10 توابع در زبان C

در زبان C به هر زيربرنامه، يك **تابع** گفته مي شود. يك تابع، تكه برنامه اي است كه داده يا داده هايي را بعنوان ورودي دريافت، و داده يا داده هايي را بعنوان خروجي باز مي گرداند. در زبان C، هر برنامه از يك يا چند تابع تشكيل مي گردد، كه يكي از آنها بايد بنام main ناميده گردد و برنامه از اين تابع شروع خواهد گرديد. تابع main مي تواند ساير توابع را فراخواني نمايد و هر يك از اين توابع نيز مي توانند به نوبه خود، توابع ديگر را فراخواني نمايند. نكته جالب اينجاست كه تابع فراخواننده نيازي به دانستن نحوه كار تابعي كه فراخواني مي كند، ندارد و تنها بايد از نحوه فراخواني و مقدار خروجي آن آگاه باشد. اين نحوه پنهانسازي جزئيات پياده سازي، نقش بسيار مهمي در مهندسي نرم افزار دارد.
تاكنون از توابع كتابخانه هاي استاندارد C استفاده نموده ايم. كتابخانه استاندارد C مجموعه اي از توابع و نوع داده ها است كه براي انجام عملياتي كه عموما مورد نياز برنامه نويسان است، طراحي شده و همراه كامپايلر در اختيار برنامه نويسان قرار داده شده است. بعنوان مثال توابع ورودي/خروجي مانند printf و scanf و يا توابع رياضي مانند sqrt و sin كه توسط بسياري از برنامه نويسان مورد استفاده قرار مي گيرند. براي استفاده از اين توابع بايد فايل سرامد مربوطه را كه تعاريف اوليه (prototype) اين توابع در آنها آمده است را توسط دستور #include در آغاز برنامه گنجاند.
اما ازطرف ديگر، برنامه نويس نيز مي تواند توابع مورد نياز خود را تعريف كرده و از آنها در برنامه خود استفاده نمايد. به اين دسته از توابع، توابع كاربر مي گوييم. در قسمت بعدي به نحوه تعريف و استفاده از توابع كاربر مي پردازيم.

10-1 توابع كاربر

بكار گيري توابع شامل دو قسمت است:
1- تعريف تابع
2- استفاده از تابع (فراخواني تابع)

ما در مثالهاي قبلي تابع main را در برنامه هاي خود تعريف مي كرديم. تعريف توابع ديگر نيز بطور مشابه مي باشد، كه جزئيات آن را بررسي خواهيم كرد. البته تابع اصلي بطور اتوماتيك در ابتداي اجراي برنامه فراخواني مي گردد و برنامه نويس صريحا آن را احضار نمي كند، ولي ساير توابع بايد از داخل تابع ديگري (از جمله main) بطور صريح فراخواني گردند. البته ما قبلا توابعي همچونscanf و printf را فراخواني كرده ايم، اما جزئيات مربوط به نحوه فراخواني را در قسمت بعدي بررسي مي نماييم.

10-1-1 تعريف تابع

قالب كلي تعريف تابع بصورت زير است:

```
<return-type>  <function-name> (<param-type param-name> , … ) {

   <local variable definitions> ;

   <statements> ;
}
```

كه توضيح هريك در زير آمده است:

1- نوع مقدار بازگشتي (<return-type>) : نوع داده مقداري كه توسط تابع بازگشت داده مي شود را نشان مي دهد. اين نوع داده مي تواند هريك از انواع داده پايه (مانند int) و يا نوع داده هاي تعريف شده توسط كاربر (مانند ساختارها) و يا يك اشاره گر باشد. اين مقدار در داخل تابع توسط دستور return به تابع فراخواننده برگشت داده مي شود. براي اين كار كافي است به شكل زير عمل نماييم:
   return (<exp>) ; و يا return <exp>;

بعبارت ديگر گذاشتن پرانتزها براي مقدار بازگشتي اختياري است.
چنانچه تابع مقدار بازگشتي نداشته باشد، از كلمه كليدي void بجاي نوع مقدار بازگشتي استفاده مي نماييم. دراينصورت، دستور return نياز به مقدار بازگشتي ندارد. البته چنانچه تابع بيش از يك مقدار بازگشتي داشته باشد، بايد از تكنيكهاي گفته شده در قسمت بعد استفاده نماييم.

2- نام تابع (<function-name>) : نام تابع در حقيقت يك شناسه است كه از همان قوانين نامگذاري مربوط به شناسه ها تبعيت مي كند. يك تابع توسط نام خود فراخواني مي گردد.

3- ليست پارامترها : پس از نام تابع و در داخل پرانتز، ليست پارامترهاي تابع قرار مي گيرد. اين ليست شامل تعريف تعدادي پارامتر است كه با كاما ',' از يكديگر جدا شده اند. تعريف هر پارامتر شامل نوع و سپس نام پارامتر مي باشد. پارامترها در حقيقت رابط بين تابع احضار كننده و تابع احضار شونده هستند. بدين صورت كه هنگامي كه تابعي فراخواني مي گردد، فراخواننده بايد اطلاعات لازم را در قالب تعدادي **آرگومان** به تابع مورد نظر ارسال نمايد. اين آرگومانها در داخل پارامترهاي متناظر تابع كپي شده و از آن پس تابع فراخواني شده مي تواند از اين پارامترها همانند متغيرهاي عادي (كه حاوي اطلاعات ارسالي از تابع فراخواننده هستند)، استفاده نمايد. مسلم است كه تعداد و نوع پارامترهاي تعريف شده در تابع، بايد با تعداد و نوع آرگومانهايي كه براي فراخواني تابع استفاده مي شوند، يكسان باشد. لازم به ذكر است كه پارامترهاي تابع، جزو متغيرهاي **محلي** آن محسوب مي گردند و در توابع ديگر شناخته شده نيستند.

