href and also after it to display it to the browser. This can be seen in the following line of code:

```
print "<br>Website was <a href=\"http://\".$website.\">$website</a>";
```

Notice how we have used “\” within the string. This is possible as we have “escaped” them by adding a backslash \ before each occurrence.

Finally, we printed the number variable `$number` using the following line of code:

```
print "<br>The number was <b>$number</b>";
```

Note in this final line how we have placed a bold HTML text tag in front of the `$number` variable so that when it is printed it comes out bold.

Operators and Loops

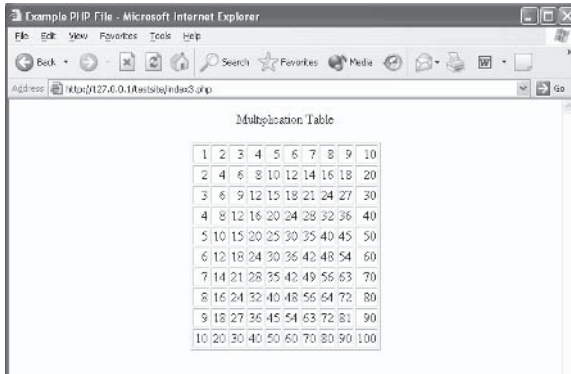
Next in our brief overview we are going to take a look at simple mathematical operators and loops. PHP supports all the basic operators as well as the shortcuts that C++ contains (such as the += operator). Let's look at the next example where we print out a multiplication table.

Example 3 — index3.php

```
<html>
  <head>
    <title>Example PHP File</title>
  </head>
  <body>
    <center>
      Multiplication Table
      <br><br>
      <table border=1>
        <?php
          for($i = 1; $i <= 10; $i++)
          {
            ?><tr><?php
              for($j = 1; $j <= 10; $j++)
              {
                print "<td align=\"right\">";
                print ($i*$j);
                print "</td>";
              }
            ?></tr><?php
          }
        ?>
      </table>
    </body>
  </html>
```

When you run Example 3 in the browser (<http://127.0.0.1/testsite/index3.php>), you should see the multiplication table shown in Figure 3-12.

Figure 3-12
Example 3
output



To display the multiplication table, we set up two for loops. The first is:

```
for($i = 1; $i <= 10; $i++)
```

This initializes a variable called `$i` to the value of 1, then increments it by 1 each time through the loop. Inside this loop, a table row is initialized in HTML and another loop is created to fill in the row with values. This can be seen here:

```
for($j = 1; $j <= 10; $j++)
{
    print "<td align=\"right\">";
    print ($i*$j);
    print "</td>";
}
```

Then, before the outer loop is terminated, the table row is ended.

Conditional Statements

Next, we will look at how to construct a simple if statement. For an example, we will expand upon the previous multiplication table example and make the top and left-hand numbers bold. Take a look at Example 4.

Example 4 — index4.php

```
<html>
  <head>
    <title>Example PHP File</title>
  </head>
  <body>
    <center>
      Multiplication Table
      <br><br>
      <table border=1>
        <?php
          for($i = 1; $i <= 10; $i++)
```

```

    {
        ?><tr><?php
        for($j = 1; $j <= 10; $j++)
        {
            print "<td align=\"right\">";

            if($i == 1 || $j == 1)
                print "<b>";

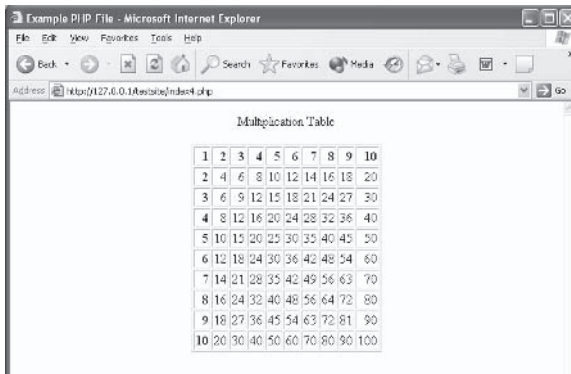
            print ($i*$j);
            print "</td>";
        }

        ?></tr><?php
    }
?>
</table>
</body>
</html>

```

When this is executed within the browser (<http://127.0.0.1/testsite/index4.php>), it should look like the following:

Figure 3-13
Example 4
output



As you can see from the code, all we have added in this example is the following two lines:

```

if($i == 1 || $j == 1)
    print "<b>";

```

All this if statement does is print the `` tag to the browser if the `$i` or `$j` variables are equal to 1. Simple stuff!

The other important conditional statement is the switch statement, which works in the same manner as it does in C++. However, in PHP it is also possible to use strings within the case. For example:

```

$mystring = "andrew";

switch($mystring)
{

```

```
    case "andrew":
        // do something
        break;

    case "teijo":
        // do something else
        break;
}
```

Arrays

As with variables, there is no need to initialize arrays and they are treated in exactly the same manner as variables. For example, we could set three values in an array as follows:

```
$myarray[0] = "Andrew";
$myarray[1] = "Teijo";
$myarray[2] = "Wes";
```

Then we could display them with the following code segment:

```
for($i=0; $i<count($myarray); $i++)
    print $myarray[$i];
```

Functions

When we start thinking about functions, we get into the area of variable scope. In PHP, when a variable is declared within a function, its value is local to that function unless the global keyword is specified before it. Let's look at a simple example of how to create a function to print a string of text that is passed into it.

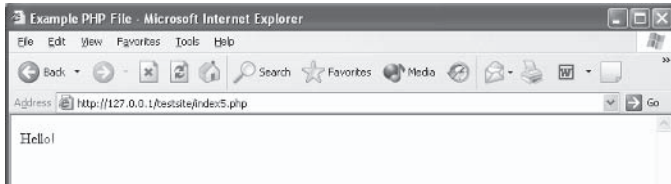
Example 5 — index5.php

```
<html>
    <head>
        <title>Example PHP File</title>
    </head>
    <body>
        <?php
            WriteString("Hello!");
        ?>
    </body>
</html>

<?php
function WriteString($theString)
{
    print $theString;
}
?>
```

When this is executed in the browser (<http://127.0.0.1/testsite/index5.php>), the following should be shown:

Figure 3-14
Example 5
output



User Input

Now we get into the territory where PHP differs from C++ in many ways, purely due to the integration with HTML. To gather input from the user, we use HTML forms. For an example of user input, we are going to create two PHP scripts. The first, `input.php`, will simply display an HTML form. The second, `output.php`, will display the information that was gathered by the form. Here are the two files that are required to make this example work.

Example 6a — `input.php`

```
<html>
  <head>
    <title>Example PHP File</title>
  </head>
  <body>
    <center>
      <form method="post" action="output.php">
        <table border=1>
          <tr>
            <td>Your name:</td><td><input
              type="text" name="YourName"></td>
            </tr>
          <tr>
            <td>Your favourite color:</td>
            <td>
              <select name="YourColor">
                <option value="Red">Red</
                  option>
                <option value="Blue">Blue</
                  option>
                <option value="Green">
                  Green</option>
              </select>
            </td>
            </tr>
          <tr>
            <td>Over 18?:</td><td><input type=
              "checkbox" name="Over18"></td>
          </tr>
        </table>
      </form>
    </center>
  </body>
</html>
```



```

        </tr>
        <tr>
            <td colspan=2>
                <center>
                    <input type="submit"
                        value="Send Data">
                </center>
            </td>
        </tr>
    </table>
</form>
</center>
</body>
</html>

```

Example 6b — output.php

```

<html>
  <head>
    <title>Example PHP File</title>
  </head>
  <body>
    <center>
      <?php
        print "Name was ".$_POST["YourName"]."<br>";
        print "Color was ".$_POST["YourColor"]."<br>";

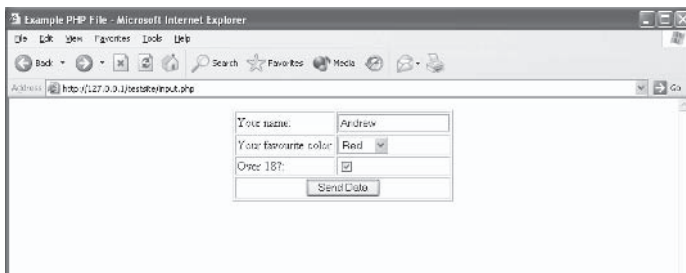
        if($_POST["Over18"] == "on")
        {
            print "Over 18 was ticked";
        }
        else
        {
            print "The user was under 18";
        }

      ?>
    </center>
  </body>
</html>

```

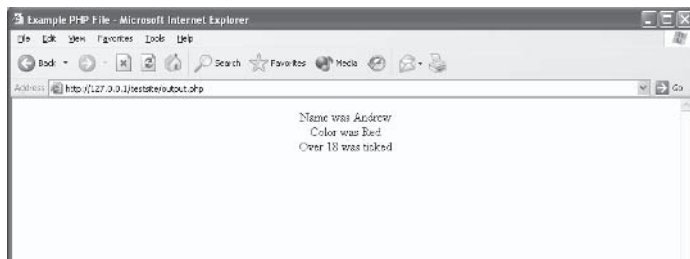
When we load the input.php file into the web browser and fill in some information, it will look similar to the following (<http://127.0.0.1/testsite/input.php>):

Figure 3-15
Example 6 —
input.php



When the Send Data submit button is clicked, we should expect something similar to the following to be visible.

Figure 3-16
Example 6—
output.php
after being
passed form
data



So how did the information get sent? When using a form, there are two methods of sending information to another page. The best method is the POST method. If you look at where we have declared the form in `input.php`, you will see this.

```
<form method="post" action="output.php">
```

Note also that we have specified the action as `output.php`, which means that any data collected by the form will be sent to the `output.php` script upon the user clicking the Submit button. When the data arrives in `output.php`, it is stored within a global array called `$_POST`, which is accessed associatively via the name the input was given on the form. For example, the name input was declared in the form as follows:

```
<input type="text" name="YourName">
```

Which means that when this is passed to `output.php`, it can be accessed by referencing:

```
$_POST["YourName"]
```

Similarly, if the form method is specified as GET (which means the data is attached to the end of the URL), the global `$_GET` array can be used to access the data in the same way.

The “Command” System

Now that we have skimmed the basics of PHP, let’s start to look at how to correctly structure our code and keep it maintainable for larger projects. What you’re about to learn is the “command” system. (I’m not sure if it has an official name, but I think it’s an okay name for it 😊.)

The idea behind this is to give our code a central processing point that then references methods to actually deal with the functionality. Let’s take a look at the following `core.php` and `welcome.php` files:

Example 7a — core.php

```
<?php

// include libraries here...
include("welcome.php");

// command processing...
switch($cmd)
{
    default:
        showWelcome();
        break;
}

?>
```

Example 7b — welcome.php

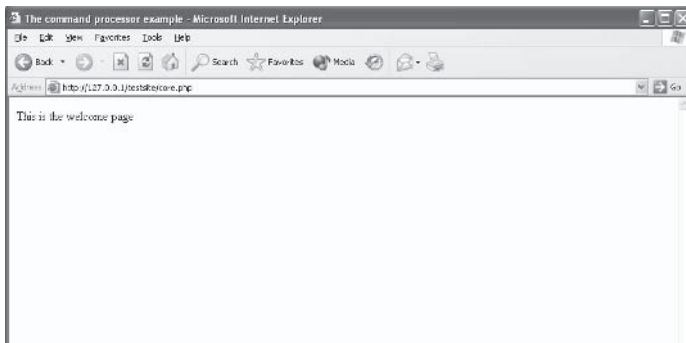
```
<?php

function showWelcome()
{
    ?>
    <html>
        <head>
            <title>The command processor example</title>
        </head>
        <body>
            This is the welcome page
        </body>
    </html>
    <?php
}

?>
```

When we execute this example in the browser (<http://127.0.0.1/testsite/core.php>), we should expect the following to be visible:

Figure 3-17
Example 7 —
core.php



In our `core.php` file, we first call the `include` function and pass in the filename `welcome.php`, which basically includes the `welcome.php` file when executing the code. Then we create a switch statement that examines the value of a `$cmd` variable (which, as it is not contained within a function, is global). We then make the default of the switch statement call the `showWelcome` method, which is defined within the `welcome.php` file and simply prints “This is the welcome page” to the browser.

Let’s expand upon this to create multiple pages. Before this, however, we are going to create a `header.php` and a `footer.php` file which will contain the methods `showHeader` and `showFooter`, respectively. We’ll then call the header method before and the footer method after the switch statement in `core.php` so we can put any standard look and feel of the page there without rewriting it for each page. These files will look as follows:

header.php

```
<?php
function showHeader()
{
    ?>
    <html>
        <head>
            <title>Command Processor</title>
        </head>
        <body bgcolor="#0000AA">
            <center>
                <table width="600" border=1 bgcolor="#FFFFFF">
                    <tr>
                        <td>
    <?php
}
?>
```

footer.php

```
<?php
function showFooter()
{
    ?>
    </td>
    </tr>
    </table>
    </center>
    </body>
</html>
```

```
    <?php
}
?>
```

If we then create the files `page1.php` and `page2.php`, adding some text and hyperlinks into them, we will have the following four files (excluding the header and footer):

Example 8a — core.php

```
<?php

// include libraries here...
include("header.php");
include("footer.php");

include("welcome.php");
include("page1.php");
include("page2.php");

// command processing...
showHeader();

switch($cmd)
{
    case "page1":
        showPage1();
        break;

    case "page2":
        showPage2();
        break;

    default:
        showWelcome();
        break;
}

showFooter();

?>
```

Example 8b — welcome.php

```
<?php
function showWelcome()
{
    ?>
    <center>
        <b>Welcome!</b>
        <br><br>
    </center>
}
```

```
        Please click one of the following pages to visit it...
    </center>
    <ul>
        <li><a href="core.php?cmd=page1">Page 1</a></li>
        <li><a href="core.php?cmd=page2">Page 2</a></li>
    </ul>
    <?php
}
?>
```

Example 8c — page1.php

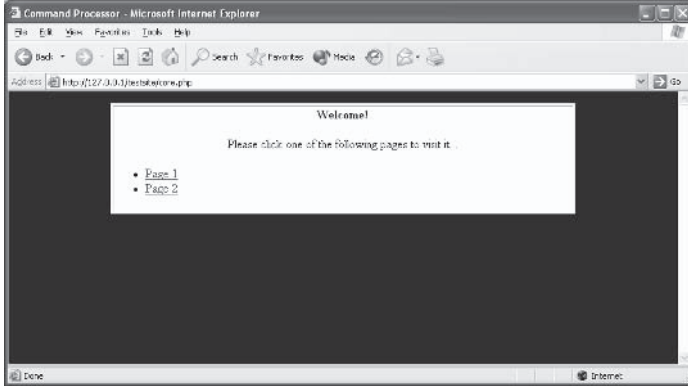
```
<?php
function showPage1()
{
    ?>
    <center>
        <b>Page 1</b>
        <br><br>
        Welcome to page 1! Why not <a href="core.php?cmd=page2">visit
        page 2?</a>
    </center>
    <?php
}
?>
```

Example 8d — page2.php

```
<?php
function showPage2()
{
    ?>
    <center>
        <b>Page 2</b>
        <br><br>
        Welcome to page 2! Why not either <a href="core.php?cmd=page1">
        go back to page 1</a>
        or <a href="core.php">visit the welcome page again.</a>
    </center>
    <?php
}
?>
```

When you load up the `core.php` file in the browser, you should see the following:

Figure 3-18
Example 8 —
`core.php`



Clicking on either of the Page 1 or Page 2 links will display the respective pages. Notice the following in the address bar of the browser when you click the Page 1 link: `http://127.0.0.1/testsite/core.php?cmd=page1`.

As you can see, the variable `cmd` has been set to the value `page1`. This is generated automatically when you use a form (with the GET method) and is done without placing the information in the URL with the POST method. To add multiple variables to the URL you can delimit them with the “&” character. For example:

```
http://127.0.0.1/testsite/core.php?cmd=page1&myothervar=blah
```

Accessing MySQL

Let’s see how we can utilize the SQL language we discovered in the previous chapter with PHP. MySQL works exceptionally well with PHP and there are in fact built-in functions for interacting with MySQL. In this section we are going to look at these functions and see how to store, retrieve, and modify information in the database directly from the web browser.

MySQL Example 1 — Connecting and Disconnecting

In this first example we are going to find out how to connect and terminate a connection to a database. Let’s first create a database in MySQL by executing the following at the MySQL command prompt:

```
mysql> CREATE DATABASE phptest;
```

After that is created, we can make a PHP script that will connect to, use, and disconnect from the database. Let's take a look at this script now, which we have called `mysql1.php`.

`mysql1.php`

```
<?php

// connect...
$dbh = mysql_connect("localhost", "root", "");

// select the database...
if(!mysql_select_db("phptest", $dbh))
    print "Unable to connect to db";

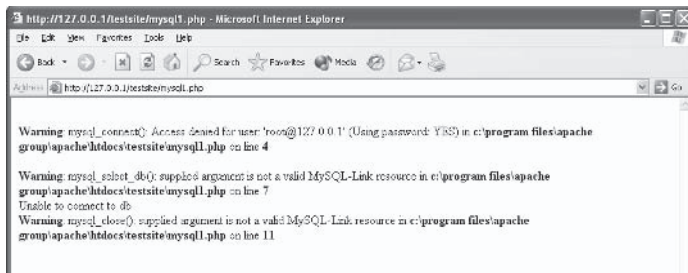
// disconnect...
mysql_close($dbh);

?>
```

In this example, we first make a call to `mysql_connect`, which takes in the host the database is running on (which 99 percent of the time will be the computer/server the script is executing on). The second parameter is the username and the third is the password required to connect to the database server.

If the connection fails, PHP will output descriptive errors to the browser, which will look something like the following (we've changed the password to be incorrect):

Figure 3-19
A connection
failure



MySQL Example 2 — Storing and Retrieving Data

Now that we know how to connect to the database, let's look at how we can store and display information stored within the database. For this, we are going to be using the command processor setup we learned earlier in this chapter.

Before we delve into the PHP, let's first create a table called `users` within the database for storing players' names, ages, aliases, and passwords. Here is the MySQL query we require to do this:


```
mysql> create table users (  
    -> id int auto_increment,  
    -> name tinytext,  
    -> age int,  
    -> alias tinytext,  
    -> password tinytext,  
    -> primary key(id));
```

Now let's look at the complete source code for this example, then we will look into detail at how it works. (Note that we also require the header.php and footer.php files we created earlier.)

mysqlcore.php

```
<?php  
  
// include libraries here...  
include("header.php");  
include("footer.php");  
  
include("users.php");  
  
// special commands (no header/footer)  
switch($cmd)  
{  
    case "doadduser":  
        doAddUser();  
        exit();  
}  
  
// command processing...  
showHeader();  
  
switch($cmd)  
{  
    case "adduser":  
        addUser();  
        break;  
  
    default:  
        showUsers();  
        break;  
}  
  
showFooter();  
  
?>
```

users.php

```

<?php
function showUsers()
{
    ?>
    <center>
        <br><b>Registered Users</b><br><br>
        <a href="mysqlcore.php?cmd=adduser">Add User</a>
        <br><br>
        <table width="450" border="1" cellpadding="4">
            <tr bgcolor="#BBBBBB">
                <td width="25%"><b>Name</b></td>
                <td width="25%"><b>Age</b></td>
                <td width="25%"><b>Alias</b></td>
                <td width="25%"><b>Password</b></td>
            </tr>
        </table>
    <?php
    // connect...
    $dbh = mysql_connect("localhost", "root", "");

    // select the database...
    mysql_select_db("phptest", $dbh);

    // fetch all the users...
    $results = mysql_query("SELECT * FROM users ORDER BY name",
                          $dbh);
    while($row = mysql_fetch_array($results))
    {
        print "<tr>";
        print "<td>".htmlspecialchars($row["name"])."</td>";
        print "<td>".htmlspecialchars($row["age"])."</td>";
        print "<td>".htmlspecialchars($row["alias"])."</td>";
        print "<td>".htmlspecialchars($row["password"])."</td>";
        print "</tr>";
    }

    mysql_close($dbh);

    ?>
    </table>
</center>
<?php

function addUser()
{
    ?>
    <center>
        <br><b>Add User Form</b><br><br>
        <form method="POST" action="mysqlcore.php">

```

```

<input type="hidden" name="cmd" value="doadduser">
<table width="450" border="1" cellpadding="4">
  <tr>
    <td>Name:</td>
    <td><input type="text" name="name"></td>
  </tr>
  <tr>
    <td>Age:</td>
    <td><input type="text" name="age" size=3></td>
  </tr>
  <tr>
    <td>Alias:</td>
    <td><input type="text" name="alias"></td>
  </tr>
  <tr>
    <td>Password:</td>
    <td><input type="password" name=
      "password"></td>
  </tr>
</table>
<br><br>
<input type="submit" value="Add User">
<input type="button" value="Cancel" onClick=
  "window.location = 'mysqlcore.php'">
</form>
<?php
}

function doAddUser()
{
  // connect...
  $dbh = mysql_connect("localhost", "root", "");

  // select the database...
  mysql_select_db("phptest", $dbh);

  // add the new user...
  mysql_query("INSERT INTO users VALUES (NULL, \"".$_POST["name"]."\",
    \"".$_POST["age"]."\", \"".$_POST["alias"]."\",
    \"".$_POST["password"]."\""); $dbh);

  // close the connection...
  mysql_close($dbh);

  // Redirect the page...
  header("Location: mysqlcore.php");
}
?>

```

When we execute this in the browser (<http://127.0.0.1/testsite/mysqlcore.php>), we should have two screens available that look as follows (note that we added two users before taking the screen shots).

Figure 3-20
Registered
users

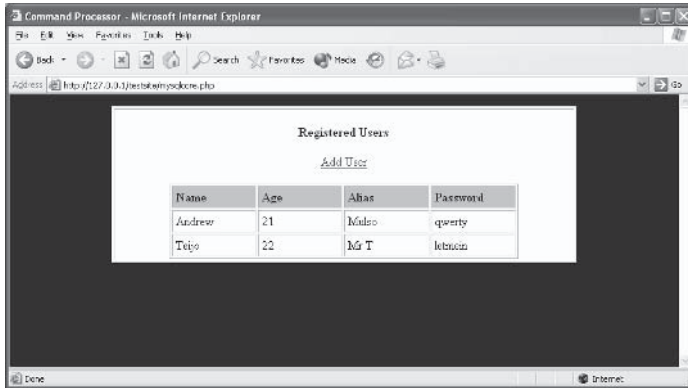
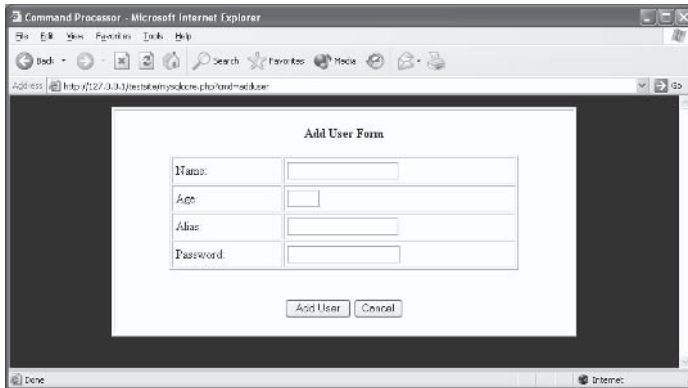


Figure 3-21
The Add User
screen



Let's now see how this works.

First, we have the `showUsers` function in `users.php`, which displays a list of all the users stored in the `users` table, ordered by the name. In this method, we first connect to the database as we did in the previous example using the following two lines of code:

```
$dbh = mysql_connect("localhost", "root", "");
mysql_select_db("phptest", $dbh);
```

After we are connected, we execute a `SELECT` query by calling the `mysql_query` method, passing in the query we wish to execute and the handle to the database (which we obtained with the `mysql_connect` method).

```
$results = mysql_query("SELECT * FROM users ORDER BY name", $dbh);
```

We then have a handle to the result set in a variable called `$results`. We can use this to actually retrieve the results into an associative array using the `mysql_fetch_array` function.

```
while($row = mysql_fetch_array($results))
{
```

The array `$row` can be accessed by passing the string names of the field names in the database into it (such as `name` and `alias`).

Within this `while` loop, a new table row (`<tr>`) is created for each entry in the database and it is printed out to the browser.

The `addUser` function simply displays the HTML form for gathering the information; however, note the following line within the form:

```
<input type="hidden" name="cmd" value="doadduser">
```

By adding a “hidden” input to the form, we can specify a command for our command processor to interpret; in this case we have specified the `doadduser` command, which will in turn call the `doAddUser` function.

The `doAddUser` function is called in the `mysqlcore.php` before the `showHeader` function is called as it is not intended to display any information. Basically, the `doAddUser` function will insert the data into the MySQL database and send the browser to a different page without the user ever knowing. The reason behind this is simple: If the browser was not redirected, the user would be inserted into the database again if he hit Refresh on the page.

So, in the `doAddUser` function, once the database connection is established, it is then a simple case of performing an INSERT query, feeding the form data into the SQL statement. This can be seen in the following line of code:

```
mysql_query("INSERT INTO users VALUES (NULL, \"\".$_POST[\"name\"].\"\",
    \"\".$_POST[\"age\"].\"\", \"\".$_POST[\"alias\"].\"\",
    \"\".$_POST[\"password\"].\"\"), $dbh);
```

After this, the connection is closed and a call to the PHP header method is made. Into that is passed the location to which the browser should be redirected. This can be seen here:

```
header("Location: mysqlcore.php");
```

MySQL Example 3 — Updating and Removing Data

In this final MySQL example, we look at how to modify and remove data directly from PHP by expanding upon the previous example.

Let’s first look at the complete modified code and then we will look into the required changes.

mysqlcore.php

```
<?php

// include libraries here...
include("header.php");
include("footer.php");

include("users.php");

// special commands (no header/footer)
switch($cmd)
{
    case "doadduser":
        doAddUser();
        exit();
    case "deleteuser":
        deleteUser();
        exit();
}

// command processing...
showHeader();

switch($cmd)
{
    case "adduser":
        addUser();
        break;

    default:
        showUsers();
        break;
}

showFooter();

?>
```

users.php

```
<?php
function showUsers()
{
    ?>
    <center>
        <br><b>Registered Users</b><br><br>
        <a href="mysqlcore.php?cmd=adduser">Add User</a>
        <br><br>
        <table width="450" border="1" cellpadding="4">
            <tr bgcolor="#BBBBBB">
```

```

        <td width="25%"><b>Name</td>
        <td width="25%"><b>Age</td>
        <td width="25%"><b>Alias</td>
        <td width="25%"><b>Password</td>
        <td width="25%"><b>Edit</td>
        <td width="25%"><b>Delete</td>
    </tr>
<?php
// connect...
$dbh = mysql_connect("localhost", "root", "");

// select the database...
mysql_select_db("phptest", $dbh);

// fetch all the users...
$results = mysql_query("SELECT * FROM users ORDER BY name",
    $dbh);
while($row = mysql_fetch_array($results))
{
    print "<tr>";
    print "<td>".htmlspecialchars($row["name"])."</td>";
    print "<td>".htmlspecialchars($row["age"])."</td>";
    print "<td>".htmlspecialchars($row["alias"])."</td>";
    print "<td>".htmlspecialchars($row["password"])."</td>";
    print "<td><input type=\"button\" value=\"Edit\"
        onClick=\"window.location = 'mysqlcore.php?cmd=
        adduser&id=".$row["id"]."\";\"></td>";
    print "<td><input type=\"button\" value=\"Delete\"
        onClick=\"window.location = 'mysqlcore.php?cmd=
        deleteuser&id=".$row["id"]."\";\"></td>";
    print "</tr>";
}

mysql_close($dbh);

?>
</table>
</center>
<?php
}

function addUser()
{
    $id = $_GET["id"];

    if($id != "")
    {
        $dbh = mysql_connect("localhost", "root", "");
        mysql_select_db("phptest", $dbh);

        $results = mysql_query("SELECT * FROM users WHERE id =
            '$id'", $dbh);
        if($row = mysql_fetch_array($results))
        {

```

```

        $name = $row["name"];
        $age = $row["age"];
        $alias = $row["alias"];
        $password = $row["password"];
    }

mysql_close($dbh);
}

?>
<center>
    <br><b>
    <?php
    if($id == "")
        print "Add User Form";
    else
        print "Edit User Form";
    ?>
</b><br><br>
<form method="POST" action="mysqlcore.php">
<input type="hidden" name="cmd" value="doadduser">
<input type="hidden" name="id" value="<?php print $id; ?>">
    <table width="450" border="1" cellpadding="4">
        <tr>
            <td>Name:</td>
            <td><input type="text" name="name" value=
                "<?php print $name; ?>"></td>
        </tr>
        <tr>
            <td>Age:</td>
            <td><input type="text" name="age" size=3 value=
                "<?php print $age; ?>"></td>
        </tr>
        <tr>
            <td>Alias:</td>
            <td><input type="text" name="alias" value=
                "<?php print $alias; ?>"></td>
        </tr>
        <tr>
            <td>Password:</td>
            <td><input type="password" name="password"
                value="<?php print $password; ?>"></td>
        </tr>
    </table>
    <br><br>
    <input type="submit" value="<?php
        if($id == "")
            print "Add User";
        else
            print "Save Changes";
    ?>">
    <input type="button" value="Cancel" onClick=
        "window.location = 'mysqlcore.php'">
</form>

```



```

    <?php
}

function doAddUser()
{
    $id = $_POST["id"];

    // connect...
    $dbh = mysql_connect("localhost", "root", "");

    // select the database...
    mysql_select_db("phptest", $dbh);

    // add the new user...
    if($id == "")
        mysql_query("INSERT INTO users VALUES (NULL,
            \"".$_POST["name"]."\", \"".$_POST["age"]."\",
            \"".$_POST["alias"]."\", \"".$_POST["password"]."\")", $dbh);
    else
    {
        // update the details...
        mysql_query("UPDATE users SET name = \"".$_POST["name"]."\", age
            = \"".$_POST["age"]."\", alias = \"".$_POST["alias"]."\",
            password = \"".$_POST["password"]."\", WHERE id = '$id'",
            $dbh);
    }

    // close the connection...
    mysql_close($dbh);

    // Redirect the page...
    header("Location: mysqlcore.php");
}

function deleteUser()
{
    $id = $_GET["id"];

    // connect...
    $dbh = mysql_connect("localhost", "root", "");

    // select the database...
    mysql_select_db("phptest", $dbh);

    // delete the user...
    mysql_query("DELETE FROM users WHERE id = '$id'", $dbh);

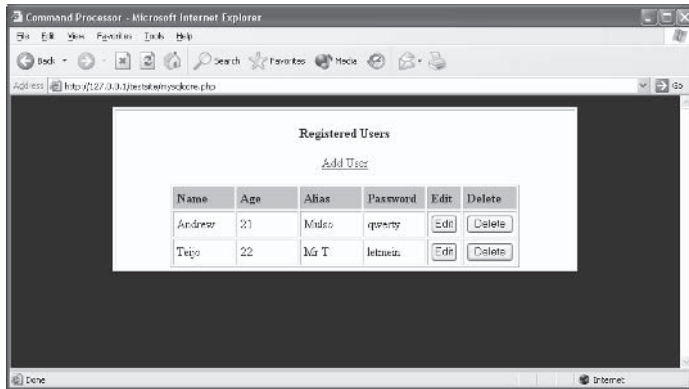
    // close the connection...
    mysql_close($dbh);

    // Redirect the page...
    header("Location: mysqlcore.php");
}
?>

```

When this is run in the browser, the main screen should look similar to the following screenshot:

Figure 3-22
Edit and Delete
buttons



As you can see from Figure 3-22, we have added two extra columns to the table to allow the editing and deletion of any of the users added to the system. When the user clicks the Edit button, an onlick JavaScript call to change the URL to the following occurs:

```
mysqlcore.php?cmd=adduser&id=[UID];
```

[UID] is replaced in the above line with the user's unique ID within the database. As you can see, however, we are sending the same adduser command as if the user clicked the Add User link at the top of the page. This is so we can reuse the same form we created for adding a user to allow the user to edit the stored information.

In the `addUser` function, we now first obtain the `id` variable passed in the URL and check whether it has been assigned a value using the following few lines of code:

```
$id = $_GET["id"];
```

```
if($id != "")
{
```

If an `id` has been passed in the URL, we then attempt to retrieve the user's details from the database using the following segment of code:

```
$dbh = mysql_connect("localhost", "root", "");
mysql_select_db("phptest", $dbh);
```

```
$results = mysql_query("SELECT * FROM users WHERE id = '$id'", $dbh);
if($row = mysql_fetch_array($results))
{
```

```
    $name = $row["name"];
    $age = $row["age"];
    $alias = $row["alias"];
    $password = $row["password"];
```

```
}  
  
mysql_close($dbh);
```

We can then print these values to the form (since if there was no data, the variables contain no data). For example, here is the name input box:

```
<input type="text" name="name" value="<?php print $name; ?>"
```

Note in the form that we also need to pass the `id` to the `doAddUser` function by including it as a hidden field in the form, as follows:

```
<input type="hidden" name="id" value="<?php print $id; ?>"
```

In the `doAddUser` function, we can once again check the value of the `id` variable (note that it comes through as a `POST` variable from the form, however). If the `id` variable is not assigned, we know it is a new user and add it to the database. If it is assigned a value, we simply create an `UPDATE` statement and amend the details stored in the database to the new values passed in from the form.

The other part of this example is where the user clicks the Delete button. When this happens, in the same way as the Edit button, we call the following URL:

```
mysqlcore.php?cmd=deleteuser&id=[UID]
```

Again, `[UID]` is substituted with the user's `id` field within the database. As can be seen from the URL, the `deleteuser` command is passed to `mysqlcore.php`. When this happens, it makes a call to the `deleteUser` function we have created, which then redirects back to the main screen after the SQL to remove the user has been executed.

Using FastTemplate

Now that we have looked at how to access MySQL from within PHP, you are probably looking at the code and thinking it is getting messy — and you would be correct.

The best way to keep our code tidy is to separate the logic from the design elements (i.e., the HTML). In this final section we are going to look at the `FastTemplate` class (<http://www.thewebmasters.net/php/FastTemplate.phtml> and also available on the CD) to see how it can solve our messy code problem.

The idea of `FastTemplate` is in its name — templates! Basically the plan is to create templates of what your pages, rows of tables, etc., should look like with placeholders for information to be filled in. The code will then assign values to these placeholders and eventually print it all to the screen.

Let's first look at a very simple example where we create a template file with two placeholders (defined by the curly brackets). Here is our template file now:

simple.tpl.html

```
<html>
  <title>{PAGE_TITLE}</title>
  <body>
    The name was specified as {NAME}
  </body>
</html>
```

This file should be saved in a folder called `tpl`, which should be a subfolder of the main testsite directory. If we open up this file in the browser directly (<http://127.0.0.1/testsite/tpl/simple.tpl.html>), it will look something like the following.

Figure 3-23
simple.tpl.html



As you can see, we have added the following two placeholders:

```
{PAGE_TITLE}
{NAME}
```

So we now need to write code that will use FastTemplate in conjunction with our `simple.tpl.html` file to generate the final page with the correct values. Let's look at this file now:

ft.php

```
<?php
include("class.FastTemplate.php");

// Create a FastTemplate object...
$tpl = new FastTemplate("./tpl");

// Define the template files...
$tpl->define(array(simple => "simple.tpl.html"));

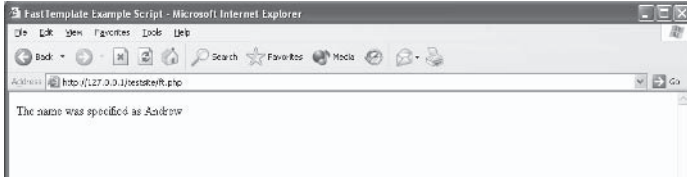
// Assign the placeholder values...
$tpl->assign("PAGE_TITLE", "FastTemplate Example Script");
$tpl->assign("NAME", "Andrew");

// Parse the template
$tpl->parse("FINAL", "simple");
```

```
// Display the final page
$tpl->FastPrint();
?>
```

When we load this into the browser (<http://127.0.0.1/testsite/ft.php>), we should see the following (note that we have moved the `class.FastTemplate.php` file from the CD into the `testsite` directory also):

Figure 3-24
ft.php



As you can see from Figure 3-24 (and if you run the script), `{PAGE_TITLE}` has been replaced with “FastTemplate Example Script” and `{NAME}` has been replaced with “Andrew.” Let’s now look line by line at how this works.

The first line simply includes the `FastTemplate` source code:

```
include("class.FastTemplate.php");
```

After this, an instance of the `FastTemplate` class is created by passing in the folder the template files is contained in as a parameter; the reference to this is then returned in the `$tpl` variable.

```
$tpl = new FastTemplate("./tpl");
```

Note that the folder can be easily changed to allow your web applications to be “skinned” with different designs without changing any of the code (apart from the folder the templates are contained in, of course).

After the `FastTemplate` instance is created, the next step is to define the actual template files we will be using within the project and assign them to keywords so that we can refer to them easily later in the code. This can be seen here:

```
$tpl->define(array(simple => "simple.tpl.html"));
```

As you can see, we have defined the keyword `simple` to refer to our `simple.tpl.html` file. Later in this section we will look at using multiple template files; however, one is sufficient for this example.

Now we can assign the values to our placeholders by making a call to the `assign` method of the `FastTemplate` class for each placeholder using the following two lines of code:

```
$tpl->assign("PAGE_TITLE", "FastTemplate Example Script");
$tpl->assign("NAME", "Andrew");
```

In the `assign` function, we first pass in the name of the placeholder we wish to assign a value to (without the curly brackets). The second parameter is simply the value we wish to assign it. Next, we call the `parse` function, passing in a placeholder to parse the data to, along with the keyword referring to the template file we wish to parse:

```
$tpl->parse("FINAL", "simple");
```

Note that the placeholder name, “FINAL,” is unimportant as this is the final parse (we’ll see in the next example why more than one parse would be required). The final line of code then prints out the last placeholder to be parsed to the browser, in this case the “FINAL” placeholder:

```
$tpl->FastPrint();
```

Multiple Templates

Now that we have looked at a simple example of using `FastTemplate`, let’s look at how multiple template files can be used to construct a small table. First, we are going to create a new template called `mainbody.tpl.html`, which will look as follows:

`mainbody.tpl.html`

```
<html>
  <title>{PAGE_TITLE}</title>
  <body>
    This text is above the table.
    <br><br>
    <table border=1 cellpadding=4>
      {TABLE_DATA}
    </table>
    <br><br>
    This text is below the table.
  </body>
</html>
```

As you can see, it is very similar to the simple template we created in the previous example; however, this time we have placed a table open tag, then a placeholder called `TABLE_DATA`, and a table close tag.

That defines our main page; now we need to define what a row in our table will look like. Assuming we would like to display a name, age, and location in our table, we can create a template called `tablerow.tpl.html` as follows:

tablerow.tpl.html

```
<tr>
    <td>{NAME}</td>
    <td>{AGE}</td>
    <td>{LOCATION}</td>
</tr>
```

Within our `tablerow` template, we have defined three placeholders to be replaced with the name, age, and location in each row of the table.

Let's now look at the final script that will create the table with three rows in it:

ft2.php

```
<?php
include("class.FastTemplate.php");

// Create a FastTemplate object...
$tpl = new FastTemplate("./tpl");

// Define the template files...
$tpl->define(array(mainbody => "mainbody.tpl.html",
                  tablerow => "tablerow.tpl.html"));

// Assign the placeholder values...
$tpl->assign("PAGE_TITLE", "FastTemplate Example Script");

$tpl->assign("NAME", "Andrew");
$tpl->assign("AGE", "21");
$tpl->assign("LOCATION", "Scotland");
$tpl->parse("TABLE_DATA", ".tablerow");

$tpl->assign("NAME", "Teijo");
$tpl->assign("AGE", "22");
$tpl->assign("LOCATION", "Finland");
$tpl->parse("TABLE_DATA", ".tablerow");

$tpl->assign("NAME", "Wes");
$tpl->assign("AGE", "unknown");
$tpl->assign("LOCATION", "USA");
$tpl->parse("TABLE_DATA", ".tablerow");

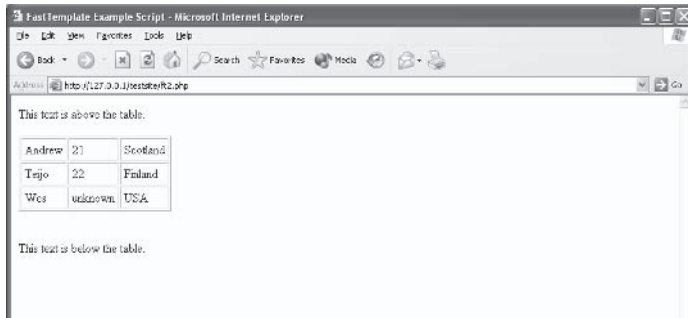
// Parse the template
$tpl->parse("FINAL", "mainbody");

// Display the final page
$tpl->FastPrint();

?>
```

When this `ft2.php` script is run in the browser, it will look like the following:

Figure 3-25
`ft2.php` output



This script is very similar to `ft.php`, but there are some interesting changes. The first is that we have defined an additional template file (and of course changed the simple one to refer to the new mainbody one), as shown below:

```
$tpl->define(array(mainbody => "mainbody.tpl.html",
                 tablerow => "tablerow.tpl.html"));
```

After this, the page title is assigned. Then we put the first row of the table in place by using the following four lines of code:

```
$tpl->assign("NAME", "Andrew");
$tpl->assign("AGE", "21");
$tpl->assign("LOCATION", "Scotland");
$tpl->parse("TABLE_DATA", ".tablerow");
```

Here we are assigning the `NAME`, `AGE`, and `LOCATION` placeholders, then parsing the `tablerow` template with these values into the `TABLE_DATA` placeholder within the `mainbody` template. Notice how we use the “.” before the `tablerow` keyword in the `parse` function. This simply means that we want the `tablerow` template to be appended to the `TABLE_DATA` keyword. (If we did not use the “.”, only the last row would be visible.)

Converting the Command Parser Example to FastTemplate

Here we are going to convert our slightly larger command parser example (where users can be added, edited, and deleted) to instead use `FastTemplate`.

The first step is to create the templates. In the command parser example, we have two main screens (the list of users and the Add User form page). The table that displays the list of users also needs to have a

template. We can also create a generic look-and-feel template instead of using the header and footer idea. So we will be creating the following list of templates:

Template	Purpose
main.tpl.html	Main look-and-feel (previously displayed with the <code>showHeader</code> and <code>showFooter</code> functions)
userlist.tpl.html	Contains the elements of the page that displays the list of users
userlist_row.tpl.html	Defines how a row in the user list should look
adduser.tpl.html	Contains the form required for the add user page

And here is what each of these template files should contain:

main.tpl.html

```
<html>
  <head>
    <title>Command Processor (using FastTemplate)</title>
  </head>
  <body bgcolor="#0000AA">
    <center>
      <table width="600" border=1 bgcolor="#FFFFFF">
        <tr>
          <td>
            {MAIN_CONTENT}
          </td>
        </tr>
      </table>
    </center>
  </body>
</html>
```

userlist.tpl.html

```
<center>
  <br><b>Registered Users</b><br><br>
  <a href="mysqlcore.php?cmd=adduser">Add User</a>
  <br><br>
  <table width="450" border="1" cellpadding="4">
    <tr bgcolor="#BBBBBB">
      <td width="25%"><b>Name</b></td>
      <td width="25%"><b>Age</b></td>
      <td width="25%"><b>Alias</b></td>
      <td width="25%"><b>Password</b></td>
      <td width="25%"><b>Edit</b></td>
      <td width="25%"><b>Delete</b></td>
    </tr>
    {TABLE_ROW}
  </table>
```

```

    </table>
</center>

```

userlist_row.tpl.html

```

<tr>
  <td width="25%">{USER_NAME}</td>
  <td width="25%">{USER_AGE}</td>
  <td width="25%">{USER_ALIAS}</td>
  <td width="25%">{USER_PASSWORD}</td>
  <td width="25%"><input type="button" value="Edit" onClick=
    "window.location = 'mysqlcore.php?cmd=adduser&id={USER_ID}'"></td>
  <td width="25%"><input type="button" value="Delete" onClick=
    "window.location = 'mysqlcore.php?cmd=deleteuser&id={USER_ID}'";></td>
</tr>

```

adduser.tpl.html

```

<center>
<br>
<b>{FORM_TYPE} User Form</b>
<br><br>
<form method="POST" action="mysqlcore.php">
  <input type="hidden" name="cmd" value="doadduser">
  <input type="hidden" name="id" value="{USER_ID}">
  <table width="450" border="1" cellpadding="4">
    <tr>
      <td>Name:</td>
      <td><input type="text" name="name" value=
        "{USER_NAME}"></td>
    </tr>
    <tr>
      <td>Age:</td>
      <td><input type="text" name="age" size=3 value=
        "{USER_AGE}"></td>
    </tr>
    <tr>
      <td>Alias:</td>
      <td><input type="text" name="alias" value=
        "{USER_ALIAS}"></td>
    </tr>
    <tr>
      <td>Password:</td>
      <td><input type="password" name="password" value=
        "{USER_PASSWORD}"></td>
    </tr>
  </table>
  <br><br>
  <input type="submit" value="{SUBMIT_TEXT}">
  <input type="button" value="Cancel" onClick="window.location =
    'mysqlcore.php'">
</form>

```

Next, we have edited `mysqlcore.php` and renamed it `ftcore.php`. Here is how it looks now:

ftcore.php

```
<?php

// include libraries here...
include("class.FastTemplate.php");

// Create a FastTemplate object...
$tpl = new FastTemplate("./tpl");

// Define the template files...
$tpl->define(array(main => "main.tpl.html",
                  userlist => "userlist.tpl.html",
                  userlist_row => "userlist_row.tpl.html",
                  adduser => "adduser.tpl.html"));

include("ftusers.php");

// special commands (no header/footer)
switch($cmd)
{
    case "doadduser":
        doAddUser();
        exit();
    case "deleteuser":
        deleteUser();
        exit();
}

// command processing...
switch($cmd)
{
    case "adduser":
        addUser();
        break;

    default:
        showUsers();
        break;
}

// Display the final page
$tpl->parse("FINAL", "main");
$tpl->FastPrint();

?>
```

And our new ftusers.php (formally users.php):

ftusers.php

```
<?php
function showUsers()
{
    global $tpl;

    // connect...
    $dbh = mysql_connect("localhost", "root", "");

    // select the database...
    mysql_select_db("phptest", $dbh);

    // fetch all the users...
    $results = mysql_query("SELECT * FROM users ORDER BY name", $dbh);
    while($row = mysql_fetch_array($results))
    {
        $tpl->assign("USER_ID", htmlspecialchars($row["id"]));
        $tpl->assign("USER_NAME", htmlspecialchars($row["name"]));
        $tpl->assign("USER_AGE", htmlspecialchars($row["age"]));
        $tpl->assign("USER_ALIAS", htmlspecialchars($row["alias"]));
        $tpl->assign("USER_PASSWORD", htmlspecialchars(
            ($row["password"])));
        $tpl->parse("TABLE_ROW", ".userlist_row");
    }

    mysql_close($dbh);

    $tpl->parse("MAIN_CONTENT", "userlist");
}

function addUser()
{
    $id = $_GET["id"];

    if($id != "")
    {
        $dbh = mysql_connect("localhost", "root", "");
        mysql_select_db("phptest", $dbh);

        $results = mysql_query("SELECT * FROM users WHERE id =
            '$id'", $dbh);
        if($row = mysql_fetch_array($results))
        {
            $tpl->assign("USER_ID", htmlspecialchars($row["id"]));
            $tpl->assign("USER_NAME", htmlspecialchars($row["name"]));
            $tpl->assign("USER_AGE", htmlspecialchars($row["age"]));
            $tpl->assign("USER_ALIAS", htmlspecialchars(
                ($row["alias"])));
            $tpl->assign("USER_PASSWORD", htmlspecialchars(
                ($row["password"])));
        }
    }
}
```

```

        mysql_close($dbh);

        $tpl->assign("FORM_TYPE", "Edit");
        $tpl->assign("SUBMIT_TEXT", "Save Changes");
    }
    else
    {
        $tpl->assign("USER_ID", "");
        $tpl->assign("USER_NAME", "");
        $tpl->assign("USER_AGE", "");
        $tpl->assign("USER_ALIAS", "");
        $tpl->assign("USER_PASSWORD", "");

        $tpl->assign("FORM_TYPE", "Add");
        $tpl->assign("SUBMIT_TEXT", "Add User");
    }

    $tpl->parse("MAIN_CONTENT", "adduser");
}

function doAddUser()
{
    $id = $_POST["id"];

    // connect...
    $dbh = mysql_connect("localhost", "root", "");

    // select the database...
    mysql_select_db("phptest", $dbh);

    // add the new user...
    if($id == "")
        mysql_query("INSERT INTO users VALUES (NULL,
            \"".$_POST["name"]."\", \"".$_POST["age"]."\",
            \"".$_POST["alias"]."\", \"".$_POST["password"]."
            \"\")", $dbh);
    else
    {
        // update the details...
        mysql_query("UPDATE users SET name = \"".$_POST["name"]."\", age
            = \"".$_POST["age"]."\", alias = \"".$_POST["alias"]."\",
            password = \"".$_POST["password"]."\" WHERE id = '$id'",
            $dbh);
    }

    // close the connection...
    mysql_close($dbh);

    // Redirect the page...
    header("Location: mysqlcore.php");
}

function deleteUser()
{

```

```
$id = $_GET["id"];

// connect...
$dbh = mysql_connect("localhost", "root", "");

// select the database...
mysql_select_db("phptest", $dbh);

// delete the user...
mysql_query("DELETE FROM users WHERE id = '$id'", $dbh);

// close the connection...
mysql_close($dbh);

// Redirect the page...
header("Location: mysqlcore.php");
}
?>
```

There is little need for a screenshot for this example as it looks exactly the same as it did before from the user's point of view. There is nothing new as far as FastTemplate goes in this final example; however, it should put you in the right direction for structuring your web-based interfaces.

Summary

In this chapter we learned how to set up an Apache server, install PHP4, and the basics of creating a web-based interface to a MySQL database. With this knowledge it is possible, with experimentation, to create complete administration tools for your online games so that they can be administered from a standard web site. Additionally, you can display player statistics and other such information on your site — or even go as far as creating an online community site for your game. In the next chapter, we move away from databases and web programming and proceed by looking into TCP/IP.

This page intentionally left blank.

Chapter 4

Introduction to TCP/IP

Introduction

Modern computer games take advantage of the Internet, the world's largest network, more and more every day. If we want to use the full potential of it in a game, we must understand how the Internet works. It doesn't help much if we can write flawless code but don't get the theory right. This chapter introduces the TCP/IP protocol and teaches how to write TCP/IP applications, or more accurately games, using the sockets application program interface (API).

What Is a Protocol?

To fully understand what TCP/IP is, we must understand what a network protocol is. Basically, a network protocol is the language that two or more computers use when they share information with each other. Naturally, all the computers must speak the same language to make the information flow between them. Imagine talking to someone who speaks a different language. You probably will not understand anything this person says to you. This would be the same as two computers using different protocols to communicate with each other. One computer simply will not understand what the other is trying to send to it. Hence, no information is shared between them.

A protocol defines the rules of communication and the format of the data. Protocols are standards that work on different kind of computers. This means that if two computers can be physically connected in a network, they can share information no matter what operating system they are running or even what type of computers they are — if they use the same protocol.

OSI Model

The International Organization for Standardization (ISO) developed a model for data communications in the late 1970s called the *Open Systems Interconnection* (OSI) model. This model is the standard for all data communications, and it also defines the basics of other standards. The model is based on seven layers, all of which have their own unique area of the data communication process. These layers are connected to each other one by one. Between two layers is an interface that these layers use to work with each other. OSI defines the layers' function and interface. It does not define how the layers are created. This makes it easy to create new data communication solutions based on the OSI model. But more importantly, it makes it easy to modify the existing ones. Because of this, we have the ability to choose from many different options to act on one layer. A good example is TCP and UDP, which are described later in this chapter. These two protocols belong to the transport layer, and we can choose which one to use in our application. We don't have to worry about anything extra, because the interface between the transport layer and the two other layers connected to it makes sure the data flows correctly. Figure 4-1 shows the OSI model layers and their order.

OSI Model Layers

Figure 4-1
OSI model
layers

7	application	7
6	presentation	6
5	session	5
4	transport	4
3	network	3
2	data link	2
1	physical	1

- **Layer 1: Physical:** The first layer takes care of all the physical data transfers. It includes all the physical electronic devices such as circuits, cables, and connectors. Basically it defines the medium to be used in the data transfer.
- **Layer 2: Data Link:** The second layer consists of our computer's network interface card and the device driver. The data is put into frames that are checked for errors and fixed if any errors exist. As game developers we should not be too concerned about this layer, because if the hardware can do normal networking, we have no problems.
- **Layer 3: Network:** The third layer is taken care of by the Internet Protocol (IP). It addresses the packets and frames and routes them to the correct address over subnets. More about IP later in this chapter.
- **Layer 4: Transport:** Layer four defines the method of data transfer. Depending on the method, the data is checked for errors, big packets are repackaged into smaller packets, and care is taken that the data is sent and received correctly. These features vary from method to method. This layer is the most interesting layer for the network programmer, as TCP and UDP, which are described later in this chapter, belong to this layer.
- **Layer 5: Session:** The fifth layer is fairly simple. It defines how two computers establish and end a session. It also takes care of the session's synchronization, and defines when a computer can transfer data and when it can receive data.
- **Layer 6: Presentation:** The sixth layer takes care of possibly compressing and/or encrypting data. It defines what the data looks like when it is being transferred.
- **Layer 7: Application:** The seventh layer defines the network application, such as file transfers or e-mail.

A good example of the usage of the OSI model is a normal phone conversation. The telecommunication companies provide us layer 1 — the cables and connectors through which the conversation is transferred from one place to another. Other companies take care of layer 2 — they build phones for us. Telephone companies' switches belong to layer 3. They direct the call to the correct place. Our phones use the method of transfer used by the switches and other phones to make our voice move from one place to another. This is layer 4. We are on layer 5 when we dial the number to call and the other end's phone rings. Modern phones modulate our speech using a method called pulse code modulation (PCM), which belongs to layer 6. Now we have a complete phone conversation (layer 7).

Internet Protocol

The protocol used in Internet communications is called TCP/IP. As the name implies, it is divided into two layers. These layers are part of the *Open Systems Interconnection* (OSI) model. See the section titled “OSI Model” for a description of the OSI model. *Transmission Control Protocol* (TCP) is part of the transport layer of the OSI model, and *Internet Protocol* (IP) is part of the network layer. TCP/IP is a protocol suite that has more members than the name suggests. One important member is another protocol from the transport layer — *User Datagram Protocol* (UDP). Both TCP and UDP are connected to the IP protocol.

Because the OSI model is based on layers, it is easy to develop better and better network solutions by just replacing one layer with a new, better one. We can change our *network interface card* (NIC), drivers, or network connection type, or we can develop a better protocol. All of these are happening all the time all around the world. Companies develop new NICs and people buy them. The same companies develop new drivers for their cards and people install them on their computers. For some people, this is not enough — they want better Internet connections. Also, the Internet Protocol is improving. The current version is IPv4, but the next public version is IPv6.

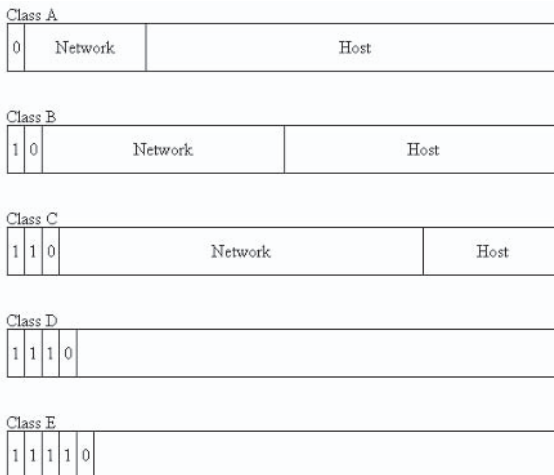
Requirements of the modern computer culture grow every day. As computers get faster and faster, people are buying more and more computers. But the reason people buy computers is not because they are fast. The reason is the Internet. Imagine what an average computer user did with his or her computer in the 1970s or 1980s. Word processing was likely one of the most common reasons for buying a computer then. Back then, average users had not even heard of the Internet. But even if they had, they didn’t dream of ever using it at home. Nowadays people need computers for everything. Or at least they think they do. Getting an Internet connection is very easy now and even fairly cheap, so there really is no point in not getting one. This has led to problems. The number of free IP addresses is running out quickly as companies reserve addresses for themselves and individuals reserve their own IPs — not necessarily for all the time, but at least for the time they are online.

Demand for a good network system for universities was increasing in the U.S. in the 1960s. This led to the development of IPv4 during the 1970s. Its addresses consist of four decimal dotted bytes, for example, 192.168.0.1. But because the human mind is better with names than numbers, some IP addresses are given names, for example, www.huntedcow.com. A *Domain Name Service* (DNS) then looks up the name and the corresponding IP address number.

The addresses are divided into subnets — class A, class B, and class C. Class A means that the first byte of the address is used to define the subnet. The very first bit is set to 0 to identify that it is a class A address. This bit is taken from the network byte, so only the remaining seven bits are used for identifying the network. The last three bytes are used to define the address itself. In class B, the first two bytes define the subnet and the other two the address. The first two bits of a class B address are 1 and 0, leaving 14 bits for the network part. In class C, the first three bytes define the subnet and the last byte the address. The first three bits of class C addresses are 1, 1, and 0, and the next 21 bits are used for the network part. There are about 16,000,000 class A IP addresses, all of which are in use already. Class B consists of about 65,000 addresses and class C has 254 addresses. The whole address space of IPv4 is about 4 billion addresses. While 4 billion is a big number, it is not big enough for IP addresses in the future.

Class D addresses are multicast addresses. The first four bits are the identifier bits. On a class D address they are 1, 1, 1, and 0. Multicast defines a group of IP addresses. So when you send something to a multicast address, you are not sending to a single computer but a group of computers. Class E addresses are reserved for future use. Figure 4-2 shows the different address classes and how their data space is divided.

Figure 4-2
Address
classes



The solution for the problem is IPv6. Among other improvements, it provides a 128-bit addressing system that increases the address space so much that we would need to colonize new worlds to use them all. At least that's the way it looks like right now. Only time will tell if it is true.

IPv6 was developed in the 1990s and will most likely go into use during the next decade. Because of this, it is a good idea to prepare for

it now. There's no point in making a network application that may not work tomorrow. Of course, IPv6 is compatible with IPv4 to a certain point, but to be absolutely sure we should develop our applications to be protocol independent from the beginning. This book teaches you how to do so.

Introduction to the Transport Layer

As we have already learned, the TCP/IP protocol suite consists of two layers of the OSI model — network and transport. The transport layer is as important to TCP/IP as the network layer. For the network programmer, it is the most interesting part. When we write network applications, we write commands for the transport layer.

In Internet communications using TCP/IP, the transport layer is either Transmission Control Protocol (TCP) or User Datagram Protocol (UDP). Sometimes people just say TCP when they really mean TCP/IP, so that may be confusing if you do not know the context.

NOTE We can also use raw sockets to get direct connection in our code to the Internet Protocol, bypassing TCP and UDP. However, this is beyond the scope of this book.

Transmission Control Protocol

Transmission Control Protocol is a connection-oriented protocol. This means that every time we want to communicate with a remote host, we must first establish a connection. Once we have established a connection, we do not have to worry about directing the messages we send to the correct place. When we are done with the connection, we must close it. TCP is also a reliable protocol. It makes sure the other end receives the messages we send, and it handles such things as duplicated packets.

Sometimes when a packet is sent to the network, it may get lost for a while. This may happen, for example, if a router is experiencing some problems. The packet gets stuck on the router and so the receiving host never sends a confirmation that the packet has reached its destination. The sending host assumes that the packet is lost forever and retransmits it after a certain time. This new packet may find another route to the receiving host, while the original packet is still stuck on the router. If the router starts working normally again, it routes the packet to the correct place and the receiving host may receive the same packet again. TCP notices the duplicate packet and destroys it, as it is not needed anymore.

User Datagram Protocol

Unlike TCP, User Datagram Protocol is a connectionless protocol. No actual connection is established between the two communicating hosts. This means that you cannot say when a connection is open or closed; you just communicate with the other host by sending it data and waiting for data back. The sending host must define the target address every time it sends something. When a host receives a message, it knows where the message came from, and therefore it can transmit something back to the remote host. UDP is not reliable, as it does not ensure that the transmission is received. It also does not take care of duplicates. However, duplicates are not that common with UDP because the sending host won't retransmit anything if the message is not received — simply because it does not care. UDP can be used in applications that need the best possible efficiency but very little reliability. Computer games fall into this category, and that is why UDP is so important to us.

UDP can be made reliable, but this requires that we write the needed checking algorithms ourselves. Some checking is always good, but no one forces us to make any.

Ports

Taking into account that most network application servers today are very simple, it would be foolish to make a network server run only one service. A game server may require all of the server's resources, so there are exceptions too. But how can we identify the service we want to use if the server is running dozens of services? We cannot connect every service one by one and check to see if it is the one we want, as this may take a long time. Additionally, some services may seem like the one we are looking for but are not. Therefore, we need to give each service a number that we define when we are connecting to it. These numbers are called *port numbers*.

The port number is a 16-bit value, so there are 65,535 possible ports available (there is no port 0 — therefore, $2^{16} - 1$). Actually, available ports are not that straightforward. Ports from 1 to 1023 are so-called well-known ports, and thus we cannot use them for our servers. These ports are reserved for common network services such as FTP (port 21) and daytime server (port 13). Ports from 1024 to 65535 are generally free to be used, but it is a good idea to check that the port we are about pick is not used by any other known service. The Internet Assigned Numbers Authority (IANA) records used ports if they are well known enough.

TCP and UDP ports are unique. For example, TCP port 1024 is not the same as UDP port 1024. Usually, if a port number is registered by the IANA, it is registered for both protocols at the same time, even if the application does not use both protocols at that time.

We need to define the port number only for the server. Clients usually use so-called *ephemeral ports*, which we have no control over. The kernel of the operating system chooses them for us. The range of ephemeral ports varies from platform to platform, as shown in Figure 4-3. We can also assign an ephemeral port for a server, but only in a special case, which is beyond the scope of this book.

Figure 4-3

	well-known ports	registered ports	dynamic / private ports
IANA	1 - 1023	1024 - 49151	49152 - 65535
		BSD ephemeral ports	Solaris ephemeral ports
		1024 - 5000	32768 - 65535

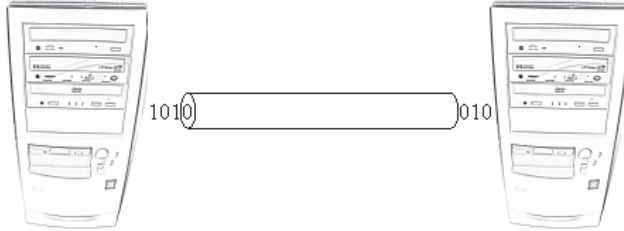
Not all ports are always open to use, even if they are not well-known ports. Firewalls block most ports on servers to keep unwanted people out of the server, so the server's administrator must open a port on the firewall before it can be used. There are different kinds of firewalls available, so consult your server's administrator about the ports.

Introduction to Sockets

Sockets API is a programming interface that we use to write network applications. It is a multiplatform API, so we can make different operating systems communicate with each other using sockets. This book teaches you how to make your network code work on multiple platforms without any changes to the code.

There is no clear definition for a socket. Different people have different opinions about them, but basically a socket is a pipe between two computers on a network through which data flows. It is not the physical cable or anything concrete like that. A socket exists only in the world of bits and bytes. The two computers each have their own unique socket, and these sockets can be identified as the two ends of the pipe. Request for Comment 147 (RFC147) defines what a socket is for the ARPA network. As the ARPA network is the predecessor of the Internet, this documentation applies to Internet sockets as well.

Figure 4-4



In computer memory, sockets are 32-bit numbers. So when a new socket is created, it is given a unique number that defines the socket on the local computer.

Socket Types

There are two kinds of sockets available: stream sockets and datagram sockets. A stream socket is a connection-oriented socket; thus, a connection has to be established before it can send or receive data. Stream sockets use the TCP transport protocol. A datagram socket is a connectionless socket. Datagram sockets use the UDP transport protocol.

You may think that a connection must be established before usage. This is not as straightforward as it sounds, though. We can think of this in terms of multiple levels of connections. The last level is when we have established the connection between two stream sockets. Before that there is a level where there is no real connection, but the two hosts know of each other and their addresses. They can sort of throw things at each other and hope that they reach their destination. With an established connection, however, the two hosts could throw the things into a pipe (the connection between the two hosts), and the probability of receiving the data would be much higher than without a connection.

In a stream socket, the data flows constantly in the socket, as the name implies. It works like a stream of water, except that it moves in both directions. Stream sockets are reliable in many ways. Both sides know if the other side disconnects or crashes and cannot receive anything anymore. Every packet is monitored to see if it reaches its destination. If a packet has not reached its destination in a certain time, it will be retransmitted.

In a datagram socket, the data is transmitted in datagrams. This means that whenever there is something to send, the data is sent to the address defined. When there is nothing to send, no data flows between the two hosts. If one side crashes, the other side will not notice unless a system has been built that checks if the other side is alive. Datagram sockets are not reliable. The packets are not monitored by TCP/IP to

see if they reach their destination. If we want, we can create our own monitoring system, but if we use UDP for the monitoring system as well, the monitoring system is also not reliable. As this may get very confusing, some people use only stream sockets.

Address

Each socket has its own address information. We can create a socket on any port as long as the port is free and it is okay to choose that port. We can also define the IP address to connect to for every socket. Usually, a single socket is not enough on a server application. We need one socket to listen to the incoming clients and another socket to handle the client/server communication. At the same time, the listening socket is still listening for clients and creates a new socket for all other clients as well. This type of server is called a *concurrent server*. If the server uses only one socket to do all the communication, it can communicate with only one client at a time. This type of server is called an *iterative server*. When creating computer games, we need to process multiple clients at once, so a concurrent server is the only way to do it. This book covers both iterative and concurrent servers.

Platforms

The sockets API works on many platforms, and thus we should know how to take advantage of it. This section covers Unix (Linux) and Windows versions of the API; once we are done, we will know how to write code that works on both platforms.

The sockets API was originally developed for the Unix operating system. The 4.2BSD (Berkeley Software Distribution) system had the first version of sockets in 1983. At the same time, TCP/IP was released widely to the public for the first time. The API has developed from these beginnings. Many Unix platforms use the same networking code as BSD does, but others, for example Linux, do not. Linux's network code has been written from scratch, but this does not mean these different implementations would not be compatible. Linux is a free, very popular Unix-based operating system. All Unix code in this book has been developed and tested on a Linux system.

Many non-Unix operating systems have the sockets API as well. For example, Microsoft Windows has its own sockets library called WinSock (Windows Sockets). WinSock is compatible with BSD sockets, but it also has many features that the BSD version does not have. Because of operating system differences, there are some noticeable differences in the APIs, but they do not interfere with compatibility when connecting two computers. The latest version of WinSock is WinSock2.

WinSock2 software development kit (SDK) is available on the Internet. This SDK is required for writing applications that use WinSock.

History of WinSock

The Windows sockets API was born October 10, 1991, at Interop '91 in San Jose, California. A committee was established to design a specification for a sockets library for the Windows operating system from a proposal by Martin Hall of JSB Corporation. There have been over 40 companies involved in the design of WinSock. WinSock is not the property of Microsoft, although it is an important part of Windows nowadays. It was developed by independent sources who were interested in taking part of this project. On January 20, 1993, the specification for WinSock 1.1 was published. This version had support for TCP/IP only.

WinSock2 was published in 1996. It provided support for multiple transport protocols, such as Novell IPX/SPX and Digital's DECNet, and officially supported the OSI model. Version 2 also includes features like multicasting and Quality of Service (QoS), both of which are explained later in this book.

WinSock is currently compatible with version 4.3 of the Berkeley Software Distribution sockets.

Summary

Because we are writing games, we consider WinSock essential. The fact is that Windows is the operating system for which most games nowadays are written, and we make no exception here. But why do we need Unix code too? We could write the game for Unix platforms, but the main reason is the server, as most network games use Unix as the platform for their servers. There are many reasons for this, but the two most important reasons are the great stability of all Unix systems and easy remote controlling. Every Unix system can be controlled remotely, so we do not actually have to sit down in front of the server every time we want to control the server. Using Windows 9x as the server platform is out of the question because of certain restrictions. Windows NT and Windows 2000 are also good options for the server platform, as they do not have the same restrictions as Windows 9x systems because they were designed for different kinds of usage. In the end, it is up to you which operating system you wish to use. But remember, the client and the server can be run on different platforms.

This page intentionally left blank.

Chapter 5

Basic Sockets Programming

Introduction

The Unix sockets API does not require any extra initialization before we can actually use the sockets. However, this is not the case for Windows sockets. What we need to initialize is the WinSock 2 dynamic-link library (DLL) and then enumerate the available protocols. As this book focuses only on the TCP/IP protocol, we only look for TCP and UDP protocols. In this chapter, we learn the basic initialization functions of WinSock. Then we see how to actually use them by creating an initialization function that does all the necessary initialization.

WinSock Initialization

This section explains the basics of WinSock initialization.

WSAStartup Function (Win32)

```
int WSAStartup(WORD wVersionRequested, LPWSADATA lpWSADATA);
```

The `WSAStartup` function is used to initialize the WinSock API. As the first parameter, we must provide the version number we request. It is a `word` value, and so we must fill the variable using the `MAKEWORD` macro. `MAKEWORD` creates an unsigned 16-bit integer from the two

unsigned characters we give it. The second parameter is a pointer used to get the data from the WinSock DLL. The data is stored in a `WSADATA` data type.

The function returns 0 if everything went fine. If not, it returns one of the non-zero values listed in Table 5-1.

Table 5-1: `WSAStartup` function return values

<code>WSASYSNOTREADY</code>	The network subsystem is not ready.
<code>WSAVERNOTSUPPORTED</code>	The version we requested is not supported.
<code>WSAEINPROGRESS</code>	A blocking WinSock 1.1 operation is in progress, or the service provider's callback function is in progress.
<code>WSAEPROCLIM</code>	The limit of WinSock tasks has been reached.
<code>WSAEFAULT</code>	<code>WSADat</code> a is not a valid pointer.

WSACleanup Function (Win32)

```
int WSACleanup(void);
```

We use `WSACleanup` to uninitialized the WinSock API. This function will unregister the WinSock DLL used by our application. Windows keeps a record of all the DLLs used by applications and updates the reference count of each DLL. If the count is higher than zero, Windows knows that this DLL is used by an application and therefore it is kept open in the system memory. If the reference count is zero, the DLL is not open.

It is our responsibility as programmers to keep Windows aware of all the registered DLLs. In other words, we must call `WSACleanup` at the end of every WinSock application if the WinSock API has been initialized.

The return value is 0 if the operation was successful. On error, the return value is `SOCKET_ERROR`, but we can get a more accurate error value by using `WSAGetLastError`. Table 5-2 lists the return values of the `WSACleanup` function.

Table 5-2: `WSACleanup` function return values

<code>WSANOTINITIALIZED</code>	WinSock API is not initialized.
<code>WSAENETDOWN</code>	A network subsystem error occurred.
<code>WSAEINPROGRESS</code>	A blocking WinSock 1.1 operation is in progress, or the service provider's callback function is in progress.

WSAEnumProtocols Function (Win32)

```
int WSAEnumProtocols(LPINT lpiProtocols, LPWSAPROTOCOL_INFO lpProtocolBuffer,
                    LPDWORD lpdwBufferLength);
```

This function is used to enumerate the available protocols on the local computer. The first parameter is a pointer to a list of protocols we wish to look for. We must create an integer list of the protocols. In this book we use only TCP and UDP, so we put only these two in the list:

- IPPROTO_TCP — Transmission Control Protocol
- IPPROTO_UDP — User Datagram Protocol

The second parameter is a pointer to a `WSAPROTOCOL_INFO` buffer that is filled when the function is run.

The third parameter is a value-result parameter. It is used to tell the function how big a buffer we need for the protocols, but the function may change it if it is not big enough. This is important, because when we start enumerating the protocols, we actually run this function twice. The first time, we do not fill the first two parameters because all we need to do is get the buffer size. We provide a pointer to a zero-size buffer so the function will increase the size for us. The next time we run the function, we give it all the needed parameter info, including the buffer size we got from the first call to `WSAEnumProtocols`.

WinSock Initialization Function

The following code shows us how to initialize WinSock API. It is a function from the network library we will create later. But as this function is essential for getting WinSock ready to be used, we introduce it here. After the code listing, we will go through the important parts to see what is really happening.

```
int dreamSock_InitializeWinSock(void)
{
    WORD versionRequested;
    WSADATA wsaData;
    DWORD bufferSize = 0;

    LPWSAPROTOCOL_INFO SelectedProtocol;
    int NumProtocols;

    // Start WinSock2. If it fails, we do not need to call WSACleanup()
    versionRequested = MAKEWORD(2, 0);
    int error = WSASStartup(versionRequested, &wsaData);

    if(error)
    {
        return 1;
    }
}
```

```
}
else
{
    // Confirm that the WinSock2 DLL supports the exact version
    // we want. If not, call WSACleanup().
    if(LOBYTE(wsaData.wVersion) != 2 || HIBYTE(wsaData.wVersion)
        != 0)
    {
        WSACleanup();
        return 1;
    }
}

// Call WSAEnumProtocols to figure out how big of a buffer we need
NumProtocols = WSAEnumProtocols(NULL, NULL, &bufferSize);

if((NumProtocols != SOCKET_ERROR) && (WSAGetLastError() !=
    WSAENOBUFS))
{
    WSACleanup();
    return 1;
}

// Allocate a buffer; call WSAEnumProtocols to get an array of
// WSAPROTOCOL_INFO structs
SelectedProtocol = (LPWSAPROTOCOL_INFO) malloc(bufferSize);

if(SelectedProtocol == NULL)
{
    WSACleanup();
    return 1;
}

// Allocate memory for protocol list and define what protocols to
// look for
int *protos = (int *) calloc(2, sizeof(int));

protos[0] = IPPROTO_TCP;
protos[1] = IPPROTO_UDP;

NumProtocols = WSAEnumProtocols(protos, SelectedProtocol, &bufferSize);

free(protos);
protos = NULL;

free(SelectedProtocol);
SelectedProtocol = NULL;

if(NumProtocols == SOCKET_ERROR)
{
    WSACleanup();
    return 1;
}
```

```
        return 0;
    }
```

First, we use the `MAKEDWORD` macro to fill the `WORD versionRequested` with the information we want. In this case, we want to check that the version number of WinSock DLL is 2.0, so we fill the word with bytes representing 2 and 0. Then we run `WSAStartup` to start initializing the WinSock API.

```
versionRequested = MAKEDWORD(2, 0);
int error = WSAStartup(versionRequested, &wsaData);
```

The function fills `wsaData` for us. We check the `wsaData's wVersion` member (`WORD`) to see what version the DLL supports. We use the `LOBYTE` and `HIBYTE` macros to check the two bytes of the `WORD`. If the bytes do not match the version number we want, we clean up WinSock and return 1 to indicate there was an error.

```
if(LOBYTE(wsaData.wVersion) != 2 || HIBYTE(wsaData.wVersion) != 0)
{
    WSACleanup();
    return 1;
}
```

Next, we need to find out how big a buffer we need for the protocols. We use `WSAEnumProtocols` for this. During the initialization process, we run this function twice. The first time we run it, we do not provide any parameters for it other than the buffer size parameter. The function will increase the buffer size for us if it is too small. We start with a zero size buffer, so the first time we run the function it will definitely fail. We want this function to fail for now. This means that if the function does not return `SOCKET_ERROR`, something is wrong, probably because the buffer size is too small if it is 0.

Then we check that the last error occurred because the buffer was too small. We do this by using the `WSAGetLastError` function. If the buffer was too small, the last error message was `WSAENOBUFS`. If it was something else, there is something else wrong and we should quit the initialization process.

```
NumProtocols = WSAEnumProtocols(NULL, NULL, &bufferSize);

if((NumProtocols != SOCKET_ERROR) && (WSAGetLastError() != WSAENOBUFS))
{
    WSACleanup();
    return 1;
}
```

Here is how we define the protocols we are looking for. First we allocate memory for the protocol list quite normally, and then simply fill the list with protocols we want. Because we only want the two TCP/IP

protocols, that is what we put in the list. The order of the list members is not important.

```
int *protos = (int *) calloc(2, sizeof(int));
protos[0] = IPPROTO_TCP;
protos[1] = IPPROTO_UDP;
```

Finally, we get to enumerate the actual protocols. We now have all the information we need for the second call to `WSAEnumProtocols`. For the first parameter we provide the protocol list; for the second, the protocol info structure pointer; and for the last, the buffer size. The return value is the number of protocols it found. The return value is `SOCKET_ERROR` if something is wrong. We do not need to check what actually is wrong. It is enough to know that something did go wrong, and we stop initializing if this happens.

```
NumProtocols = WSAEnumProtocols(protos, SelectedProtocol, &bufferSize);
```

NOTE `WSAEnumProtocols` looks for protocols installed on your Windows operating system. If you do not have TCP/IP protocols installed on your operating system, `WSAEnumProtocols` will not find them.

Error Handling

There is one WinSock function in the previous code that we have not discussed yet. This is the `WSAGetLastError` function, which is a WinSock-only function.

WSAGetLastError Function (Win32)

```
int WSAGetLastError(void);
```

This function retrieves the value of the last error that occurred in the last Windows sockets operation function. Some functions only indicate that an error occurred but do not give the actual error value. This function is used to get the error value.

Unix does not have a function to receive the last error that occurred. It does, however, have the global variable `errno` that works just like the return value of `WSAGetLastError`, but you do not have to fetch it by running a function. The error values on different platforms may have equal integer values (they also may differ), but their constant names usually are not the same. For example, WinSock error values have the prefix “WSA.”

Sockets Data Types

Sockets have more than one address data type because different protocols require unique information. We can still write protocol-independent code thanks to the generic address data types. These data types are explained here.

Platform-specific Data Types

As we will be writing our code for multiple platforms, we must be aware of different data types that correspond to the same thing on different platforms. We can easily define new data types and rename the required data types to match the other platform's types. This can be done because usually only the data type name is different; the actual data is the same. Table 5-3 shows two data types that have different names on Unix and Win32 but are really the same thing.

Table 5-3: Equivalent data types

Data Type	Unix	Win32
Address length data type	socklen_t	int
Socket descriptor data type	int	SOCKET

From now on we use the more informative data type name in the text and all source code examples. For socket address length, we use `socklen_t` instead of `int`, and for socket descriptors, we use `SOCKET` instead of `int`.

We learn how to create these new definitions for our data types in Tutorial 2, “Creating Your Network Library.”

Address Structures

The address information of the sockets is stored in structures. Naturally, IPv4 and IPv6 have their own structures.

IPv4 Address Structure

The following code shows the structure for IPv4:

```
struct in_addr
{
    in_addr_t s_addr;    // 32-bit IP address
};
```

```

struct sockaddr_in
{
    uint8_t sin_len;           // structure length: 16 bytes
    sa_family_t sin_family;   // protocol family: AF_INET
    in_port_t sin_port;       // 16-bit port number
    struct in_addr sin_addr;   // 32-bit IP address
    char sin_zero[8];         // Not used
};

```

Let's take a look at the structure members. The structure length variable is handled by the kernel, so we do not need to worry about it. The protocol family for IPv4 addresses is always `AF_INET`. The port number is stored in a 16-bit unsigned integer in network byte order. The IP address is stored in a 32-bit unsigned integer, also in network byte order. It is not stored in the normal dotted decimal format. We discuss how to get the dotted decimal format out of the 32-bit unsigned integer later.

IPv6 Address Structure

This code shows the IPv6 address structure:

```

struct in6_addr
{
    uint8_t s6_addr[16];      // 128-bit IP address
};

#define SIN6_LEN              // required for the compiler

struct sockaddr_in6
{
    uint8_t sin6_len;         // structure length: 24 bytes
    sa_family_t sin6_family; // protocol family: AF_INET6
    in_port_t sin6_port;      // 16-bit port number
    uint32_t sin6_flowinfo;   // 32-bit flow label and priority
    struct in6_addr sin6_addr; // 128-bit IP address
};

```

`SIN6_LEN` must be defined if the length member is supported by the system. The structure length of an IPv6 address is 24 bytes, but the kernel takes care of this. The protocol family for IPv6 addresses is `AF_INET6`. The port number is a 16-bit unsigned integer, just like the IPv4 ports. The IPv6 address structure has a member that is not included in the IPv4 address structure. It stores the flow label and priority values. The first 24 bits are used for the flow label, the next 4 bits are for the priority, and the remaining 4 bits are reserved for future use. The IP address is stored in 16 8-bit unsigned integers.

Generic Address Structure

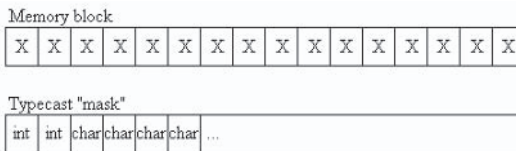
Having more than one protocol family that can be used creates a small problem. Because IPv4 and IPv6 have their own address structures, we cannot pass either of the structures as a pointer to any socket functions that use the address structure as a parameter, and still remain protocol independent. That is why we have a generic address structure, which is shown below.

```
struct sockaddr
{
    uint8_t sa_len;           // structure length
    sa_family_t sa_family;    // protocol family
    char sa_data[14];        // the address (either IPv4 or IPv6)
};
```

By typecasting the protocol-specific address structures into this generic structure, we can use any version of address structures in any of the socket functions. Of course, the functions must have this generic structure as the parameter instead of the protocol-specific ones.