4- تعريف متغيرهاي محلي (<local variable definitions>) : در اين قسمت هرگونه متغيري كه براي انجام وظايف محوله به تابع مورد نياز باشد، تعريف مي گردد. اين متغيرها نيز همانند پارامترهاي تابع، **محلي** محسوب مي شوند و در ساير توابع شناخته شده نيستند. با شروع هر تابع، اين متغيرها بطور اتوماتيك ايجاد شده و پس از خاتمه آن از بين مي روند.

5- دستورات برنامه (<statements>) : در اين قسمت، دستورات تشكيل دهنده بدنه تابع كه وظايف مورد نظر برنامه نويس را انجام مي دهند، قرار مي گيرند. لازم به ذكر است كه اين قسمت بايد يك (يا چند) دستور return داشته باشيم كه كنترل را به تابع فراخواننده باز گرداند. البته درمورد توابعي كه مقداري را باز نمي گردانند، درصورت عدم وجود دستور return، كنترل پس از رسيدن به انتهاي تابع يعني { بطور خودكار به تابع فراخواننده باز مي گردد.

نكته مهم ديگر مكان تعريف توابع در يك برنامه C است. يك برنامه C مي تواند داراي يك يا چند تابع باشد كه يكي از آنها بايد حتما main ناميده شود و همانطور كه قبلا نيز گفته شد، اجراي برنامه از اين تابع آغاز مي گردد. توابع مي توانند به هر ترتيبي تعريف شوند، اما معمولا تابع main در آخر توابع ديگر تعريف مي گردد؛ گرچه اين مسئله اجباري نيست. توابع بايد بصورت پشت سر هم تعريف گردند و برخلاف بعضي از زبانهاي ديگر، نمي توان يك تابع را در داخل تابع ديگر تعريف كرد. بعبارت ديگر، كليه توابع در يك سطح قرار دارند و هيچ تابعي، شامل تابع ديگر نمي باشد.

10-1-2 فراخواني توابع

براي فراخواني يك تابع بايد از نام آن بعلاوه ليست آرگومانهاي متناسب با پارامترهاي تابع استفاده كرد. نكته مهم آنستكه بايد تعداد، ترتيب و نوع آرگومانهاي ارسالي با پارامترهاي متناظرشان در تعريف تابع، منطبق باشد. در غيراينصورت ممكن است خطاي نحوي و يا حتي خطاي منطقي رخ دهد. هنگاميكه يك تابع فراخواني مي گردد، اجراي تابع فراخواننده بطور موقت متوقف شده و كنترل اجرا به تابع فراخواني شده منتقل مي گردد. پس از اتمام تابع فراخواني شده و اجراي دستور return توسط آن، كنترل اجرا به تابع فراخواننده بازگشته و اجرا را از دستور بعدي، از سر مي گيرد.
چنانچه تابع هيچ مقداري را بازنگرداند، مي توان آن را بصورت يك دستور مستقل فراخواني كرد. بعنوان مثال :

```
clrscr() ;
```

اما توابعي كه يك مقدار خروجي را باز مي گردانند، مي توان در يك عبارت نسبت دهي يا محاسباتي نيز بكاربرد. بعنوان مثال مي توان تابع sqrt (كه يك عدد را دريافت و جذر آن را باز مي گرداند) را بصورت زير استفاده كرد :

```
a = sqrt(10) ;
a = 2 * sqrt(b) + c ;
```

نكته مهم آنستكه چنانچه فراخواني تابع توسط مقدار باشد (به بخش 4-10 مراجعه كنيد)، آنگاه مي توان بجاي يك متغير يا يك ثابت، يك عبارت محاسباتي را نيز به تابع ارسال كرد. بعنوان مثال فراخواني زير مجاز است:

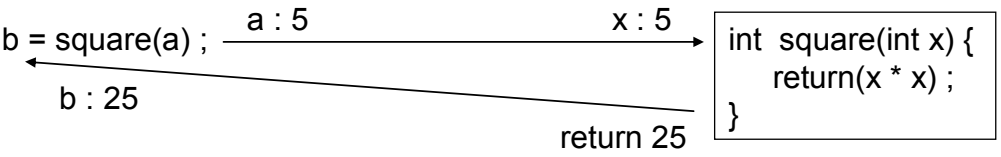
```
a = sqrt(2*b+8) ;
```

در اينحالت، ابتدا عبارت محاسباتي ارزيابي شده و سپس مقدار آن بعنوان آرگومان به تابع ارسال مي گردد. البته درصورتيكه فراخواني توسط ارجاع باشد، فقط يك متغير مي تواند به تابع ارسال گردد و عبارت محاسباتي و يا حتي يك ثابت به تنهايي نيز مورد قبول نخواهد بود.
براي آشنايي بيشتر با نحوه فراخواني توابع، به شكل زير توجه كنيد:

```
void  main() {
   int  a, b;

   a = 5 ;

   b = square(a) ;
```




```
   printf("a=%d and b=%d",a,b)
}
```

```
a = 5 and b = 25
```

10-2 چند نمونه از توابع

براي آشنايي بيشتر با نحوه كار توابع به ذكر چند مثال مي پردازيم.