Typecasting means that you are providing a mask for your pointer in the memory. This mask is used to divide the block of memory into the variables that are stored there. There is no need to typecast a pointer if the data type we want to use is the original data type. The beginning of the data types must match the generic one. But as we see in the protocol-specific address structures and the generic address structure, only the first two members of the structures match. The third member in the generic structure is chars only, and these chars are used to store the actual address information. The function itself must understand this. Figure 5-1 shows an example of how the typecast “mask” works. In this example, the first two members of the cast structure are 8-bit integers, or shorts (one X in the memory block means 8 bits), and after that there are only characters that hold the data in a format that can be transformed into any data type.

Figure 5-1
Typecasting



TIP As you have probably already noticed, typecasting is a useful way to make code work with various data types. It is a good idea to take advantage of it whenever possible.

Basic Sockets Functions

There are some functions in the sockets API that nearly all sockets applications use at some point. These functions are explained in this section.

socket Function (Unix, Win32)

```
SOCKET socket(int family, int type, int protocol);
```

This function creates a socket with the provided information. This function only creates the descriptor of the socket; it does not really start using any port or any IP address yet. If you are familiar with Unix programming, you may notice that the socket descriptors are just like file descriptors on Unix.

The first parameter (`int family`) specifies the protocol family. It can be one of these:

- `AF_INET`: IPv4 protocols
- `AF_INET6`: IPv6 protocols
- `AF_ROUTE`: Routing sockets

There are more options for this parameter on different platforms. We cover only the first two, `AF_INET` and `AF_INET6`, in this book.

The second parameter (`int type`) defines the socket type, which can be one of the following:

- `SOCK_STREAM`: Stream socket
- `SOCK_DGRAM`: Datagram socket
- `SOCK_RAW`: Raw socket

This book covers only `SOCK_STREAM` and `SOCK_DGRAM`. `SOCK_STREAM` is used for TCP sockets and `SOCK_DGRAM` for UDP sockets.

The third parameter (`int protocol`) is set to 0 when using either stream or datagram sockets.

Win32 return values:

- Success: A nonnegative descriptor (integer)
- Failure: `INVALID_SOCKET`

Unix return values:

- Success: A nonnegative descriptor (integer)
- Failure: `-1`

bind Function (Unix, Win32)

```
int bind(SOCKET s, const struct sockaddr *addr, socklen_t addrlen);
```

The `bind` function makes your socket have its own address information, i.e., the IP address and the port number. The function binds the local address information to each socket. You cannot define a non-local IP address when calling `bind`, but you can define any port number (keeping in mind that not all ports are available ports). This function is usually used right after the call to the `socket` function.

To listen to incoming events on a certain port, you must bind the port to the socket first. You cannot listen to incoming events if you have not bound the socket to the port to be listened to. Both stream and datagram sockets must be bound to an address before they can be used. You do not have to bind a client's local sockets, but it is okay to do so.

The first parameter (`SOCKET s`) defines the unbound socket to be bound. The socket must have been created with the `socket` function. The second parameter (`const struct sockaddr *addr`) is a pointer to a `sockaddr` address information structure. The third parameter (`socklen_t addrlen`) is the size of the address information structure.

The address you are about to bind may already be in use by another application or by your own application. If this is the case, `bind` returns the error value `EADDRINUSE` (Unix) or `WSEADDRINUSE` (Windows) (the value must be retrieved by using the `WSAGetLastError` function in Windows and the `errno` variable in Unix). With certain socket options we can still bind the address, even if it is already in use.

If you set the port number to 0, the operating system's kernel will choose an ephemeral port. Usually this is the first free ephemeral port. Client applications use ephemeral ports because there really is no need for us to know the port the client uses. The remote host the client is sending data to can figure out the port itself.

If we have multiple network interface cards on our local host, we can either choose the IP address for the socket ourselves or we can make the kernel choose it for us. To do this, we use the constant `INADDR_ANY`.

NOTE A single host can have multiple IP addresses if it has multiple network interface cards. Usually only server machines have this kind of arrangement.

If an error is encountered with the `bind` function, we must use the `WSAGetLastError` function in Windows or the `errno` variable in Unix to get the actual error value.

Win32 return values:

- Success: 0
- Failure: `SOCKET_ERROR`

Unix return values:

- Success: 0
- Failure: `-1`

connect Function (Unix, Win32)

```
int connect(SOCKET s, const struct sockaddr *addr, socklen_t addrlen);
```

This function is used to connect a TCP client with a TCP server. We do not need to know the client's address information, hence there is no need to call the `bind` function on the client before we call `connect`. The kernel will do all the dirty work for us in this case. It chooses the ephemeral port and retrieves the IP address information.

The first parameter (`SOCKET s`) defines the socket to connect. The second parameter (`const struct sockaddr *addr`) must provide information about the server's address: the IP address and port number. If we fail to give the correct address information for the server, the connection will fail. The third parameter (`socklen_t addrlen`) is used to define the length of the address structure in bytes.

This function does not return before the TCP's three-way handshake is complete. If an error occurs before that, the function will return. So when this function returns, we know that either the connection is established or it could not be established.

The client will wait a total of 75 seconds for the remote server to respond to its SYN segment of the three-way handshake. If no response is received during this time, the function returns an error. If the client does receive a response to the SYN segment, but it is an RST (reset), it means that the server machine is running but not waiting for connections at the TCP port we specified. The function will return an error value.

These and any other specific error values must be retrieved by using the `WSAGetLastError` function in Windows or the `errno` variable in Unix.

Win32 return values:

- Success: 0
- Failure: `SOCKET_ERROR`

Unix return values:

- Success: 0
- Failure: `-1`

listen Function (Unix, Win32)

```
int listen(SOCKET s, int backlog);
```

The `listen` function is used for connection-oriented sockets to set the socket to listen for incoming connections. This means that it is not used for UDP sockets. This function is called after the calls to the `socket` and `bind` functions. This function must be called before the `accept` function is called. This is a server-side function.

The first parameter (`SOCKET s`) must be a bound, unconnected socket. The socket must have been created with the `socket` function, and it must be bound with the `bind` function. The second and last parameter (`int backlog`) defines the maximum number of connections allowed. This sounds very simple but in fact is not. First of all, all operating systems have their own maximum for this. For example, Windows NT 4.0 Server has a maximum backlog value of 100. Normally, on operating systems that are not designed for server usage, the backlog value is about 4 to 5.

The kernel keeps track of two connection queues: incomplete and complete connections. The latter one is clear; they are the connections that have been established all the way. This means that the TCP three-way handshake is complete with them. The incomplete connections are connections that have the first part of the TCP three-way handshake complete. On a packet level it means that the first SYN packet from the client has been received by the server. The sum of these two queues cannot exceed the backlog value.

After this function successfully returns, the used socket is called a *listening socket*.

If the backlog value is set to be larger than the maximum of the operating system, the value is silently set to the nearest valid value. “Silently” means that there is no way to know about it; no error value is returned. If the queue is full and there is an incoming connection, `listen` will return `ECONNREFUSED` on Unix and `WSAECONNREFUSED` on Windows. These error values must be retrieved by the `WSAGetLastError` function in Windows or the `errno` variable in Unix.

Win32 return values:

- Success: 0
- Failure: `SOCKET_ERROR`

Unix return values:

- Success: 0
- Failure: `-1`

accept Function (Unix, Win32)

```
int accept(SOCKET s, struct sockaddr *addr, socklen_t *addrlen);
```

This function is used to accept an incoming TCP connection. This function is run after the `listen` function has filled the completed connections queue with at least one connection. When the queue is empty and this function is run, the process is either put to sleep or moves on to the next command, depending on the socket I/O option (blocking or non-blocking). This is a server-side function.

The first parameter (`SOCKET s`) defines the socket that holds the connection to be accepted. This must be a listening socket. The second parameter (`struct sockaddr *addr`) is used to get the address information of the client whose connection we are just about to accept. This can be set to `NULL` if we are not interested in this information. The third parameter (`socklen_t *addrlen`) is a value-result parameter, which must be set to the size of the address structure before calling `accept`. When the function returns, this parameter holds the number of bytes allocated for the address structure by the function. This can be set to `NULL` if we are not interested in this information.

After `accept` successfully returns, the used socket is called a *connected socket*.

The return value is a new descriptor for the socket if everything went fine. The listening socket given to the function as a parameter stays untouched. It is not removed, and its descriptor is not changed. We can still listen for new incoming connections with the very same socket. What we get from `accept` is a brand new connected socket.

If there was an error, `accept` returns an error value indicating it, but to get accurate error values we use the `WSAGetLastError` function in Windows or the `errno` variable in Unix.

So if we have a TCP server that will serve multiple clients at once, we create one listening socket and then spawn new connected sockets for each connection.

Win32 return values:

- Success: A nonnegative descriptor (integer)
- Failure: `INVALID_SOCKET`

Unix return values:

- Success: A nonnegative descriptor (integer)
- Failure: `-1`

close Function (Unix)/closesocket Function (Win32)

```
int close(SOCKET s);
int closesocket(SOCKET s);
```

The Unix and Windows versions of this function have different names, but the function is the same on both operating systems.

This function closes the TCP socket. Before actually terminating the connection, TCP will send all queued data. We cannot send or receive any more data even if the connection has not been terminated for good. When all queued data is sent, TCP's four-packet termination process initiates. We can make this function work differently by adjusting socket options. We discuss these options in Chapter 6, "I/O Operations."

The socket descriptor is returned to be reused. So after closing a socket, we may encounter a socket with the same descriptor as the one we closed. However, they have nothing do with each other. For example, if we close a socket and then immediately create a new one, the new socket must be initialized normally even if the descriptor is the same as the one we just closed.

If an error occurs, we need to use the `WSAGetLastError` function in Windows or the `errno` variable in Unix to retrieve the actual error value.

Win32 return values:

- Success: 0
- Failure: `SOCKET_ERROR`

Unix return values:

- Success: 0
- Failure: -1

Input/Output Functions

There are four basic functions for sending/receiving data in the sockets API — two for each operation. All four are explained in this section.

send Function (Unix, Win32)

```
int send(SOCKET s, const void *buf, size_t len, int flags);
```

This function is used to send data to a socket. The socket must be successfully connected before this function can be used. This function is usually used by stream (TCP) sockets only, but it is possible to use this

with datagram (UDP) sockets too. The UDP socket must be connected using the sockets `connect` function, but that is beyond the scope of this book.

The first parameter (`SOCKET s`) is the socket to which we want to send the data. It must be a connected socket. The second parameter (`const void *buf`) is the data itself. It is a pointer to a buffer containing the data we want to send. The third parameter (`size_t len`) defines the number of bytes to send. We do not need to send the whole data buffer (and it is not always even possible). The fourth and last parameter (`int flags`) is used to set different flags (options) with the current send process. These flags are temporary, so they must be set every time they are required by a send process. We can OR various flags together. Table 5-4 lists the possible flags.

HINT OR'ing is a bit-wise operation. Bit-wise operations are the lowest level of operations available as computers work using bits — 1s and 0s. Because of this, they are also very fast and take very little memory. I recommend learning more about bit-wise operations and using them wherever possible.

Table 5-4: send/sendto flags

Flags	Description	Unix	Win32
MSG_DONTROUTE	Target host is locally connected to the network. Do not look for the target host from a routing table.	Yes	Yes
MSG_DONTWAIT	Set the current output non-blocking. Do not wait for the output of data.	Yes	No
MSG_OOB	Send out-of-band data.	Yes	Yes

When `send` returns, it either returns the number of bytes sent or indicates that an error occurred. We cannot send zero size data with TCP protocol. If we get a 0 return value from `send`, it means that the other host has closed the connection or the connection is broken. If there was an error, we need to retrieve the actual error values by using the `WSAGetLastError` function in Windows or the `errno` variable in Unix. Even if `send` returns successfully, it does not mean that the other end successfully received the data.

Win32 return values:

- Success: Number of bytes sent
- Failure: `SOCKET_ERROR`

Unix return values:

- Success: Number of bytes sent
- Failure: -1

recv Function (Unix, Win32)

```
int recv(SOCKET s, void *buf, size_t len, int flags);
```

This function is very similar to the `send` function, but this time we are receiving data instead of sending it. The parameters are very similar.

The first parameter (`SOCKET s`) is the socket we want to receive the data. It must be a connected socket. The second parameter (`void *buf`) is the data buffer. It is a pointer to a buffer where we want to store the data. The third parameter (`size_t len`) defines the number of bytes to receive. The fourth parameter (`int flags`) works the same way as in the `send` function. We can set temporary socket input options here by OR'ing various options together or simply set one on. Table 5-5 lists the input flags.

Table 5-5: `recv/recvfrom` flags

Flags	Description	Unix	Win32
<code>MSG_DONTWAIT</code>	Set the current output non-blocking. Do not wait for the output of data.	Yes	No
<code>MSG_OOB</code>	Receive out-of-band data.	Yes	Yes
<code>MSG_PEEK</code>	Peek at the incoming data. The data is copied to the receive buffer, but it is not removed from the incoming data queue.	Yes	Yes
<code>MSG_WAITALL</code>	Wait for all the data to be received before returning.	Yes	No

The return value of `recv` tells us how many bytes it received or, in case of an error, an error value. If `recv` returns 0, the other side has closed the connection or the connection is broken. If there was an error, we need to retrieve the actual error values by using the `WSAGetLastError` function in Windows or the `errno` variable in Unix.

Win32 return values:

- Success: Number of bytes received
- Failure: `SOCKET_ERROR`

Unix return values:

- Success: Number of bytes received
- Failure: -1

sendto Function (Unix, Win32)

```
int sendto(SOCKET s, const void *buf, size_t len, int flags, const struct
           sockaddr *to, socklen_t addrlen);
```

Like the `send` function, `sendto` is also used to send data to a socket, but with this function it is not necessary for the socket to be connected. This is the function that is normally used for datagram (UDP) socket output, but it can be used for stream (TCP) sockets too.

The first parameter (`SOCKET s`) is the socket to which we want to send the data. The second parameter (`const void *buf`) is the data itself. It is a pointer to a buffer containing the data we want to send. The third parameter (`size_t len`) defines the number of bytes to send. We do not need to send the whole data buffer (and it is not always even possible). The fourth parameter (`int flags`) is used to set different options on with the current send process. These flags are temporary, and so they must be set every time they are required by a send process. We can OR various flags together. The flags listed in Table 5-4 work with `sendto` also. The fifth parameter (`const struct sockaddr *to`) is the address structure where we want to send the data. The address structure must contain the IP address and port. The sixth and last parameter (`socklen_t addrlen`) is used to define the size of the address structure.

Similarly to the `send` function, `sendto` returns the number of bytes sent or indicates that an error occurred. It is okay to send zero size datagrams. When we send zero size datagrams, only the IP (v4 or v6) header and the UDP header are sent, so `sendto` can return 0 and still function normally. If there was an error, we need to retrieve the actual error values by using the `WSAGetLastError` function in Windows or the `errno` variable in Unix. Even if `sendto` returns successfully, it does not mean that the other end successfully received the data.

Win32 return values:

- Success: Number of bytes sent
- Failure: `SOCKET_ERROR`

Unix return values:

- Success: Number of bytes sent
- Failure: -1

recvfrom Function (Unix, Win32)

```
int recvfrom(SOCKET s, void *buf, size_t len, int flags, struct sockaddr
*from, socklen_t *addrlen);
```

This function is usually used for datagram (UDP) socket input. It is similar to the `recv` function, but we define the IP address and port to read from as well as the socket.

The first parameter (`SOCKET s`) is the socket where we want to receive the data. The second parameter (`void *buf`) is the data buffer. Again, it is a pointer to a buffer where we want to store the data. The third parameter (`size_t len`) defines the number of bytes to receive. The fourth parameter (`int flags`) is used to set the temporary input flags. Table 5-5 lists the possible flags. The fifth parameter (`struct sockaddr *from`) is a pointer to the address structure that holds the address information of the host we want to receive data from. The sixth parameter (`socklen_t *addrlen`) is a value-result argument. It defines the size of the address structure and also returns the updated size when the function returns.

The number of received bytes is returned if everything went fine. Unlike with the `recv` function, a 0 return value does not mean that the other host has closed the connection. That is obvious as there is no such thing as connection in UDP. This also means that we can write datagrams of zero size. If there was an error, we need to retrieve the actual error values by using the `WSAGetLastError` function in Windows or the `errno` variable in Unix.

Win32 return values:

- Success: Number of bytes received
- Failure: `SOCKET_ERROR`

Unix return values:

- Success: Number of bytes received
- Failure: `-1`

Address Data Conversion Functions

We humans prefer to have our information in written text or numbers instead of just ones and zeroes like computers. An Internet address is one of those things that may be displayed in many ways, depending on who, or better yet what, uses it. Humans can remember the addresses best when they are written as text, for example, “www.hunted-cow.co.uk.” As computers do not have to worry about forgetting

anything, they can be optimized to store the addresses in a format that takes the least amount of memory.

We have three functions for IPv4 addresses and two functions for IPv6 addresses. These two new functions are protocol independent though, so it does not matter which protocol we use with them.

We explain only one of these functions here because we need that function in our network library, and the others are easy to understand after you understand this one.

inet_aton Function (Unix, Win32)

```
int inet_aton(const char *string, struct in_addr *address);
```

This function converts the provided string address into a network byte ordered binary value. Note that depending on the DNS server configuration, the string can be an IP address in numeric or dotted decimal format or in written text as most web addresses are.

Client/Server Programming

Now that we have everything set up for the actual network programming, we can move on to write our server and client code. Servers and clients work very differently. Most of the time the server is waiting for connections to come in. Once one does come in, it serves the client that is connected.

Server Methods

Servers can be very passive applications. There can be times when they do not do anything at all. This can happen if there are no clients to be served. We must keep this in mind when creating the servers, because if we make our server loop when there are no connections, it can easily drain all the CPU time just by looping an “empty” loop.

If our server is supposed to serve only one client at a time, it is called an iterative server. It does not listen for incoming connections once a connection is already open. When the connection terminates, the server starts to listen for a new connection. This kind of server is not very useful, because multiple clients may be “on hold” since the server can process only one connection at a time.

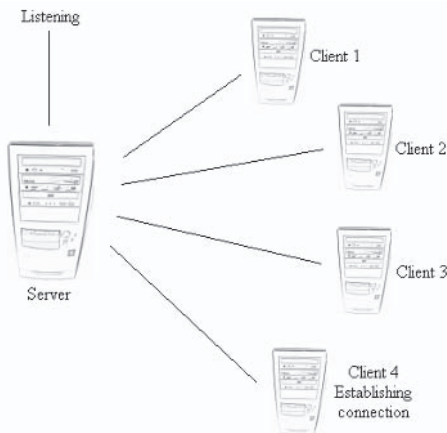
Computer game servers simply cannot work this way, because it is against the principal idea of these servers. The idea is that multiple players connect to a server and then communicate via that server (i.e., play the game). That is why this kind of play is called *multiplay*.

Servers that process multiple clients at once are called concurrent servers. When the server is started, it starts listening for incoming connections. When a connection comes in, it creates a child process (a *thread*) and the main program continues listening for incoming connections. The main program in this case is often also a thread, and not really the actual main program. The reason is simple: If it were the main program, it would not be much of a server. The main program is the very core, the part that initiates the listening functions and so on.

Figure 5-2 shows how a concurrent server works. The listening socket is always there and, depending on the state of the application (i.e., whether it will allow any more connections), it will accept all the new connections coming in. Clients 1 to 3 have already established a connection and are being served normally by the server. Client 4 has just connected to the server, and is still in the process of the TCP three-way handshake. The listening socket is already free and has started to listen for more connections.

Looking from the outside, a concurrent UDP server works the same way as TCP servers. Clients inform that they are there and the server takes care of them. There are lots of technical differences though, and these are discussed in the next section.

Figure 5-2
A concurrent
server



Clients

Clients are active applications. They initiate the connection to the server, so they never wait for things to happen (other than wait for data from the server).

The client application is the application that the normal user uses. In the world of gaming, the client is the game itself. The player should not know anything else about the server except the address and port

number. Sometimes even this information is built into the game, so the player does not have to set the IP address and/or port manually.

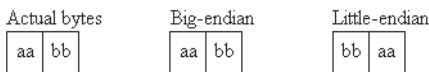
When working in a *local area network* (LAN), we can create a server search system, which is one way to eliminate the need for manually setting the address information. We discuss this system in Chapter 6, “I/O Operations.”

Byte Ordering

There is no standard way of ordering bytes in computer memory. Most PCs nowadays use Intel’s way of ordering the bytes, which is to store them in little-endian order. This means that if we have a two-byte variable, the last actual byte is stored first in the memory. Big-endian means that the bytes are stored in the correct order — first byte first.

Different processors store and access the bytes differently. As we said, Intel’s way is to store them in little-endian order. But because we are creating multiplatform applications, we cannot just use this one way. We must have a way to transform the bytes into a format that all computers understand. Therefore, we have to use *network byte order*. The network byte order is big-endian with the Internet protocols we use. There are functions to convert network byte ordered bytes into host byte ordered.

Figure 5-3
Byte ordering



Creating a Server

Now we are going to learn how to create a server on Unix and Windows, using both TCP and UDP. Let’s go through the most important events when creating a server.

Every sockets application (client or server) must first create the socket. Depending on the protocols we use, the parameters change accordingly.

```
// A stream (TCP) IPv4 socket  
SOCKET listeningSocket;
```

```
listeningSocket = socket(AF_INET, SOCK_STREAM, 0);
```

Then, when the socket is successfully created, we usually fill in the address information of the server. We do not have to enter the local IP address of the server if we have only one network interface card on the server machine. If we have more than one, we would just enter the IP address we want to use. Here we assume that there is only one card, so

we let the kernel automatically fill in the IP address. But we do have to enter the port number ourselves. We must remember the restrictions that exist when choosing a port number for our server. Chapter 4, “Introduction to TCP/IP” explains those things that limit port number availability.

```
struct sockaddr *servAddr;
struct sockaddr_in *inetServAddr;
int portNumber;

// Allocate memory for the address structure and set it to zero.
servAddr = (struct sockaddr *) malloc(sizeof(sockaddr));
memset((char *) servAddr, 0, sizeof(sockaddr));

// Fill the address structure.
servAddr->sa_family = (u_short) AF_INET;
inetServAddr = (struct sockaddr_in *) servAddr;
inetServAddr->sin_port = htons((u_short) portNumber);
```

Of course, if we are developing a server for personal use only (LAN only), we can forget some of these restrictions. But even in this case we cannot use just any number, because the operating systems use some ports without having any external application installed. Because we are developing a game server for the public, we must choose a port that has no restrictions at all.

Once we have filled in the required address information, we must bind this information to the socket we created in the beginning.

```
// Bind the address information to the socket.
error = bind(listeningSocket, servAddr, sizeof(sockaddr));
```

NOTE Whatever we are programming, we should never forget to check every possible function for errors. A lot of crashes that people blame on the operating system are actually caused by an application that does not handle errors properly.

Now we need to discuss the TCP and UDP code separately because of their obvious differences.

TCP

As we have already learned, TCP is the easy transport protocol (for the programmer). This is also true when creating servers. All we need to do now is make the server listen for incoming connections and accept them. For every new connection, the server creates a child process and a new socket (if we are talking about a concurrent server).

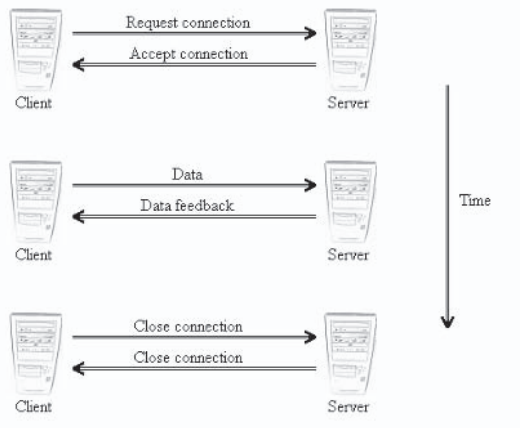
```
// Listen for incoming connections. Queue max 5 connections.
error = listen(listeningSocket, 5);
```

...

```
// Accept the connection. Accept is a blocking function.
connectedSocket = accept(listeningSocket, NULL, NULL);
```

Figure 5-4 shows normal TCP client/server operation. You can see how much easier and faster it is to use UDP, as shown in Figure 5-5.

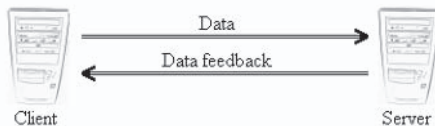
Figure 5-4
Normal TCP
client/server
operation



UDP

Because UDP is a connectionless protocol, it does not have a function for listening for incoming connections. The principal idea of UDP is that it simply does not have to listen to them. A UDP server just reads the incoming datagrams and acts how the programmer wants it to act.

Figure 5-5
Normal UDP
client/server
operation



But how do we keep the server organized with all the clients sending data to it? A game server needs to send data to a client pretty much whenever it is required, so we cannot let the client do all the active sending.

The solution for this is a so-called “knocking” system. When the client wants to tell the server that it exists and wants to interact with the server, the client sends a “knock” datagram to the server. This datagram can be anything; it is up to you to decide what information is stored within it. When the server receives a datagram or message like this, it updates its list of clients and either creates a new child process to serve the client or simply acknowledges that there may be incoming

messages from that client. In the latter case, the server uses the one and only socket to interact with all the clients.

A UDP server that uses only one socket to interact with clients must always retrieve the address information when it is receiving data (assuming that it wants to send data back also). With UDP we cannot simply send data to a socket without providing the exact address. This kind of server is not necessarily an iterative server. It may process multiple clients at a time, as it responds only when the client sends data to the server.

Simple Echo TCP Server

It is time to create our very first sockets application: a simple echo TCP server. This server is an iterative server (serves only one client at a time), and its function is to return the character we send to it in uppercase. The protocol used is TCP, but later we will see how easy it is to make a UDP server from our existing TCP code. This example program is of no real use and is intended to show the basic idea of every server, which is the following: A request from the client is sent to the server, and the server sends feedback to the client. In this example, the request is simply the letter sent to the server, and the feedback is the uppercase letter that is sent back to the client.

The following code shows the simple echo TCP server in its entirety. The code is written for the Windows operating system, but it is easy to change it so that it compiles and runs on Unix too. However, that is not the idea of this example, so let's skip it for now.

The functional concept of this server is that the server listens for a connection to come in and when one does, it accepts it. Then the client is supposed to send single letters to the server, and the server will send them back in uppercase. If the user on the client end enters the letter "q" (lowercase), the server stops automatically. The whole application is stopped, so it will not accept any other connections. For this example, there is no need to write any extra checking if the user is exiting on the client end, because the `recv` function will notice if the connection is closed.

```
#ifndef _WINSOCKAPI_
#define _WINSOCKAPI_
#endif

#include <windows.h>
#include <winsock2.h>

#include <stdio.h>

// Declare the sockets we use.
```

```
SOCKET listeningSocket;
SOCKET connectedSocket;

int InitSockets(void)
{
    struct sockaddr *servAddr;
    struct sockaddr_in *inetServAddr;

    int error = 0;

    // Create the socket.
    listeningSocket = socket(AF_INET, SOCK_STREAM, 0);

    if(listeningSocket == INVALID_SOCKET)
    {
        printf("error: socket() failed");
        return -1;
    }

    // Allocate memory for the address structure and set it to zero.
    servAddr = (struct sockaddr *) malloc(sizeof(sockaddr));
    memset((char *) servAddr, 0, sizeof(sockaddr));

    // Fill the address structure.
    servAddr->sa_family = (u_short) AF_INET;
    inetServAddr = (struct sockaddr_in *) servAddr;
    inetServAddr->sin_port = htons((u_short) 9009);

    // Bind the address information to the socket.
    error = bind(listeningSocket, servAddr, sizeof(sockaddr));

    if(error == SOCKET_ERROR)
    {
        printf("error: bind() failed");
        free(servAddr);
        return -1;
    }

    free(servAddr);
    servAddr = NULL;

    // Listen for incoming connections. Queue only one connection.
    error = listen(listeningSocket, 1);

    if(error == SOCKET_ERROR)
    {
        printf("error: listen() failed");
        return -1;
    }

    // Accept the connection. Accept is a blocking function.
    connectedSocket = accept(listeningSocket, NULL, NULL);

    if(connectedSocket == INVALID_SOCKET)
```

```
    {
        printf("error: socket() failed");
        return -1;
    }

    return 0;
}

void ServerProcess(void)
{
    int connectionOpen;

    char buf[2];

    connectionOpen = 1;

    // Loop as long as connection is open.
    while(connectionOpen)
    {
        // Read the incoming data from the connected socket.
        if(recv(connectedSocket, buf, 2, 0))
        {
            // Set the received letter to uppercase and
            // make sure the string ends after that by setting the next
            // byte to NULL.
            buf[0] = toupper(buf[0]);
            buf[1] = '\0';

            printf("Got message from client: %s\n", buf);

            // Send the feedback.
            if(send(connectedSocket, buf, 2, 0) == SOCKET_ERROR)
            {
                connectionOpen = 0;
            }
        }
        else
        {
            closesocket(connectedSocket);
            connectionOpen = 0;
        }
    }
}

int main(void)
{
    if(NET_WinSockInitialize() != 0)
    {
        printf("Critical error, quitting\n");

        return -1;
    }

    if(InitSockets() != 0)
```

```
    {
        printf("Critical error, quitting\n");

        WSACleanup();

        return -1;
    }

    ServerProcess();

    WSACleanup();

    return 0;
}
```

Now let's see what is going on in the program. Let's start from the main function, as that is where the application always starts.

main Function

You may notice that there is not much in the `main` function, specifically no basic socket functions. But there are two function calls that are very important. Let's have a closer look at them:

```
NET_WinSockInitialize();
```

This is the function we introduced at the beginning of this chapter. It belongs to the network library that we will create later on. This function initializes the WinSock API for us. After a successful call, the WinSock API is ready to be used.

NOTE On Unix, we do not call the `NET_WinSockInitialize` function.

```
WSACleanup();
```

This function is very important also. It is used to uninitialized WinSock. We are not allowed to exit the application without calling this function if the WinSock API is initialized. As we have already learned, one of this function's tasks is to unregister the WinSock DLL from our application. Windows' DLL registration system will lose track of the registered DLLs if we do not call this function at the end of each WinSock application.

InitSockets Function

The `InitSockets` function does all the initialization of sockets so they can be used and the server process itself can start.

First we create the listening socket and check that it is successfully created. A listening socket is a socket that the server uses to listen for

incoming connections. Once a connection comes in and is accepted, the server will start using a connected socket and leave the listening socket free for other clients.

The socket we create here is an IPv4 stream socket. Because it is a stream socket, it uses the TCP transport protocol. The flags parameter is set to 0 as it is currently not used in the sockets API. Then we check whether the socket descriptor is invalid. If it is, we simply exit the program. If the socket is created as it is supposed to be, we move on.

```
listeningSocket = socket(AF_INET, SOCK_STREAM, 0);

if(listeningSocket == INVALID_SOCKET)
{
    printf("error: socket() failed");
    return -1;
}
```

Now we fill in the address information of the server. First we need to allocate memory for the structure and set it to 0. Then we fill the structure with the required information: protocol family and port number. There is no need to tell the program the IP address of the computer, as it can automatically retrieve it. If there is more than one network interface card installed, we can choose the one we want here. For this example program we have selected the port number 9009.

```
servAddr = (struct sockaddr *) malloc(sizeof(sockaddr));
memset((char *) servAddr, 0, sizeof(sockaddr));

// Fill the address structure.
servAddr->sa_family = (u_short) AF_INET;
inetServAddr = (struct sockaddr_in *) servAddr;
inetServAddr->sin_port = htons((u_short) 9009);
```

Now that we have the address information ready in the structure, let's tell the socket to use that information. We bind the information to the listening socket with the `bind` function. Again, we must check whether an error occurred in the call to `bind`. If so, we need to free the allocated memory before we are allowed to exit. If everything went fine, we free the memory we allocated for the address structure and move on again.

```
// Bind the address information to the socket.
error = bind(listeningSocket, servAddr, sizeof(sockaddr));

if(error == SOCKET_ERROR)
{
    printf("error: bind() failed");
    free(servAddr);
    return -1;
}
```



```
free(servAddr);
servAddr = NULL;
```

Everything is now ready for listening for incoming connections. At first glance it may seem odd that it is just one call to a function without any “loop as long as there are no connections coming in” loop. Does the function just run once and then the process moves on to the next one? In this case, no. Some theory is required here to understand this. A socket can be blocking or non-blocking. With a blocking socket, some sockets functions will go to sleep when there is no action of any kind that needs to be processed. A non-blocking socket, on the other hand, will not put the functions to sleep. Once the functions are called, they check if there is an action to process; if there is not, they return and the next command (function) of the process is run. By default, all sockets are blocking. Blocking and non-blocking I/O is discussed more in Chapter 6, “I/O Operations.”

So if we see a fragment of code similar to the following code, we cannot say how the application will perform, as we do not know if the socket is blocking or not. But as we do know that it is blocking in this example, we know that the application will stop at the `listen` function as long as there are no incoming connections.

Now we put the listening socket to the use for which it was created by passing it as a parameter to the `listen` function. We set the backlog value to 1 because our server will process only one client at a time and, better yet, one client per instance.

Again, we cannot forget checking for errors. If `listen` fails, we exit the application.

```
// Listen for incoming connections. Queue only one connection.
error = listen(listeningSocket, 1);

if(error == SOCKET_ERROR)
{
    printf("error: listen() failed");
    return -1;
}
```

Now we have reached the `accept` function, so we know that someone wants to connect to our server. What we do now is create a new socket for the soon-to-be connected client. We pass the listening socket as a parameter to `accept`, as the connection we want to accept is on that socket. We have no interest in the address information of the client, so pass `NULL` as the two remaining parameters.

Next we check for errors. If the connected socket is invalid after a call to `accept`, we exit the application. If it is valid, we are done initializing the server.

```
// Accept the connection.
connectedSocket = accept(listeningSocket, NULL, NULL);

if(connectedSocket == INVALID_SOCKET)
{
    printf("error: socket() failed");
    return -1;
}
```

ServerProcess Function

Now we get to the server process itself. This function has the main loop that every program has. It is looped as long as the connection is open. Let's have a closer look at the input/output functions we use here:

```
// Read the incoming data from the connected socket.
if(recv(connectedSocket, buf, 2, 0))
{
    // Set the received letter to uppercase and
    // make sure the string ends after that by setting the next
    // byte to NULL.
    buf[0] = toupper(buf[0]);
    buf[1] = '\0';

    printf("Got message from client: %s\n", buf);

    // Send the feedback.
    if(send(connectedSocket, buf, 2, 0) == SOCKET_ERROR)
    {
        connectionOpen = 0;
    }
}
else
{
    closesocket(connectedSocket);
    connectionOpen = 0;
}
```

At this point in the program, we can forget about the listening socket. This program is not going to use it anymore. The socket we use from now on is the connected socket that we got from the `accept` function.

Let's get the data flowing! We call the `recv` function inside an `if` statement because we need to know if `recv` really read data from the socket or if it returned 0 to indicate that the connection has been closed or is lost. This is possible with a blocking socket, as `recv` will not return before it has data to read or before it notices that the connection is closed. A non-blocking socket would return 0 from a call to `recv` if there is no data to read but the connection is still alive.

We read the data to a very small buffer in this example, only two bytes in size. When we notice that data has arrived, we process it.

First, we simply set the received letter to uppercase, and then make sure the next letter is NULL; therefore, the string ends after the first letter. After this is done, we show the user what we received and send it back to the client. If we could not send the data (if `send` returns less than 0), we assume that the connection is broken and exit the program.

We close the socket and exit the program if `recv` returns 0.

Remember that in this example, because the sockets we use are blocking, the socket functions do not return if there is nothing happening (for example, if `recv` is not receiving data).

Simple Echo UDP Server

As we want to use UDP as well as TCP in our socket programs, we now modify the simple echo TCP server code to work with UDP. Modifying the TCP code is rather easy. The following shows all of the UDP server code.

```
#ifndef _WINSOCKAPI_
#define _WINSOCKAPI_
#endif

#include <windows.h>
#include <winsock2.h>

#include <stdio.h>

// Declare the sockets we use.
SOCKET Socket;

int InitSockets(void)
{
    struct sockaddr *servAddr;
    struct sockaddr_in *inetServAddr;

    int error = 0;

    // Create the socket.
    Socket = socket(AF_INET, SOCK_DGRAM, 0);

    if(Socket < 0)
    {
        printf("error: socket() failed");
        return -1;
    }

    // Allocate memory for the address structure and set it to zero.
    servAddr = (struct sockaddr *) malloc(sizeof(sockaddr));
    memset((char *) servAddr, 0, sizeof(sockaddr));

    // Fill the address structure.
    servAddr->sa_family = (u_short) AF_INET;
```

```
inetServAddr          = (struct sockaddr_in *) servAddr;
inetServAddr->sin_port = htons((u_short) 9009);

// Bind the address information to the socket.
error = bind(Socket, servAddr, sizeof(sockaddr));

if(error == SOCKET_ERROR)
{
    printf("error: bind() failed");
    free(servAddr);
    return -1;
}

free(servAddr);
servAddr = NULL;

return 0;
}

void ServerProcess(void)
{
    struct sockaddr_in inetClientAddr;
    int clientLen;

    int connectionOpen;

    char buf[2];

    clientLen = sizeof(inetClientAddr);

    connectionOpen = 1;

    // Loop as long as connection is open.
    while(connectionOpen)
    {
        // Read the incoming data from the connected socket.
        if(recvfrom(Socket, buf, 2, 0, (struct sockaddr *)
            &inetClientAddr, &clientLen))
        {
            // Set the received letter to uppercase and
            // make sure the string ends after that by setting the next
            // byte to NULL.
            buf[0] = toupper(buf[0]);
            buf[1] = '\0';

            printf("Got message from client: %s\n", buf);

            // Send the feedback.
            if(sendto(Socket, buf, 2, 0,
                (struct sockaddr *) &inetClientAddr, clientLen) ==
                SOCKET_ERROR)
            {
                connectionOpen = 0;
            }
        }
    }
}
```

```
        }
        else
        {
            connectionOpen = 0;
        }
    }
}

int main(void)
{
    if(NET_WinSockInitialize() != 0)
    {
        printf("Critical error, quitting\n");

        return -1;
    }

    if(InitSockets() != 0)
    {
        printf("Critical error, quitting\n");

        WSACleanup();

        return -1;
    }

    ServerProcess();

    WSACleanup();

    return 0;
}
```

The biggest change in the UDP code in comparison to the TCP code is that we have only one socket. On the TCP server, we had a listening socket and a connected socket. On the UDP server, we have only one “generic” socket because there is no need to listen for incoming connections and connect them. This one UDP socket just reads the incoming data and sends data back.

The following sections provide a closer look at the changes.

InitSockets Function

First, we change the calls to the `socket` and `bind` functions to match these two function calls. We have replaced the listening socket with the generic socket and we are now creating a datagram socket instead of a stream socket.

After calling `bind` we are done. If we were using TCP, we would start the listening process now. But with UDP there is no need, so we can start the server process function right after we have bound the local address information to the socket.

```
Socket = socket(AF_INET, SOCK_DGRAM, 0);  
...  
error = bind(Socket, servAddr, sizeof(sockaddr));
```

ServerProcess Function

For the server processing function, we declare two variables that are not used in the TCP version. The first one is the Internet client address structure, and the second one is an integer holding the length of the structure. We need to set the length variable to the size of the structure before we pass it to any function, as shown in the following code.

```
struct sockaddr_in inetClientAddr;  
int clientLen;  
...  
clientLen = sizeof(inetClientAddr);
```

The next thing we have changed is that we have replaced the `recv` and `send` functions with the `recvfrom` and `sendto` functions. They work almost like the ones in the TCP version, but there are two new parameters in `recvfrom` and `sendto`.

We must pass the address information structure and its length as parameters in both functions. In `recvfrom`, the address structure is filled by the function. When the function is receiving data, it fills the structure with the corresponding address where the data is coming from. The function also updates the length variable, as it is a value-result argument.

Now that we have the address information for where the data came from, we use it to send data to the correct host. We pass the address structure and the length of the structure to `sendto`, updated by `recvfrom`. This way we are always sending data to the correct host, because in this example program we only send data when we have first received it. If we had a program that required sending data even when we have not received anything (we need the address information though, hence at least one datagram must have been received before), we would need to store all the addresses of the clients we want to send data to. That is why it is a good idea to have some kind of a system where there is a dedicated message for informing the server of the client.

```
if(recvfrom(Socket, buf, 2, 0, (struct sockaddr *) &inetClientAddr,  
            &clientLen))  
...  
if(sendto(Socket, buf, 2, 0, (struct sockaddr *) &inetClientAddr,  
          clientLen) == SOCKET_ERROR)
```

These are all the changes we need to make the server work using the UDP protocol. The UDP version of the server works a little bit differently as it will not exit when the client is shut down. This is because there is no connection that is closed. This feature has its good and bad sides. One good thing is that the server can be used effectively because only one process is running all the time. A bad thing is the fact that we do not know when a client crashes or something else like that happens. But, as we have already said many times, UDP is unreliable, but it can be made reliable.

Creating a Client

What would we do with a server if we did not have a client application? Nothing. So let's make one. Remember that in computer games, the game itself is the client, so you must design your game so that sending and receiving data does not interfere with other parts of the game too much. Too much? It is almost impossible to make it work so that the communications library would not have any effect on the game flow.

First, let's look at the functions we need for all clients.

TCP

We create the client-side socket exactly like the one on the server. We must set it to use the same protocols on both ends (IPv4 and TCP in this case).

```
SOCKET Socket;  
Socket = socket(AF_INET, SOCK_STREAM, 0);
```

Then we convert the Internet address from the server IP number to a form that the computer can use. After that we fill the address structure, but we do not bind this address to the socket ourselves, because the `connect` function will do it for us.

```
struct sockaddr_in inetServAddr;  
int portNumber;  
  
u_long inetAddr = inet_addr(IPAddress);  
  
memset((char *) &inetServAddr, 0, sizeof(inetServAddr));  
inetServAddr.sin_family = AF_INET;  
inetServAddr.sin_port = htons((u_short) portNumber);  
inetServAddr.sin_addr.s_addr = inetAddr;  
  
error = connect(Socket, (struct sockaddr *) &inetServAddr,  
               sizeof(inetServAddr));
```

That is all the basic initializing there is to do on the client side when using the TCP protocol.

UDP

Creating a UDP socket requires only one modification: We need to change the second parameter of the `socket` function to `SOCK_DGRAM`. That is all. We fill the address structure in the same way as when using TCP.

```
SOCKET Socket;  
Socket = socket(AF_INET, SOCK_DGRAM, 0);
```

As the UDP protocol is a connectionless protocol, we do not need to run the `connect` function at all when using UDP. This is the biggest difference in initializing client sockets.

When sending data to a UDP server, we need to pass the address information of the server to the sending function every time we run it. Therefore we must store the address structure globally to be able to access it from all functions.

Simple Echo TCP Client

As we can see in the following code, the client source code is not very different from the server code.

```
#ifndef _WINSOCKAPI_  
#define _WINSOCKAPI_  
#endif  
  
#include <windows.h>  
#include <winsock2.h>  
  
#include <stdio.h>  
  
// Declare the only socket we need.  
SOCKET Socket;  
  
int InitSockets(char *IPAddress)  
{  
    struct sockaddr_in inetServAddr;  
    int error = 0;  
  
    // Create a TCP socket.  
    Socket = socket(AF_INET, SOCK_STREAM, 0);  
  
    if(Socket < 0)  
    {  
        printf("error: socket() failed");  
        return -1;  
    }  
}
```



```
    }

    // Create the Internet address from the IP number
    u_long inetAddr = inet_addr(IPAddress);

    memset((char *) &inetServAddr, 0, sizeof(inetServAddr));
    inetServAddr.sin_family      = AF_INET;
    inetServAddr.sin_port       = htons((u_short) 9009);
    inetServAddr.sin_addr.s_addr = inetAddr;

    // Try to connect the TCP server.
    error = connect(Socket, (struct sockaddr *) &inetServAddr,
                    sizeof(inetServAddr));

    if(error != 0)
    {
        printf("error: could not find server.\n");
        return -1;
    }

    return 0;
}

void ClientProcess(void)
{
    int connectionOpen;

    char transmitBuf[3];
    char receiveBuf[3];
    strcpy(transmitBuf, "");
    strcpy(receiveBuf, "");

    connectionOpen = 1;

    // Loop as long as connection is open.
    while(connectionOpen)
    {
        // Get the string to send.
        if(gets(transmitBuf))
        {
            if(strcmp(transmitBuf, "q") == 0)
            {
                closesocket(Socket);
                connectionOpen = 0;
                break;
            }

            // Send the transmit buffer to the socket.
            if(send(Socket, transmitBuf, 2, 0) == SOCKET_ERROR)
            {
                connectionOpen = 0;
            }
        }
    }
}
```

```
        // Read the incoming data from the connected socket.
        if(recv(Socket, receiveBuf, 2, 0))
        {
            printf("Got reply from server: %s\n", receiveBuf);
        }
        else
        {
            connectionOpen = 0;
        }
    }
}

int main(int argc, char *argv[])
{
    if(argc < 2)
    {
        printf("Usage: SimpleEchoTCPClient.exe <Server IP>\n");
        return -1;
    }

    NET_WinSockInitialize();

    if(InitSockets(argv[1]) != 0)
    {
        printf("Critical error, quitting\n");

        WSACleanup();

        return -1;
    }

    ClientProcess();

    WSACleanup();

    return 0;
}
```

Now let's see what is different.

main Function

The only thing that is different in this function in comparison to the server code is the following: the two parameters in the `main` function, the `if` statement, and the call to the `InitSockets` function. The `if` statement checks whether the user provided enough arguments when running the executable. The first argument is the executable name itself (in the form the user entered it), and the second argument in this case should be the IP address of the server. If there are not enough arguments, the program displays a “usage” message telling the user what arguments are needed. The call to the `InitSockets` function is

a little bit different, because now we need to pass the IP address information to it.

Other parts of the `main` function are similar to the server code — just some simple function calls and the cleanup code.

```
int main(int argc, char *argv[])
{
    if(argc < 2)
    {
        printf("Usage: SimpleEchoTCPClient.exe <Server IP>\n");
        return -1;
    }
    ...

    if(InitSockets(argv[1]) != 0)
    ...
}
```

InitSockets Function

The most obvious change in this function is that we now provide the server IP address within a parameter. We need to do this to make it possible to enter any IP address when running the client. We could hardcode an address to the code, but that would not be very wise. Then we should always have the server on the hardcoded IP address (which is not always even possible).

```
int InitSockets(char *IPaddress)
```

As we can see in the following code, the socket is created exactly like it is on the server end. There really is no way to make it different because we must use the very same protocols. Once more, we must check for errors.

```
// Create a TCP socket.
Socket = socket(AF_INET, SOCK_STREAM, 0);

if(Socket < 0)
{
    printf("error: socket() failed");
    return -1;
}
```

Now that we have the server IP address in a string, we need to convert it to a form the computer understands. After that we reset the address structure memory to 0. Then we fill the address information structure with this address and the well-known port number, which is 9009 in this example. We must convert the integer value 9009 (host byte ordered value) to network byte ordered format. We also set the protocol family to `AF_INET` since we are using IPv4.

```
// Create the Internet address from the IP number
u_long inetAddr = inet_addr(IPaddress);

memset((char *) &inetServAddr, 0, sizeof(inetServAddr));
inetServAddr.sin_family      = AF_INET;
inetServAddr.sin_port        = htons((u_short) 9009);
inetServAddr.sin_addr.s_addr = inetAddr;
```

Then all there is left to do in this function is to connect the server. To do this we use `connect`. You probably noticed that we do not call `bind` at all on the client. This is because we do not have to; `connect` does the address binding for us. Therefore, we must provide the address structure we just filled earlier for `connect`. We typecast the structure to the generic address format, because `connect` is designed to work on both IPv4 and IPv6, and so it accepts only generic format addresses.

It is very important to check for errors here. If there is an error, it usually means that we could not find the server. Some other errors may occur too, but usually it is enough to inform the user that we could not connect the server.

```
// Try to connect the TCP server.
error = connect(Socket, (struct sockaddr *) &inetServAddr,
                sizeof(inetServAddr));

if(error != 0)
{
    printf("error: could not find server.\n");
    return -1;
}
```

`connect` also is a blocking function, but not like other blocking functions. If we set the socket to non-blocking mode, the `connect` function is not affected by this. It will still wait a certain amount of time for the connection to succeed, and after that time it will fail.

ClientProcess Function

This function matches the `ServerProcess` function on the server. Its purpose is to wait for the user to enter the letter to send and then send it. After that it will wait for a response from the server, then the loop starts all over again. Let's see what is going on in the function.

We have separate buffers for transmitting and receiving to prevent mix-ups. The application could work with one buffer only, but it is much better for the programmer that we have two buffers. In bigger programs, it is sometimes a must to have different buffers.

```
char transmitBuf[3];
char receiveBuf[3];
```

```
strcpy(transmitBuf, "");
strcpy(receiveBuf, "");
```

Here we get the letter to send into the transmit buffer using the `gets` function. Some problems arise if the user enters more than one letter, but in this example we do not worry about that because it really is beyond the scope of this example. For now, it is enough that we take care of this on the server by making sure the string ends after the first letter.

Next we check whether the letter entered was “q” (lowercase). If it was, we close the socket and exit the loop, thus exiting the whole program. The server will notice that we have closed the socket and it will exit too.

If the letter we entered was something other than “q,” we send it to the server. Because a string always contains at least two bytes, assuming that the string is not empty (one letter + NULL), and because we want to send only one letter, we send two bytes.

If `send` returns `SOCKET_ERROR`, something went wrong and we exit the program without any extra checking.

```
// Get the string to send.
if(gets(transmitBuf))
{
    if(strcmp(transmitBuf, "q") == 0)
    {
        closesocket(Socket);
        connectionOpen = 0;
        break;
    }

    // Send the transmit buffer to the socket.
    if(send(Socket, transmitBuf, 2, 0) == SOCKET_ERROR)
    {
        connectionOpen = 0;
    }
}
```

After we have sent the data to the server, we immediately start to wait for the response by using the function `recv`. If the number of bytes received is more than zero (the return value of `recv` is more than zero), it means that data has successfully arrived. If not, we exit the loop and the whole program.

After this, the loop starts all over again, assuming that no errors have been encountered and that the user did not enter the letter “q.”

```
// Read the incoming data from the connected socket.
if(recv(Socket, receiveBuf, 2, 0))
{
    printf("Got reply from server: %s\n", receiveBuf);
}
else
```

```
{
    connectionOpen = 0;
}
```

Simple Echo UDP Client

The following code shows the UDP version of the simple echo client.

```
#ifndef _WINSOCKAPI_
#define _WINSOCKAPI_
#endif

#include <windows.h>
#include <winsock2.h>

#include <stdio.h>

// Declare the only socket we need.
SOCKET Socket;

struct sockaddr_in inetServAddr;
int servLen = sizeof(inetServAddr);

int InitSockets(char *IPaddress)
{
    int error = 0;

    // Create a TCP socket.
    Socket = socket(AF_INET, SOCK_DGRAM, 0);

    if(Socket < 0)
    {
        printf("error: socket() failed");
        return -1;
    }

    // Create the Internet address from the IP number
    u_long inetAddr = inet_addr(IPaddress);

    memset((char *) &inetServAddr, 0, sizeof(inetServAddr));
    inetServAddr.sin_family      = AF_INET;
    inetServAddr.sin_port       = htons((u_short) 9009);
    inetServAddr.sin_addr.s_addr = inetAddr;

    return 0;
}

void ClientProcess(void)
{
    int connectionOpen;

    char transmitBuf[3];
    char receiveBuf[3];
```

```
strcpy(transmitBuf, "");
strcpy(receiveBuf, "");

connectionOpen = 1;

// Loop as long as connection is open.
while(connectionOpen)
{
    // Get the string to send.
    if(gets(transmitBuf))
    {
        // Send the transmit buffer to the socket.
        if(sendto(Socket, transmitBuf, 2, 0,
            (struct sockaddr *) &inetServAddr, servLen) ==
            SOCKET_ERROR)
        {
            connectionOpen = 0;
        }
    }

    // If the letter the user entered is "q", stop the application.
    if(strcmp(transmitBuf, "q") == 0)
    {
        connectionOpen = 0;
        break;
    }

    // Read the incoming data from the connected socket.
    if(recvfrom(Socket, receiveBuf, 2, 0, NULL, NULL))
    {
        printf("Got reply from server: %s\n", receiveBuf);
    }
    else
    {
        connectionOpen = 0;
    }
}

int main(int argc, char *argv[])
{
    if(argc < 2)
    {
        printf("Usage: SimpleEchoTCPCClient.exe <Server IP>\n");
        return -1;
    }

    NET_WinSockInitialize();

    if(InitSockets(argv[1]) != 0)
    {
        printf("Critical error, quitting\n");
    }
}
```

```
        WSACleanup();
    }
    return -1;
}

ClientProcess();

WSACleanup();

return 0;
}
```

Note that it is not much different from the TCP version. The following sections look at what has changed.

InitSockets Function

We create the socket exactly like in the TCP version but with one change. Instead of creating a stream socket, we create a datagram socket. To do this, we set the second parameter to `SOCK_DGRAM`.

Then we fill the address information structure that holds the server's address. Again, we do it exactly like in the TCP version, only this time the structure is a global variable as we need it elsewhere in the code. Once the structure is filled completely, we are done. There is no need to call `bind` or `connect` and we can move on to the client process function.

```
Socket = socket(AF_INET, SOCK_DGRAM, 0);
```

ClientProcess Function

In this function, we replace `send` and `recv` with the following functions and remove the socket closing functions. Unlike on the server-side code, we do not need to receive any data before we know the address of the server. This is obvious, is it not? If we do not know the address of the server, we hit the wall. We cannot do anything without that piece of information. Fortunately in computer games today, the servers can be found automatically using a built-in or external application that uses a server (we do not need to know the address of this server, as it is built in) to retrieve the IP addresses and ports of the servers. Let's assume that we know the address of the simple echo server and that we have entered the correct address when running this client. Now we use the address information structure that we filled in for the `InitSockets` function to send the data to the correct host. Then we assume that the server address does not change and we do not care about checking the address structure returned by `recvfrom`.

```
if(sendto(Socket, transmitBuf, 2, 0, (struct sockaddr *) &inetServAddr,
servLen) == SOCKET_ERROR)
```


...

```
if(recvfrom(Socket, receiveBuf, 2, 0, NULL, NULL))
```

NOTE It is possible that the server address changes between a `sendto` and `recvfrom`. For example, a concurrent UDP server spawns the connections on new ephemeral ports because it must use the well-known port for listening for incoming connections. So when a client sends data to the server, the server spawns a child process and a new port for that “connection.” Then the next time the client receives data from the server it must update its address information for the server because the port has changed.

Running the Simple Echo Application

To try out the simple echo application, we must first start the server program and then start the client program. We must start the programs in this order because the client application will exit if it cannot find anything on the server port.

To run the server, from the command prompt type:

```
> SimpleEchoTCPServer.exe
```

Then we run the client, passing the server IP address as the first and only argument:

```
> SimpleEchoTCPClient.exe 127.0.0.1
```

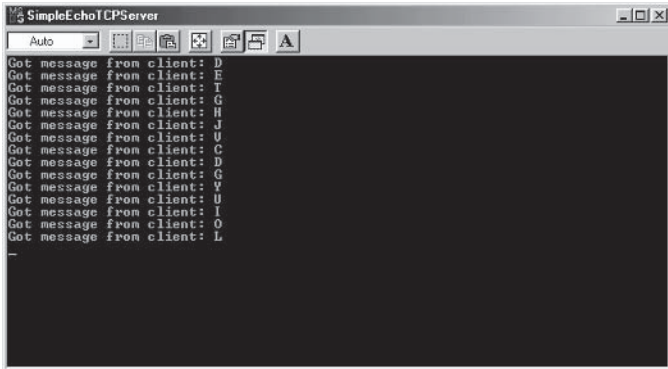
In this example, we are running both programs on the same host, and therefore we can use the IP address 127.0.0.1, which is a local host address. When you use this address, you are pointing to your own computer.

Now that both of the programs are running, we can start sending data back and forth. On the client side, enter any letter except “q” and then press Enter. You will see on the serverside how the letter is received, and almost immediately on the client side how the letter is sent back in uppercase.

To stop the application, enter the letter “q” on the client program. The TCP version will end both client and server, but the UDP version ends only the client.

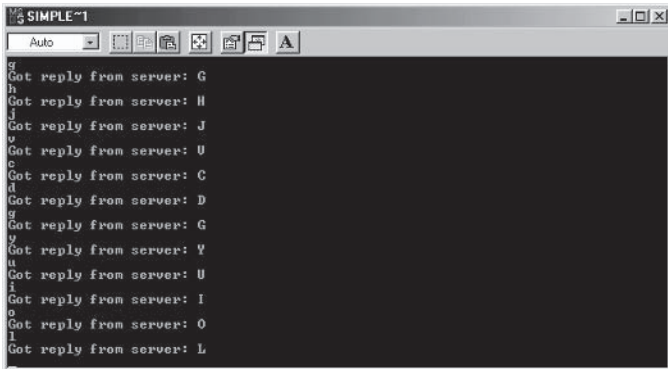
Figure 5-6 and 5-7 show a normal run of the simple echo application.

Figure 5-6
Simple echo
server running



```
SimpleEchoTCPServer
Auto
Got message from client: D
Got message from client: E
Got message from client: T
Got message from client: G
Got message from client: H
Got message from client: J
Got message from client: U
Got message from client: C
Got message from client: D
Got message from client: G
Got message from client: V
Got message from client: U
Got message from client: I
Got message from client: O
Got message from client: L
```

Figure 5-7
Simple echo
client running



```
SIMPLE~1
Auto
Got reply from server: G
h
Got reply from server: H
J
Got reply from server: J
u
Got reply from server: U
c
Got reply from server: C
d
Got reply from server: D
g
Got reply from server: G
y
Got reply from server: V
u
Got reply from server: U
i
Got reply from server: I
o
Got reply from server: O
l
Got reply from server: L
```

Summary

In this chapter, we encountered the basic socket functions for the first time. We learned about the parameters we pass to them and the values they return. We learned some of the technical differences between TCP and UDP. More importantly, we learned how to use the functions by creating our first sockets application, the simple echo application. We can now move on to more advanced technology, as we now know the basics.

This page intentionally left blank.

Chapter 6

I/O Operations

Introduction

A network input/output operation requires more than just knowing how to send and receive data. We need to know when to start receiving the data and how to set it up. This chapter discusses the input/output operations of socket network events. We will also learn a new way to send data and how to modify the way our sockets act.

Detecting Network Events

To be able to start reading the socket for incoming data at the correct time, we need to know when there is something to read. We could poll the read function on the socket all the time, but that is not always a good way to do it. We need to be able to wake up the read function whenever there is any incoming data. The following functions are used to accomplish this:

- `select` — Unix, Win32
- `WSAAsyncSelect/WSAEventSelect` — Win32

`select` (Unix, Win32)

```
int select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,
           const struct timeval *timeout);
```

The `select` function is used to set the kernel to wake up the process when a network event occurs. We can set the function to wake up the

process on any type of network event (writing, reading, and exception condition pending) at the same time. Plus, we can set a timeout value to make the process wake up after a certain amount of time.

The first parameter (`int maxfdpl`) defines the number of socket descriptors to test for network events. Its value is set to the highest socket descriptor to test plus one, because it is the number of descriptors (and the descriptor values begin from 0). So if we want to test our socket for network events, we should set this value to (at least) the socket descriptor plus one.

The second, third, and fourth parameters (`fd_set *readset`, `fd_set *writerset`, and `fd_set *exceptset`) define the network events we want to test for. We discuss only the reading and writing events in this book. These parameters are pointers to `fd_set` type of data, which is used to store the notification of a possible network event. After the `select` function is run, we check if the socket is a member of a set by using the macro `FD_SET`. The macros are explained below.

The fifth and last parameter (`const struct timeval *timeout`) defines the timeout value for the function. This is a pointer to a `timeval` structure, which holds two members: seconds and microseconds. To make the function wait forever, we set this parameter to `NULL`. We can set the timeout value to 0 seconds and 0 microseconds. In that case, the function returns immediately after first checking for the network events.

Win32 return values:

- Success: Positive number of ready descriptors, 0 on timeout
- Failure: `SOCKET_ERROR`

Unix return values:

- Success: Positive number of ready descriptors; 0 on timeout
- Failure: `-1`

Macros

The following four macros are used to modify and check the socket descriptor sets:

- `FD_ZERO`
- `FD_SET`
- `FD_CLR`
- `FD_ISSET`

`FD_ZERO` resets a set so that no socket descriptors belong to it. It is a good idea to do this before any other macro is used.

`FD_SET` adds the socket descriptor to the set.
`FD_CLR` removes the socket from the set.
`FD_ISSET` checks if the socket descriptor is a member of the set.

WSAAsyncSelect (Win32)

```
int WSAAsyncSelect(SOCKET s, HWND hWnd, unsigned int wMsg, long lEvent);
```

This function is used to set Windows' kernel to send a message to notify of a network event on a socket. This system can provide information about numerous network events, such as the following:

- `FD_READ` — Ready to read data
- `FD_WRITE` — Ready to write data
- `FD_ACCEPT` — Incoming connection
- `FD_CLOSE` — Socket closing

The first parameter (`SOCKET s`) defines the socket to monitor. This does not have to be a connected socket, because we can also monitor for incoming connections with this function. Note that datagram sockets do not tell about incoming connections — only stream sockets do.

The second parameter (`HWND hWnd`) defines the window handle to which the message is sent. The third parameter (`unsigned int wMsg`) defines the message to send when the event defined by the next parameter occurs. We can (and should) create our own message for this. The fourth parameter (`long lEvent`) defines the network event(s) to monitor. We can define multiple events at once by OR'ing them together here, such as `FD_READ | FD_WRITE`.

Win32 return values:

- Success: 0
- Failure: `SOCKET_ERROR`

WSAEventSelect (Win32)

```
int WSAEventSelect(SOCKET s, WSAEVENT hEventObject, long lNetworkEvents);
```

This function sets an event object to receive a notification of the specified network events. The event object can then be used to see which network event happened, if any. The network events we can specify are exactly the same as in the `WSAAsyncSelect` function.

This function gives us some more breathing room because we do not have to tie the network events to a window. We can specify any number of event objects and then use them wherever we wish.

The first parameter (`SOCKET s`) defines the socket to monitor. The second parameter (`WSAEVENT hEventObject`) defines the event

object handle that will receive the notification of the network events. The third parameter (`long lNetworkEvents`) defines the network events to monitor. As with the `WSAAsyncSelect` function, we can OR multiple events together.

Win32 return values:

- Success: 0
- Failure: `SOCKET_ERROR`

WSAWaitForMultipleEvents (Win32)

```
DWORD WSAWaitForMultipleEvents(DWORD cEvents, const WSAEVENT far *lphEvents,  
                                BOOL fWaitAll, DWORD dwTimeOut, BOOL fAlertable);
```

This function polls for an event to happen and actually tells us if there is a network event that we should process. This function returns either when there is a network event happening (one we are waiting for) or when the timeout value has been reached. We can set this value to infinite so the function returns only when an event occurs.

The first parameter (`DWORD cEvents`) is the number of events to wait for. This is the number of members in the `lphEvents` array (parameter two). At least one must be specified, but we cannot specify more than what `WSA_MAXIMUM_WAIT_EVENTS` specifies.

The second parameter (`const WSAEVENT far *lphEvents`) is a pointer to the array of network event objects. The third parameter (`BOOL fWaitAll`) defines whether the function should wait for all the events to occur before it returns. Possible values are `TRUE` and `FALSE` (1 and 0). If set to `FALSE`, this function returns when at least one event occurs.

The fourth parameter (`DWORD dwTimeOut`) defines the timeout value in milliseconds. If this value is reached, the function returns no matter what, even if the `fWaitAll` flag is set to `TRUE` and the timeout value is reached. The fifth and last parameter (`BOOL fAlertable`) defines whether the function should return if there is an I/O completion routine queued by the system for execution. Possible values are `TRUE` and `FALSE` (1 and 0).

Win32 return values:

- Success: The event object that caused the function to return

Event Object

An event object is a normal Windows handle that is used to store the state of a network event (or any other event). For example, if we want to know when there is data to read on a socket, we create an event

object, set it to inform us of incoming network events, and check its state.

Let's take a look at a more detailed example. First we create the event object handle and a socket:

```
HANDLE readEvent;  
SOCKET s;
```

We presume the socket is initialized properly somewhere. Then we set the event object to receive notification of incoming data:

```
WSAEventSelect(s, readEvent, FD_READ);
```

And finally, when we are ready to read data from a socket, we check if there is anything to read:

```
WSAEVENT EventArray[1];  
EventArray[0] = SocketInputEvent;
```

```
int waitStatus = WSAWaitForMultipleEvents(1, EventArray, FALSE, WSA_INFINITE,  
FALSE);
```

Multithreading

It is safe to say that multithreading is a must in a non-iterative network application. It is the only way to keep all the clients handled properly by the server since there can be multiple clients connected to one server at a time. If each client waited for the server to handle the other clients first, we could call our system “Wait Wait Wait.”

NOTE A UDP server does not need multithreading as much as a TCP server. This is because a UDP client can just “throw in” a message to the server, then wait for a response from the server — and we are done. The server could handle all the clients on one socket, thus removing one reason for multithreading. It depends on the design of the UDP network application whether we should use multithreading or not.

What Is Multithreading?

Before we tell you how to make our application multithreaded, let's take a moment to think about what multithreading really is. Multithreading means that there is more than one process running in one application. These processes run constantly and at the same time. All the threads and the main application share the same memory. This means that if you have a global variable, it can be accessed and modified by each thread. Each process usually has its own loop to keep it running. A normal single-threaded application has only one main loop,

which is the backbone of the application. When this loop breaks, the application terminates. This is the same for every extra thread. When a thread reaches the end of the thread function, it terminates and the thread is destroyed. When the main application ends, so do the threads.

Although the threads seem to be running all at the same time, this is not actually true on a single-processor system. One processor can process only one thing at a time, so the threads are really run one by one, but only a little bit at a time.

An example from real life explains this best. Imagine you have three papers to write. You could write one completely, then write the next one, and so on. But if you wanted to write all the papers at the same time (for some weird reason), you would have to write one word at a time for each paper. First, you write one word on the first paper, then write one on the next paper, and lastly write one on the third paper. You are not really writing them at the same time, but if you do it really fast (I mean *really* fast), it seems like you actually are writing them simultaneously, because the words seem to appear on the papers at the same time. This is exactly the same for multithreading. It is all about speed.

CreateThread (Win32)

```
HANDLE CreateThread(LPSECURITY_ATTRIBUTES lpThreadAttributes, DWORD
                    dwStackSize, LPTHREAD_START_ROUTINE lpStartAddress,
                    LPVOID lpParameter, DWORD dwCreationFlags, LPWORD
                    lpThreadId);
```

The `CreateThread` function creates a new thread on Windows.

The first parameter (`LPSECURITY_ATTRIBUTES lpThreadAttributes`) is a pointer to a security attributes structure. In this book, we always set this to `NULL`. The second parameter (`DWORD dwStackSize`) defines the stack size for the thread. If this is set to 0, the default value is used. This value is the size of the calling thread's stack. The third parameter (`LPTHREAD_START_ROUTINE lpStartAddress`) specifies the thread's routine function.

The fourth parameter (`LPVOID lpParameter`) is a pointer to the parameter that will be passed to the thread's routine function. The fifth parameter (`DWORD dwCreationFlags`) defines the flags for how to create the thread. We can only set this to 0 or `CREATE_SUSPENDED`. If the latter option is used, the thread starts in suspended mode and will not start before the `ResumeThread` function is called. The sixth parameter (`LPWORD lpThreadId`) is a pointer to a variable that will receive the thread identifier.

Win32 return values:

- Success: Handle to the thread
- Failure: NULL

pthread_create (Unix)

```
pthread_create(pthread_t *tid, const pthread_attr_t *attr, void *(*func)
               (void *), void *arg);
```

The `pthread_create` function creates a new thread on Unix. This is similar to Windows' `CreateThread` function.

The first parameter (`pthread_t *tid`) is a pointer to the variable that will receive the thread identifier. The second parameter (`const pthread_attr_t *attr`) is a pointer to the thread attributes variable. Usually we set this parameter to `NULL` to use the defaults. The third parameter (`void *(*func)(void *)`) defines the thread routine function. This is similar to the Windows `CreateThread` function's `lpStartAddress` parameter. The last parameter (`void *arg`) specifies the parameter to pass to the thread routine function. This is similar to the Windows `CreateThread` function's `lpParameter` parameter.

Unix return values:

- Success: 0
- Failure: Non-zero

I/O Strategy

Each network application has its own input/output strategy for how data flow is controlled within the application. For example, will the server open multiple sockets for the clients (UDP) or will it use multithreading? Let's take a look at some of the possible strategies:

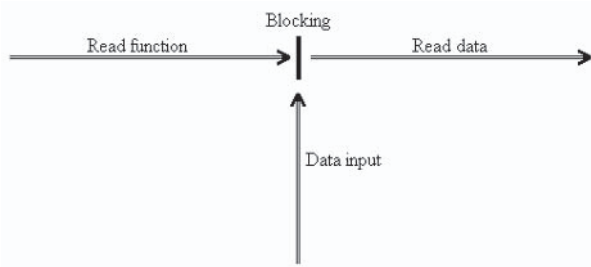
- Blocking I/O
- Non-blocking I/O
- Signal-driven I/O
- Multiplexing I/O

Blocking I/O

Blocking I/O is the simplest form of I/O strategies. As we have already learned, a socket can be blocking or non-blocking. A blocking socket means that some of the socket functions we run, such as passing a blocking socket as a parameter, wait for the action to be accomplished

before they return. For example, if we try to read data off a blocking socket, the read function we use will not return before the data is read. If there is no one sending anything to that socket, the function will block until there is someone sending data to it. Each socket is blocking by default.

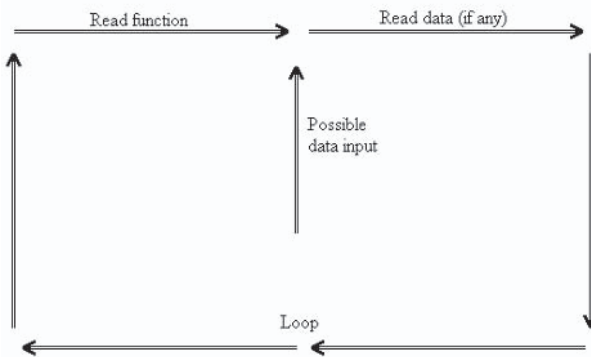
Figure 6-1
Blocking I/O



Non-blocking I/O

We can set a socket to non-blocking if we want to. Then the socket functions will return even if the action cannot be accomplished immediately. Usually this means that we need to loop the function to create our own blocking effect. This is called polling. If we do not poll the incoming data, we would most likely miss it because the data must have reached the local host before we call the function to read the data.

Figure 6-2
Non-blocking I/O

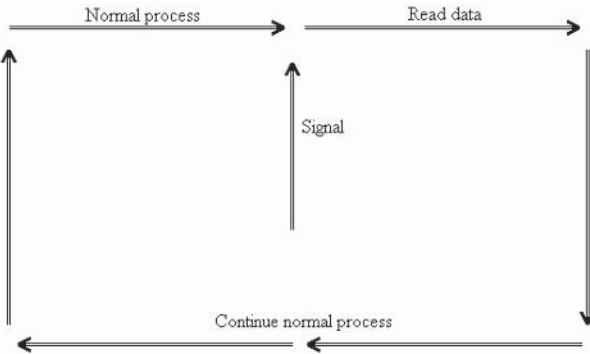


Signal-driven I/O

We do not always have to use the data reading function to monitor the socket for incoming data. We can set up our operating system to signal us when incoming data is available for us to read. Then all we have to do is call the function to read the data from the socket. The obvious advantage in this strategy compared to the two previous ones is that we

can do other things while we wait for the data to come in. The application that waits for data input can run the rest of the application when there is no data to be read. Then when data input exists, everything else is stopped for the time it takes to read the data and possibly process it.

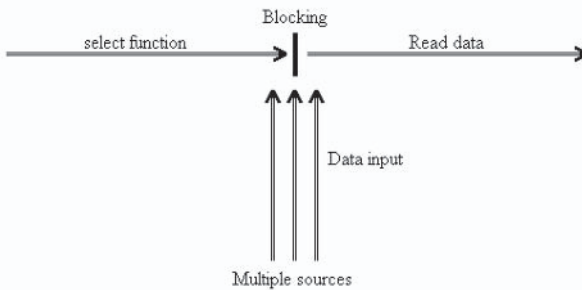
Figure 6-3
Signal-driven
I/O



Multiplexing I/O

Another way to avoid using the actual data reading function to tell if there is incoming data is to use the `select` function. It is used similarly to a blocking socket read function call, as the `select` function will stop and wait for the data to come in. Multiplexing means that we have multiple sources, but we use only one at a time. We need to be able to choose the source that has something to offer us. We use the `select` function for that. This is the advantage of multiplexing I/O; we can wait for more than one socket to have data to read with one function call.

Figure 6-4
Multiplexing
I/O



By combining these I/O strategies with multithreading, we can unleash all the power of our network applications.

I/O Control

Now that we know some of the possible I/O strategies, we need to know how to get our sockets to work using the strategy we choose. Some do not need any extra setting up, but some do. Also, we can create some nice features for our network application by controlling the I/O mode of our sockets. To control the input/output mode of our sockets, we use the following two functions:

- `ioctl/ioctlsocket`
- `setsockopt`

`ioctl (Unix)/ioctlsocket (Win32)`

```
int ioctl(int fd, int request, ... /* void *arg */);
int ioctlsocket(SOCKET s, long cmd, u_long FAR *argp);
```

These two functions work alike; `ioctl` is for Unix and `ioctlsocket` is for Windows.

The functions are used to control the I/O mode of the socket. The most common use for these functions is setting the blocking/non-blocking mode of a socket.

The first parameter (`int fd/SOCKET s`) defines the socket to control. The second parameter (`int request/long cmd`) is the command to give the socket. This is used with the third parameter (`void *arg/u_long FAR *argp`), which tells the function how to give out the command.

The Windows version (`ioctlsocket`) does not have all the commands that the Unix version (`ioctl`) has. The only command we really use within this book is `FIONBIO`, which sets/clears the non-blocking flag of the socket.

Win32 return values:

- Success: 0
- Failure: `SOCKET_ERROR`

Unix return values:

- Success: 0
- Failure: `-1`

An example best explains the usage of this function. To set a socket into non-blocking mode, we call the I/O control function like this:

```
u_long on = 1;

ioctl(mysocket, FIONBIO, &on);           // Unix
ioctlsocket(mysocket, FIONBIO, &on);    // Windows
```

To set the non-blocking mode off:

```
u_long off = 0;

ioctl(mysocket, FIONBIO, &off);           // Unix
ioctlsocket(mysocket, FIONBIO, &off);     // Windows
```

setsockopt/getsockopt (Unix, Win32)

```
int setsockopt(int sockfd, int level, int optname, const void *optval,
               socklen_t optlen);
```

```
int getsockopt(int sockfd, int level, int optname, void *optval,
               socklen_t *optlen);
```

These functions are used to set and get the socket options. There are numerous socket options available, but we discuss only a few in this book.

The first parameter of both functions (`int sockfd`) defines the socket to set the option for. The second parameter of both functions (`int level`) specifies the level at which the options are defined. Possible values (for compatibility issues) are `SOL_SOCKET` and `IPPROTO_TCP`. The third parameter is also the same for both functions (`int optname`); it defines the actual option to set.

The fourth parameter of `setsockopt` (`const void *optval`) is a pointer to the buffer where the value for the option is stored. The fourth parameter of `getsockopt` (`void *optval`) is a pointer to the buffer where the value of the option will be stored.

The fifth parameter of both functions (`socklen_t optlen/socklen_t *optlen`) defines the size of the buffer used in parameter four.

In this book we use only the `SOL_SOCKET` level, as the other levels are beyond the scope of this book. Here are some options from the `SOL_SOCKET` level that we should be aware of:

- `SO_BROADCAST` — Set/get broadcasting on/off
- `SO_LINGER` — Set/get lingering on/off
- `SO_RCVBUF` — Set/get receive buffer size
- `SO_SNDBUF` — Set/get send buffer size
- `SO_TYPE` — Get socket type

Win32 return values:

- Success: 0
- Failure: `SOCKET_ERROR`

Unix return values:

- Success: 0
- Failure: -1

Let's take a look at an example of how to make our socket linger on a call to `close/closesocket` to make sure all the data is sent before the socket is closed:

```
struct linger Ling;

Ling.l_onoff = 1;
Ling.l_linger = 0;

setsockopt(mysocket, SOL_SOCKET, SO_LINGER, (const char *) &Ling,
           sizeof(struct linger));
```

When we want to close the socket, we should put the `close/closesocket` function call in a loop that loops as long as the function returns successfully:

```
shutdown(mysocket, SD_BOTH);           // Windows

int ret = WSAEWOULDBLOCK;

while(ret == WSAEWOULDBLOCK)
    ret = closesocket(mysocket);      // Windows

--

shutdown(mysocket, SD_RDWR);          // Unix

int ret = WSAEWOULDBLOCK;

while(ret == WSAEWOULDBLOCK)
    ret = close(mysocket);           // Unix
```

shutdown (Unix, Win32)

```
int shutdown(int sockfd, int howto);
```

This function is used to disable sending and receiving on a socket. It does not close the socket, but depending on the parameters we pass to it, the socket will not be able to send or receive (or both) any more data.

The first parameter (`int sockfd`) defines the socket to shut down. The second parameter (`int howto`) specifies how to shut down the socket. Possible values are listed in Table 6-1.

Table 6-1: howto parameter values

OS	Shut Reading	Shut Writing	Shut Both
Unix	SHUT_RD	SHUT_WR	SHUT_RDWR
Windows	SD_RECEIVE	SD_SEND	SD_BOTH

Win32 return values:

- Success: 0
- Failure: SOCKET_ERROR

Unix return values:

- Success: 0
- Failure: -1

Broadcasting

Broadcasting is a very useful way of sending data. When you broadcast, you send to everybody on the network with only one call to the sending function. There are some restrictions however, which make broadcasting much less interesting than it first sounds. First of all, it can be used only with datagram sockets. This means that we need to use UDP if we want to broadcast. The second, and much more limiting, issue is that it can be used only on a local area network. We cannot broadcast messages through the Internet, because broadcasting works using a unique broadcast IP address that every LAN has. Usually it is of the form subnet.255. For example if our subnet is 192.168.0, the broadcast address is 192.168.0.255. We can also use a “global” broadcast address, 255.255.255.255, that works on all LANs.

NOTE IPv6 does not support broadcasting. IPv6 uses multicasting instead, but that is beyond the scope of this book.

Searching for Servers

One very good use for broadcasting is to search for available servers on a LAN. Anyone who has played a network game on a LAN knows that it is annoying to have to continually enter the IP address of a certain server to be able to connect to it. And if you do not know what servers there are to join, you have to find that out beforehand. Broadcasting is a solution for this. We assume that all the servers use the same port and that the client application (the game) knows this port number. Now all we need to do is broadcast to that port number and wait for the servers

to send their info to us. All the servers that are up and running in the LAN will receive the broadcast message. They can find out the client's address from the datagram and send a reply to that address. The client will then build a list of servers that replied in a certain amount of time.

Broadcast Function

To make our socket broadcast, we need to set the socket option of it as shown in the following code. We also need to create a datagram socket and fill in the address information so that the data is sent to the broadcast address. In this example, we use the address 255.255.255.255, which works on all LANs as a broadcast address.

```
void Broadcast(char *buf, size_t count)
{
    // Define on
    const int on = 1;

    // Create a datagram socket
    SOCKET sock;
    sock = socket(AF_INET, SOCK_DGRAM, 0);

    struct sockaddr_in servaddr;
    socklen_t len;

    // Use the example broadcast address
    u_long inetAddr = inet_addr("255.255.255.255");

    // Fill address information structure
    memset(&servaddr, 0, sizeof(struct sockaddr_in));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(9009);
    servaddr.sin_addr.s_addr = inetAddr;

    len = sizeof(servaddr);

    // Set socket broadcasting option on
    setsockopt(sock, SOL_SOCKET, SO_BROADCAST,
               (const char *) &on, sizeof(on));

    // Broadcast!
    sendto(sock, buf, count, 0,
           (struct sockaddr *) &servaddr, len);
}
```

Now let's discuss the code.

First, we must define the socket option switch in a variable because the `setsockopt` function always retrieves the switch from a variable.

```
// Define on
const int on = 1;
```

Then we create a normal datagram socket to be used in the broadcasting operation.

```
// Create a datagram socket
SOCKET sock;
sock = socket(AF_INET, SOCK_DGRAM, 0);
```

Next, the address information structure must be filled with the correct information. The IP address is set to the example broadcast address — 255.255.255.255. The port number is set to 9009, assuming that is the port that all the recipients use.

```
// Use the example broadcast address
u_long inetAddr = inet_addr("255.255.255.255");

// Fill address information structure
memset(&servaddr, 0, sizeof(struct sockaddr_in));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(9009);
servaddr.sin_addr.s_addr = inetAddr;
```

And finally, the socket option `SO_BROADCAST` is set on and we broadcast the message using the normal `sendto` function.

```
// Set socket broadcasting option on
setsockopt(sock, SOL_SOCKET, SO_BROADCAST,
           (const char *) &on, sizeof(on));

// Broadcast!
sendto(sock, buf, count, 0,
       (struct sockaddr *) &servaddr, len);
```

Summary

In this chapter we learned what I/O strategies we can use in our network applications and how to implement them. We also learned the concept of multithreading and the reasons to use it. Along with the I/O methods, we learned a new way to send data efficiently in a LAN — by broadcasting. We are now ready to create our own network library.

This page intentionally left blank.