برنامه 1) برنامه اي بنويسيد كه مقدار تركيب n به k را محاسبه نمايد.

```
#include <stdio.h>

long int  factorial(int  number) {
   int i;
   long int f = 1L ;

   for (i=1; i<=n; i++)
      f *= i ;
   return(f) ;
}
```

```
void main() {
   int n,k,result ;

   printf("Please enter n and k : ");
   scanf("%d %d",&n,&k) ;
   result = factorial(n) / ( factorial(k) * factorial(n-k) ) ;
   printf("result = %d",result) ;
}
```

```
Please enter n and k : 5  3
result = 10
```

برنامه 2) برنامه اي بنويسيد كه تعدادي عدد مثبت را دريافت و سپس حداكثر آنها را محاسبه و چاپ نمايد. براي اينكار از تابعي بنام getMax كه دو عدد را دريافت و حداكثر آنها را باز مي گرداند، استفاده نماييد.

```
#include <stdio.h>

int  getMax(int a, int b) {
   if (a>b)  return(a);
   else return(b) ;
}

void main() {
  int  i, n, max, number ;

  printf("Please enter n: ");
  scanf("%d",&n);

  max = -1;
  for (i=0; i<n; i++) {
    printf("Please enter number: ");
    scanf("%d", &number);
    max = getMax(number, max);
  }

  printf("Maximum is : %d", max);
}
```

```
Please enter n: 4
Please enter number: 10
Please enter number: 8
Please enter number: 32
Please enter number: 25
Maximum is 32
```

در مثال زير به توابعی می پردازيم كه مقدار خروجی ندارند.

برنامه 3) تابعی بنویسید که یک خط 40 تایی از علامت * را چاپ نماید. سپس با استفاده از آن برنامه ای بنویسید که ابتدا پیام Hello و سپس خط جداکننده فوق را چاپ کند.

```
#include <stdio.h>

void starLine() {
  int i;

  for (i=0; i<40; i++)
     printf("*");
  printf("\n");
}

void main() {

  printf("Hello\n");
  starLine() ;
}
```

```
Hello
****************************************
```

البته درچنین مواردی، بهتر است برنامه نویس تابع رسم خط را بصورت کلی تری بنویسد؛ بطوریکه در سایر برنامه ها نیز بتواند از آن استفاده نماید. مثلا چنانچه تابع را بگونه ای بنویسیم که کاراکتر جداکننده و تعداد آن را بعنوان ورودی دریافت کند، حالت کلی تری پیدا خواهد کرد.

```
#include <stdio.h>

void separatorLine(char sep,int n) {
  int i;

  for (i=0; i<n; i++)
     printf("%c",sep);
  printf("\n");
}

void main() {

  printf("Hello\n");
  separatorLine('-' ,60) ;
}
```

```
Hello
------------------------------------------------------------
```

مثال بعدی، یک برنامه کامل برای نشان دادن نحوه کار با توابع است:

برنامه 4) يك دانشگاه قصد دارد به دانشجويان مقاطع مختلف خود امتيازدهي نمايد. نحوه تخصيص امتيازها بشرح زير است:

- دانشجويان كارشناسي

- معدل کل 0.8 x
- شرکت در مسابقات و فعاليتهاي علمي 0.2 x

- دانشجويان کارشناسي ارشد
  - معدل کل 0.6 x
  - مقالات ارائه شده در کنفرانسها 0.4 x

- دانشجويان دکترا
  - معدل کل 0.5 x
  - تعداد مقالات کنفرانس 0.2 x
  - تعداد مقالات مجله علمي 0.3 x

برنامه اي بنويسيد که براي تعدادي دانشجو از مقاطع مختلف، اطلاعات لازم را دريافت و پس از محاسبه امتياز آنها، ميانگين امتيازات و حداکثر امتياز براي هر مقطع را بطور جداگانه محاسبه و چاپ نمايد.

```
#include <stdio.h>

char getStudentType() {
  char type;

  do {
     printf(" B : BSc student.\n");
     printf(" M : MSc student.\n");
     printf(" P : PhD student.\n");
     printf(" Q : Quit.\n");
     printf("Please enter student type : ") ;
     type = getch() ;
     if (type >= 'a'  && type <='z')   type -= 32 ;
     putch(type);
  } while (type !='B' && type !='M' && type !='P' && type !='Q') ;

  return(type);
}

int  getBScStudent() {
   float average;
   int contest;

   printf("BSc Student :\n");
   printf("Please enter average : ");
   scanf("%f",&average);
   printf("Please enter number of contests : ");
   scanf("%d",&contest) ;
   return(0.8 * average + 0.2 * contest);
}

int  getMScStudent() {
   float average;
   int confPapers;
```

```
    printf("MSc Student :\n");
    printf("Please enter average : ");
    scanf("%f",&average);
    printf("Please enter number of conference papers : ");
    scanf("%d",&confPapers) ;
    return(0.6 * average + 0.4 * confPapers);
}

int  getPhDStudent() {
    float average;
    int confPapers, journalPapers;

    printf("PhD Student :\n");
    printf("Please enter average : ");
    scanf("%f",&average);
    printf("Please enter number of conference papers : ");
    scanf("%d",&confPapers) ;
    printf("Please enter number of journal papers : ");
    scanf("%d",&journalPapers) ;
    return(0.5 * average + 0.2 * confPapers + 0.3 * journalPapers);
}

void main() {
    int  BScNo, MScNo, PhDNo;
    float  grade, BScAverage, MScAverage, PhDAverage ;
    float  BScMax, MScMax,PhDMax ;
    char type;

    BScNo = MScNo = PhDNo = 0;
    BScAverage = MScAverage = PhDAverage = 0.0 ;
    BScMax = MScMax = PhDMax = -1.0 ;

    do {
        type = getStudentType() ;
        switch (type) {
            case 'B' :  grade = getBScStudent() ;
                        BScNo ++;
                        BScAverage += grade ;
                        if (grade > BScMax)  BScMax = grade;
                        break ;

            case 'M' :  grade = getMScStudent() ;
                        MScNo ++;
                        MScAverage += grade ;
                        if (grade > MScMax)  MScMax = grade;
                        break ;

            case 'P' :  grade = getPhDStudent() ;
                        PhDNo ++;
```

```
                PhDAverage += grade ;
                if (grade > PhDMax)  PhDMax = grade;
                break ;
        }
    } while (grade != 'Q') ;

    printf("Results :\n");
    printf("BSc : Maximum :%f and Average : %f",BScMax,BscAverage);
    printf("MSc : Maximum :%f and Average : %f",MscMax,MscAverage);
    printf("PhD : Maximum :%f and Average : %f",PhDMax,PhDAverage);
}
```

10-3 نمونه اولیه توابع

همانطور که قبلا گفته شد، توابع می توانند به هر ترتیبی تعریف شوند، ولی معمولا تابع main در انتها قرار می گیرد. اما اگر بخواهیم دقیقتر صحبت کنیم، تعریف هر تابع باید قبل از فراخوانی آن صورت پذیرد. چنانچه تابعی قبل از آنکه تعریف شود، فراخوانی گردد؛ یک خطای کامپایل رخ خواهد داد. اما می توان این خطا را با استفاده از نمونه اولیه تابع (prototype) که به آن پیش تعریف نیز گفته می شود، از بین برد. یک نمونه اولیه تابع، شامل نام تابع، نوع داده ای که باز می گرداند و همچنین تعداد و نوع پارامترهای تابع است. برای مثال، نمونه اولیه تابع factorial که در بالا تعریف شد، بصورت زیر است:

```
long int  factorial(int) ;
```

این نمونه اولیه می گوید که تابع factorial یک آرگومان از نوع int دریافت و یک مقدار از نوع long int باز می گرداند. علامت ; انتهای دستور نشان می دهد که این فقط یک نمونه اولیه بوده و شامل تعریف بدنه تابع نمی باشد.
نمونه اولیه یک تابع حتما باید با تعریف آن (که در قسمتهای بعدی برنامه آمده است) یکسان باشد و گرنه یک خطای کامپایل رخ خواهد داد.
کامپایلر از نمونه اولیه تابع برای بررسی درستی نحوه فراخوانی تابع (از نظر تعداد و نوع آرگومانها) استفاده میکند. لازم به یادآوری است که فقط درصورتی نیاز به استفاده از نمونه اولیه داریم که فراخوانی تابع، پیش از تعریف آن صورت پذیرد. درصورتیکه تابع پیش از فراخوانی تعریف شود، از همان تعریف، بعنوان نمونه اولیه نیز استفاده می شود.

10-4 انواع فراخواني توابع

بطورکلی، به دو روش می توان یک تابع را فراخوانی کرد:

- فراخوانی توسط مقدار (Call by Value)
- فراخوانی توسط ارجاع (Call by Reference)

که هريك را جداگانه بررسي مي نماييم.

10-4-1 فراخواني توسط مقدار

در فراخوانی توسط مقدار، آرگومانهای ارسالی توسط تابع فراخواننده، در پارامترهای متناظر تابع فراخوانی شده، کپی می گردند. بنابراین تابع فراخوانی شده عملیات خود را برروی یک کپی از آرگومانهای ارسالی انجام می دهد. درنتیجه، در صورت انجام هرگونه تغییری برروی این کپی توسط تابع فراخوانی شده، متغیر اصلی در تابع فراخواننده تغییر نخواهد کرد. مزیت این نوع فراخوانی در این است که می توان بدون هیچ نگرانی از تغییر ناخواسته متغیرها، آنها را به هر تابعی ارسال کرد، چرا که فقط یک کپی از آنها به تابع ارسال می شود.