Tutorials

The tutorial section leads you step by step through how to create a working online game. Since we wish to focus on the network programming side, we have supplied you with a simple OpenGL-based 2D graphics library.

The first tutorial consists of an introduction to our graphics library. It explains the basics of the library and also gives you code examples to experiment with. Then we move on to developing our network library, which is reusable so that it can be used in your own online titles as well as in our sample online game. Once we have the foundations, we then move on to how to implement a login and lobby system that is ready for implementing our online game. In the final tutorial, we develop our final working online game that you can both learn from and expand upon.

In creating these tutorials, we have aimed to give you a practical hands-on approach to learning how to create an online game. We have found that other books tell you more about how things work rather than explain how to actually get it working so you can experiment with it.

Remember that all the source code for the tutorials can be found on the companion CD.

This page intentionally left blank.

Tutorial 1

Using 2DLIB

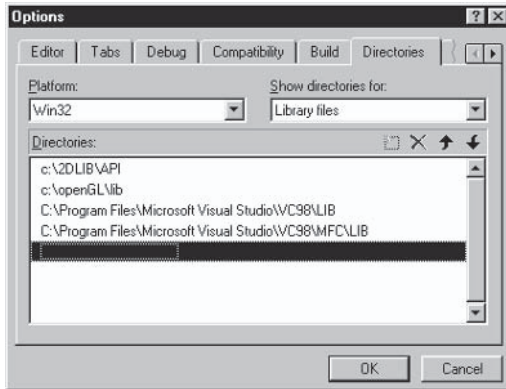
Introduction

In this first tutorial, we cover the use of our simple 2D OpenGL library that we will be using in the following tutorials to create our online network game. The reason we are covering this is to keep confusing graphics code from getting in the way of the core network code, making the network code easier for us to understand. You will quickly be able to understand and use our 2D library, but if you prefer, you can use your own DirectX or OpenGL routines.

Configuring Visual Studio

The OpenGL library and our 2D library are located on the companion CD. First, you must add the 2D library's API directory into the Visual Studio environment. To do this select Tools, Options... from the main Visual Studio menu. Now select the Directories tab on the dialog box that is visible in the center of the screen.

Figure 1



Next, add the library's API directory to both the Include files and Library files. This will allow Visual Studio to find the 2D library when you try to compile a program that uses it.

Finally, you need to add the OpenGL library into Visual Studio. You do this in the same way except you need to add the Include folder into the Include directories and the Library folder into the Library directories.

If you would like additional information on the creation and use of static link libraries, see Chapter 1.

Creating a Skeleton Project

It is generally a good idea to create basic code that never changes so you can reuse it in future programs and projects. Here we create a simple windowed program using the 2D graphics library that will create a window on the screen. This skeleton can be developed further into any Windows application, as you will see throughout this online game tutorial.

First, we must create a workspace for our project. This entails creating a project, adding our 2D library and OpenGL into the workspace, and finally adding a main source file to code in.

Creating the Workspace

To create our workspace, select File, New... from the main Visual Studio menu. Next, click the Projects tab and select Win32 Application (not a console application). Set where you wish the project to be created (i.e., c:\projects\) and give the project a name (i.e., OnlineGame). Once you click OK, the next window will ask which type of application you would like to create. For this project, leave it as an empty project and

click Finish. Simply click OK on the next screen, as it is simply a summary of what you have just done.

Now that we have our workspace created, we need to add the 2D library into the workspace so it is compiled with our code (so the functions are available to us).

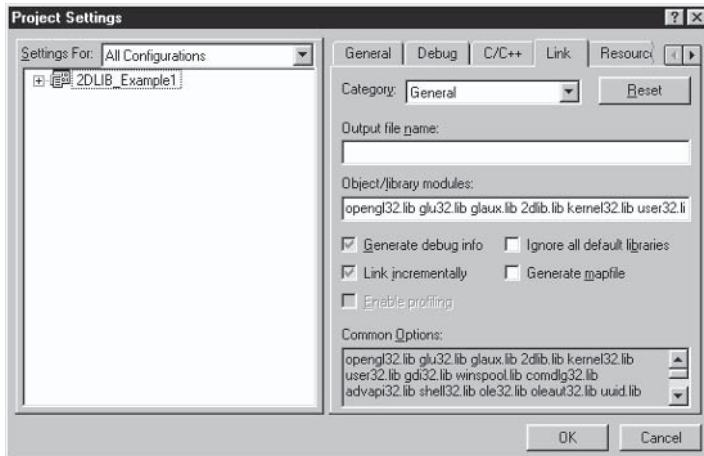
Adding the Static Libraries

To add the static libraries, select Project, Settings from the main Visual Studio menu. On the dialog box that appears, change the drop-down box at the top left from Win32 Debug to All Configurations. This means that when we add in the library it will be included in both the debug and release versions of our game code. Next select the Link tab on the right of the dialog and add the following *before* kernel32.lib:

```
opengl32.lib glu32.lib glaux.lib 2dlib.lib
```

This will add the required OpenGL libraries and our 2D library into the workspace when we compile the program.

Figure 2



Adding the Source File

Finally, we need to add our main source file into the project. To do this, select File, New... from the main Visual Studio menu and select the Files tab on the dialog that appears. Now select C++ Source File and enter "main" in the File name edit box. Click OK and the main source file will be added into our project.

Creating a Basic Windowed Application with 2DLIB

Our workspace is now ready to have code added. For the 2D library to function, we need to create a Windows message loop and a callback function for the window, which is passed into the 2D library as a parameter. Now we will cover the main parts of the main program. The complete code can be found at the end of this section.

The WinMain Function

The following code segment is the main Windows message loop, but it also contains our initialization code and main drawing loop.

```
int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst, LPSTR lpszArgs,
                  int nWinMode)
{
    MSG msg;

    GFX_Init("App Title", 640, 480, 16, 0, WndProc);

    while(msg.message != WM_QUIT)
    {
        if(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        else
        {
            GFX_Begin();
            {
            }
            GFX_End();
        }
    }

    GFX_Shutdown();
    return msg.wParam;
}
```

The `GFX_Init()` function sets up our window for use by the 2D graphics library. The first parameter is the title of the window, although this is not relevant if you wish to create a full-screen application. The next two parameters determine the width and height of the window in pixels. The fourth specifies the color depth you wish to use; usually this is set to 16 bits per pixel, but other values such as 24 or 32 can be used too. Next is a flag to determine whether you wish to run the

application full screen or not; set this to 0 for windowed or 1 for full-screen mode. The final parameter is a pointer to the Windows procedure function that is explained in the next section.

```
GFX_Init("App Title", 640, 480, 16, 0, WndProc);
```

Next is the message loop for Windows; this controls all the Windows messages that are required to allow the program to run correctly in the Windows environment.

```
while(msg.message != WM_QUIT)
{
    if(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    else
    {
        GFX_Begin();
        {
        }
        GFX_End();
    }
}
```

As you can see in the preceding code, the Windows messaging loop is a simple while loop that continues until a quit message is received. The `PeekMessage` function is a faster version of the `GetMessage` function and is the optimal way to get Windows messages. This is not important for us, but the else part of the if statement is, however. This is where we place all our 2D drawing commands; it is, if you like, our game loop.

Once the 2D library is initialized using the `GFX_Init()` function, you must call `GFX_Begin()` before you start drawing and then `GFX_End()` once everything is drawn. The first function is used to clear the drawing buffer and ready it for the next frame of graphics to be drawn to it. The ending function is used to swap the buffer onto the visible screen so the user can see it without any shearing or other nasty graphical glitches.

Finally, once the user quits the program, the graphics library must be shut down. To do this, we simply call `GFX_Shutdown()` after the while loop. This closes the graphics library and frees any memory that was allocated internally by the 2D graphics library.

The Windows Procedure

Every time our window receives a Windows message, such as a key being pressed on the keyboard, this function is called to process the message correctly. The code that follows is the entire Windows procedure.

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_CLOSE:
        {
            PostQuitMessage(0);
            return 0;
        }

        case WM_KEYDOWN:
        {
            keys[wParam] = TRUE;
            return 0;
        }

        case WM_KEYUP:
        {
            keys[wParam] = FALSE;
            return 0;
        }

        case WM_SIZE:
        {
            GFX_Resize(LOWORD(lParam),HIWORD(lParam));
            return 0;
        }
    }

    // Pass all unhandled messages to DefWindowProc
    return DefWindowProc(hWnd,uMsg,wParam,lParam);
}
```

In basic terms, the function is simply a switch statement that reacts correctly to different events (cases) in Windows. If the function has no handling routine for an event, it simply passes it back to Windows to deal with accordingly.

It is possible to add other event handles to this procedure, but the four we have included in this skeleton application are all that are required for now. Let's take a look at the events in a little more detail.

The WM_CLOSE Event

```
case WM_CLOSE:
{
    PostQuitMessage(0);
    return 0;
}
```

This event is triggered when the user clicks on the X button at the top right of a window or when a close message is manually sent to the window. When a WM_CLOSE message is sent, this routine sends a WM_QUIT message that, if you remember from earlier, is the condition for

our main while loop (i.e., when a `WM_QUIT` message is received, our program will shut down and exit).

The `WM_KEYDOWN` Event

```
case WM_KEYDOWN:
{
    keys[wParam] = TRUE;
    return 0;
}
```

In the 2D library, we have defined an array that allows for easy use of the keyboard without having to learn `DirectInput` or another similar input library. When a key has been pressed on the keyboard, this routine sets the correct key in the array to `TRUE`, meaning that the key is currently down. Later in this tutorial we will cover how to use the keyboard for input.

The `WM_KEYUP` Event

```
case WM_KEYUP:
{
    keys[wParam] = FALSE;
    return 0;
}
```

This routine works in the same way as the `WM_KEYDOWN` routine except that it handles the event of a key being released on the keyboard. When the key has been released, it sets the correct value in the keys array to `FALSE`, meaning the key is not being pressed.

The `WM_SIZE` Event

```
case WM_SIZE:
{
    GFX_Resize(LOWORD(lParam),HIWORD(lParam));
    return 0;
}
```

This event handles the resizing of a windowed mode application. It is not relevant for full-screen applications, but it does no harm to leave it in. It tells the 2D library the new width and height of the window so that it can react accordingly. (The width is the low word of the `lParam` and the height is the high word of the `lParam`.)

The Complete Code

The final part we require is what files to include. We need to include the Windows header file (since we are creating a Windows application), the three OpenGL header files for the 2D library, and the 2D library's header files. Therefore, our complete code listing for the skeleton application resembles the following:

```
#include <windows.h>
#include <gl/gl.h>
#include <gl/glu.h>
#include <gl/glaux.h>
#include <2dlib.h>

// WINDOWS PROCEDURE
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_CLOSE:
        {
            PostQuitMessage(0);
            return 0;
        }

        case WM_KEYDOWN:
        {
            keys[wParam] = TRUE;
            return 0;
        }

        case WM_KEYUP:
        {
            keys[wParam] = FALSE;
            return 0;
        }

        case WM_SIZE:
        {
            GFX_Resize(LOWORD(lParam),HIWORD(lParam));
            return 0;
        }
    }

    // Pass all unhandled messages to DefWindowProc
    return DefWindowProc(hWnd,uMsg,wParam,lParam);
}

// WINDOWS MESSAGE LOOP AND APPLICATION ENTRY POINT
int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst, LPSTR lpszArgs,
                  int nWinMode)
{
    MSG msg;

    GFX_Init("Skeleton App", 640, 480, 16, 0, WndProc);

    while(msg.message != WM_QUIT)
    {
```

```
        if(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        else
        {
            GFX_Begin();
            {
            }
            GFX_End();
        }
    }
    GFX_Shutdown();
    return msg.wParam;
}
```

If everything has gone according to plan, when you compile and execute the preceding code you should be able to see the following screen:

Figure 3



Not very exciting, is it? However, it is very useful as we now have the foundation for our 2D graphics engine, meaning we can easily use the 2D library to create primitives and bitmapped graphics on the screen.

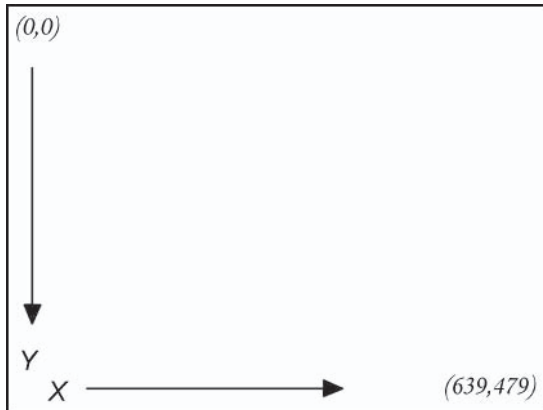
Using the 2DLIB Graphics Routines

Now we will cover all the graphical functions in the 2D library with an explanation of how to use them. The final sections of this tutorial demonstrate a couple of examples of how to use these functions with the skeleton application. Remember that all graphics functions must be placed between the `GFX_Begin()` and `GFX_End()` functions or the graphics will not be visible on the screen.

2D Positions on the Screen

For the x coordinates, the values start from 0 at the left-hand side of the application and end at the width of your application minus 1. The y coordinates start from 0 at the top of your application and end at the height of your application minus 1. Figure 4 displays what the screen coordinates would be like for a 640 x 480 application in the format (x, y).

Figure 4



Use of Colors

Each primitive drawing function in the library has three parameters at the end that specify the red, green, and blue values for what you wish to draw. These values range as integers from 0 to 255, where 0 is black and 255 is full brightness. Therefore, if you wanted your color to be bright red, you would use 255 red, 0 green, and 0 blue.

Plotting a Single Pixel

```
void GFX_Pixel(int x, int y, int r, int g, int b);
```

The `GFX_Pixel()` function will display a single pixel on the screen at the specified (x, y) position. The position is specified by the first two parameters of the function, the first being x and the second being the y position. The final three parameters determine the color of the pixel in RGB format. (See the information about colors in the “Use of Colors” section.)

Drawing a Line

```
void GFX_Line(int x1, int y1, int x2, int y2, int r, int g, int b);
```

The `GFX_Line()` function draws a line between two points. The first two parameters are the (x, y) position of the first point. The next two parameters are the (x, y) position of the point to connect the first point to. The final three are to specify the color in RGB format.

Drawing a Rectangle/Filled Rectangle

```
void GFX_Rect(int x1, int y1, int x2, int y2, int r, int g, int b);
```

```
void GFX_RectFill(int x1, int y1, int x2, int y2, int r, int g, int b);
```

The `GFX_Rect()` and `GFX_RectFill()` functions draw a rectangle based on the two points you specify. The first two parameters determine the top-left corner for the rectangle and the next two specify the bottom-right corner for the rectangle. Again, the final three determine the color in RGB format.

The first function draws an outlined rectangle in the specified color, whereas the second draws a filled rectangle in the specified color.

Drawing a Triangle/Filled Triangle

```
void GFX_Tri(int x1, int y1, int x2, int y2, int x3, int y3, int r, int g,  
            int b);
```

```
void GFX_TriFill(int x1, int y1, int x2, int y2, int x3, int y3, int r,  
               int g, int b);
```

The `GFX_Tri()` and `GFX_TriFill()` functions allow you to draw triangles and filled triangles on the screen. The first six parameters determine the three two-dimensional coordinates that are required to construct a triangle. The final three then determine the red, green, and blue values the same way as the other primitive functions.

Graphic Loading Functions

Our 2D library has support for Windows bitmaps and 24-bit and 32-bit uncompressed Truevision Targa format. First, we need to declare a variable that will hold our graphic data. You do this as follows:

```
GFX_IMAGE2D my_image;
```

This creates a variable called `my_image` that you can use with the graphics loading commands in the library to load pictures from your hard drive in the application.

Next, you want to actually load an image. Let's say we have a Windows bitmap image called `brick.bmp`. We would call the following function to load it into our `my_image` variable. Notice you pass a pointer to the `GFX_IMAGE2D` variable, followed by the filename of the graphic you wish to load.

```
GFX_LoadBitmap(&my_image, "brick.bmp");
```

The Truevision Targa loading function works in the same manner. If you have an image called `phone.tga`, you would load it like this:

```
GFX_LoadTarga(&my_image, "phone.tga");
```

That is all there is to loading graphics. Here are the prototypes for the graphics loading functions for your reference:

```
void GFX_LoadBitmap(GFX_IMAGE2D *pImage, char *filename);
```

```
void GFX_LoadTarga(GFX_IMAGE2D *pImage, char *filename);
```

Graphics Display (Blitting) Function

Once you have successfully loaded in your graphics, you use the following function to display them on the screen:

```
void GFX_Blit(GFX_IMAGE2D *pImage, int x, int y, int w, int h, float rotate);
```

The first parameter is a pointer to a `GFX_IMAGE2D`. You simply pass a pointer to a graphic you have loaded using one of the loading functions explained previously. Next you specify the (x, y) position using the second and third parameters of the function. The function also handles image scaling so you can state the width and height at which the image is to be drawn on the screen. These are set in the fourth and fifth parameters, respectively. Finally, the function handles image rotation in degrees. Therefore, you can specify a floating-point value between 0.0 and 360.0 for the image to be drawn at (the image rotates around its center point).

For example, if you wanted to display an image you loaded into a `my_image` variable at point (25, 50) with a width of 100, a height of 150, and no rotation, you would call the following function:

```
GFX_Blit(&my_image, 25, 50, 100, 150, 0.0);
```

If you then decided you wanted it in the same position and with the same dimensions but at an angle of 30 degrees, the function would be as follows:

```
GFX_Blit(&my_image, 25, 50, 100, 150, 30.0);
```

That is all there is to the graphical side of the library. As you can see, there is nothing very complicated in it, and it simplifies many of the initialization procedures and other such things.

Keyboard Input Method

This is a very simple method of getting key input. We would really recommend using a library such as `DirectInput`, but this makes the process simple and it does what we require it to do for now.

To check a key on the keyboard, simply place an if statement after the `GFX_End()` function as follows:

```
if(keys[VkKeyScan('a')])
{
    // place what is to be done here
}
```

The preceding if statement would check if the “a” key had been pressed. The “a” can be replaced with any other character, symbol, or number from the keyboard. For example, if you wished to check if the “]” key had been pressed you would use this statement:

```
if(keys[VkKeyScan(']')])
{
    // place what is to be done here
}
```

There are special cases such as the function keys (F1, F2, etc.) and other keys like the Spacebar and Return. Table 1 lists most of these special cases and an if statement follows that shows how to use a special key.

Table 1: Special keys

Statement	Description
VK_F1	Function key F1
VK_F2	Function key F2
VK_F3	Function key F3
VK_F4	Function key F4
VK_F5	Function key F5
VK_F6	Function key F6
VK_F7	Function key F7
VK_F8	Function key F8
VK_F9	Function key F9
VK_F10	Function key F10
VK_F11	Function key F11
VK_F12	Function key F12
VK_INSERT	Insert key
VK_HOME	Home key
VK_PAGE_UP	Page Up key
VK_DELETE	Delete key
VK_END	End key
VK_PAGE_DOWN	Page Down key
VK_UP	Arrow key up
VK_DOWN	Arrow key down
VK_LEFT	Arrow key left
VK_RIGHT	Arrow key right
VK_ESCAPE	Escape key
VK_SPACE	Spacebar
VK_CONTROL	Control key
VK_ALT	Alt key
VK_ADD	Numeric keypad + key
VK_SUBTRACT	Numeric keypad – key
VK_MULTIPLY	Numeric keypad * key
VK_DIVIDE	Numeric keypad / key

Statement	Description
VK_EQUALS	= key
VK_TAB	Tab key

Unlike the other if statements where you use a function (`VkKeyScan`) to get the correct value for the character, you simply place the `VK_` value as follows:

```
if(keys[VK_SPACE])
{
    // place what is to be done here
}
```

The preceding example would check to see if the Spacebar had been pressed; if so, it would execute the code within the if statement.

2DLIB Example 1 — Moving Primitives with the Cursor Keys

In this first example application, we create a program that allows the user to move a filled rectangle around the screen using the cursor keys, but not outside the borders of the application. In addition, the user will be able to select whether the rectangle is red, green, or blue by pressing r, g, or b, respectively. The complete code listing for this example is at the end of this section.

First, we must set up a project file for this application. This is done in exactly the same manner as the skeleton application that was explained earlier in this tutorial. Once we have the skeleton application running, the first thing we must do is create variables to store the position, dimensions, and color of the rectangle. We declare these globally as follows:

```
int rect_x, rect_y;    // (x, y) position
int rect_w, rect_h;   // width and height

float rect_r;         // red color
float rect_g;         // green color
float rect_b;         // blue color
```

Next, after the initialization code for 2DLIB, we want to set these variables for the rectangle. This is done as follows:

```
rect_x = 10;          // x position is 10
rect_y = 10;          // y position is 10
rect_w = 100;         // width is 100 pixels
rect_h = 100;         // height is 100 pixels
```

```

rect_r = 255.0;    // red is full intensity
rect_g = 0.0;     // no blue
rect_b = 0.0;     // no green

```

Now, when we draw our rectangle it will appear at position (10, 10) and be 100 pixels in both height and width. In addition, we have set the red value to the maximum and the green and blue to the minimum, making the rectangle appear a bright red color. In the drawing loop, we will then place the following function to draw it:

```

GFX_RectFill(rect_x, rect_y, rect_x+rect_w, rect_y+rect_h, rect_r, rect_g,
              rect_b);

```

This will draw the filled rectangle at the position we specified, using the values from the variables we previously declared. For more information on this function, see the “Using the 2DLIB Graphics Routines” section.

Finally, we wish to be able to move the rectangle around the screen, so we need to have the correct keyboard code to do this. Here is what the movement code should look like:

```

if(keys[VK_UP])
{
    if(rect_y > 0)
        rect_y--;
}
if(keys[VK_DOWN])
{
    if(rect_y+rect_h < 480)
        rect_y++;
}
if(keys[VK_RIGHT])
{
    if(rect_x+rect_w < 640)
        rect_x++;
}
if(keys[VK_LEFT])
{
    if(rect_x > 0)
        rect_x--;
}

```

This simply adjusts the variables for the rectangle based on which arrow key the user presses on the keyboard. The colors work in a similar fashion, as shown below:

```

if(keys[VkKeyScan('r')])
{
    rect_r = 255.0;
    rect_g = 0.0;
    rect_b = 0.0;
}
if(keys[VkKeyScan('g')])

```

```

{
    rect_r = 0.0;
    rect_g = 255.0;
    rect_b = 0.0;
}
if(keys[VkKeyScan('b')])
{
    rect_r = 0.0;
    rect_g = 0.0;
    rect_b = 255.0;
}

```

This code checks whether r, g, or b has been pressed, and if so, changes the rectangle variables to set the specified color.

That covers everything for our first example. When you compile the following code, you should be able to move the rectangle around with the arrow keys and change the color with the r, g, and b keys.

Complete Code Listing for Example 1

```

#include <windows.h>
#include <gl/gl.h>
#include <gl/glu.h>
#include <gl/glaux.h>
#include <2dlib.h>

// APPLICATION SPECIFIC

int rect_x, rect_y;    // (x, y) position
int rect_w, rect_h;   // width and height

float rect_r;         // red color
float rect_g;         // green color
float rect_b;         // blue color

// WINDOWS PROCEDURE

LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_CLOSE:
        {
            PostQuitMessage(0);
            return 0;
        }

        case WM_KEYDOWN:
        {
            keys[wParam] = TRUE;
            return 0;
        }
    }
}

```

```

        case WM_KEYUP:
        {
            keys[wParam] = FALSE;
            return 0;
        }

        case WM_SIZE:
        {
            GFX_Resize(LOWORD(lParam),HIWORD(lParam));
            return 0;
        }
    }

    // Pass all unhandled messages to DefWindowProc
    return DefWindowProc(hWnd,uMsg,wParam,lParam);
}

// WINDOWS MESSAGE LOOP AND APPLICATION ENTRY POINT

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst, LPSTR lpszArgs,
                  int nWinMode)
{
    MSG msg;

    // Initialize 2DLIB
    GFX_Init("2DLIB - Example 1", 640, 480, 16, 0, WndProc);

    // Set up the rectangle
    rect_x = 10;      // x position is 10
    rect_y = 10;      // y position is 10
    rect_w = 100;     // width is 100 pixels
    rect_h = 100;     // height is 100 pixels

    rect_r = 255.0;   // red is full intensity
    rect_g = 0.0;     // no blue
    rect_b = 0.0;     // no green

    // Start the Windows message loop
    while(msg.message != WM_QUIT)
    {
        if(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        else
        {
            GFX_Begin();
            {
                // Draw the rectangle
            }
        }
    }
}

```

```
        GFX_RectFill(rect_x, rect_y, rect_x+rect_w,
                    rect_y+rect_h, rect_r, rect_g, rect_b);
    }
    GFX_End();
}

// Check for keyboard input

// -> Movement
if(keys[VK_UP])
{
    if(rect_y > 0)
        rect_y--;
}
if(keys[VK_DOWN])
{
    if(rect_y+rect_h < 480)
        rect_y++;
}
if(keys[VK_RIGHT])
{
    if(rect_x+rect_w < 640)
        rect_x++;
}
if(keys[VK_LEFT])
{
    if(rect_x > 0)
        rect_x--;
}

// -> Colors
if(keys[VkKeyScan('r')])
{
    rect_r = 255.0;
    rect_g = 0.0;
    rect_b = 0.0;
}
if(keys[VkKeyScan('g')])
{
    rect_r = 0.0;
    rect_g = 255.0;
    rect_b = 0.0;
}
if(keys[VkKeyScan('b')])
{
    rect_r = 0.0;
    rect_g = 0.0;
    rect_b = 255.0;
}
}
GFX_Shutdown();
return msg.wParam;
}
```


2DLIB Example 2 — Loading and Rotating Graphics

In this example we load a single bitmap file, then display it twice on the screen and make the two copies rotate in opposite directions.

The first thing we must do is create a skeleton project as described earlier in this tutorial. Once we have this in place, we can begin by creating one variable that will hold the image and another to hold the current angle in degrees.

```
GFX_IMAGE2D cdrom;  
float rotation;
```

The preceding declarations are global. The first is used to store the image in a variable called `cdrom`, and the second is used to store the angle in degrees at which the image is to be rotated. To spin the second image in the opposite direction, we simply subtract the rotation value from 360 degrees.

Next, we need to load our graphic file into the application. In this case, the file is an image of a cdrom and is called `cdrom.bmp`. Therefore, to load it we would want to call the `GFX_LoadBitmap()` function as follows:

```
GFX_LoadBitmap(&cdrom, "cdrom.bmp");
```

This would load the `cdrom.bmp` into the `cdrom` global variable. This is done after we initialize the 2D library. Once this is done we can display the image on the screen using the following technique:

```
GFX_Blit(&cdrom, 32, 100, 256, 256, rotation);  
GFX_Blit(&cdrom, 352, 100, 256, 256, 360-rotation);
```

These two functions are placed between the `GFX_Begin()` and `GFX_End()` functions as they display the image onto the screen. The first image is displayed at (32, 100) and the second is displayed at (352, 100). Both images are 256 x 256 pixels. Additionally, the first is rotated clockwise and the latter is rotated counterclockwise.

The only thing left to do now is adjust the `rotation` variable every time through the loop. This is simply done as follows:

```
rotation++;  
if(rotation>360)  
{  
    rotation -= 360;  
}
```

When the rotation value becomes greater than 360 degrees, 360 is subtracted from the value.

That completes the second example. When you compile and execute the following code listing, you should see two images of CD-ROMs displayed on the screen spinning in opposite directions.

Complete Code Listing for Example 2

```
#include <windows.h>
#include <gl/gl.h>
#include <gl/glu.h>
#include <gl/glaux.h>
#include <2dlib.h>

// APPLICATION SPECIFIC

GFX_IMAGE2D cdrom; // variable to hold 'cdrom' graphic
float rotation; // variable to hold current rotation value

// WINDOWS PROCEDURE
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_CLOSE:
        {
            PostQuitMessage(0);
            return 0;
        }

        case WM_KEYDOWN:
        {
            keys[wParam] = TRUE;
            return 0;
        }

        case WM_KEYUP:
        {
            keys[wParam] = FALSE;
            return 0;
        }

        case WM_SIZE:
        {
            GFX_Resize(LOWORD(lParam), HIWORD(lParam));
            return 0;
        }
    }

    // Pass all unhandled messages to DefWindowProc
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
```

```
}

// WINDOWS MESSAGE LOOP AND APPLICATION ENTRY POINT
int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst, LPSTR lpszArgs,
                  int nWinMode)
{
    MSG msg;

    // Initialize 2DLIB
    GFX_Init("2DLIB - Example 1", 640, 480, 16, 0, WndProc);

    // Load in the image
    GFX_LoadBitmap(&cdrom, "cdrom.bmp");

    // Start the Windows message loop
    while(msg.message != WM_QUIT)
    {
        if(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        else
        {
            GFX_Begin();
            {
                GFX_Blit(&cdrom, 32, 100, 256, 256, rotation);
                GFX_Blit(&cdrom, 352, 100, 256, 256, 360-rotation);
            }
            GFX_End();

            rotation++;
            if(rotation>360)
            {
                rotation -= 360;
            }
        }

        // Check for keyboard input
    }
    GFX_Shutdown();
    return msg.wParam;
}
```

Summary

In this tutorial, you learned how to create a simple 2D graphics application using our 2D OpenGL library. This knowledge is sufficient for you to understand the network game programming tutorials that are to follow, although we would recommend learning more about OpenGL and Direct3D, as they are very useful pieces of knowledge. On a final note, it is also possible to add OpenGL commands directly. The library creates an OpenGL window with a two-dimensional orthographic view; therefore you could, for example, change it to a perspective view and create a 3D application, but this is beyond the scope of these tutorials. The network code, however, is not tied to any one graphics library, so you can implement any style of game using the same network library whether it is 2D or 3D.

This page intentionally left blank.

Tutorial 2

Creating Your Network Library

Introduction

In this tutorial we learn how to create the most essential part of our tutorial game: the network library. We learn how to send and receive data, how to ping hosts, and so on. We make a library of all the network code in order to gain access to it in every application we want. In the case of our tutorial game, we use the library in the game itself (client software) and in the server. Also, if we want to make any extra servers with functionality different from the normal game server, we would most likely want to use the same library for that too.

We create the library so that it can be used under both Windows and Unix (Linux). The basic idea is that the game (client) is run on Windows and the server on Unix. But because of our platform-independent code, we do not have to worry about the network code if we want to change platforms.

We are going to create our network library using only UDP protocol, as it is more suitable for computer games than TCP. You could use both, but to make things simpler, we only concentrate on UDP. You may wonder if this is what other games on the market do. The answer is yes; all the games you play on the Internet use UDP protocol. Well, most of them anyway.

The name for the library we are creating here is `dreamSock`. The name comes from a game development group to which the authors of this book belonged. The group is no longer active, but `dreamSock` lives on!

Why Create a Network Library of Our Own?

One big question arises when you start to think about creating your own network library: Why? Why would anyone want to make one when there are perfectly good network libraries out there already? For example, why not use Microsoft's `DirectPlay`?

`DirectPlay` is an excellent library, and because it is designed solely for gaming, it is an excellent choice for a game programmer. But it has one major drawback: You do not have Unix connectivity with it. Now you may think that no one really needs that. Well, it is really up to you if you want Unix connectivity, but because Unix is an absolutely stable system, it is the best choice for the server platform. There are some examples of bad experiences stemming from not using any Unix operating system as the server platform in recent gaming history, so why repeat those mistakes?

If you create your own network library using sockets, you can have full Unix connectivity if you want. Plus, you will have a library that fits your needs exactly. And you can always improve upon it.

Planning the Structure

Before doing anything else we should take a moment to think about the structure of the library. We need to know all the facts about the library before we start to code it. If we started coding the library without any planning beforehand, we would most likely need to modify the structure many times before it met our requirements.

So let's look at some of the facts. Fact number one is that the library must be platform independent. This leads to the conclusion that it is best to use platform-independent functions as much as we can, and also divide the code into separate files to store platform-dependent code. We can easily add support for more platforms this way. We need one main source file that wraps up all the code and selects the correct functions to be run.

Fact number two is that in a network connection, we have a server, the clients, and the connection between them (or in the case of UDP, only the packets we send between them). So it is a good idea to make

separate C++ classes — a class for the server, a class for the client, and a class for the messages between them.

We also want an easy-to-use interface for the library so the C++ classes internally take care of everything for the end user. But it would be nice to have access to the socket functions too, so we will make global functions for them first. Because our game needs to have only Windows and Unix support, we create three source files and their header files: `dreamSock.cpp`, `dreamSock.h`, `dreamSockGlobal.cpp`, `dreamWinSock.cpp`, and `dreamLinuxSock.cpp`. We also create `dreamSockLog.cpp` and `dreamSockLog.h` for our logging system. Some people want to put each class into a separate file, and that is fine, but we will put all the classes into `dreamSock.cpp` and `dreamSock.h`.

A network library may vary in size, depending on the functionality we want, so it is a good idea to make the code as modular as possible. Therefore, we use the C++ class as the base of our library, with each object being one connection on a client and one service on a server (with multiple connections). Other factors do not affect the structure of the library much.

Planning the Functionality

Knowing the basic functionality before starting to code is also a must. Of course, we could just write functions for sending and receiving data without anything extra, but what would that accomplish? Nothing. But we are going to do that anyway. Yes, it sounds stupid, but read on. We write some global functions before we write the actual library functionality, and then we use these global functions in the C++ class we are going to write.

We want to create the library to make it as easy to use as possible. We do this by moving all the complex network technology from the end programmer to the background, so that the end programmer can merely point and tell the library what to do. The end programmer should not worry about binding address information to a socket; that is a job for the network library. The end programmer should not have to understand the complex technical functionality of the network code. And most of all, the network library must take care of sending the data to the correct location.

Identifying Hosts

Being able to tell one host from another is not really required when sending because whenever we send data, we send it to all the clients at once (or only to the server). We do send unique data to a host when it

connects to the server, but we only reply to the address from which the connection request came. A client, on the other hand, can only send to a server.

The clients that join a server are stored in a client list. This is a linked list, so when a client disconnects, the original order of the list is lost. Clients are stored in the order in which they join the server. It is possible to send to a certain client by browsing that list and comparing the addresses to see which one is the one we want, but normally there is no need for that.

But we do need to identify a host when we receive data from it. How else would we know what client to process? When we receive data, we also receive the address it came from, and we simply compare this address to the ones we have stored.

Sending Data to Hosts

Sending data to hosts is not as straightforward as it may seem at first. This is because we need to decide if we want to let the clients send data between each other directly without having the server as a router of any kind. To be able to send data directly between clients, all the clients need to know the address information of all the other clients. If there are a lot of clients in the network, it would consume vast amounts of memory and time (as the address information must be sent via the network). Also, the server cannot process data that is sent between two clients. This is why we decided not to allow clients to send data between each other directly.

In fact, we will not provide functionality to send data between clients indirectly either. That is the job for the end user if it is required. Normally, a client sends its information to the server, and the server then sends that information to every client (even to the one that originally sent it to the server).

We will create a message sequence system that allows us to see if a packet is dropped. Each packet is given a sequence number, and the remote host knows what packet number to expect next. If those numbers do not match, one more packet was lost somewhere. We will continue processing the packets normally, but the number of dropped packets is noted. It is up to the network application whether it cares about dropped packets.

What about multithreading, then? Do we need to open a socket for each client on the server? With UDP, the answer is no. All we need is one socket on the server side that takes care of sending and receiving data for every client. That is possible because we store the clients' addresses in the client list, and we can then pick the address from there when we want to send data or when we need to identify a host. Do not

worry about the server missing some packets even though it uses only one socket, as the operating system buffers all the incoming messages for us.

So to summarize: A client sends data only to the server, and the server sends data to all clients at once. Each host opens only one socket.

Pinging — Calculating Network Latency

Pinging is a useful way to see how fast the network connection is between two hosts. Pinging is very simple: Have you ever seen a submarine movie where a sub pings another sub to see if it is there (or how far away it is)? The sound is deflected from the other sub's hull, and those on the first sub will hear the ping sound come back. Pinging in network programming is exactly the same. We send a ping message to the host we want, and then we listen for the ping to come back. Then we calculate the time difference between sending and receiving the ping, and there it is, network latency between those two hosts. Remember that this value is the time it takes for a client to send a message and receive a response, so it takes roughly half of that time for one message to reach its destination.

We create pinging directly in our network library. Only the server calculates the values because we want to be able to tell all the clients' pings to everyone.

Timing Out

Some clients may crash without being able to disconnect properly from the server, so we need a way to drop those clients from the server. One way to do that is to keep track of when the client sent us something. If the client did not send us anything in a certain amount of time, we assume it has crashed and should be dropped from the server. We implement this in our network library with the timeout value set to 30 seconds.

Building the Library

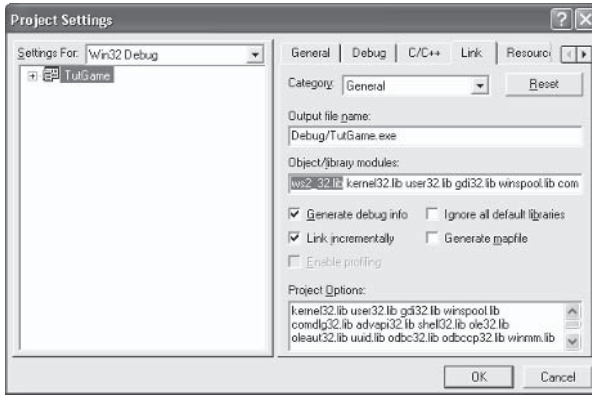
Now we will learn how to build the library file. We will actually learn to set up the compiler so that it builds a library file for us, which can then be linked to other applications.

Windows

Building and compiling the library on Windows is very easy with Microsoft Visual C++ 6.0. Because we are building a library, there is no need (and no way) to link any extra libraries with it. All the linking of other libraries is done when we build the application that uses our library. But there is a way to make our library link other libraries automatically.

Because this network library requires the WinSock 2 library on Windows, we need to link it with the application. We could link it normally among the other libraries in Microsoft Visual C++ 6.0 settings, as shown in the following figure.

Figure 1



But we can also add the following line in our network library's header file to make the WinSock 2 library be linked automatically whenever our network library is linked.

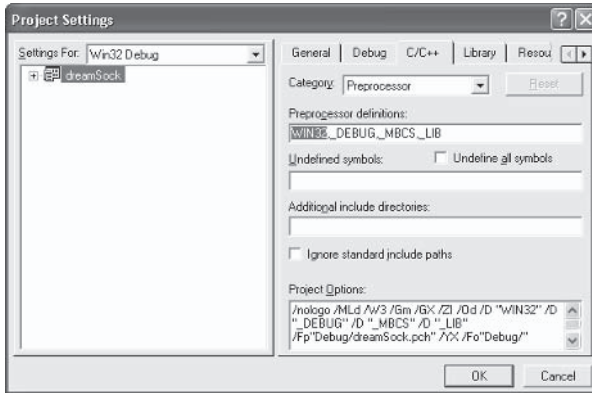
```
#pragma comment (lib, "ws2_32.lib")
```

We must also define the constant `WIN32`, as shown in the following piece of code, to make the compiler compile the correct version of the source code.

```
#define Win32
```

Or we can add it to the preprocessor list as shown in the following figure.

Figure 2



This is our own constant, so we must also know where and how to use it. It is used to define parts of code that belong to Windows only or Unix only. It is used as follows:

```
#ifdef Win32
// place Windows code here
#else
// place other platform code here (in our case: Unix/Linux code)
#endif
```

Unix/Linux

Building the library on Unix is a little bit more complicated, as is pretty much everything on Unix systems. We need to use the following makefile and a command-line compiler. This should work on most Unix systems, but depending on your setup, you may have to change some things. Note also that `C_ARGS` is commented out, but you can uncomment it to see more warnings during compilation.

```
#C_ARGS = -Wall
CC = gcc

all: dreamSock.o dreamSockLog.o dreamLinuxSock.o dreamSockGlobal.o
ar q libdreamSock.a dreamSock.o dreamSockLog.o dreamLinuxSock.o
dreamSockGlobal.o
cp libdreamSock.a ../server/
cp dreamSock.h ../server/
cp dreamSockLog.h ../server/

clean:
rm -f *.o
rm libdreamSock.a
```

```
dreamSock.o:      dreamSock.cpp
                 $(CC) $(C_ARGS) -c dreamSock.cpp

dreamSockLog.o:  dreamSockLog.cpp
                 $(CC) $(C_ARGS) -c dreamSockLog.cpp

dreamLinuxSock.o: dreamLinuxSock.cpp
                 $(CC) $(C_ARGS) -c dreamLinuxSock.cpp

dreamSockGlobal.o: dreamSockGlobal.cpp
                  $(CC) $(C_ARGS) -c dreamSockGlobal.cpp
```

Now all we need to do is run `make` on the command line and the library is built. The above makefile will copy the required header files and library file to a directory called `server`. That is where you should put your server code.

NOTE Make sure you have enough rights to make the library on the Unix machine you use.

Creating Independent Code

In this section we discuss how we can make our code platform independent. We use the same function calls, but the insides differ. To the end programmer who will be using the library, it looks as if the code is exactly the same, regardless of the platform we are using.

Creating Definitions for Data Types

To create the new definitions we must know which platform is used for the build. This is very easy: You can either add a definition for the current platform at the beginning of your code or use the preprocessor of the compiler. Both do the same thing really. If you want to use `#define`, just add this line when you are compiling under Windows:

```
#define Win32
```

You do not have to add a `define` for Unix builds because you can simply comment out the definition of `WIN32`. This kind of system assumes that we are going to use the code only on Windows and some other (unspecified) operating system. In our case, the other operating system is Unix (or Linux). We could provide definitions for all the operating systems we are going to support, but it is easier this way when we use only two known systems.

Using the preprocessor is even easier. Just add the string “Win32” in the Windows version makefile (or workspace settings) preprocessor’s list of definitions. Now you do not have to make any modifications to your code when you move from one platform to another. The makefile you use defines the platform for the code.

But how do we check if the definition of WIN32 is there? Again, this is very simple.

```
#ifndef Win32
    typedef int socklen_t; // will be run if we are on Win32
#else
    typedef int SOCKET; // will be run if we are not on Win32
#endif
```

This small piece of code checks the platform used during the build. The compiler ignores the non-true statement, and it will not be added to the executable. Hence, it does not slow down the run-time process.

Now we will create definitions for some data types, so we can use one data type identifier on all platforms. These are the types to define:

- `socklen_t = int`
- `SOCKET = int`
- `TRUE = 1`
- `FALSE = 0`

Here is how we do that in our code:

```
// Define SOCKET data type for Unix (defined in WinSock for Win32)
// And socklen_t for Win32
#ifndef Win32
    typedef int socklen_t;
#else
    typedef int SOCKET;

    #ifndef TRUE
    #define TRUE 1
    #endif
    #ifndef FALSE
    #define FALSE 0
    #endif
#endif
```

Log System

It is a good idea to keep a log of what is happening so we can see exactly what happened when. For example, if something does go wrong we do not have to shut down the server and start investigating. We can first check the log file to see if that helps.

The log system goes into the `dreamSockLog.cpp` and `dreamSockLog.h` files. Let's first take a look at the header file:

```
#ifndef __DREAMSOCKLOG_H__
#define __DREAMSOCKLOG_H__

#ifdef Win32

class dreamConsole
{
public:
    dreamConsole(char *title);
    ~dreamConsole();
    void println(char *string, int type, ...);
};

#define CONSOLE_NOTIFY 0
#define CONSOLE_ERROR 1
#define CONSOLE_FATAL 2

// Function prototypes
void StartLogConsole(void);
#endif

// Function prototypes
void StartLog(void);
void StopLog(void);
void LogString(char *string, ...);

#endif
```

As you see, the log system's header file is rather simple. You should notice that the `dreamConsole` class is defined only for Windows. This console class is used to display run-time log information. Unix systems do not need that because they are nothing but big consoles themselves.

StartLogConsole Function

The `StartLogConsole` function simply creates a `dreamConsole` object.

```
#ifdef Win32
void StartLogConsole(void)
{
    console = new dreamConsole("dreamSock application");
}
#endif
```

dreamConsole Constructor

The constructor function for `dreamConsole` is listed here:

```
dreamConsole::dreamConsole(char *title)
{
    AllocConsole();
    SetConsoleTitle(title);
}
```

dreamConsole Destructor

The destructor function for `dreamConsole` is as follows:

```
dreamConsole::~~dreamConsole()
{
    FreeConsole();
}
```

The constructor and destructor functions allocate and deallocate the Win32 console system.

println Function

The `println` function is used to print a line on the console screen.

```
void dreamConsole::println(char *string, int type, ...)
{
    char buf[1024];
    char buf2[1024];
    va_list ap;
    va_start(ap, string);
    vsprintf(buf, string, ap);
    va_end(ap);

    sprintf(buf2, "-> %s\n", buf);

    HANDLE console = GetStdHandle(STD_OUTPUT_HANDLE);

    switch(type)
    {
    case 0:
        SetConsoleTextAttribute(console, FOREGROUND_GREEN |
                                   FOREGROUND_INTENSITY);
        break;

    case 1:
        SetConsoleTextAttribute(console, FOREGROUND_RED |
                                   FOREGROUND_INTENSITY);
        break;
    }
}
```



```

    WriteConsole(console, buf2, strlen(buf2), NULL, NULL);
}

```

Here we first get a handle to a standard console output and then set the attributes for the text color and background. Finally the text is written on the console screen.

```

HANDLE console = GetStdHandle(STD_OUTPUT_HANDLE);

switch(type)
{
case 0:
    SetConsoleTextAttribute(console, FOREGROUND_GREEN |
                                FOREGROUND_INTENSITY);
    break;

case 1:
    SetConsoleTextAttribute(console, FOREGROUND_RED | FOREGROUND_INTENSITY);
    break;
}

WriteConsole(console, buf2, strlen(buf2), NULL, NULL);

```

StartLog Function

StartLog is used to start the log system.

```

void StartLog(void)
{
    time_t current = time(NULL);

    if((LogFile = fopen("dreamSock.log", "w")) != NULL)
    {
        fprintf(LogFile, "Log file started %s", ctime(&current));

        fclose(LogFile);
    }

    if((LogFile = fopen("dreamSock.log", "a")) != NULL)
    {
    }
}

```

The following piece of code is used to retrieve the current time. This information is written every time something is logged so we know when it happened.

```
time_t current = time(NULL);
```

We open the file for writing and write the current time at the beginning of the file. Then we close the log file and open it for adding so we can start adding text to it. We leave the file handle open.

```
if((LogFile = fopen("dreamSock.log", "w")) != NULL)
{
    fprintf(LogFile, "Log file started %s", ctime(&current));
    fclose(LogFile);
}

if((LogFile = fopen("dreamSock.log", "a")) != NULL)
{
}
```

LogString Function

LogString is used to add a string to the log file.

```
void LogString(char *string, ...)
{
    char buf[1024];
    va_list ap;
    va_start(ap, string);
    vsprintf(buf, string, ap);
    va_end(ap);

    // Get current time and date
    time_t current = time(NULL);

    char timedata[64];
    sprintf(timedata, ctime(&current));

    // Remove linefeed from time/date string
    int i = 0;

    while(timedata[i] != '\n')
    {
        i++;
    }

    timedata[i] = '\0';

    // Output log string
#ifdef Win32
    fprintf(LogFile, "%s: %s\n", timedata, buf);
#else
    if(console)
        console->println(buf, 0);
// Linux outputs to screen and to the open file when running as daemon
printf("%s: %s\n", timedata, buf);
#endif
}
```

First we will format the text so that we can add any variable values to it. The final string will be stored in string `buf`.

```
char buf[1024];
va_list ap;
va_start(ap, string);
vsprintf(buf, string, ap);
va_end(ap);
```

Again we retrieve the current time to add a time stamp with each log string. We remove the almost annoying linefeed character from the end of the `timedate` string so we can write stuff on the same line as the time stamp.

```
time_t current = time(NULL);

char timedate[64];
sprintf(timedate, ctime(&current));

// Remove linefeed from time/date string
int i = 0;

while(timedate[i] != '\n')
{
    i++;
}

timedate[i] = '\0';
```

We write the time stamp and the string and follow it by a linefeed, so the user does not have to add the linefeed to the string every time.

```
// Output log string
#ifdef Win32
fprintf(LogFile, "%s: %s\n", timedate, buf);

if(console)
    console->println(buf, 0);
#else
// Linux outputs to screen and to the open file when running as daemon
printf("%s: %s\n", timedate, buf);
#endif
```

StopLog Function

After we are done logging, we need to stop the log system. That is done with the following `StopLog` function:

```
void StopLog(void)
{
    fclose(LogFile);

#ifdef Win32
    if(console != NULL)
```

```
        delete console;
    #endif
}
```

The log file handle is closed. On Windows systems, the console object is removed if it exists.

Getting Started

Let's get started. It is a good idea to start writing the code so that we can test it as soon as possible. It is very frustrating to write tons of code and not know if it works at all. It is also much easier to debug the code if we notice when a bug appears (of course, we are aiming not to meet any critters). So we will start from scratch and move on to the final library step by step.

Setting Up Source and Header Files

Before we actually start coding the library, we need to set up some things in the source files and header files. In the main source file, `dreamSock.cpp`, we need to have the following build-time definition check in the Windows version to make sure that WinSock 2 API is used instead of an older version:

```
#ifndef _WINSOCKAPI_
#define _WINSOCKAPI_
#endif
```

If we do not define `_WINSOCKAPI_` before including `windows.h`, an old version of the WinSock header file will be included and the build will fail. So the preceding piece of code is required for all WinSock 2 applications that include the `windows.h` header.

The beginning of `dreamSock.cpp` looks like this:

```
#ifdef Win32
// Windows-specific headers
    #ifndef _WINSOCKAPI_
    #define _WINSOCKAPI_
    #endif
    #include <windows.h>
    #include <winsock2.h>
#else
// Unix-specific headers
    #include <memory.h>
    #include <errno.h>
    #include <sys/ioctl.h>
    #include <sys/socket.h>
    #include <sys/time.h>
    #include <netinet/in.h>
```

```

#include <arpa/inet.h>
#endif

// Common headers
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <ctype.h>
#include <time.h>
#include "dreamSock.h"
#include "dreamSockLog.h"

```

As we can see in the preceding code, we have separate header files for Windows and Unix, along with some common header files that both operating systems require. As we learned in the section titled “Creating Independent Code,” we use build-time definition checks to see what piece of code is to be linked and what is not, depending on the platform we are on.

dreamSock.h File

```

#ifndef __DREAMSOCK_H
#define __DREAMSOCK_H

#include "dreamSockLog.h"

#ifdef Win32
#pragma comment (lib,"ws2_32.lib")
#pragma message ("Auto linking WinSock2 library")

#include <winsock2.h>
#else
#include <string.h>
#include <netinet/in.h>
#endif

#include <stdio.h>
#include <stddef.h>

// Define SOCKET data type for Unix (defined in WinSock for Win32)
// And socklen_t for Win32
#ifdef Win32
typedef int socklen_t;
#else
typedef int SOCKET;

#ifndef TRUE
#define TRUE 1
#endif
#ifndef FALSE
#define FALSE 0
#endif

```

```
#endif

// Host types
#define DREAMSERVER          -1
#define DREAMCLIENT        0

// Connection protocols
#define DREAMSOCK_TCP       0
#define DREAMSOCK_UDP       1

// Connection states
#define DREAMSOCK_CONNECTING 0
#define DREAMSOCK_CONNECTED  1
#define DREAMSOCK_DISCONNECTING 2
#define DREAMSOCK_DISCONNECTED 4

// Error codes
#define DREAMSOCK_SERVER_ERROR 1
#define DREAMSOCK_CLIENT_ERROR 2

#ifdef Win32
    #define DREAMSOCK_INVALID_SOCKET INVALID_SOCKET
#else
    #define DREAMSOCK_INVALID_SOCKET -1
#endif

// System messages
// Note (for all messages – system and user):
// positive = sequenced message
// negative = unsequenced message
#define DREAMSOCK_MES_CONNECT -101
#define DREAMSOCK_MES_DISCONNECT -102
#define DREAMSOCK_MES_ADDCLIENT -103
#define DREAMSOCK_MES_REMOVECLIENT -104
#define DREAMSOCK_MES_PING -105

// Introduce classes
class dreamMessage;
class dreamClient;
class dreamServer;
class dreamSock;

class dreamMessage
{
private:
    bool        overFlow;
    int         maxSize;
    int         size;
    int         readCount;

    char        *GetNewPoint(int length);

public:
    void        Init(char *d, int length);
```

```

void          Clear(void);
void          Write(void *d, int length);
void          AddSequences(dreamClient *client);

void          WriteByte(char c);
void          WriteShort(short c);
void          WriteLong(long c);
void          WriteFloat(float c);
void          WriteString(char *s);
void          BeginReading(void);
void          BeginReading(int s);
char          *Read(int s);
char          ReadByte(void);
short         ReadShort(void);
long          ReadLong(void);
float         ReadFloat(void);
char          *ReadString(void);

bool          GetOverFlow(void)          {return overFlow;}
int           GetSize(void)              {return size;}
void          SetSize(int s)              {size = s;}

char          *data;
char          outgoingData[1400];
};

class dreamClient
{
private:
    void          DumpBuffer(void);
    void          ParsePacket(dreamMessage *mes);

    int           connectionState;        // Connecting, connected,
                                           // disconnecting, disconnected

    unsigned short outgoingSequence;      // Outgoing packet sequence
    unsigned short incomingSequence;      // Incoming packet sequence
    unsigned short incomingAcknowledged;   // Last packet acknowledged
                                           // by other end
    unsigned short droppedPackets;        // Dropped packets

    int           serverPort;              // Port
    char          serverIP[32];            // IP address
    int           index;                   // Client index (starts
                                           // from 1, running number)
    char          name[32];                 // Client name

    SOCKET        socket;                  // Socket
    struct sockaddr myaddress;              // Socket address

    int           pingSent;                 // When did we send ping?
    int           ping;                     // Network latency

```

```

int          lastMessageTime;

bool         init;

public:

              dreamClient();
              ~dreamClient();

int          Initialize(char *localIP, char *remoteIP, int port);
void         Uninitialize(void);
void         Reset(void);
void         SendConnect(char *name);
void         SendDisconnect(void);
void         SendPing(void);

void         SetConnectionState(int con) {connectionState = con;}
int          GetConnectionState(void) {return connectionState;}

int          GetPacket(char *data, struct sockaddr *from);
void         SendPacket(void);
void         SendPacket(dreamMessage *message);

unsigned short GetOutgoingSequence(void) {return outgoingSequence;}
void         SetOutgoingSequence(unsigned short seq)
              {outgoingSequence = seq;}
void         IncreaseOutgoingSequence(void) {outgoingSequence++;}
unsigned short GetIncomingSequence(void) {return incomingSequence;}
void         SetIncomingSequence(unsigned short seq)
              {incomingSequence = seq;}
unsigned short GetIncomingAcknowledged(void) {return
              incomingAcknowledged;}
void         SetIncomingAcknowledged(unsigned short seq)
              {incomingAcknowledged = seq;}
unsigned short GetDroppedPackets(void) {return droppedPackets;}
void         SetDroppedPackets(unsigned short drop)
              {droppedPackets = drop;}

bool         GetInit(void) {return init;}

int          GetIndex(void) {return index;}
void         SetIndex(int ind) {index = ind;}

char         *GetName(void) {return name;}
void         SetName(char *n) {strcpy(name, n);}

SOCKET      GetSocket(void) {return socket;}
void         SetSocket(SOCKET sock) {socket = sock;}

struct sockaddr *GetSocketAddress(void) {return &myaddress;}
void         SetSocketAddress(struct sockaddr *address)
              {memcpy(&myaddress, address, sizeof(struct
              sockaddr));}

int          GetPingSent(void)          {return pingSent;}

```



```

        void          SetPing(int p)          {ping = p;}

        int          GetLastMessageTime(void) {return lastMessageTime;}
        void          SetLastMessageTime(int t) {lastMessageTime = t;}

        dreamMessage message;
        dreamClient  *next;
};

class dreamServer
{
private:
    void          SendAddClient(dreamClient *newClient);
    void          SendRemoveClient(dreamClient *client);
    void          AddClient(struct sockaddr *address, char *name);
    void          RemoveClient(dreamClient *client);
    void          ParsePacket(dreamMessage *mes, struct sockaddr
                               *address);
    int           CheckForTimeout(char *data, struct sockaddr *from);

    dreamClient  *clientList;

    int          port;           // Port
    SOCKET       socket;        // Socket
    int          runningIndex;   // Running index numbers for new
                               // clients

    bool         init;

public:
                                dreamServer();
                                ~dreamServer();

    int          Initialize(char *localIP, int serverPort);
    void         Uninitialize(void);
    void         SendPing(void);
    int          GetPacket(char *data, struct sockaddr *from);
    void         SendPackets(void);

    bool         GetInit(void) {return init;}
    dreamClient *GetClientList(void) {return clientList;}

    int          GetPort(void) {return port;}
};

/*****

dreamSock global functions

*****/

// Function prototypes
int dreamSock_Initialize(void);
int dreamSock_InitializeWinSock(void);

```

```

void dreamSock_Shutdown(void);

SOCKET dreamSock_Socket(int protocol);
int dreamSock_SetNonBlocking(SOCKET sock, u_long setMode);
int dreamSock_SetBroadcasting(SOCKET sock, int mode);
int dreamSock_StringToSockaddr(char *addressString, struct sockaddr *sadr);
SOCKET dreamSock_OpenUDPSocket(char netInterface[32], int port);
void dreamSock_CloseSocket(SOCKET sock);

int dreamSock_GetPacket(SOCKET sock, char *data, struct sockaddr *from);
void dreamSock_SendPacket(SOCKET sock, int length, char *data, struct
                          sockaddr addr);
void dreamSock_Broadcast(SOCKET sock, int length, char *data, int port);

#ifdef Win32
int dreamSock_Linux_GetCurrentSystemTime(void);
#else
int dreamSock_Win_GetCurrentSystemTime(void);
#endif

int dreamSock_GetCurrentSystemTime(void);

#endif

```

dreamMessage Class

Now let's start creating our library by first creating the header files. Here we list the dreamMessage class, which is typically used to build outgoing messages and parse incoming messages.

```

class dreamMessage
{
private:
    bool        overFlow;
    int         maxSize;
    int         size;
    int         readCount;

    char        *GetNewPoint(int length);

public:
    void        Init(char *d, int length);
    void        Clear(void);
    void        Write(void *d, int length);
    void        AddSequences(dreamClient *client);

    void        WriteByte(char c);
    void        WriteShort(short c);
    void        WriteLong(long c);
    void        WriteFloat(float c);
    void        WriteString(char *s);
    void        BeginReading(void);
    void        BeginReading(int s);
    char        *Read(int s);

```

```

char          ReadByte(void);
short         ReadShort(void);
long          ReadLong(void);
float         ReadFloat(void);
char          *ReadString(void);

bool          GetOverFlow(void) {return overFlow;}
int           GetSize(void) {return size;}
void          SetSize(int s) {size = s;}

char          *data;
char          outgoingData[1400];
};

```

We take a closer look at the methods and variables later in this tutorial, but as you can probably see, this class is rather simple. It contains many write and read methods, as well as size control methods.

dreamClient Class

Next comes the dreamClient class, which is a bit more complicated than dreamMessage. Do not worry too much about that now, everything will be explained in detail later.

```

class dreamClient
{
private:
    void          DumpBuffer(void);
    void          ParsePacket(dreamMessage *mes);

    int           connectionState;          // Connecting, connected,
                                           // disconnecting, disconnected

    unsigned short outgoingSequence;       // Outgoing packet sequence
    unsigned short incomingSequence;       // Incoming packet sequence
    unsigned short incomingAcknowledged;    // Last packet acknowledged
                                           // by other end
    unsigned short droppedPackets;         // Dropped packets

    int           serverPort;               // Port
    char          serverIP[32];             // IP address
    int           index;                   // Client index (starts from
                                           // 1, running number)
    char          name[32];                 // Client name

    SOCKET        socket;                   // Socket
    struct sockaddr myaddress;              // Socket address

    int           pingSent;                 // When did we send ping?
    int           ping;                     // Network latency

    int           lastMessageTime;

```

```

bool          init;

public:

                                dreamClient();
                                ~dreamClient();

int           Initialize(char *localIP, char *remoteIP, int port);
void          Uninitialize(void);
void          Reset(void);
void          SendConnect(char *name);
void          SendDisconnect(void);
void          SendPing(void);

void          SetConnectionState(int con) {connectionState = con;}
int           GetConnectionState(void) {return connectionState;}

int           GetPacket(char *data, struct sockaddr *from);
void          SendPacket(void);
void          SendPacket(dreamMessage *message);

unsigned short GetOutgoingSequence(void) {return outgoingSequence;}
void          SetOutgoingSequence(unsigned short seq)
              {outgoingSequence = seq;}
void          IncreaseOutgoingSequence(void) {outgoingSequence++;}
unsigned short GetIncomingSequence(void) {return incomingSequence;}
void          SetIncomingSequence(unsigned short seq)
              {incomingSequence = seq;}
unsigned short GetIncomingAcknowledged(void) {return
              incomingAcknowledged;}
void          SetIncomingAcknowledged(unsigned short seq)
              {incomingAcknowledged = seq;}
unsigned short GetDroppedPackets(void) {return droppedPackets;}
void          SetDroppedPackets(unsigned short drop)
              {droppedPackets = drop;}

bool          GetInit(void) {return init;}

int           GetIndex(void) {return index;}
void          SetIndex(int ind) {index = ind;}

char          *GetName(void) {return name;}
void          SetName(char *n) {strcpy(name, n);}

SOCKET        GetSocket(void) {return socket;}
void          SetSocket(SOCKET sock) {socket = sock;}

struct sockaddr *GetSocketAddress(void) {return &myaddress;}
void          SetSocketAddress(struct sockaddr *address)
              {memcpy(&myaddress, address, sizeof(struct
              sockaddr));}

int           GetPingSent(void) {return pingSent;}
void          SetPing(int p) {ping = p;}

```

```

        int             GetLastMessageTime(void) {return lastMessageTime;}
        void            SetLastMessageTime(int t) {lastMessageTime = t;}

        dreamMessage    message;
        dreamClient     *next;
};

```

This class is used on both the server and client applications. On the server side, this class works as a linked list to store all the clients that have connected to that server. When the server wants to send data to a client, it uses an object in this list to do so. On the client side, this class is used to achieve and maintain the connection with the server.

dreamServer Class

Last but not least is the server class — `dreamServer`. This is actually simpler than you might think, because all the magic happens in the `dreamClient` class.

```

class dreamServer
{
private:
    void            SendAddClient(dreamClient *newClient);
    void            SendRemoveClient(dreamClient *client);
    void            AddClient(struct sockaddr *address, char *name);
    void            RemoveClient(dreamClient *client);
    void            ParsePacket(dreamMessage *mes, struct sockaddr
                               *address);
    int             CheckForTimeout(char *data, struct sockaddr *from);

    dreamClient     *clientList;

    int             port;           // Port
    SOCKET          socket;        // Socket
    int             runningIndex;  // Running index numbers for new
                               // clients

    bool            init;

public:
                                dreamServer();
                                ~dreamServer();

    int             Initialize(char *localIP, int serverPort);
    void            Uninitialize(void);
    void            SendPing(void);
    int             GetPacket(char *data, struct sockaddr *from);
    void            SendPackets(void);

    bool            GetInit(void)      {return init;}
    dreamClient     *GetClientList(void) {return clientList;}
};

```

```
int GetPort(void) {return port;}  
};
```

This class is used only on the server side. Its function is to listen for incoming clients and communicate with them as long as they are connected. This class contains a linked list of all connected clients, and that is where this class handles network communication.

Global Setup Functions

There are some global setup functions that are not in any class but are very important for the library. These functions are:

- `int dreamSock_Initialize(void);`
- `int dreamSock_InitializeWinSock(void);`
- `void dreamSock_Shutdown(void);`

We have one global variable in our library that is not in any class. This is a boolean variable (`bool dreamSock_init`) to store information about `dreamSock`'s current state — whether it has been initialized or not. This is important to know because many functions may call `dreamSock_Initialize` or `dreamSock_Shutdown` in our application. With the `dreamSock_init` variable, we can avoid running initialization multiple times.

`dreamSock_Initialize`

This function is used to initialize the library, and it is a platform-independent function. This means that whatever the platform is, this function must be run before any other `dreamSock` function can be used. As we can see in the following code listing, this function currently has no use on Unix (other than starting the log system). But this function must exist to keep the source code platform independent. Also, we can add Unix code later if we need to. On Windows, this runs a specific function to initialize the WinSock API. The following code listing is the first version of the `dreamSock_Initialize` function. We will update this function later on.

```
int dreamSock_Initialize(void)  
{  
    if(dreamSock_init == true)  
        return 0;  
  
    dreamSock_init = true;  
  
    StartLog();  
  
#ifdef Win32  
    return dreamSock_InitializeWinSock();  
#endif
```

```

#else
    return 0;
#endif
}

```

dreamSock_InitializeWinSock

This function, in contrast to the previous function, is used to initialize only WinSock API. The programmer using our library does not have to worry about this function, as `dreamSock_Initialize` will automatically run this if we are on Windows. More accurately speaking, the compiler will compile the source code so that this function will be run. We introduced and examined this function in detail in Chapter 5.

```

int dreamSock_InitializeWinSock(void)
{
    WORD versionRequested;
    WSADATA wsaData;
    DWORD bufferSize = 0;

    LPWSAPROTOCOL_INFO SelectedProtocol;
    int NumProtocols;

    // Start WinSock2. If it fails, we do not need to call WSACleanup()
    versionRequested = MAKEWORD(2, 0);
    int error = WSASStartup(versionRequested, &wsaData);

    if(error)
    {
        LogString("FATAL ERROR: WSASStartup failed (error = %d)", error);
        return 1;
    }
    else
    {
        LogString("WSASStartup OK");

        // Confirm that the WinSock2 DLL supports the exact version
        // we want. If not, call WSACleanup().
        if(LOBYTE(wsaData.wVersion) != 2 || HIBYTE(wsaData.wVersion)
            != 0)
        {
            LogString("FATAL ERROR: WinSock2 DLL does not support the
                correct version (%d.%d)",
                LOBYTE(wsaData.wVersion), HIBYTE(wsaData.wVersion));

            WSACleanup();
            return 1;
        }
    }

    // Call WSAEnumProtocols to figure out how big of a buffer we need
    NumProtocols = WSAEnumProtocols(NULL, NULL, &bufferSize);
}

```

```
if((NumProtocols != SOCKET_ERROR) && (WSAGetLastError() != WSAENOBUFS))
{
    WSACleanup();
    return 1;
}

// Allocate a buffer and call WSAEnumProtocols to get an array of
// WSAPROTOCOL_INFO structs
SelectedProtocol = (LPWSAPROTOCOL_INFO) malloc(bufferSize);

if(SelectedProtocol == NULL)
{
    WSACleanup();
    return 1;
}

// Allocate memory for protocol list and define the protocols to
// look for
int *protos = (int *) calloc(2, sizeof(int));

protos[0] = IPPROTO_TCP;
protos[1] = IPPROTO_UDP;

NumProtocols = WSAEnumProtocols(protos, SelectedProtocol, &bufferSize);

free(protos);
protos = NULL;

free(SelectedProtocol);
SelectedProtocol = NULL;

if(NumProtocols == SOCKET_ERROR)
{
    LogString("FATAL ERROR: Didn't find any required protocols");
    WSACleanup();
    return 1;
}

return 0;
}
```

dreamSock_Shutdown

This function, naturally, is used to shut down our network library. Like `dreamSock_Initialize`, this function also currently has no real use on Unix. But again, it must exist and must be run on both operating systems' code to achieve full platform independency. On Windows this function will shut down the WinSock API. It must be run to shut down the API; otherwise, we may be interfering the WinSock DLL registration process. The following code listing shows the first version of the `dreamSock_Shutdown` function (we will update this function later).


```

void dreamSock_Shutdown(void)
{
    if(dreamSock_init == false)
        return;

    LogString("Shutting down dreamSock");

    dreamSock_init = false;

    StopLog();

#ifdef Win32
    WSACleanup();
#endif
}

```

Global Socket Functions

Before we go into detail about all the methods and variables in the classes, we should take a look at the global socket functions that are used throughout the classes. The purpose of these functions is to wrap some common socket functions into more compact ones, so it is easier to start programming the actual network application.

Here are the functions:

```

SOCKET dreamSock_Socket(int protocol)
{
    int type;
    int proto;
    SOCKET sock;

    // Check which protocol to use
    if(protocol == DREAMSOCK_TCP)
    {
        type = SOCK_STREAM;
        proto = IPPROTO_TCP;
    }
    else
    {
        type = SOCK_DGRAM;
        proto = IPPROTO_UDP;
    }

    // Create the socket
    if((sock = socket(AF_INET, type, proto)) == -1)
    {
        LogString("dreamSock_Socket - socket() failed");
    }

#ifdef Win32
    errno = WSAGetLastError();
    LogString("Error: socket() code %d : %s", errno,
        strerror(errno));
#endif
}

```

```
#else
    LogString("Error: socket() : %s", strerror(errno));
#endif

    return DREAMSOCK_INVALID_SOCKET;
}

return sock;
}

int dreamSock_SetNonBlocking(SOCKET sock, u_long setMode)
{
    u_long set = setMode;

    // Set the socket option
#ifdef Win32
    return ioctlsocket(sock, FIONBIO, &set);
#else
    return ioctl(sock, FIONBIO, &set);
#endif
}

int dreamSock_SetBroadcasting(SOCKET sock, int mode)
{
    // make it broadcast capable
    if(setsockopt(sock, SOL_SOCKET, SO_BROADCAST, (char *) &mode,
        sizeof(int)) == -1)
    {
        LogString("DreamSock_SetBroadcasting failed");
    }

#ifdef Win32
    int err = WSAGetLastError();
    LogString("Error code %d: setsockopt() : %s", err,
        strerror(err));
#else
    LogString("Error code %d: setsockopt() : %s", errno,
        strerror(errno));
#endif

    return DREAMSOCK_INVALID_SOCKET;
}

return 0;
}

int dreamSock_StringToSockaddr(char *addressString, struct sockaddr *sadr)
{
    char copy[128];

    memset(sadr, 0, sizeof(struct sockaddr));
```

```

struct sockaddr_in *addressPtr = (struct sockaddr_in *) saddr;

addressPtr->sin_family = AF_INET;
addressPtr->sin_port = htons(0);

strcpy(copy, addressString);

// If the address string begins with a number, assume an IP address
if(copy[0] >= '0' && copy[0] <= '9')
{
    *(int *) &addressPtr->sin_addr = inet_addr(copy);
    return 0;
}
else return 1;
}

SOCKET dreamSock_OpenUDPSocket(char *netInterface, int port)
{
    SOCKET sock;

    struct sockaddr_in address;

    sock = dreamSock_Socket(DREAMSOCK_UDP);

    if(sock == DREAMSOCK_INVALID_SOCKET)
        return sock;

    dreamSock_SetNonBlocking(sock, 1);
    dreamSock_SetBroadcasting(sock, 1);

    // If no address string provided, use any interface available
    if(!netInterface || !netInterface[0] || !strcmp(netInterface,
        "localhost"))
    {
        LogString("No net interface given, using any interface
            available");
        address.sin_addr.s_addr = htonl(INADDR_ANY);
    }
    else
    {
        LogString("Using net interface = '%s'", netInterface);
        dreamSock_StringToSockaddr(netInterface, (struct sockaddr *)
            &address);
    }

    // If no port number provided, use any port number available
    if(port == 0)
    {
        LogString("No port defined, picking one for you");
        address.sin_port = 0;
    }
    else
    {

```

```
        address.sin_port = htons((short) port);
    }

    address.sin_family = AF_INET;

    // Bind the address to the socket
    if(bind(sock, (struct sockaddr *) &address, sizeof(address)) == -1)
    {
#ifdef Win32
        errno = WSAGetLastError();
        LogString("Error code %d: bind() : %s", errno, strerror(errno));
#else
        LogString("Error code %d: bind() : %s", errno, strerror(errno));
#endif

        return DREAMSOCK_INVALID_SOCKET;
    }

    // Get the port number (if we did not define one,
    // we get the assigned port number here)
    socklen_t len = sizeof(address);
    getsockname(sock, (struct sockaddr *) &address, &len);

    LogString("Opening UDP port = %d", ntohs(address.sin_port));

    return sock;
}

void dreamSock_CloseSocket(SOCKET sock)
{
#ifdef Win32
    closesocket(sock);
#else
    close(sock);
#endif
}

int dreamSock_GetPacket(SOCKET sock, char *data, struct sockaddr *from)
{
    int ret;
    struct sockaddr tempFrom;
    socklen_t fromlen;

    fromlen = sizeof(tempFrom);

    ret = recvfrom(sock, data, 1400, 0, (struct sockaddr *) &tempFrom,
                  &fromlen);

    // Copy the address to the from pointer
    if(from != NULL)
        *(struct sockaddr *) from = tempFrom;
}
```

```

        if(ret == -1)
        {
#ifdef Win32
            errno = WSAGetLastError();

            // Silently handle wouldblock
            if(errno == WSAEWOULDBLOCK)
                return ret;

            if(errno == WSAEMSGSIZE)
            {
                // ERROR: Oversize packet

                return ret;
            }

            LogString("Error code %d: recvfrom() : %s", errno,
                strerror(errno));
#else
            // Silently handle wouldblock
            if(errno == EWOULDBLOCK || errno == ECONNREFUSED)
                return ret;

            LogString("Error code %d: recvfrom() : %s", errno,
                strerror(errno));
#endif

            return ret;
        }

        return ret;
    }

void dreamSock_SendPacket(SOCKET sock, int length, char *data, struct
    sockaddr addr)
{
    int ret;

    ret = sendto(sock, data, length, 0, &addr, sizeof(addr));

    if(ret == -1)
    {
#ifdef Win32
        errno = WSAGetLastError();

        // Silently handle wouldblock
        if(errno == WSAEWOULDBLOCK)
            return;

        LogString("Error code %d: sendto() : %s", errno,
            strerror(errno));
#else
        // Silently handle wouldblock

```

```
        if(errno == EWOULDBLOCK)
            return;

        LogString("Error code %d: sendto() : %s", errno,
                strerror(errno));
    #endif
    }
}

void dreamSock_Broadcast(SOCKET sock, int length, char *data, int port)
{
    struct sockaddr_in servaddr;
    socklen_t len;

    // Use broadcast address
    u_long inetAddr = inet_addr("255.255.255.255");

    // Fill address information structure
    memset(&servaddr, 0, sizeof(struct sockaddr_in));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(port);
    servaddr.sin_addr.s_addr = inetAddr;

    len = sizeof(servaddr);

    // Broadcast!
    int ret = sendto(sock, data, length, 0, (struct sockaddr *) &servaddr,
                    len);

    if(ret == -1)
    {
    #ifdef Win32
        errno = WSAGetLastError();

        // Silently handle wouldblock
        if(errno == WSAEWOULDBLOCK)
            return;

        LogString("Error code %d: sendto() : %s", errno,
                strerror(errno));
    #else
        // Silently handle wouldblock
        if(errno == EWOULDBLOCK)
            return;

        LogString("Error code %d: sendto() : %s", errno,
                strerror(errno));
    #endif
    }
}

int dreamSock_GetCurrentSystemTime(void)
```

```

{
#ifdef Win32
    return dreamSock_Linux_GetCurrentSystemTime();
#else
    return dreamSock_Win_GetCurrentSystemTime();
#endif
}

```

dreamSock_Socket Function

This function creates a UDP or TCP socket, based on the given parameter.

```

SOCKET dreamSock_Socket(int protocol)
{
    int type;
    int proto;
    SOCKET sock;

    // Check which protocol to use
    if(protocol == DREAMSOCK_TCP)
    {
        type = SOCK_STREAM;
        proto = IPPROTO_TCP;
    }
    else
    {
        type = SOCK_DGRAM;
        proto = IPPROTO_UDP;
    }

    // Create the socket
    if((sock = socket(AF_INET, type, proto)) == -1)
    {
        LogString("dreamSock_Socket - socket() failed");
    }

#ifdef Win32
    errno = WSAGetLastError();
    LogString("Error: socket() code %d : %s", errno,
        strerror(errno));
#else
    LogString("Error: socket() : %s", strerror(errno));
#endif

    return DREAMSOCK_INVALID_SOCKET;
}

return sock;
}

```

This function takes one parameter (`int protocol`) and uses it to define what kind of socket to create. The actual library functionality does not include TCP networking, so `DREAMSOCK_TCP` is there for later use only. Of course, you can still create a TCP socket with this

function, and then use that socket with basic socket functions. So all this function really does is run the `socket` function and do some error checking for it.

dreamSock_SetNonBlocking Function

This function sets a socket blocking or non-blocking, based on the given parameters. Most of the time it is used to set a socket non-blocking, hence the name.

```
int dreamSock_SetNonBlocking(SOCKET sock, u_long setMode)
{
    u_long set = setMode;

    // Set the socket option
#ifdef Win32
    return ioctlsocket(sock, FIONBIO, &set);
#else
    return ioctl(sock, FIONBIO, &set);
#endif
}
```

The function takes two parameters (`SOCKET sock` and `u_long setMode`). The first one defines the socket to modify and the latter sets the mode. If `setMode` is 1, the socket is set non-blocking.

dreamSock_SetBroadcasting Function

This is used to set a socket to broadcast over a local area network.

```
int dreamSock_SetBroadcasting(SOCKET sock, int mode)
{
    // make it broadcast capable
    if(setsockopt(sock, SOL_SOCKET, SO_BROADCAST, (char *) &mode,
        sizeof(int)) == -1)
    {
        LogString("DreamSock_SetBroadcasting failed");
    }

#ifdef Win32
    int err = WSAGetLastError();
    LogString("Error code %d: setsockopt() : %s", err,
        strerror(err));
#else
    LogString("Error code %d: setsockopt() : %s", errno,
        strerror(errno));
#endif

    return DREAMSOCK_INVALID_SOCKET;
}

return 0;
}
```


Like `dreamSock_SetNonBlocking`, this function also has two parameters (`SOCKET sock` and `int mode`) that define the socket to modify and the mode to set. If `mode` is 1, the socket is set to broadcast.

Note A socket does not broadcast automatically even if it is set to broadcast. To broadcast over a local area network, you must use the broadcast address 255.255.255.255.

dreamSock_StringToSockaddr Function

This function converts the address string (for example, 192.168.0.1) to socket address format.

```
int dreamSock_StringToSockaddr(char *addressString, struct sockaddr *sadr)
{
    char copy[128];

    memset(sadr, 0, sizeof(struct sockaddr));

    struct sockaddr_in *addressPtr = (struct sockaddr_in *) sadr;

    addressPtr->sin_family = AF_INET;
    addressPtr->sin_port = htons(0);

    strcpy(copy, addressString);

    // If the address string begins with a number, assume an IP address
    if(copy[0] >= '0' && copy[0] <= '9')
    {
        *(int *) &addressPtr->sin_addr = inet_addr(copy);
        return 0;
    }
    else return 1;
}
```

This function takes two parameters (`char *addressString` and `struct sockaddr *sadr`). The first one is used to give the function the IP address string, which we want to convert to a socket address. The latter one is the socket address itself.

```
if(copy[0] >= '0' && copy[0] <= '9')
{
    *(int *) &addressPtr->sin_addr = inet_addr(copy);
    return 0;
}
else return 1;
```

The function checks whether the given address string actually is an IP string by checking if the first letter is a number. Otherwise, the function fails and returns 1 to indicate this.

dreamSock_OpenUDPSocket Function

Here is a function that does the magic to open a UDP socket and even binds a port to it while doing so.

```
SOCKET dreamSock_OpenUDPSocket(char *netInterface, int port)
{
    SOCKET sock;

    struct sockaddr_in address;

    sock = dreamSock_Socket(DREAMSOCK_UDP);

    if(sock == DREAMSOCK_INVALID_SOCKET)
        return sock;

    dreamSock_SetNonBlocking(sock, 1);
    dreamSock_SetBroadcasting(sock, 1);

    // If no address string provided, use any interface available
    if(!netInterface || !netInterface[0] || !strcmp(netInterface,
        "localhost"))
    {
        LogString("No net interface given, using any interface
            available");
        address.sin_addr.s_addr = htonl(INADDR_ANY);
    }
    else
    {
        LogString("Using net interface = '%s'", netInterface);
        dreamSock_StringToSockaddr(netInterface, (struct sockaddr *)
            &address);
    }

    // If no port number provided, use any port number available
    if(port == 0)
    {
        LogString("No port defined, picking one for you");
        address.sin_port = 0;
    }
    else
    {
        address.sin_port = htons((short) port);
    }

    address.sin_family = AF_INET;

    // Bind the address to the socket
    if(bind(sock, (struct sockaddr *) &address, sizeof(address)) == -1)
    {
#ifdef Win32
        errno = WSAGetLastError();
        LogString("Error code %d: bind() : %s", errno, strerror(errno));
#else

```

```

        LogString("Error code %d: bind() : %s", errno, strerror(errno));
#endif

        return DREAMSOCK_INVALID_SOCKET;
    }

    // Get the port number (if we did not define one,
    // we get the assigned port number here)
    socklen_t len = sizeof(address);
    getsockname(sock, (struct sockaddr *) &address, &len);

    LogString("Opening UDP port = %d", ntohs(address.sin_port));

    return sock;
}

```

The function takes two parameters (`char *netInterface, int port`). The first one defines which local IP address we want to use for this socket. Remember that a computer can indeed have more than one IP address (for example, if it has more than one network interface card). If this parameter is `NULL`, the default network interface is used.

```

if(!netInterface || !netInterface[0] || !strcmp(netInterface, "localhost"))
{
    LogString("No net interface given, using any interface available");
    address.sin_addr.s_addr = htonl(INADDR_ANY);
}
else
{
    LogString("Using net interface = '%s'", netInterface);
    dreamSock_StringToSockaddr(netInterface, (struct sockaddr *) &address);
}

```

The same happens for the port number. If no port is defined, the function lets the kernel pick one for you.

```

if(port == 0)
{
    LogString("No port defined, picking one for you");
    address.sin_port = 0;
}
else
{
    address.sin_port = htons((short) port);
}

```

We do not always know what port number the socket received, so we should get the address information. This is done in the following piece of code:

```

socklen_t len = sizeof(address);
getsockname(sock, (struct sockaddr *) &address, &len);

LogString("Opening UDP port = %d", ntohs(address.sin_port));

```

As you can see, this function uses many of the global functions we have introduced here. This makes the functionality simpler now that we know how the small parts of this function works. Nice, eh?

dreamSock_CloseSocket Function

At some point we most likely want to close a socket. Here is a function for that.

```
void dreamSock_CloseSocket(SOCKET sock)
{
#ifdef Win32
    closesocket(sock);
#else
    close(sock);
#endif
}
```

This is a simple function, but we still have to use two different basic socket functions in it. We must use different function calls on different platforms, but the functionality is the same; only the names of the functions are different.

dreamSock_GetPacket Function

We need a function for receiving data, and this is it.

```
int dreamSock_GetPacket(SOCKET sock, char *data, struct sockaddr *from)
{
    int ret;
    struct sockaddr tempFrom;
    socklen_t fromlen;

    fromlen = sizeof(tempFrom);

    ret = recvfrom(sock, data, 1400, 0, (struct sockaddr *) &tempFrom,
                  &fromlen);

    // Copy the address to the from pointer
    if(from != NULL)
        *(struct sockaddr *) from = tempFrom;

    if(ret == -1)
    {
#ifdef Win32
        errno = WSAGetLastError();

        // Silently handle wouldblock
        if(errno == WSAEWOULDBLOCK)
            return ret;

        if(errno == WSAEMSGSIZE)
        {
```

```

        // ERROR: Oversize packet
        return ret;
    }

    LogString("Error code %d: recvfrom() : %s", errno,
             strerror(errno));
#else
    // Silently handle wouldblock
    if(errno == EWOULDBLOCK || errno == ECONNREFUSED)
        return ret;

    LogString("Error code %d: recvfrom() : %s", errno,
             strerror(errno));
#endif

    return ret;
}

return ret;
}

```

This function retrieves the incoming data. It has three parameters (`SOCKET sock`, `char *data`, `struct sockaddr *from`). The first one is obviously used to define the socket to read from, the second one stores the incoming data, and the last one stores the sender's address information. So every time we receive data, we know the IP address and port of the sender. On a successful call to `recvfrom`, this function returns the amount of data read in bytes. If the socket is non-blocking and there is no incoming data, this function returns `-1`. If the socket is blocking, this function never returns until there is incoming data.

dreamSock_SendPacket Function

We probably want to send some data too, and this is the function we would use.

```

void dreamSock_SendPacket(SOCKET sock, int length, char *data, struct
                          sockaddr addr)
{
    int ret;

    ret = sendto(sock, data, length, 0, &addr, sizeof(addr));

    if(ret == -1)
    {
#ifdef Win32
        errno = WSAGetLastError();

        // Silently handle wouldblock
        if(errno == WSAEWOULDBLOCK)

```

```

        return;

        LogString("Error code %d: sendto() : %s", errno,
                  strerror(errno));
#else
        // Silently handle wouldblock
        if(errno == EWOULDBLOCK)
            return;

        LogString("Error code %d: sendto() : %s", errno,
                  strerror(errno));
#endif
    }
}

```

This function works pretty much the same way as `dreamSock_GetPacket`. The difference is the direction of the data. This time we send it. The function has four parameters (`SOCKET sock`, `int length`, `char *data`, `struct sockaddr addr`). The first defines the socket, the second defines the length of the data to send (in bytes), the third is the data itself, and the fourth is the address of the recipient. Again, this function returns the amount of data sent in bytes. If something went wrong, the function returns `-1`.

dreamSock_Broadcast Function

This function is not used in the `dreamSock` library, but you can use it for broadcasting.

```

void dreamSock_Broadcast(SOCKET sock, int length, char *data, int port)
{
    struct sockaddr_in servaddr;
    socklen_t len;

    // Use broadcast address
    u_long inetAddr = inet_addr("255.255.255.255");

    // Fill address information structure
    memset(&servaddr, 0, sizeof(struct sockaddr_in));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(port);
    servaddr.sin_addr.s_addr = inetAddr;

    len = sizeof(servaddr);

    // Broadcast!
    int ret = sendto(sock, data, length, 0, (struct sockaddr *)
                    &servaddr, len);

    if(ret == -1)
    {

```

```

#ifdef Win32
    errno = WSAGetLastError();

    // Silently handle wouldblock
    if(errno == WSAEWOULDBLOCK)
        return;

    LogString("Error code %d: sendto() : %s", errno,
              strerror(errno));
#else
    // Silently handle wouldblock
    if(errno == EWOULDBLOCK)
        return;

    LogString("Error code %d: sendto() : %s", errno,
              strerror(errno));
#endif
}

```

This function works the same way as `dreamSock_SendPacket`, but this time we do not define the entire address of the recipient, only the port. This is because we have to send to a broadcast address instead of a known remote address.

dreamSock_GetCurrentSystemTime Function

We need to be aware of time, and this is the function for that.

```

int dreamSock_GetCurrentSystemTime(void)
{
#ifdef Win32
    return dreamSock_Linux_GetCurrentSystemTime();
#else
    return dreamSock_Win_GetCurrentSystemTime();
#endif
}

```

This function wraps the platform-specific functions, which are introduced next.

dreamSock_Linux_GetCurrentSystemTime Function

This is the Unix/Linux version of `dreamSock_GetCurrentSystemTime`.

```

int dreamSock_Linux_GetCurrentSystemTime(void)
{
    struct timeval tp;
    struct timezone tzp;
    static int basetime;

```

```
gettimeofday(&tp, &tzp);

if(!basetime)
{
    basetime = tp.tv_sec;
    return tp.tv_usec / 1000;
}

return (tp.tv_sec - basetime) * 1000 + tp.tv_usec / 1000;
}
```

The function returns the current system time (starting from 0).

```
if(!basetime)
{
    basetime = tp.tv_sec;
    return tp.tv_usec / 1000;
}
```

In this piece of code we retrieve the base time, which is the current absolute time. This happens only once. After we have the base time value, we use it to calculate the application run time by subtracting the base time from the current time.

dreamSock_Win_GetCurrentSystemTime Function

This is the Windows version of `dreamSock_GetCurrentSystemTime`.

```
int dreamSock_Win_GetCurrentSystemTime(void)
{
    int curtime;
    static int base;
    static bool initialized = false;

    if(!initialized)
    {
        base = timeGetTime() & 0xffff0000;
        initialized = true;
    }

    curtime = timeGetTime() - base;

    return curtime;
}
```

This function returns the current system time (starting from 0).

```
if(!initialized)
{
    base = timeGetTime() & 0xffff0000;
    initialized = true;
}
```


In this piece of code we retrieve the base time, which is the current absolute time. This happens only once, and after we have the base-time value, we use it to calculate the application run time by subtracting the base time from the current time.

Retrieving Error Values

You most likely noticed that in most of the functions we check a variable called `errno`, which is obviously short for error number. This variable holds the error value number if something goes wrong. On Windows, we need to get the value ourselves, by using the following piece of code.

```
errno = WSAGetLastError();
```

On Unix systems, this is not required as the socket and other functions fill in `errno` for us.

We can then try to retrieve a text string concerning that error value since a plain number does not tell us much. This is done with the following piece of code:

```
strerror(errno)
```

This function does not always give us anything useful, but it is worth a try.

Summary of Global Functions

Those are all the global socket functions we need and which form the basis of our `dreamSock` network library. Remember that these functions can be used directly if you need to do something that is not done in the library.

Creating `dreamSock` Network Library

Now we will begin creating our network library by creating the three C++ classes that make the library come alive. We will start off by creating the message buffer handling system, `dreamMessage` class. Then we move on to the most complex system — the `dreamClient` class. The last one is the `dreamServer` class. Once these classes are ready, we can start writing our network application. So let's get kicking!

dreamMessage Class Member Variables

Each class has its own member variables. The following are the variables for `dreamMessage` and their usage:

```
private:
    bool        overFlow;
    int         maxSize;
    int         size;
    int         readCount;

public:
    char        *data;
    char        outgoingData[1400];
```

The boolean variable `overFlow` is used to store message size overflow information. Normally this variable is set to `FALSE`, but whenever we try to write more bytes into the message than the maximum size is, this variable is set to `TRUE`. When this happens, you cannot send the message. The integer `maxSize` defines the maximum size of the message in bytes. We can set this manually. The integer `size` tells us the current size of the message. The integer `readCount` stores the number of bytes read from the message so far.

The char `data` is a pointer to the message's data buffer. We must set this to point to an actual char buffer. The char `outgoingData` is a helper data buffer for outgoing data. We may make the data pointer point to this buffer, but it is not required.

dreamMessage Class Functionality

The `dreamMessage` class is used to create the messages that we send over the network. This class does not have the functionality to send the messages, only to build and read them. Let's see how everything works. The following code listing shows all the methods of the `dreamMessage` class:

```
void dreamMessage::Init(char *d, int length)
{
    data        = d;
    maxSize     = length;
    size        = 0;
    readCount   = 0;
    overFlow    = false;
}

void dreamMessage::Clear(void)
{
    size        = 0;
    readCount   = 0;
    overFlow    = false;
}
```

```
}

char *dreamMessage::GetNewPoint(int length)
{
    char *tempData;

    // Check for overflow
    if(size + length > maxSize)
    {
        Clear();
        overFlow = true;
    }

    tempData = data + size;
    size += length;

    return tempData;
}

void dreamMessage::AddSequences(dreamClient *client)
{
    WriteShort(client->GetOutgoingSequence());
    WriteShort(client->GetIncomingSequence());
}

void dreamMessage::Write(void *d, int length)
{
    memcpy(GetNewPoint(length), d, length);
}

void dreamMessage::WriteByte(char c)
{
    char *buf;

    buf = GetNewPoint(1);

    memcpy(buf, &c, 1);
}

void dreamMessage::WriteShort(short c)
{
    char *buf;

    buf = GetNewPoint(2);

    memcpy(buf, &c, 2);
}

void dreamMessage::WriteLong(long c)
{
    char *buf;

    buf = GetNewPoint(4);
}
```

```
        memcpy(buf, &c, 4);
    }

void dreamMessage::WriteFloat(float c)
{
    char *buf;

    buf = GetNewPoint(4);

    memcpy(buf, &c, 4);
}

void dreamMessage::WriteString(char *s)
{
    if(!s)
    {
        return;
    }
    else
        Write(s, strlen(s) + 1);
}

void dreamMessage::BeginReading(void)
{
    readCount = 0;
}

void dreamMessage::BeginReading(int s)
{
    size = s;
    readCount = 0;
}

char *dreamMessage::Read(int s)
{
    static char c[2048];

    if(readCount+s > size)
        return NULL;
    else
        memcpy(&c, &data[readCount], s);

    readCount += s;

    return c;
}

char dreamMessage::ReadByte(void)
{
    char c;

    if(readCount+1 > size)
        c = -1;
    else
```

```
        memcpy(&c, &data[readCount], 1);
    readCount++;
    return c;
}

short dreamMessage::ReadShort(void)
{
    short c;

    if(readCount+2 > size)
        c = -1;
    else
        memcpy(&c, &data[readCount], 2);

    readCount += 2;

    return c;
}

long dreamMessage::ReadLong(void)
{
    long c;

    if(readCount+4 > size)
        c = -1;
    else
        memcpy(&c, &data[readCount], 4);

    readCount += 4;

    return c;
}

float dreamMessage::ReadFloat(void)
{
    float c;

    if(readCount+4 > size)
        c = -1;
    else
        memcpy(&c, &data[readCount], 4);

    readCount += 4;

    return c;
}

char *dreamMessage::ReadString(void)
{
    static char string[2048];
    int  l, c;
```

```
    l = 0;

    do
    {
        c = ReadByte();

        if (c == -1 || c == 0)
            break;

        string[l] = c;
        l++;
    } while(l < sizeof(string)-1);

    string[l] = 0;

    return string;
}
```

Init Function

The `Init` function is pretty straightforward. All it does is initialize the class's member variables.

```
void dreamMessage::Init(char *d, int length)
{
    data         = d;
    maxSize      = length;
    size         = 0;
    readCount    = 0;
    overFlow     = false;
}
```

This function takes two parameters (`char *d, int length`). The first sets the buffer for the message's actual data. Whenever we write something to a message, we really write it into this buffer. Notice that `dreamMessage`'s `data` member variable is only a pointer. So we must have the buffer elsewhere, and then using `Init` function, we make the data pointer point to that buffer.

The second parameter defines the maximum size of a message in bytes. The function also resets some other variables.

Clear Function

The `Clear` function resets the message's member variables without modifying the data buffer.

```
void dreamMessage::Clear(void)
{
    size         = 0;
    readCount    = 0;
    overFlow     = false;
}
```

GetNewPoint Function

This function gets a new point from the data buffer based on the given parameter.

```
char *dreamMessage::GetNewPoint(int length)
{
    char *tempData;

    // Check for overflow
    if(size + length > maxSize)
    {
        Clear();
        overFlow = true;
    }

    tempData = data + size;
    size += length;

    return tempData;
}
```

This function takes only one parameter (`int length`), which defines the point in the buffer to retrieve. The function then returns a pointer that points to that position. With this function we can move around in the data buffer and write data to its correct location.

The function checks if we are trying to write more than the maximum size of the message. If yes, the message is cleared and the overflow flag is raised.

```
if(size + length > maxSize)
{
    Clear();
    overFlow = true;
}
```

The following piece of code is essential to this function. The temporary pointer `tempData` is set to point to the message's data pointer and this pointer is moved by the current size of the message. Now we have a completely new, untouched position in the data buffer. Well, that may not always be true because the data buffer itself may hold some old data at that position. But because we have the variable that holds the current size of the message, we do not need to worry about the old data. We will never read over to the old data part, and we will always write on top of the old one.

```
tempData = data + size;
size += length;
```

AddSequences Function

Understanding what this function does requires us to understand the `dreamClient` class first, so do not worry about this function yet. All you need to know now is that this function writes two shorts into the message's data buffer. These two shorts are the client's message sequence numbers. There is more about this later in this tutorial.

```
void dreamMessage::AddSequences(dreamClient *client)
{
    WriteShort(client->GetOutgoingSequence());
    WriteShort(client->GetIncomingSequence());
}
```

This function takes one parameter (`dreamClient *client`). This parameter is a pointer to a `dreamClient` object. Each client uses this function to add its own message sequences to the message.

Write Function

Here we have the first writing function. This is a generic write function that lets us write any type of data to the message.

```
void dreamMessage::Write(void *d, int length)
{
    memcpy(GetNewPoint(length), d, length);
}
```

This function takes two parameters (`void *d, int length`), which define the data to write and the length of the data. Normally when we write data, we use the other write functions that are discussed in this section. When we use those functions, we do not need to define the size of the data ourselves because the size is known and is always the same. But when we use the generic `Write` function, we have to define the size, since it can be anything.

The function gets a new point in the data buffer and copies the given data to that point.

WriteByte Function

The `WriteByte` function writes a byte into the message.

```
void dreamMessage::WriteByte(char c)
{
    char *buf;

    buf = GetNewPoint(1);

    memcpy(buf, &c, 1);
}
```


This function takes the data to write as a parameter and then writes it to the buffer.

WriteShort Function

The `WriteShort` function writes a short into the message.

```
void dreamMessage::WriteShort(short c)
{
    char *buf;

    buf = GetNewPoint(2);

    memcpy(buf, &c, 2);
}
```

This function takes the data to write as a parameter and writes it to the buffer. A short takes 2 bytes.

WriteLong Function

The `WriteLong` function writes a long into the message.

```
void dreamMessage::WriteLong(long c)
{
    char *buf;

    buf = GetNewPoint(4);

    memcpy(buf, &c, 4);
}
```

This function takes the data to write as a parameter and writes it to the buffer. A long takes 4 bytes.

WriteFloat Function

The `WriteFloat` function writes a float into the message.

```
void dreamMessage::WriteFloat(float c)
{
    char *buf;

    buf = GetNewPoint(4);

    memcpy(buf, &c, 4);
}
```

This function takes the data to write as a parameter and then writes it to the buffer. A float takes 4 bytes.

WriteString Function

The `WriteString` function writes a string into the message.

```
void dreamMessage::WriteString(char *s)
{
    if(!s)
    {
        return;
    }
    else
        Write(s, strlen(s) + 1);
}
```

This function takes the data to write as a parameter and writes it to the buffer. If the string is `NULL`, nothing is written. We do not need to define the length of the string, because we can use the `strlen` function to get that.

BeginReading Function

There are two versions of the `BeginReading` function. They both set the current size of the message.

```
void dreamMessage::BeginReading(void)
{
    readCount = 0;
}

void dreamMessage::BeginReading(int s)
{
    size = s;
    readCount = 0;
}
```

The purpose of this function is to reset the `readCount` member variable so reading begins from the beginning. The second version takes one parameter (`int s`) that defines the current size of the message.

Read Function

This function reads a defined amount of data.

```
char *dreamMessage::Read(int s)
{
    static char c[2048];

    if(readCount+s > size)
        return NULL;
    else
        memcpy(&c, &data[readCount], s);

    readCount += s;
}
```

```
        return c;
    }
```

This function has one parameter (`int s`) that defines the amount of data to read. Member variable `readCount` keeps track of the current position in the data buffer.

The function checks for buffer overflow, as shown in the following piece of code.

```
if(readCount+s > size)
    return NULL;
else
    memcpy(&c, &data[readCount], s);
```

If we try to read more data than the message holds, nothing is read.

ReadByte Function

This function reads a byte from the data buffer.

```
char dreamMessage::ReadByte(void)
{
    char c;

    if(readCount+1 > size)
        c = -1;
    else
        memcpy(&c, &data[readCount], 1);

    readCount++;

    return c;
}
```

Once the reading is done, the function returns a byte that holds the data it just read. The function checks for buffer overflow.

ReadShort Function

This function reads a short from the data buffer.

```
short dreamMessage::ReadShort(void)
{
    short c;

    if(readCount+2 > size)
        c = -1;
    else
        memcpy(&c, &data[readCount], 2);

    readCount += 2;

    return c;
}
```

Once the reading is done, the function returns a short that holds the data it just read. The function checks for buffer overflow.

ReadLong Function

This function reads a long from the data buffer.

```
long dreamMessage::ReadLong(void)
{
    long c;

    if(readCount+4 > size)
        c = -1;
    else
        memcpy(&c, &data[readCount], 4);

    readCount += 4;

    return c;
}
```

Once the reading is done, the function returns a long that holds the data it just read. The function checks for buffer overflow.

ReadFloat Function

This function reads a float from the data buffer.

```
float dreamMessage::ReadFloat(void)
{
    float c;

    if(readCount+4 > size)
        c = -1;
    else
        memcpy(&c, &data[readCount], 4);

    readCount += 4;

    return c;
}
```

Once the reading is done, the function returns a float that holds the data it just read. The function checks for buffer overflow.

ReadString Function

This function reads a string from the data buffer.

```
char *dreamMessage::ReadString(void)
{
    static char string[2048];
    int l, c;
```



```
    unsigned short    droppedPackets;    // Dropped packets

    int               serverPort;        // Port
    char              serverIP[32];      // IP address
    int               index;             // Client index (starts from 1,
                                        // running number)
    char              name[32];          // Client name

    SOCKET            socket;            // Socket
    struct sockaddr    myaddress;        // Socket address

    int               pingSent;          // When did we send ping?
    int               ping;              // Network latency

    int               lastMessageTime;

    bool              init;

public:
    dreamMessage      message;
    dreamClient       *next;
```

The integer `connectionState` tells us the current state of the connection. It can be one of these: `connecting`, `connected`, `disconnecting`, or `disconnected`. The unsigned short `outgoingSequence` keeps track of the message numbers that we have sent. The first one has number or sequence 1, the next one 2, and so on. The unsigned short `incomingSequence` keeps track of received messages' sequence numbers. This number is set every time we receive a message from a remote host. The sequence is set to match the remote host's `outgoingSequence`. This may be old information though, because the sequence is picked from the last received message, and before we got that message, the remote host may have sent another packet. But that is not usually a problem so do not worry about it. The unsigned short `incomingAcknowledged` keeps track of the last packet the remote host acknowledged. This is actually the remote host's `incomingSequence`. Now we have a nice little loop so we don't have to worry about dropped packets. If a packet is dropped, it is lost, but the sequence numbers will not get mixed up. The unsigned short `droppedPackets` stores information about the amount of dropped packets this frame. The integer `serverPort` stores the server's port number. The char `serverIP` stores the server's IP address. The integer `index` is unique for every client. The server gives this number to us. The number is incremented starting from 1. Every time a client connects the server, this number increases and the new client gets that new number. This number is unique to each client since the index number is never decreased on the server. The char `name` stores the name of this client.

The `SOCKET` `socket` variable holds the socket for each client. This is not used on the server side. The struct `sockaddr_myaddress` is the socket's socket address.

The integer `pingSent` tells us the time we sent our ping message. The integer `ping` tells us the network latency. The integer `lastMessageTime` tells us when the server received the last message from this client. The boolean `Init` is a flag that tells us whether the client has been initialized.

The dreamMessage `message` is the client's internal message. This is more like a helper variable than a requirement. We can use an external message to send data, but we also can use this one. The dreamClient `next` is a pointer to the next client in the client list. This is used on the server only, as clients do not have client lists.

dreamClient Class Functionality

The `dreamClient` class is used on both the server side and the client side. On the server side, there is a list of clients that controls all the communication with the real clients. On the client side, this class is used to connect to the server. Here are all the methods of `dreamClient`:

```
dreamClient::dreamClient()
{
    connectionState      = DREAMSOCK_DISCONNECTED;

    outgoingSequence     = 1;
    incomingSequence     = 0;
    incomingAcknowledged = 0;
    droppedPackets       = 0;

    init                 = false;

    serverPort           = 0;

    pingSent             = 0;
    ping                 = 0;

    lastMessageTime     = 0;

    next = NULL;
}

dreamClient::~~dreamClient()
{
    dreamSock_CloseSocket(socket);
}

int dreamClient::Initialize(char *localIP, char *remoteIP, int port)
{
    // Initialize dreamSock if it is not already initialized
```

```
dreamSock_Initialize();

// Save server's address information for later use
serverPort = port;
strcpy(serverIP, remoteIP);

LogString("Server's information: IP address: %s, port: %d", serverIP,
         serverPort);

// Create client socket
socket = dreamSock_OpenUDPSocket(localIP, 0);

// Check that the address is not empty
u_long inetAddr = inet_addr(serverIP);

if(inetAddr == INADDR_NONE)
{
    return DREAMSOCK_CLIENT_ERROR;
}

if(socket == DREAMSOCK_INVALID_SOCKET)
{
    return DREAMSOCK_CLIENT_ERROR;
}

init = true;

return 0;
}

void dreamClient::Uninitialize(void)
{
    dreamSock_CloseSocket(socket);

    Reset();

    init = false;
}

void dreamClient::Reset(void)
{
    connectionState      = DREAMSOCK_DISCONNECTED;

    outgoingSequence     = 1;
    incomingSequence     = 0;
    incomingAcknowledged = 0;
    droppedPackets       = 0;

    pingSent             = 0;
    ping                 = 0;

    lastMessageTime     = 0;

    next = NULL;
}
```



```
}

void dreamClient::DumpBuffer(void)
{
    char data[1400];
    int ret;

    while((ret = dreamSock_GetPacket(socket, data, NULL)) > 0)
    {
    }
}

void dreamClient::SendConnect(char *name)
{
    // Dump buffer so there won't be any old packets to process
    DumpBuffer();

    connectionState = DREAMSOCK_CONNECTING;

    message.Init(message.outgoingData, sizeof(message.outgoingData));
    message.WriteByte(DREAMSOCK_MES_CONNECT);
    message.WriteString(name);

    SendPacket(&message);
}

void dreamClient::SendDisconnect(void)
{
    message.Init(message.outgoingData, sizeof(message.outgoingData));
    message.WriteByte(DREAMSOCK_MES_DISCONNECT);

    SendPacket(&message);
    Reset();

    connectionState = DREAMSOCK_DISCONNECTING;
}

void dreamClient::SendPing(void)
{
    pingSent = dreamSock_GetCurrentSystemTime();

    message.Init(message.outgoingData, sizeof(message.outgoingData));
    message.WriteByte(DREAMSOCK_MES_PING);

    SendPacket(&message);
}

void dreamClient::ParsePacket(dreamMessage *mes)
{
    mes->BeginReading();
    int type = mes->ReadByte();

    // Check if the type is a positive number
    // = is the packet sequenced
```

```
if(type > 0)
{
    unsigned short sequence      = mes->ReadShort();
    unsigned short sequenceAck   = mes->ReadShort();

    if(sequence <= incomingSequence)
    {
        LogString("Client: (sequence: %d <= incoming seq: %d)",
            sequence, incomingSequence);

        LogString("Client: Sequence mismatch");
    }

    droppedPackets = sequence - incomingSequence + 1;

    incomingSequence = sequence;
    incomingAcknowledged = sequenceAck;
}

// Parse through the system messages
switch(type)
{
case DREAMSOCK_MES_CONNECT:
    connectionState = DREAMSOCK_CONNECTED;

    LogString("LIBRARY: Client: got connect confirmation");
    break;

case DREAMSOCK_MES_DISCONNECT:
    connectionState = DREAMSOCK_DISCONNECTED;

    LogString("LIBRARY: Client: got disconnect confirmation");
    break;

case DREAMSOCK_MES_ADDCLIENT:
    LogString("LIBRARY: Client: adding a client");
    break;

case DREAMSOCK_MES_REMOVECLIENT:
    LogString("LIBRARY: Client: removing a client");
    break;

case DREAMSOCK_MES_PING:
    SendPing();
    break;
}
}

int dreamClient::GetPacket(char *data, struct sockaddr *from)
{
    // Check if the client is set up or if it is disconnecting
    if(!socket)
        return 0;
}
```

```

    int ret;

    dreamMessage mes;
    mes.Init(data, sizeof(data));

    ret = dreamSock_GetPacket(socket, mes.data, from);

    if(ret <= 0)
        return 0;

    mes.SetSize(ret);

    // Parse system messages
    ParsePacket(&mes);

    return ret;
}

void dreamClient::SendPacket(void)
{
    // Check that everything is set up
    if(!socket || connectionState == DREAMSOCK_DISCONNECTED)
    {
        LogString("SendPacket error: Could not send because the client
            is disconnected");
        return;
    }

    // If the message overflowed, do not send it
    if(message.GetOverflow())
    {
        LogString("SendPacket error: Could not send because the buffer
            overflowed");
        return;
    }

    // Check if serverPort is set. If it is, we are a client sending to
    // the server. Otherwise we are a server sending to a client.
    if(serverPort)
    {
        struct sockaddr_in sendToAddress;
        memset((char *) &sendToAddress, 0, sizeof(sendToAddress));

        u_long inetAddr = inet_addr(serverIP);
        sendToAddress.sin_port = htons((u_short) serverPort);
        sendToAddress.sin_family = AF_INET;
        sendToAddress.sin_addr.s_addr = inetAddr;

        dreamSock_SendPacket(socket, message.GetSize(), message.data,
            *(struct sockaddr *) &sendToAddress);
    }
    else
    {
        dreamSock_SendPacket(socket, message.GetSize(), message.data,

```

```
        myaddress);
    }

    // Check if the packet is sequenced
    message.BeginReading();
    int type = message.ReadByte();

    if(type > 0)
    {
        outgoingSequence++;
    }
}

void dreamClient::SendPacket(dreamMessage *theMes)
{
    // Check that everything is set up
    if(!socket || connectionState == DREAMSOCK_DISCONNECTED)
    {
        LogString("SendPacket error: Could not send because the client
            is disconnected");
        return;
    }

    // If the message overflowed, do not send it
    if(theMes->GetOverflow())
    {
        LogString("SendPacket error: Could not send because the buffer
            overflowed");
        return;
    }

    // Check if serverPort is set. If it is, we are a client sending to
    // the server. Otherwise we are a server sending to a client.
    if(serverPort)
    {
        struct sockaddr in sendToAddress;
        memset((char *)&sendToAddress, 0, sizeof(sendToAddress));

        u_long inetAddr = inet_addr(serverIP);
        sendToAddress.sin_port = htons((u_short) serverPort);
        sendToAddress.sin_family = AF_INET;
        sendToAddress.sin_addr.s_addr = inetAddr;

        dreamSock_SendPacket(socket, theMes->GetSize(), theMes->data,
            *(struct sockaddr *) &sendToAddress);
    }
    else
    {
        dreamSock_SendPacket(socket, theMes->GetSize(), theMes->data,
            myaddress);
    }

    // Check if the packet is sequenced
    theMes->BeginReading();
}
```

```

        int type = theMes->ReadByte();

        if(type > 0)
        {
            outgoingSequence++;
        }
    }
}

```

dreamClient Constructor

The `dreamClient` constructor sets up everything for network connection.

```

dreamClient::dreamClient()
{
    connectionState      = DREAMSOCK_DISCONNECTED;

    outgoingSequence     = 1;
    incomingSequence     = 0;
    incomingAcknowledged = 0;
    droppedPackets       = 0;

    init                 = false;

    serverPort          = 0;
    pingSent            = 0;
    ping                = 0;
    lastMessageTime     = 0;

    next = NULL;
}

```

dreamClient Destructor

The destructor does nothing but try to close the socket. It does not matter if the socket is already closed.

```

dreamClient::~~dreamClient()
{
    dreamSock_CloseSocket(socket);
}

```

Initialize Function

This function initializes the client and readies it for a network connection with the server. This function does not send anything to the server, as it only sets up the client. This is a client-side function only.

```

int dreamClient::Initialize(char *localIP, char *remoteIP, int port)
{
    // Initialize dreamSock if it is not already initialized
    dreamSock_Initialize();

    // Save server's address information for later use
}

```

```
serverPort = port;
strcpy(serverIP, remoteIP);

LogString("Server's information: IP address: %s, port: %d", serverIP,
serverPort);

// Create client socket
socket = dreamSock_OpenUDPsocket(localIP, 0);

// Check that the address is not empty
u_long inetAddr = inet_addr(serverIP);

if(inetAddr == INADDR_NONE)
{
    return DREAMSOCK_CLIENT_ERROR;
}

if(socket == DREAMSOCK_INVALID_SOCKET)
{
    return DREAMSOCK_CLIENT_ERROR;
}

init = true;

return 0;
}
```

This function takes three parameters (`char *localIP`, `char *remoteIP`, and `int port`). The first parameter defines the local IP address to use (if we have multiple network interface cards). If this is `NULL`, the default is used. The second parameter is the server's IP address and the third one is the server's port number.

The function then opens a UDP socket using the given information.

```
socket = dreamSock_OpenUDPsocket(localIP, 0);
```

After this function successfully returns, the client is ready to start sending data to the server.

Uninitialize Function

This function uninitializes the client's network connection ability.

```
void dreamClient::Uninitialize(void)
{
    dreamSock_CloseSocket(socket);

    Reset();

    init = false;
}
```

The socket is closed and all the member variables are reset with the `Reset` function.

Reset Function

The `Reset` function does what the name tells you — it resets the client.

```
void dreamClient::Reset(void)
{
    connectionState      = DREAMSOCK_DISCONNECTED;

    outgoingSequence     = 1;
    incomingSequence     = 0;
    incomingAcknowledged = 0;
    droppedPackets       = 0;

    pingSent             = 0;
    ping                 = 0;
    lastMessageTime      = 0;

    next = NULL;
}
```

All the important member variables are set to their initial state so a connection process can be started all over again if required.

DumpBuffer Function

This function's purpose is to dump the incoming data buffer by reading all the incoming packets and just dumping them (not processing them at all).

```
void dreamClient::DumpBuffer(void)
{
    char data[1400];
    int ret;

    while((ret = dreamSock_GetPacket(socket, data, NULL)) > 0)
    {
    }
}
```

The buffer is dumped by calling `dreamSock_GetPacket` as long there is anything to read. This way a new connection will not get the packets from any old connections.

System Messages vs. User Messages

Now let's take a moment to talk about system messages and user messages. A system message is a message that is built into the system (in this case, into our network library). A user message, on the other hand, is the user's own message, and all the processing for that happens in the final network application, not in the library.

Every message's first byte must contain its type. A message cannot be identified without having a type value attached to it. This is the first thing that we read from a packet, and once we have identified the message type we can move on to the correct direction.

We will create five system messages for `dreamSock` — one for connecting to the server, one for disconnecting from the server, one for adding a client, and one for removing a client. The fifth is for pinging clients to calculate the network latency. They are defined as follows:

```
#define DREAMSOCK_MES_CONNECT      -101
#define DREAMSOCK_MES_DISCONNECT  -102
#define DREAMSOCK_MES_ADDCLIENT   -103
#define DREAMSOCK_MES_REMOVECLIENT -104
#define DREAMSOCK_MES_PING        -105
```

Did you notice that the messages' type value is actually negative? This is used to tell `dreamSock` that the message should not be sequenced. And that means that no sequence numbers are attached to the messages. This works for user messages also; just make the type value negative. We will see how all this works later.

`dreamSock` will then process these messages and act accordingly. All the system messages can be reprocessed in the final application. This is very useful as we can then use the same messages for adding a client, for example. You may wonder why we would need to add a client there also, if that is done in the library already. Well, the answer is the game's data for each client. Since `dreamClient` only holds the information required for the network connection, we probably need another structure to hold the game data. But enough about that for now.

We will now create some functions for `dreamSock` that process the system messages. So when we want to connect a server, we just call one function. Simple, but effective.

SendConnect Function

This function sends a connection request to the server.

```
void dreamClient::SendConnect(char *name)
{
    // Dump buffer so there won't be any old packets to process
    DumpBuffer();

    connectionState = DREAMSOCK_CONNECTING;

    message.Init(message.outgoingData, sizeof(message.outgoingData));
    message.WriteByte(DREAMSOCK_MES_CONNECT);
    message.WriteString(name);

    SendPacket(&message);
}
```


The function takes one parameter (`char *name`), which is used to define the name of the client. The function will add this name to the message and then send it to the server. When the server receives this message, it will add a client to its client list and send an “add client” message to each connected client.

As you can see, we use the `dreamMessage` class to create the message. First we initialize it by setting the data buffer and its maximum size. Then we write one byte to it, which tells the system the type of the message. Finally we add the name string to the message and send the packet to the server. The `SendPacket` function is introduced later in this tutorial.

```
message.Init(message.outgoingData, sizeof(message.outgoingData));
message.WriteByte(DREAMSOCK_MES_CONNECT);
message.WriteString(name);

SendPacket(&message);
```

SendDisconnect Function

This function sends a disconnect message to the server.

```
void dreamClient::SendDisconnect(void)
{
    message.Init(message.outgoingData, sizeof(message.outgoingData));
    message.WriteByte(DREAMSOCK_MES_DISCONNECT);

    SendPacket(&message);
    Reset();

    connectionState = DREAMSOCK_DISCONNECTING;
}
```

This function works the same way as `SendConnect` does. It sends a system message to the server, telling it that we want to disconnect. After the message is sent, the client resets itself.

SendPing Function

This function sends a ping message to the server.

```
void dreamClient::SendPing(void)
{
    pingSent = dreamSock_GetCurrentSystemTime();

    message.Init(message.outgoingData, sizeof(message.outgoingData));
    message.WriteByte(DREAMSOCK_MES_PING);

    SendPacket(&message);
}
```

This function is run when the server pings the client first. The function responds to the server by pinging back.

ParsePacket Function

This function parses incoming system messages and handles sequenced messages.

```
void dreamClient::ParsePacket(dreamMessage *mes)
{
    mes->BeginReading();
    int type = mes->ReadByte();

    // Check if the type is a positive number
    // = is the packet sequenced
    if(type > 0)
    {
        unsigned short sequence      = mes->ReadShort();
        unsigned short sequenceAck   = mes->ReadShort();

        if(sequence <= incomingSequence)
        {
            LogString("Client: (sequence: %d <= incoming seq: %d)",
                    sequence, incomingSequence);

            LogString("Client: Sequence mismatch");
        }

        droppedPackets = sequence - incomingSequence + 1;

        incomingSequence = sequence;
        incomingAcknowledged = sequenceAck;
    }

    // Parse through the system messages
    switch(type)
    {
    case DREAMSOCK_MES_CONNECT:
        connectionState = DREAMSOCK_CONNECTED;

        LogString("LIBRARY: Client: got connect confirmation");
        break;

    case DREAMSOCK_MES_DISCONNECT:
        connectionState = DREAMSOCK_DISCONNECTED;

        LogString("LIBRARY: Client: got disconnect confirmation");
        break;

    case DREAMSOCK_MES_ADDCLIENT:
        LogString("LIBRARY: Client: adding a client");
        break;
    }
```

```

        case DREAMSOCK_MES_REMOVECLIENT:
            LogString("LIBRARY: Client: removing a client");
            break;

        case DREAMSOCK_MES_PING:
            SendPing();
            break;
    }
}

```

This function takes one parameter (`dreamMessage *mes`). It is a pointer to the message to parse.

Parsing begins by reading the type of the message from the packet:

```

mes->BeginReading();
int type = mes->ReadByte();

```

Then we check if the type value is a positive number — in other words, whether the message is sequenced. If it is, we read the sequence numbers from the packet. First is the remote host's outgoing sequence number and then its incoming sequence number. We call them incoming sequence and acknowledged sequence. Sound confusing? Well, at first it may be so, but if you really think about it for a while, it is quite simple.

```

if(type > 0)
{
    unsigned short sequence      = mes->ReadShort();
    unsigned short sequenceAck   = mes->ReadShort();

    if(sequence <= incomingSequence)
    {
        LogString("Client: (sequence: %d <= incoming seq: %d)",
                 sequence, incomingSequence);

        LogString("Client: Sequence mismatch");
    }

    droppedPackets = sequence - incomingSequence + 1;

    incomingSequence = sequence;
    incomingAcknowledged = sequenceAck;
}

```

Then we parse the system messages. On the client side, there is nothing important going on here, if you do not count the connection state changes and ping response. These are mainly confirmations only.

```

// Parse through the system messages
switch(type)
{
    case DREAMSOCK_MES_CONNECT:
        connectionState = DREAMSOCK_CONNECTED;

```

```
        LogString("LIBRARY: Client: got connect confirmation");
        break;

    case DREAMSOCK_MES_DISCONNECT:
        connectionState = DREAMSOCK_DISCONNECTED;

        LogString("LIBRARY: Client: got disconnect confirmation");
        break;

    case DREAMSOCK_MES_ADDCLIENT:
        LogString("LIBRARY: Client: adding a client");
        break;

    case DREAMSOCK_MES_REMOVECLIENT:
        LogString("LIBRARY: Client: removing a client");
        break;

    case DREAMSOCK_MES_PING:
        SendPing();
        break;
}

```

GetPacket Function

This function reads packets from the server and then moves them on to parsing.

```
int dreamClient::GetPacket(char *data, struct sockaddr *from)
{
    // Check if the client is set up or if it is disconnecting
    if(!socket)
        return 0;

    int ret;

    dreamMessage mes;
    mes.Init(data, sizeof(data));

    ret = dreamSock_GetPacket(socket, mes.data, from);

    if(ret <= 0)
        return 0;

    mes.SetSize(ret);

    // Parse system messages
    ParsePacket(&mes);

    return ret;
}

```

This function takes two parameters (`char *data`, `struct sockaddr *from`). The first one is a pointer to the data buffer, which will be filled by this function. The other is a socket address pointer,

which will also be filled by this function. So when we receive a packet, we also get the address from where it came. This information can be ignored, but it is useful to know.

We read the data into a `dreamMessage` so we can then parse it in the `ParsePacket` function. If `dreamSock_GetPacket` returns a negative number or 0, we do not process the packet any further, because there is no packet to process.

```
dreamMessage mes;
mes.Init(data, sizeof(data));

ret = dreamSock_GetPacket(socket, mes.data, from);

if(ret <= 0)
    return 0;

mes.SetSize(ret);

// Parse system messages
ParsePacket(&mes);
```

SendPacket Function (Internal Message)

There are two versions of the `SendPacket` function. This one sends the internal message of `dreamClient` (`message` member variable). This function can be run on both the server side and the client side. In fact, the server sends packets to clients with this very function.

```
void dreamClient::SendPacket(void)
{
    // Check that everything is set up
    if(!socket || connectionState == DREAMSOCK_DISCONNECTED)
    {
        LogString("SendPacket error: Could not send because the client
            is disconnected");
        return;
    }

    // If the message overflowed, do not send it
    if(message.GetOverflow())
    {
        LogString("SendPacket error: Could not send because the buffer
            overflowed");
        return;
    }

    // Check if serverPort is set. If it is, we are a client sending to
    // the server. Otherwise we are a server sending to a client.
    if(serverPort)
    {
        struct sockaddr_in sendToAddress;
```

```

memset((char *) &sendToAddress, 0, sizeof(sendToAddress));

u_long inetAddr = inet_addr(serverIP);
sendToAddress.sin_port = htons((u_short) serverPort);
sendToAddress.sin_family = AF_INET;
sendToAddress.sin_addr.s_addr = inetAddr;

dreamSock_SendPacket(socket, message.GetSize(), message.data,
                    *(struct sockaddr *) &sendToAddress);
}
else
{
    dreamSock_SendPacket(socket, message.GetSize(), message.data,
                        myaddress);
}

// Check if the packet is sequenced
message.BeginReading();
int type = message.ReadByte();

if(type > 0)
{
    outgoingSequence++;
}
}

```

First we do some checking that it is okay to send data. We check that we have a socket and that we are not disconnected. If the message overflowed, we will not send it because some parts of the message are missing.

```

// Check that everything is set up
if(!socket || connectionState == DREAMSOCK_DISCONNECTED)
{
    LogString("SendPacket error: Could not send because the client is
              disconnected");
    return;
}

// If the message overflowed, do not send it
if(message.GetOverflow())
{
    LogString("SendPacket error: Could not send because the buffer
              overflowed");
    return;
}

```

Then we check if we are a server trying to send to a client or vice versa. An easy way to check this is to see if the server port number is set. If it is, we are a client sending to the server, because only clients set the `serverPort` variable. So we fill in the server's address information and send the packet to that address. On the server side we just

use the client's address, which is set when the client connects to the server.

Notice that the data we send comes from the `dreamClient`'s `message` member variable.

```
if(serverPort)
{
    struct sockaddr_in sendToAddress;
    memset((char *) &sendToAddress, 0, sizeof(sendToAddress));

    u_long inetAddr = inet_addr(serverIP);
    sendToAddress.sin_port = htons((u_short) serverPort);
    sendToAddress.sin_family = AF_INET;
    sendToAddress.sin_addr.s_addr = inetAddr;

    dreamSock_SendPacket(socket, message.GetSize(), message.data,
                        *(struct sockaddr *) &sendToAddress);
}
else
{
    dreamSock_SendPacket(socket, message.GetSize(), message.data,
                        myaddress);
}
```

Finally we increase the outgoing sequence number, if the message is sequenced.

```
// Check if the packet is sequenced
message.BeginReading();
int type = message.ReadByte();

if(type > 0)
{
    outgoingSequence++;
}
```

SendPacket Function (External Message)

This function sends an external message given as a parameter. It can be run on both the server side and the client side.

```
void dreamClient::SendPacket(dreamMessage *theMes)
{
    // Check that everything is set up
    if(!socket || connectionState == DREAMSOCK_DISCONNECTED)
    {
        LogString("SendPacket error: Could not send because the client
                  is disconnected");
        return;
    }

    // If the message overflowed, do not send it
    if(theMes->GetOverflow())
```

```

    {
        LogString("SendPacket error: Could not send because the buffer
                overflowed");
        return;
    }

    // Check if serverPort is set. If it is, we are a client sending to
    // the server. Otherwise we are a server sending to a client.
    if(serverPort)
    {
        struct sockaddr_in sendToAddress;
        memset((char *) &sendToAddress, 0, sizeof(sendToAddress));

        u_long inetAddr = inet_addr(serverIP);
        sendToAddress.sin_port = htons((u_short) serverPort);
        sendToAddress.sin_family = AF_INET;
        sendToAddress.sin_addr.s_addr = inetAddr;

        dreamSock_SendPacket(socket, theMes->GetSize(), theMes->data,
                            *(struct sockaddr *) &sendToAddress);
    }
    else
    {
        dreamSock_SendPacket(socket, theMes->GetSize(), theMes->data,
                            myaddress);
    }

    // Check if the packet is sequenced
    theMes->BeginReading();
    int type = theMes->ReadByte();

    if(type > 0)
    {
        outgoingSequence++;
    }
}

```

This function works exactly the same way as the internal message version. The only difference is that this function takes one parameter (`dreamMessage *theMes`), which is the message to send.

```
dreamSock_SendPacket(socket, theMes->GetSize(), theMes->data, myaddress);
```

dreamClient Summary

That concludes the `dreamClient` methods. Now we have the client side fully ready, but we cannot do anything with it yet, because the server side is not ready. So what are we waiting for? Let's go finish our library!

dreamServer Class Member Variables

Here are the member variables for `dreamServer` and their purposes.

```
private:
    dreamClient *clientList;

    int         port;           // Port
    SOCKET      socket;        // Socket
    int         runningIndex;   // Running index numbers for new clients

    bool        init;
```

The `dreamClient` `clientList` is a linked list of all the connected clients. If it is `NULL`, no clients are connected. The integer `port` stores the port number of the server. The `SOCKET` `socket` holds the socket of the server. All network communication happens through this socket.

The integer `runningIndex` is an index number for incoming clients. Every time a client connects to the server, this number is increased and given to the client as its index number. The number is unique for every client. The boolean `init` tells us whether the server has been initialized.

dreamServer Class Functionality

The `dreamServer` class is used only on the server side. This class handles the incoming clients and their network connections. Here are all the methods of `dreamServer`:

```
dreamServer::dreamServer()
{
    init                = false;

    port                = 0;
    runningIndex        = 1;
    socket               = 0;
    clientList          = NULL;
}

dreamServer::~~dreamServer()
{
    dreamClient *list = clientList;
    dreamClient *next;

    while(list != NULL)
    {
        next = list->next;

        if(list)
        {
            free(list);
        }
    }
}
```

```
        }

        list = next;
    }

    clientList = NULL;

    dreamSock_CloseSocket(socket);
}

int dreamServer::Initialize(char *localIP, int serverPort)
{
    // Initialize dreamSock if it is not already initialized
    dreamSock_Initialize();

    // Store the server IP and port for later use
    port = serverPort;

    // Create server socket
    socket = dreamSock_OpenUDPSocket(localIP, port);

    if(socket == DREAMSOCK_INVALID_SOCKET)
    {
        return DREAMSOCK_SERVER_ERROR;
    }

    init = true;

    return 0;
}

void dreamServer::Uninitialize(void)
{
    dreamSock_CloseSocket(socket);

    init = false;
}

void dreamServer::SendAddClient(dreamClient *newClient)
{
    // Send connection confirmation
    newClient->message.Init(newClient->message.outgoingData,
        sizeof(newClient->message.outgoingData));

    newClient->message.WriteByte(DREAMSOCK_MES_CONNECT);    // type
    newClient->SendPacket();

    // Send 'Add client' message to every client
    dreamClient *client = clientList;

    // First inform the new client of the other clients
    for(; client != NULL; client = client->next)
    {
```

```

newClient->message.Init(newClient->message.outgoingData,
    sizeof(newClient->message.outgoingData));

newClient->message.WriteByte(DREAMSOCK_MES_ADDCLIENT); // type

if(client == newClient)
{
    newClient->message.WriteByte(1); // local client
    newClient->message.WriteByte(client->GetIndex());
    newClient->message.WriteString(client->GetName());
}
else
{
    newClient->message.WriteByte(0); // not local client
    newClient->message.WriteByte(client->GetIndex());
    newClient->message.WriteString(client->GetName());
}

newClient->SendPacket();
}

// Then tell the others about the new client
for(client = clientList; client != NULL; client = client->next)
{
    if(client == newClient)
        continue;

    client->message.Init(client->message.outgoingData,
        sizeof(client->message.outgoingData));

    client->message.WriteByte(DREAMSOCK_MES_ADDCLIENT); // type

    client->message.WriteByte(0);
    client->message.WriteByte(newClient->GetIndex());
    client->message.WriteString(newClient->GetName());

    client->SendPacket();
}
}

void dreamServer::SendRemoveClient(dreamClient *client)
{
    int index = client->GetIndex();

    // Send 'Remove client' message to every client
    dreamClient *list = clientList;

    for(; list != NULL; list = list->next)
    {
        list->message.Init(list->message.outgoingData,
            sizeof(list->message.outgoingData));

        list->message.WriteByte(DREAMSOCK_MES_REMOVECLIENT); // type
        list->message.WriteByte(index); // index
    }
}

```

```
    }

    SendPackets();

    // Send disconnection confirmation
    client->message.Init(client->message.outgoingData,
        sizeof(client->message.outgoingData));

    client->message.WriteByte(DREAMSOCK_MES_DISCONNECT);
    client->SendPacket();
}

void dreamServer::SendPing(void)
{
    // Send ping message to every client
    dreamClient *list = clientList;

    for(; list != NULL; list = list->next)
    {
        list->SendPing();
    }
}

void dreamServer::AddClient(struct sockaddr *address, char *name)
{
    // First get a pointer to the beginning of client list
    dreamClient *list = clientList;
    dreamClient *prev;
    dreamClient *newClient;

    LogString("LIB: Adding client, index %d", runningIndex);

    // No clients yet, adding the first one
    if(clientList == NULL)
    {
        LogString("LIB: Server: Adding first client");

        clientList = (dreamClient *) calloc(1, sizeof(dreamClient));

        clientList->SetSocket(socket);
        clientList->SetSocketAddress(address);

        clientList->SetConnectionState(DREAMSOCK_CONNECTING);
        clientList->SetOutgoingSequence(1);
        clientList->SetIncomingSequence(0);
        clientList->SetIncomingAcknowledged(0);
        clientList->SetIndex(runningIndex);
        clientList->SetName(name);
        clientList->next = NULL;

        newClient = clientList;
    }
    else
    {
```

```
    LogString("LIB: Server: Adding another client");

    prev = list;
    list = clientList->next;

    while(list != NULL)
    {
        prev = list;
        list = list->next;
    }

    list = (dreamClient *) calloc(1, sizeof(dreamClient));

    list->SetSocket(socket);
    list->SetSocketAddress(address);

    list->SetConnectionState(DREAMSOCK_CONNECTING);
    list->SetOutgoingSequence(1);
    list->SetIncomingSequence(0);
    list->SetIncomingAcknowledged(0);
    list->SetIndex(runningIndex);
    list->SetName(name);
    list->next = NULL;

    prev->next = list;

    newClient = list;
}

runningIndex++;

SendAddClient(newClient);
}

void dreamServer::RemoveClient(dreamClient *client)
{
    dreamClient *list = NULL;
    dreamClient *prev = NULL;
    dreamClient *next = NULL;

    int index = client->GetIndex();

    LogString("LIB: Removing client with index %d", index);

    SendRemoveClient(client);

    for(list = clientList; list != NULL; list = list->next)
    {
        if(client == list)
        {
            if(prev != NULL)
            {
                prev->next = client->next;
            }
        }
    }
}
```

```
        }
        break;
    }
    prev = list;
}

if(client == clientList)
{
    LogString("LIB: Server: removing first client in list");

    if(list) next = list->next;

    if(client) free(client);
    client = NULL;
    clientList = next;
}
else
{
    LogString("LIB: Server: removing a client");

    if(list) next = list->next;

    if(client) free(client);
    client = next;
}
}

void dreamServer::ParsePacket(dreamMessage *mes, struct sockaddr *address)
{
    mes->BeginReading();
    int type = mes->ReadByte();

    // Find the correct client by comparing addresses
    dreamClient *cList = clientList;

    // If we do not have clients yet, skip to message type checking
    if(cList != NULL)
    {
        for(; cList != NULL; cList = cList->next)
        {
            if(memcmp(cList->GetSocketAddress(), address,
                sizeof(address)) == 0)
            {
                break;
            }
        }

        if(cList != NULL)
        {
            cList->SetLastMessageTime(dreamSock_
                GetCurrentSystemTime());
        }
    }
}
```

```

        // Check if the type is a positive number
        // -> is the packet sequenced
        if(type > 0)
        {
            unsigned short sequence      = mes->ReadShort();
            unsigned short sequenceAck    = mes->ReadShort();

            if(sequence <= cList->GetIncomingSequence())
            {
                LogString("LIB: Server: Sequence mismatch
                    (sequence: %ld <= incoming seq: %ld)",
                    sequence, cList->GetIncomingSequence());
            }

            cList->SetDroppedPackets(sequence -
                cList->GetIncomingSequence() - 1);

            cList->SetIncomingSequence(sequence);
            cList->SetIncomingAcknowledged(sequenceAck);
        }

        // Wait for one message before setting state to connected
        if(cList->GetConnectionState() == DREAMSOCK_CONNECTING)
            cList->SetConnectionState(DREAMSOCK_CONNECTED);
    }

    // Parse through the system messages
    switch(type)
    {
    case DREAMSOCK_MES_CONNECT:
        AddClient(address, mes->ReadString());

        LogString("LIBRARY: Server: a client connected successfully");
        break;

    case DREAMSOCK_MES_DISCONNECT:
        if(cList == NULL)
            break;
    }

    RemoveClient(cList);

    LogString("LIBRARY: Server: a client disconnected");
    break;

    case DREAMSOCK_MES_PING:
        cList->SetPing(dreamSock_GetCurrentSystemTime() -
            cList->GetPingSent());
        break;
}

int dreamServer::CheckForTimeout(char *data, struct sockaddr *from)
{
    int currentTime = dreamSock_GetCurrentSystemTime();

```

```
dreamClient *cList = clientList;
dreamClient *next;

for(; cList != NULL;)
{
    next = cList->next;

    // Don't timeout when connecting
    if(cList->GetConnectionState() == DREAMSOCK_CONNECTING)
    {
        cList = next;
        continue;
    }

    // Check if the client has been silent for 30 seconds
    // If yes, assume crashed and remove the client
    if(currentTime - cList->GetLastMessageTime() > 30000)
    {
        LogString("Client timeout, disconnecting (%d - %d = %d)",
            currentTime, cList->GetLastMessageTime(),
            currentTime - cList->GetLastMessageTime());

        // Build a 'fake' message so the application will also
        // receive notification of a client disconnecting
        dreamMessage mes;
        mes.Init(data, sizeof(data));
        mes.WriteByte(DREAMSOCK_MES_DISCONNECT);

        *(struct sockaddr *) from = *cList->GetSocketAddress();

        RemoveClient(cList);

        return mes.GetSize();
    }

    cList = next;
}

return 0;
}

int dreamServer::GetPacket(char *data, struct sockaddr *from)
{
    // Check if the server is set up
    if(!socket)
        return 0;

    // Check for timeout
    int timeout = CheckForTimeout(data, from);

    if(timeout)
        return timeout;
}
```



```
// Wait for a while or incoming data
int maxfd = socket;
fd_set allset;
struct timeval waittime;

waittime.tv_sec = 10 / 1000;
waittime.tv_usec = (10 % 1000) * 1000;

FD_ZERO(&allset);
FD_SET(socket, &allset);

fd_set reading = allset;

int nready = select(maxfd + 1, &reading, NULL, NULL, &waittime);

if(!nready)
    return 0;

// Read data of the socket
int ret = 0;

dreamMessage mes;
mes.Init(data, sizeof(data));

ret = dreamSock_GetPacket(socket, mes.data, from);

if(ret <= 0)
    return 0;

mes.SetSize(ret);

// Parse system messages
ParsePacket(&mes, from);

return ret;
}

void dreamServer::SendPackets(void)
{
    // Check if the server is set up
    if(!socket)
        return;

    dreamClient *cList = clientList;

    for(; cList != NULL; cList = cList->next)
    {
        if(cList->message.GetSize() == 0)
            continue;

        cList->SendPacket();
    }
}
```

dreamServer Constructor

The dreamServer constructor makes everything ready for clients to join the server.

```
dreamServer::dreamServer()
{
    init            = false;

    port            = 0;
    runningIndex    = 1;
    socket          = 0;
    clientList      = NULL;
}
```

dreamServer Destructor

The dreamServer destructor frees all the allocated memory, if any.

```
dreamServer::~~dreamServer()
{
    dreamClient *list = clientList;
    dreamClient *next;

    while(list != NULL)
    {
        next = list->next;

        if(list)
        {
            free(list);
        }

        list = next;
    }

    clientList = NULL;

    dreamSock_CloseSocket(socket);
}
```

The client list is parsed through and all the allocated memory is freed. The server socket is also closed.

Note that before we free the memory of a client in the list, we must store its “next” pointer to an external pointer, because the original is lost once the memory is freed. Then we start it all over again.

```
next = list->next;

if(list)
{
    free(list);
```

```
}  
  
list = next;
```

Initialize Function

This function initializes `dreamServer` by creating the socket so clients start connecting to it.

```
int dreamServer::Initialize(char *localIP, int serverPort)  
{  
    // Initialize dreamSock if it is not already initialized  
    dreamSock_Initialize();  
  
    // Store the server IP and port for later use  
    port = serverPort;  
  
    // Create server socket  
    socket = dreamSock_OpenUDPSocket(localIP, port);  
  
    if(socket == DREAMSOCK_INVALID_SOCKET)  
    {  
        return DREAMSOCK_SERVER_ERROR;  
    }  
  
    init = true;  
  
    return 0;  
}
```

The function takes two parameters (`char *localIP` and `int serverPort`). The first one defines the local IP address we want to use. If this is `NULL`, the default is used. The latter parameter sets the server port.

Uninitialize Function

Here we uninitialize `dreamServer` by closing the socket.

```
void dreamServer::Uninitialize(void)  
{  
    dreamSock_CloseSocket(socket);  
  
    init = false;  
}
```

SendAddClient Function

This function sends an “add client” message to each connected client and a connection confirmation to the new client.

```
void dreamServer::SendAddClient(dreamClient *newClient)  
{
```

```
// Send connection confirmation
newClient->message.Init(newClient->message.outgoingData,
    sizeof(newClient->message.outgoingData));

newClient->message.WriteByte(DREAMSOCK_MES_CONNECT); // type
newClient->SendPacket();

// Send 'Add client' message to every client
dreamClient *client = clientList;

// First inform the new client of the other clients
for(; client != NULL; client = client->next)
{
    newClient->message.Init(newClient->message.outgoingData,
        sizeof(newClient->message.outgoingData));

    newClient->message.WriteByte(DREAMSOCK_MES_ADDCLIENT); // type

    if(client == newClient)
    {
        newClient->message.WriteByte(1); // local client
        newClient->message.WriteByte(client->GetIndex());
        newClient->message.WriteString(client->GetName());
    }
    else
    {
        newClient->message.WriteByte(0); // not local client
        newClient->message.WriteByte(client->GetIndex());
        newClient->message.WriteString(client->GetName());
    }

    newClient->SendPacket();
}

// Then tell the others about the new client
for(client = clientList; client != NULL; client = client->next)
{
    if(client == newClient)
        continue;

    client->message.Init(client->message.outgoingData,
        sizeof(client->message.outgoingData));

    client->message.WriteByte(DREAMSOCK_MES_ADDCLIENT); // type

    client->message.WriteByte(0);
    client->message.WriteByte(newClient->GetIndex());
    client->message.WriteString(newClient->GetName());

    client->SendPacket();
}
}
```

This function takes one parameter (`dreamClient *newClient`). This is a pointer to the client that connected the server and triggered this function to be run.

First we send a connect confirmation to the new client:

```
newClient->message.Init(newClient->message.outgoingData,
    sizeof(newClient->message.outgoingData));

newClient->message.WriteByte(DREAMSOCK_MES_CONNECT);    // type
newClient->SendPacket();
```

Then we send an “add client” message to all the connected clients (including the new client). This tells the clients to add a client to the game. With this message we tell the clients their index numbers and send a local client flag. This means that the new client knows to assign a local client pointer to its own client list.

```
if(client == newClient)
{
    newClient->message.WriteByte(1);    // local client
    newClient->message.WriteByte(client->GetIndex());
    newClient->message.WriteString(client->GetName());
}
else
{
    newClient->message.WriteByte(0);    // not local client
    newClient->message.WriteByte(client->GetIndex());
    newClient->message.WriteString(client->GetName());
}
```

SendRemoveClient Function

Here we send a “remove client” message to each client.

```
void dreamServer::SendRemoveClient(dreamClient *client)
{
    int index = client->GetIndex();

    // Send 'Remove client' message to every client
    dreamClient *list = clientList;

    for(; list != NULL; list = list->next)
    {
        list->message.Init(list->message.outgoingData,
            sizeof(list->message.outgoingData));

        list->message.WriteByte(DREAMSOCK_MES_REMOVECLIENT);    // type
        list->message.WriteByte(index);                            // index
    }

    SendPackets();

    // Send disconnection confirmation
```

```

        client->message.Init(client->message.outgoingData,
                            sizeof(client->message.outgoingData));

        client->message.WriteByte(DREAMSOCK_MES_DISCONNECT);
        client->SendPacket();
    }

```

This function takes one parameter (`dreamClient *client`), which is a pointer to the client to remove. First we tell all the clients to remove the client and then we send the disconnecting client a confirmation of disconnection.

SendPing Function

This function pings the clients to calculate the network latency.

```

void dreamServer::SendPing(void)
{
    // Send ping message to every client
    dreamClient *list = clientList;

    for(; list != NULL; list = list->next)
    {
        list->SendPing();
    }
}

```

What this function actually does is call the `SendPing` function of each client. The server then sends the ping message to each client.

AddClient Function

This function adds a client to the server's client list. The function is run when a client connects to the server.

```

void dreamServer::AddClient(struct sockaddr *address, char *name)
{
    // First get a pointer to the beginning of client list
    dreamClient *list = clientList;
    dreamClient *prev;
    dreamClient *newClient;

    LogString("LIB: Adding client, index %d", runningIndex);

    // No clients yet, adding the first one
    if(clientList == NULL)
    {
        LogString("LIB: Server: Adding first client");

        clientList = (dreamClient *) calloc(1, sizeof(dreamClient));

        clientList->SetSocket(socket);
        clientList->SetSocketAddress(address);
    }
}

```

```

        clientList->SetConnectionState(DREAMSOCK_CONNECTING);
        clientList->SetOutgoingSequence(1);
        clientList->SetIncomingSequence(0);
        clientList->SetIncomingAcknowledged(0);
        clientList->SetIndex(runningIndex);
        clientList->SetName(name);
        clientList->next = NULL;

        newClient = clientList;
    }
    else
    {
        LogString("LIB: Server: Adding another client");

        prev = list;
        list = clientList->next;

        while(list != NULL)
        {
            prev = list;
            list = list->next;
        }

        list = (dreamClient *) calloc(1, sizeof(dreamClient));

        list->SetSocket(socket);
        list->SetSocketAddress(address);

        list->SetConnectionState(DREAMSOCK_CONNECTING);
        list->SetOutgoingSequence(1);
        list->SetIncomingSequence(0);
        list->SetIncomingAcknowledged(0);
        list->SetIndex(runningIndex);
        list->SetName(name);
        list->next = NULL;

        prev->next = list;

        newClient = list;
    }

    runningIndex++;

    SendAddClient(newClient);
}

```

The function takes two parameters (`struct sockaddr *address` and `char *name`). The first one is the socket address of the new client, and the second one is the client's name. This information is then added to the client's own structure. After the server has added the information to its own list, it sends the "add client" message to all the connected clients.

The function checks if the client list is empty; if it is, it creates the first entry for it. All the same information is stored for each client. The following piece of code shows the first entry being added.

```
// No clients yet, adding the first one
if(clientList == NULL)
{
    LogString("LIB: Server: Adding first client");

    clientList = (dreamClient *) calloc(1, sizeof(dreamClient));

    clientList->SetSocket(socket);
    clientList->SetSocketAddress(address);

    clientList->SetConnectionState(DREAMSOCK_CONNECTING);
    clientList->SetOutgoingSequence(1);
    clientList->SetIncomingSequence(0);
    clientList->SetIncomingAcknowledged(0);
    clientList->SetIndex(runningIndex);
    clientList->SetName(name);
    clientList->next = NULL;

    newClient = clientList;
}
```

RemoveClient Function

This function removes a client from the client list.

```
void dreamServer::RemoveClient(dreamClient *client)
{
    dreamClient *list = NULL;
    dreamClient *prev = NULL;
    dreamClient *next = NULL;

    int index = client->GetIndex();

    LogString("LIB: Removing client with index %d", index);

    SendRemoveClient(client);

    for(list = clientList; list != NULL; list = list->next)
    {
        if(client == list)
        {
            if(prev != NULL)
            {
                prev->next = list->next;
            }

            break;
        }
    }

    prev = list;
}
```



```

    }

    if(client == clientList)
    {
        LogString("LIB: Server: removing first client in list");

        if(list) next = list->next;

        if(client) free(client);
        client = NULL;
        clientList = next;
    }
    else
    {
        LogString("LIB: Server: removing a client");

        if(list) next = list->next;

        if(client) free(client);
        client = next;
    }
}

```

The function takes one parameter (`dreamClient *client`). This is a pointer to the client to remove. The server first sends each client a message to remove the client and then removes the client itself. We must update the client list so that the previous client's "next" pointer does not point to the removed client anymore. We want it to point to the next client in the list.

```

for(list = clientList; list != NULL; list = list->next)
{
    if(client == list)
    {
        if(prev != NULL)
        {
            prev->next = list->next;
        }

        break;
    }

    prev = list;
}

```

We also need to make sure that if the first entry on the list is removed, the next one in the list becomes the new first entry.

```

if(client == clientList)
{
    LogString("LIB: Server: removing first client in list");

    if(list) next = list->next;
}

```

```

        if(client) free(client);
        client = NULL;
        clientList = next;
    }
    else
    {
        LogString("LIB: Server: removing a client");

        if(list) next = list->next;

        if(client) free(client);
        client = next;
    }
}

```

ParsePacket Function

This function parses the server's system messages and handles sequenced messages.

```

void dreamServer::ParsePacket(dreamMessage *mes, struct sockaddr *address)
{
    mes->BeginReading();
    int type = mes->ReadByte();

    // Find the correct client by comparing addresses
    dreamClient *cList = clientList;

    // If we do not have clients yet, skip to message type checking
    if(cList != NULL)
    {
        for(; cList != NULL; cList = cList->next)
        {
            if(memcmp(cList->GetSocketAddress(), address,
                sizeof(address)) == 0)
            {
                break;
            }
        }

        if(cList != NULL)
        {
            cList->SetLastMessageTime(dreamSock_
                GetCurrentSystemTime());

            // Check if the type is a positive number
            // -> is the packet sequenced
            if(type > 0)
            {
                unsigned short sequence = mes->ReadShort();
                unsigned short sequenceAck = mes->ReadShort();

                if(sequence <= cList->GetIncomingSequence())
                {

```

```

        LogString("LIB: Server: Sequence mismatch
            (sequence: %ld <= incoming seq: %ld)",
                sequence, cList->GetIncomingSequence());
    }

    cList->SetDroppedPackets(sequence -
        cList->GetIncomingSequence() - 1);

    cList->SetIncomingSequence(sequence);
    cList->SetIncomingAcknowledged(sequenceAck);
}

// Wait for one message before setting state to connected
if(cList->GetConnectionState() == DREAMSOCK_CONNECTING)
    cList->SetConnectionState(DREAMSOCK_CONNECTED);
}

}

// Parse through the system messages
switch(type)
{
case DREAMSOCK_MES_CONNECT:
    AddClient(address, mes->ReadString());

    LogString("LIBRARY: Server: a client connected successfully");
    break;

case DREAMSOCK_MES_DISCONNECT:
    if(cList == NULL)
        break;

    RemoveClient(cList);

    LogString("LIBRARY: Server: a client disconnected");
    break;

case DREAMSOCK_MES_PING:
    cList->SetPing(dreamSock_GetCurrentSystemTime() -
        cList->GetPingSent());
    break;
}
}
}

```

The function takes two parameters (`dreamMessage *mes` and `struct sockaddr *address`). The first one is a pointer to the message to parse. The second one is the socket address of the client from which we got the message. We compare this address to the ones in the client list, and if it matches one there, the message is from an old client. If it does not find a match, the client is new. Note that we just break the search loop if we find a match, and then we use the last pointer as our client pointer.

```

for(; cList != NULL; cList = cList->next)
{
    if(memcmp(cList->GetSocketAddress(), address, sizeof(address)) == 0)
    {
        break;
    }
}

```

The sequences are processed just like on the client side. We also control the connection state of a client here. When a client connects, its connection state is set to “connecting.” When we get a message from this client again, we set the connection state to “connected.” So we wait for one message from the client before we give it a “connected” state. This is to make sure we do not time out the client before it has even had time to connect properly.

```

// Wait for one message before setting state to connected
if(cList->GetConnectionState() == DREAMSOCK_CONNECTING)
    cList->SetConnectionState(DREAMSOCK_CONNECTED);

```

Finally, we parse the system messages. The ping value is calculated by subtracting the “ping sent” time from the current time.

```

// Parse through the system messages
switch(type)
{
case DREAMSOCK_MES_CONNECT:
    AddClient(address, mes->ReadString());

    LogString("LIBRARY: Server: a client connected successfully");
    break;

case DREAMSOCK_MES_DISCONNECT:
    if(cList == NULL)
        break;

    RemoveClient(cList);

    LogString("LIBRARY: Server: a client disconnected");
    break;

case DREAMSOCK_MES_PING:
    cList->SetPing(dreamSock_GetCurrentSystemTime() -
        cList->GetPingSent());
    break;
}

```

CheckForTimeout Function

Here we check if a client times out. This means that the server has not received any message from the client in a certain amount of time.

```

int dreamServer::CheckForTimeout(char *data, struct sockaddr *from)
{

```

```

int currentTime = dreamSock_GetCurrentSystemTime();

dreamClient *cList = clientList;
dreamClient *next;

for(; cList != NULL;)
{
    next = cList->next;

    // Don't timeout when connecting
    if(cList->GetConnectionState() == DREAMSOCK_CONNECTING)
    {
        cList = next;
        continue;
    }

    // Check if the client has been silent for 30 seconds
    // If yes, assume crashed and remove the client
    if(currentTime - cList->GetLastMessageTime() > 30000)
    {
        LogString("Client timeout, disconnecting (%d - %d = %d)",
            currentTime, cList->GetLastMessageTime(),
            currentTime - cList->GetLastMessageTime());

        // Build a 'fake' message so the application will also
        // receive notification of a client disconnecting
        dreamMessage mes;
        mes.Init(data, sizeof(data));
        mes.WriteByte(DREAMSOCK_MES_DISCONNECT);

        *(struct sockaddr *) from = *cList->GetSocketAddress();

        RemoveClient(cList);

        return mes.GetSize();
    }

    cList = next;
}

return 0;
}

```

This function takes two parameters (`char *data` and `struct sockaddr *from`). These parameters are used to create a “fake” message to tell everybody a client timed out. Actually, all they will know is that the client disconnected, because this fake message makes the server believe it got a “disconnect” message from the client that timed out.

We should not time out while the client is connecting.

```

// Don't timeout when connecting
if(cList->GetConnectionState() == DREAMSOCK_CONNECTING)

```

```

{
    cList = next;
    continue;
}

```

Here we check if the client has been silent for 30 seconds; if it has, we create the fake message to make the server remove the client.

```

// Check if the client has been silent for 30 seconds
// If yes, assume crashed and remove the client
if(currentTime - cList->GetLastMessageTime() > 30000)
{
    LogString("Client timeout, disconnecting (%d - %d = %d)",
             currentTime, cList->GetLastMessageTime(), currentTime -
             cList->GetLastMessageTime());

    // Build a 'fake' message so the application will also
    // receive notification of a client disconnecting
    dreamMessage mes;
    mes.Init(data, sizeof(data));
    mes.WriteByte(DREAMSOCK_MES_DISCONNECT);

    *(struct sockaddr *) from = *cList->GetSocketAddress();

    RemoveClient(cList);

    return mes.GetSize();
}

```

GetPacket Function

This function receives packets from the clients.

```

int dreamServer::GetPacket(char *data, struct sockaddr *from)
{
    // Check if the server is set up
    if(!socket)
        return 0;

    // Check for timeout
    int timeout = CheckForTimeout(data, from);

    if(timeout)
        return timeout;

    // Wait for a while or incoming data
    int maxfd = socket;
    fd_set allset;
    struct timeval waittime;

    waittime.tv_sec = 10 / 1000;
    waittime.tv_usec = (10 % 1000) * 1000;

    FD_ZERO(&allset);

```

```

    FD_SET(socket, &allset);

    fd_set reading = allset;

    int nready = select(maxfd + 1, &reading, NULL, NULL, &waittime);

    if(!nready)
        return 0;

    // Read data of the socket
    int ret = 0;

    dreamMessage mes;
    mes.Init(data, sizeof(data));

    ret = dreamSock_GetPacket(socket, mes.data, from);

    if(ret <= 0)
        return 0;

    mes.SetSize(ret);

    // Parse system messages
    ParsePacket(&mes, from);

    return ret;
}

```

The function takes two parameters (`char *data` and `struct sockaddr *from`). The first one is a pointer to the incoming data buffer, which holds the actual data that comes from the client. The second one is the socket address of the client from which we got the data.

First we check for timeouts, and then we start to listen to the server's socket. We use the `select` function to save processor time by waiting for incoming data. If no data is received in 10 milliseconds, the function returns. If we did not do this, the server would consume all the processor time, and the server machine could not run anything else efficiently.

```

// Wait for a while or incoming data
int maxfd = socket;
fd_set allset;
struct timeval waittime;

waittime.tv_sec = 10 / 1000;
waittime.tv_usec = (10 % 1000) * 1000;

FD_ZERO(&allset);
FD_SET(socket, &allset);

fd_set reading = allset;

```

```
int nready = select(maxfd + 1, &reading, NULL, NULL, &waittime);

if(!nready)
    return 0;
```

After that we read the socket's data and parse for system messages.

```
// Read data of the socket
int ret = 0;

dreamMessage mes;
mes.Init(data, sizeof(data));

ret = dreamSock_GetPacket(socket, mes.data, from);

if(ret <= 0)
    return 0;

mes.SetSize(ret);

// Parse system messages
ParsePacket(&mes, from);

return ret;
```

SendPackets Function

This function sends the internal messages of all the clients in the client list. The server should write the messages into these internal messages inside `dreamClient` and then send them all at once with this function.

```
void dreamServer::SendPackets(void)
{
    // Check if the server is set up
    if(!socket)
        return;

    dreamClient *cList = clientList;

    for(; cList != NULL; cList = cList->next)
    {
        if(cList->message.GetSize() == 0)
            continue;

        cList->SendPacket();
    }
}
```

This function goes through all of the clients in the server's client list and sends off all of their internal messages by using `dreamClient's SendPacket` function.

dreamServer Summary

We have now created dreamServer, which processes all the clients on the server side, and we know how to send to clients and how to receive messages from them.

Summary

That concludes our network library tutorial. We created the dream-Message class to build and read network messages, the dreamClient class to handle client-side and server-side network connections, and the dreamServer class to handle the clients on the server side.

We now have a fully working library ready, and we can start writing our multiplayer game.

Tutorial 3

Creating a Basic Network Application with dreamSock

Introduction

In this tutorial we learn how to create a basic network application using dreamSock. This application will be the sign-in part of our tutorial game. We do not talk about the actual sign-in process that happens with MySQL but instead learn how to create a server application and a client application that work together over a network. We have our network library dreamSock ready, so we will be using that for the network communications.

Planning the Functionality

We are now going to create a basic dreamSock network application in the form of the sign-in server of our tutorial game. The final game will have three such applications: sign-in, lobby, and the game itself. So when the game is ready, the server will actually be running more than one service. Sign-in and lobby services are running all the time, and the game services are run when needed (when games are created).

All these parts of the application work exactly the same way; only the messages are different.

On both the server and client we have a loop that reads the socket every frame. Messages are sent as needed.

Catching Exceptions

Catching exceptions from the loops we run is a very, very good idea, especially on the server. Imagine a client connecting to the server and then somehow crashing and “confusing” the server code (hopefully our code is good enough not to do that though). It is possible that the server does something that it should not do, such as trying to use some unallocated memory. Without catching exceptions, the server would crash completely, leaving the clients hanging. But if we use try and catch statements to catch this unplanned exception, the server code will ignore the malfunction and move on. The parts of the code that depended on what the server was trying to do will also fail, but the server will keep running and serving all the other clients. Also, when a client catches an exception, it can safely disconnect from the server before it is shut down.

Creating a Basic Client Application

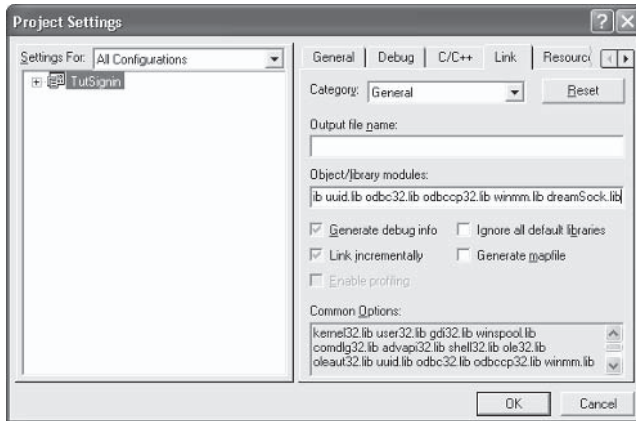
We will now create a basic client application, which is a normal Windows dialog-based program. We have the dialogs ready, so all we need to do now is link our network library with the program we are creating.

To do this, select Project, Settings from the main Visual Studio menu. Then select the Link tab in the dialog that appears in the middle of the screen. Ensure that the top-left drop-down box is set to All Configurations, then add the following library before or after the other libraries in the Object/library modules edit box:

```
dreamSock.lib
```

When you enter these libraries, the dialog should look like Figure 1.

Figure 1



Once this is done, we can make new source code files called `main.cpp`, `signin.cpp`, `main.h`, `common.h`, `network.h`, and `signin.h`. We will write our client code into these files.

signin.h File

It is a good idea to start with header files, because that is where our data structures are, and we need those before we can do anything with the application. So let's take a look at the `signin.h` file that contains class `CSignIn`.

```
#ifndef __SIGNIN_H__
#define __SIGNIN_H__

#include "network.h"
#include "main.h"

#define SIGNIN_RESULT_ACCEPTED          200
#define SIGNIN_RESULT_USERNAMEBAD      201
#define SIGNIN_RESULT_PASSWORDBAD     202
#define SIGNIN_RESULT_MYSQLERROR      203

class CSignIn
{
private:
    dreamClient      *networkClient;
    clientLoginData *clientList;
    clientLoginData *localClient;    // Pointer to the local client
                                        // in the client list

public:
    CSignIn();
};
```

```

    dreamClient      *GetNetworkClient(void) {return networkClient;}
    clientLoginData  *GetLocalClient(void)  {return localClient;}

    void  ReadPackets(void);

    void  AddClient(int local, int index, char *name);
    void  RemoveClient(int index);
    void  RemoveClients(void);

    void  SendSignIn(char *nickname, char *firstname, char *surname,
                    int age, char *gender, char *password, char *email);
    void  SendKeepAlive(void);

    void  Connect(char *name);
    void  Disconnect(void);

    void  RunNetwork(int msec);
};

extern CSignin Signin;

#endif

```

CSignin Class

Every dreamSock network application should have a class similar to this one (or many such classes as you learn in the other tutorials). This is the network client's base class, because it holds the dreamClient object and other members that are closely connected to networking. We look at these members later in this tutorial.

```

class CSignin
{
private:
    dreamClient      *networkClient;
    clientLoginData  *clientList;
    clientLoginData  *localClient;    // Pointer to the local client
                                     // in the client list

public:
    CSignin();

    dreamClient      *GetNetworkClient(void) {return networkClient;}
    clientLoginData  *GetLocalClient(void)  {return localClient;}

    void  ReadPackets(void);

    void  AddClient(int local, int index, char *name);
    void  RemoveClient(int index);
    void  RemoveClients(void);

```

```
void SendSignIn(char *nickname, char *firstname, char *surname,
                int age, char *gender, char *password, char *email);
void SendKeepAlive(void);

void Connect(char *name);
void Disconnect(void);

void RunNetwork(int msec);
};
```

network.h File

This header file holds some common data structures and definitions that are used throughout the application. The importance of this file grows in the later tutorials, as more message types are created. `network.h` holds the `clientLoginData` structure that is used to store each client's data. As you have probably noticed, the `CSignIn` class has a linked list of this data type.

```
#ifndef NETWORK_H
#define NETWORK_H

#define USER_MES_SERVEREXIT    1
#define USER_MES_SIGNIN       2
#define USER_MES_KEEPLIVE     3

typedef struct clientLoginData
{
    int            index;
    char          nickname[30];
    clientLoginData *next;
} clientLoginData;

#endif
```

main.h File

This is the main header and contains only some externs (one for now).

```
#ifndef __TUTMAIN_H__
#define __TUTMAIN_H__

extern HWND hWnd_LoginDialog;

#endif
```

common.h File

The common.h file is used to include the header files all at once. This makes things simpler, as all you need to do is include one header file.