در زبان C، در حالت عادی فراخوانی توسط مقدار صورت می پذیرد. برای روشن شدن موضوع به مثال زیر توجه کنید:

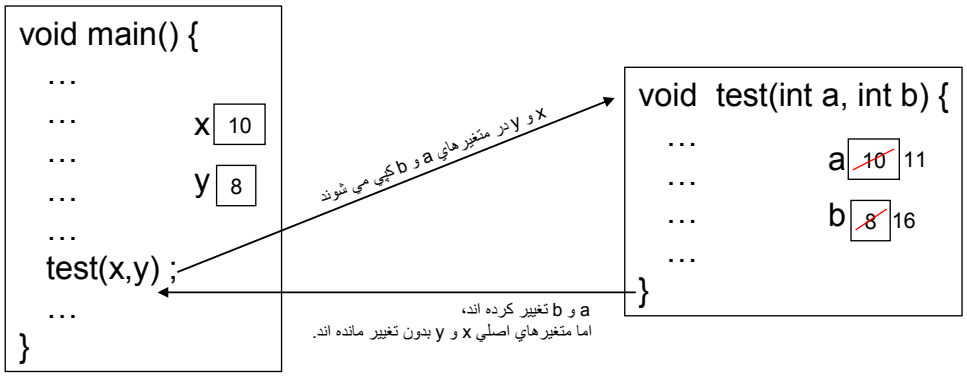
```
void  test(int a, int b) {
  printf("Function test : a=%d and b=%d \n",a,b);
  a ++ ;
  b *= 2;
  printf("Function test : a=%d and b=%d \n",a,b);
}

void main() {
  int x, y;

  x = 10;
  y = 8;
  printf("Function main : x=%d and y=%d \n",x,y);
  test(x,y) ;
  printf("Function main : x=%d and y=%d \n",x,y);
}
```

```
Function main : x=10 and y=8
Function test : a=10 and b=8
Function test : a=11 and b=16
Function main : x=10 and y=8
```

همانگونه که مشاهده مي کنيد، گرچه پارامترهاي a و b در داخل تابع test تغيير کرده اند، اما پس از بازگشت به تابع فراخواننده يعني main، آرگومانهاي متناظر ب آنها يعني x و y همچنان همان مقادير اوليه خود را دارا هستند. شکل زير نحوه کار را نشان مي دهد.



همانطور که گفته شد، اين نحوه فراخواني حالت پيش فرض در زبان C بوده ومعمولا نيز برنامه نويسان ترجيح مي دهند از اين روش براي ارسال آرگومانها به توابع استفاده نمايند؛ چرا که متغيرها را در برابر تغييرات ناخواسته در هنگام ارسال به توابع ديگر حفظ مي کند. اما گاهي لازم است که تابع فراخواني شده بتواند مقدار متغيرهاي دريافتي را تغيير دهد، و اين تغييرات در متغيرهاي اصلي ارسال شده از طرف تابع فراخواننده نيز اعمال شود. در اينصورت بايد از روش فراخواني توسط ارجاع استفاده نماييم.

10-4-2 فراخواني توسط ارجاع

در اين روش فراخواني، خود آرگومانهاي اصلي (بجاي يك كپي از آنها) به تابع فراخواني شده ارسال مي گردد. در نتيجه، هرگونه تغييري در پارامترهاي تابع فراخواني شده، مقدار آرگومانهاي اصلي در تابع فراخواني كننده را نيز تغيير خواهد داد.
در C اوليه، تنها راه فراخواني توابع توسط ارجاع، استفاده از متغيرهاي اشاره گر بود كه در فصول بعدي اين روش بررسي خواهد گرديد. اما در استاندارد جديد C، مي توان اين كار را به روش ساده تر ي و با استفاده از **متغيرهاي ارجاعي** انجام داد.
يك متغير ارجاعي، در حقيقت يك **نام مترادف** و يا يك **جايگزين** براي يك متغير ديگر است. براي تعريف يك متغير ارجاعي از علامت & پس از نوع متغير مورد ارجاع استفاده مي كنيم. بعنوان مثال به نمونه زير توجه كنيد:

```
int  a = 10;
int  &r = a;
```

در خط اول a بعنوان يك متغير صحيح تعريف شده و مقدار اوليه 10 به آن داده شده است. در خط دوم، r بعنوان يك متغير ارجاعي تعريف شده است كه به a ارجاع مي نمايد. از اين پس متغير r، نام ديگري براي متغير a محسوب مي شود و هرگونه تغييري در r، باعث تغيير a نيز خواهد شد. به برنامه زير دقت كنيد:

```
#include <stdio.h>

void main() {
  int  a = 10;
  int  &r = a;

  r ++;
  printf("a=%d and r=%d \n",a,r);

  a ++;
  printf("a=%d and r=%d \n",a,r);
}
```

```
a = 11 and r = 11
a = 12 and r = 12
```

مثال فوق نشان مي دهد كه در حقيقت a و r يك متغير هستند و هرگونه تغييري در هريك از اين دو، ديگري را نيز تغيير خواهد داد.
براي ارسال آرگومانها توسط ارجاع، كافي است پارامترهاي تابع مورد نظر را بصورت متغير ارجاعي تعريف نماييم. در اينصورت، اين پارامترها در حقيقت يك ارجاع به آرگومانهاي ارسالي خواهند بود و در نتيجه هرگونه تغييري در آنها، آرگومانهاي اصلي در تابع فراخواني كننده را نيز تغيير خواهد داد. بعبارت بهتر، در اين روش بجاي آنكه يك كپي از آرگومانها در پارامترها قرار گيرد، **خود** آرگومانها به تابع فراخواني شده ارسال خواهند شد. بعنوان مثال، همان تابع قسمت قبلي را كه بصورت فراخواني توسط مقدار عمل مي كرد، مجددا به روش فراخواني با ارجاع باز نويسي مي كنيم تا تفاوت اين دو مشخص گردد:

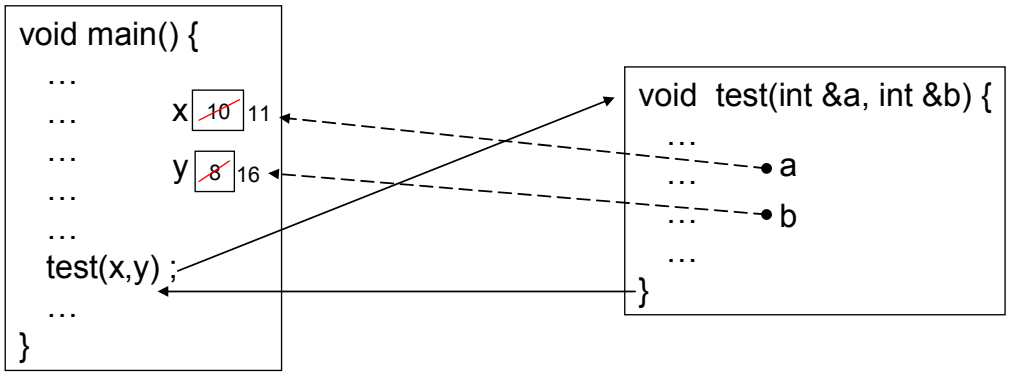
```
void  test(int &a, int &b) {
  printf("Function test : a=%d and b=%d \n",a,b);
  a ++ ;
  b *= 2;
  printf("Function test : a=%d and b=%d \n",a,b);
}

void main() {
  int x, y;
```

```
  x = 10;
  y = 8;
  printf("Function main : x=%d and y=%d \n",x,y);
  test(x,y) ;
  printf("Function main : x=%d and y=%d \n",x,y);
}
```

```
Function main : x=10 and y=8
Function test : a=10 and b=8
Function test : a=11 and b=16
Function main : x=11 and y=16
```

همانگونه كه مشاهده مي كنيد، تغيير پارامترهاي a و b در داخل تابع test باعث شده كه آرگومانهاي متناظر آنها در تابع فراخواننده، يعني x و y نيز تغيير نمايند. شكل زير نحوه كار را نشان مي دهد.



اكنون سوال اصلي اين است كه در چه مواردي از فراخواني توسط ارجاع استفاده كنيم؟ فراخواني توسط ارجاع در دو مورد كاربرد دارد، كه هريك را با ذكر يك مثال توضيح مي دهيم.

1- مي دانيم كه هر تابع فقط مي تواند يك مقدار را توسط دستور return باز گرداند. در مواردي كه تابع بايد بيش از يك مقدار را باز گرداند، تنها راه استفاده از فراخواني توسط ارجاع است. بدين صورت كه تعدادي پارامتر **خروجي** را بصورت ارجاعي به تابع ارسال مي كنيم، و تابع پس از انجام محاسبات، خروجي را در اين متغيرها قرار داده و باز مي گرداند. از آنجا كه پارامترها توسط ارجاع ارسال شده اند، تغييرات اعمال شده توسط تابع (قرار دادن مقدار خروجي در آنها) به تابع فراخواننده منتقل خواهد شد.

برنامه 5) برنامه اي بنويسيد كه ضرايب يك معادله درجه دوم را دريافت و ريشه هاي آن را محاسبه وچاپ نمايد.

حل) براي حل اين مسئله، ابتدا تابعي بنام equation مي نويسيم كه ضرايب يك معادله درجه دوم را دريافت و ريشه هاي آن را بازگرداند. اما اين تابع ممكن است صفر، يك و يا دو ريشه را بازگرداند. بنابراين ناچاريم دو پارامتر خروجي بنام x1 و x2 به آن ارسال نماييم تا ريشه ها را در آنها قرار دهد. در ضمن مقدار خروجي خود تابع نيز، تعداد ريشه ها خواهد بود.

```
#include <stdio.h>
#include <math.h>

int  equation(int a, int b, int c, float &x1, float &x2) {
  float  delta;
```

```
  if (a==0) return(0) ;

  delta = b*b – 4*a*c ;
  if (delta < 0) return(0) ;
  else if (delta == 0) {
          x1 = -b / (2*a) ;
          return(1) ;
        }
        else {
          delta = sqrt(delta) ;
          x1 = (-b + delta) / (2*a) ;
          x2 = (-b – delta) / (2*a) ;
          return(2) ;
        }
}

void main() {
  int  resultNo, coef1, coef2, coef3;
  float result1,result2 ;

  printf("Please enter coefficients: ");
  scanf("%d %d %d", &coef1, &coef2, &coef3);

  resultNo = equation(coef1,coef2,coef3,result1,result2);
  if (resultNo == 0)
    printf("There is no answer! \n");
  else if (resultNo==1)
          printf("There is 1 answer, x=%f \n", x1);
        else printf("There are 2 answers, x1=%f and x2=%f \n", x1, x2);
}
```

2- در مواردي كه آرگومان ارسالي به تابع، همزمان نقش ورودي و خروجي را دارا باشد. يعني آرگومان علاوه بر آنكه مقداري را به تابع ارسال مي كند، ممكن است مقداري را به تابع فراخواننده نيز بازگرداند. مثال زير اين موضوع را روشن مي كند.