```
#ifndef __COMMON_H__
#define __COMMON_H__

#include "dreamSock.h"

#include "network.h"
#include "signin.h"
#include "main.h"

#endif
```

main.cpp File

Here is the main application source file:

```
/******
/* Programming Multiplayer Games      */
/* Tutorial game client                */
/* Programming:                        */
/* Teijo Hakala                       */
/******

#include "common.h"
#include "resource.h"

// Some global stuff
CSignin Signin;

char serverIP[32];

HINSTANCE hInst;
HWND hWnd_Application;
HWND hWnd_CreateAccountDialog;
HWND hWnd_LoginDialog;

//-----
// Name: empty()
// Desc:
//-----
LRESULT CALLBACK ApplicationProc(HWND hWnd, UINT uMsg, WPARAM wParam,
                                LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_CLOSE:
        {
            PostQuitMessage(0);
        }
    }
}
```

```

        break;
    }

    default:
        break;
}

// Pass all unhandled messages to DefWindowProc
return DefWindowProc(hWnd, uMsg, wParam, lParam);
}

//-----
// Name: empty()
// Desc:
//-----
LRESULT CALLBACK CreateAccountDialogProc(HWND hWnd, UINT uMsg, WPARAM wParam,
                                         LPARAM lParam)
{
    char nickname[30];
    char firstname[50];
    char surname[50];
    int age;
    char gender[10];
    char password[50];
    char password2[50];
    char email[150];

    int ret;

    switch (uMsg)
    {
        case WM_COMMAND:
        {
            switch (LOWORD(wParam))
            {
                case IDC_CREATEACCOUNT_CANCEL:
                    DestroyWindow(hWnd_CreateAccountDialog);
                    hWnd_CreateAccountDialog = NULL;
                    break;

                case IDC_CREATEACCOUNT_CONTINUE:

                    // -> First get the IP address of the server
                    // from the dialog
                    GetDlgItemText(hWnd_CreateAccountDialog,
                                   IDC_CREATEACCOUNT_IPADDRESS, serverIP, 20);

                    // -> Store the player data in local variables
                    GetDlgItemText(hWnd_CreateAccountDialog,
                                   IDC_CREATEACCOUNT_NICKNAME, nickname, 30);
                    GetDlgItemText(hWnd_CreateAccountDialog,
                                   IDC_CREATEACCOUNT_FIRSTNAME, firstname, 50);
                    GetDlgItemText(hWnd_CreateAccountDialog,
                                   IDC_CREATEACCOUNT_SURNAME, surname, 50);
            }
        }
    }
}

```



```

age = GetDlgItemInt(hWnd_CreateAccountDialog,
    IDC_CREATEACCOUNT_AGE, NULL, FALSE);
GetDlgItemText(hWnd_CreateAccountDialog,
    IDC_CREATEACCOUNT_GENDER, gender, 10);
GetDlgItemText(hWnd_CreateAccountDialog,
    IDC_CREATEACCOUNT_PASSWORD1, password, 50);
GetDlgItemText(hWnd_CreateAccountDialog,
    IDC_CREATEACCOUNT_PASSWORD2, password2, 50);
GetDlgItemText(hWnd_CreateAccountDialog,
    IDC_CREATEACCOUNT_EMAIL, email, 150);

// -> Check that all fields have been filled in
if(!strcmp(nickname,"") || !strcmp
    (firstname,"") || !strcmp(surname,"") ||
    !strcmp(gender,"") || !strcmp(password,"")
    || !strcmp(email,"") || age < 1)
{
    MessageBox(hWnd_CreateAccountDialog,
        "Not all fields have been filled
        in!\n\nPlease check and try
        again...", "Information Error",
        MB_OK);
    break;
}

// -> Check to see if passwords match
if(strcmp(password,password2))
{
    MessageBox(hWnd_CreateAccountDialog,
        "The two passwords you entered do
        not match!\n\nPlease check and
        try again...", "Password Error",
        MB_OK);
    break;
}

ret = Signin.GetNetworkClient()->Initialize
    ("", serverIP, 30002);

if(ret == DREAMSOCK_CLIENT_ERROR)
{
    char text[64];
    sprintf(text, "Could not open client
        socket");

    MessageBox(NULL, text, "Error", MB_OK);
}

Signin.Connect(nickname);

DestroyWindow(hWnd_CreateAccountDialog);
hWnd_CreateAccountDialog = NULL;

```

```

        Signin.SendSignIn(nickname, firstname, surname,
                          age, gender, password, email);

        break;
    default:
        break;
    }
    return 0;
}
case WM_CLOSE:
    break;

case WM_DESTROY:
    break;
}

return 0;
}

//-----
// Name: empty()
// Desc:
//-----
LRESULT CALLBACK LoginDialogProc(HWND hwnd, UINT uMsg, WPARAM wParam,
                                  LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_COMMAND:
        {
            switch(LOWORD(wParam))
            {
                case IDC_LOGIN_QUIT:
                    PostQuitMessage(0);
                    break;

                case IDC_LOGIN_CREATEACCOUNT:
                    if(!hwnd_CreateAccountDialog)
                    {
                        hwnd_CreateAccountDialog =
                            CreateDialog(hInst, MAKEINTRESOURCE
                                (IDD_CREATEACCOUNT),
                                hwnd_Application, (DLGPROC)
                                CreateAccountDialogProc);
                    }
                    break;

                case IDC_DOLOGIN:
                    break;

                default:
                    break;
            }
        }
    }
    return 0;
}

```

```

        }
        case WM_CLOSE:
        {
            PostQuitMessage(0);
            break;
        }
    }

    return 0;
}

//-----
// Name: empty()
// Desc:
//-----
int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  TCHAR *pCmdLine, int nCmdShow)
{
    int time, oldTime, newTime;

    WNDCLASSEX wcl;

    // Create our main window
    wcl.cbSize = sizeof(WNDCLASSEX);

    wcl.hInstance      = hInstance;
    wcl.lpszClassName  = "ArmyWar";
    wcl.lpfnWndProc    = ApplicationProc;
    wcl.style          = 0;

    wcl.hIcon          = LoadIcon(NULL, IDI_APPLICATION);
    wcl.hIconSm        = LoadIcon(NULL, IDI_WINLOGO);
    wcl.hCursor        = LoadCursor(NULL, IDC_ARROW);

    wcl.lpszMenuName   = NULL;
    wcl.cbClsExtra     = 0;
    wcl.cbWndExtra     = 0;

    wcl.hbrBackground = (HBRUSH) GetStockObject(LTGRAY_BRUSH);

    if(!RegisterClassEx(&wcl)) return 0;

    hWnd_Application = CreateWindow(
        "ArmyWar",
        "ARMY WAR Online 2.0",
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        640,
        480,
        HWND_DESKTOP,
        NULL,
        hInstance,
        NULL
    );
}

```

```
);

// Initialize the network library
if(dreamSock_Initialize() != 0)
{
    MessageBox(NULL, "Error initializing Communication Library!",
               "Fatal Error", MB_OK);

    return 1;
}

ShowWindow(hWnd_Application, nCmdShow);
UpdateWindow(hWnd_Application);

// Set global instance variable
hInst = hInstance;

// Display the LoginDialog
hWnd_LoginDialog = CreateDialog(hInst, MAKEINTRESOURCE(IDD_LOGINDIALOG),
                                hWnd_Application, (DLGPROC)LoginDialogProc);

MSG msg;
BOOL bMsg = FALSE;

PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE);

oldTime = dreamSock_GetCurrentSystemTime();

bool done = false;

try
{
    while(!done)
    {
        while(PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE))
        {
            if(!GetMessage(&msg, NULL, 0, 0))
            {
                Signin.Disconnect();
                done = true;
            }

            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }

        do
        {
            newTime = dreamSock_GetCurrentSystemTime();
            time = newTime - oldTime;
        } while (time < 1);

        // Run sign-in network
```

```

        Signin.RunNetwork(time);

        oldTime = newTime;
    }
}
catch(...)
{
    Signin.Disconnect();

    LogString("Unknown Exception caught in main loop");

    MessageBox(NULL, "Unknown Exception caught in main loop",
        "Error", MB_OK | MB_TASKMODAL);

    return -1;
}

return msg.wParam;
}

```

Global Variables

There are some global variables that we need to create also. Global variables should generally be avoided, but sometimes they make life simpler!

```

// Some global stuff
CSignin Signin;

char serverIP[32];

HINSTANCE hInst;
HWND hWnd_Application;
HWND hWnd_CreateAccountDialog;
HWND hWnd_LoginDialog;

```

The CSignin Signin is an object that we use for the client's network part. The char `serverIP` holds the server's IP address string once it is entered in the main login window. The other global variables are required for dialogs and such.

CreateAccountDialogProc Function

This is a dialog procedure function for creating a new account. It checks that the required information has been entered and connects the sign-in server.

```

LRESULT CALLBACK CreateAccountDialogProc(HWND hWnd, UINT uMsg, WPARAM wParam,
                                         LPARAM lParam)
{
    char nickname[30];

```

```

char firstname[50];
char surname[50];
int age;
char gender[10];
char password[50];
char password2[50];
char email[150];

int ret;

switch (uMsg)
{
    case WM_COMMAND:
    {
        switch (LOWORD(wParam))
        {
            case IDC_CREATEACCOUNT_CANCEL:
                DestroyWindow(hWnd_CreateAccountDialog);
                hWnd_CreateAccountDialog = NULL;
                break;

            case IDC_CREATEACCOUNT_CONTINUE:

                // -> First get the IP address of the server
                // from the dialog
                GetDlgItemText(hWnd_CreateAccountDialog,
                    IDC_CREATEACCOUNT_IPADDRESS, serverIP, 20);

                // -> Store the player data in local variables
                GetDlgItemText(hWnd_CreateAccountDialog,
                    IDC_CREATEACCOUNT_NICKNAME, nickname, 30);
                GetDlgItemText(hWnd_CreateAccountDialog,
                    IDC_CREATEACCOUNT_FIRSTNAME, firstname, 50);
                GetDlgItemText(hWnd_CreateAccountDialog,
                    IDC_CREATEACCOUNT_SURNAME, surname, 50);
                age = GetDlgItemInt(hWnd_CreateAccountDialog,
                    IDC_CREATEACCOUNT_AGE, NULL, FALSE);
                GetDlgItemText(hWnd_CreateAccountDialog,
                    IDC_CREATEACCOUNT_GENDER, gender, 10);
                GetDlgItemText(hWnd_CreateAccountDialog,
                    IDC_CREATEACCOUNT_PASSWORD1, password, 50);
                GetDlgItemText(hWnd_CreateAccountDialog,
                    IDC_CREATEACCOUNT_PASSWORD2, password2, 50);
                GetDlgItemText(hWnd_CreateAccountDialog,
                    IDC_CREATEACCOUNT_EMAIL, email, 150);

                // -> Check that all fields have been filled in
                if(!strcmp(nickname,"") ||
                    !strcmp(firstname,"") || !strcmp(surname,"")
                    || !strcmp(gender,"") || !strcmp(password,"")
                    || !strcmp(email,"") || age < 1)
                {
                    MessageBox(hWnd_CreateAccountDialog,
                        "Not all fields have been filled

```

```

        in!\n\nPlease check and try
        again...", "Information Error",
        MB_OK);
        break;
    }

    // -> Check to see if passwords match
    if(strcmp(password,password2))
    {
        MessageBox(hWnd_CreateAccountDialog,
            "The two passwords you entered do
            not match!\n\nPlease check and
            try again...", "Password Error",
            MB_OK);
        break;
    }

    ret = Signin.GetNetworkClient()->Initialize("",
        serverIP, 30002);

    if(ret == DREAMSOCK_CLIENT_ERROR)
    {
        char text[64];
        sprintf(text, "Could not open client
            socket");

        MessageBox(NULL, text, "Error", MB_OK);
    }

    Signin.Connect(nickname);

    DestroyWindow(hWnd_CreateAccountDialog);
    hWnd_CreateAccountDialog = NULL;

    Signin.SendSignIn(nickname, firstname, surname,
        age, gender, password, email);

        break;
    default:
        break;
    }
    return 0;
}
case WM_CLOSE:
    break;

case WM_DESTROY:
    break;
}

return 0;
}

```

Here we see how to connect the server. As you can see, we use the `CSignin` class object `Signin`. First we initialize our network client with `dreamClient`'s `Initialize` function. We want to use the default local IP and the server IP we entered in the text box. The sign-in server uses UDP port 30002. If everything went fine, we connect to the server and send our sign-in information. The functions we use here will be introduced later in this tutorial.

```
ret = Signin.GetNetworkClient()->Initialize("", serverIP, 30002);

if(ret == DREAMSOCK_CLIENT_ERROR)
{
    char text[64];
    sprintf(text, "Could not open client socket");

    MessageBox(NULL, text, "Error", MB_OK);
}

Signin.Connect(nickname);

DestroyWindow(hWnd_CreateAccountDialog);
hWnd_CreateAccountDialog = NULL;

Signin.SendSignIn(nickname, firstname, surname, age, gender, password, email);
```

WinMain Function

The second function with new code here is the `WinMain` function. As you know, this is the first function that runs in a Windows environment, so we put our initialization code here (among other things).

```
int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  TCHAR *pCmdLine, int nCmdShow)
{
    int time, oldTime, newTime;

    WNDCLASSEX wcl;

    // Create our main window
    wcl.cbSize = sizeof(WNDCLASSEX);

    wcl.hInstance      = hInstance;
    wcl.lpszClassName  = "ArmyWar";
    wcl.lpfnWndProc    = ApplicationProc;
    wcl.style          = 0;

    wcl.hIcon          = LoadIcon(NULL, IDI_APPLICATION);
    wcl.hIconSm        = LoadIcon(NULL, IDI_WINLOGO);
    wcl.hCursor        = LoadCursor(NULL, IDC_ARROW);

    wcl.lpszMenuName   = NULL;
    wcl.cbClsExtra     = 0;
```



```

wcl.cbWndExtra      = 0;

wcl.hbrBackground = (HBRUSH) GetStockObject(LTGRAY_BRUSH);

if(!RegisterClassEx(&wcl)) return 0;

hWnd_Application = CreateWindow(
    "ArmyWar",
    "ARMY WAR Online 2.0",
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    640,
    480,
    HWND_DESKTOP,
    NULL,
    hInstance,
    NULL
);

// Initialize the network library
if(dreamSock_Initialize() != 0)
{
    MessageBox(NULL, "Error initializing Communication Library!",
        "Fatal Error", MB_OK);

    return 1;
}

ShowWindow(hWnd_Application, nCmdShow);
UpdateWindow(hWnd_Application);

// Set global instance variable
hInst = hInstance;

// Display the LoginDialog
hWnd_LoginDialog = CreateDialog(hInst, MAKEINTRESOURCE(IDD_LOGIN_DIALOG),
    hWnd_Application, (DLGPROC)LoginDialogProc);

MSG msg;
BOOL bMsg = FALSE;

PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE);

oldTime = dreamSock_GetCurrentSystemTime();

bool done = false;

try
{
    while(!done)
    {
        while(PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE))
        {

```

```

        if(!GetMessage(&msg, NULL, 0, 0))
        {
            Signin.Disconnect();
            done = true;
        }

        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    do
    {
        newTime = dreamSock_GetCurrentSystemTime();
        time = newTime - oldTime;
    } while (time < 1);

    // Run sign-in network
    Signin.RunNetwork(time);

    oldTime = newTime;
}
}
catch(...)
{
    Signin.Disconnect();

    LogString("Unknown Exception caught in main loop");

    MessageBox(NULL, "Unknown Exception caught in main loop",
        "Error", MB_OK | MB_TASKMODAL);

    return -1;
}

return msg.wParam;
}

```

The main window here is created the same way it is in the next tutorial. Because it is shown there, we skip it here. Initializing dreamSock interests us more, and it is done as follows. All we do is call the `dreamSock_Initialize` function and see if it succeeded or not. If it returns non-zero, the function failed to initialize dreamSock for some reason.

```

// Initialize the network library
if(dreamSock_Initialize() != 0)
{
    MessageBox(NULL, "Error initializing Communication Library!",
        "Fatal Error", MB_OK);

    return 1;
}

```

The next thing that interests us is the main application loop. We use try and catch statements to try to catch some undesired exceptions. If it does catch something, the client tries to disconnect from the server.

The main loop calculates each frame's time in milliseconds. This is done by first getting the current time when the loop ends, and then taking the current time during the loop and subtracting it from the old loop end time. This way we see how long it takes to run the loop one time. This is called frame time and it is used as a parameter for Signin's `RunNetwork` function.

```
MSG msg;
BOOL bMsg = FALSE;

PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE);

oldTime = dreamSock_GetCurrentSystemTime();

bool done = false;

try
{
    while(!done)
    {
        while(PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE))
        {
            if(!GetMessage(&msg, NULL, 0, 0))
            {
                Signin.Disconnect();
                done = true;
            }

            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }

        do
        {
            newTime = dreamSock_GetCurrentSystemTime();
            time = newTime - oldTime;
        } while (time < 1);

        // Run sign-in network
        Signin.RunNetwork(time);

        oldTime = newTime;
    }
}
catch(...)
{
    Signin.Disconnect();
}
```

```

        LogString("Unknown Exception caught in main loop");

        MessageBox(NULL, "Unknown Exception caught in main loop", "Error",
                    MB_OK | MB_TASKMODAL);

        return -1;
    }

```

signin.cpp File — CSignin Class Methods

This file contains all the methods required for the sign-in network client to operate. We read and write data to the network in this source file.

```

/*****
/* Programming Multiplayer Games      */
/* Tutorial game client                */
/* Programming:                        */
/* Teijo Hakala                       */
*****/

#include "common.h"

//-----
// Name: empty()
// Desc:
//-----
CSignin::CSignin()
{
    networkClient    = new dreamClient;
    clientList       = NULL;
}

//-----
// Name: empty()
// Desc:
//-----
CSignin::~CSignin()
{
    delete networkClient;
}

//-----
// Name: empty()
// Desc:
//-----
void CSignin::ReadPackets(void)
{
    char data[1400];
    struct sockaddr address;

    int type;
    int ind;

```

```
int local;
int ret;
char name[30];

dreamMessage mes;
mes.Init(data, sizeof(data));

while(ret = networkClient->GetPacket(mes.data, &address))
{
    mes.SetSize(ret);
    mes.BeginReading();

    type = mes.ReadByte();

    switch(type)
    {
    case DREAMSOCK_MES_ADDCLIENT:
        local = mes.ReadByte();
        ind = mes.ReadByte();
        strcpy(name, mes.ReadString());

        AddClient(local, ind, name);
        break;

    case DREAMSOCK_MES_REMOVECLIENT:
        ind = mes.ReadByte();

        RemoveClient(ind);
        break;

    case USER_MES_SERVEREXIT:
        MessageBox(NULL, "Server disconnected", "Info", MB_OK);
        Disconnect();
        break;

    case USER_MES_SIGNIN:
        // Skip sequences
        mes.ReadShort();
        mes.ReadShort();

        ret = mes.ReadShort();

        LogString("Got lobby signin respond %d", ret);

        if(ret != SIGNIN_RESULT_ACCEPTED)
        {
            MessageBox(hWnd_LoginDialog,
                "Sign-in did not succeed. Try again.",
                "Error", MB_OK);
        }
        else
        {
            MessageBox(hWnd_LoginDialog,
                "Sign-in successful. You can now login.",
```

```

        "Info", MB_OK);
    }

    Disconnect();
    break;
}
}

//-----
// Name: empty()
// Desc:
//-----
void CSiginin::AddClient(int local, int ind, char *name)
{
    // First get a pointer to the beginning of client list
    clientLoginData *list = clientList;
    clientLoginData *prev;

    LogString("App: Client: Adding client with index %d", ind);

    // No clients yet, adding the first one
    if(clientList == NULL)
    {
        LogString("App: Client: Adding first client");

        clientList = (clientLoginData *) calloc(1,
            sizeof(clientLoginData));

        if(local)
        {
            LogString("App: Client: This one is local");
            localClient = clientList;
        }

        clientList->index = ind;
        strcpy(clientList->nickname, name);

        clientList->next = NULL;
    }
    else
    {
        LogString("App: Client: Adding another client");

        prev = list;
        list = clientList->next;

        while(list != NULL)
        {
            prev = list;
            list = list->next;
        }
    }
}

```

```

        list = (clientLoginData *) calloc(1, sizeof(clientLoginData));

        if(local)
        {
            LogString("App: Client: This one is local");
            localClient = list;
        }

        list->index = ind;
        strcpy(list->nickname, name);

        list->next = NULL;
        prev->next = list;
    }
}

//-----
// Name: empty()
// Desc:
//-----
void CSignin::RemoveClient(int ind)
{
    clientLoginData *list = clientList;
    clientLoginData *prev = NULL;
    clientLoginData *next = NULL;

    for(; list != NULL; list = list->next)
    {
        if(list->index == ind)
        {
            if(prev != NULL)
            {
                prev->next = list->next;
            }

            break;
        }

        prev = list;
    }

    if(list == clientList)
    {
        if(list)
        {
            next = list->next;
            free(list);
        }

        list = NULL;
        clientList = next;
    }
    else

```

```

        {
            if(list)
            {
                next = list->next;
                free(list);
            }

            list = next;
        }
    }

//-----
// Name: empty()
// Desc:
//-----
void CSiginin::RemoveClients(void)
{
    clientLoginData *list = clientList;
    clientLoginData *next;

    while(list != NULL)
    {
        if(list)
        {
            next = list->next;
            free(list);
        }

        list = next;
    }

    clientList = NULL;
}

//-----
// Name: empty()
// Desc:
//-----
void CSiginin::SendSignIn(char *nickname, char *firstname, char *surname, int
                           age, char *gender, char *password, char *email)
{
    char data[1400];
    dreamMessage message;
    message.Init(data, sizeof(data));

    message.WriteByte(USER_MES_SIGNIN);
    message.AddSequences(networkClient);
    message.WriteString(nickname);
    message.WriteString(firstname);
    message.WriteString(surname);
    message.WriteByte(age);
    message.WriteString(gender);
    message.WriteString(password);
}

```



```

        message.WriteString(email);

        networkClient->SendPacket(&message);
    }

//-----
// Name: empty()
// Desc:
//-----
void CSigin::SendKeepAlive(void)
{
    char data[1400];
    dreamMessage message;
    message.Init(data, sizeof(data));

    message.WriteByte(USER_MES_KEEPALIVE);
    message.AddSequences(networkClient);

    networkClient->SendPacket(&message);
}

//-----
// Name: empty()
// Desc:
//-----
void CSigin::Connect(char *name)
{
    LogString("CSigin::Connect");

    networkClient->SendConnect(name);
}

//-----
// Name: empty()
// Desc:
//-----
void CSigin::Disconnect(void)
{
    LogString("CSigin::Disconnect");

    localClient = NULL;

    RemoveClients();
    networkClient->SendDisconnect();
}

//-----
// Name: empty()
// Desc:
//-----
void CSigin::RunNetwork(int msec)
{
    if(networkClient->GetConnectionState() == DREAMSOCK_DISCONNECTED)

```

```

        return;

static int keepalive = 0;
keepalive += msec;

    if(keepalive > 20000)
    {
        SendKeepAlive();
        keepalive = 0;
    }

    ReadPackets();
}

```

CSignin Constructor

Here is the constructor for CSignin. All it does is create the dreamClient object and set `clientList` to NULL, which is very important. Otherwise, the client would not know that the list is empty.

```

CSignin::CSignin()
{
    networkClient    = new dreamClient;
    clientList       = NULL;
}

```

CSignin Destructor

This is the destructor for CSignin. The dreamClient object is deleted.

```

CSignin::~CSignin()
{
    delete networkClient;
}

```

ReadPackets Function

Finally we get to read some packets from the server! This function listens to the sign-in server and parses through system and user messages.

```

void CSignin::ReadPackets(void)
{
    char data[1400];
    struct sockaddr address;

    int type;
    int ind;
    int local;
    int ret;
    char name[30];
}

```

```

dreamMessage mes;
mes.Init(data, sizeof(data));

while(ret = networkClient->GetPacket(mes.data, &address))
{
    mes.SetSize(ret);
    mes.BeginReading();

    type = mes.ReadByte();

    switch(type)
    {
    case DREAMSOCK_MES_ADDCLIENT:
        local = mes.ReadByte();
        ind = mes.ReadByte();
        strcpy(name, mes.ReadString());

        AddClient(local, ind, name);
        break;

    case DREAMSOCK_MES_REMOVECLIENT:
        ind = mes.ReadByte();

        RemoveClient(ind);
        break;

    case USER_MES_SERVEREXIT:
        MessageBox(NULL, "Server disconnected", "Info", MB_OK);
        Disconnect();
        break;

    case USER_MES_SIGNIN:
        // Skip sequences
        mes.ReadShort();
        mes.ReadShort();

        ret = mes.ReadShort();

        LogString("Got lobby signin respond %d", ret);

        if(ret != SIGNIN_RESULT_ACCEPTED)
        {
            MessageBox(hWnd_LoginDialog,
                "Sign-in did not succeed. Try again.",
                "Error", MB_OK);
        }
        else
        {
            MessageBox(hWnd_LoginDialog,
                "Sign-in successful. You can now login.",
                "Info", MB_OK);
        }

        Disconnect();
    }
}

```

```

        break;
    }
}

```

When the server sends us any data, this is the method that reads the socket's data. If no data is coming, the function returns immediately.

Here we have two variables that are filled by dreamClient's `GetPacket` function. One is the data buffer where all the incoming data goes (we will set this up soon), and the other is a socket address structure that will hold the remote host's address after we have first received something from it. Remember that a client receives data only from the server, so this address should always be the same (and we normally have no use for it at all).

```

char data[1400];
struct sockaddr address;

```

There are some variables here that are used to store parts of the incoming data. These are not required, but they make life a bit easier.

```

int type;
int ind;
int local;
int ret;
char name[30];

```

Next we initialize a `dreamMessage` object to hold the incoming message. See how simple it is to set the data buffer for a message?

```

dreamMessage mes;
mes.Init(data, sizeof(data));

```

Now we are ready to read the packet of the socket, if there is any. When there is, we read the incoming data to our `dreamMessage` object and set the correct size for it. Then we begin reading the message, message type first.

```

while(ret = networkClient->GetPacket(mes.data, &address))
{
    mes.SetSize(ret);
    mes.BeginReading();

    type = mes.ReadByte();
    ...
}

```

Now that we know the message's type, we can parse the rest of the message. As you can see, there are `dreamSock` system messages and user messages here. We use the same "add client" and "remove client" messages as `dreamSock` to add and remove clients.

If the client receives a "server exit" message, we know the server has shut down and we should uninitialize the client. This is a user

message; we will see how to send those in the server-side part of this tutorial.

If we receive a “sign-in” message, we know that the server has processed our sign-in request and it has a response for us. So we read it from the message. Before we can read the response part of the message, we must read the message sequence part. We cannot just read the response because the message sequence numbers come first in the message array. In this case, we simply read and ignore the sequence numbers and then move on to the actual message. Then the client disconnects from the server, because there is nothing else the sign-in server has to offer to us.

```
switch(type)
{
case DREAMSOCK_MES_ADDCLIENT:
    local = mes.ReadByte();
    ind   = mes.ReadByte();
    strcpy(name, mes.ReadString());

    AddClient(local, ind, name);
    break;

case DREAMSOCK_MES_REMOVECLIENT:
    ind = mes.ReadByte();

    RemoveClient(ind);
    break;

case USER_MES_SERVEREXIT:
    MessageBox(NULL, "Server disconnected", "Info", MB_OK);
    Disconnect();
    break;

case USER_MES_SIGNIN:
    // Skip sequences
    mes.ReadShort();
    mes.ReadShort();

    ret = mes.ReadShort();

    LogString("Got lobby signin respond %d", ret);

    if(ret != SIGNIN_RESULT_ACCEPTED)
    {
        MessageBox(hWnd_LoginDialog,
            "Sign-in did not succeed. Try again.", "Error", MB_OK);
    }
    else
    {
        MessageBox(hWnd_LoginDialog,
            "Sign-in successful. You can now login.", "Info", MB_OK);
    }
}
```

```
    }  
  
    Disconnect();  
    break;  
}
```

AddClient Function

This function adds a client to the client's own client list. This list holds the data essential to the network application itself.

```
void CSiginin::AddClient(int local, int ind, char *name)  
{  
    // First get a pointer to the beginning of client list  
    clientLoginData *list = clientList;  
    clientLoginData *prev;  
  
    LogString("App: Client: Adding client with index %d", ind);  
  
    // No clients yet, adding the first one  
    if(clientList == NULL)  
    {  
        LogString("App: Client: Adding first client");  
  
        clientList = (clientLoginData *) calloc(1,  
            sizeof(clientLoginData));  
  
        if(local)  
        {  
            LogString("App: Client: This one is local");  
            localClient = clientList;  
        }  
  
        clientList->index = ind;  
        strcpy(clientList->nickname, name);  
  
        clientList->next = NULL;  
    }  
    else  
    {  
        LogString("App: Client: Adding another client");  
  
        prev = list;  
        list = clientList->next;  
  
        while(list != NULL)  
        {  
            prev = list;  
            list = list->next;  
        }  
  
        list = (clientLoginData *) calloc(1, sizeof(clientLoginData));  
  
        if(local)
```

```

        {
            LogString("App: Client: This one is local");
            localClient = list;
        }

        list->index = ind;
        strcpy(list->nickname, name);

        list->next = NULL;
        prev->next = list;
    }
}

```

This function works the same way as the `AddClient` function of `dreamServer`. The only difference is that we give more data to store as a parameter (`int local, int ind, char *name`). If the first parameter is 1, the client we are adding to the list is the local client. So in this case, set the `localClient` pointer to point to the correct client in the list. The other parameters are data to put in the client list.

RemoveClient Function

This function removes a client from the client's own client list.

```

void CSiginin::RemoveClient(int ind)
{
    clientLoginData *list = clientList;
    clientLoginData *prev = NULL;
    clientLoginData *next = NULL;

    for(; list != NULL; list = list->next)
    {
        if(list->index == ind)
        {
            if(prev != NULL)
            {
                prev->next = list->next;
            }

            break;
        }

        prev = list;
    }

    if(list == clientList)
    {
        if(list)
        {
            next = list->next;
            free(list);
        }
    }
}

```

```

        }

        list = NULL;
        clientList = next;
    }
    else
    {
        if(list)
        {
            next = list->next;
            free(list);
        }

        list = next;
    }
}

```

The function takes one parameter (`int ind`) that is used to identify the client to remove. In the following loop, we locate the correct client in the list and update the list. After that, the client can be safely removed from memory.

```

for(; list != NULL; list = list->next)
{
    if(list->index == ind)
    {
        if(prev != NULL)
        {
            prev->next = list->next;
        }

        break;
    }

    prev = list;
}

```

RemoveClients Function

This function removes all clients from the client's own client list.

```

void CSignin::RemoveClients(void)
{
    clientLoginData *list = clientList;
    clientLoginData *next;

    while(list != NULL)
    {
        if(list)
        {
            next = list->next;
            free(list);
        }
    }
}

```



```
        list = next;
    }

    clientList = NULL;
}
```

SendSignIn Function

This function sends the “sign-in” message to the server.

```
void CSigin::SendSignIn(char *nickname, char *firstname, char *surname,
                       int age, char *gender, char *password, char *email)
{
    char data[1400];
    dreamMessage message;
    message.Init(data, sizeof(data));

    message.WriteByte(USER_MES_SIGNIN);
    message.AddSequences(networkClient);
    message.WriteString(nickname);
    message.WriteString(firstname);
    message.WriteString(surname);
    message.WriteByte(age);
    message.WriteString(gender);
    message.WriteString(password);
    message.WriteString(email);

    networkClient->SendPacket(&message);
}
```

This function takes several parameters, which are the data to send to the server. The only notable thing here is that we add the sequence numbers right after the message type.

SendKeepAlive Function

This function is used to keep the client alive on the server if necessary.

```
void CSigin::SendKeepAlive(void)
{
    char data[1400];
    dreamMessage message;
    message.Init(data, sizeof(data));

    message.WriteByte(USER_MES_KEEPAKALIVE);
    message.AddSequences(networkClient);

    networkClient->SendPacket(&message);
}
```

Connect Function

This function runs `dreamClient`'s `SendConnect` function to connect to the server.

```
void CSiginin::Connect(char *name)
{
    LogString("CSiginin::Connect");

    networkClient->SendConnect(name);
}
```

Disconnect Function

This function runs `dreamClient`'s `SendDisconnect` function to disconnect from the server. We also remove all the clients from the local client list and set the local client pointer to `NULL` since they depend on the connection.

```
void CSiginin::Disconnect(void)
{
    LogString("CSiginin::Disconnect");

    localClient = NULL;

    RemoveClients();
    networkClient->SendDisconnect();
}
```

RunNetwork Function

This function is run every frame to keep the network system running.

```
void CSiginin::RunNetwork(int msec)
{
    if(networkClient->GetConnectionState() == DREAMSOCK_DISCONNECTED)
        return;

    static int keepalive = 0;
    keepalive += msec;

    if(keepalive > 20000)
    {
        SendKeepAlive();
        keepalive = 0;
    }

    ReadPackets();
}
```

The function takes one parameter (`int msec`), the frame time. We use it to calculate time. Here, we use it to send a “keep alive” message

in 20-second intervals. The function's most important task is to read the packets of the socket.

Creating a Basic Server Application

Now all we have to do is make the server application. The server works basically the same way as the client application, but with some differences of course. This server can be built and run on Windows and Unix systems. Below is the makefile for Unix.

```
#C_ARGS = -Wall
CC = gcc

all: dreamSockServer

clean:
    rm -f *.o

dreamSockServer: main.o signin.o
    $(CC) -o dreamSockServer main.o signin.o -L. -ldreamSock -lm -lstdc++

main.o: main.cpp
    $(CC) $(C_ARGS) -c main.cpp -L. -ldreamSock

signin.o: signin.cpp
    $(CC) $(C_ARGS) -c signin.cpp -L. -ldreamSock
```

We need to link `dreamSock.lib` into the server program. That is done the same way as in the client application.

We will create the following source code files: `main.cpp`, `signin.cpp`, `common.h`, `network.h`, and `signin.h`.

signin.h File

Like in the client application, `signin.h` contains the most important class of the server application. Class `CSignInServer` is used for all the network operations we need to process.

```
#ifndef __SIGNIN_H__
#define __SIGNIN_H__

#include "network.h"

#define SIGNIN_RESULT_ACCEPTED 200
#define SIGNIN_RESULT_USERNAMEBAD 201
#define SIGNIN_RESULT_PASSWORDBAD 202
#define SIGNIN_RESULT_MYSQLERROR 203

class CSignInServer
{
```

```
private:
    dreamServer      *networkServer;
    clientLoginData *clientList;

public:
    CSiginServer();
    ~CSiginServer();

    int  InitNetwork(void);
    void ShutdownNetwork(void);

    void ReadPackets(void);
    void SendExitNotification(void);

    void AddClient(void);
    void RemoveClient(struct sockaddr *address);
    void RemoveClients(void);

    void Frame(int msec);
};

#endif
```

CSiginServer Class

Here is the CSiginServer class's member methods and variables. As you can see, it contains a dreamServer object similar to how the client application contained a dreamClient object. It also contains another client list on the server side (remember that dreamServer also has one). This is used to store the application-specific data, and both lists have the same clients in the same order. We will take a closer look at the clientLoginData structure next.

```
class CSiginServer
{
private:
    dreamServer      *networkServer;
    clientLoginData *clientList;

public:
    CSiginServer();
    ~CSiginServer();

    int  InitNetwork(void);
    void ShutdownNetwork(void);

    void ReadPackets(void);
    void SendExitNotification(void);

    void AddClient(void);
    void RemoveClient(struct sockaddr *address);
```

```
        void RemoveClients(void);

        void Frame(int msec);
};
```

network.h File

This file contains some common data types and definitions, just like on the client side. There is an important structure in this file: `clientLoginData`. This structure is used to hold the clients' application-specific data. In this basic network application, there is nothing application specific, but later on in the tutorial game we will see some real data here (like the players' position on the map). Notice that there is a pointer to a `dreamClient` object here. It is set to point to a correct client when we are adding a client. After that we can use this one structure to handle the clients because we have their application-specific data here, and we can also access their network-specific functions through the `netClient` pointer.

```
#ifndef NETWORK_H
#define NETWORK_H

#include "dreamSock.h"

#define USER_MES_SERVEREXIT    1
#define USER_MES_SIGNIN      2

typedef struct clientLoginData
{
    dreamClient          *netClient;
    clientLoginData     *next;
} clientLoginData;

#endif
```

common.h File

This file is used to include common header files all at once.

```
#ifndef __COMMON_H__
#define __COMMON_H__

#include "dreamSock.h"

#include "network.h"
#include "signin.h"

#endif
```

main.cpp File

This file is the main source file for the server. As you can see, it contains a bit more code than the client version because we have code for both Unix and Windows systems here.

```
/******  
/* Programming Multiplayer Games      */  
/* Tutorial game server                */  
/* Programming:                        */  
/* Teijo Hakala                        */  
/******  
  
#ifdef Win32  
#ifndef _WINSOCKAPI_  
#define _WINSOCKAPI_  
#endif  
  
#include <windows.h>  
  
#endif  
  
#ifdef Win32  
#include <shellapi.h>  
  
#else  
#include <signal.h>  
#include <syslog.h>  
#include <errno.h>  
#include <unistd.h>  
#include <sys/time.h>  
#include <sys/stat.h>  
#include <sys/types.h>  
#include <sys/wait.h>  
#include <fcntl.h>  
#endif  
  
#include "common.h"  
  
#include <string.h>  
#include <stdlib.h>  
#include <stdio.h>  
  
// Win32 only  
#ifdef Win32  
  
// Unix only  
#else  
int runningDaemon;  
#endif  
  
CSigninServer Signin;
```

```

#ifdef Win32

//-----
// Name: empty()
// Desc:
//-----
LRESULT CALLBACK WindowProc(HWND WindowhWnd, UINT Message, WPARAM wParam,
                             LPARAM lParam)
{
    // Process messages
    switch(Message)
    {
        case WM_CREATE:
            break;

        case WM_DESTROY:
            PostQuitMessage(0);
            break;

        default:
            break;
    }

    return DefWindowProc(WindowhWnd, Message, wParam, lParam);
}

//-----
// Name: WinMain()
// Desc: Windows app start position
//-----
int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR
                    lpCmdLine, int nCmdShow)
{
    WNDCLASS WinClass;

    WinClass.style           = 0;
    WinClass.lpfnWndProc     = WindowProc;
    WinClass.cbClsExtra     = 0;
    WinClass.cbWndExtra     = 0;
    WinClass.hInstance      = hInstance;
    WinClass.hIcon          = LoadIcon(NULL, IDI_APPLICATION);
    WinClass.hCursor        = LoadCursor(NULL, IDC_ARROW);
    WinClass.hbrBackground  = (HBRUSH) (COLOR_MENU);
    WinClass.lpszMenuName   = 0;
    WinClass.lpszClassName  = "WINCLASS1";

    if(!RegisterClass(&WinClass))
        return 0;

    HWND hwnd = CreateWindow(WinClass.lpszClassName,
                            "dreamSock server",
                            WS_OVERLAPPEDWINDOW | WS_VISIBLE,
                            320,
                            240,

```

```
        320, 240,  
        NULL,  
        NULL,  
        hInstance,  
        NULL);  
  
ShowWindow(hwnd, SW_HIDE);  
  
StartLogConsole();  
  
if(Signin.InitNetwork() == 1)  
{  
    PostQuitMessage(0);  
}  
  
MSG WinMsg;  
bool done = false;  
int time, oldTime, newTime;  
  
// first peek at the message without removing it  
PeekMessage(&WinMsg, hwnd, 0, 0, PM_NOREMOVE);  
  
oldTime = dreamSock_GetCurrentSystemTime();  
  
try  
{  
    while(!done)  
    {  
        while (PeekMessage(&WinMsg, NULL, 0, 0, PM_NOREMOVE))  
        {  
            if(!GetMessage(&WinMsg, NULL, 0, 0))  
            {  
                Signin.ShutdownNetwork();  
                dreamSock_Shutdown();  
  
                done = true;  
            }  
  
            TranslateMessage(&WinMsg);  
            DispatchMessage(&WinMsg);  
        }  
  
        do  
        {  
            newTime = dreamSock_GetCurrentSystemTime();  
            time = newTime - oldTime;  
        } while (time < 1);  
  
        Signin.Frame(time);  
  
        oldTime = newTime;  
    }  
}  
catch(...)
```



```

        {
            LogString("Unknown Exception caught in main loop");

            Signin.ShutdownNetwork();
            dreamSock_Shutdown();

            MessageBox(NULL, "Unknown Exception caught in main loop",
                "Error", MB_OK | MB_TASKMODAL);

            return -1;
        }

        return WinMsg.wParam;
    }

#else

//-----
// Name: daemonInit()
// Desc: Initialize Unix daemon
//-----
static int daemonInit(void)
{
    printf("Running daemon...\n\n");

    runningDaemon = 1;

    pid_t pid;

    if((pid = fork()) < 0)
    {
        return -1;
    }
    else if(pid != 0)
    {
        exit(0);
    }

    setsid();

    umask(0);

    close(1);
    close(2);
    close(3);

    return 0;
}

//-----
// Name: keyPress()
// Desc: Check for a keypress
//-----
int keyPress(void)

```

```

    {
        static char keypressed;
        struct timeval waittime;
        int num_chars_read;
        fd_set mask;

        FD_SET(0, &mask);

        waittime.tv_sec = 0;
        waittime.tv_usec = 0;

        if(select(1, &mask, 0, 0, &waittime))
        {
            num_chars_read = read(0, &keypressed, 1);

            if(num_chars_read == 1)
                return ((int) keypressed);
        }

        return (-1);
    }

//-----
// Name: main()
// Desc: Unix app start position
//-----
int main(int argc, char **argv)
{
    LogString("Welcome to Army War Server v2.0");
    LogString("-----\n");

    if(argc > 1)
    {
        if(strcmp(argv[1], "-daemon") == 0)
        {
            daemonInit();
        }
    }

    // Ignore the SIGPIPE signal, so the program does not terminate if the
    // pipe gets broken
    signal(SIGPIPE, SIG_IGN);

    if(Signin.InitNetwork() == 1)
    {
        exit(0);
    }

    LogString("Init successful");

    int time, oldTime, newTime;

    oldTime = dreamSock_GetCurrentSystemTime();

```

```
// App main loop
try
{
    if(runningDaemon)
    {
        // Keep server alive
        while(1)
        {
            do
            {
                newTime = dreamSock_GetCurrentSystemTime();
                time = newTime - oldTime;
            } while (time < 1);

            Signin.Frame(time);

            oldTime = newTime;
        }
    }
    else
    {
        // Keep server alive (wait for keypress to kill it)
        while(keyPress() == -1)
        {
            do
            {
                newTime = dreamSock_GetCurrentSystemTime();
                time = newTime - oldTime;
            } while (time < 1);

            Signin.Frame(time);

            oldTime = newTime;
        }
    }
}
catch(...)
{
    Signin.ShutdownNetwork();
    dreamSock_Shutdown();

    LogString("Unknown Exception caught in main loop");

    return -1;
}

LogString("Shutting down everything");

Signin.ShutdownNetwork();
dreamSock_Shutdown();

return 0;
```

```

}
#endif

```

Global Variables

There are some global variables that we should take a look at first:

```

int runningDaemon;

CSigninServer Signin;

```

The integer `runningDaemon` is a Unix-only variable that tells us if the program is set to run as a daemon program (on the background). The `CSigninServer Signin` is the object that controls the server's network communications.

WindowProc Function

Let's begin with Windows functions. This first one is the window procedure function that handles the window commands. There is nothing really interesting here as we will be using our log console system as the main window.

```

LRESULT CALLBACK WindowProc(HWND WindowhWnd, UINT Message, WPARAM wParam,
                             LPARAM lParam)
{
    // Process messages
    switch(Message)
    {
        case WM_CREATE:
            break;

        case WM_DESTROY:
            PostQuitMessage(0);
            break;

        default:
            break;
    }

    return DefWindowProc(WindowhWnd, Message, wParam, lParam);
}

```

WinMain Function

This is the application entry point in Windows.

```

int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow)
{

```

```

WNDCLASS WinClass;

WinClass.style           = 0;
WinClass.lpfnWndProc    = WindowProc;
WinClass.cbClsExtra     = 0;
WinClass.cbWndExtra    = 0;
WinClass.hInstance     = hInstance;
WinClass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
WinClass.hCursor       = LoadCursor(NULL, IDC_ARROW);
WinClass.hbrBackground = (HBRUSH) (COLOR_MENU);
WinClass.lpszMenuName  = 0;
WinClass.lpszClassName = "WINCLASS1";

if(!RegisterClass(&WinClass))
    return 0;

HWND hwnd = CreateWindow(WinClass.lpszClassName,
                        "dreamSock server",
                        WS_OVERLAPPEDWINDOW | WS_VISIBLE,
                        320,
                        240,
                        320, 240,
                        NULL,
                        NULL,
                        hInstance,
                        NULL);

ShowWindow(hwnd, SW_HIDE);

StartLogConsole();

if(Signin.InitNetwork() == 1)
{
    PostQuitMessage(0);
}

MSG WinMsg;
bool done = false;
int time, oldTime, newTime;

// first peek at the message without removing it
PeekMessage(&WinMsg, hwnd, 0, 0, PM_NOREMOVE);

oldTime = dreamSock_GetCurrentSystemTime();

try
{
    while(!done)
    {
        while (PeekMessage(&WinMsg, NULL, 0, 0, PM_NOREMOVE))
        {
            if(!GetMessage(&WinMsg, NULL, 0, 0))
            {

```

```

        Signin.ShutdownNetwork();
        dreamSock_Shutdown();

        done = true;
    }

    TranslateMessage(&WinMsg);
    DispatchMessage(&WinMsg);
}

do
{
    newTime = dreamSock_GetCurrentSystemTime();
    time = newTime - oldTime;
} while (time < 1);

Signin.Frame(time);

oldTime = newTime;
}
}
catch(...)
{
    LogString("Unknown Exception caught in main loop");

    Signin.ShutdownNetwork();
    dreamSock_Shutdown();

    MessageBox(NULL, "Unknown Exception caught in main loop",
        "Error", MB_OK | MB_TASKMODAL);

    return -1;
}

return WinMsg.wParam;
}

```

We want to see what is happening on the server, so instead of a normal window we are going to open a console window. First though, we need to open a real window and hide it. It runs the window procedure for us and keeps the program alive. Then we simply start up our log console (implemented into dreamSock).

```

HWND hwnd = CreateWindow(WinClass.lpszClassName,
    "dreamSock server",
    WS_OVERLAPPEDWINDOW | WS_VISIBLE,
    320,
    240,
    320, 240,
    NULL,
    NULL,
    hInstance,
    NULL);

```

```
ShowWindow(hwnd, SW_HIDE);
```

```
StartLogConsole();
```

Then we can initialize the sign-in server using the `CSigninServer` method `InitNetwork`. If something goes wrong here, we just bail out from the program by posting a quit message to the main loop.

```
if(Signin.InitNetwork() == 1)
{
    PostQuitMessage(0);
}
```

The main loop works exactly the same as in the client application. We try to catch exceptions here too, and that is a very useful thing to do — it can keep the server running in cases when it normally would crash. Just like in the client application, we calculate the frame time and use it in the `RunNetwork` method of `CSigninServer`.

```
MSG WinMsg;
bool done = false;
int time, oldTime, newTime;

// first peek at the message without removing it
PeekMessage(&WinMsg, hwnd, 0, 0, PM_NOREMOVE);

oldTime = dreamSock_GetCurrentSystemTime();

try
{
    while(!done)
    {
        while (PeekMessage(&WinMsg, NULL, 0, 0, PM_NOREMOVE))
        {
            if(!GetMessage(&WinMsg, NULL, 0, 0))
            {
                Signin.ShutdownNetwork();
                dreamSock_Shutdown();

                done = true;
            }

            TranslateMessage(&WinMsg);
            DispatchMessage(&WinMsg);
        }

        do
        {
            newTime = dreamSock_GetCurrentSystemTime();
            time = newTime - oldTime;
        } while (time < 1);

        Signin.Frame(time);
    }
}
```

```
        oldTime = newTime;
    }
}
catch(...)
{
    LogString("Unknown Exception caught in main loop");

    Signin.ShutdownNetwork();
    dreamSock_Shutdown();

    MessageBox(NULL, "Unknown Exception caught in main loop", "Error",
               MB_OK | MB_TASKMODAL);

    return -1;
}
```

daemonInit Function

This function makes the program run as a daemon on Unix systems.

```
static int daemonInit(void)
{
    printf("Running daemon...\n\n");

    runningDaemon = 1;

    pid_t pid;

    if((pid = fork()) < 0)
    {
        return -1;
    }
    else if(pid != 0)
    {
        exit(0);
    }

    setsid();

    umask(0);

    close(1);
    close(2);
    close(3);

    return 0;
}
```

First we create a copy of our process with the `fork` function. Then we can close the parent process with the `exit` function.

```
pid_t pid;

if((pid = fork()) < 0)
```



```
{
    return -1;
}
else if(pid != 0)
{
    exit(0);
}
```

Next we set a new process group for the process, detaching it from the terminal we used:

```
setsid();
```

Then we set the file creation mask to 0, so we can create files and so on:

```
umask(0);
```

Finally, we close the standard I/O descriptors (stdin, stdout, and stderr) because they are not required.

```
close(1);
close(2);
close(3);
```

keyPress Function

This function tells us if a key has been pressed down. It is used on non-daemon programs only.

```
int keyPress(void)
{
    static char keypressed;
    struct timeval waittime;
    int num_chars_read;
    fd_set mask;

    FD_SET(0, &mask);

    waittime.tv_sec = 0;
    waittime.tv_usec = 0;

    if(select(1, &mask, 0, 0, &waittime))
    {
        num_chars_read = read(0, &keypressed, 1);

        if(num_chars_read == 1)
            return ((int) keypressed);
    }

    return (-1);
}
```

main Function

The last function here is the application entry point for Unix systems.

```
int main(int argc, char **argv)
{
    LogString("Welcome to Army War Server v2.0");
    LogString("-----\n");

    if(argc > 1)
    {
        if(strcmp(argv[1], "-daemon") == 0)
        {
            daemonInit();
        }
    }

    // Ignore the SIGPIPE signal, so the program does not terminate if the
    // pipe gets broken
    signal(SIGPIPE, SIG_IGN);

    if(Signin.InitNetwork() == 1)
    {
        exit(0);
    }

    LogString("Init successful");

    int time, oldTime, newTime;

    oldTime = dreamSock_GetCurrentSystemTime();

    // App main loop
    try
    {
        if(runningDaemon)
        {
            // Keep server alive
            while(1)
            {
                do
                {
                    newTime = dreamSock_GetCurrentSystemTime();
                    time = newTime - oldTime;
                } while (time < 1);

                Signin.Frame(time);

                oldTime = newTime;
            }
        }
    }
}
```

```

        else
        {
            // Keep server alive (wait for keypress to kill it)
            while(keyPress() == -1)
            {
                do
                {
                    newTime = dreamSock_GetCurrentSystemTime();
                    time = newTime - oldTime;
                } while (time < 1);

                Signin.Frame(time);

                oldTime = newTime;
            }
        }
    }
    catch(...)
    {
        Signin.ShutdownNetwork();
        dreamSock_Shutdown();

        LogString("Unknown Exception caught in main loop");

        return -1;
    }

    LogString("Shutting down everything");

    Signin.ShutdownNetwork();
    dreamSock_Shutdown();

    return 0;
}

```

If we give a “-daemon” parameter for the program when we start it, the daemon is initialized:

```

if(argc > 1)
{
    if(strcmp(argv[1], "-daemon") == 0)
    {
        daemonInit();
    }
}

```

There are two version of the main loop. One is for the daemon and the other is for the foreground application. If the program is not running as a daemon, we can terminate the program by pressing any key. Otherwise, the main loop works the same way as in Windows.

```

// Keep server alive (wait for keypress to kill it)
while(keyPress() == -1)
{
    do

```

```

    {
        newTime = dreamSock_GetCurrentSystemTime();
        time = newTime - oldTime;
    } while (time < 1);

    Signin.Frame(time);

    oldTime = newTime;
}

```

signin.cpp File — CSiginServer Class Methods

This is the signin.cpp file that contains the methods for CSiginServer.

```

/*****
/* Programming Multiplayer Games      */
/* Tutorial game server                */
/* Programming:                        */
/* Teijo Hakala                        */
*****/

#include "common.h"
#include <malloc.h>

//-----
// Name: empty()
// Desc:
//-----
CSiginServer::CSiginServer()
{
    networkServer = new dreamServer;
    clientList    = NULL;
}

//-----
// Name: empty()
// Desc:
//-----
CSiginServer::~CSiginServer()
{
    delete networkServer;
}

//-----
// Name: InitNetwork()
// Desc: Initialize network
//-----
int CSiginServer::InitNetwork(void)
{
    // Initialize dreamSock and the server
    if(dreamSock_Initialize() != 0)
    {
        LogString("Error initializing Communication Library!");
    }
}

```

```

        return 1;
    }

    int ret = networkServer->Initialize("", 30002);

    if(ret == DREAMSOCK_SERVER_ERROR)
    {
#ifdef Win32
        char text[64];
        sprintf(text, "Could not open server on port %d",
            networkServer->GetPort());

        MessageBox(NULL, text, "Error", MB_OK);
#else
        LogString("Could not open server on port %d",
            networkServer->GetPort());
#endif
        return 1;
    }

    return 0;
}

//-----
// Name: ShutdownNetwork()
// Desc: Shut down network
//-----
void CSiginServer::ShutdownNetwork(void)
{
    LogString("Shutting down sign-in server...");

    SendExitNotification();
    RemoveClients();
}

//-----
// Name: empty()
// Desc:
//-----
void CSiginServer::ReadPackets(void)
{
    char data[1400];

    int type;
    int ret;

    // Some incoming data
    char password[50];
    int respond;

    char nickname[30];
    char surname[30];
    char firstname[30];
    char gender[10];

```

```
char email[30];
int age;

struct sockaddr address;

clientLoginData *cList;

dreamMessage mes;
mes.Init(data, sizeof(data));

// Get the packet from the socket
try
{
    while(ret = networkServer->GetPacket(mes.data, &address))
    {
        mes.SetSize(ret);
        mes.BeginReading();

        type = mes.ReadByte();

        // Check the type of the message
        switch(type)
        {
            case DREAMSOCK_MES_CONNECT:
                AddClient();
                break;

            case DREAMSOCK_MES_DISCONNECT:
                RemoveClient(&address);
                break;

            case USER_MES_SIGNIN:
                // Skip sequences
                mes.ReadShort();
                mes.ReadShort();

                strcpy(nickname, mes.ReadString());
                strcpy(firstname, mes.ReadString());
                strcpy(surname, mes.ReadString());
                age = mes.ReadByte();
                strcpy(gender, mes.ReadString());
                strcpy(password, mes.ReadString());
                strcpy(email, mes.ReadString());

                LogString("Signin: Player %s signed in", nickname);

                // MySQL connection comes here

                respond = SIGNIN_RESULT_ACCEPTED;

                // Find the correct client by comparing addresses
                cList = clientList;

                for(; cList != NULL; cList = cList->next)
```

```

        {
            if(memcmp(cList->netClient->
                GetSocketAddress(), &address,
                sizeof(address)) == 0)
            {
                cList->netClient->message.Init
                    (cList->netClient->
                    message.outgoingData,
                    sizeof(cList->netClient->
                    message.outgoingData));

                cList->netClient->message.WriteByte
                    (USER_MES_SIGNIN); // type
                cList->netClient->message.AddSequences
                    (cList->netClient); // sequences
                cList->netClient->message.WriteShort
                    (respond); // respond
                cList->netClient->SendPacket();

                LogString("Sending signin respond");

                break;
            }
        }
        break;
    }
}
}
catch(...)
{
    LogString("Unknown Exception caught in Signin ReadPackets loop");
}

#ifdef Win32
    MessageBox(NULL, "Unknown Exception caught in Signin ReadPackets
        loop", "Error", MB_OK | MB_TASKMODAL);
#endif
}

//-----
// Name: empty()
// Desc:
//-----
void CSiginServer::SendExitNotification(void)
{
    clientLoginData *toClient = clientList;

    for(; toClient != NULL; toClient = toClient->next)
    {
        toClient->netClient->message.Init(toClient->netClient->
            message.outgoingData, sizeof(toClient->netClient->
            message.outgoingData));
    }
}

```

```

        toClient->netClient->message.WriteByte(USER_MES_SERVEREXIT);
        // type
        toClient->netClient->message.AddSequences(toClient->netClient);
        // sequences
    }

    networkServer->SendPackets();
}

//-----
// Name: empty()
// Desc:
//-----
void CSignInServer::AddClient(void)
{
    // First get a pointer to the beginning of client list
    clientLoginData *list = clientList;
    clientLoginData *prev;
    dreamClient *netList = networkServer->GetClientList();

    // No clients yet, adding the first one
    if(clientList == NULL)
    {
        LogString("App: Server: Adding first client");

        clientList = (clientLoginData *) calloc(1,
            sizeof(clientLoginData));

        clientList->netClient = netList;

        clientList->next = NULL;
    }
    else
    {
        LogString("App: Server: Adding another client");

        prev = list;
        list = clientList->next;
        netList = netList->next;

        while(list != NULL)
        {
            prev = list;
            list = list->next;
            netList = netList->next;
        }

        list = (clientLoginData *) calloc(1, sizeof(clientLoginData));

        list->netClient = netList;

        list->next = NULL;

        prev->next = list;
    }
}

```



```

    }
}

//-----
// Name: empty()
// Desc:
//-----
void CSiginServer::RemoveClient(struct sockaddr *address)
{
    clientLoginData *list = clientList;
    clientLoginData *prev = NULL;
    clientLoginData *next = NULL;

    for(; list != NULL; list = list->next)
    {
        if(memcmp(list->netClient->GetSocketAddress(), address,
            sizeof(address)) == 0)
        {
            if(prev != NULL)
            {
                prev->next = list->next;
            }

            break;
        }

        prev = list;
    }

    if(list == clientList)
    {
        if(list)
        {
            next = list->next;
            free(list);
        }

        list = NULL;
        clientList = next;
    }
    else
    {
        if(list)
        {
            next = list->next;
            free(list);
        }

        list = next;
    }
}

```

```

//-----
// Name: empty()
// Desc:
//-----
void CSiginServer::RemoveClients(void)
{
    clientLoginData *list = clientList;
    clientLoginData *next;

    while(list != NULL)
    {
        if(list)
        {
            next = list->next;
            free(list);
        }

        list = next;
    }

    clientList = NULL;
}

//-----
// Name: empty()
// Desc:
//-----
void CSiginServer::RunNetwork(int msec)
{
    ReadPackets();
}

```

CSiginServer Constructor

Here is the constructor for CSiginServer. The dreamServer object is created and `clientList` is set to `NULL`.

```

CSiginServer::CSiginServer()
{
    networkServer = new dreamServer;
    clientList = NULL;
}

```

CSiginServer Destructor

Here is the destructor for CSiginServer. The dreamServer object is deleted.

```

CSiginServer::~CSiginServer()
{
    delete networkServer;
}

```

InitNetwork Function

This function initializes the server's socket so we can start serving our clients. The sign-in server uses UDP port 30002.

```
int CSigninServer::InitNetwork(void)
{
    // Initialize dreamSock and the server
    if(dreamSock_Initialize() != 0)
    {
        LogString("Error initializing Communication Library!");
        return 1;
    }

    int ret = networkServer->Initialize("", 30002);

    if(ret == DREAMSOCK_SERVER_ERROR)
    {
#ifdef Win32
        char text[64];
        sprintf(text, "Could not open server on port %d",
            networkServer->GetPort());

        MessageBox(NULL, text, "Error", MB_OK);
#else
        LogString("Could not open server on port %d",
            networkServer->GetPort());
#endif
        return 1;
    }

    return 0;
}
```

ShutdownNetwork Function

This function uninitializes the server and tells everyone we are going down.

```
void CSigninServer::ShutdownNetwork(void)
{
    LogString("Shutting down sign-in server...");

    SendExitNotification();
    RemoveClients();
}
```

ReadPackets Function

This function reads the packets from the clients. It functions the same way as it does on the client side. Messages are read into a dreamMessage, and they are parsed based on the message type.

```
void CSigninServer::ReadPackets(void)
{
    char data[1400];

    int type;
    int ret;

    // Some incoming data
    char password[50];
    int respond;

    char nickname[30];
    char surname[30];
    char firstname[30];
    char gender[10];
    char email[30];
    int age;

    struct sockaddr address;

    clientLoginData *cList;

    dreamMessage mes;
    mes.Init(data, sizeof(data));

    // Get the packet from the socket
    try
    {
        while(ret = networkServer->GetPacket(mes.data, &address))
        {
            mes.SetSize(ret);
            mes.BeginReading();

            type = mes.ReadByte();

            // Check the type of the message
            switch(type)
            {
                case DREAMSOCK_MES_CONNECT:
                    AddClient();
                    break;

                case DREAMSOCK_MES_DISCONNECT:
                    RemoveClient(&address);
                    break;

                case USER_MES_SIGNIN:
                    // Skip sequences
                    mes.ReadShort();
                    mes.ReadShort();

                    strcpy(nickname, mes.ReadString());
                    strcpy(firstname, mes.ReadString());
                    strcpy(surname, mes.ReadString());
```


Here we process the sign-in message that comes from a client. We do not use MySQL in this tutorial, but you can see where the MySQL connection would be placed. Obviously when we have the information to put into the database, we would put it there.

```

case USER_MES_SIGNIN:
    // Skip sequences
    mes.ReadShort();
    mes.ReadShort();

    strcpy(nickname, mes.ReadString());
    strcpy(firstname, mes.ReadString());
    strcpy(surname, mes.ReadString());
    age = mes.ReadByte();
    strcpy(gender, mes.ReadString());
    strcpy(password, mes.ReadString());
    strcpy(email, mes.ReadString());

    LogString("Signin: Player %s signed in", nickname);

    // MySQL connection comes here

    respond = SIGNIN_RESULT_ACCEPTED;
...

```

We need to find out which host sent us this message. We compare the addresses of the old clients (in the client list) to the address the message came from. When we find the correct one, we send the sign-in response.

```

// Find the correct client by comparing addresses
cList = clientList;

for(; cList != NULL; cList = cList->next)
{
    if(memcmp(cList->netClient->GetSocketAddress(), &address,
        sizeof(address)) == 0)
    {
        cList->netClient->message.Init(cList->netClient->
            message.outgoingData, sizeof(cList->netClient->
            message.outgoingData));

        cList->netClient->message.WriteByte(USER_MES_SIGNIN);
        // type
        cList->netClient->message.AddSequences(cList->netClient);
        // sequences
        cList->netClient->message.WriteShort(respond);
        // respond
        cList->netClient->SendPacket();

        LogString("Sending signin respond");
    }
}

```

```

        break;
    }
}

```

SendExitNotification Function

This function sends a packet to each client to tell them that the server is going down. We use the `dreamClient`'s internal message, so we can use `dreamServer`'s `SendPackets` function to send the packets all at once.

```

void CSiginServer::SendExitNotification(void)
{
    clientLoginData *toClient = clientList;

    for(; toClient != NULL; toClient = toClient->next)
    {
        toClient->netClient->message.Init(toClient->netClient->
            message.outgoingData, sizeof(toClient->netClient->
            message.outgoingData));

        toClient->netClient->message.WriteByte(USER_MES_SERVEREXIT);
        // type
        toClient->netClient->message.AddSequences(toClient->netClient);
        // sequences
    }

    networkServer->SendPackets();
}

```

AddClient Function

The `AddClient` function adds a client to the server application's own client list (filling the application-specific data). Notice how we set the `netClient` pointer to point to the matching client in `dreamSock`'s client list.

```

void CSiginServer::AddClient(void)
{
    // First get a pointer to the beginning of client list
    clientLoginData *list = clientList;
    clientLoginData *prev;
    dreamClient *netList = networkServer->GetClientList();

    // No clients yet, adding the first one
    if(clientList == NULL)
    {
        LogString("App: Server: Adding first client");

        clientList = (clientLoginData *) calloc(1,
            sizeof(clientLoginData));
    }
}

```

```

        clientList->netClient = netList;

        clientList->next = NULL;
    }
    else
    {
        LogString("App: Server: Adding another client");

        prev = list;
        list = clientList->next;
        netList = netList->next;

        while(list != NULL)
        {
            prev = list;
            list = list->next;
            netList = netList->next;
        }

        list = (clientLoginData *) calloc(1, sizeof(clientLoginData));

        list->netClient = netList;

        list->next = NULL;

        prev->next = list;
    }
}

```

RemoveClient Function

This function removes a client from the server application's client list.

```

void CSiginServer::RemoveClient(struct sockaddr *address)
{
    clientLoginData *list = clientList;
    clientLoginData *prev = NULL;
    clientLoginData *next = NULL;

    for(; list != NULL; list = list->next)
    {
        if(memcmp(list->netClient->GetSocketAddress(), address,
            sizeof(address)) == 0)
        {
            if(prev != NULL)
            {
                prev->next = list->next;
            }

            break;
        }
    }
}

```



```

        prev = list;
    }

    if(list == clientList)
    {
        if(list)
        {
            next = list->next;
            free(list);
        }

        list = NULL;
        clientList = next;
    }
    else
    {
        if(list)
        {
            next = list->next;
            free(list);
        }

        list = next;
    }
}

```

This function takes one parameter (`struct sockaddr *address`), which is the socket address of the client to remove. The address is looked up from the client list, and the matching client is removed.

RemoveClients Function

This function removes all the clients from the server application's client list.

```

void CSiginServer::RemoveClients(void)
{
    clientLoginData *list = clientList;
    clientLoginData *next;

    while(list != NULL)
    {
        if(list)
        {
            next = list->next;
            free(list);
        }

        list = next;
    }

    clientList = NULL;
}

```

RunNetwork Function

This function is run every frame. Its purpose is to keep the network running.

```
void CSiginServer::RunNetwork(int msec)
{
    ReadPackets();
}
```

Summary

In this tutorial we learned how to create a basic network application with dreamSock. We learned how to create both a client application and a server application. Together they make a complete network application. We now know how to send and receive messages, so we can move on to more complicated things in our game tutorial. But before we do, we create the game lobby in the next tutorial.

This page intentionally left blank.

Tutorial 4

Creating the Game Lobby

Introduction

In this tutorial we learn how to create the lobby for our game, where the players can chat and create games to play. We will see how to use the network library to make the lobby work, but we will also learn how to create the lobby dialog itself. The lobby dialog is the screen where you see all the players, their chat messages, and the existing games. After this tutorial we can actually create and join the games in the lobby, but we cannot start the game yet. Starting and making the game work is explained in Tutorial 5, “Creating Your Online Game.”

Creating the Lobby Client Application

We will first create the client application for the lobby, which is an updated version of the previous tutorial. This time when you log in, the lobby dialog is opened.

Creating the Dialogs

It is best to start off with creating the dialogs that we will use in the lobby. When we have the dialogs ready, we know what buttons there are, what dialog opens when you press the buttons, and so on.

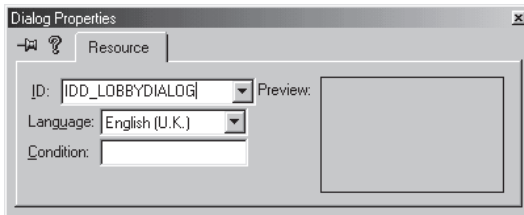
Lobby Dialog

The lobby dialog is the main screen of the lobby. There you see all the players who have logged in, their chat messages, and the existing games. You can also see the buttons for creating a new game and joining an existing one. This dialog should also have a button for logging out from the server.

We create the lobby dialog exactly the same way we created the earlier dialogs for login and signup. As we recall from the earlier tutorial, the dialogs are created from the Insert menu in Visual Studio. Select Resource..., Dialog to choose what to create, and then click the New button to create a new dialog. Now you see a new dialog (like the one when you created the login dialogs) in front of you, but it does not look at all like we want it to. So let's modify it.

Before we do anything else, we rename the dialog to `IDD_LOBBYDIALOG`. Click the right mouse button on the name of the new dialog in the list on the left side. Click on Properties and type the new name in. Now you can close the properties window.

Figure 1

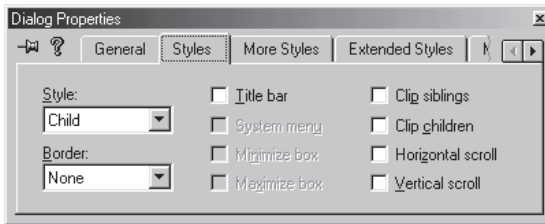


Once the name is set up properly, we can change the style of the dialog to what we want. Right-click anywhere and click on Properties in the pop-up menu (or double-click on the empty space in the dialog preview window) to open the Dialog Properties window.

The General tab does not have anything interesting for us in this case, so move on to the next tab — the Styles tab. In the Styles tab you see many check boxes and two combo boxes. Uncheck all the check boxes, choose Child as the dialog style from the upper combo box, and choose None for the border in the lower combo box. Now our dialog is a child window and has no border of its own. It also does not have a title bar showing its name. We do all of this because this dialog fits into the

game main window, and therefore the lobby dialog actually is not a window of its own.

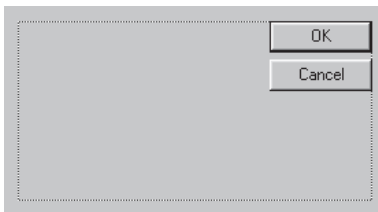
Figure 2



The other tabs in the Dialog Properties window do not interest us, so it can be closed now.

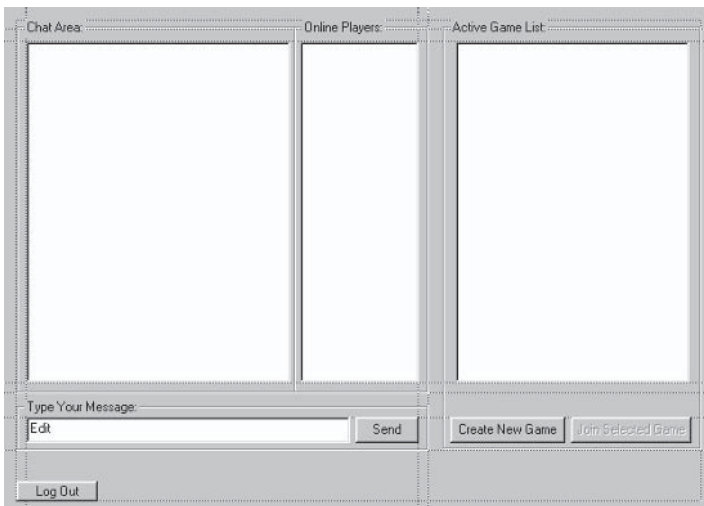
Next, remove the OK and Cancel buttons that were added automatically to the dialog when it was created.

Figure 3



Now we have an empty dialog that is ready for adding new buttons, edit boxes, and lists. Figure 4 shows what the complete lobby dialog should look like.

Figure 4



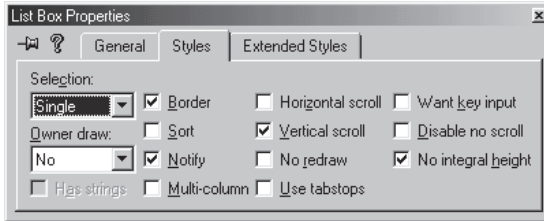
Now that the dialog is all laid out, we need to name the controls so we can use them in the code as well. They do already have names, but those are the default names based on the control type, so to make

everything as clear as possible, we name the controls to correspond to their usage. Here are the three list control names starting from left to right:

```
IDC_CHATLIST
IDC_PLAYERLIST
IDC_GAMELIST
```

We do not wish to sort any of the list items in any list in the lobby, so we disable the sorting option from each one. Double-click on the list (or right-click and choose Properties from the pop-up menu) to open the List Box Properties window. Uncheck the Sort check box on the Styles tab and close the properties window. Repeat this for all the list boxes.

Figure 5



The chat edit box and the Send button are named as follows:

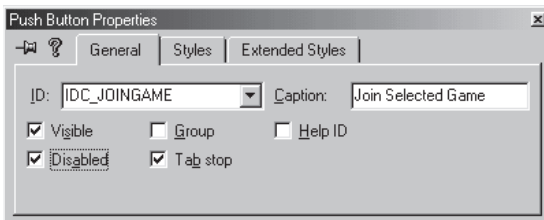
```
IDC_CHATMESSAGE
IDC_SENDCHATMESSAGE
```

The Create New Game, Join Selected Game, and Log Out buttons are named as follows:

```
IDC_CREATEGAME
IDC_JOININGAME
IDC_LOGOUT
```

Note that the Join Selected Game button is disabled by default because you can join a game only when you have selected one and when the selected game is not in progress. Disabling the button by default is done from the Push Button Properties window, which is opened by double-clicking on the button (or right-clicking on it and choosing Properties from the pop-up menu). Check the Disabled check box and close the properties window.

Figure 6

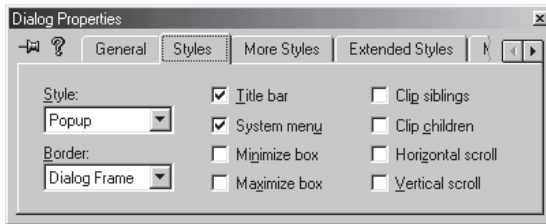


The lobby dialog is complete. All we need to do now is create the other dialogs that open when you press the buttons in the lobby dialog.

Create Game Dialog

We create the Create Game dialog the same way as the lobby dialog. Let's start by naming the dialog `IDD_CREATEGAME` in the Dialog Properties window. Next, open another Dialog Properties window by double-clicking on the empty space in the preview window.

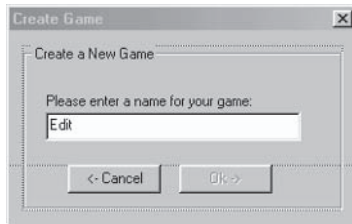
Figure 7



On the Styles tab, set the dialog style to `Popup` and the border to `Dialog Frame`. Make sure the `Title bar` check box is checked, and close the properties window.

Lay out the dialog as shown in the following figure.

Figure 8



Then name the controls as shown in the following list, putting the edit box name first, then the Cancel and OK buttons.

```
IDC_GAMENAME
IDC_CANCELCREATEGAME
IDC_DOCREATEGAME
```

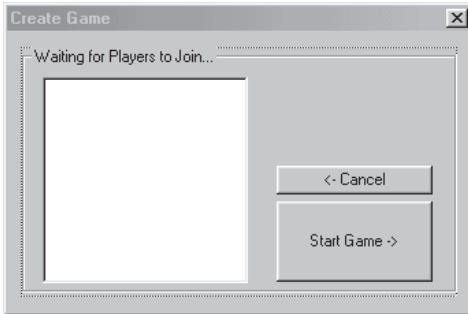
When the user presses the OK button, a dialog will open that shows the players who have joined that game and gives the game host the ability to start or cancel the game.

Create View Players Dialog

Again, we create another dialog as we have done before, this time naming it `IDD_CREATEVIEWPLAYERS`. This dialog has the same properties as the Create Game dialog, so we will not go through them here.

Lay out the dialog as shown in the following figure.

Figure 9



Then name the controls as listed here. The first name in the list is for the player list box, the second for the Cancel button, and the last one for the Start Game button.

```
IDC_PLAYERSINGAME
IDC_CANCELGAME
IDC_STARTGAME
```

The Start Game button starts the actual game, but in this tutorial we stop at this dialog. Pressing the Start Game button at this point does nothing.

Join Game Dialog

Once more we create a dialog as we have created all the other dialogs. This is the very last dialog to create for this project. Name it `IDD_JOINVIEWPLAYERS`, as it is used only to show the players who have joined the game. We can also cancel the join if we do not wish to be part of the game after all.

Lay out the dialog as shown here.



Figure 10

Then name the two controls as follows:

```
IDC_JOINPLAYERSINGAME
IDC_JOINCANCEL
```

Now all the dialogs are done, and we can move on to program the lobby system.

Lobby System Code

The lobby's network part works the same way as the sign-in system's network part, so we will not focus on that here. If you have not read Tutorial 3 yet, you should do so now. This tutorial shares a lot of similar code with that one, but here we focus more on how to make the lobby work and build it on top of our network system.

Lobby Client Code

Let's begin with the client-side application. First we will take a look at the source code files: lobby.cpp, main.cpp, lobby.h, main.h, and network.h.

lobby.h File

This is the file for the lobby's CLobby class. It is similar to the CSignIn class in the previous tutorial.

```
#ifndef __LOBBY_H__
#define __LOBBY_H__

#include "network.h"
#include "main.h"

#define LOBBYLOGIN_RESULT_ACCEPTED 200
#define LOBBYLOGIN_RESULT_USERNAMEBAD 201
#define LOBBYLOGIN_RESULT_PASSWORDBAD 202
#define LOBBYLOGIN_RESULT_MYSQLERROR 203

class CLobby
{
private:
    dreamClient *networkClient;
    clientLoginData *clientList;
    clientLoginData *localClient; // Pointer to the local client in
                                // the client list
};
```

```

        int                gameAmount;
        int                timeConnecting;    // How long we have tried
                                                // to connect

public:
    CLobby();
    ~CLobby();

    dreamClient            *GetNetworkClient(void) {return networkClient;}
    clientLoginData        *GetLocalClient(void)  {return localClient;}

    void                  RefreshPlayerList(void);
    void                  RefreshGameList(void);
    void                  RefreshJoinedPlayersList(void);

    void                  ReadPackets(void);

    void                  AddClient(int local, int index, char *name);
    void                  RemoveClient(int index);
    void                  RemoveClients(void);

    void                  AddGame(char *name, int index, bool inProgress);
    void                  RemoveGame(char *name);
    void                  RemoveGames(void);

    void                  RequestGameData(void);
    void                  SendChat(char *text);
    void                  SendCreateGame(char *gamename);
    void                  SendRemoveGame(char *gamename);
    void                  SendStartGame(int ind);
    void                  SendKeepAlive(void);

    void                  Connect(char *name, char *password);
    void                  Disconnect(void);

    void                  RunNetwork(int msec);

    int                   GetGameAmount(void)    {return gameAmount;}
};

extern CLobby Lobby;

#endif

```

There are a couple of member variables that we should take a look at:

```

int                gameAmount;
int                timeConnecting;

```

The integer `gameAmount` tells us how many games exist at the moment. The integer `timeConnecting` tells us how long we have tried to connect to the lobby server (in milliseconds).

network.h File

This file is almost the same as network.h in the sign-in tutorial, but now we have more message types.

```
#ifndef NETWORK_H
#define NETWORK_H

#define USER_MES_SERVEREXIT    1
#define USER_MES_LOGIN        2
#define USER_MES_SIGNIN      3
#define USER_MES_CHAT        4
#define USER_MES_CREATEGAME  5
#define USER_MES_REMOVEGAME  6
#define USER_MES_GAMEDATA    7
#define USER_MES_STARTGAME   8
#define USER_MES_MAPDATA     9
#define USER_MES_KEEPLIVE   10

typedef struct clientLoginData
{
    int            index;
    char          nickname[30];
    clientLoginData *next;
} clientLoginData;

#endif
```

main.h File

This is the main application header file, which sounds more important than it is. We have only some externs and function prototypes here.

```
#ifndef __TUTMAIN_H__
#define __TUTMAIN_H__

LRESULT CALLBACK ApplicationProc(HWND hwnd, UINT uMsg, WPARAM wParam,
                                LPARAM lParam);

extern char serverIP[32];

extern HINSTANCE hInst;
extern HWND hwnd_Application;
extern HWND hwnd_CreateAccountDialog;
extern HWND hwnd_LoginDialog;
extern HWND hwnd_LobbyDialog;

extern HWND hwnd_CreateGameDialog;
extern HWND hwnd_JoinGameDialog;

extern HWND hwnd_CreateViewPlayersDialog;

#endif
```

main.cpp File

This time we will not look at the entire file as it is basically the same as before. Instead, we will look at the new functions directly.

CreateViewPlayersDialogProc Function

This function handles the dialog for viewing players who join the created game. The dialog has Start Game and Cancel buttons, and they are processed here.

```
LRESULT CALLBACK CreateViewPlayersDialogProc(HWND hWnd, UINT uMsg, WPARAM
                                             wParam, LPARAM lParam)
{
    switch (uMsg)
    {
    case WM_COMMAND:
        switch(LOWORD(wParam))
        {
            case IDC_STARTGAME:
                DestroyWindow(hWnd_CreateViewPlayersDialog);
                break;

            case IDC_CANCELGAME:
                DestroyWindow(hWnd_CreateViewPlayersDialog);
                break;

            default:
                break;
        }

        break;
    }

    return 0;
}
```

CreateGameDialogProc Function

This dialog procedure takes in the name for a new game and sends a create game message to the server if the player wants to create a game. It also makes sure that you cannot press the Create Game button unless you have entered a name for the game.

```
LRESULT CALLBACK CreateGameDialogProc(HWND hWnd, UINT uMsg, WPARAM wParam,
                                       LPARAM lParam)
{
    char gamename[32];

    switch (uMsg)
    {
```

```
case WM_COMMAND:
    switch(LOWORD(wParam))
    {
    case IDC_DOCREATEGAME:
        GetWindowText(GetDlgItem(hWnd_CreateGameDialog,
            IDC_GAMENAME), gamename, 32);
        DestroyWindow(hWnd_CreateGameDialog);

        hWnd_CreateViewPlayersDialog = CreateDialog(hInst,
            MAKEINTRESOURCE(IDD_CREATEVIEWPLAYERS),
            hWnd_Application, (DLGPROC)
            CreateViewPlayersDialogProc);

        ShowWindow(hWnd_CreateViewPlayersDialog, SW_SHOW);

        Lobby.SendCreateGame(gamename);
        break;

    case IDC_CANCELCREATEGAME:
        DestroyWindow(hWnd_CreateGameDialog);
        break;

    default:
        if(SendMessage(GetDlgItem(hWnd_CreateGameDialog,
            IDC_GAMENAME), EM_GETMODIFY, 0, 0))
        {
            GetWindowText(GetDlgItem(hWnd_CreateGameDialog,
                IDC_GAMENAME), gamename, 32);

            if(strcmp(gamename, "") == 0)
            {
                EnableWindow(GetDlgItem(hWnd_CreateGameDialog,
                    IDC_DOCREATEGAME), FALSE);
            }
            else
            {
                EnableWindow(GetDlgItem(hWnd_CreateGameDialog,
                    IDC_DOCREATEGAME), TRUE);
            }
        }

        break;
    }

    break;
}

return 0;
}
```

JoinGameDialogProc Function

This dialog procedure runs in the background of the dialog that shows us the players who have joined the game.

```
LRESULT CALLBACK JoinGameDialogProc(HWND hWnd, UINT uMsg, WPARAM wParam,
                                     LPARAM lParam)
{
    switch (uMsg)
    {
    case WM_COMMAND:
        switch(LOWORD(wParam))
        {
        case IDC_JOINCANCEL:
            DestroyWindow(hWnd_JoinGameDialog);
            break;

        default:
            break;
        }

        break;
    }

    return 0;
}
```

LoginDialogProc Function

This function takes in the login information and tries to connect the server. The lobby server runs on UDP port 30003.

```
LRESULT CALLBACK LoginDialogProc(HWND hWnd, UINT uMsg, WPARAM wParam,
                                  LPARAM lParam)
{
    char nickname[30];
    char password[50];
    int ret;

    switch (uMsg)
    {
    case WM_COMMAND:
        {
            switch(LOWORD(wParam))
            {
            case IDC_LOGIN_QUIT:
                PostQuitMessage(0);
                break;

            case IDC_LOGIN_CREATEACCOUNT:
                if(!hWnd_CreateAccountDialog)
                {
                    hWnd_CreateAccountDialog =
                        CreateDialog(hInst,
```

```

        MAKEINTRESOURCE(IDD_CREATEACCOUNT),
        hWnd_Application, (DLGPROC)
        CreateAccountDialogProc);
    }
    break;

case IDC_DOLOGIN:
    // -> First get the IP address of the server
    // from the dialog
    GetDlgItemText(hWnd_LoginDialog,
        IDC_LOGIN_IPADDRESS, serverIP, 16);

    // -> Store the player data in local variables
    GetDlgItemText(hWnd_LoginDialog,
        IDC_LOGIN_NICKNAME, nickname, 30);
    GetDlgItemText(hWnd_LoginDialog,
        IDC_LOGIN_PASSWORD, password, 50);

    // -> Hide the login window
    ShowWindow(hWnd_LoginDialog, SW_HIDE);

    ret = Lobby.GetNetworkClient()->Initialize("",
        serverIP, 30003);

    if(ret == DREAMSOCK_CLIENT_ERROR)
    {
        char text[64];
        sprintf(text, "Could not open client
            socket");

        MessageBox(NULL, text, "Error", MB_OK);
    }

    Lobby.Connect(nickname, password);

    break;

default:
    break;
}
return 0;
}
case WM_CLOSE:
{
    PostQuitMessage(0);
    break;
}
}
return 0;
}

```


LobbyDialogProc Function

Now here is the main lobby dialog procedure function. It controls all the user commands in the lobby screen, such as chat messaging.

```
LRESULT CALLBACK LobbyDialogProc(HWND hWnd, UINT uMsg, WPARAM wParam,
                                LPARAM lParam)
{
    char chatMessage[256];
    char temp[256];
    int selectedGame;

    switch (uMsg)
    {
    case WM_COMMAND:
        switch(LOWORD(wParam))
        {
        case IDC_LOGOUT:
            Lobby.Disconnect();

            // Hide the lobby
            ShowWindow(hWnd_LobbyDialog, SW_HIDE);
            // Recreate the login dialog
            hWnd_LoginDialog = CreateDialog(hInst, MAKEINTRESOURCE
                (IDD_LOGIN_DIALOG), hWnd_Application,
                (DLGPROC)LoginDialogProc);
            break;

        case IDC_SENDCHATMESSAGE:
            GetWindowText(GetDlgItem(hWnd_LobbyDialog,
                IDC_CHATMESSAGE), temp, 255);

            sprintf(chatMessage, "%s: ", Lobby.GetLocalClient()->
                nickname);
            strcat(chatMessage, temp);

            Lobby.SendChat(chatMessage);

            SetWindowText(GetDlgItem(hWnd_LobbyDialog,
                IDC_CHATMESSAGE), "");
            break;

        case IDC_CREATEGAME:
            hWnd_CreateGameDialog = CreateDialog(hInst,
                MAKEINTRESOURCE(IDD_CREATEGAME),
                hWnd_Application, (DLGPROC) CreateGameDialogProc);

            ShowWindow(hWnd_CreateGameDialog, SW_SHOW);
            break;

        case IDC_JOINGAME:
            selectedGame = SendMessage(GetDlgItem(hWnd_LobbyDialog,
                IDC_GAMELIST), LB_GETCURSEL, 0, 0);
```

```

        hWnd_JoinGameDialog = CreateDialog(hInst,
            MAKEINTRESOURCE(IDD_JOINVIEWPLAYERS),
            hWnd_Application, (DLGPROC) JoinGameDialogProc);

        ShowWindow(hWnd_JoinGameDialog, SW_SHOW);

        break;

    default:
        int count = SendMessage(GetDlgItem(hWnd_LobbyDialog,
            IDC_GAMELIST), LB_GETSELCOUNT, 0, 0);

        if(count)
        {
            int sel = SendMessage(GetDlgItem(hWnd_LobbyDialog,
                IDC_GAMELIST), LB_GETCURSEL, 0, 0);
        }
        else
        {
            EnableWindow(GetDlgItem(hWnd_LobbyDialog,
                IDC_JOININGAME), FALSE);
        }

        break;
    }

    break;

    case WM_CLOSE:
        Lobby.Disconnect();
        PostQuitMessage(0);
        break;
    }

    return 0;
}

```

If the player presses the Log Out button, the client disconnects from the server and hides the lobby window, bringing the login dialog in front.

```

    case IDC_LOGOUT:
        Lobby.Disconnect();

        // Hide the lobby
        ShowWindow(hWnd_LobbyDialog, SW_HIDE);
        // Recreate the login dialog
        hWnd_LoginDialog = CreateDialog(hInst, MAKEINTRESOURCE(IDD_LOGINDIALOG),
            hWnd_Application, (DLGPROC) LoginDialogProc);
        break;

```

When the player presses the Send Chat Message button, the text entered in the chat text box is sent to the server in a chat message. The player's name is also included in the text.

```

case IDC_SENDCHATMESSAGE:
    GetWindowText(GetDlgItem(hWnd_LobbyDialog, IDC_CHATMESSAGE), temp, 255);

    sprintf(chatMessage, "%s: ", Lobby.GetLocalClient()->nickname);
    strcat(chatMessage, temp);

    Lobby.SendChat(chatMessage);

    SetWindowText(GetDlgItem(hWnd_LobbyDialog, IDC_CHATMESSAGE), "");
    break;

```

When the player presses the Create Game button, the create game dialog opens.

```

case IDC_CREATEGAME:
    hWnd_CreateGameDialog = CreateDialog(hInst, MAKEINTRESOURCE
        (IDD_CREATEGAME),
        hWnd_Application, (DLGPROC) CreateGameDialogProc);

    ShowWindow(hWnd_CreateGameDialog, SW_SHOW);
    break;

```

When the player presses the Join Game button, the join game dialog opens. The selected game is retrieved with a Windows system function.

```

case IDC_JOINGAME:
    selectedGame = SendMessage(GetDlgItem(hWnd_LobbyDialog, IDC_GAMELIST),
        LB_GETCURSEL, 0, 0);

    hWnd_JoinGameDialog = CreateDialog(hInst, MAKEINTRESOURCE
        (IDD_JOINVIEWPLAYERS), hWnd_Application, (DLGPROC)
        JoinGameDialogProc);

    ShowWindow(hWnd_JoinGameDialog, SW_SHOW);

    break;

```

WinMain Function

This function is almost the same as before but with a small difference. This time we run the frame for both sign-in and lobby systems.

```

int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    TCHAR *pCmdLine, int nCmdShow)
{
    int time, oldTime, newTime;

    WNDCLASSEX wcl;

    // Create our main window
    wcl.cbSize = sizeof(WNDCLASSEX);

    wcl.hInstance = hInstance;
    wcl.lpszClassName = "ArmyWar";

```

```
wc1.lpfWndProc = ApplicationProc;
wc1.style = 0;

wc1.hIcon = LoadIcon(NULL, IDI_APPLICATION);
wc1.hIconSm = LoadIcon(NULL, IDI_WINLOGO);
wc1.hCursor = LoadCursor(NULL, IDC_ARROW);

wc1.lpszMenuName = NULL;
wc1.cbClsExtra = 0;
wc1.cbWndExtra = 0;

wc1.hbrBackground = (HBRUSH) GetStockObject(LTGRAY_BRUSH);

if(!RegisterClassEx(&wc1)) return 0;

hWnd_Application = CreateWindow(
    "ArmyWar",
    "ARMY WAR Online 2.0",
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    640,
    480,
    HWND_DESKTOP,
    NULL,
    hInstance,
    NULL
);

// Initialize the network library
if(dreamSock_Initialize() != 0)
{
    MessageBox(NULL, "Error initializing Communication Library!",
        "Fatal Error", MB_OK);

    return 1;
}

ShowWindow(hWnd_Application, nCmdShow);
UpdateWindow(hWnd_Application);

// Set global instance variable
hInst = hInstance;

// Display the login dialog
hWnd_LoginDialog = CreateDialog(hInst, MAKEINTRESOURCE(IDD_LOGINDIALOG),
    hWnd_Application, (DLGPROC)LoginDialogProc);

// Create the lobby
hWnd_LobbyDialog = CreateDialog(hInst, MAKEINTRESOURCE(IDD_LOBBYDIALOG),
    hWnd_Application, (DLGPROC)LobbyDialogProc);

MSG msg;
```

```

BOOL bMsg = FALSE;

PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE);

oldTime = dreamSock_GetCurrentSystemTime();

bool done = false;

try
{
    while(!done)
    {
        while(PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE))
        {
            if(!GetMessage(&msg, NULL, 0, 0))
            {
                Lobby.Disconnect();
                Signin.Disconnect();
                done = true;
            }

            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }

        do
        {
            newTime = dreamSock_GetCurrentSystemTime();
            time = newTime - oldTime;
        } while (time < 1);

        // Run lobby and sign-in network
        Lobby.RunNetwork(time);
        Signin.RunNetwork(time);

        // Run the game frame here

        oldTime = newTime;
    }
}
catch(...)
{
    Lobby.Disconnect();
    Signin.Disconnect();

    LogString("Unknown Exception caught in main loop");

    MessageBox(NULL, "Unknown Exception caught in main loop",
        "Error", MB_OK | MB_TASKMODAL);

    return -1;
}

```

```
        return msg.wParam;
    }

```

Here we run the frames for sign-in and lobby:

```
// Run lobby and sign-in network
Lobby.RunNetwork(time);
Signin.RunNetwork(time);

```

lobby.cpp File — CLobby Class Methods

Here is the lobby.cpp file. As you can see, it is almost the same as signin.cpp. This time we will focus only on new and modified functions.

```

/*****
/* Programming Multiplayer Games      */
/* Tutorial game client                */
/* Programming:                       */
/* Teijo Hakala                       */
*****/

#include "common.h"
#include "resource.h"

//-----
// Name: empty()
// Desc:
//-----
CLobby::CLobby()
{
    networkClient    = new dreamClient;
    clientList       = NULL;
    gameAmount       = 0;
    timeConnecting   = 0;
}

//-----
// Name: empty()
// Desc:
//-----
CLobby::~CLobby()
{
    delete networkClient;
}

//-----
// Name: empty()
// Desc:
//-----
void CLobby::RefreshPlayerList(void)
{

```

```

        SendMessage(GetDlgItem(hWnd_LobbyDialog, IDC_PLAYERLIST),
                    LB_RESETCONTENT, 0, 0);

        clientLoginData *list = clientList;

        for(; list != NULL; list = list->next)
        {
            SendMessage(GetDlgItem(hWnd_LobbyDialog, IDC_PLAYERLIST),
                        LB_ADDSTRING, 0, (LPARAM) list->nickname);
        }
    }

//-----
// Name: empty()
// Desc:
//-----
void CLobby::RefreshGameList(void)
{
}

//-----
// Name: empty()
// Desc:
//-----
void CLobby::RefreshJoinedPlayersList(void)
{
}

//-----
// Name: empty()
// Desc:
//-----
void CLobby::ReadPackets(void)
{
    char data[1400];
    struct sockaddr address;

    int type;
    int ind;
    int local;
    int ret;
    char name[30];

    dreamMessage mes;
    mes.Init(data, sizeof(data));

    while(ret = networkClient->GetPacket(mes.data, &address))
    {
        mes.SetSize(ret);
        mes.BeginReading();

        type = mes.ReadByte();

        switch(type)

```

```
{
case DREAMSOCK_MES_ADDCLIENT:
    local = mes.ReadByte();
    ind   = mes.ReadByte();
    strcpy(name, mes.ReadString());

    AddClient(local, ind, name);
    break;

case DREAMSOCK_MES_REMOVECLIENT:
    ind = mes.ReadByte();

    LogString("Got removeclient %d message", ind);

    RemoveClient(ind);
    break;

case USER_MES_SERVEREXIT:
    MessageBox(NULL, "Server disconnected", "Info", MB_OK);
    Disconnect();
    break;

case USER_MES_LOGIN:
    // Skip sequences
    mes.ReadShort();
    mes.ReadShort();

    ret = mes.ReadShort();

    LogString("Got lobby login respond %d", ret);

    if(ret != LOBBYLOGIN_RESULT_ACCEPTED)
    {
        MessageBox(NULL, "Nickname or password is not valid",
                    "Error", MB_OK);

        Disconnect();

        return;
    }

    SetWindowText(hWnd_Application, "ARMY WAR Online 2.0 -
                    connected");
    timeConnecting = -1;

    ShowWindow(hWnd_LobbyDialog, SW_SHOW);

    break;

case USER_MES_CHAT:
    // Skip sequences
    mes.ReadShort();
    mes.ReadShort();
```



```

        SendMessage(GetDlgItem(hWnd_LobbyDialog, IDC_CHATLIST),
                    LB_ADDSTRING, 0, (LPARAM) mes.ReadString());

        break;
    }
}

//-----
// Name: empty()
// Desc:
//-----
void CLobby::AddClient(int local, int ind, char *name)
{
    // First get a pointer to the beginning of client list
    clientLoginData *list = clientList;
    clientLoginData *prev;

    LogString("App: Client: Adding client with index %d", ind);

    // No clients yet, adding the first one
    if(clientList == NULL)
    {
        LogString("App: Client: Adding first client");

        clientList = (clientLoginData *) calloc(1,
            sizeof(clientLoginData));

        if(local)
        {
            LogString("App: Client: This one is local");
            localClient = clientList;
        }

        clientList->index = ind;
        strcpy(clientList->nickname, name);
        strcpy(clientList->nickname, name);

        clientList->next = NULL;
    }
    else
    {
        LogString("App: Client: Adding another client");

        prev = list;
        list = clientList->next;

        while(list != NULL)
        {
            prev = list;
            list = list->next;
        }
    }
}

```

```

list = (clientLoginData *) calloc(1, sizeof(clientLoginData));

if(local)
{
    LogString("App: Client: This one is local");
    localClient = list;
}

list->index = ind;
strcpy(list->nickname, name);

list->next = NULL;

prev->next = list;
}

RefreshPlayerList();

// If we just joined the game, request all the game data
if(local)
    RequestGameData();
}

//-----
// Name: empty()
// Desc:
//-----
void CLobby::RemoveClient(int ind)
{
    clientLoginData *list = clientList;
    clientLoginData *prev = NULL;
    clientLoginData *next = NULL;

    for(; list != NULL; list = list->next)
    {
        if(list->index == ind)
        {
            if(prev != NULL)
            {
                prev->next = list->next;
            }

            break;
        }

        prev = list;
    }

    if(list == clientList)
    {
        if(list)
        {
            next = list->next;
            free(list);
        }
    }
}

```

```

        }

        list = NULL;
        clientList = next;
    }
    else
    {
        if(list)
        {
            next = list->next;
            free(list);
        }

        list = next;
    }

    RefreshPlayerList();
}

//-----
// Name: empty()
// Desc:
//-----
void CLobby::RemoveClients(void)
{
    clientLoginData *list = clientList;
    clientLoginData *next;

    while(list != NULL)
    {
        if(list)
        {
            next = list->next;
            free(list);
        }

        list = next;
    }

    clientList = NULL;
}

//-----
// Name: empty()
// Desc:
//-----
void CLobby::AddGame(char *name, int ind, bool inProgress)
{
}

//-----
// Name: empty()
// Desc:
//-----

```

```
void CLobby::RemoveGame(char *name)
{
}

//-----
// Name: empty()
// Desc:
//-----
void CLobby::RemoveGames(void)
{
}

//-----
// Name: empty()
// Desc:
//-----
void CLobby::RequestGameData(void)
{
    char data[1400];
    dreamMessage message;
    message.Init(data, sizeof(data));

    message.WriteByte(USER_MES_GAMEDATA);
    message.AddSequences(networkClient);

    networkClient->SendPacket(&message);
}

//-----
// Name: empty()
// Desc:
//-----
void CLobby::SendChat(char *text)
{
    char data[1400];
    dreamMessage message;
    message.Init(data, sizeof(data));

    message.WriteByte(USER_MES_CHAT);
    message.AddSequences(networkClient);
    message.WriteString(text);

    networkClient->SendPacket(&message);
}

//-----
// Name: empty()
// Desc:
//-----
void CLobby::SendCreateGame(char *gamename)
{
    char data[1400];
    dreamMessage message;
```

```

        message.Init(data, sizeof(data));

        message.WriteByte(USER_MES_CREATEGAME);
        message.AddSequences(networkClient);
        message.WriteString(gamename);

        networkClient->SendPacket(&message);
    }

//-----
// Name: empty()
// Desc:
//-----
void CLobby::SendRemoveGame(char *gamename)
{
    char data[1400];
    dreamMessage message;
    message.Init(data, sizeof(data));

    message.WriteByte(USER_MES_REMOVEGAME);
    message.AddSequences(networkClient);
    message.WriteString(gamename);

    networkClient->SendPacket(&message);
}

//-----
// Name: empty()
// Desc:
//-----
void CLobby::SendStartGame(int ind)
{
    char data[1400];
    dreamMessage message;
    message.Init(data, sizeof(data));

    message.WriteByte(USER_MES_STARTGAME);
    message.AddSequences(networkClient);
    message.WriteByte(ind);

    networkClient->SendPacket(&message);
}

//-----
// Name: empty()
// Desc:
//-----
void CLobby::SendKeepAlive(void)
{
    char data[1400];
    dreamMessage message;
    message.Init(data, sizeof(data));

    message.WriteByte(USER_MES_KEEPLIVE);

```

```

        message.AddSequences(networkClient);

        networkClient->SendPacket(&message);
    }

//-----
// Name: empty()
// Desc:
//-----
void CLobby::Connect(char *name, char *password)
{
    LogString("CLobby::Connect");

    timeConnecting = 0;
    SetWindowText(hWnd_Application, "ARMY WAR Online 2.0 - connecting ...");

    networkClient->SendConnect(name);

    dreamMessage message;
    char data[1400];

    message.Init(data, sizeof(data));
    message.WriteByte(USER_MES_LOGIN);           // type
    message.AddSequences(networkClient);         // sequences
    message.WriteString(name);                  // name
    message.WriteString(password);              // password

    networkClient->SendPacket(&message);
}

//-----
// Name: empty()
// Desc:
//-----
void CLobby::Disconnect(void)
{
    LogString("CLobby::Disconnect");

    timeConnecting = -1;
    SetWindowText(hWnd_Application, "ARMY WAR Online 2.0");

    localClient = NULL;

    RemoveClients();
    RemoveGames();
    networkClient->SendDisconnect();
}

//-----
// Name: empty()
// Desc:
//-----
void CLobby::RunNetwork(int msec)
{

```

```

        if(networkClient->GetConnectionState() == DREAMSOCK_DISCONNECTED)
            return;

        static int time = 0;
        static int keepalive = 0;
        time += msec;
        keepalive += msec;

        if(keepalive > 20000)
        {
            SendKeepAlive();
            keepalive = 0;
        }

        // If timeconnecting is negative, we are connected or connection failed
        if(timeConnecting > -1)
            timeConnecting += msec;

        if(timeConnecting > 3000)
        {
            SetWindowText(hWnd_Application, "ARMY WAR Online 2.0");

            MessageBox(NULL, "Could not connect", "Connection error", MB_OK);
            timeConnecting = -1;

            ShowWindow(hWnd_LoginDialog, SW_SHOW);
        }

        // framerate is too high
        if(time < (1000 / 30))
            return;

        time = 0;

        ReadPackets();
    }

```

RefreshPlayerList Function

This function updates the player list whenever called. First, the whole list is reset, and then it is filled with all the connected players.

```

void CLobby::RefreshPlayerList(void)
{
    SendMessage(GetDlgItem(hWnd_LobbyDialog, IDC_PLAYERLIST),
        LB_RESETCONTENT, 0, 0);

    clientLoginData *list = clientList;

    for(; list != NULL; list = list->next)
    {

```

```

        SendMessage(GetDlgItem(hWnd_LobbyDialog, IDC_PLAYERLIST),
                    LB_ADDSTRING, 0, (LPARAM) list->nickname);
    }
}

```

ReadPackets Function

Here is the `ReadPackets` function once again. The lobby system has a few of its own messages that we will look at now.

```

void CLobby::ReadPackets(void)
{
    char data[1400];
    struct sockaddr address;

    int type;
    int ind;
    int local;
    int ret;
    char name[30];

    dreamMessage mes;
    mes.Init(data, sizeof(data));

    while(ret = networkClient->GetPacket(mes.data, &address))
    {
        mes.SetSize(ret);
        mes.BeginReading();

        type = mes.ReadByte();

        switch(type)
        {
            case DREAMSOCK_MES_ADDCLIENT:
                local = mes.ReadByte();
                ind = mes.ReadByte();
                strcpy(name, mes.ReadString());

                AddClient(local, ind, name);
                break;

            case DREAMSOCK_MES_REMOVECLIENT:
                ind = mes.ReadByte();

                LogString("Got removeclient %d message", ind);

                RemoveClient(ind);
                break;

            case USER_MES_SERVEREXIT:
                MessageBox(NULL, "Server disconnected", "Info", MB_OK);
                Disconnect();
        }
    }
}

```



```

        break;

    case USER_MES_LOGIN:
        // Skip sequences
        mes.ReadShort();
        mes.ReadShort();

        ret = mes.ReadShort();

        LogString("Got lobby login respond %d", ret);

        if(ret != LOBBYLOGIN_RESULT_ACCEPTED)
        {
            MessageBox(NULL, "Nickname or password is not
                valid", "Error", MB_OK);

            Disconnect();

            return;
        }

        SetWindowText(hWnd_Application, "ARMY WAR Online 2.0 -
            connected");
        timeConnecting = -1;

        ShowWindow(hWnd_LobbyDialog, SW_SHOW);

        break;

    case USER_MES_CHAT:
        // Skip sequences
        mes.ReadShort();
        mes.ReadShort();

        SendMessage(GetDlgItem(hWnd_LobbyDialog, IDC_CHATLIST),
            LB_ADDSTRING, 0, (LPARAM) mes.ReadString());

        break;
    }
}
}

```

If the client receives a login message, we know that the server has processed our login request and now we get the result. If it was not accepted, we disconnect from the server. Otherwise, we show the lobby dialog and set a new window title that has the text “connected” in it. The variable `timeConnecting` is set to `-1` to indicate that it should not be counted anymore. More on this variable later.

```

case USER_MES_LOGIN:
    // Skip sequences
    mes.ReadShort();
    mes.ReadShort();

```

```

ret = mes.ReadShort();

LogString("Got lobby login respond %d", ret);

if(ret != LOBBYLOGIN_RESULT_ACCEPTED)
{
    MessageBox(NULL, "Nickname or password is not valid", "Error",
                MB_OK);

    Disconnect();

    return;
}

SetWindowText(hWnd_Application, "ARMY WAR Online 2.0 - connected");
timeConnecting = -1;

ShowWindow(hWnd_LobbyDialog, SW_SHOW);

break;

```

Let's say someone has written something to the chat system, and the server has sent us the chat message. Here is how we put it up on the chat window.

```

case USER_MES_CHAT:
    // Skip sequences
    mes.ReadShort();
    mes.ReadShort();

    SendMessage(GetDlgItem(hWnd_LobbyDialog, IDC_CHATLIST), LB_ADDSTRING,
                0, (LPARAM) mes.ReadString());

break;

```

RequestGameData Function

This function sends a "request game data" message to the server.

```

void CLobby::RequestGameData(void)
{
    char data[1400];
    dreamMessage message;
    message.Init(data, sizeof(data));

    message.WriteByte(USER_MES_GAMEDATA);
    message.AddSequences(networkClient);

    networkClient->SendPacket(&message);
}

```

SendChat Function

This function sends a “chat” message to the server.

```
void CLobby::SendChat(char *text)
{
    char data[1400];
    dreamMessage message;
    message.Init(data, sizeof(data));

    message.WriteByte(USER_MES_CHAT);
    message.AddSequences(networkClient);
    message.WriteString(text);

    networkClient->SendPacket(&message);
}
```

SendCreateGame Function

This function sends a “create game” message to the server.

```
void CLobby::SendCreateGame(char *gamename)
{
    char data[1400];
    dreamMessage message;
    message.Init(data, sizeof(data));

    message.WriteByte(USER_MES_CREATEGAME);
    message.AddSequences(networkClient);
    message.WriteString(gamename);

    networkClient->SendPacket(&message);
}
```

SendRemoveGame Function

This function sends a “remove game” message to the server.

```
void CLobby::SendRemoveGame(char *gamename)
{
    char data[1400];
    dreamMessage message;
    message.Init(data, sizeof(data));

    message.WriteByte(USER_MES_REMOVEGAME);
    message.AddSequences(networkClient);
    message.WriteString(gamename);

    networkClient->SendPacket(&message);
}
```

SendStartGame Function

This function sends a “start game” message to the server.

```
void CLobby::SendStartGame(int ind)
{
    char data[1400];
    dreamMessage message;
    message.Init(data, sizeof(data));

    message.WriteByte(USER_MES_STARTGAME);
    message.AddSequences(networkClient);
    message.WriteByte(ind);

    networkClient->SendPacket(&message);
}
```

Connect Function

This function connects the lobby server and then sends the login request right after that. The variable `timeConnecting` is set to 0, so the system will start counting how long we have tried to connect.

```
void CLobby::Connect(char *name, char *password)
{
    LogString("CLobby::Connect");

    timeConnecting = 0;
    SetWindowText(hWnd_Application, "ARMY WAR Online 2.0 - connecting ...");

    networkClient->SendConnect(name);

    dreamMessage message;
    char data[1400];

    message.Init(data, sizeof(data));
    message.WriteByte(USER_MES_LOGIN); // type
    message.AddSequences(networkClient); // sequences
    message.WriteString(name); // name
    message.WriteString(password); // password

    networkClient->SendPacket(&message);
}
```

Disconnect Function

This function disconnects from the lobby server. Everything is uninitialized.

```
void CLobby::Disconnect(void)
{
    LogString("CLobby::Disconnect");
}
```

```

timeConnecting = -1;
SetWindowText(hWnd_Application, "ARMY WAR Online 2.0");

localClient = NULL;

RemoveClients();
RemoveGames();
networkClient->SendDisconnect();
}

```

RunNetwork Function

This function is run every frame to keep the lobby network running.

```

void CLobby::RunNetwork(int msec)
{
    if(networkClient->GetConnectionState() == DREAMSOCK_DISCONNECTED)
        return;

    static int time = 0;
    static int keepalive = 0;
    time += msec;
    keepalive += msec;

    if(keepalive > 20000)
    {
        SendKeepAlive();
        keepalive = 0;
    }

    // If timeConnecting is negative, we are connected or connection failed
    if(timeConnecting > -1)
        timeConnecting += msec;

    if(timeConnecting > 3000)
    {
        SetWindowText(hWnd_Application, "ARMY WAR Online 2.0");

        MessageBox(NULL, "Could not connect", "Connection error", MB_OK);
        timeConnecting = -1;

        ShowWindow(hWnd_LoginDialog, SW_SHOW);
    }

    // framerate is too high
    if(time < (1000 / 30))
        return;

    time = 0;

    ReadPackets();
}

```

Here we count how long we have tried to connect to the server. If it takes longer than it normally should, it probably means that the server is down or we have the wrong address. In this case, we tell the user that the client could not connect to the server, and we bring the login window back up.

```
// If timeConnecting is negative, we are connected or connection failed
if(timeConnecting > -1)
    timeConnecting += msec;

if(timeConnecting > 3000)
{
    SetWindowText(hWnd_Application, "ARMY WAR Online 2.0");

    MessageBox(NULL, "Could not connect", "Connection error", MB_OK);
    timeConnecting = -1;

    ShowWindow(hWnd_LoginDialog, SW_SHOW);
}
```

Unimplemented Functions

There are some functions that cannot be implemented until we have the game data structure ready. We will implement these functions in the next tutorial, “Creating Your Online Game.”

```
void CLobby::RefreshGameList(void)
{
}

void CLobby::RefreshJoinedPlayersList(void)
{
}

void CLobby::AddGame(char *name, int ind, bool inProgress)
{
}

void CLobby::RemoveGame(char *name)
{
}

void CLobby::RemoveGames(void)
{
}
```

Lobby Server Code

We now learn how to create the lobby server. It is very similar to the sign-in server, and so we will only take a look at the new and modified parts of the code.

lobby.h File

Here we have the lobby.h file, which contains CLobbyServer class. It works just like CSigninServer.

```
#ifndef _LOBBY_H
#define _LOBBY_H

#define LOBBY_RESULT_ACCEPTED          200
#define LOBBY_RESULT_USERNAMEBAD      201
#define LOBBY_RESULT_PASSWORDBAD      202
#define LOBBY_RESULT_MYSQLERROR       203

class CLobbyServer
{
private:
    dreamServer      *networkServer;
    clientLoginData  *clientList;

    int gameAmount;

public:
    CLobbyServer();
    ~CLobbyServer();

    int  InitNetwork(void);
    void ShutdownNetwork(void);

    void ReadPackets(void);
    void SendExitNotification(void);

    void AddClient(void);
    void RemoveClient(struct sockaddr *address);
    void RemoveClients(void);

    void AddGame(char *name);
    void RemoveGame(char *name);
    void RemoveGames(void);

    void Frame(int msec);

    int  GetGameAmount(void)    {return gameAmount;}
};

#endif
```

There is one new variable that tells us how many games exist at the moment: `int gameAmount;`

network.h File

This file is almost the same as the one in the sign-in tutorial, but now we have some more message types.

```
#ifndef NETWORK_H
#define NETWORK_H

#include "dreamSock.h"

#define USER_MES_SERVEREXIT    1
#define USER_MES_LOGIN        2
#define USER_MES_SIGNIN       3
#define USER_MES_CHAT         4
#define USER_MES_CREATEGAME    5
#define USER_MES_REMOVEGAME    6

typedef struct clientLoginData
{
    dreamClient      *netClient;
    clientLoginData *next;
} clientLoginData;

#endif
```

main.cpp File

This file has so few changes that we will skip right to them. The application entry functions (`WinMain` and `main`) now initialize both the sign-in and lobby servers, and run both server's frames.

```
CLobbyServer Lobby;
CSigninServer Signin;

...

if(Lobby.InitNetwork() == 1)
{
    PostQuitMessage(0);
}

if(Signin.InitNetwork() == 1)
{
    PostQuitMessage(0);
}

...
```



```
Lobby.Frame(time);
Signin.Frame(time);
```

lobby.cpp File — CLobbyServer Class Methods

Here is the lobby.cpp file, which contains the code for CLobbyServer. Note that the lobby server runs on UDP port 30003.

We will now take a look at the new and modified functions.

```

/*****
/* Programming Multiplayer Games      */
/* Tutorial game server                */
/* Programming:                        */
/* Teijo Hakala                        */
*****/

#include "common.h"
#include <malloc.h>
#include <mysql++>

//-----
// Name: empty()
// Desc:
//-----
CLobbyServer::CLobbyServer()
{
    networkServer = new dreamServer;

    clientList = NULL;

    gameAmount = 0;
}

//-----
// Name: empty()
// Desc:
//-----
CLobbyServer::~CLobbyServer()
{
    delete networkServer;
}

//-----
// Name: InitNetwork()
// Desc: Initialize network
//-----
int CLobbyServer::InitNetwork(void)
{
    // Initialize dreamSock and the server
    if(dreamSock_Initialize() != 0)
    {
        LogString("Error initializing Communication Library!");
        return 1;
    }
}

```

```

    }

    int ret = networkServer->Initialize("", 30003);

    if(ret == DREAMSOCK_SERVER_ERROR)
    {
#ifdef Win32
        char text[64];
        sprintf(text, "Could not open server on port %d",
            networkServer->GetPort());

        MessageBox(NULL, text, "Error", MB_OK);
#else
        LogString("Could not open server on port %d",
            networkServer->GetPort());
#endif
        return 1;
    }

    return 0;
}

//-----
// Name: ShutdownNetwork()
// Desc: Shut down network
//-----
void CLobbyServer::ShutdownNetwork(void)
{
    LogString("Shutting down lobby server...");

    SendExitNotification();
    RemoveClients();
    RemoveGames();
}

//-----
// Name: empty()
// Desc:
//-----
void CLobbyServer::ReadPackets(void)
{
    char data[1400];

    int type;
    int ret;

    // Some incoming data
    char name[30];
    char password[50];
    char chatter[50];
    int respond;
    int ind;

    struct sockaddr address;

```

```

clientLoginData *cList;

dreamMessage mes;
mes.Init(data, sizeof(data));

// Get the packet from the socket
try
{
    while(ret = networkServer->GetPacket(mes.data, &address))
    {
        mes.SetSize(ret);
        mes.BeginReading();

        type = mes.ReadByte();

        // Check the type of the message
        switch(type)
        {
            case DREAMSOCK_MES_CONNECT:
                AddClient();
                break;

            case DREAMSOCK_MES_DISCONNECT:
                RemoveClient(&address);

                if(cList == NULL)
                    RemoveGames();

                break;

            case USER_MES_LOGIN:
                // Skip sequences
                mes.ReadShort();
                mes.ReadShort();

                strcpy(name, mes.ReadString());
                strcpy(password, mes.ReadString());

                LogString("Lobby: Player %s logged in", name);

                try
                {
                    // -> Create a connection to the database
                    Connection con("onlinedata", "127.0.0.1");

                    // -> Create a query object that is bound
                    // to our connection
                    Query query = con.query();

                    // -> Assign the query to that object
                    query << "SELECT id,firstname,password FROM
                        playerdata WHERE nickname = \"\" <<
                        name << \"\"";
                }
            }
        }
    }
}

```

```

// -> Store the results from the query
Result res = query.store();

Result::iterator i;
Row row;
i = res.begin();

if(i!=res.end())
{
    row = *i;
    if(!strcmp(password, row["password"]))
    {
        // -> Update the 'lastlogin'
        // field to current date and time
        query << "UPDATE playerdata SET
                lastlogin = NULL WHERE
                id = " << row["id"];
        query.execute();

        // -> Set the player to 'online'
        query << "UPDATE playerdata SET
                online = 1 WHERE id = "
                << row["id"];
        query.execute();

        // -> Player login successful!
        respond = LOBBYLOGIN_RESULT_
                ACCEPTED;
    }
    else
    {
        // -> Password did not match
        respond = LOBBYLOGIN_RESULT_
                PASSWORDBAD;
    }
}
else
{
    // -> Nickname could not be found
    respond = LOBBYLOGIN_RESULT_USERNAMEBAD;
}
}

catch (BadQuery er) // handle any connection errors
{
    // -> MySQL server not running?
    respond = LOBBYLOGIN_RESULT_MYSQLERROR;
}

respond = LOBBY_RESULT_ACCEPTED;

// Find the correct client by comparing addresses
c1List = clientList;

for(; c1List != NULL; c1List = c1List->next)

```

```

    {
        if(memcmp(cList->netClient->
            GetSocketAddress(), &address,
            sizeof(address)) == 0)
        {
            cList->netClient->message.Init
                (cList->netClient->
                message.outgoingData,
                sizeof(cList->netClient->
                message.outgoingData));

            cList->netClient->message.WriteByte
                (USER_MES_LOGIN); // type
            cList->netClient->message.AddSequences
                (cList->netClient); // sequences
            cList->netClient->message.WriteShort
                (respond); // respond
            cList->netClient->SendPacket();

            LogString("Sending lobby login
                respond");

            break;
        }
    }
    break;

case USER_MES_CHAT:
    // Skip sequences
    mes.ReadShort();
    mes.ReadShort();

    // Read chat text
    strcpy(chatter, mes.ReadString());

    // Send the chat text to everybody
    for(cList = clientList; cList != NULL; cList =
        cList->next)
    {
        cList->netClient->message.Init(cList->
            netClient->message.outgoingData,
            sizeof(cList->netClient->
            message.outgoingData));

        cList->netClient->message.WriteByte
            (USER_MES_CHAT); // type
        cList->netClient->message.AddSequences
            (cList->netClient); // sequences
        cList->netClient->message.WriteString
            (chatter); // text
    }

networkServer->SendPackets();

```

```
        break;

    case USER_MES_CREATEGAME:
        // Skip sequences
        mes.ReadShort();
        mes.ReadShort();

        // Read game name
        strcpy(name, mes.ReadString());

        AddGame(name);

        // Find the correct client by comparing addresses
        cList = clientList;

        for(; cList != NULL; cList = cList->next)
        {
            if(memcmp(cList->netClient->
                GetSocketAddress(), &address,
                sizeof(address)) == 0)
            {
                ind = cList->netClient->GetIndex();
                break;
            }
        }

        // Send to everybody
        for(cList = clientList; cList != NULL; cList =
            cList->next)
        {
            cList->netClient->message.Init(cList->
                netClient->message.outgoingData,
                sizeof(cList->netClient->
                    message.outgoingData));

            cList->netClient->message.WriteByte
                (USER_MES_CREATEGAME); // type
            cList->netClient->message.AddSequences
                (cList->netClient); // sequences
            cList->netClient->message.WriteString(name);
            // game name
            cList->netClient->message.WriteShort(ind);
            // host's index
            cList->netClient->message.WriteByte(0);
            // in progress?
        }

        networkServer->SendPackets();

        break;

    case USER_MES_REMOVEGAME:
        // Skip sequences
        mes.ReadShort();
```

```

        mes.ReadShort();

        // Read game name
        strcpy(name, mes.ReadString());

        LogString("REMOVING %s", name);

        RemoveGame(name);

        // Send to everybody
        for(cList = clientList; cList != NULL; cList =
            cList->next)
        {
            cList->netClient->message.Init(cList->
                netClient->message.outgoingData,
                sizeof(cList->netClient->
                    message.outgoingData));

            cList->netClient->message.WriteByte(USER_
                MES_REMOVEGAME); // type
            cList->netClient->message.AddSequences
                (cList->netClient); // sequences
            cList->netClient->message.WriteString(name);
            // game name
        }

        networkServer->SendPackets();

        break;
    }
}
catch(...)
{
    LogString("Unknown Exception caught in Lobby ReadPackets loop");

#ifdef Win32
    MessageBox(NULL, "Unknown Exception caught in Lobby ReadPackets
        loop", "Error", MB_OK | MB_TASKMODAL);
#endif
}

//-----
// Name: empty()
// Desc:
//-----
void CLobbyServer::SendExitNotification(void)
{
    clientLoginData *toClient = clientList;

    for(; toClient != NULL; toClient = toClient->next)
    {

```

```

        toClient->netClient->message.Init(toClient->netClient->
            message.outgoingData, sizeof(toClient->netClient->
            message.outgoingData));

        toClient->netClient->message.WriteByte(USER_MES_SERVEREXIT);
        // type
        toClient->netClient->message.AddSequences(toClient->netClient);
        // sequences
    }

    networkServer->SendPackets();
}

//-----
// Name: empty()
// Desc:
//-----
void CLobbyServer::AddClient(void)
{
    // First get a pointer to the beginning of the client list
    clientLoginData *list = clientList;
    clientLoginData *prev;
    dreamClient *netList = networkServer->GetClientList();

    // No clients yet, adding the first one
    if(clientList == NULL)
    {
        LogString("App: Server: Adding first client");

        clientList = (clientLoginData *) calloc(1,
            sizeof(clientLoginData));

        clientList->netClient = netList;

        clientList->next = NULL;
    }
    else
    {
        LogString("App: Server: Adding another client");

        prev = list;
        list = clientList->next;
        netList = netList->next;

        while(list != NULL)
        {
            prev = list;
            list = list->next;
            netList = netList->next;
        }

        list = (clientLoginData *) calloc(1, sizeof(clientLoginData));
    }
}

```



```

        list->netClient = netList;

        list->next = NULL;

        prev->next = list;
    }
}

//-----
// Name: empty()
// Desc:
//-----
void CLobbyServer::RemoveClient(struct sockaddr *address)
{
    clientLoginData *list = clientList;
    clientLoginData *prev = NULL;
    clientLoginData *next = NULL;

    for(; list != NULL; list = list->next)
    {
        if(memcmp(list->netClient->GetSocketAddress(), address,
            sizeof(address)) == 0)
        {
            if(prev != NULL)
            {
                prev->next = list->next;
            }

            break;
        }

        prev = list;
    }

    if(list == clientList)
    {
        if(list)
        {
            next = list->next;
            free(list);
        }

        list = NULL;
        clientList = next;
    }
    else
    {
        if(list)
        {
            next = list->next;
            free(list);
        }

        list = next;
    }
}

```

```
    }
}

//-----
// Name: empty()
// Desc:
//-----
void CLobbyServer::RemoveClients(void)
{
    clientLoginData *list = clientList;
    clientLoginData *next;

    while(list != NULL)
    {
        if(list)
        {
            next = list->next;
            free(list);
        }

        list = next;
    }

    clientList = NULL;
}

//-----
// Name: empty()
// Desc:
//-----
void CLobbyServer::AddGame(char *name)
{
}

//-----
// Name: empty()
// Desc:
//-----
void CLobbyServer::RemoveGame(char *name)
{
}

//-----
// Name: empty()
// Desc:
//-----
void CLobbyServer::RemoveGames(void)
{
}

//-----
// Name: empty()
// Desc:
//-----
```

```
void CLobbyServer::Frame(int msec)
{
    ReadPackets();
}
```

ReadPackets Function

This function reads the packets from the lobby clients.

```
void CLobbyServer::ReadPackets(void)
{
    char data[1400];

    int type;
    int ret;

    // Some incoming data
    char name[30];
    char password[50];
    char chatter[50];
    int respond;
    int ind;

    struct sockaddr address;

    clientLoginData *cList;

    dreamMessage mes;
    mes.Init(data, sizeof(data));

    // Get the packet from the socket
    try
    {
        while(ret = networkServer->GetPacket(mes.data, &address))
        {
            mes.SetSize(ret);
            mes.BeginReading();

            type = mes.ReadByte();

            // Check the type of the message
            switch(type)
            {
                case DREAMSOCK_MES_CONNECT:
                    AddClient();
                    break;

                case DREAMSOCK_MES_DISCONNECT:
                    RemoveClient(&address);

                    if(cList == NULL)
                        RemoveGames();
            }
        }
    }
}
```

```
break;

case USER_MES_LOGIN:
    // Skip sequences
    mes.ReadShort();
    mes.ReadShort();

    strcpy(name, mes.ReadString());
    strcpy(password, mes.ReadString());

    LogString("Lobby: Player %s logged in", name);

try
{
    // -> Create a connection to the database
    Connection con("onlinedata", "127.0.0.1");

    // -> Create a query object that is bound
    // to our connection
    Query query = con.query();

    // -> Assign the query to that object
    query << "SELECT id,firstname,password FROM
            playerdata WHERE nickname = \" <<
            name << \"\"";

    // -> Store the results from the query
    Result res = query.store();

    Result::iterator i;
    Row row;
    i = res.begin();

    if(i!=res.end())
    {
        row = *i;
        if(!strcmp(password, row["password"]))
        {
            // -> Update the 'lastlogin'
            // field to current date and time
            query << "UPDATE playerdata SET
                    lastlogin = NULL WHERE
                    id = \" << row["id"];
            query.execute();

            // -> Set the player to 'online'
            query << "UPDATE playerdata SET
                    online = 1 WHERE id = \"
                    << row["id"];
            query.execute();

            // -> Player login successful!
            respond = LOBBYLOGIN_RESULT_
                ACCEPTED;
```

```

    }
    else
    {
        // -> Password did not match
        respond = LOBBYLOGIN_RESULT_
                PASSWORDBAD;
    }
}
else
{
    // -> Nickname could not be found
    respond = LOBBYLOGIN_RESULT_USERNAMEBAD;
}
}
catch (BadQuery er) // handle any connection errors
{
    // -> MySQL server not running?
    respond = LOBBYLOGIN_RESULT_MYSQLERROR;
}

// Find the correct client by comparing addresses
cList = clientList;

for(; cList != NULL; cList = cList->next)
{
    if(memcmp(cList->netClient->
        GetSocketAddress(), &address,
        sizeof(address)) == 0)
    {
        cList->netClient->message.Init(cList->
            netClient->message.outgoingData,
            sizeof(cList->netClient->
                message.outgoingData));

        cList->netClient->message.WriteByte
            (USER_MES_LOGIN); // type
        cList->netClient->message.AddSequences
            (cList->netClient); // sequences
        cList->netClient->message.WriteShort
            (respond); // respond
        cList->netClient->SendPacket();

        LogString("Sending lobby login
            respond");

        break;
    }
}
break;

case USER_MES_CHAT:
    // Skip sequences
    mes.ReadShort();

```

```
mes.ReadShort();

// Read chat text
strcpy(chatter, mes.ReadString());

// Send the chat text to everybody
for(cList = clientList; cList != NULL; cList =
    cList->next)
{
    cList->netClient->message.Init(cList->
        netClient->message.outgoingData,
        sizeof(cList->netClient->
            message.outgoingData));

    cList->netClient->message.WriteByte
        (USER_MES_CHAT); // type
    cList->netClient->message.AddSequences
        (cList->netClient); // sequences
    cList->netClient->message.WriteString
        (chatter); // text
}

networkServer->SendPackets();

break;

case USER_MES_CREATEGAME:
    // Skip sequences
    mes.ReadShort();
    mes.ReadShort();

    // Read game name
    strcpy(name, mes.ReadString());

    AddGame(name);

    // Find the correct client by comparing addresses
    cList = clientList;

    for(; cList != NULL; cList = cList->next)
    {
        if(memcmp(cList->netClient->
            GetSocketAddress(), &address,
            sizeof(address)) == 0)
        {
            ind = cList->netClient->GetIndex();
            break;
        }
    }

    // Send to everybody
    for(cList = clientList; cList != NULL; cList =
        cList->next)
    {
```

```

        cList->netClient->message.Init(cList->
            netClient->message.outgoingData,
            sizeof(cList->netClient->
                message.outgoingData));

        cList->netClient->message.WriteByte(USER_
            MES_CREATEGAME); // type
        cList->netClient->message.AddSequences
            (cList->netClient); // sequences
        cList->netClient->message.WriteString(name);
        // game name
        cList->netClient->message.WriteShort(ind);
        // host's index
        cList->netClient->message.WriteByte(0);
        // in progress?
    }

    networkServer->SendPackets();

    break;

case USER_MES_REMOVEGAME:
    // Skip sequences
    mes.ReadShort();
    mes.ReadShort();

    // Read game name
    strcpy(name, mes.ReadString());

    LogString("REMOVING %s", name);

    RemoveGame(name);

    // Send to everybody
    for(cList = clientList; cList != NULL; cList =
        cList->next)
    {
        cList->netClient->message.Init(cList->
            netClient->message.outgoingData,
            sizeof(cList->netClient->
                message.outgoingData));

        cList->netClient->message.WriteByte(USER_
            MES_REMOVEGAME); // type
        cList->netClient->message.AddSequences
            (cList->netClient); // sequences
        cList->netClient->message.WriteString(name);
        // game name
    }

    networkServer->SendPackets();

    break;
}

```

```

    }
}
catch(...)
{
    LogString("Unknown Exception caught in Lobby ReadPackets loop");
}

#ifdef Win32
    MessageBox(NULL, "Unknown Exception caught in Lobby ReadPackets
        loop", "Error", MB_OK | MB_TASKMODAL);
#endif
}
}

```

In case of a chat message, we forward the message to every client and do nothing else.

```

case USER_MES_CHAT:
    // Skip sequences
    mes.ReadShort();
    mes.ReadShort();

    // Read chat text
    strcpy(chatter, mes.ReadString());

    // Send the chat text to everybody
    for(c1List = clientList; c1List != NULL; c1List = c1List->next)
    {
        c1List->netClient->message.Init(c1List->netClient->
            message.outgoingData, sizeof(c1List->netClient->
            message.outgoingData));

        c1List->netClient->message.WriteByte(USER_MES_CHAT); // type
        c1List->netClient->message.AddSequences(c1List->netClient);
        // sequences
        c1List->netClient->message.WriteString(chatter); // text
    }

    networkServer->SendPackets();

    break;

```

If a client sends us the “create game” message, we first create a game on the server side and then forward the message to every client, so they can add a game too. We find out the index number of the game creator, so that player will open a different kind of dialog than the rest (where he or she can start the game).

```

case USER_MES_CREATEGAME:
    // Skip sequences
    mes.ReadShort();
    mes.ReadShort();

```



```

// Read game name
strcpy(name, mes.ReadString());

AddGame(name);

// Find the correct client by comparing addresses
cList = clientList;

for(; cList != NULL; cList = cList->next)
{
    if(memcmp(cList->netClient->GetSocketAddress(), &address,
        sizeof(address)) == 0)
    {
        ind = cList->netClient->GetIndex();
        break;
    }
}

// Send to everybody
for(cList = clientList; cList != NULL; cList = cList->next)
{
    cList->netClient->message.Init(cList->netClient->
        message.outgoingData, sizeof(cList->netClient->
        message.outgoingData));

    cList->netClient->message.WriteByte(USER_MES_CREATEGAME);
    // type
    cList->netClient->message.AddSequences(cList->netClient);
    // sequences
    cList->netClient->message.WriteString(name); // game name
    cList->netClient->message.WriteShort(ind); // host's index
    cList->netClient->message.WriteByte(0); // in progress?
}

networkServer->SendPackets();

break;

```

If we get a “remove game” message, the server first removes the game locally, and then tells the clients about it so they will remove it too.

```

case USER_MES_REMOVEGAME:
    // Skip sequences
    mes.ReadShort();
    mes.ReadShort();

    // Read game name
    strcpy(name, mes.ReadString());

    LogString("REMOVING %s", name);

    RemoveGame(name);

```

```
// Send to everybody
for(cList = clientList; cList != NULL; cList = cList->next)
{
    cList->netClient->message.Init(cList->netClient->
        message.outgoingData, sizeof(cList->netClient->
        message.outgoingData));

    cList->netClient->message.WriteByte(USER_MES_REMOVEGAME);
    // type
    cList->netClient->message.AddSequences(cList->netClient);
    // sequences
    cList->netClient->message.WriteString(name);    // game name
}

networkServer->SendPackets();

break;
```

Unimplemented Functions

The server also has some unimplemented functions (just like the lobby client) at this point. These cannot be implemented until we have the game data structure.

```
void CLobbyServer::AddGame(char *name)
{
}

void CLobbyServer::RemoveGame(char *name)
{
}

void CLobbyServer::RemoveGames(void)
{
}
```

Summary

In this tutorial we learned how to create a working game lobby. We learned how to handle players who are logging in, create games, and send chat messages. Now we have only one final step to take: making the actual game. This is covered in the final tutorial, “Creating Your Online Game.”

This page intentionally left blank.

Tutorial 5

Creating Your Online Game

Introduction

This is the last tutorial for our project of creating our very own online game. In this tutorial we learn how to make our game handle player input and show the result on the local and remote clients. Basically we learn how we should send the player movement and other command data to the other clients. The server will route all the commands to all the clients, and we can then see the movement of the players and the flag on our screens. We will also learn how to create a dead reckoning system (also known as client prediction). This is a method of making the players move smoothly even when the server is not responding very quickly.

All the hosts participating in the network game must produce exactly the same frames. If that is not the case, the server's frame is used. But the rule of thumb is that the server and a client always produce the same frames. Obviously when we send this over the network to the remote clients, they try their best to match it but cannot do it perfectly. So when a client moves a remote client's player on the screen, it is possible that the result is wrong. In this case, the server tells us where to put it.

Designing the Functionality

There are some methods to make our game work better over a network. These methods should be kept in mind when designing our game's network functionality.

Frame Time

All computer games should keep track of the frame time. This is especially important in a multiplayer game because all the computers could be running the game at slightly different speeds (depending on how powerful the computer is). So when we move something in the game world, we should scale the velocity of the movement with the current frame time. This way the movement is smooth on every frame, and no client is moving the objects faster than any other.

Compressing Messages

While there are obviously ways to compress almost any kind of data, we are not talking about compressing the message byte by byte. Of course, one could implement that kind of method if required, but the packets usually are so small that it is not needed (or even useful).

We are talking about delta compression. Delta means change over a time. For example, if we press a button down now, we need to tell that to the server only once, right? Why should we send that same information every frame when it already knows that the key is pressed down? Then again, if we release the key now, we need to send that information to the server. Between pressing down and releasing the key, we do not need to inform the server about that key in anyway. That is called delta compression.

Another good example is sending the player coordinates from the server to a client. Why send them if they have not changed?

Delta compression is achieved by using single bits of a byte (or any length of data) to tell what is included in a message. This is called flagging, and each bit in the byte is a flag. For example, if the "key" flag is up, it means that this message contains key data.

Flags are defined in values of power of two. The first one is 1, the second one is 2, then 4, 8, 16, 32, 64, etc. A byte consists of 8 bits, so it can have eight of these values. Here are the command flags used in this tutorial game:

```
#define CMD_KEY          1
#define CMD_HEADING     2
#define CMD_ORIGIN      4
#define CMD_BULLET     8
```

```
#define CMD_FLAG      16
#define CMD_KILL     32
```

The following code shows how to set the “key” flag up and down:

```
flags |= CMD_KEY;      // Set the flag UP
flags &= ~CMD_KEY;    // Set the flag DOWN
```

The following code shows how to check if a flag is up:

```
if(flags & CMD_KEY)
```

So setting a flag up or down is actually setting a bit to 1 or 0. And because a byte is the smallest amount of data you can send over a network, this is a good way send lots of boolean values in one byte.

Dead Reckoning

Dead reckoning, or client prediction, is an advanced method of scaling down our required network bandwidth. The server does not have to inform the clients as often as it would have to without dead reckoning.

Dead reckoning is a method used in various places (i.e., not only in network games), such as aviation. It is used to predict our position at a given time, using our current position and velocity to calculate it.

Normally when we press a key or do any action in the game, information about this is sent to the server, and the server sends it to each client. When the information reaches the clients, they make the action happen. Now, as you can imagine, there is a slight delay before the clients receive that info and make the action happen. There is nothing we can do about the delay concerning remote clients, but the local client is a totally different matter. Obviously if we press a key and send it to the server, we already know that we did it. So why wait for the server to tell us that?

The idea of dead reckoning is this: When we press a key to make our player move forward, we send this information to the server, but we also immediately start to move our player forward on the local machine. When the server has processed the message and sent us information, we look at the position where the server thinks the player is, and compare it to what we thought it to be at that time (we have to save the old results for comparison). If they match, there is no problem. If they do not match, we obey the server and move the player to the position it told us. Sometimes this may be seen as player warping — because the server is always right. Not the client. No matter what people tell you.

When we use dead reckoning, the server does not have to send data to us as often. We save lots of bandwidth this way, and that is nothing but a good thing.

Frame History

There is no need to store every frame in the history array. We store only the last 64 frames. To get the correct index in the array, we use the following method:

```
totalFrame & (COMMAND_HISTORY_SIZE-1)
```

This returns a value from 0 to a maximum of 63 (COMMAND_HISTORY_SIZE-1).

We use the message sequences as our frame counter, so usually you will see something like this:

```
outgoingSequence & (COMMAND_HISTORY_SIZE-1)
```

Handling Messages

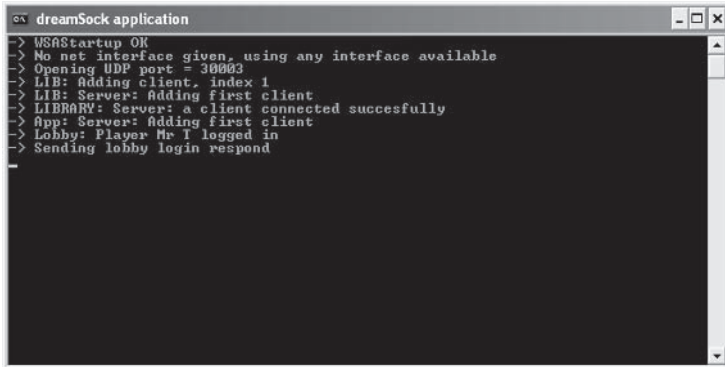
Because we use dead reckoning, we can save some network bandwidth by making the server send messages in 100 ms (or any other suitable time) intervals. Why can we do this then? Because the clients will try to predict the frames during this 100 ms by just moving the objects in the direction they were moving in the last known frame (using the last known velocity). But the server must get information about every client frame, so all the clients have to send frame data every frame.

The server reads incoming packets every frame, and when it receives a message containing frame data, it runs the game logic for that client. So the server is not constantly running the game logic for all the clients; it only runs it when a frame data message is received. Keeping in mind that the clients send frame data on every frame, we end up running the game logic about every frame on the server too. This method keeps the server and clients in sync, even if the other is running on a significantly faster computer.

Game Server Code

In contrast to the previous tutorials, here we begin with the server-side code because it may be a bit easier to understand. And when we know how the server works, it is easier to understand the client code. So let's see how the game server code works. Like before, the basic network communication methods are the same, so we will not focus on them anymore. We need some new source code files: `server.cpp`, `network.cpp`, and `server.h`. Also, some of the old files have changed a little bit.

Figure 1



```
dreamSock application
-> WSASStartup OK
-> No net interface given, using any interface available
-> Opening UDP port = 30003
-> LIB: Adding client, index 1
-> LIB: Server: Adding first client
-> LIBRARY: Server: a client connected succesfully
-> App: Server: Adding first client
-> Lobby: Player Mr T logged in
-> Sending lobby login respond
```

server.h File

This header file contains the application-specific data structures. This means that the player data structures are here. Also the game's main class is located in this file. This class has the network interface methods that we learned to create in the other tutorials. A `CArmyWarServer` object is created each time a new game is created, and this object processes that game's functions.

```
#ifndef SERVER_H
#define SERVER_H

#include <string.h>

#define NORTH 0
#define NORTHEAST 45
#define EAST 90
#define SOUTHEAST 135
#define SOUTH 180
#define SOUTHWEST 225
#define WEST 270
#define NORTHWEST 315

#define BLUE_TEAM 0
#define RED_TEAM 1

typedef struct
{
    float x;
    float y;
} VECTOR2D;

typedef struct bullet_t
{
    VECTOR2D vel; // Velocity
    VECTOR2D origin; // Position
```



```

        bool        shot;        // Is the bullet in the air?
        int         lifetime;    // Lifetime in ms
    } bullet_t;

typedef struct
{
    int key;          // Pressed keys
    int heading;     // Heading

    VECTOR2D vel;    // Velocity
    VECTOR2D origin; // Position

    bullet_t bullet; // Bullet

    int msec;        // How long to run command (in ms)
} command_t;

typedef struct clientData
{
    command_t frame[COMMAND_HISTORY_SIZE];
    command_t serverFrame;
    command_t command;

    long processedFrame;

    dreamClient *netClient;

    VECTOR2D startPos;
    bool team;
    bool diedThisFrame;

    clientData *next;
} clientData;

class CArmyWarServer
{
private:
    dreamServer *networkServer;

    clientData *clientList; // Client list
    int clients;           // Number of clients

    int realtime;          // Real server up-time in ms
    int servertime;       // Server frame * 100 ms
    float frametime;      // Frame time in seconds

    char gamename[32];
    bool inProgress;

    bool mapdata[100][100];
    int index;

    float flagX;

```

```

float      flagY;
clientData *playerWithFlag;
bool       updateFlag;
bool       updateKill;

int        redScore;
int        blueScore;

long       framenum;

public:
    CArmyWarServer();
    ~CArmyWarServer();

    // Network.cpp
    void ReadPackets(void);
    void SendCommand(void);
    void SendExitNotification(void);
    void ReadDeltaMoveCommand(dreamMessage *mes, clientData *client);
    void BuildMoveCommand(dreamMessage *mes, clientData *client);
    void BuildDeltaMoveCommand(dreamMessage *mes, clientData *client);

    // Server.cpp
    int  InitNetwork(int gameAmount);
    void ShutdownNetwork(void);
    void GenerateRandomMap(void);
    void CalculateVelocity(command_t *command, float frametime);
    void CalculateHeading(command_t *command);
    void CalculateBulletVelocity(command_t *command);
    void MovePlayers(void);
    void MovePlayer(clientData *client);
    void CheckFlagCollisions(void);

    void AddClient(void);
    void RemoveClient(struct sockaddr *address);
    void RemoveClients(void);
    void Frame(int msec);

    clientData *GetClientList(void) {return clientList;}

    void SetName(char *n)           {strcpy(gamename, n);}
    char *GetName(void)            {return gamename;}

    void SetInProgress(bool p)     {inProgress = p;}
    bool GetInProgress(void)       {return inProgress;}

    void SetIndex(int ind)        {index = ind;}
    int  GetIndex(void)           {return index;}

    CArmyWarServer *next;
};
#endif

```

Each frame's commands are kept in a structure, which is seen below. All the variables are easy to understand except for one perhaps, and that is the integer `msec`. It is basically that frame's frame time, and hence it is the length of the command (in milliseconds).

```
int key;           // Pressed keys
int heading;      // Heading

VECTOR2D vel;     // Velocity
VECTOR2D origin;  // Position

bullet_t bullet;  // Bullet

int msec;         // How long to run command (in ms)
```

The players' game-specific data (such as their origin) is stored in the `clientData` structure. As you can see, there is more than one `command_t` object. One is for the current frame, one is for the last known server frame, and one is a history of the last 64 frames.

```
command_t frame[COMMAND_HISTORY_SIZE];
command_t serverFrame;
command_t command;

long processedFrame;

dreamClient *netClient;

VECTOR2D startPos;
bool team;
bool diedThisFrame;

clientData *next;
```

The `command_t frame` is an array (history) of the last 64 frames. The `command_t serverFrame` is the last known server frame. The `command_t command` is the current frame. The `long processedFrame` tells us the last frame we have processed. This is a frame history indexed value (0 to 63).

The `dreamClient netClient` is a pointer to the `dreamClient` network client that matches with this player. The `VECTOR2D startPos` is the start position for the player. The boolean `team` is the player's team (red = 1 or blue = 0). The boolean `diedThisFrame` is a flag that indicates if the player died this frame (used to inform all the clients which players died during the current frame). The `clientData next` is a pointer to the next client.

`CArmyWarServer` also has some member variables that we should look at before we move on:

```
private:
    dreamServer *networkServer;

    clientData *clientList;    // Client list
    int        clients;       // Number of clients

    int        realtime;      // Real server up-time in ms
    int        servertime;    // Server frame * 100 ms
    float      frametime;     // Frame time in seconds

    char       gamename[32];
    bool       inProgress;

    bool       mapdata[100][100];
    int        index;

    float      flagX;
    float      flagY;
    clientData *playerWithFlag;
    bool       updateFlag;
    bool       updateKill;

    int        redScore;
    int        blueScore;

    long       framenum;
```

The `dreamServer networkServer` is the `dreamServer` network server object. The `clientData clientList` is the game-specific client list holding the players' data such as their current origin. The integer `clients` tells us how many clients have joined this game. The integer `realtime` tells us in real time how long this server has been running (in milliseconds). The integer `servertime` is used to make each network frame last for 100 milliseconds. The formula for this variable is: `server frame number * 100 ms`.

The `char gamename` stores the game's name. The boolean `inProgress` is a flag that tells us if the game is in progress or not. The boolean `mapdata` is an array that holds the map information. If a node is true, a tree exists in that position. If the value is false, there is nothing but grass.

The integer `index` tells us the index number of the game. The floats `flagX` and `flagY` store the flag origin. The `clientData playerWithFlag` is a pointer to the player with the flag (if any).

The boolean `updateFlag` is a flag that is set when the flag information should be sent to each client. The boolean `updateKill` is a flag that is set when someone dies. This information should be sent to each client. The integers `redScore` and `blueScore` keep track of the team scores. The long `frameNum` counts the frames.

network.h File

Here is the final version of the server-side network.h. There are some new definitions here, such as command history size and the keyboard commands.

```
#ifndef NETWORK_H
#define NETWORK_H

#include "dreamSock.h"

#define COMMAND_HISTORY_SIZE          64

#define KEY_UP                        1
#define KEY_DOWN                      2
#define KEY_LEFT                      4
#define KEY_RIGHT                    8
#define KEY_WEAPON                   16

#define CMD_KEY                       1
#define CMD_HEADING                  2
#define CMD_ORIGIN                   4
#define CMD_BULLET                   8
#define CMD_FLAG                     16
#define CMD_KILL                     32

#define USER_MES_FRAME               1
#define USER_MES_NONDELTAFRAME      2
#define USER_MES_SERVEREXIT         3
#define USER_MES_LOGIN               4
#define USER_MES_SIGNIN              5
#define USER_MES_CHAT                6
#define USER_MES_CREATEGAME          7
#define USER_MES_REMOVEGAME          8
#define USER_MES_GAMEDATA            9
#define USER_MES_STARTGAME          10
#define USER_MES_MAPDATA             11

typedef struct clientLoginData
{
    dreamClient          *netClient;
    clientLoginData     *next;
} clientLoginData;

#endif
```

main.cpp File

The server's main source code file has not changed much. Only the application's main functions (WinMain and main) have something new. We now run each game's frame right after the lobby and sign-in frames. We go through the whole game list and run every game's frames.

```

Lobby.Frame(time);
Signin.Frame(time);

CArmyWarServer *list = Lobby.GetGameList();

for(; list != NULL; list = list->next)
{
    list->Frame(time);
}

```

network.cpp File — CArmyWarServer Class Part 1

Network.cpp holds the network part of the CArmyWarServer class. Here we receive, process, build, and send packets.

```

/*****
/* Programming Multiplayer Games      */
/* Tutorial game server                */
/* Programming:                        */
/* Teijo Hakala                       */
*****/

#include "common.h"
#include <math.h>

#ifdef Win32
#include <windows.h>
#include <mmsystem.h>
#include <assert.h>
#endif

//-----
// Name: InitNetwork()
// Desc: Initialize network
//-----
int CArmyWarServer::InitNetwork(int gameAmount)
{
    if(dreamSock_Initialize() != 0)
    {
        LogString("Error initializing Communication Library!");
        return 1;
    }

    LogString("Initializing game %d", gameAmount);

    // Create the game servers on new ports, starting from 30004
    int ret = networkServer->Initialize("", 30004 + gameAmount);

    if(ret == DREAMSOCK_SERVER_ERROR)
    {
#ifdef Win32
        char text[64];

```

```

        sprintf(text, "Could not open server on port %d",
                networkServer->GetPort());

        MessageBox(NULL, text, "Error", MB_OK);
#else
        LogString("Could not open server on port %d",
                networkServer->GetPort());
#endif
        return 1;
    }

    return 0;
}

//-----
// Name: ShutdownNetwork()
// Desc: Shut down network
//-----
void CArmyWarServer::ShutdownNetwork(void)
{
    LogString("Shutting down game server...");

    RemoveClients();

    networkServer->Uninitialize();
}

//-----
// Name: empty()
// Desc:
//-----
void CArmyWarServer::ReadPackets(void)
{
    char data[1400];

    int type;
    int ret;

    struct sockaddr address;

    clientData *cList;

    dreamMessage mes;
    mes.Init(data, sizeof(data));

    // Get the packet from the socket
    try
    {
        while(ret = networkServer->GetPacket(mes.data, &address))
        {
            mes.SetSize(ret);
            mes.BeginReading();
        }
    }
}

```

```
type = mes.ReadByte();

// Check the type of the message
switch(type)
{
case DREAMSOCK_MES_CONNECT:
    AddClient();
    break;

case DREAMSOCK_MES_DISCONNECT:
    RemoveClient(&address);
    break;

case USER_MES_FRAME:
    // Skip sequences
    mes.ReadShort();
    mes.ReadShort();

    // Find the correct client by comparing addresses
    cList = clientList;

    for(; cList != NULL; cList = cList->next)
    {
        if(memcmp(cList->netClient->
            GetSocketAddress(), &address,
            sizeof(address)) == 0)
        {
            ReadDeltaMoveCommand(&mes, cList);
            MovePlayer(cList);

            break;
        }
    }

    break;

case USER_MES_NONDELTAFRAME:
    cList = clientList;
    clientData *dataClient;

    // Find the correct client by comparing addresses
    for(; cList != NULL; cList = cList->next)
    {
        if(memcmp(cList->netClient->
            GetSocketAddress(), &address,
            sizeof(address)) == 0)
        {
            break;
        }
    }
}
```



```

cList->netClient->message.Init(cList->netClient->
    message.outgoingData, sizeof(cList->
    netClient->message.outgoingData));

cList->netClient->message.WriteByte(USER_MES_
    NONDELTAFRAME);
cList->netClient->message.WriteShort(cList->
    netClient->GetOutgoingSequence());
cList->netClient->message.WriteShort(cList->
    netClient->GetIncomingSequence());

for(dataClient = clientList; dataClient != NULL;
    dataClient = dataClient->next)
{
    BuildMoveCommand(&cList->netClient->message,
        dataClient);
}

cList->netClient->SendPacket();

break;

case USER_MES_STARTGAME:
    // Skip sequences
    mes.ReadShort();
    mes.ReadShort();

    // Send to everybody
    for(cList = clientList; cList != NULL; cList =
        cList->next)
    {
        cList->netClient->message.Init(cList->
            netClient->message.outgoingData,
            sizeof(cList->netClient->
            message.outgoingData));

        cList->netClient->message.WriteByte(USER_
            MES_MAPDATA); // type
        cList->netClient->message.AddSequences
            (cList->netClient); // sequences

        for(int i = 0; i < 100; i++)
        {
            for(int j = 0; j < 100; j++)
            {
                if(mapdata[i][j] == true)
                {
                    cList->netClient->
                        message.WriteByte(i);
                    cList->netClient->
                        message.WriteByte(j);
                }
            }
        }
    }
}

```

```

    }

    networkServer->SendPackets();

    // Send to everybody
    for(cList = clientList; cList != NULL; cList =
        cList->next)
    {
        cList->netClient->message.Init(cList->
            netClient->message.outgoingData,
            sizeof(cList->netClient->
                message.outgoingData));

        cList->netClient->message.WriteByte(USER_
            MES_STARTGAME); // type
        cList->netClient->message.AddSequences
            (cList->netClient); // sequences
    }

    networkServer->SendPackets();

    break;
}
}
}
catch(...)
{
    LogString("Unknown Exception caught in Lobby ReadPackets loop");

#ifdef Win32
    MessageBox(NULL, "Unknown Exception caught in Lobby ReadPackets
        loop", "Error", MB_OK | MB_TASKMODAL);
#endif
}

//-----
// Name: empty()
// Desc:
//-----
void CArmyWarServer::AddClient(void)
{
    // First get a pointer to the beginning of the client list
    clientData *list = clientList;
    clientData *prev;
    dreamClient *netList = networkServer->GetClientList();

    // No clients yet, adding the first one
    if(clientList == NULL)
    {
        LogString("App: Server: Adding first client");

        clientList = (clientData *) calloc(1, sizeof(clientData));
    }
}

```

```
    clientList->netClient = netList;

    if(clients % 2 == 0)
    {
        clientList->team = RED_TEAM;

        clientList->startPos.x = 46.0f * 32.0f + ((clients/2) *
                                                    32.0f);
        clientList->startPos.y = 96.0f * 32.0f;
    }
    else
    {
        clientList->team = BLUE_TEAM;

        clientList->startPos.x = 46.0f * 32.0f + ((clients/2) *
                                                    32.0f);
        clientList->startPos.y = 4.0f * 32.0f;
    }

    clientList->command.origin.x = clientList->startPos.x;
    clientList->command.origin.y = clientList->startPos.y;

    clientList->next = NULL;
}
else
{
    LogString("App: Server: Adding another client");

    prev = list;
    list = clientList->next;
    netList = netList->next;

    while(list != NULL)
    {
        prev = list;
        list = list->next;
        netList = netList->next;
    }

    list = (clientData *) calloc(1, sizeof(clientData));

    list->netClient = netList;

    if(clients % 2 == 0)
    {
        list->team = RED_TEAM;

        list->startPos.x = 46.0f * 32.0f + ((clients/2) * 32.0f);
        list->startPos.y = 96.0f * 32.0f;
    }
    else
    {
        list->team = BLUE_TEAM;
```

```

        list->startPos.x = 46.0f * 32.0f + ((clients/2) * 32.0f);
        list->startPos.y = 4.0f * 32.0f;
    }

    list->command.origin.x = list->startPos.x;
    list->command.origin.y = list->startPos.y;

    list->next = NULL;

    prev->next = list;
}

clients++;
}

//-----
// Name: empty()
// Desc:
//-----
void CArmyWarServer::RemoveClient(struct sockaddr *address)
{
    clientData *list = clientList;
    clientData *prev = NULL;
    clientData *next = NULL;

    for(; list != NULL; list = list->next)
    {
        if(memcmp(list->netClient->GetSocketAddress(), address,
            sizeof(address)) == 0)
        {
            if(prev != NULL)
            {
                prev->next = list->next;
            }

            break;
        }

        prev = list;
    }

    // Drop the flag if player with flag exits the game
    if(list == playerWithFlag)
    {
        playerWithFlag = NULL;
        updateFlag = true;
    }

    if(list == clientList)
    {
        if(list)
        {
            next = list->next;
            free(list);
        }
    }
}

```

```

        }

        list = NULL;
        clientList = next;
    }
    else
    {
        if(list)
        {
            next = list->next;
            free(list);
        }

        list = next;
    }

    clients--;
}

//-----
// Name: empty()
// Desc:
//-----
void CArmyWarServer::RemoveClients(void)
{
    clientData *list = clientList;
    clientData *next;

    while(list != NULL)
    {
        if(list)
        {
            next = list->next;
            free(list);
        }

        list = next;
    }

    clientList = NULL;
    clients = 0;
}

//-----
// Name: empty()
// Desc:
//-----
void CArmyWarServer::SendCommand(void)
{
    clientData *toClient;
    clientData *dataClient;

    // Fill messages
    for(toClient = clientList; toClient != NULL; toClient = toClient->next)

```

```

    {
        toClient->netClient->message.Init(toClient->netClient->
            message.outgoingData, sizeof(toClient->netClient->
            message.outgoingData));

        toClient->netClient->message.WriteByte(USER_MES_FRAME); // type
        toClient->netClient->message.AddSequences(toClient->netClient);
        // sequences

        for(dataClient = clientList; dataClient != NULL; dataClient =
            dataClient->next)
        {
            BuildDeltaMoveCommand(&toClient->netClient->message,
                dataClient);
        }
    }

    // Send messages to all clients
    networkServer->SendPackets();

    // Store the sent command in history
    for(toClient = clientList; toClient != NULL; toClient = toClient->next)
    {
        int i = (toClient->netClient->GetOutgoingSequence() - 1) &
            (COMMAND_HISTORY_SIZE-1);

        memcpy(&toClient->frame[i], &toClient->command,
            sizeof(command_t));
    }
}

//-----
// Name: empty()
// Desc:
//-----
void CArmyWarServer::SendExitNotification(void)
{
    clientData *toClient = clientList;

    for(; toClient != NULL; toClient = toClient->next)
    {
        toClient->netClient->message.Init(toClient->netClient->
            message.outgoingData, sizeof(toClient->netClient->
            message.outgoingData));

        toClient->netClient->message.WriteByte(USER_MES_SERVEREXIT);
        // type
        toClient->netClient->message.AddSequences(toClient->netClient);
        // sequences
    }

    networkServer->SendPackets();
}

```

```

//-----
// Name: empty()
// Desc:
//-----
void CArmyWarServer::ReadDeltaMoveCommand(dreamMessage *mes, clientData
                                         *client)
{
    int flags = 0;

    // Flags
    flags = mes->ReadByte();

    // Key
    if(flags & CMD_KEY)
    {
        client->command.key = mes->ReadByte();

        LogString("Client %d: read CMD_KEY (%d)", client->netClient->
                  GetIndex(), client->command.key);
    }

    // Read time to run command
    client->command.msec = mes->ReadByte();
}

//-----
// Name: empty()
// Desc:
//-----
void CArmyWarServer::BuildMoveCommand(dreamMessage *mes, clientData *client)
{
    // Add to the message
    // Key
    mes->WriteByte(client->command.key);

    // Heading
    mes->WriteShort(client->command.heading);

    // Origin
    mes->WriteFloat(client->command.origin.x);
    mes->WriteFloat(client->command.origin.y);
    mes->WriteFloat(client->command.vel.x);
    mes->WriteFloat(client->command.vel.y);

    mes->WriteFloat(client->command.bullet.origin.x);
    mes->WriteFloat(client->command.bullet.origin.y);
    mes->WriteFloat(client->command.bullet.vel.x);
    mes->WriteFloat(client->command.bullet.vel.y);
    mes->WriteShort(client->command.bullet.lifetime);
    mes->WriteByte(client->command.bullet.shot);

    // Flag & points
    if(playerWithFlag)
        mes->WriteShort(playerWithFlag->netClient->GetIndex());
}

```

```
        else
            mes->WriteShort(-1);

        mes->WriteFloat(flagX);
        mes->WriteFloat(flagY);

        mes->WriteByte(redScore);
        mes->WriteByte(blueScore);

        mes->WriteByte(client->command.msec);
    }

//-----
// Name: empty()
// Desc:
//-----
void CArmyWarServer::BuildDeltaMoveCommand(dreamMessage *mes, clientData
                                           *client)
{
    int flags = 0;

    int last = (client->netClient->GetOutgoingSequence() - 1) &
               (COMMAND_HISTORY_SIZE-1);

    // Check what needs to be updated
    if(client->frame[last].key != client->command.key)
    {
        flags |= CMD_KEY;
    }

    if(client->frame[last].origin.x != client->command.origin.x ||
       client->frame[last].origin.y != client->command.origin.y)
    {
        flags |= CMD_ORIGIN;
    }

    if(client->command.bullet.shot)
    {
        if(client->frame[last].bullet.origin.x != client->
           command.bullet.origin.x ||
           client->frame[last].bullet.origin.y != client->
           command.bullet.origin.y)
        {
            flags |= CMD_BULLET;
        }
    }

    if(client->frame[last].bullet.shot != client->command.bullet.shot)
    {
        flags |= CMD_BULLET;
    }

    if(updateFlag == true)
    {
```



```
        flags |= CMD_FLAG;
    }

    if(updateKill == true)
    {
        flags |= CMD_KILL;
    }

    // Add to the message
    // Flags
    mes->WriteByte(flags);

    // Key
    if(flags & CMD_KEY)
    {
        mes->WriteByte(client->command.key);
    }

    if(flags & CMD_ORIGIN || flags & CMD_BULLET)
    {
        mes->WriteByte(client->processedFrame &
            (COMMAND_HISTORY_SIZE-1));
    }

    // Origin
    if(flags & CMD_ORIGIN)
    {
        mes->WriteFloat(client->command.origin.x);
        mes->WriteFloat(client->command.origin.y);

        mes->WriteFloat(client->command.vel.x);
        mes->WriteFloat(client->command.vel.y);
    }

    // Origin
    if(flags & CMD_BULLET)
    {
        mes->WriteFloat(client->command.bullet.origin.x);
        mes->WriteFloat(client->command.bullet.origin.y);

        mes->WriteFloat(client->command.bullet.vel.x);
        mes->WriteFloat(client->command.bullet.vel.y);

        mes->WriteByte(client->command.bullet.shot);
    }

    // Flag & points
    if(flags & CMD_FLAG)
    {
        if(playerWithFlag)
            mes->WriteShort(playerWithFlag->netClient->GetIndex());
        else
```

```

        mes->WriteShort(0);

        mes->WriteFloat(flagX);
        mes->WriteFloat(flagY);

        mes->WriteByte(redScore);
        mes->WriteByte(blueScore);
    }

    // Someone died
    if(flags & CMD_KILL)
    {
        mes->WriteByte(client->diedThisFrame);
    }

    mes->WriteByte(client->command.msec);
}

```

InitNetwork Function

This function has only one thing different from the previous implementations — the port number and the way we choose it. Every created game opens its own port, and we start from port 30004. The next game would open port number 30005 and the next one 30006 and so on.

```

// Create the game servers on new ports, starting from 30004
int ret = networkServer->Initialize("", 30004 + gameAmount);

```

ReadPackets Function

If a “frame” message is received from a client, the commands are read using the `ReadDeltaMoveCommand` function. The commands are delta-compressed, meaning only the values that have changed are included in the packet. Once the commands are read, they are put into action by running the `MovePlayer` function.

```

case USER_MES_FRAME:
    // Skip sequences
    mes.ReadShort();
    mes.ReadShort();

    // Find the correct client by comparing addresses
    cList = clientList;

    for(; cList != NULL; cList = cList->next)
    {
        if(memcmp(cList->netClient->GetSocketAddress(), &address,
            sizeof(address)) == 0)
        {
            ReadDeltaMoveCommand(&mes, cList);
            MovePlayer(cList);

            break;
        }
    }
}

```

```

    }
}
break;

```

If we received a “non-delta frame” message, it means that a client wants us to send it the current absolute values of the game. The `BuildMoveCommand` function is used to send the absolute values.