برنامه 6) برنامه اي بنويسيد كه دو عدد را از كاربر دريافت و با استفاده از يك تابع، مقدار آن دو را جابجا نموده و حاصل را چاپ نمايد.

```
#include <stdio.h>

void swap(int &a, int &b) {
  int temp;
  temp = a;
  a = b;
  b = temp;
}

void main() {
  int x, y;
```

```
  printf("enter two numbers : ");
  scanf("%d %d",&x,&y);

  printf("x = %d and y = %d \n", x, y);
  swap(x,y);
  printf("x = %d and y = %d \n", x, y);
}
```

```
enter two numbers : 8  12
x = 8  and  y = 12
x = 12  and  y = 8
```

دقت كنيد كه در برنامه فوق، چنانچه آرگومانها بصورت مقدار به تابع swap ارسال مي شد، پس از بازگشت از تابع، مقادير متغيرهاي x و y هيچ تغييري نكرده و همان مقادير قبلي خود را دارا بودند.
نكته بسيار مهم ديگري كه بايد بدان توجه كرد، اين است كه چنانچه پارامتري بصورت فراخواني توسط ارجاع تعريف شده باشد، تنها مي توان يك متغير را بعنوان آرگومان به آن ارسال كرد؛ و ارسال ثابت يا عبارت محاسباتي مجاز نيست. بعنوان مثال فراخوانيهاي زير براي تابع swap مجاز نيستند:

```
swap(8,10) ; // compile error
swap(x+5, y); // compile error
```

10-5 حوزه شناخت متغير

يكي از مسائل مهم در مورد متغيرها، حوزه شناخت (Scope) متغير است كه تعيين مي نمايد متغير در چه قسمتهايي از برنامه شناخته شده است و در چه قسمتهايي قابل استفاده نيست. بطور كلي متغيرها به دو دسته تقسيم مي شوند:

1- متغيرهاي محلي (local variable): متغيرهايي هستند كه در داخل يك بلاك {} تعريف شده اند و فقط در محدوده همان بلاك شناخته شده هستند. نمونه اين دسته از متغيرها، متغيرهاي محلي توابع هستند كه در داخل بلاك مربوط به تابع تعريف مي شوند و فقط در همان تابع شناخته شده هستند. البته دو تابع مختلف مي توانند داراي متغيرهاي همنام باشند، كه در اينصورت اين دو متغير مجزا بوده و هيچ ارتباطي به يكديگر ندارند. لازم به ذكر است كه پارامترهاي يك تابع نيز جزو متغيرهاي محلي آن تابع محسوب مي گردند.
نكته ديگر اينكه گرچه معمولا متغيرهاي محلي در داخل تابع تعريف مي شوند، اما هر بلوك دلخواه مي تواند داراي متغيرهاي محلي باشد. مثلا يك دستور if مركب مي تواند در داخل بلوك خود، متغيرهاي محلي را تعريف كند كه فقط در داخل همان بلوك شناخته شده باشند (به مثال بعدي مراجعه كنيد).

2- متغيرهاي سراسري (global variable): متغيرهاي هستند كه در خارج كليه بلوكها (و توابع از جمله main) تعريف شده اند و در كل توابع برنامه (در حقيقت كل فايل مربوط به برنامه) شناخته شده و قابل استفاده مي باشند. از آنجا كه كليه توابع برنامه به اين متغيرها دسترسي دارند، هرگونه تغييري در اين متغيرها توسط يكي از توابع، در ساير توابع نيز قابل رويت خواهد بود.

به مثال زير توجه كنيد :

```
int  a ;  //متغير سراسري

void main() {
  int b;  // main متغير محلي براي
```

```
  a = 10 ; // متغير سراسري دراينجا قابل دسترسي است
  b = 5 ; // متغير محلي دراينجا قابل دسترسي است


  if (a>0) {
    int c ; // متغير محلي براي بلوك
    c = 8;
    // متغيرهاي a و b نيز در اينجا قابل دسترسي هستند
    ....
  }

  // متغير c در اينجا شناخته شده نيست
}

void test() {
  int c ; // متغير محلي براي test

  c = 5;   // متغير محلي در اينجا قابل دسترسي است
  a = 20;   // متغير سراسري در اينجا قابل دسترسي است
}
```

نكته مهم آنستكه درصورتيكه يك تابع داراي يك متغير محلي همنام با يك متغير سراسري باشد، در اينصورت هرگونه ارجاع به اين نام مشترك، به متغير محلي رجوع خواهد كرد (به مثال قسمت بعدي رجوع شود).
چه موقع از متغير هاي سراسري استفاده كنيم؟ متغيرهاي سراسري هنگامي مفيد هستند كه يك داده بين چندين تابع بصورت مشترك استفاده شود. در اين حالت نيازي به ارسال متغيرهاي مشترك از طريق پارامترها نمي باشد. اما متاسفانه از آنجا كه متغيرهاي سراسري در كليه توابع در دسترس هستند، ممكن است بصورت ناخواسته دچار تغيير شوند. علاوه براين اشكال زدايي آنها نيز بسيار مشكل است، چرا كه محل بروز خطا مشخص نيست و هريك از توابع ممكن است مقدار متغير را تغيير داده باشند. بنابراين در برنامه نويسي ساختيافته توصيه مي گردد تا حد ممكن از متغيرهاي سراسري استفاده **نكنيد.**

10-6 رده های ذخیره سازی

رده ذخیره سازی یک متغیر، مدت زمان حضور آن را در برنامه تعیین می نماید. بعبارت دیگر، رده ذخیره سازی تعیین می نماید یک متغیر چه موقع بوجود می آید و چه زمانی از بین می رود. در هنگام تعریف یک متغیر، باید به همراه نوع آن، رده ذخیره سازی آن را نیز مشخص کرد. چنانچه اینکار صورت نپذیرد، کامپایلر از رده ذخیره سازی پیش فرض استفاده خواهد کرد. از آنجایی که در کلیه مثالهایی که تاکنون ذکر شده اند، رده ذخیره سازی بطور صریح مشخص نشده بود، همگی از رده پیش فرض در نظر گرفته شده اند.
بطور کلی دو رده ذخیره سازی برای متغیرها وجود دارد:

- رده ذخیره سازی اتوماتیک
- رده ذخیره سازی ایستا

که هریک را جداگانه بررسی می نماییم.