```

case USER_MES_NONDELTAFRAME:
    cList = clientList;
    clientData *dataClient;

    // Find the correct client by comparing addresses
    for(; cList != NULL; cList = cList->next)
    {
        if(memcmp(cList->netClient->GetSocketAddress(), &address,
            sizeof(address)) == 0)
        {
            break;
        }
    }

    cList->netClient->message.Init(cList->netClient->message.outgoingData,
        sizeof(cList->netClient->message.outgoingData));

    cList->netClient->message.WriteByte(USER_MES_NONDELTAFRAME);
    cList->netClient->message.WriteShort(cList->netClient->
        GetOutgoingSequence());
    cList->netClient->message.WriteShort(cList->netClient->
        GetIncomingSequence());

    for(dataClient = clientList; dataClient != NULL; dataClient =
        dataClient->next)
    {
        BuildMoveCommand(&cList->netClient->message, dataClient);
    }

    cList->netClient->SendPacket();

    break;

```

Receiving a “start game” message means that the game host wants to start the game. We send the map data to each client and inform them to start the game.

```

case USER_MES_STARTGAME:
    // Skip sequences
    mes.ReadShort();
    mes.ReadShort();

    // Send to everybody
    for(cList = clientList; cList != NULL; cList = cList->next)

```

```

    {
        cList->netClient->message.Init(cList->netClient->
            message.outgoingData, sizeof(cList->netClient->
            message.outgoingData));

        cList->netClient->message.WriteByte(USER_MES_MAPDATA); // type
        cList->netClient->message.AddSequences(cList->netClient);
        // sequences

        for(int i = 0; i < 100; i++)
        {
            for(int j = 0; j < 100; j++)
            {
                if(mapdata[i][j] == true)
                {
                    cList->netClient->message.WriteByte(i);
                    cList->netClient->message.WriteByte(j);
                }
            }
        }
    }

networkServer->SendPackets();

// Send to everybody
for(cList = clientList; cList != NULL; cList = cList->next)
{
    cList->netClient->message.Init(cList->netClient->
        message.outgoingData, sizeof(cList->netClient->
        message.outgoingData));

    cList->netClient->message.WriteByte(USER_MES_STARTGAME); // type
    cList->netClient->message.AddSequences(cList->netClient);
    // sequences
}

networkServer->SendPackets();

break;

```

SendCommand Function

This function sends the current commands to each client. The commands are then stored in frame history.

```

void CArmyWarServer::SendCommand(void)
{
    clientData *toClient;
    clientData *dataClient;

    // Fill messages
    for(toClient = clientList; toClient != NULL; toClient = toClient->next)
    {

```

```

toClient->netClient->message.Init(toClient->netClient->
    message.outgoingData, sizeof(toClient->netClient->
    message.outgoingData));

toClient->netClient->message.WriteByte(USER_MES_FRAME); // type
toClient->netClient->message.AddSequences(toClient->netClient);
// sequences

for(dataClient = clientList; dataClient != NULL; dataClient =
    dataClient->next)
{
    BuildDeltaMoveCommand(&toClient->netClient->message,
        dataClient);
}

// Send messages to all clients
networkServer->SendPackets();

// Store the sent command in history
for(toClient = clientList; toClient != NULL; toClient = toClient->next)
{
    int i = (toClient->netClient->GetOutgoingSequence() - 1) &
        (COMMAND_HISTORY_SIZE-1);

    memcpy(&toClient->frame[i], &toClient->command,
        sizeof(command_t));
}
}

```

ReadDeltaMoveCommand Function

This function reads a delta-compressed command from a client. A client only sends information about keypresses.

```

void CArmyWarServer::ReadDeltaMoveCommand(dreamMessage *mes, clientData
    *client)
{
    int flags = 0;

    // Flags
    flags = mes->ReadByte();

    // Key
    if(flags & CMD_KEY)
    {
        client->command.key = mes->ReadByte();

        LogString("Client %d: read CMD_KEY (%d)",
            client->netClient->GetIndex(), client->command.key);
    }
}

```

```
    // Read time to run command
    client->command.msec = mes->ReadByte();
}
```

The function takes two parameters (`dreamMessage *mes` and `clientData *client`). The first one is a pointer to the message to read, and the second one is a pointer to the client that owns this message.

We first read the flags byte, which is always there. Depending on the value of this byte, we read the key command. Finally, we read the frame time of this command.

BuildMoveCommand Function

This function builds a non-delta-compressed command message for a client.

```
void CAmyWarServer::BuildMoveCommand(dreamMessage *mes, clientData *client)
{
    // Add to the message
    // Key
    mes->WriteByte(client->command.key);

    // Heading
    mes->WriteShort(client->command.heading);

    // Origin
    mes->WriteFloat(client->command.origin.x);
    mes->WriteFloat(client->command.origin.y);
    mes->WriteFloat(client->command.vel.x);
    mes->WriteFloat(client->command.vel.y);

    mes->WriteFloat(client->command.bullet.origin.x);
    mes->WriteFloat(client->command.bullet.origin.y);
    mes->WriteFloat(client->command.bullet.vel.x);
    mes->WriteFloat(client->command.bullet.vel.y);
    mes->WriteShort(client->command.bullet.lifetime);
    mes->WriteByte(client->command.bullet.shot);

    // Flag & points
    if(playerWithFlag)
        mes->WriteShort(playerWithFlag->netClient->GetIndex());
    else
        mes->WriteShort(-1);

    mes->WriteFloat(flagX);
    mes->WriteFloat(flagY);

    mes->WriteByte(redScore);
    mes->WriteByte(blueScore);

    mes->WriteByte(client->command.msec);
}
```

The function takes two parameters (`dreamMessage *mes` and `clientData *client`). The first one is a pointer to the message to write to, and the second is a pointer to the client that owns the message. The current status of each important variable is written to the message and sent to the client.

BuildDeltaMoveCommand Function

This function builds a delta-compressed command message for a client.

```
void CAmyWarServer::BuildDeltaMoveCommand(dreamMessage *mes, clientData
                                         *client)
{
    int flags = 0;

    int last = (client->netClient->GetOutgoingSequence() - 1) &
               (COMMAND_HISTORY_SIZE-1);

    // Check what needs to be updated
    if(client->frame[last].key != client->command.key)
    {
        flags |= CMD_KEY;
    }

    if(client->frame[last].origin.x != client->command.origin.x ||
       client->frame[last].origin.y != client->command.origin.y)
    {
        flags |= CMD_ORIGIN;
    }

    if(client->command.bullet.shot)
    {
        if(client->frame[last].bullet.origin.x !=
           client->command.bullet.origin.x ||
           client->frame[last].bullet.origin.y !=
           client->command.bullet.origin.y)
        {
            flags |= CMD_BULLET;
        }
    }

    if(client->frame[last].bullet.shot != client->command.bullet.shot)
    {
        flags |= CMD_BULLET;
    }

    if(updateFlag == true)
    {
        flags |= CMD_FLAG;
    }

    if(updateKill == true)
    {
```

```
        flags |= CMD_KILL;
    }

    // Add to the message
    // Flags
    mes->WriteByte(flags);

    // Key
    if(flags & CMD_KEY)
    {
        mes->WriteByte(client->command.key);
    }

    if(flags & CMD_ORIGIN || flags & CMD_BULLET)
    {
        mes->WriteByte(client->processedFrame &
            (COMMAND_HISTORY_SIZE-1));
    }

    // Origin
    if(flags & CMD_ORIGIN)
    {
        mes->WriteFloat(client->command.origin.x);
        mes->WriteFloat(client->command.origin.y);

        mes->WriteFloat(client->command.vel.x);
        mes->WriteFloat(client->command.vel.y);
    }

    // Origin
    if(flags & CMD_BULLET)
    {
        mes->WriteFloat(client->command.bullet.origin.x);
        mes->WriteFloat(client->command.bullet.origin.y);

        mes->WriteFloat(client->command.bullet.vel.x);
        mes->WriteFloat(client->command.bullet.vel.y);

        mes->WriteByte(client->command.bullet.shot);
    }

    // Flag & points
    if(flags & CMD_FLAG)
    {
        if(playerWithFlag)
            mes->WriteShort(playerWithFlag->netClient->GetIndex());
        else
            mes->WriteShort(0);

        mes->WriteFloat(flagX);
        mes->WriteFloat(flagY);

        mes->WriteByte(redScore);
    }
}
```



```

        mes->WriteByte(blueScore);
    }

    // Someone died
    if(flags & CMD_KILL)
    {
        mes->WriteByte(client->diedThisFrame);
    }

    mes->WriteByte(client->command.msec);
}

```

The function takes two parameters (`dreamMessage *mes` and `clientData *client`). The first one is a pointer to the message to write to, and the second is a pointer to the client that owns the message.

First we check what has changed since the last frame and set the correct flags:

```

int flags = 0;

int last = (client->netClient->GetOutgoingSequence() - 1) &
           (COMMAND_HISTORY_SIZE-1);

// Check what needs to be updated
if(client->frame[last].key != client->command.key)
{
    flags |= CMD_KEY;
}

if(client->frame[last].origin.x != client->command.origin.x ||
   client->frame[last].origin.y != client->command.origin.y)
{
    flags |= CMD_ORIGIN;
}

if(client->command.bullet.shot)
{
    if(client->frame[last].bullet.origin.x !=
       client->command.bullet.origin.x ||
       client->frame[last].bullet.origin.y != client->command.bullet.origin.y)
    {
        flags |= CMD_BULLET;
    }
}

if(client->frame[last].bullet.shot != client->command.bullet.shot)
{
    flags |= CMD_BULLET;
}

if(updateFlag == true)
{
    flags |= CMD_FLAG;
}

```

```
}  
  
if(updateKill == true)  
{  
    flags |= CMD_KILL;  
}
```

We write the flags to the message and then we add the changed values to the message by checking which flags are up:

```
// Add to the message  
// Flags  
mes->WriteByte(flags);  
  
// Key  
if(flags & CMD_KEY)  
{  
    mes->WriteByte(client->command.key);  
}  
  
if(flags & CMD_ORIGIN || flags & CMD_BULLET)  
{  
    mes->WriteByte(client->processedFrame & (COMMAND_HISTORY_SIZE-1));  
}  
  
// Origin  
if(flags & CMD_ORIGIN)  
{  
    mes->WriteFloat(client->command.origin.x);  
    mes->WriteFloat(client->command.origin.y);  
  
    mes->WriteFloat(client->command.vel.x);  
    mes->WriteFloat(client->command.vel.y);  
}  
  
// Origin  
if(flags & CMD_BULLET)  
{  
    mes->WriteFloat(client->command.bullet.origin.x);  
    mes->WriteFloat(client->command.bullet.origin.y);  
  
    mes->WriteFloat(client->command.bullet.vel.x);  
    mes->WriteFloat(client->command.bullet.vel.y);  
  
    mes->WriteByte(client->command.bullet.shot);  
}  
  
// Flag & points  
if(flags & CMD_FLAG)  
{  
    if(playerWithFlag)  
        mes->WriteShort(playerWithFlag->netClient->GetIndex());  
    else  
        mes->WriteShort(0);  
}
```

```

        mes->WriteFloat(flagX);
        mes->WriteFloat(flagY);

        mes->WriteByte(redScore);
        mes->WriteByte(blueScore);
    }

    // Someone died
    if(flags & CMD_KILL)
    {
        mes->WriteByte(client->diedThisFrame);
    }

    mes->WriteByte(client->command.msec);

```

server.cpp File — CArmyWarServer Class Part 2

This file contains the last part of the CArmyWarServer class. This part is the logic part where we process the commands.

```

/*****
/* Programming Multiplayer Games      */
/* Tutorial game server                */
/* Programming:                        */
/* Teijo Hakala                        */
*****/

#include "common.h"
#include <fstream>
#include <math.h>
#include <malloc.h>
#include <stdlib.h>

//-----
// Name: empty()
// Desc:
//-----
float VectorLength(VECTOR2D *vec)
{
    return (float) sqrt(vec->x*vec->x + vec->y*vec->y);
}

//-----
// Name: empty()
// Desc:
//-----
VECTOR2D VectorSubtract(VECTOR2D *vec1, VECTOR2D *vec2)
{
    VECTOR2D vec;

    vec.x = vec1->x - vec2->x;
    vec.y = vec1->y - vec2->y;
}

```

```

        return vec;
    }

//-----
// Name: empty()
// Desc:
//-----
CArmyWarServer::CArmyWarServer()
{
    networkServer    = new dreamServer;

    clientList       = NULL;
    clients          = 0;

    realtime         = 0;
    servertime       = 0;

    inProgress       = false;

    index            = 0;
    next             = NULL;

    flagX            = 0.0f;
    flagY            = 0.0f;
    playerWithFlag   = NULL;
    updateFlag       = false;

    redScore         = 0;
    blueScore        = 0;

    framenum         = 0;
}

//-----
// Name: empty()
// Desc:
//-----
CArmyWarServer::~CArmyWarServer()
{
    delete networkServer;
}

//-----
// Name: empty()
// Desc:
//-----
void CArmyWarServer::GenerateRandomMap(void)
{
    // Make all land passable
    for(int i = 0; i < 100; i++)
    {
        for(int j = 0; j < 100; j++)
        {
            mapdata[i][j] = false;
        }
    }
}

```

```

    }
}

// Use the game's index number for random seed
srand(index);

// Place some random trees (avoiding the players' start locations)
for(int m = 0; m < 300; m++)
{
    mapdata[rand()%100][((rand()%80)+10)] = true;
}

// Set the flag position
flagX = 49*32;
flagY = 49*32;

playerWithFlag = NULL;
}

//-----
// Name: empty()
// Desc:
//-----
void CArmyWarServer::CalculateVelocity(command_t *command, float frametime)
{
    int checkX;
    int checkY;

    float multiplier = 100.0f;

    command->vel.x = 0.0f;
    command->vel.y = 0.0f;

    if(command->key & KEY_UP)
    {
        checkX = (int) (command->origin.x/32.0f);
        checkY = (int) ((command->origin.y - multiplier * frametime) /
            32.0f);

        if(mapdata[checkX][checkY] == false)
            command->vel.y += -multiplier * frametime;
    }

    if(command->key & KEY_DOWN)
    {
        checkX = (int) (command->origin.x / 32.0f);
        checkY = (int) ((command->origin.y + multiplier * frametime) /
            32.0f);

        if(mapdata[checkX][checkY] == false)
            command->vel.y += multiplier * frametime;
    }

    if(command->key & KEY_LEFT)

```

```

    {
        checkX = (int) ((command->origin.x - multiplier * frametime) /
                        32.0f);
        checkY = (int) (command->origin.y / 32.0f);

        if(mapdata[checkX][checkY] == false)
            command->vel.x += -multiplier * frametime;
    }

    if(command->key & KEY_RIGHT)
    {
        checkX = (int) ((command->origin.x + multiplier * frametime) /
                        32.0f);
        checkY = (int) (command->origin.y / 32.0f);

        if(mapdata[checkX][checkY] == false)
            command->vel.x += multiplier * frametime;
    }
}

//-----
// Name: empty()
// Desc:
//-----
void CArmyWarServer::CalculateHeading(command_t *command)
{
    // Right
    if((command->vel.x > 0.0f) &&
        (command->vel.y == 0.0f))
    {
        command->heading = EAST;
    }

    // Left
    if((command->vel.x < 0.0f) &&
        (command->vel.y == 0.0f))
    {
        command->heading = WEST;
    }

    // Down
    if((command->vel.y > 0.0f) &&
        (command->vel.x == 0.0f))
    {
        command->heading = SOUTH;
    }

    // Up
    if((command->vel.y < 0.0f) &&
        (command->vel.x == 0.0f))
    {
        command->heading = NORTH;
    }
}

```

```
// Down-Right
if((command->vel.x > 0.0f) &&
    (command->vel.y > 0.0f))
{
    command->heading = SOUTHEAST;
}

// Up-Right
if((command->vel.x > 0.0f) &&
    (command->vel.y < 0.0f))
{
    command->heading = NORTHEAST;
}

// Down-Left
if((command->vel.x < 0.0f) &&
    (command->vel.y > 0.0f))
{
    command->heading = SOUTHWEST;
}

// Up-Left
if((command->vel.x < 0.0f) &&
    (command->vel.y < 0.0f))
{
    command->heading = NORTHWEST;
}
}

//-----
// Name: empty()
// Desc:
//-----
void CArmyWarServer::CalculateBulletVelocity(command_t *command)
{
    command->bullet.shot = true;

    if(command->heading == NORTH)
    {
        command->bullet.vel.x = 0.0f;
        command->bullet.vel.y = -200.0f;
    }
    if(command->heading == SOUTH)
    {
        command->bullet.vel.x = 0.0f;
        command->bullet.vel.y = 200.0f;
    }
    if(command->heading == EAST)
    {
        command->bullet.vel.x = 200.0f;
        command->bullet.vel.y = 0.0f;
    }
    if(command->heading == WEST)
    {
```

```

        command->bullet.vel.x = -200.0f;
        command->bullet.vel.y = 0.0f;
    }

    if(command->heading == NORTHEAST)
    {
        command->bullet.vel.x = 200.0f;
        command->bullet.vel.y = -200.0f;
    }
    if(command->heading == NORTHWEST)
    {
        command->bullet.vel.x = -200.0f;
        command->bullet.vel.y = -200.0f;
    }
    if(command->heading == SOUTHEAST)
    {
        command->bullet.vel.x = 200.0f;
        command->bullet.vel.y = 200.0f;
    }
    if(command->heading == SOUTHWEST)
    {
        command->bullet.vel.x = -200.0f;
        command->bullet.vel.y = 200.0f;
    }
}

//-----
// Name: empty()
// Desc:
//-----
void CArmyWarServer::MovePlayer(clientData *client)
{
    float clientFrametime;

    float multiplier = 100.0f;

    clientFrametime = client->command.msec / 1000.0f;;

    CalculateVelocity(&client->command, clientFrametime);
    CalculateHeading(&client->command);

    // Move the client based on the commands
    client->command.origin.x += client->command.vel.x;
    client->command.origin.y += client->command.vel.y;

    // Bullet
    if(client->command.bullet.shot == false)
    {
        client->command.bullet.origin.x = client->command.origin.x;
        client->command.bullet.origin.y = client->command.origin.y;
    }
    else
    {

```



```

client->command.bullet.lifetime += (int) (clientFrametime *
                                         1000.0f);

if(client->command.bullet.lifetime > 2000)
{
    client->command.bullet.shot = false;
    client->command.bullet.lifetime = 0;

    client->command.bullet.origin.x = client->command.origin.x;
    client->command.bullet.origin.y = client->command.origin.y;
}
}

if(client->command.key & KEY_WEAPON && client->command.bullet.shot ==
false)
{
    CalculateBulletVelocity(&client->command);
}

if(client->command.bullet.shot)
{
    client->command.bullet.origin.x += client->command.bullet.vel.x
    * clientFrametime;
    client->command.bullet.origin.y += client->command.bullet.vel.y
    * clientFrametime;
}

// Check for bullet hits
if(client->command.bullet.shot)
{
    for(clientData *client2 = clientList; client2 != NULL; client2 =
client2->next)
    {
        if(client == client2)
            continue;

        client2->diedThisFrame = false;

        VECTOR2D pos = client2->command.origin;
        pos.x += 16.0f;
        pos.y += 16.0f;

        VECTOR2D vec = VectorSubtract(&client->
command.bullet.origin, &pos);
        float distance = VectorLength(&vec);

        if(distance < 16.0f)
        {
            // Player dies
            client2->command.origin.x = client2->startPos.x;
            client2->command.origin.y = client2->startPos.y;
            client2->diedThisFrame = true;

            if(client2 == playerWithFlag)

```

```

        {
            playerWithFlag = NULL;
            updateFlag = true;
        }

        updateKill = true;

        client->command.bullet.shot = false;

        break;
    }
}

int f = client->netClient->GetIncomingSequence() &
(COMMAND_HISTORY_SIZE-1);
client->processedFrame = f;
}

//-----
// Name: empty()
// Desc:
//-----
void CArmyWarServer::CheckFlagCollisions(void)
{
    if(playerWithFlag != NULL)
    {
        // Move the flag with the player
        flagX = playerWithFlag->command.origin.x;
        flagY = playerWithFlag->command.origin.y;

        // Check if the player is at home base
        if((playerWithFlag) && (playerWithFlag->team == BLUE_TEAM))
        {
            if(playerWithFlag->command.origin.x+16 > (49*32) &&
                playerWithFlag->command.origin.x+16 < (50*32) &&
                playerWithFlag->command.origin.y+16 > (3*32) &&
                playerWithFlag->command.origin.y+16 < (4*32))
            {
                flagX = 49*32;
                flagY = 49*32;
                playerWithFlag = NULL;
                blueScore++;

                updateFlag = true;
            }
        }
        if((playerWithFlag) && (playerWithFlag->team == RED_TEAM))
        {
            if(playerWithFlag->command.origin.x+16 > (49*32) &&
                playerWithFlag->command.origin.x+16 < (50*32) &&
                playerWithFlag->command.origin.y+16 > (97*32) &&
                playerWithFlag->command.origin.y+16 < (98*32))
            {

```

```

        flagX = 49*32;
        flagY = 49*32;
        playerWithFlag = NULL;
        redScore++;

        updateFlag = true;
    }
}
else
{
    // Check if anyone is in contact with the flag
    clientData *list = clientList;

    for(; list != NULL; list = list->next)
    {
        if(list->command.origin.x+16 > flagX && list->
            command.origin.x+16 < flagX+32 &&
            list->command.origin.y+16 > flagY && list->
            command.origin.y+16 < flagY+32)
        {
            char team[10];

            if(list->team == RED_TEAM)
                strcpy(team, "RED team");
            else
                strcpy(team, "BLUE team");

            LogString("FLAG hit : player %d: %s, %s", list->
                netClient->GetIndex(), list->netClient->
                GetName(), team);

            playerWithFlag = list;

            updateFlag = true;
            return;
        }
    }
}

//-----
// Name: empty()
// Desc:
//-----
void CArmyWarServer::Frame(int msec)
{
    realtime += msec;
    frametime = msec / 1000.0f;

    // Read packets from clients
    ReadPackets();

    if(inProgress == false)

```

```

        return;

    // Check if someone hit the flag
    CheckFlagCollisions();

    // Wait full 100 ms before allowing to send
    if(realtime < servertime)
    {
        // never let the time get too far off
        if(servertime - realtime > 100)
        {
            realtime = servertime - 100;
        }

        return;
    }

    // Bump frame number, and calculate new servertime
    framenum++;
    servertime = framenum * 100;

    if(servertime < realtime)
        realtime = servertime;

    SendCommand();

    // Reset update flags
    updateFlag = false;
    updateKill = false;
}

```

GenerateRandomMap Function

Here we generate the map by randomizing the tree positions. This is slightly controlled by setting the game index number as the random seed. The original idea was to randomize exactly the same map on each client by using the same random seed, but it seems that the Unix and Windows random seeds are not always compatible. So the map data must be sent over the network to the clients (shown in the `ReadPackets` function).

```

void CArmyWarServer::GenerateRandomMap(void)
{
    // Make all land passable
    for(int i = 0; i < 100; i++)
    {
        for(int j = 0; j < 100; j++)
        {
            mapdata[i][j] = false;
        }
    }

    // Use the game's index number for random seed

```

```

    srand(index);

    // Place some random trees (avoiding the players' start locations)
    for(int m = 0; m < 300; m++)
    {
        mapdata[rand()%100][((rand()%80)+10)] = true;
    }

    // Set the flag position
    flagX = 49*32;
    flagY = 49*32;

    playerWithFlag = NULL;
}

```

CalculateVelocity Function

This function calculates the velocity on a given command and frame time.

```

void CAmyWarServer::CalculateVelocity(command_t *command, float frametime)
{
    int checkX;
    int checkY;

    float multiplier = 100.0f;

    command->vel.x = 0.0f;
    command->vel.y = 0.0f;

    if(command->key & KEY_UP)
    {
        checkX = (int) (command->origin.x / 32.0f);
        checkY = (int) ((command->origin.y - multiplier * frametime) /
                        32.0f);

        if(mapdata[checkX][checkY] == false)
            command->vel.y += -multiplier * frametime;
    }

    if(command->key & KEY_DOWN)
    {
        checkX = (int) (command->origin.x / 32.0f);
        checkY = (int) ((command->origin.y + multiplier * frametime) /
                        32.0f);

        if(mapdata[checkX][checkY] == false)
            command->vel.y += multiplier * frametime;
    }

    if(command->key & KEY_LEFT)
    {
        checkX = (int) ((command->origin.x - multiplier * frametime) /
                        32.0f);
    }
}

```

```

        checkY = (int) (command->origin.y / 32.0f);

        if(mapdata[checkX][checkY] == false)
            command->vel.x += -multiplier * frametime;
    }

    if(command->key & KEY_RIGHT)
    {
        checkX = (int) ((command->origin.x + multiplier * frametime) /
            32.0f);
        checkY = (int) (command->origin.y / 32.0f);

        if(mapdata[checkX][checkY] == false)
            command->vel.x += multiplier * frametime;
    }
}

```

The function takes two parameters (`command_t *command` and `float frametime`). The first one is a pointer to the command to use, and the second one is the frame time value. The velocity is calculated based on the command and frame time, but we also check if we are colliding with a tree.

We first calculate the new position to which the object would move after the command commences. If a tree exists in that position, we do nothing and the object does not move (it collides with the tree). If there is no tree, the object moves normally.

```

checkX = (int) (command->origin.x / 32.0f);
checkY = (int) ((command->origin.y - multiplier * frametime) / 32.0f);

if(mapdata[checkX][checkY] == false)
    command->vel.y += -multiplier * frametime;

```

CalculateHeading Function

This function is usually run after `CalculateVelocity`, as we need the velocity values here to calculate the heading for the given command.

```

void CAmyWarServer::CalculateHeading(command_t *command)
{
    // Right
    if((command->vel.x > 0.0f) &&
        (command->vel.y == 0.0f))
    {
        command->heading = EAST;
    }

    // Left
    if((command->vel.x < 0.0f) &&
        (command->vel.y == 0.0f))
    {
        command->heading = WEST;
    }
}

```

```
    }

    // Down
    if((command->vel.y > 0.0f) &&
        (command->vel.x == 0.0f))
    {
        command->heading = SOUTH;
    }

    // Up
    if((command->vel.y < 0.0f) &&
        (command->vel.x == 0.0f))
    {
        command->heading = NORTH;
    }

    // Down-Right
    if((command->vel.x > 0.0f) &&
        (command->vel.y > 0.0f))
    {
        command->heading = SOUTHEAST;
    }

    // Up-Right
    if((command->vel.x > 0.0f) &&
        (command->vel.y < 0.0f))
    {
        command->heading = NORTHEAST;
    }

    // Down-Left
    if((command->vel.x < 0.0f) &&
        (command->vel.y > 0.0f))
    {
        command->heading = SOUTHWEST;
    }

    // Up-Left
    if((command->vel.x < 0.0f) &&
        (command->vel.y < 0.0f))
    {
        command->heading = NORTHWEST;
    }
}
```

The function takes one parameter (`command_t *command`). It is a pointer to the command to use. The function simply looks at the velocity values and determines the heading of the object.

CalculateBulletVelocity Function

This function calculates the velocity for the bullet on a given command. This function is run only when firing the bullet, so the velocity here

sets the initial heading of the bullet. The velocity is not scaled with frame time because this function is not called each frame. Frame time scaling is done when moving the bullet.

```
void CArmyWarServer::CalculateBulletVelocity(command_t *command)
{
    command->bullet.shot = true;

    if(command->heading == NORTH)
    {
        command->bullet.vel.x = 0.0f;
        command->bullet.vel.y = -200.0f;
    }
    if(command->heading == SOUTH)
    {
        command->bullet.vel.x = 0.0f;
        command->bullet.vel.y = 200.0f;
    }
    if(command->heading == EAST)
    {
        command->bullet.vel.x = 200.0f;
        command->bullet.vel.y = 0.0f;
    }
    if(command->heading == WEST)
    {
        command->bullet.vel.x = -200.0f;
        command->bullet.vel.y = 0.0f;
    }
    if(command->heading == NORTHEAST)
    {
        command->bullet.vel.x = 200.0f;
        command->bullet.vel.y = -200.0f;
    }
    if(command->heading == NORTHWEST)
    {
        command->bullet.vel.x = -200.0f;
        command->bullet.vel.y = -200.0f;
    }
    if(command->heading == SOUTHEAST)
    {
        command->bullet.vel.x = 200.0f;
        command->bullet.vel.y = 200.0f;
    }
    if(command->heading == SOUTHWEST)
    {
        command->bullet.vel.x = -200.0f;
        command->bullet.vel.y = 200.0f;
    }
}
```

The function takes one parameter (`command_t *command`) that is the pointer to the command to use. This function needs to run after the

player's heading has been calculated, because the bullet's heading will be the same.

MovePlayer Function

This function does the actual moving of the players and bullets. Note that the formulas we use to calculate the velocities and positions must match the ones used on the client side. This function is run every time we receive a "frame" message from a client (a client sends one on every frame, so we end up running the same amount of frames as the client).

```
void CAmyWarServer::MovePlayer(clientData *client)
{
    float clientFrametime;

    float multiplier = 100.0f;

    clientFrametime = client->command.msec / 1000.0f;

    CalculateVelocity(&client->command, clientFrametime);
    CalculateHeading(&client->command);

    // Move the client based on the commands
    client->command.origin.x += client->command.vel.x;
    client->command.origin.y += client->command.vel.y;

    // Bullet
    if(client->command.bullet.shot == false)
    {
        client->command.bullet.origin.x = client->command.origin.x;
        client->command.bullet.origin.y = client->command.origin.y;
    }
    else
    {
        client->command.bullet.lifetime += (int) (clientFrametime *
                                                1000.0f);

        if(client->command.bullet.lifetime > 2000)
        {
            client->command.bullet.shot = false;
            client->command.bullet.lifetime = 0;

            client->command.bullet.origin.x = client->command.origin.x;
            client->command.bullet.origin.y = client->command.origin.y;
        }
    }

    if(client->command.key & KEY_WEAPON && client->command.bullet.shot ==
        false)
    {
```

```
        CalculateBulletVelocity(&client->command);
    }

    if(client->command.bullet.shot)
    {
        client->command.bullet.origin.x += client->command.bullet.vel.x
            * clientFrametime;
        client->command.bullet.origin.y += client->command.bullet.vel.y
            * clientFrametime;
    }

    // Check for bullet hits
    if(client->command.bullet.shot)
    {
        for(clientData *client2 = clientList; client2 != NULL; client2
            = client2->next)
        {
            if(client == client2)
                continue;

            client2->diedThisFrame = false;

            VECTOR2D pos = client2->command.origin;
            pos.x += 16.0f;
            pos.y += 16.0f;

            VECTOR2D vec = VectorSubtract(&client->
                command.bullet.origin, &pos);
            float distance = VectorLength(&vec);

            if(distance < 16.0f)
            {
                // Player dies
                client2->command.origin.x = client2->startPos.x;
                client2->command.origin.y = client2->startPos.y;
                client2->diedThisFrame = true;

                if(client2 == playerWithFlag)
                {
                    playerWithFlag = NULL;
                    updateFlag = true;
                }

                updateKill = true;

                client->command.bullet.shot = false;

                break;
            }
        }
    }

    int f = client->netClient->GetIncomingSequence() &
        (COMMAND_HISTORY_SIZE-1);
```

```

        client->processedFrame = f;
    }

```

The function takes one parameter (`clientData *client`). This is a pointer to the client to move. We use the client's own frame time, because we must run exactly the same frames as the client. The frame time is given to us in the packet the client sent us.

```

clientFrametime = client->command.msec / 1000.0f;

```

The bullet lives for 2 seconds before it is removed. The following piece of code shows how we calculate the lifetime. We simply add the frame time to the bullet's lifetime and then check if 2 seconds have passed.

```

client->command.bullet.lifetime += (int) (clientFrametime * 1000.0f);

if(client->command.bullet.lifetime > 2000)
{
    client->command.bullet.shot = false;
    client->command.bullet.lifetime = 0;

    client->command.bullet.origin.x = client->command.origin.x;
    client->command.bullet.origin.y = client->command.origin.y;
}

```

The following code shows how to check for bullet hits. First of all, the bullet must be shot in order to do that. Then we loop through all the players except the one who owns the bullet and check if the bullet position is within 16 pixels of a player's position. If it is, the bullet hit and killed that player. If that player was carrying the flag, it is dropped. The flags are set up to indicate that a player died and we should update the clients, and more precisely that this player died.

```

// Check for bullet hits
if(client->command.bullet.shot)
{
    for(clientData *client2 = clientList; client2 != NULL; client2 =
        client2->next)
    {
        if(client == client2)
            continue;

        client2->diedThisFrame = false;

        VECTOR2D pos = client2->command.origin;
        pos.x += 16.0f;
        pos.y += 16.0f;

        VECTOR2D vec = VectorSubtract(&client->command.bullet.origin,
            &pos);
        float distance = VectorLength(&vec);
    }
}

```

```

        if(distance < 16.0f)
        {
            // Player dies
            client2->command.origin.x = client2->startPos.x;
            client2->command.origin.y = client2->startPos.y;
            client2->diedThisFrame = true;

            if(client2 == playerWithFlag)
            {
                playerWithFlag = NULL;
                updateFlag = true;
            }

            updateKill = true;

            client->command.bullet.shot = false;

            break;
        }
    }
}

```

The last thing we do in this function is set the `processedFrame` variable for the client. This is a history array indexed value, and it is sent to the client for comparison reasons. (The client compares this frame to the one it processed itself, and this number is the index number that identifies the frame.)

```

int f = client->netClient->GetIncomingSequence() & (COMMAND_HISTORY_SIZE-1);
client->processedFrame = f;

```

CheckFlagCollisions Function

This function checks for possible flag collisions, both if a player picked up the flag and if a player carried the flag to the target area.

```

void CArmyWarServer::CheckFlagCollisions(void)
{
    if(playerWithFlag != NULL)
    {
        // Move the flag with the player
        flagX = playerWithFlag->command.origin.x;
        flagY = playerWithFlag->command.origin.y;

        // Check if the player is at home base
        if((playerWithFlag) && (playerWithFlag->team == BLUE_TEAM))
        {
            if(playerWithFlag->command.origin.x+16 > (49*32) &&
                playerWithFlag->command.origin.x+16 < (50*32) &&
                playerWithFlag->command.origin.y+16 > (3*32) &&
                playerWithFlag->command.origin.y+16 < (4*32))
            {
                flagX = 49*32;
                flagY = 49*32;
            }
        }
    }
}

```


Frame Function

This function runs the frame on the server. It reads the packets from the clients, moves the players if they send us something, and sends the commands to all the clients. Commands are sent in 100-millisecond intervals to save required bandwidth. Dead reckoning will take care of the clients moving even when they do not receive a packet from the server.

```
void CAmyWarServer::Frame(int msec)
{
    realtime += msec;
    frametime = msec / 1000.0f;

    // Read packets from clients
    ReadPackets();

    if(inProgress == false)
        return;

    // Check if someone hit the flag
    CheckFlagCollisions();

    // Wait full 100 ms before allowing to send
    if(realtime < servertime)
    {
        // never let the time get too far off
        if(servertime - realtime > 100)
        {
            realtime = servertime - 100;
        }

        return;
    }

    // Bump frame number, and calculate new servertime
    framenum++;
    servertime = framenum * 100;

    if(servertime < realtime)
        realtime = servertime;

    SendCommand();

    // Reset update flags
    updateFlag = false;
    updateKill = false;
}
```

This function takes one parameter (`int msec`), the frame time of the server. Packets are read every frame, but they are not sent every frame.

Here we check if enough time has passed to send packets. If not, the function returns and does not reach the sending function.

```
// Wait full 100 ms before allowing to send
if(realtime < servertime)
{
    // Never let the time get too far off
    if(servertime - realtime > 100)
    {
        realtime = servertime - 100;
    }

    return;
}
```

A new server time is calculated when the old one is reached:

```
// Bump frame number, and calculate new servertime
framenum++;
servertime = framenum * 100;
```

Commands are sent to each client:

```
SendCommand();
```

lobby.cpp File

Now that we have the game data structures, we can implement the functions we introduced in Tutorial 4.

AddGame Function

This function adds a game to the server's game list. If the list does not exist yet, it is created. This means that no games exist yet, so the game we are adding is the first one. Otherwise, we just add a new game to the list.

The game's information is filled in (name, index number, etc.). Then the function tries to open a game server for the new game, using the `InitNetwork` function. Each game uses its own port number, which is based on the game's index number.

Finally, the game's counter is increased.

```
void CLobbyServer::AddGame(char *name)
{
    // First get a pointer to the beginning of client list
    CArmyWarServer *list = gameList;
    CArmyWarServer *prev;

    // No clients yet, adding the first one
    if(gameList == NULL)
    {
```

```
        gameList = new CArmyWarServer;

        gameList->SetName(name);
        gameList->next = NULL;
        gameList->SetIndex(gameAmount);
        gameList->GenerateRandomMap();

        if(gameList->InitNetwork(GetGameAmount()) != 0)
        {
            LogString("Could not create game server");
        }
    }
    else
    {
        prev = list;
        list = gameList->next;

        while(list != NULL)
        {
            prev = list;
            list = list->next;
        }

        list = new CArmyWarServer;

        list->SetName(name);
        list->next = NULL;
        list->SetIndex(gameAmount);
        list->GenerateRandomMap();

        if(list->InitNetwork(GetGameAmount()) != 0)
        {
            LogString("Could not create game server");
        }

        prev->next = list;
    }

    gameAmount++;
}
```

RemoveGame Function

This function removes the selected game from the server's game list. First, the correct game is looked up by going through the game list and comparing the names. Once the game is found, the game's network is shut down and the game list is updated. If the game is the last in the list, the list is marked empty by pointing it to NULL (or as you can see in the code, by pointing it to the next game in the list, which is NULL).

The game counter is then decreased.


```
void CLobbyServer::RemoveGame(char *name)
{
    CArmyWarServer *list = gameList;
    CArmyWarServer *prev = NULL;
    CArmyWarServer *next = NULL;

    for( ; list != NULL; list = list->next)
    {
        if(strcmp(name, list->GetName()) == 0)
        {
            if(prev != NULL)
            {
                prev->next = list->next;
            }

            break;
        }

        prev = list;
    }

    if(list == gameList)
    {
        if(list)
        {
            list->ShutdownNetwork();

            next = list->next;
            delete list;
        }

        list = NULL;
        gameList = next;
    }
    else
    {
        if(list)
        {
            list->ShutdownNetwork();

            next = list->next;
            delete list;
        }

        list = next;
    }

    gameAmount--;
}
```

RemoveGames Function

This function simply removes all the games in the game list.

```
void CLobbyServer::RemoveGames(void)
{
    CArmyWarServer *list = gameList;
    CArmyWarServer *next;

    while(list != NULL)
    {
        if(list)
        {
            list->ShutdownNetwork();

            next = list->next;
            delete list;
        }

        list = next;
    }

    gameList = NULL;
    gameAmount = 0;
}
```

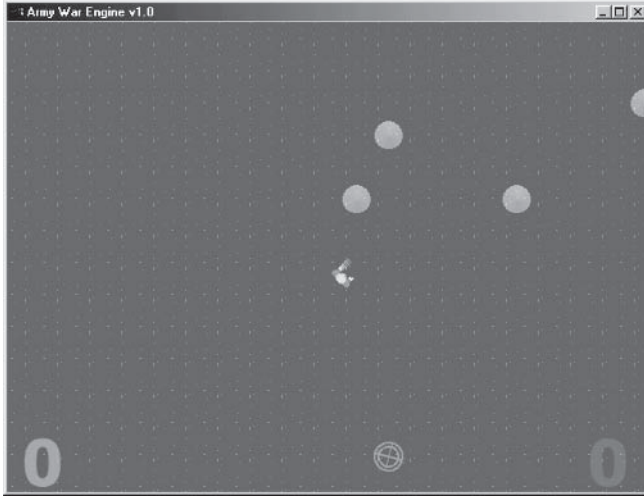
Summary of Server Code

That is it for the server-side code. We now have a working server for our game that receives data and moves the players, and sends only in 100-millisecond intervals to save bandwidth. Now all we need is a client connecting to the server.

Game Client Code

Now to the final part of this tutorial, the client-side code. We have some new source code files: `client.cpp`, `network.cpp`, and `client.h`. Also some of the old client-side files have changed a little bit.

Figure 2



client.h File

Like on the server side (`server.h`), this header file contains the application-specific data structures. This is where the player data structures are, along with the game's main class and some definitions. The main class has the network interface methods that we learned to create in the previous tutorials (like `ReadPackets()` and so on).

```
#ifndef CLIENT_H
#define CLIENT_H

#include <gl/gl.h>
#include <gl/glu.h>
#include <gl/glaux.h>
#include <2dlib.h>

#include "network.h"

#define NORTH      0
#define NORTHEAST 45
#define EAST       90
#define SOUTHEAST 135
#define SOUTH      180
#define SOUTHWEST 225
#define WEST       270
#define NORTHWEST 315

#define BLUE_TEAM 0
#define RED_TEAM  1

typedef struct
{
```

```

        float x;
        float y;
    } VECTOR2D;

typedef struct bullet_t
{
    VECTOR2D    vel;
    VECTOR2D    origin;
    VECTOR2D    predictedOrigin;

    bool        shot;
    int         lifetime;
} bullet_t;

typedef struct
{
    int         key;
    int         heading;

    VECTOR2D    vel;
    VECTOR2D    origin;
    VECTOR2D    predictedOrigin;

    bullet_t    bullet;

    int         msec;
} command_t;

typedef struct clientData
{
    command_t    frame[COMMAND_HISTORY_SIZE]; // frame history
    command_t    serverFrame;                // the latest frame from
                                              // server
    command_t    command;                    // current frame's commands

    int         index;

    int         processedFrame;

    VECTOR2D    startPos;
    bool        team;
    char        nickname[30];
    char        password[30];

    clientData  *next;
} clientData;

// The main application class interface
class CAmyWar
{
private:
    // Methods

    // Client.cpp

```

```

void InitializeEngine(void);
void DrawMap(void);
void CheckVictory(void);
void KillPlayer(int index);
clientData *GetClientPointer(int index);

void CheckPredictionError(int a);
void CheckBulletPredictionError(int a);
void CalculateVelocity(command_t *command, float frametime);
void CalculateHeading(command_t *command);
void CalculateBulletVelocity(command_t *command);
void PredictMovement(int prevFrame, int curFrame);
void MoveObjects(void);

void AddClient(int local, int index, char *name);
void RemoveClient(int index);
void RemoveClients(void);

// Network.cpp
void ReadPackets(void);
void SendCommand(void);
void SendRequestNonDeltaFrame(void);
void ReadMoveCommand(dreamMessage *mes, clientData *client);
void ReadDeltaMoveCommand(dreamMessage *mes, clientData *client);
void BuildDeltaMoveCommand(dreamMessage *mes, clientData *theClient);

// Variables

// Network variables
dreamClient *networkClient;

clientData *clientList;           // Client list
clientData *localClient;         // Pointer to the local client in the
                                // client list
int clients;

clientData inputClient;          // Handles all keyboard input

// Graphic declarations
GFX_IMAGE2D grass;               // variable to hold graphic
GFX_IMAGE2D redman;              // variable to hold graphic
GFX_IMAGE2D blueman;             // variable to hold graphic
GFX_IMAGE2D tree;                // variable to hold graphic
GFX_IMAGE2D redtarget;           // variable to hold graphic
GFX_IMAGE2D bluetarget;          // variable to hold graphic
GFX_IMAGE2D flag;                // variable to hold graphic
GFX_IMAGE2D rednumbers[10];      // variable to hold graphic
GFX_IMAGE2D bluenumbers[10];    // variable to hold graphic

float frametime;

char gamename[32];
bool inProgress;

```

```
bool init;

// Tile scroll positions
int scrollX;
int scrollY;
int tileScrollX;
int tileScrollY;

bool mapdata[100][100];
int gameIndex;

float targetRotation;           // variable to rotate the target images

int redScore;
int blueScore;

float flagX;
float flagY;
clientData *playerWithFlag;

public:
    CArmyWar();
    ~CArmyWar();

// Client.cpp
void Shutdown(void);
void CheckKeys(void);
void Frame(void);
void RunNetwork(int msec);

// Network.cpp
void StartConnection(int ind);
void Connect(void);
void Disconnect(void);
void SendStartGame(void);

void SetName(char *n)           {strcpy(gamename, n);}
char *GetName(void)             {return gamename;}

void SetGameIndex(int index)    {gameIndex = index;}
int GetGameIndex(void)          {return gameIndex;}

clientData *GetClientList(void) {return clientList;}

void SetInProgress(bool p)      {inProgress = p;}
bool GetInProgress(void)        {return inProgress;}

    CArmyWar *next;
};

#endif
```

The command data structures are the same as on the server side, so we will skip that part now. The client data structure is slightly different, but is very easy to understand.

The directions in which a player can move are given in the following definitions:

```
#define NORTH      0
#define NORTHEAST 45
#define EAST       90
#define SOUTHEAST 135
#define SOUTH      180
#define SOUTHWEST 225
#define WEST       270
#define NORTHWEST 315
```

The CArmyWar class has some variables that need explaining, so here goes.

```
// Network variables
dreamClient *networkClient;

clientData *clientList;           // Client list
clientData *localClient;         // Pointer to the local client in the
                                  // client list

int clients;

clientData inputClient;          // Handles all keyboard input

// Graphic declarations
GFX_IMAGE2D grass;               // variable to hold graphic
GFX_IMAGE2D redman;              // variable to hold graphic
GFX_IMAGE2D blueman;             // variable to hold graphic
GFX_IMAGE2D tree;                // variable to hold graphic
GFX_IMAGE2D redtarget;           // variable to hold graphic
GFX_IMAGE2D bluetarget;          // variable to hold graphic
GFX_IMAGE2D flag;                // variable to hold graphic
GFX_IMAGE2D rednumbers[10];      // variable to hold graphic
GFX_IMAGE2D bluenumbers[10];     // variable to hold graphic

float frametime;

char gamename[32];
bool inProgress;
bool init;

// Tile scroll positions
int scrollX;
int scrollY;
int tileScrollX;
int tileScrollY;

bool mapdata[100][100];
int gameIndex;
```

```
float targetRotation;           // variable to rotate the target images

int redScore;
int blueScore;

float flagX;
float flagY;
clientData *playerWithFlag;
```

The `dreamClient networkClient` is the `dreamClient` network client object. The `clientData *clientList` is the list of all the clients connected on the same server as we are. The `clientData *localClient` is the local client on that list. The `clientData inputClient` is used to handle the keyboard input.

The ints `scrollX` and `scrollY` are used to store the amount the screen has scrolled (in pixels). The ints `tileScrollX` and `tileScrollY` are used to store the amount the screen has scrolled (in tiles). A tile is 32 pixels in height and width.

The `bool mapdata[100][100]` stores the map data. A 0 means plain grass and a 1 means a tree. The floats `flagX` and `flagY` store the flags' coordinates.

The `clientData *playerWithFlag` is a pointer to the player who is carrying the flag. If no one is carrying the flag, this pointer is `NULL`.

network.h File

The `network.h` file on the client side now looks like this:

```
#ifndef NETWORK_H
#define NETWORK_H

#define COMMAND_HISTORY_SIZE 64

#define KEY_UP 1
#define KEY_DOWN 2
#define KEY_LEFT 4
#define KEY_RIGHT 8
#define KEY_WEAPON 16

#define CMD_KEY 1
#define CMD_HEADING 2
#define CMD_ORIGIN 4
#define CMD_BULLET 8
#define CMD_FLAG 16
#define CMD_KILL 32

#define USER_MES_FRAME 1
#define USER_MES_NONDELTAFRAME 2
#define USER_MES_SERVEREXIT 3
#define USER_MES_LOGIN 4
#define USER_MES_SIGNIN 5
```



```

#define USER_MES_CHAT          6
#define USER_MES_CREATEGAME    7
#define USER_MES_REMOVEGAME    8
#define USER_MES_GAMEDATA      9
#define USER_MES_STARTGAME     10
#define USER_MES_MAPDATA       11
#define USER_MES_KEEPLIVE      12

typedef struct clientLoginData
{
    int          index;
    char         nickname[30];
    clientLoginData *next;
} clientLoginData;

#endif

```

common.h File

This header file is just a header file wrapper. So you only need to include this header, and all the required header files are provided for you in the correct order.

```

#ifndef __COMMON_H__
#define __COMMON_H__

#include "dreamSock.h"

#include "client.h"
#include "network.h"
#include "lobby.h"
#include "signin.h"
#include "main.h"

#endif

```

main.cpp File

Our game's base initialization happens in this file, which includes some new functions. All the windows are created and handled here. This file does not include game logic.

VectorLength and VectorSubtract Functions

These functions are helper functions to do some vector calculations. The first one returns the length of a vector, and the second one subtracts one vector from another.

```

float VectorLength(VECTOR2D *vec)
{
    return (float) sqrt(vec->x*vec->x + vec->y*vec->y);
}

```

```

}

VECTOR2D VectorSubtract(VECTOR2D *vec1, VECTOR2D *vec2)
{
    VECTOR2D vec;

    vec.x = vec1->x - vec2->x;
    vec.y = vec1->y - vec2->y;

    return vec;
}

```

ApplicationProc Function

This function now handles keyboard input and maintains the Join Game button's state.

```

case WM_KEYDOWN:
{
    keys[wParam] = TRUE;
    break;
}

case WM_KEYUP:
{
    keys[wParam] = FALSE;
    break;
}

default:
    if(!Lobby.GetGameAmount())
    {
        EnableWindow(GetDlgItem(hWnd_LobbyDialog, IDC_JOINGAME), FALSE);
    }
    break;
}

```

Dialog Procedures

The dialog procedure functions have also changed, as we now actually send the data that goes with the dialogs. They are very simple additions, so they are not listed here.

Main Loop

The game's main loop is almost the same as in the earlier tutorials, but this time it also handles network and keyboard input/output for the game itself. The new part can be seen here:

```

// If we have a local game, run the frames for it
if(Lobby.GetLocalGame() != NULL)
{
    Lobby.GetLocalGame()->RunNetwork(time);
    Lobby.GetLocalGame()->CheckKeys();
}

```

```

        Lobby.GetLocalGame()->Frame();
    }

```

network.cpp File

This file contains the network-only code for the client. The connection can be started with these methods and data can be sent and received. Also, the client list is updated with these functions. There are many functions that are similar to the ones on the server side, so those functions will not be explained here.

```

#include "common.h"

//-----
// Name: empty()
// Desc:
//-----
void CArmWar::StartConnection(int ind)
{
    LogString("StartConnection %d", ind);

    gameIndex = ind;

    int ret = networkClient->Initialize("", serverIP, 30004 + gameIndex);

    if(ret == DREAMSOCK_CLIENT_ERROR)
    {
        char text[64];
        sprintf(text, "Could not open client socket");

        MessageBox(NULL, text, "Error", MB_OK);
    }

    Connect();
}

//-----
// Name: empty()
// Desc:
//-----
void CArmWar::ReadPackets(void)
{
    char data[1400];
    struct sockaddr address;

    clientData *cIList;

    int type;
    int ind;
    int local;
    int ret;
}

```

```
char name[50];

dreamMessage mes;
mes.Init(data, sizeof(data));

while(ret = networkClient->GetPacket(mes.data, &address))
{
    mes.SetSize(ret);
    mes.BeginReading();

    type = mes.ReadByte();

    switch(type)
    {
    case DREAMSOCK_MES_ADDCLIENT:
        local = mes.ReadByte();
        ind = mes.ReadByte();
        strcpy(name, mes.ReadString());

        AddClient(local, ind, name);
        break;

    case DREAMSOCK_MES_REMOVECLIENT:
        ind = mes.ReadByte();

        LogString("Got removeclient %d message", ind);

        RemoveClient(ind);

        if(clientList == NULL)
        {
            LogString("clientList == NULL, sending remove game
                %s", gamename);
            Lobby.SendRemoveGame(gamename);
        }
        break;

    case USER_MES_FRAME:
        // Skip sequences
        mes.ReadShort();
        mes.ReadShort();

        for(cList = clientList; cList != NULL; cList =
            cList->next)
        {
            LogString("Reading DELTAFRAME for client %d",
                cList->index);
            ReadDeltaMoveCommand(&mes, cList);
        }

        break;

    case USER_MES_NONDELTAFRAME:
        // Skip sequences
```

```

        mes.ReadShort();
        mes.ReadShort();

        cList = clientList;

        for(cList = clientList; cList != NULL; cList =
            cList->next)
        {
            LogString("Reading NONDELTAFRAME for client %d",
                cList->index);
            ReadMoveCommand(&mes, cList);
        }

        break;

case USER_MES_SERVEREXIT:
    MessageBox(NULL, "Server disconnected", "Info", MB_OK);
    Disconnect();
    break;

case USER_MES_STARTGAME:
    // Skip sequences
    mes.ReadShort();
    mes.ReadShort();

    DestroyWindow(hWnd_JoinGameDialog);

    InitializeEngine();

    break;

case USER_MES_MAPDATA:
    // Skip sequences
    mes.ReadShort();
    mes.ReadShort();

    for(int m = 0; m < 300; m++)
    {
        int i = mes.ReadByte();
        int j = mes.ReadByte();

        mapdata[i][j] = true;
    }

    break;
    }
}

//-----
// Name: empty()
// Desc:
//-----

```

```
void CAmyWar::AddClient(int local, int ind, char *name)
{
    // First get a pointer to the beginning of the client list
    clientData *list = clientList;
    clientData *prev;

    LogString("App: Client: Adding client with index %d", ind);

    // No clients yet, adding the first one
    if(clientList == NULL)
    {
        LogString("App: Client: Adding first client");

        clientList = (clientData *) calloc(1, sizeof(clientData));

        if(local)
        {
            LogString("App: Client: This one is local");
            localClient = clientList;
        }

        clientList->index = ind;
        strcpy(clientList->nickname, name);

        if(clients % 2 == 0)
            clientList->team = RED_TEAM;
        else
            clientList->team = BLUE_TEAM;

        clientList->next = NULL;
    }
    else
    {
        LogString("App: Client: Adding another client");

        prev = list;
        list = clientList->next;

        while(list != NULL)
        {
            prev = list;
            list = list->next;
        }

        list = (clientData *) calloc(1, sizeof(clientData));

        if(local)
        {
            LogString("App: Client: This one is local");
            localClient = list;
        }

        list->index = ind;
```

```

        strcpy(list->nickname, name);

        if(clients % 2 == 0)
            list->team = RED_TEAM;
        else
            list->team = BLUE_TEAM;

        list->next = NULL;
        prev->next = list;
    }

    clients++;

    // If we just joined the game, request a non-delta compressed frame
    if(local)
        SendRequestNonDeltaFrame();

    Lobby.RefreshJoinedPlayersList();
}

//-----
// Name: empty()
// Desc:
//-----
void CArmyWar::RemoveClient(int ind)
{
    clientData *list = clientList;
    clientData *prev = NULL;
    clientData *next = NULL;

    // Look for correct client and update list
    for(; list != NULL; list = list->next)
    {
        if(list->index == ind)
        {
            if(prev != NULL)
            {
                prev->next = list->next;
            }

            break;
        }

        prev = list;
    }

    // First entry
    if(list == clientList)
    {
        if(list)
        {
            next = list->next;
            free(list);
        }
    }
}

```

```

        }

        list = NULL;
        clientList = next;
    }

    // Other
    else
    {
        if(list)
        {
            next = list->next;
            free(list);
        }

        list = next;
    }

    clients--;

    Lobby.RefreshJoinedPlayersList();
}

```

```

//-----
// Name: empty()
// Desc:
//-----
void CArmyWar::RemoveClients(void)

```

```

{
    clientData *list = clientList;
    clientData *next;

    while(list != NULL)
    {
        if(list)
        {
            next = list->next;
            free(list);
        }

        list = next;
    }

    clientList = NULL;
    clients = 0;
}

```

```

//-----
// Name: empty()
// Desc:
//-----
void CArmyWar::SendCommand(void)
{
    if(networkClient->GetConnectionState() != DREAMSOCK_CONNECTED)

```



```

        return;

    dreamMessage message;
    char data[1400];

    int i = networkClient->GetOutgoingSequence() & (COMMAND_HISTORY_SIZE-1);

    message.Init(data, sizeof(data));
    message.WriteByte(USER_MES_FRAME); // type
    message.AddSequences(networkClient); // sequences

    // Build delta-compressed move command
    BuildDeltaMoveCommand(&message, &inputClient);

    // Send the packet
    networkClient->SendPacket(&message);

    // Store the command to the input client's history
    memcpy(&inputClient.frame[i], &inputClient.command, sizeof(command_t));

    clientData *cList = clientList;

    // Store the commands to the clients' history
    for(; cList != NULL; cList = cList->next)
    {
        memcpy(&cList->frame[i], &cList->command, sizeof(command_t));
    }
}

//-----
// Name: empty()
// Desc:
//-----
void CArmyWar::SendStartGame(void)
{
    char data[1400];
    dreamMessage message;
    message.Init(data, sizeof(data));

    message.WriteByte(USER_MES_STARTGAME);
    message.AddSequences(networkClient);

    networkClient->SendPacket(&message);
}

//-----
// Name: empty()
// Desc:
//-----
void CArmyWar::SendRequestNonDeltaFrame(void)
{
    char data[1400];
    dreamMessage message;

```

```

        message.Init(data, sizeof(data));

        message.WriteByte(USER_MES_NONDELTAFRAME);
        message.AddSequences(networkClient);

        networkClient->SendPacket(&message);
    }

//-----
// Name: empty()
// Desc:
//-----
void CArmyWar::Connect(void)
{
    if(init)
    {
        LogString("ArmyWar already initialized");
        return;
    }

    LogString("CArmyWar::Connect");

    init = true;

    networkClient->SendConnect(Lobby.GetLocalClient()->nickname);
}

//-----
// Name: empty()
// Desc:
//-----
void CArmyWar::Disconnect(void)
{
    if(!init)
        return;

    LogString("CArmyWar::Disconnect");

    init = false;
    localClient = NULL;
    memset(&inputClient, 0, sizeof(clientData));

    networkClient->SendDisconnect();
}

//-----
// Name: empty()
// Desc:
//-----
void CArmyWar::ReadMoveCommand(dreamMessage *mes, clientData *client)
{
    // Key
    client->serverFrame.key = mes->ReadByte();
}

```

```

// Heading
client->serverFrame.heading      = mes->ReadShort();

// Origin
client->serverFrame.origin.x     = mes->ReadFloat();
client->serverFrame.origin.y     = mes->ReadFloat();
client->serverFrame.vel.x        = mes->ReadFloat();
client->serverFrame.vel.y        = mes->ReadFloat();

client->serverFrame.bullet.origin.x = mes->ReadFloat();
client->serverFrame.bullet.origin.y = mes->ReadFloat();
client->serverFrame.bullet.vel.x    = mes->ReadFloat();
client->serverFrame.bullet.vel.y    = mes->ReadFloat();
client->serverFrame.bullet.lifetime = mes->ReadShort();
client->serverFrame.bullet.shot     = mes->ReadByte();

int playerWithFlagIndex          = mes->ReadShort();

if(playerWithFlagIndex != -1)
{
    playerWithFlag = GetClientPointer(playerWithFlagIndex);
}

flagX = mes->ReadFloat();
flagY = mes->ReadFloat();

redScore = mes->ReadByte();
blueScore = mes->ReadByte();

// Read time to run command
client->serverFrame.msec = mes->ReadByte();

memcpy(&client->command, &client->serverFrame, sizeof(command_t));

// Fill the history array with the position we got
for(int f = 0; f < COMMAND_HISTORY_SIZE; f++)
{
    client->frame[f].predictedOrigin.x = client->command.origin.x;
    client->frame[f].predictedOrigin.y = client->command.origin.y;
    client->frame[f].bullet.predictedOrigin.x = client->
        command.bullet.origin.x;
    client->frame[f].bullet.predictedOrigin.y = client->
        command.bullet.origin.y;
}
}

//-----
// Name: empty()
// Desc:
//-----
void CAmyWar::ReadDeltaMoveCommand(dreamMessage *mes, clientData *client)
{
    int processedFrame;

```

```
int flags = 0;

// Flags
flags = mes->ReadByte();

// Key
if(flags & CMD_KEY)
{
    client->serverFrame.key = mes->ReadByte();

    client->command.key = client->serverFrame.key;
    LogString("Client %d: Read key %d", client->index, client->
        command.key);
}

if(flags & CMD_ORIGIN || flags & CMD_BULLET)
{
    processedFrame = mes->ReadByte();
    client->processedFrame = processedFrame;
}

// Origin
if(flags & CMD_ORIGIN)
{
    client->serverFrame.origin.x = mes->ReadFloat();
    client->serverFrame.origin.y = mes->ReadFloat();

    client->serverFrame.vel.x = mes->ReadFloat();
    client->serverFrame.vel.y = mes->ReadFloat();

    if(client == localClient)
    {
        CheckPredictionError(processedFrame);
    }
    else
    {
        client->command.origin.x = client->serverFrame.origin.x;
        client->command.origin.y = client->serverFrame.origin.y;
    }
}

if(flags & CMD_BULLET)
{
    client->serverFrame.bullet.origin.x = mes->ReadFloat();
    client->serverFrame.bullet.origin.y = mes->ReadFloat();
    client->serverFrame.bullet.vel.x = mes->ReadFloat();
    client->serverFrame.bullet.vel.y = mes->ReadFloat();
    client->serverFrame.bullet.shot = mes->ReadByte();

    client->command.bullet.shot = client->serverFrame.bullet.shot;

    if(client == localClient)
    {
        CheckBulletPredictionError(processedFrame);
    }
}
```

```

        }
        else
        {
            client->command.bullet.origin.x = client->
                serverFrame.bullet.origin.x;
            client->command.bullet.origin.y = client->
                serverFrame.bullet.origin.y;
        }
    }

    // Flag & points
    if(flags & CMD_FLAG)
    {
        int playerWithFlagIndex = mes->ReadShort();

        if(playerWithFlagIndex != 0)
        {
            LogString("FLAG playerWithFlagIndex %d",
                playerWithFlagIndex);
            playerWithFlag = GetClientPointer(playerWithFlagIndex);
        }
        else
        {
            playerWithFlag = NULL;
        }

        flagX = mes->ReadFloat();
        flagY = mes->ReadFloat();

        redScore = mes->ReadByte();
        blueScore = mes->ReadByte();

        CheckVictory();
    }

    // Someone died
    if(flags & CMD_KILL)
    {
        int died = mes->ReadByte();

        if(died)
            KillPlayer(client->index);
    }

    // Read time to run command
    client->command.msec = mes->ReadByte();
}

//-----
// Name: empty()
// Desc:
//-----
void CArmyWar::BuildDeltaMoveCommand(dreamMessage *mes, clientData *theClient)
{

```

```

int flags = 0;
int last = (networkClient->GetOutgoingSequence() - 1) &
           (COMMAND_HISTORY_SIZE-1);

// Check what needs to be updated
if(theClient->frame[last].key != theClient->command.key)
    flags |= CMD_KEY;

// Add to the message
// Flags
mes->WriteByte(flags);

// Key
if(flags & CMD_KEY)
{
    mes->WriteByte(theClient->command.key);
}

mes->WriteByte(theClient->command.msec);
}

//-----
// Name: empty()
// Desc:
//-----
void CArmyWar::RunNetwork(int msec)
{
    static int time = 0;
    time += msec;

    // Framerate is too high
    if(time < (1000 / 60))
        return;

    frametime = time / 1000.0f;
    time = 0;

    // Read packets from server, and send new commands
    ReadPackets();
    SendCommand();

    int ack = networkClient->GetIncomingAcknowledged();
    int current = networkClient->GetOutgoingSequence();

    // Check that we haven't gone too far
    if(current - ack > COMMAND_HISTORY_SIZE)
        return;

    // Predict the frames that we are waiting on from the server
    for(int a = ack + 1; a < current; a++)
    {
        int prevframe = (a-1) & (COMMAND_HISTORY_SIZE-1);
        int frame = a & (COMMAND_HISTORY_SIZE-1);

```

```

        PredictMovement(prevframe, frame);
    }

    MoveObjects();
}

```

StartConnection Function

This function starts a specified connection. The parameter `ind` is used to specify the index number of the game to which we want to connect. Each game uses its own port, which is 30004 plus the game's index number. So the first game is run on port 30004, the second one on port 30005, and so on.

```

void CArmyWar::StartConnection(int ind)
{
    LogString("StartConnection %d", ind);

    gameIndex = ind;

    int ret = networkClient->Initialize("", serverIP, 30004 + gameIndex);

    if(ret == DREAMSOCK_CLIENT_ERROR)
    {
        char text[64];
        sprintf(text, "Could not open client socket");

        MessageBox(NULL, text, "Error", MB_OK);
    }

    Connect();
}

```

SendCommand Function

The `SendCommand` function is used to send the commands to the server. This function is run every frame, so every frame's commands are sent to the server.

```

void CArmyWar::SendCommand(void)
{
    if(networkClient->GetConnectionState() != DREAMSOCK_CONNECTED)
        return;

    dreamMessage message;
    char data[1400];

    int i = networkClient->GetOutgoingSequence() & (COMMAND_HISTORY_SIZE-1);

    message.Init(data, sizeof(data));
    message.WriteByte(USER_MES_FRAME); // type
    message.AddSequences(networkClient); // sequences
}

```

```

// Build delta-compressed move command
BuildDeltaMoveCommand(&message, &inputClient);

// Send the packet
networkClient->SendPacket(&message);

// Store the command to the input client's history
memcpy(&inputClient.frame[i], &inputClient.command, sizeof(command_t));

clientData *cList = clientList;

// Store the commands to the clients' history
for(; cList != NULL; cList = cList->next)
{
    memcpy(&cList->frame[i], &cList->command, sizeof(command_t));
}
}

```

A message is always filled on every frame and contains that frame's commands. The message is built using the `BuildDeltaMoveCommand()` function. That function will be explained later in this tutorial. When the message is built, it is ready to be sent to the server. Remember that this happens on every frame, even if nothing has changed from the previous frame. The `BuildDeltaMoveCommand()` function takes care of keeping track of any changes.

```

message.Init(data, sizeof(data));
message.WriteByte(USER_MES_FRAME); // type
message.AddSequences(networkClient); // sequences

// Build delta-compressed move command
BuildDeltaMoveCommand(&message, &inputClient);

// Send the packet
networkClient->SendPacket(&message);

```

Every frame, the message is stored into message history as seen below. An index number for the history table is calculated by looking at the outgoing sequence number of the client. Every frame, this number is one bigger than the last time. Remember that the index number starts at 0 and ends at `COMMAND_HISTORY_SIZE - 1` (`COMMAND_HISTORY_SIZE` is defined as 64 in `network.h`). After 64 frames, the history starts to get overwritten as that data is too old anyway.

```

int i = networkClient->GetOutgoingSequence() & (COMMAND_HISTORY_SIZE-1);

...

// Store the command to the input client's history
memcpy(&inputClient.frame[i], &inputClient.command, sizeof(command_t));

clientData *cList = clientList;

```



```
// Store the commands to the clients' history
for(; cList != NULL; cList = cList->next)
{
    memcpy(&cList->frame[i], &cList->command, sizeof(command_t));
}

```

SendStartGame Function

This function builds a “start game” message and sends it to the server.

```
void CAmyWar::SendStartGame(void)
{
    char data[1400];
    dreamMessage message;
    message.Init(data, sizeof(data));

    message.WriteByte(USER_MES_STARTGAME);
    message.AddSequences(networkClient);

    networkClient->SendPacket(&message);
}

```

SendRequestNonDeltaFrame Function

This function builds a “request non-delta frame” message and sends it to the server.

```
void CAmyWar::SendRequestNonDeltaFrame(void)
{
    char data[1400];
    dreamMessage message;
    message.Init(data, sizeof(data));

    message.WriteByte(USER_MES_NONDELTAFRAME);
    message.AddSequences(networkClient);

    networkClient->SendPacket(&message);
}

```

Connect Function

This function sends a “connect” message to the server, telling the server we want start a connection with it. UDP protocol does not really connect to a server, so we sort of fake it this way.

```
void CAmyWar::Connect(void)
{
    if(init)
    {
        LogString("ArmyWar already initialized");
        return;
    }
}

```

```

    LogString("CArmyWar::Connect");

    init = true;

    networkClient->SendConnect(Lobby.GetLocalClient()->nickname);
}

```

Disconnect Function

This function disconnects from the server by sending a “disconnect” message. Some local variables are reset so they can be reused.

```

void CArmyWar::Disconnect(void)
{
    if(!init)
        return;

    LogString("CArmyWar::Disconnect");

    init = false;
    localClient = NULL;
    memset(&inputClient, 0, sizeof(clientData));

    networkClient->SendDisconnect();
}

```

ReadMoveCommand Function

This function reads the non-delta (absolute values) commands of a message. The parameters define the message to read and the client that owns the commands.

```

void CArmyWar::ReadMoveCommand(dreamMessage *mes, clientData *client)
{
    // Key
    client->serverFrame.key = mes->ReadByte();

    // Heading
    client->serverFrame.heading = mes->ReadShort();

    // Origin
    client->serverFrame.origin.x = mes->ReadFloat();
    client->serverFrame.origin.y = mes->ReadFloat();
    client->serverFrame.vel.x = mes->ReadFloat();
    client->serverFrame.vel.y = mes->ReadFloat();

    client->serverFrame.bullet.origin.x = mes->ReadFloat();
    client->serverFrame.bullet.origin.y = mes->ReadFloat();
    client->serverFrame.bullet.vel.x = mes->ReadFloat();
    client->serverFrame.bullet.vel.y = mes->ReadFloat();
    client->serverFrame.bullet.lifetime = mes->ReadShort();
    client->serverFrame.bullet.shot = mes->ReadByte();
}

```

```

int playerWithFlagIndex          = mes->ReadShort();

if(playerWithFlagIndex != -1)
{
    playerWithFlag = GetClientPointer(playerWithFlagIndex);
}

flagX = mes->ReadFloat();
flagY = mes->ReadFloat();

redScore = mes->ReadByte();
blueScore = mes->ReadByte();

// Read time to run command
client->serverFrame.msec = mes->ReadByte();

memcpy(&client->command, &client->serverFrame, sizeof(command_t));

// Fill the history array with the position we got
for(int f = 0; f < COMMAND_HISTORY_SIZE; f++)
{
    client->frame[f].predictedOrigin.x = client->command.origin.x;
    client->frame[f].predictedOrigin.y = client->command.origin.y;
    client->frame[f].bullet.predictedOrigin.x = client->
        command.bullet.origin.x;
    client->frame[f].bullet.predictedOrigin.y = client->
        command.bullet.origin.y;
}
}

```

ReadDeltaMoveCommand Function

This function reads the delta (change from last known value) commands. First the flags are read to see which commands are included in this message. Then each included command is read and stored to the `serverFrame` structure.

```

void CArmyWar::ReadDeltaMoveCommand(dreamMessage *mes, clientData *client)
{
    int processedFrame;
    int flags = 0;

    // Flags
    flags = mes->ReadByte();

    // Key
    if(flags & CMD_KEY)
    {
        client->serverFrame.key = mes->ReadByte();

        client->command.key = client->serverFrame.key;
    }
}

```

```
        LogString("Client %d: Read key %d", client->index, client->
            command.key);
    }

    if(flags & CMD_ORIGIN || flags & CMD_BULLET)
    {
        processedFrame = mes->ReadByte();
        client->processedFrame = processedFrame;
    }

    // Origin
    if(flags & CMD_ORIGIN)
    {
        client->serverFrame.origin.x = mes->ReadFloat();
        client->serverFrame.origin.y = mes->ReadFloat();

        client->serverFrame.vel.x = mes->ReadFloat();
        client->serverFrame.vel.y = mes->ReadFloat();

        if(client == localClient)
        {
            CheckPredictionError(processedFrame);
        }
        else
        {
            client->command.origin.x = client->serverFrame.origin.x;
            client->command.origin.y = client->serverFrame.origin.y;
        }
    }

    if(flags & CMD_BULLET)
    {
        client->serverFrame.bullet.origin.x = mes->ReadFloat();
        client->serverFrame.bullet.origin.y = mes->ReadFloat();
        client->serverFrame.bullet.vel.x = mes->ReadFloat();
        client->serverFrame.bullet.vel.y = mes->ReadFloat();
        client->serverFrame.bullet.shot = mes->ReadByte();

        client->command.bullet.shot = client->serverFrame.bullet.shot;

        if(client == localClient)
        {
            CheckBulletPredictionError(processedFrame);
        }
        else
        {
            client->command.bullet.origin.x = client->
                serverFrame.bullet.origin.x;
            client->command.bullet.origin.y = client->
                serverFrame.bullet.origin.y;
        }
    }

    // Flag & points
```

```

    if(flags & CMD_FLAG)
    {
        int playerWithFlagIndex = mes->ReadShort();

        if(playerWithFlagIndex != 0)
        {
            LogString("FLAG playerWithFlagIndex %d",
                    playerWithFlagIndex);
            playerWithFlag = GetClientPointer(playerWithFlagIndex);
        }
        else
        {
            playerWithFlag = NULL;
        }

        flagX = mes->ReadFloat();
        flagY = mes->ReadFloat();

        redScore = mes->ReadByte();
        blueScore = mes->ReadByte();

        CheckVictory();
    }

    // Someone died
    if(flags & CMD_KILL)
    {
        int died = mes->ReadByte();

        if(died)
            KillPlayer(client->index);
    }

    // Read time to run command
    client->command.msec = mes->ReadByte();
}

```

BuildDeltaMoveCommand Function

This function is used to build the command message based on local inputs. It is pretty simple; if a key has been pressed this frame, that key command is written into the message. Then the frame time is written. The parameters define the message to write to and the input data to use (normally `inputClient`).

```

void CAmyWar::BuildDeltaMoveCommand(dreamMessage *mes, clientData *theClient)
{
    int flags = 0;
    int last = (networkClient->GetOutgoingSequence() - 1) &
        (COMMAND_HISTORY_SIZE-1);

    // Check what needs to be updated

```

```

if(theClient->frame[last].key != theClient->command.key)
    flags |= CMD_KEY;

// Add to the message
// Flags
mes->WriteByte(flags);

// Key
if(flags & CMD_KEY)
{
    mes->WriteByte(theClient->command.key);
}

mes->WriteByte(theClient->command.msec);
}

```

RunNetwork Function

This function runs the network so the data will flow. Because this is client-side code, we can and should keep the frame rate under control or we might end up with clients that are too fast. The function reads and sends packets, and then finally moves the local objects so the player sees what is happening in the game.

```

void CAmyWar::RunNetwork(int msec)
{
    static int time = 0;
    time += msec;

    // Framerate is too high
    if(time < (1000 / 60))
        return;

    frametime = time / 1000.0f;
    time = 0;

    // Read packets from server and send new commands
    ReadPackets();
    SendCommand();

    int ack = networkClient->GetIncomingAcknowledged();
    int current = networkClient->GetOutgoingSequence();

    // Check that we haven't gone too far
    if(current - ack > COMMAND_HISTORY_SIZE)
        return;

    // Predict the frames that we are waiting on from the server
    for(int a = ack + 1; a < current; a++)
    {
        int prevframe = (a-1) & (COMMAND_HISTORY_SIZE-1);
        int frame = a & (COMMAND_HISTORY_SIZE-1);
    }
}

```

```

        PredictMovement(prevframe, frame);
    }

    MoveObjects();
}

```

You can see how the network is run below. First we read packets from the server, then we send new commands back to the server. After that, the interesting part begins: client prediction or dead reckoning. The local client will produce its own versions of the frames that it is waiting on from the server. The client first produces the frame that we last got from the server, so the client and the server agree on all the player positions and so on. Now that we are sending packets to the server each frame, but the server only sends packets to us in 100 ms intervals, we must guess what the frames in between look like. So the client produces these frames by using the known player velocities and so on. Once we get a new frame from the server, it all starts from the beginning.

```

// Read packets from server and send new commands
ReadPackets();
SendCommand();

int ack = networkClient->GetIncomingAcknowledged();
int current = networkClient->GetOutgoingSequence();

// Check that we haven't gone too far
if(current - ack > COMMAND_HISTORY_SIZE)
    return;

```

The current frame is produced with the following code. We must produce some past frames first, because we do not know if there was a prediction error (a difference between the server's frame and ours). If there was an error in one of the past frames, it becomes fixed once the server tells us the real frame data. It might take some time for the server to do that though, so we must keep producing the old frames until we know the real values of the frame data (player positions and movement).

```

// Predict the frames that we are waiting on from the server
for(int a = ack + 1; a < current; a++)
{
    int prevframe = (a-1) & (COMMAND_HISTORY_SIZE-1);
    int frame = a & (COMMAND_HISTORY_SIZE-1);

    PredictMovement(prevframe, frame);
}

MoveObjects();

```

client.cpp File

This file contains the game logic, meaning that here we have the functions that move the players and check for bullet or flag hits. The file in its entirety is listed here:

```
#include "common.h"

//-----
// Name: empty()
// Desc:
//-----
CArmyWar::CArmyWar()
{
    networkClient    = new dreamClient;
    clientList       = NULL;
    localClient      = NULL;
    clients          = 0;

    memset(&inputClient, 0, sizeof(clientData));
    memset(&mapdata, 0, sizeof(mapdata));

    frametime        = 0.0f;

    inProgress       = false;
    init             = false;

    scrollX          = 0;
    scrollY          = 0;

    tileScrollX     = 0;
    tileScrollY     = 0;

    gameIndex       = 0;

    targetRotation   = 0.0f;

    redScore        = 0;
    blueScore       = 0;

    playerWithFlag  = NULL;

    next            = NULL;
}

//-----
// Name: empty()
// Desc:
//-----
CArmyWar::~CArmyWar()
{
    delete networkClient;
```



```

}

//-----
// Name: empty()
// Desc:
//-----
void CArmyWar::InitializeEngine(void)
{
    // Init the graphics engine
    GFX_Init("Army War Engine v2.0", 640, 480, 16, 0, ApplicationProc);

    // Load required graphics
    GFX_LoadBitmap(&grass, "gfx\grass.bmp");
    GFX_LoadBitmap(&redman, "gfx\redman.bmp");
    GFX_LoadBitmap(&blueman, "gfx\blueman.bmp");
    GFX_LoadBitmap(&tree, "gfx\tree.bmp");
    GFX_LoadBitmap(&redtarget, "gfx\redtarget.bmp");
    GFX_LoadBitmap(&bluetarget, "gfx\bluetarget.bmp");
    GFX_LoadBitmap(&flag, "gfx\flag.bmp");

    GFX_LoadBitmap(&rednumbers[0], "gfx\red0.bmp");
    GFX_LoadBitmap(&rednumbers[1], "gfx\red1.bmp");
    GFX_LoadBitmap(&rednumbers[2], "gfx\red2.bmp");
    GFX_LoadBitmap(&rednumbers[3], "gfx\red3.bmp");
    GFX_LoadBitmap(&rednumbers[4], "gfx\red4.bmp");
    GFX_LoadBitmap(&rednumbers[5], "gfx\red5.bmp");
    GFX_LoadBitmap(&rednumbers[6], "gfx\red6.bmp");
    GFX_LoadBitmap(&rednumbers[7], "gfx\red7.bmp");
    GFX_LoadBitmap(&rednumbers[8], "gfx\red8.bmp");
    GFX_LoadBitmap(&rednumbers[9], "gfx\red9.bmp");

    GFX_LoadBitmap(&bluenumbers[0], "gfx\blue0.bmp");
    GFX_LoadBitmap(&bluenumbers[1], "gfx\blue1.bmp");
    GFX_LoadBitmap(&bluenumbers[2], "gfx\blue2.bmp");
    GFX_LoadBitmap(&bluenumbers[3], "gfx\blue3.bmp");
    GFX_LoadBitmap(&bluenumbers[4], "gfx\blue4.bmp");
    GFX_LoadBitmap(&bluenumbers[5], "gfx\blue5.bmp");
    GFX_LoadBitmap(&bluenumbers[6], "gfx\blue6.bmp");
    GFX_LoadBitmap(&bluenumbers[7], "gfx\blue7.bmp");
    GFX_LoadBitmap(&bluenumbers[8], "gfx\blue8.bmp");
    GFX_LoadBitmap(&bluenumbers[9], "gfx\blue9.bmp");

    // Set the scroll positions
    scrollX = 40*32;

    if(localClient->team == RED_TEAM)
        scrollY = 90*32;
    else
        scrollY = 0;

    // Set the flag position
    flagX = 49*32;
    flagY = 49*32;

```

```

        playerWithFlag = NULL;

        // Reset score counters
        redScore    = 0;
        blueScore   = 0;
    }

    //-----
    // Name: empty()
    // Desc:
    //-----
    void CArmyWar::Shutdown(void)
    {
        Disconnect();

        GFX_Begin();
        GFX_Shutdown();
        GFX_End();
    }

    //-----
    // Name: empty()
    // Desc:
    //-----
    void CArmyWar::DrawMap(void)
    {
        int heading = 0;

        // Work out how many tiles have been scrolled
        tileScrollX = scrollX/32;
        tileScrollY = scrollY/32;

        for(int i = (tileScrollX)-2; i < (tileScrollX)+21; i++)
        {
            for(int j = (tileScrollY)+15; j > (tileScrollY)-2; j-)
            {
                GFX_Blit(&grass, (32*i)-(scrollX), (32*j)-(scrollY), 32,
                        32, 0);

                // Draw a tree if required
                if(mapdata[i][j] == true)
                {
                    GFX_Blit(&tree, (32*i+16)-(scrollX), (32*j+16)-
                            (scrollY), 32, 32, 0);
                }

                // Draw the static targets
                if(i==49 && j==3)
                {
                    // draw the blue target
                    GFX_Blit(&bluetarget, (32*i)-(scrollX), (32*j)-
                            (scrollY), 32, 32, targetRotation);
                }
            }
        }
    }

```

```

        if(i==49 && j==97)
        {
            // draw the red target
            GFX_Blit(&redtarget, (32*i)-(scrollX),(32*j)-
                (scrollY), 32, 32, targetRotation);
        }
    }

// Render the flag
GFX_Blit(&flag, ((int) flagX)-(scrollX), ((int) flagY)-(scrollY),
    32, 32, 0);

// Render players
clientData *list = clientList;

for(; list != NULL; list = list->next)
{
    if(list->team == RED_TEAM)
    {
        GFX_Blit(&redman, ((int) list->command.origin.x)-(scrollX),
            ((int) list->command.origin.y)-(scrollY),
            32, 32, (float) list->command.heading);
    }

    if(list->team == BLUE_TEAM)
    {
        GFX_Blit(&blueman, ((int) list->command.origin.x)-
            (scrollX), ((int) list->command.origin.y)-
            (scrollY), 32, 32, (float) list->command.heading);
    }

// Render bullets
if(list->command.bullet.shot)
{
    if(list->team == RED_TEAM)
    {
        GFX_RectFill(((int) list->command.bullet.origin.x-2)-
            (scrollX), ((int) list->
            command.bullet.origin.y-2)-(scrollY),
            ((int) list->command.bullet.origin.x+2)-
            (scrollX), ((int) list->
            command.bullet.origin.y+2)-(scrollY),
            200, 0, 0);
    }

    if(list->team == BLUE_TEAM)
    {
        GFX_RectFill(((int) list->command.bullet.origin.x-2)-
            (scrollX), ((int) list->
            command.bullet.origin.y-2)-(scrollY),
            ((int) list->command.bullet.origin.x+2)-
            (scrollX), ((int) list->

```

```

        command.bullet.origin.y+2)-(scrollY),
        0, 0, 200);
    }
}

// Finally, render the team scores
GFX_Blit(&rednumbers[redScore], 5, 410, 64, 64, 0);
GFX_Blit(&bluenumbers[blueScore], 570, 410, 64, 64, 0);
}

//-----
// Name: empty()
// Desc:
//-----
void CArmyWar::Frame(void)
{
    if(!localClient)
        return;

    // Scroll the map to follow the local player
    if((localClient->command.origin.x - scrollX) > 340)
    {
        if(scrollX <= 3200-(19*32)-2)
            scrollX += 2;
    }

    if((localClient->command.origin.x - scrollX) < 300)
    {
        if(scrollX >= 2)
            scrollX -= 2;
    }

    if((localClient->command.origin.y - scrollY) > 260)
    {
        if(scrollY <= 3200-(15*32)-2)
            scrollY += 2;
    }

    if((localClient->command.origin.y - scrollY) < 220)
    {
        if(scrollY >= 2)
            scrollY -= 2;
    }

    // Move the flag with the player
    if(playerWithFlag)
    {
        flagX = playerWithFlag->command.origin.x;
        flagY = playerWithFlag->command.origin.y;
    }

    // Rotate the target images
    if(targetRotation < 360)

```

```

        targetRotation += 1;
    else
        targetRotation -= targetRotation;

    // Draw map
    GFX_Begin();
    {
        DrawMap();
    }
    GFX_End();
}

//-----
// Name: empty()
// Desc:
//-----
void CArmyWar::CheckVictory(void)
{
    if(localClient == NULL)
        return;

    // Check team scores
    if(redScore > 1)
    {
        if(localClient->team == RED_TEAM)
        {
            MessageBox(NULL, "Your team (RED) won!", "Victory", MB_OK);
        }
        else
        {
            MessageBox(NULL, "The other team (RED) won", "Failure",
                MB_OK);
        }

        Shutdown();
    }
    if(blueScore > 1)
    {
        if(localClient->team == BLUE_TEAM)
        {
            MessageBox(NULL, "Your team (BLUE) won!", "Victory",
                MB_OK);
        }
        else
        {
            MessageBox(NULL, "The other team (BLUE) won", "Failure",
                MB_OK);
        }

        Shutdown();
    }
}
}

```

```
//-----  
// Name: empty()  
// Desc:  
//-----  
void CArmyWar::KillPlayer(int index)  
{  
    LogString("Player %d died", index);  
  
    clientData *client = GetClientPointer(index);  
  
    if(client == NULL)  
        return;  
  
    client->command.origin.x = client->startPos.x;  
    client->command.origin.y = client->startPos.y;  
}  
  
//-----  
// Name: empty()  
// Desc:  
//-----  
clientData *CArmyWar::GetClientPointer(int index)  
{  
    for(clientData *cList = clientList; cList != NULL; cList =  
        cList->next)  
    {  
        if(cList->index == index)  
            return cList;  
    }  
  
    return NULL;  
}  
  
//-----  
// Name: empty()  
// Desc:  
//-----  
void CArmyWar::CheckKeys(void)  
{  
    inputClient.command.key = 0;  
  
    if(keys[VK_ESCAPE])  
    {  
        Shutdown();  
  
        keys[VK_ESCAPE] = false;  
    }  
  
    if(keys[VK_DOWN])  
    {  
        inputClient.command.key |= KEY_DOWN;  
    }  
  
    if(keys[VK_UP])
```

```

    {
        inputClient.command.key |= KEY_UP;
    }

    if(keys[VK_LEFT])
    {
        inputClient.command.key |= KEY_LEFT;
    }

    if(keys[VK_RIGHT])
    {
        inputClient.command.key |= KEY_RIGHT;
    }

    if(keys[VK_SPACE])
    {
        inputClient.command.key |= KEY_WEAPON;
    }

    inputClient.command.msec = (int) (frametime * 1000);
}

//-----
// Name: empty()
// Desc:
//-----
void CAmyWar::CheckPredictionError(int a)
{
    if(a < 0 && a > COMMAND_HISTORY_SIZE)
        return;

    float errorX = localClient->serverFrame.origin.x -
        localClient->frame[a].predictedOrigin.x;

    float errorY = localClient->serverFrame.origin.y -
        localClient->frame[a].predictedOrigin.y;

    // Fix the prediction error
    if((errorX != 0.0f) || (errorY != 0.0f))
    {
        localClient->frame[a].predictedOrigin.x = localClient->
            serverFrame.origin.x;
        localClient->frame[a].predictedOrigin.y = localClient->
            serverFrame.origin.y;

        localClient->frame[a].vel.x = localClient->serverFrame.vel.x;
        localClient->frame[a].vel.y = localClient->serverFrame.vel.y;

        LogString("Prediction error for frame %d: %f, %f\n", a,
            errorX, errorY);
    }
}

```

```

//-----
// Name: empty()
// Desc:
//-----
void CArmyWar::CheckBulletPredictionError(int a)
{
    if(a < 0 && a > COMMAND_HISTORY_SIZE)
        return;

    float errorX = localClient->serverFrame.bullet.origin.x - localClient->
        frame[a].bullet.predictedOrigin.x;
    float errorY = localClient->serverFrame.bullet.origin.y - localClient->
        frame[a].bullet.predictedOrigin.y;

    // Fix the prediction error
    if((errorX != 0.0f) || (errorY != 0.0f))
    {
        localClient->frame[a].bullet.predictedOrigin.x = localClient->
            serverFrame.bullet.origin.x;
        localClient->frame[a].bullet.predictedOrigin.y = localClient->
            serverFrame.bullet.origin.y;

        localClient->frame[a].bullet.vel.x = localClient->
            serverFrame.bullet.vel.x;
        localClient->frame[a].bullet.vel.y = localClient->
            serverFrame.bullet.vel.y;

        LogString("Bullet prediction error for frame %d:
            %f, %f\n", a, errorX, errorY);
    }
}

//-----
// Name: empty()
// Desc:
//-----
void CArmyWar::CalculateVelocity(command_t *command, float frametime)
{
    int checkX;
    int checkY;

    float multiplier = 100.0f;

    command->vel.x = 0.0f;
    command->vel.y = 0.0f;

    if(command->key & KEY_UP)
    {
        checkX = (int) (command->origin.x / 32.0f);
        checkY = (int) ((command->origin.y - multiplier * frametime) /
            32.0f);

        if(mapdata[checkX][checkY] == false)
            command->vel.y += -multiplier * frametime;
    }
}

```



```

    }

    if(command->key & KEY_DOWN)
    {
        checkX = (int) (command->origin.x / 32.0f);
        checkY = (int) ((command->origin.y + multiplier * frametime) /
                        32.0f);

        if(mapdata[checkX][checkY] == false)
            command->vel.y += multiplier * frametime;
    }

    if(command->key & KEY_LEFT)
    {
        checkX = (int) ((command->origin.x - multiplier * frametime) /
                        32.0f);
        checkY = (int) (command->origin.y / 32.0f);

        if(mapdata[checkX][checkY] == false)
            command->vel.x += -multiplier * frametime;
    }

    if(command->key & KEY_RIGHT)
    {
        checkX = (int) ((command->origin.x + multiplier * frametime) /
                        32.0f);
        checkY = (int) (command->origin.y / 32.0f);

        if(mapdata[checkX][checkY] == false)
            command->vel.x += multiplier * frametime;
    }
}

//-----
// Name: empty()
// Desc:
//-----
void CArmyWar::CalculateHeading(command_t *command)
{
    // Right
    if((command->vel.x > 0.0f) &&
        (command->vel.y == 0.0f))
    {
        command->heading = EAST;
    }

    // Left
    if((command->vel.x < 0.0f) &&
        (command->vel.y == 0.0f))
    {
        command->heading = WEST;
    }
}

```

```
// Down
if((command->vel.y > 0.0f) &&
    (command->vel.x == 0.0f))
{
    command->heading = SOUTH;
}

// Up
if((command->vel.y < 0.0f) &&
    (command->vel.x == 0.0f))
{
    command->heading = NORTH;
}

// Down-Right
if((command->vel.x > 0.0f) &&
    (command->vel.y > 0.0f))
{
    command->heading = SOUTHEAST;
}

// Up-Right
if((command->vel.x > 0.0f) &&
    (command->vel.y < 0.0f))
{
    command->heading = NORTHEAST;
}

// Down-Left
if((command->vel.x < 0.0f) &&
    (command->vel.y > 0.0f))
{
    command->heading = SOUTHWEST;
}

// Up-Left
if((command->vel.x < 0.0f) &&
    (command->vel.y < 0.0f))
{
    command->heading = NORTHWEST;
}
}

//-----
// Name: empty()
// Desc:
//-----
void CArmyWar::CalculateBulletVelocity(command_t *command)
{
    command->bullet.shot = true;

    if(command->heading == NORTH)
    {
        command->bullet.vel.x = 0.0f;
    }
}
```

```

        command->bullet.vel.y = -200.0f;
    }
    if(command->heading == SOUTH)
    {
        command->bullet.vel.x = 0.0f;
        command->bullet.vel.y = 200.0f;
    }
    if(command->heading == EAST)
    {
        command->bullet.vel.x = 200.0f;
        command->bullet.vel.y = 0.0f;
    }
    if(command->heading == WEST)
    {
        command->bullet.vel.x = -200.0f;
        command->bullet.vel.y = 0.0f;
    }

    if(command->heading == NORTHEAST)
    {
        command->bullet.vel.x = 200.0f;
        command->bullet.vel.y = -200.0f;
    }
    if(command->heading == NORTHWEST)
    {
        command->bullet.vel.x = -200.0f;
        command->bullet.vel.y = -200.0f;
    }
    if(command->heading == SOUTHEAST)
    {
        command->bullet.vel.x = 200.0f;
        command->bullet.vel.y = 200.0f;
    }
    if(command->heading == SOUTHWEST)
    {
        command->bullet.vel.x = -200.0f;
        command->bullet.vel.y = 200.0f;
    }
}

//-----
// Name: empty()
// Desc:
//-----
void CAmyWar::PredictMovement(int prevFrame, int curFrame)
{
    if(!localClient)
        return;

    float frametime = inputClient.frame[curFrame].msec / 1000.0f;

    localClient->frame[curFrame].key = inputClient.frame[curFrame].key;

    //

```

```
// Player ->
//

// Process commands
CalculateVelocity(&localClient->frame[curFrame], frametime);
CalculateHeading(&localClient->frame[curFrame]);

// Calculate new predicted origin
localClient->frame[curFrame].predictedOrigin.x =
    localClient->frame[prevFrame].predictedOrigin.x +
    localClient->frame[curFrame].vel.x;

localClient->frame[curFrame].predictedOrigin.y =
    localClient->frame[prevFrame].predictedOrigin.y +
    localClient->frame[curFrame].vel.y;

// Copy values to "current" values
localClient->command.predictedOrigin.x = localClient->
    frame[curFrame].predictedOrigin.x;
localClient->command.predictedOrigin.y = localClient->
    frame[curFrame].predictedOrigin.y;
localClient->command.vel.x = localClient->frame[curFrame].vel.x;
localClient->command.vel.y = localClient->frame[curFrame].vel.y;
localClient->command.heading = localClient->frame[curFrame].heading;

//
// Bullet ->
//

// First set the previous values
localClient->frame[curFrame].bullet.shot = localClient->
    frame[prevFrame].bullet.shot;
localClient->frame[curFrame].bullet.vel.x = localClient->
    frame[prevFrame].bullet.vel.x;
localClient->frame[curFrame].bullet.vel.y = localClient->
    frame[prevFrame].bullet.vel.y;
localClient->frame[curFrame].bullet.lifetime = localClient->
    frame[prevFrame].bullet.lifetime;

// The bullet is carried by the player
if(localClient->frame[curFrame].bullet.shot == false)
{
    localClient->frame[curFrame].bullet.predictedOrigin.x =
        localClient->frame[curFrame].predictedOrigin.x;
    localClient->frame[curFrame].bullet.predictedOrigin.y =
        localClient->frame[curFrame].predictedOrigin.y;
}
else
{
    localClient->frame[curFrame].bullet.lifetime += (int) (frametime
        * 1000.0f);

    if(localClient->frame[curFrame].bullet.lifetime > 2000)
```

```

    {
        localClient->frame[curFrame].bullet.shot = false;
        localClient->frame[curFrame].bullet.lifetime = 0;

        localClient->frame[curFrame].bullet.predictedOrigin.x =
            localClient->frame[curFrame].predictedOrigin.x;
        localClient->frame[curFrame].bullet.predictedOrigin.y =
            localClient->frame[curFrame].predictedOrigin.y;
    }
}

// Calculate the heading for the bullet only when firing
if(localClient->frame[curFrame].key & KEY_WEAPON &&
    localClient->frame[curFrame].bullet.shot == false)
{
    CalculateBulletVelocity(&localClient->frame[curFrame]);
}

// If the bullet is in the air (shot), update its origin
if(localClient->frame[curFrame].bullet.shot)
{
    localClient->frame[curFrame].bullet.predictedOrigin.x =
        localClient->frame[prevFrame].bullet.predictedOrigin.x +
        localClient->frame[curFrame].bullet.vel.x * frametime;

    localClient->frame[curFrame].bullet.predictedOrigin.y =
        localClient->frame[prevFrame].bullet.predictedOrigin.y +
        localClient->frame[curFrame].bullet.vel.y * frametime;
}

// Copy values to "current" values
localClient->command.bullet.predictedOrigin.x = localClient->
    frame[curFrame].bullet.predictedOrigin.x;
localClient->command.bullet.predictedOrigin.y = localClient->
    frame[curFrame].bullet.predictedOrigin.y;
localClient->command.bullet.vel.x = localClient->
    frame[curFrame].bullet.vel.x;
localClient->command.bullet.vel.y = localClient->
    frame[curFrame].bullet.vel.y;
}

//-----
// Name: empty()
// Desc:
//-----
void CAmyWar::MoveObjects(void)
{
    if(!localClient)
        return;

    clientData *client = clientList;

    for(; client != NULL; client = client->next)
    {

```

```

// Remote players
if(client != localClient)
{
    CalculateVelocity(&client->command, frametime);
    CalculateHeading(&client->command);

    client->command.origin.x += client->command.vel.x;
    client->command.origin.y += client->command.vel.y;

    client->command.bullet.origin.x += client->
        serverFrame.bullet.vel.x * frametime;
    client->command.bullet.origin.y += client->
        serverFrame.bullet.vel.y * frametime;
}

// Local player
else
{
    client->command.origin.x = client->
        command.predictedOrigin.x;
    client->command.origin.y = client->
        command.predictedOrigin.y;

    client->command.bullet.origin.x = client->
        command.bullet.predictedOrigin.x;
    client->command.bullet.origin.y = client->
        command.bullet.predictedOrigin.y;
}
}
}

```

CArmyWar Constructor and Destructor Functions

These two functions initialize and uninitialize the CArmyWar object that contains game-specific data. The constructor allocates memory and sets all the variables to initial states. The destructor frees the allocated memory, which is very important (otherwise we would have a memory leak).

```

CArmyWar::CArmyWar()
{
    networkClient    = new dreamClient;
    clientList       = NULL;
    localClient      = NULL;
    clients          = 0;

    memset(&inputClient, 0, sizeof(clientData));
    memset(&mapdata, 0, sizeof(mapdata));

    frametime        = 0.0f;

    inProgress       = false;
}

```

```

        init            = false;

        scrollX         = 0;
        scrollY         = 0;

        tileScrollX    = 0;
        tileScrollY    = 0;

        gameIndex      = 0;

        targetRotation = 0.0f;

        redScore       = 0;
        blueScore      = 0;

        playerWithFlag = NULL;

        next           = NULL;
    }

CArmyWar::~CArmyWar()
{
    delete networkClient;
}

```

InitializeEngine Function

We are still initializing things with this function. This one's purpose is to initialize 2DLIB — our graphics engine — and load the graphics into memory. It also sets the flag to its correct initial position and moves the screen (scrolls) to the correct position, based on which team the local player is on. Notice how the team score bitmaps are loaded into an array, so we can use the team scores as an index to that array to get the correct bitmap number.

```

void CArmyWar::InitializeEngine(void)
{
    // Init the graphics engine
    GFX_Init("Army War Engine v2.0", 640, 480, 16, 0, ApplicationProc);

    // Load required graphics
    GFX_LoadBitmap(&grass, "gfx\\grass.bmp");
    GFX_LoadBitmap(&redman, "gfx\\redman.bmp");
    GFX_LoadBitmap(&blueman, "gfx\\blueman.bmp");
    GFX_LoadBitmap(&tree, "gfx\\tree.bmp");
    GFX_LoadBitmap(&redtarget, "gfx\\redtarget.bmp");
    GFX_LoadBitmap(&bluetarget, "gfx\\bluetarget.bmp");
    GFX_LoadBitmap(&flag, "gfx\\flag.bmp");

    GFX_LoadBitmap(&rednumbers[0], "gfx\\red0.bmp");
    GFX_LoadBitmap(&rednumbers[1], "gfx\\red1.bmp");
    GFX_LoadBitmap(&rednumbers[2], "gfx\\red2.bmp");
    GFX_LoadBitmap(&rednumbers[3], "gfx\\red3.bmp");
}

```

```

GFX_LoadBitmap(&rednumbers[4], "gfx\\red4.bmp");
GFX_LoadBitmap(&rednumbers[5], "gfx\\red5.bmp");
GFX_LoadBitmap(&rednumbers[6], "gfx\\red6.bmp");
GFX_LoadBitmap(&rednumbers[7], "gfx\\red7.bmp");
GFX_LoadBitmap(&rednumbers[8], "gfx\\red8.bmp");
GFX_LoadBitmap(&rednumbers[9], "gfx\\red9.bmp");

GFX_LoadBitmap(&bluenumbers[0], "gfx\\blue0.bmp");
GFX_LoadBitmap(&bluenumbers[1], "gfx\\blue1.bmp");
GFX_LoadBitmap(&bluenumbers[2], "gfx\\blue2.bmp");
GFX_LoadBitmap(&bluenumbers[3], "gfx\\blue3.bmp");
GFX_LoadBitmap(&bluenumbers[4], "gfx\\blue4.bmp");
GFX_LoadBitmap(&bluenumbers[5], "gfx\\blue5.bmp");
GFX_LoadBitmap(&bluenumbers[6], "gfx\\blue6.bmp");
GFX_LoadBitmap(&bluenumbers[7], "gfx\\blue7.bmp");
GFX_LoadBitmap(&bluenumbers[8], "gfx\\blue8.bmp");
GFX_LoadBitmap(&bluenumbers[9], "gfx\\blue9.bmp");

// Set the scroll positions
scrollX = 40*32;

if(localClient->team == RED_TEAM)
    scrollY = 90*32;
else
    scrollY = 0;

// Set the flag position
flagX = 49*32;
flagY = 49*32;

playerWithFlag = NULL;

// Reset score counters
redScore = 0;
blueScore = 0;
}

```

Shutdown Function

The `Shutdown` function disconnects from the server (if connected) and shuts down the graphics engine.

```

void CArmyWar::Shutdown(void)
{
    Disconnect();

    GFX_Begin();
    GFX_Shutdown();
    GFX_End();
}

```


DrawMap Function

Now we finally get to do something besides initializing things. This function draws the game map, trees, grass, players, flag, and bullets. First we need to figure out their positions on the screen by looking at the scroll coordinates. One tile is 32 pixels in height and width, so it is easy to calculate which tiles to draw. Just divide the actual pixel scroll values by 32, and you get the first tiles to draw (top-left corner). Then we loop through the visible tiles and draw what lies in that tile. To get the correct position on screen, you need to multiply the tile scroll value by 32 and then subtract the pixel scroll value from that.

```
void CArmyWar::DrawMap(void)
{
    int heading = 0;

    // Work out how many tiles have been scrolled
    tileScrollX = scrollX/32;
    tileScrollY = scrollY/32;

    for(int i = (tileScrollX)-2; i < (tileScrollX)+21; i++)
    {
        for(int j = (tileScrollY)+15; j > (tileScrollY)-2; j-)
        {
            GFX_Blit(&grass, (32*i)-(scrollX),(32*j)-(scrollY), 32,
                    32, 0);

            // Draw a tree if required
            if(mapdata[i][j] == true)
            {
                GFX_Blit(&tree, (32*i+16)-(scrollX),(32*j+16)-
                        (scrollY), 32, 32, 0);
            }

            // Draw the static targets
            if(i==49 && j==3)
            {
                // draw the blue target
                GFX_Blit(&bluetarget, (32*i)-(scrollX),(32*j)-
                        (scrollY), 32, 32, targetRotation);
            }

            if(i==49 && j==97)
            {
                // draw the red target
                GFX_Blit(&redtarget, (32*i)-(scrollX),(32*j)-
                        (scrollY), 32, 32, targetRotation);
            }
        }
    }
}
```

```
// Render the flag
GFX_Blit(&flag, ((int) flagX)-(scrollX), ((int) flagY)-(scrollY),
        32, 32, 0);

// Render players
clientData *list = clientList;

for(; list != NULL; list = list->next)
{
    if(list->team == RED_TEAM)
    {
        GFX_Blit(&redman, ((int) list->command.origin.x)-(scrollX),
                ((int) list->command.origin.y)-(scrollY),
                32, 32, (float) list->command.heading);
    }

    if(list->team == BLUE_TEAM)
    {
        GFX_Blit(&blueman, ((int) list->command.origin.x)-
                (scrollX), ((int) list->command.origin.y)-
                (scrollY), 32, 32, (float) list->command.heading);
    }

    // Render bullets
    if(list->command.bullet.shot)
    {
        if(list->team == RED_TEAM)
        {
            GFX_RectFill(((int) list->command.bullet.origin.x-2)-
                (scrollX), ((int) list->
                command.bullet.origin.y-2)-(scrollY),
                ((int) list->command.bullet.origin.x+2)-
                (scrollX), ((int) list->
                command.bullet.origin.y+2)-(scrollY),
                200, 0, 0);
        }

        if(list->team == BLUE_TEAM)
        {
            GFX_RectFill(((int) list->command.bullet.origin.x-2)-
                (scrollX), ((int) list->
                command.bullet.origin.y-2)-(scrollY),
                ((int) list->
                command.bullet.origin.x+2)-(scrollX),
                ((int) list->command.bullet.origin.y+2)-
                (scrollY), 0, 0, 200);
        }
    }
}

// Finally, render the team scores
GFX_Blit(&rednumbers[redScore], 5, 410, 64, 64, 0);
GFX_Blit(&bluenumbers[blueScore], 570, 410, 64, 64, 0);
}
```

Here we draw a tree by looking at the map data. If the value on that tile is 1 (or true in other words), we draw a tree. Otherwise, we draw only plain grass.

```
// Draw a tree if required
if(mapdata[i][j] == true)
{
    GFX_Blit(&tree, (32*i+16)-(scrollX), (32*j+16)-(scrollY), 32, 32, 0);
}
```

Frame Function

This function's purpose is to handle the game logic. It scrolls the screen by trying to follow the player's position. Also, if someone is carrying the flag, the flag is moved with the player. The teams' base indicators are rotated to add some graphical effects. Finally, the map is drawn with the current positions.

```
void CArmyWar::Frame(void)
{
    if(!localClient)
        return;

    // Scroll the map to follow the local player
    if((localClient->command.origin.x - scrollX) > 340)
    {
        if(scrollX <= 3200-(19*32)-2)
            scrollX += 2;
    }

    if((localClient->command.origin.x - scrollX) < 300)
    {
        if(scrollX >= 2)
            scrollX -= 2;
    }

    if((localClient->command.origin.y - scrollY) > 260)
    {
        if(scrollY <= 3200-(15*32)-2)
            scrollY += 2;
    }

    if((localClient->command.origin.y - scrollY) < 220)
    {
        if(scrollY >= 2)
            scrollY -= 2;
    }

    // Move the flag with the player
    if(playerWithFlag)
    {
        flagX = playerWithFlag->command.origin.x;
        flagY = playerWithFlag->command.origin.y;
    }
}
```

```
    }

    // Rotate the target images
    if(targetRotation < 360)
        targetRotation += 1;
    else
        targetRotation -= targetRotation;

    // Draw map
    GFX_Begin();
    {
        DrawMap();
    }
    GFX_End();
}
```

CheckVictory Function

This function checks the victory conditions. If they are met, it notifies the player whose team won the game and then shuts down the game engine after the player presses the OK button.

```
void CAmyWar::CheckVictory(void)
{
    if(localClient == NULL)
        return;

    // Check team scores
    if(redScore > 1)
    {
        if(localClient->team == RED_TEAM)
        {
            MessageBox(NULL, "Your team (RED) won!", "Victory", MB_OK);
        }
        else
        {
            MessageBox(NULL, "The other team (RED) won", "Failure",
                MB_OK);
        }

        Shutdown();
    }
    if(blueScore > 1)
    {
        if(localClient->team == BLUE_TEAM)
        {
            MessageBox(NULL, "Your team (BLUE) won!", "Victory",
                MB_OK);
        }
        else
        {
            MessageBox(NULL, "The other team (BLUE) won", "Failure",
                MB_OK);
        }
    }
}
```

```

    }
    Shutdown();
}
}

```

KillPlayer Function

The `KillPlayer` function makes sure that when a player is shot, the player is moved back to the start position. This function fixes the prediction error that would occur if the server only sent the new position (without this function, the client would keep the shot player at the position where he or she was shot).

```

void CArmyWar::KillPlayer(int index)
{
    LogString("Player %d died", index);

    clientData *client = GetClientPointer(index);

    if(client == NULL)
        return;

    client->command.origin.x = client->startPos.x;
    client->command.origin.y = client->startPos.y;
}

```

GetClientPointer Function

This function returns a pointer to a player in the client list, and the player is chosen with the player index number given as a parameter. If no such index number is found, the function returns `NULL`, so we must check the result before using it.

```

clientData *CArmyWar::GetClientPointer(int index)
{
    for(clientData *cList = clientList; cList != NULL; cList =
        cList->next)
    {
        if(cList->index == index)
            return cList;
    }

    return NULL;
}

```

CheckKeys Function

Here we check the input keys and set the commands to correspond to keypresses. Notice that we use `inputClient` here. At the beginning of the function we reset the commands because we check the commands every frame.

```
void CAmyWar::CheckKeys(void)
{
    inputClient.command.key = 0;

    if(keys[VK_ESCAPE])
    {
        Shutdown();

        keys[VK_ESCAPE] = false;
    }

    if(keys[VK_DOWN])
    {
        inputClient.command.key |= KEY_DOWN;
    }

    if(keys[VK_UP])
    {
        inputClient.command.key |= KEY_UP;
    }

    if(keys[VK_LEFT])
    {
        inputClient.command.key |= KEY_LEFT;
    }

    if(keys[VK_RIGHT])
    {
        inputClient.command.key |= KEY_RIGHT;
    }

    if(keys[VK_SPACE])
    {
        inputClient.command.key |= KEY_WEAPON;
    }

    inputClient.command.msec = (int) (frametime * 1000);
}
```

CheckPredictionError Function

This function might seem a little bit odd at first, but it really is not odd at all. It compares the frame received from the server and the frame produced by the local client. If there is any difference between them, the server's frame is used. In other words, we always use the server's frame, but because the server sends us frame data in 100 ms intervals we need to produce the frames in between ourselves. As there is no way to determine when exactly the server gives us the next frame, we check every frame the server gives us. When the server gives us a new frame, it is actually old already (because of network lag and other processing times). So we have to compare an old frame that is stored in our frame history (or command history). If there is an error, the client

fixes the error in the past frame and reproduces all the frames all the way to the current one. This is seen in the `RunNetwork()` function (`network.cpp`).

```
void CAmyWar::CheckPredictionError(int a)
{
    if(a < 0 && a > COMMAND_HISTORY_SIZE)
        return;

    float errorX =
        localClient->serverFrame.origin.x - localClient->
        frame[a].predictedOrigin.x;

    float errorY =
        localClient->serverFrame.origin.y - localClient->
        frame[a].predictedOrigin.y;

    // Fix the prediction error
    if((errorX != 0.0f) || (errorY != 0.0f))
    {
        localClient->frame[a].predictedOrigin.x = localClient->
            serverFrame.origin.x;
        localClient->frame[a].predictedOrigin.y = localClient->
            serverFrame.origin.y;

        localClient->frame[a].vel.x = localClient->serverFrame.vel.x;
        localClient->frame[a].vel.y = localClient->serverFrame.vel.y;

        LogString("Prediction error for frame %d: %f, %f\n", a,
            errorX, errorY);
    }
}
```

CheckBulletPredictionError Function

This function is similar to the `CheckPredictionError()` function, except that this one is for bullets.

```
void CAmyWar::CheckBulletPredictionError(int a)
{
    if(a < 0 && a > COMMAND_HISTORY_SIZE)
        return;

    float errorX = localClient->serverFrame.bullet.origin.x - localClient->
        frame[a].bullet.predictedOrigin.x;
    float errorY = localClient->serverFrame.bullet.origin.y - localClient->
        frame[a].bullet.predictedOrigin.y;

    // Fix the prediction error
    if((errorX != 0.0f) || (errorY != 0.0f))
    {
        localClient->frame[a].bullet.predictedOrigin.x = localClient->
            serverFrame.bullet.origin.x;
        localClient->frame[a].bullet.predictedOrigin.y = localClient->
```

```

        serverFrame.bullet.origin.y;

    localClient->frame[a].bullet.vel.x = localClient->
        serverFrame.bullet.vel.x;
    localClient->frame[a].bullet.vel.y = localClient->
        serverFrame.bullet.vel.y;

    LogString("Bullet prediction error for frame %d:  %f, %f\n", a,
        errorX, errorY);
}
}

```

CalculateVelocity Function

This function calculates the local player's movement velocity (or direction really, as there is no acceleration). The function does some simple collision detection by looking at the map data at the position to which the player is moving. If there is a tree, the player moves nowhere. Notice that we use `frametime` to make the player move the correct amount in any given time. This is very, very important, especially in a network game. It makes the players move the same amount on all the computers in any given time (for example, if you keep the "up" key pressed for one second).

```

void CArmyWar::CalculateVelocity(command_t *command, float frametime)
{
    int checkX;
    int checkY;

    float multiplier = 100.0f;

    command->vel.x = 0.0f;
    command->vel.y = 0.0f;

    if(command->key & KEY_UP)
    {
        checkX = (int) (command->origin.x / 32.0f);
        checkY = (int) ((command->origin.y - multiplier * frametime) /
            32.0f);

        if(mapdata[checkX][checkY] == false)
            command->vel.y += -multiplier * frametime;
    }

    if(command->key & KEY_DOWN)
    {
        checkX = (int) (command->origin.x / 32.0f);
        checkY = (int) ((command->origin.y + multiplier * frametime) /
            32.0f);

        if(mapdata[checkX][checkY] == false)
            command->vel.y += multiplier * frametime;
    }
}

```



```

    }

    if(command->key & KEY_LEFT)
    {
        checkX = (int) ((command->origin.x - multiplier * frametime) /
                        32.0f);
        checkY = (int) (command->origin.y / 32.0f);

        if(mapdata[checkX][checkY] == false)
            command->vel.x += -multiplier * frametime;
    }

    if(command->key & KEY_RIGHT)
    {
        checkX = (int) ((command->origin.x + multiplier * frametime) /
                        32.0f);
        checkY = (int) (command->origin.y / 32.0f);

        if(mapdata[checkX][checkY] == false)
            command->vel.x += multiplier * frametime;
    }
}

```

CalculateHeading Function

This function is used to convert the player's velocity to simple heading values (in degrees).

```

void CArmyWar::CalculateHeading(command_t *command)
{
    // Right
    if((command->vel.x > 0.0f) &&
        (command->vel.y == 0.0f))
    {
        command->heading = EAST;
    }

    // Left
    if((command->vel.x < 0.0f) &&
        (command->vel.y == 0.0f))
    {
        command->heading = WEST;
    }

    // Down
    if((command->vel.y > 0.0f) &&
        (command->vel.x == 0.0f))
    {
        command->heading = SOUTH;
    }

    // Up
    if((command->vel.y < 0.0f) &&
        (command->vel.x == 0.0f))

```

```
{
    command->heading = NORTH;
}

// Down-Right
if((command->vel.x > 0.0f) &&
    (command->vel.y > 0.0f))
{
    command->heading = SOUTHEAST;
}

// Up-Right
if((command->vel.x > 0.0f) &&
    (command->vel.y < 0.0f))
{
    command->heading = NORTHEAST;
}

// Down-Left
if((command->vel.x < 0.0f) &&
    (command->vel.y > 0.0f))
{
    command->heading = SOUTHWEST;
}

// Up-Left
if((command->vel.x < 0.0f) &&
    (command->vel.y < 0.0f))
{
    command->heading = NORTHWEST;
}
}
```

PredictMovement Function

This function produces the local frames, or in other words, predicts player movement. It produces a frame and hopes it is the same as what the server comes up with (and it will be the same if all the data reaches the server and the client).

What the function does is take the previous frame and then add current velocities to the previous positions. And that is mostly it. It also updates some other variables, like the bullet lifetime, but it is updated the same way as the positions: You take the previous lifetime value and add current frametime.

```
void CAmyWar::PredictMovement(int prevFrame, int curFrame)
{
    if(!localClient)
        return;

    float frametime = inputClient.frame[curFrame].msec / 1000.0f;
```

```

localClient->frame[curFrame].key = inputClient.frame[curFrame].key;

//
// Player ->
//

// Process commands
CalculateVelocity(&localClient->frame[curFrame], frametime);
CalculateHeading(&localClient->frame[curFrame]);

// Calculate new predicted origin
localClient->frame[curFrame].predictedOrigin.x = localClient->
    frame[prevFrame].predictedOrigin.x + localClient->
    frame[curFrame].vel.x;

localClient->frame[curFrame].predictedOrigin.y = localClient->
    frame[prevFrame].predictedOrigin.y + localClient->
    frame[curFrame].vel.y;

// Copy values to "current" values
localClient->command.predictedOrigin.x = localClient->
    frame[curFrame].predictedOrigin.x;
localClient->command.predictedOrigin.y = localClient->
    frame[curFrame].predictedOrigin.y;
localClient->command.vel.x = localClient->frame[curFrame].vel.x;
localClient->command.vel.y = localClient->frame[curFrame].vel.y;
localClient->command.heading = localClient->frame[curFrame].heading;

//
// Bullet ->
//

// First set the previous values
localClient->frame[curFrame].bullet.shot = localClient->
    frame[prevFrame].bullet.shot;
localClient->frame[curFrame].bullet.vel.x = localClient->
    frame[prevFrame].bullet.vel.x;
localClient->frame[curFrame].bullet.vel.y = localClient->
    frame[prevFrame].bullet.vel.y;
localClient->frame[curFrame].bullet.lifetime = localClient->
    frame[prevFrame].bullet.lifetime;

// The bullet is carried by the player
if(localClient->frame[curFrame].bullet.shot == false)
{
    localClient->frame[curFrame].bullet.predictedOrigin.x =
        localClient->frame[curFrame].predictedOrigin.x;
    localClient->frame[curFrame].bullet.predictedOrigin.y =
        localClient->frame[curFrame].predictedOrigin.y;
}
else
{

```

```

localClient->frame[curFrame].bullet.lifetime += (int) (frametime
    * 1000.0f);

if(localClient->frame[curFrame].bullet.lifetime > 2000)
{
    localClient->frame[curFrame].bullet.shot = false;
    localClient->frame[curFrame].bullet.lifetime = 0;

    localClient->frame[curFrame].bullet.predictedOrigin.x =
        localClient->frame[curFrame].predictedOrigin.x;
    localClient->frame[curFrame].bullet.predictedOrigin.y =
        localClient->frame[curFrame].predictedOrigin.y;
}
}

// Calculate the heading for the bullet only when firing
if(localClient->frame[curFrame].key & KEY_WEAPON && localClient->
    frame[curFrame].bullet.shot == false)
{
    CalculateBulletVelocity(&localClient->frame[curFrame]);
}

// If the bullet is in the air (shot), update its origin
if(localClient->frame[curFrame].bullet.shot)
{
    localClient->frame[curFrame].bullet.predictedOrigin.x =
        localClient->frame[prevFrame].bullet.predictedOrigin.x +
        localClient->frame[curFrame].bullet.vel.x * frametime;

    localClient->frame[curFrame].bullet.predictedOrigin.y =
        localClient->frame[prevFrame].bullet.predictedOrigin.y +
        localClient->frame[curFrame].bullet.vel.y * frametime;
}

// Copy values to "current" values
localClient->command.bullet.predictedOrigin.x = localClient->
    frame[curFrame].bullet.predictedOrigin.x;
localClient->command.bullet.predictedOrigin.y = localClient->
    frame[curFrame].bullet.predictedOrigin.y;
localClient->command.bullet.vel.x = localClient->
    frame[curFrame].bullet.vel.x;
localClient->command.bullet.vel.y = localClient->
    frame[curFrame].bullet.vel.y;
}

```

MoveObjects Function

This function moves all the remote players and the local player to their new positions. Then the map can be drawn to show the player positions. The remote players are moved by adding their last known velocity to their current origin. The server tells us the real origin in time, and there may be some warping if the client's origin differs from the server's origin.

This is one way to do it, but to get smoother movement, you could add dead reckoning or client prediction on the remote players too. But we will leave that to you.

```
void CAmyWar::MoveObjects(void)
{
    if(!localClient)
        return;

    clientData *client = clientList;

    for(; client != NULL; client = client->next)
    {
        // Remote players
        if(client != localClient)
        {
            CalculateVelocity(&client->command, frametime);
            CalculateHeading(&client->command);

            client->command.origin.x += client->command.vel.x;
            client->command.origin.y += client->command.vel.y;

            client->command.bullet.origin.x += client->
                serverFrame.bullet.vel.x * frametime;
            client->command.bullet.origin.y += client->
                serverFrame.bullet.vel.y * frametime;
        }

        // Local player
        else
        {
            client->command.origin.x = client->
                command.predictedOrigin.x;
            client->command.origin.y = client->
                command.predictedOrigin.y;

            client->command.bullet.origin.x = client->
                command.bullet.predictedOrigin.x;
            client->command.bullet.origin.y = client->
                command.bullet.predictedOrigin.y;
        }
    }
}
```

lobby.cpp File

Now that we have the game data structures, we can implement the functions we introduced in Tutorial 4.

RefreshGameList Function

This function refreshes the game list in the lobby dialog. First the contents of the list are completely removed, and then everything is added again to match the current game list. If a game is in progress, text is added after the game's name to inform us of that.

```
void CLobby::RefreshGameList(void)
{
    char temp[128];

    SendMessage(GetDlgItem(hWnd_LobbyDialog, IDC_GAMELIST),
                LB_RESETCONTENT, 0, 0);

    CArmyWar *list = gameList;

    for( ; list != NULL; list = list->next)
    {
        strcpy(temp, list->GetName());

        if(list->GetInProgress())
        {
            strcat(temp, " (in progress)");
        }

        SendMessage(GetDlgItem(hWnd_LobbyDialog, IDC_GAMELIST),
                    LB_ADDSTRING, 0, (LPARAM) temp);
    }
}
```

RefreshJoinedPlayersList Function

Here we refresh the list of joined players. The list is first reset so that it can be filled again with the current information. Note that this function updates the list for both the game host and the players who joined the game.

```
void CLobby::RefreshJoinedPlayersList(void)
{
    if(localGame == NULL)
        return;

    SendMessage(GetDlgItem(hWnd_CreateViewPlayersDialog,
                            IDC_PLAYERSINGAME), LB_RESETCONTENT, 0, 0);
```

```
SendMessage(GetDlgItem(hWnd_JoinGameDialog,
                      IDC_JOINPLAYERSINGAME), LB_RESETCONTENT, 0, 0);

clientData *list = localGame->GetClientList();

for( ; list != NULL; list = list->next)
{
    SendMessage(GetDlgItem(hWnd_CreateViewPlayersDialog,
                          IDC_PLAYERSINGAME), LB_ADDSTRING, 0,
                (LPARAM) list->nickname);

    SendMessage(GetDlgItem(hWnd_JoinGameDialog,
                          IDC_JOINPLAYERSINGAME), LB_ADDSTRING, 0,
                (LPARAM) list->nickname);
}
}
```

Other Unimplemented Functions

The other unimplemented functions (`AddGame`, `RemoveGame`, and `RemoveGames`), work the same way as they do on the server side, so there is no need to explain them here. Only the `AddGame` function is slightly different, as it has some new parameters. These parameters only contain game information, such as the index number and “in-progress” flag. These parameters do not affect the functionality of the function; they are only stored in the game’s structure.

Summary

All good things must come to an end, and so ends this tutorial. Here we covered the final version of the Army War v2.0 server and client. We learned how to move the players and other objects in the game world, not just by moving them on our local screen but also by moving them on the server. We learned that the server is always right, not the client. We also now know that by using frame time to calculate the amount of movement in one frame, we can keep all the clients and the server in sync.

Now that the 2D Army War game has been fully covered in this tutorial and once you understand the concept of multiplayer game programming, you can put your skills to the test by programming the 3D Army War engine (located on the companion CD) to include multiplayer code. Try to understand which parts are essential to make the code work on any game, and you will be able to make a 3D multiplayer game. All you need is patience and time. Good luck!

Index

* indicates the reference includes code listing.

2D library,
 adding to workspace, 183
 creating windowed application with,
 184-189*
 graphical functions, 190-193
2D library example,
 loading graphics, 200-202*
 moving primitives, 195-199*
 rotating graphics, 200-202*
2D positions, displaying, 190

A

accept function, 128
account, creating, 316-319*
AddClient function, 293-295*, 333-334*,
 366-367*
AddGame function, 478-479*
address classes, 107
address conversion, 133
 functions, 133-134
address structures,
 generic, 123
 IPv4, 121-122
 IPv6, 122
AddSequences function, 255*
adduser.tpl.html, 97*
alias naming restrictions, 20
ALTER TABLE command, 26-28
Apache web server,
 installing PHP4 on, 60-63
 setting up, 57-60
application,
 creating, 184-189*
 creating with dreamSock, 305-369*
application layer, 105
ApplicationProc function, 489*
arrays, using in PHP, 70

B

BeginReading function, 257*
big-endian order, 136
bind function, 125-126
blocking I/O, 169-170
blocking socket, 144
Broadcast function, 176-177*
broadcasting, 175
BuildDeltaMoveCommand function,
 454-458*, 508-509*
BuildMoveCommand function, 453-454*
bullets,
 calculating velocity of, 470-472*
 comparing, 534-535*
 moving, 472-475*
byte ordering, 136

C

CalculateBulletVelocity function, 470-472*
CalculateHeading function, 469-470*,
 536-537*
CalculateVelocity function, 468-469*,
 535-536*
CArmyWar class,
 constructor, 525-526*
 destructor, 525-526*
 initializing, 525-526*
 uninitializing, 525-526*
CArmyWarServer class methods, 437-449*,
 458-467*
CheckBulletPredictionError function,
 534-535*
CheckFlagCollisions function, 475-476*
CheckForTimeout function, 299-301*
CheckKeys function, 532-533*
CheckPredictionError function, 533-534*
CheckVictory function, 531-532*

Clear function, 253*
 client application, creating, 306-338*
 client list, 208
 returning pointer to player in, 532*
 client prediction, 429
 client.cpp, 511-525*
 client.h, 482-487*
 ClientDlgProc callback procedure, 7-8
 ClientProcess function
 TCP, 155-157
 UDP, 159-160
 clients, 135-136
 adding to client list, 293-295*,
 333-334*, 366-367*
 checking for timing out, 299-301*
 connecting with server, 126
 creating, 150, 481-542*
 creating for game lobby, 371-405*
 creating TCP, 151-153*
 creating UDP, 157-159*
 initializing, 268-269*
 keeping alive, 336*
 pinging, 293*
 receiving packets from, 301-303*
 removing from client list, 295-297*,
 334-336*, 366-368
 resetting, 270*
 sending messages to, 290-293*
 uninitializing, 269*
 CLobby class methods, 389-398*
 CLobbyServer class methods, 408-418*
 close function, 129
 closesocket function, 129
 code, structuring, 73-78
 colors, specifying, 190
 column naming restrictions, 20
 commands,
 building, 453-458*
 reading, 452-453*, 505-508*
 sending, 451-452*, 502-504*
 common.h, 310*, 340*, 488
 concurrent server, 112, 135
 conditional statements,
 using in PHP, 68-70
 using in SQL, 37-39
 connect function, 126
 Connect function, 337*, 403*, 504-505*
 connected socket, 128

connection, starting, 502*
 connectionless protocol, 109
 connection-oriented protocol, 108
 core.php, 74*, 76*
 CREATE DATABASE command, 20-21
 CREATE TABLE command, 25
 CreateAccountDialogProc function,
 316-319*
 CreateGameDialogProc function, 380-381*
 CreateThread function, 168-169
 CreateViewPlayersDialogProc function,
 380*
 CSigin class, 308-309*
 constructor, 329*
 destructor, 329*
 global variables, 316
 methods, 323-329*
 CSiginServer class, 339-340*
 constructor, 361*
 destructor, 361*
 global variables, 347
 methods, 355-361*

D

daemon, 18
 daemonInit function, 351-352*
 data,
 adding to table, 30-32, 41
 deleting from table, 34-35
 importing, 44-46
 linking, 42-43
 modifying in table, 32-34
 reading, 257-260*
 receiving, 243-244*
 receiving in socket, 131-133
 relating, 42-43
 removing, 84-90*
 requesting, 401*
 retrieving, 50-53*
 sending, 208-209, 244-245*
 sending to socket, 129-132
 storing, 79-84*
 updating, 53-55*, 84-90*
 writing, 255*, 257*
 data buffer,
 dumping, 270*
 retrieving point from, 254*
 writing to, 255*

- data definition language, 19
 - using, 19-29
 - data input, 44-46
 - data link layer, 105
 - data manipulation language, 19, 29
 - using, 29-39
 - data types,
 - and platforms, 121
 - defining, 212-213
 - in MySQL, 22-23
 - database,
 - adding table to, 25
 - backing up to file, 47-49
 - connecting to, 50-53
 - connecting to with PHP script, 78-79*
 - creating, 20-21
 - creating relational, 40
 - dropping, 21-22
 - dropping table from, 28-29
 - modifying table in, 26-28
 - naming restrictions, 20
 - relational, 40
 - removing data from, 84-90*
 - restoring, 49
 - storing data in, 79-84*
 - updating data in, 84-90*
 - viewing, 19-20
 - database, relational, 40 *see also* database
 - datagram socket, 111-112
 - dead reckoning, 429
 - debug version, 12
 - DELETE command, 34-35
 - delta compression, 428-429
 - DESCRIBE command, 26
 - DialogBox function, 7
 - dialogs, *see also* windows
 - creating, 6-7*
 - creating for game lobby, 372-377
 - Disconnect function, 337*, 403-404*, 505*
 - doAddUser function, 84
 - Domain Name Service, 106
 - DrawMap function, 528-530*
 - dreamClient class, 226-228*
 - constructor, 268*
 - destructor, 268*
 - functionality of, 262-268*
 - variables, 260-262
 - dreamConsole
 - constructor, 215
 - destructor, 215
 - dreamMessage class, 225-226*
 - functionality of, 249-253*
 - variables, 249
 - dreamServer class, 228-229*
 - constructor, 289*
 - destructor, 289-290*
 - functionality of, 280-288*
 - variables, 280
 - dreamsock, 206
 - dreamSock, creating network application
 - with, 305-369*
 - dreamSock application, planning
 - functionality of, 306
 - dreamSock network library, creating, 248-303*
 - dreamSock.h, 220-225*
 - dreamSock_Broadcast function, 245-246*
 - dreamSock_CloseSocket function, 243*
 - dreamSock_GetCurrentSystemTime function, 246*
 - dreamSock_GetPacket function, 243-244*
 - dreamSock_Initialize function, 229-230*
 - dreamSock_InitializeWinSock function, 230-231*
 - dreamSock_Linux_GetCurrentSystemTime function, 246-247*
 - dreamSock_OpenUDPSocket function, 241-243*
 - dreamSock_SendPacket function, 244-245*
 - dreamSock_SetBroadcasting function, 239-240*
 - dreamSock_SetNonBlocking function, 239*
 - dreamSock_Shutdown function, 231-232*
 - dreamSock_Socket function, 238-239*
 - dreamSock_StringToSockaddr function, 240*
 - dreamSock_Win_GetCurrentSystemTime function, 247-248*
- DROP DATABASE command, 21-22
- DROP TABLE command, 28-29
- DumpBuffer function, 270*
- E**
- echo application, running, 160-161
 - echo TCP client, creating, 151-153*
 - echo TCP server, creating, 139-142*

echo UDP client, creating, 157-159*
 echo UDP server, creating, 146-150*
 engine,
 initializing, 526-527*
 shutting down, 527*
 ephemeral ports, 110
 errno variable, using to retrieve error
 number, 248
 error handling, 120
 error values, retrieving, 248
 event object, 166-167
 events,
 detecting network, 163
 processing, 186-187
 Excel, using to enter data, 45-46
 exceptions, catching, 306

F

FastTemplate, 90
 converting to, 95-101
 using, 91-95*
 flag collisions, checking for, 475-476*
 footer.php, 75-76*
 Frame function, 477-478*, 530-531*
 frame history, 430
 frame time, monitoring, 428
 frames,
 comparing, 533-534*
 running, 477-478*
 storing, 430
 ft.php, 91-92*
 ft2.php, 94*
 ftcore.php, 98*
 ftusers.php, 99-101*
 functions,
 global, 229-248*
 graphical, 190-193
 using in PHP, 70-71

G

game,
 adding to game list, 478-479*
 creating log system for, 214-219*
 removing from game list, 479-481*
 game client, creating, 481-542*
 game list,
 adding game to, 478-479*
 refreshing, 541*
 removing game from, 479-481*

game lobby,
 creating, 371-425*
 creating dialogs for, 372-377,
 380-386*
 game logic, handling, 530-531*
 game map,
 drawing, 528-530*
 generating, 467-468*
 game platform, choosing, 113
 game server,
 creating, 430-481*
 reasons for creating, 15
 game tutorial code
 client.cpp, 511-525*
 client.h, 482-487*
 common.h, 310*, 340*, 488
 dreamSock.h, 220-225*
 lobby.cpp, 389-398*, 408-418*,
 478-481*, 541-542*
 lobby.h, 377-378*, 406-407*
 main.cpp, 310-316*, 341-347*,
 380-389*, 407-408*, 436-437*,
 488-490*
 main.h, 309*, 379*
 network.cpp, 437-449*, 490-502*
 network.h, 309*, 340*, 379*, 407*,
 436*, 487-488*
 server.cpp, 458-467*
 server.h, 431-435*
 signin.cpp, 323-329*, 355-361*
 signin.h, 307-308*, 338-339*
 GenerateRandomMap function, 467-468*
 GET method, 73
 GetClientPointer function, 532*
 GetNewPoint function, 254*
 GetPacket function, 275-276*, 301-303*
 getsockopt function, 173-174
 GFX_Blit function, 192-193
 GFX_Line function, 191
 GFX_LoadBitmap function, 192
 GFX_Pixel function, 191
 GFX_Rect function, 191
 GFX_RectFill function, 191
 GFX_Tri function, 191
 GFX_TriFill function, 191
 global functions, 229-248*
 graphical functions, 190-193

- graphics,
 - loading, 200-202*
 - rotating, 200-202*
- graphics engine,
 - initializing, 526-527*
 - shutting down, 527*
- H**
- header files, setting up for network library, 219-220*
- header.php, 75*
- heading, calculating, 469-470*
- hosts,
 - identifying, 207-208
 - sending data to, 208-209
- HWND, 4
- I**
- I/O, controlling, 172
- I/O strategies, 169
- images,
 - displaying, 192-193
 - loading, 192, 200-202*
 - rotating, 200-202*
- Include folder, 10
- index.php, 64*
- index2.php, 65*
- index3.php, 67*
- index4.php, 68-69*
- index5.php, 70*
- inet_aton function, 134
- Init function, 253*
- initialization, Windows, 115-120
- Initialize function, 268-269*, 290*
- InitializeEngine function, 526-527*
- InitializeWinSock function, 117-119*
- InitNetwork function, 362*, 449*
- InitSockets function
 - TCP, 142-145, 154-155
 - UDP, 148-149, 159
- input,
 - checking for, 532-533*
 - handling, 489*
- input.php, 71-72*
- input/output functions, 172-175
- INSERT command, 30-32
- International Organization for Standardization, *see* ISO
- Internet Protocol, *see* IP
- ioctl function, 172-173
- ioctlsocket function, 172-173
- IP, 106-108
 - address classes, 107
 - addresses, 107
- IPv4, 106-107
 - address structure, 121-122
- IPv6, 107-108
 - address structure, 122
- ISO, 104
- iterative server, 112, 134
- J**
- JoinGameDialogProc function, 382*
- K**
- key, checking for status of, 352*
- keyboard input, 193-195
 - checking for, 532-533*
 - handling, 489*
- keyPress function, 352*
- KillPlayer function, 532*
- L**
- LAN, 136
- LAN server, searching for, 175-176
- latency, calculating, 209
- Lib folder, 10
- LIKE command, 39
- line, drawing, 191
- link table, 40
 - creating, 40-41
- Linux, building network library in, 211-212
- listen function, 127
- little-endian order, 136
- LOAD command, 44
- lobby.cpp, 389-398*, 408-418*, 478-481*, 541-542*
- lobby.h, 377-378*, 406-407*
- LobbyDialogProc function, 384-386*
- local area network, *see* LAN
- log system, 213-214
 - creating, 214-219*
- LoginDialogProc function, 382-383*
- LogString function, 217-218*
- loops, using in PHP, 67-68

M

macros, socket descriptor, 164-165
 main function, 353-355*
 TCP, 153-154
 main.cpp, 310-316*, 341-347*, 380-389*,
 407-408*, 436-437*, 488-490*
 main.h, 309*, 379*
 main.tpl.html, 96*
 mainbody.tpl.html, 93*
 map,
 drawing, 528-530*
 generating, 467-468*
 message loop, creating, 184-185*
 messages,
 building, 508-509*
 compressing, 428-429
 functions for working with, 271-279*
 handling, 430
 notification, 366*
 parsing, 297-299*, 329-333*
 reading, 362-366*, 399-401*,
 418-425*, 449-451*
 sending, 402-403*, 504-505*
 sending to clients, 303*
 sending to server, 336*
 system vs. user, 270-271
 update, 8-9
 messaging system, in Windows, 3-4
 MoveObjects function, 539-540*
 MovePlayer function, 472-475*
 multicast IP addresses, 107
 multiplayer, 134
 multiple templates, using, 93-95*
 multiplexing, 171
 multiplexing I/O, 171
 multitasking, 4
 multithreading, 4, 167-168
 MySQL, 15
 connecting to database with, 78-79*
 data types, 22-23
 installing, 16-19
 storing data with, 79-84*
 using, 78-90*
 MySQL C++ interface, *see* MySQL++
 MySQL example,
 connecting to database and retrieving
 data, 50-53*

 updating data from an application,
 53-55*

MySQL server, running, 19

MySQL++, 50

mysql1.php, 79*

mysqlcore.php, 80*, 85*

mysqldump utility, 47

N

NET_WinSockInitialize function, TCP, 142
 network,

 initializing, 449*

 running, 509-510*

network application, creating with
 dreamSock, 305-369*

network byte order, 136

network events, detecting, 163

network functionality, designing, 428-430

network latency, calculating, 209

network layer, 105

network library,

 building, 209-212

 creating dreamSock, 248-303*

 initializing, 229-230*

 planning, 206-207

 reasons for creating, 205-206

 setting up files for, 219-220*

 shutting down, 231-232*

network protocol, *see* protocol

network system, running, 337-338*, 369*,
 404-405*

network.cpp, 437-449*, 490-502*

network.h, 309*, 340*, 379*, 407*, 436*,
 487-488*

non-blocking I/O, 170

non-blocking socket, 144

O

Open Systems Interconnection model, *see*
 OSI model

OpenGL libraries, adding to workspace,
 183

operators, using in PHP, 67-68

OSI model, 104

 layers, 104-105

 usage of, 105

output.php, 72*

P

page1.php, 77
 page2.php, 77
 ParsePacket function, 273-275*, 297-299*
 PHP,
 arrays in, 70
 conditional statements in, 68-70
 connecting to database with, 78-79*
 functions in, 70-71
 operators and loops in, 67-68
 testing installation of, 62
 updating data with, 84-90*
 user input in, 71-73
 using, 63-73
 variables in, 65-66
 PHP4, installing on Apache web server,
 60-63
 physical layer, 105
 pinging, 209
 pixel, plotting, 191
 platforms,
 and data types, 121, 212-213
 and sockets, 113
 and sockets API, 112-113
 choosing, 113
 Unix, 113
 Windows, 113
 player,
 calculating heading of, 536-537*
 calculating velocity of, 535-536*
 moving, 472-475*, 539-540*
 moving back to start position, 532*
 predicting movement of, 537-539*
 returning pointer to, 532*
 player list,
 refreshing, 541-542*
 updating, 398-399*
 port numbers, 109
 ports, 109-110
 POST method, 73
 PredictMovement function, 537-539*
 presentation layer, 105
 primary key, 40
 primitives, moving, 195-199*
 println function, 215-216*
 program, running as daemon, 351-352*
 project,
 adding static libraries to, 183

 creating, 182-189
 protocol, 103-104
 connectionless, 109
 connection-oriented, 108
 pthread_create function, 169

R

Read function, 257-258*
 ReadByte function, 258*
 ReadDeltaMoveCommand function,
 452-453*, 506-508*
 ReadFloat function, 259*
 ReadLong function, 259*
 ReadMoveCommand function, 505-506*
 ReadPackets function, 329-333*, 362-366*,
 399-401*, 418-425*, 449-451*
 ReadShort function, 258-259*
 ReadString function, 259-260*
 rectangle, drawing, 191
 recv function, 131-132
 recvfrom function, 133
 RefreshGameList function, 541*
 RefreshJoinedPlayersList function,
 541-542*
 RefreshPlayerList function, 398-399*
 relational database, 40-41 *see also* database
 release version, 12
 RemoveClient function, 295-297*,
 334-335*, 367-368*
 RemoveClients function, 335-336*, 368*
 RemoveGame function, 479-480*
 RemoveGames function, 481*
 RequestGameData function, 401*
 Reset function, 270*
 RunNetwork function, 337-338*, 369*,
 404-405*, 509-510*

S

select function, 163-164
 SELECT statements,
 conditional, 37-39
 using, 35-39
 send function, 129-131
 SendAddClient function, 290-292*
 SendChat function, 402*
 SendCommand function, 451-452*,
 502-504*
 SendConnect function, 271-272*
 SendCreateGame function, 402*

- SendDisconnect function, 272*
- SendDlgItemMessage function, 9
- SendExitNotification function, 366*
- SendKeepAlive function, 336*
- SendPacket function, 276-279*
- SendPackets function, 303*
- SendPing function, 272-273*, 293*
- SendRemoveClient function, 292-293*
- SendRemoveGame function, 402*
- SendRequestNonDeltaFrame function, 504*
- SendSignIn function, 336*
- SendStartGame function, 403*, 504*
- sendto function, 132
- server application, creating, 338-369*
- server.cpp, 458-467*
- server.h, 431-435*
- ServerProcess function
 - TCP, 145-146*
 - UDP, 149-150
- servers, 134-135
 - connecting to, 337*, 403*
 - connecting with client, 126
 - creating, 136-137, 430-481*
 - creating for game lobby, 406-425*
 - creating TCP, 139-142*
 - creating UDP, 146-150*
 - disconnecting from, 337*, 403-404*, 527*
 - initializing, 362*
 - reasons for storing game data on, 15
 - searching for, 175-176
 - types of, 112, 134-135
 - uninitializing, 362*
- session layer, 105
- setsockopt function, 173-174
- SHOW DATABASES command, 19-22
- SHOW TABLES command, 25-26
- shutdown function, 174-175
- Shutdown function, 527*
- ShutdownNetwork function, 362*
- signal-driven I/O, 170-171
- sign-in system, creating, 306-369*
- signin.cpp, 323-329*, 355-361*
- signin.h, 307-308*, 338-339*
- simple.tpl.html, 91*
- skeleton project, creating, 182-189
- socket descriptor macros, 164-165
- socket function, 124
- socket functions, global, 232-248*
- sockets, 110-111
 - addresses of, 112
 - binding address to, 125-126
 - blocking, 144
 - broadcasting, 245-246*
 - closing, 129, 243*, 290*
 - connected, 128
 - controlling I/O mode of, 172-173
 - converting address string of, 240*
 - creating, 124, 238-239*, 290*
 - data types of, 121
 - disabling, 174-175
 - getting options for, 173-174
 - initializing, 142-145, 148-149, 154-155, 159
 - non-blocking, 144
 - opening UDP, 241-243*
 - receiving data in, 131-133
 - sending data to, 129-132
 - setting blocking mode of, 239*
 - setting for incoming connections, 127
 - setting options for, 173-174
 - setting to broadcast, 239-240*
 - types of, 111-112
- sockets API, 110
 - and platforms, 112-113
- sockets functions,
 - basic, 124-129
 - input/output, 129-133
- source files,
 - adding to project, 183
 - setting up for network library, 219-220*
- SQL, 19
- StartConnection function, 502*
- StartLog function, 216-217*
- StartLogConsole function, 214
- static libraries, adding to project, 183
- static link library, 9-10
 - creating, 10-13
 - structure of, 10
 - using, 13-14
- StopLog function, 218-219*
- stream socket, 111
- structured query language, *see* SQL

system messages vs. user messages,
270-271

system time, checking, 246-248

T

tablerow.tpl.html, 94*

tables,

- adding data to, 30-32, 41
- adding to relational database, 40-41
- creating, 24-26
- deleting data from, 34-35
- dropping, 28-29
- modifying, 26-28
- modifying data in, 32-34
- naming restrictions, 20
- viewing columns in, 26

TCP, 106, 108

TCP client, 150-151

- creating, 151-153*

TCP connection,

- accepting incoming, 128
- listening for, 127

TCP server, 137-138

- creating, 139-142*

TCP/IP, 106

templates, 90

- creating, 95-96*
- using, 91-95*
- using multiple, 93-95*

test.php, 62*

text file, using to enter data, 44-45

thread, 134-135

- creating, 168-169

timing out, 209

Transmission Control Protocol, *see* TCP

Transmission Control Protocol/Internet
Protocol, *see* TCP/IP

transport layer, 105, 108

triangle, drawing, 191

typecasting, 123

U

UDP, 106, 109

UDP client, 151

- creating, 157-159*

UDP server, 138-139

- creating, 146-150*

UDP socket, opening, 241-243*

Uninitialize function, 269*, 290*

Unix,

- as game platform, 113
- building network library in, 211-212
- input/output functions, 172-175
- multithreading in, 169
- sockets functions, 124-133

UPDATE command, 32-34

update message, 8-9

USE command, 25

User Datagram Protocol, *see* UDP

user input, in PHP, 71-73

user messages vs. system messages,
270-271

userlist.tpl.html, 96-97*

userlist_row.tpl.html, 97*

users.php, 81-82*, 85-88*

V

variables,

- initializing, 253*
- resetting, 235*
- using in PHP, 65-66

VectorLength function, 488-489*

vectors, working with, 488-489*

VectorSubtract function, 488-489*

velocity, calculating, 468-472*

victory conditions, checking, 531-532*

Visual Studio,

- configuring, 181-182
- creating static link library with,
10-13
- creating window with, 4-6
- finding static link library with, 13-14

W

web server, *see* Apache web server

welcome.php, 74*, 76-77*

well-known ports, 109

WHERE command, 37-39

Win32 functions,

- input/output, 172-175
- sockets, 124-133

windowed application, creating, 184-189*

WindowProc function, 347*

windows, *see also* dialogs

- creating, 4-6
- handle, 4
- updating, 8-9

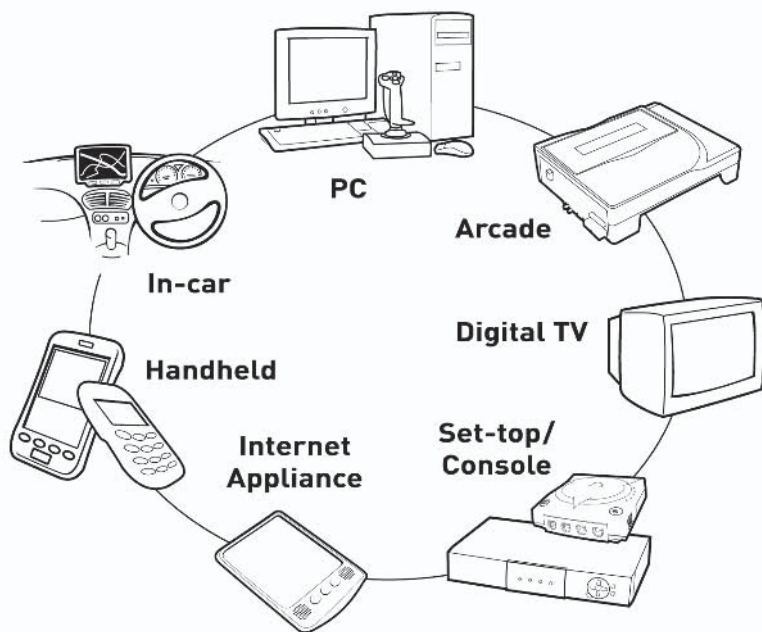
- Windows,
 - as game platform, 113
 - building network library in, 210-211
 - callback procedure, *see* WndProc
 - messaging system, 3-4
 - multithreading in, 167-168
- Windows Sockets, *see* WinSock
- WinMain function, 5, 7-8, 184-185*, 319-323*, 347-351*, 386-389*
- WinSock, 112-113
 - initializing, 115-120
- WinSock API, initializing, 230-231*
- WM_CLOSE event, 186-187
- WM_KEYDOWN event, 187
- WM_KEYUP event, 187
- WM_SIZE event, 187
- WndProc, 5, 185-186*
- workspace,
 - adding static libraries to, 183
 - creating, 182-183
- Write function, 255*
- WriteByte function, 255-256*
- WriteFloat function, 256*
- WriteLong function, 256*
- WriteShort function, 256*
- WriteString function, 257*
- WSAAsyncSelect function, 165
- WSACleanup function, 116
- WSAEnumProtocols function, 117, 120
- WSAEventSelect function, 165-166
- WSAGetLastError function, 120
 - using to retrieve error number, 248
- WSAStartup function, 115-116
- WSAWaitForMultipleEvents function, 166

This page intentionally left blank.



POWERVR

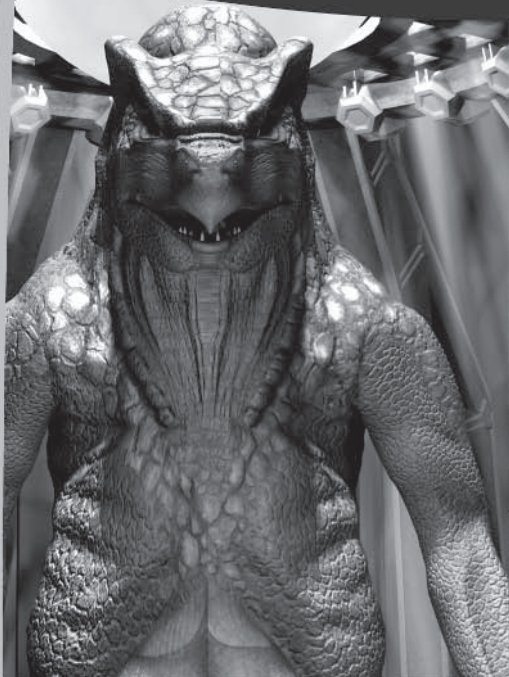
Visionary IP



www.powervr.com

www.pvrdev.com

Get **Deep** into 3D!



DEEP EXPLORATION.

Search, View, Translate, Animate, Render and Publish!

Deep Exploration™ provides easy navigation tools that let you search and view your 2D graphics and 3D models. It gives artists everywhere a production edge, including the ability to quickly translate 2D and 3D file formats with animation included. Deep Exploration creates high quality fast renderings of 3D objects and scenes for use in many graphic applications. It is also a powerful web publication tool, ideal for creating interactive 3D content for web-based presentations.

DEEP PAINT 3D.

3D Painting and Texturing

Deep Paint 3D® provides artists with an intuitive, easy to use tool to paint and texture 3D models interactively in 3D! It uses textures or natural media paints which can be brushed directly or projected onto 3D models and scenes. This creative environment supports an integrated workflow with 3ds max®, Maya®, SoftImage® and LightWave 3D®. Deep Paint 3D comes complete with a bi-directional interface to Photoshop® and special support for the Wacom® Intuos™ pressure sensitive tablet.

DEEP UV.

Ultimate UV Mapping

Deep UV™ is a superior set of tools for the creation and modification of UV mapping for polygonal and sub-division surface models within an interactive 2D and 3D UV mapping environment. Deep UV includes fast and easy to use tools such as soft selection, relax and 3D UV selection. Whether you're a beginner or a professional 3D artist, you'll have the most efficient UV mapping possible with both automatic and advanced features. Deep UV can be used standalone, or with Deep Paint 3D.

RIGHT HEMISPHERE.

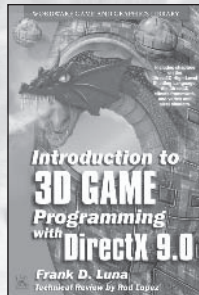
www.righthemisphere.com

Copyright © 2003, Right Hemisphere. All rights reserved.
Master of Orion 3 © 2003, QubeSoft Sony Music © 2003, Hyundai Technologies

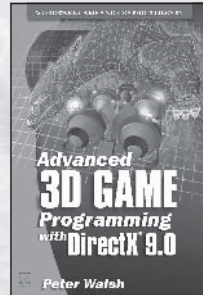
Check out Wordware's market-
featuring the following new

DirectX Game Programming

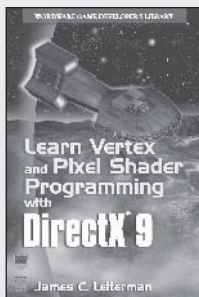
Extensive coverage
on a wide range of
DirectX topics.



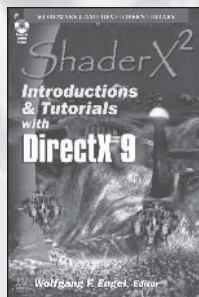
Introduction to 3D Game Programming with DirectX 9.0
1-55622-913-5 • \$49.95
6 x 9 • 424 pp.



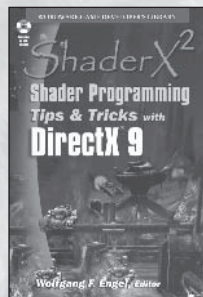
Advanced 3D Game Programming with DirectX 9.0
1-55622-968-2 • \$59.95
6 x 9 • 552 pp.



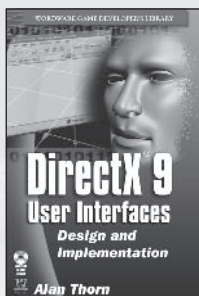
Learn Vertex and Pixel Shader Programming with DirectX 9
1-55622-287-4 • \$34.95
6 x 9 • 304 pp.



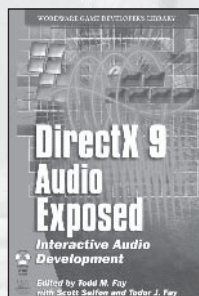
ShaderX²: Introductions & Tutorials with DirectX 9
1-55622-902-X • \$44.95
6 x 9 • 384 pp.



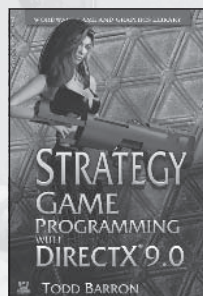
ShaderX²: Shader Programming Tips & Tricks with DirectX 9
1-55622-988-7 • \$59.95
6 x 9 • 728 pp.



DirectX 9 User Interfaces: Design and Implementation
1-55622-249-1 • \$44.95
6 x 9 • 376 pp.



DirectX 9 Audio Exposed: Interactive Audio Development
1-55622-288-2 • \$59.95
6 x 9 • 568 pp.



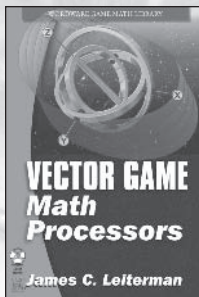
Strategy Game Programming with DirectX 9.0
1-55622-922-4 • \$59.95
6 x 9 • 560 pp.

for more?

leading Game Developer's Library releases and backlist titles.

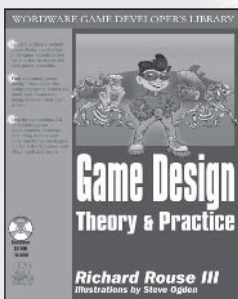


Java 1.4 Game Programming
1-55622-963-1 • \$59.95
6 x 9 • 672 pp.

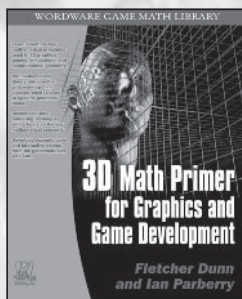


Vector Game Math Processors
1-55622-921-6 • \$59.95
6 x 9 • 528 pp.

Game Programming and Production



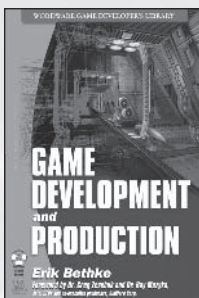
Game Design: Theory and Practice
1-55622-735-3 • \$49.95
7½ x 9¼ • 608 pp.



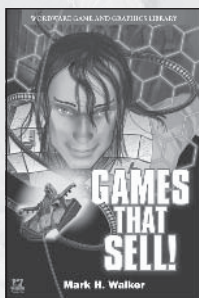
3D Math Primer for Graphics and Game Development
1-55622-911-9 • \$49.95
7½ x 9¼ • 448 pp.



Game Design Foundations
1-55622-973-9 • \$39.95
6 x 9 • 400 pp.



Game Development and Production
1-55622-951-8 • \$49.95
6 x 9 • 432 pp.



Games That Sell!
1-55622-950-X • \$34.95
6 x 9 • 336 pp.



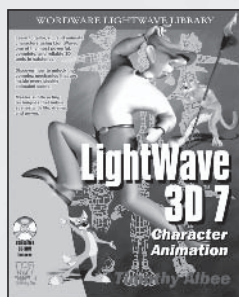
CGI Filmmaking: The Creation of Ghost Warrior
1-55622-227-0 • \$49.95
9 x 7 • 344 pp.

Use the following coupon code for online specials:

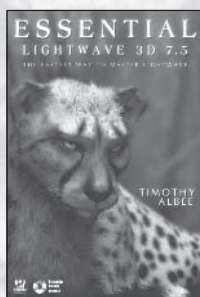
pmg0766

Looking for more?

Check out Wordware's market-leading Game Developer's Library featuring the following new releases, backlist, and upcoming titles.



LightWave 3D 7 Character Animation
1-55622-901-1 • \$49.95
7½ x 9¼ • 360 pp.

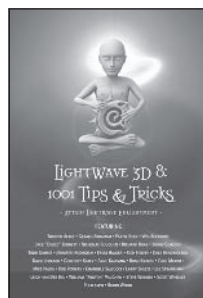


Essential LightWave 3D 7.5
1-55622-226-2 • \$44.95
6 x 9 • 424 pp.

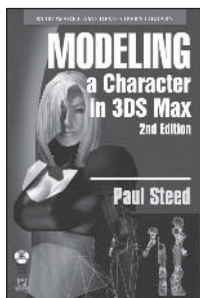


LightWave 3D 7.5 Lighting
1-55622-354-4 • \$69.95
6 x 9 • 496 pp.

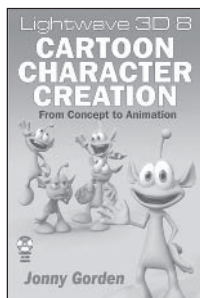
Coming Soon



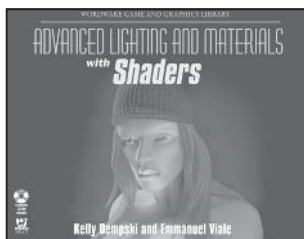
LightWave 3D 8: 1001 Tips & Tricks
1-55622-090-1 • \$39.95
6 x 9 • 500 pp.



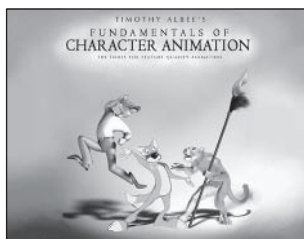
Modeling a Character in 3DS Max (2nd Edition)
1-55622-088-X • \$44.95
6 x 9 • 550 pp.



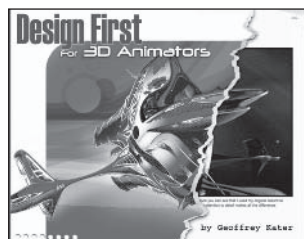
LightWave 3D 8 Cartoon Character Creation
1-55622-083-9 • \$49.95
6 x 9 • 500 pp.



Advanced Lighting and Materials with Shaders
1-55622-292-0 • \$59.95
9 x 7 • 500 pp.



Fundamentals of Character Animation
1-55622-248-3 • \$49.95
9 x 7 • 400 pp.



Design First for 3D Animators
1-55622-085-5 • \$49.95
9 x 7 • 350 pp.

Visit us online at www.wordware.com for more information.

Use the following coupon code for online specials: **pmg0766**

About the CD

The companion CD contains all the source code from the book as well as many useful tools and applications discussed in the book.

The CD will autorun when you insert it in your CD drive. Click the Continue button on the page that appears to view the contents. (If the CD does not autorun, simply use Windows Explorer to browse the CD.)

The directories and their contents are:

- Software — MySQL for Linux and Windows, DBI modules for Perl, WinZip 8.0 SHAREWARE EVALUATION version (not the registered version), the Apache package, the FastTemplate class, and Adobe Acrobat Reader
- Source — Source code from the book
- Libraries — dreamSock, OpenGL, 2DLIB, 3DLIB, and MySQL++

CD/Source Code Usage License Agreement

Please read the following CD/Source Code usage license agreement before opening the CD and using the contents therein:

1. By opening the accompanying software package, you are indicating that you have read and agree to be bound by all terms and conditions of this CD/Source Code usage license agreement.
2. The compilation of code and utilities contained on the CD and in the book are copyrighted and protected by both U.S. copyright law and international copyright treaties, and is owned by Wordware Publishing, Inc. Individual source code, example programs, help files, freeware, shareware, utilities, and evaluation packages, including their copyrights, are owned by the respective authors.
3. No part of the enclosed CD or this book, including all source code, help files, shareware, freeware, utilities, example programs, or evaluation programs, may be made available on a public forum (such as a World Wide Web page, FTP site, bulletin board, or Internet news group) without the express written permission of Wordware Publishing, Inc. or the author of the respective source code, help files, shareware, freeware, utilities, example programs, or evaluation programs.
4. You may not decompile, reverse engineer, disassemble, create a derivative work, or otherwise use the enclosed programs, help files, freeware, shareware, utilities, or evaluation programs except as stated in this agreement.
5. The software, contained on the CD and/or as source code in this book, is sold without warranty of any kind. Wordware Publishing, Inc. and the authors specifically disclaim all other warranties, express or implied, including but not limited to implied warranties of merchantability and fitness for a particular purpose with respect to defects in the disk, the program, source code, sample files, help files, freeware, shareware, utilities, and evaluation programs contained therein, and/or the techniques described in the book and implemented in the example programs. In no event shall Wordware Publishing, Inc., its dealers, its distributors, or the authors be liable or held responsible for any loss of profit or any other alleged or actual private or commercial damage, including but not limited to special, incidental, consequential, or other damages.
6. One (1) copy of the CD or any source code therein may be created for backup purposes. The CD and all accompanying source code, sample files, help files, freeware, shareware, utilities, and evaluation programs may be copied to your hard drive. With the exception of freeware and shareware programs, at no time can any part of the contents of this CD reside on more than one computer at one time. The contents of the CD can be copied to another computer, as long as the contents of the CD contained on the original computer are deleted.
7. You may not include any part of the CD contents, including all source code, example programs, shareware, freeware, help files, utilities, or evaluation programs in any compilation of source code, utilities, help files, example programs, freeware, shareware, or evaluation programs on any media, including but not limited to CD, disk, or Internet distribution, without the express written permission of Wordware Publishing, Inc. or the owner of the individual source code, utilities, help files, example programs, freeware, shareware, or evaluation programs.
8. You may use the source code, techniques, and example programs in your own commercial or private applications unless otherwise noted by additional usage agreements as found on the CD.

WARNING By opening the CD package, you accept the terms and conditions of the CD/Source Code Usage License Agreement. Additionally, opening the CD package makes this book nonreturnable.