10-6-1 رده ذخیره سازی اتوماتیک

متغیرهای متعلق به رده ذخیره سازی اتوماتیک، هنگام ورود به بلوکی که این متغیرها در آن اعلان شده اند، ایجاد شده و در طول اجرای این بلوک در حافظه وجود دارند؛ به محض خاتمه بلوک، این متغیرها نیز از بین رفته و حافظه آنها پس گرفته می شود. متغیرهای محلی (شامل پارامترهای توابع)، معمولا از این رده می

باشند. بعنوان مثال، متغیرهای محلی یک تابع، به محض فراخوانی تابع ایجاد می شوند و در حین اجرای تابع در حافظه حضور دارند. با پایان یافتن اجرای تابع، این متغیرها نیز از بین می روند. این مسئله باعث می شود که صرفه جویی قابل توجهی در حافظه داشته باشیم. چرا که هرگاه به متغیری نیاز داریم ایجاد شده و با پایان یافتن کار نیز حافظه آن آزاد می شود.
برای تعریف یک متغیر اتوماتیک، باید از کلمه کلیدی auto، قبل از مشخصه نوع متغیر، استفاده نماییم. بعنوان مثال به نمونه زیر دقت نمایید :

```
void  test(int a, int b) {
  auto  int  i;
  float  k;
  …
  // دستورات تابع
}
```

در تابع فوق، متغیر i از رده ذخیره سازی اتوماتیک تعریف شده است. اما نکته مهم اینجاست که متغیرهای محلی، بطور پیش فرض متعلق به رده ذخیره سازی اتوماتیک هستند. بدین معنا که چنانچه رده ذخیره سازی آنها بطور صریح مشخص نشود(مانند متغیر kدر مثال فوق)، از رده اتوماتیک در نظر گرفته می شوند. بنابراین در مثال فوق نه تنها متغیر i، بلکه متغیرهای محلی k، a و b نیز از رده اتوماتیک درنظر گرفته می شوند. بهمین دلیل معمولا برنامه نویسان از کلمه کلیدی auto استفاده نمی کنند.
نوع دیگری از متغیرهای اتوماتیک نیز وجود دارد. همانطور که می دانید کلیه متغیرهای برنامه در حافظه اصلی کامپیوتر (RAM) ذخیره می گردند. اما پردازنده اصلی (CPU) برای پردازش داده ها باید ابتدا آنها را از حافظه اصلی به یک حافظه داخلی بنام ثبات یا Register منتقل نماید. گرچه سرعت دسترسی به حافظه اصلی بسیار بالا است، اما بهرحال مدتی زمان صرف این انتقال می شود. پس از آن پردازنده می تواند با سرعت بسیار بالا عملیات را برروی ثباتهای داخلی خود انجام داده و حاصل را مجددا به حافظه اصلی منتقل نماید. حال اگر متغیری داشته باشیم که عملیات زیادی بر روی آن انجام شود، مانند متغیرهای شمارنده حلقه، می توانیم از کامپایلر بخواهیم که آن را بجای حافظه اصلی، بطور مستقیم در داخل یکی از ثباتهای پردازنده ذخیره نماید. با این کار دیگر نیازی به عملیات انتقال نیست و سرعت بنحو قابل ملاحظه ای افزایش می یابد. برای اینکار کافی است که از کلمه کلیدی register استفاده نماییم. بعنوان مثال در تابع قبلی می توانستیم متغیر i را بصورت زیر تعریف نماییم:

```
register  int  i;
```

با این وجود دقت کنید که تعداد ثباتهای پردازنده محدود است و کامپایلر فقط درصورتی این درخواست را قبول می نماید که ثبات خالی وجود داشته باشد؛ در غیراینصورت متغیر را در حافظه اصلی ذخیره خواهد کرد(اما هیچ خطایی اعلام نخواهد کرد). همانطور که گفته شد فقط از متغیرهای بسیار پرکاربرد، بخصوص متغیرهایی که در داخل یک حلقه استفاده می شوند، بعنوان متغیر ثبات استفاده نمایید.

10-6-2 رده ذخیره سازی ایستا

متغیرهای متعلق به رده ذخیره سازی ایستا، از ابتدای آغاز برنامه ایجاد می شوند و تا پایان برنامه نیز در حافظه حضور دارند. متغیرهای سراسری به این دسته متعلق هستند. با شروع اجرای برنامه به متغیرهای سراسری حافظه تخصیص داده می شود. پس از آن کلیه توابع قادر به دیدن و تغییر مقدار آنها هستند، اما نمی توانند حافظه تخصیص یافته به این متغیرها را بازپس بگیرند. در پایان و پس از خاتمه تابع main، حافظه تخصیص یافته به این متغیرها بازپس گرفته می شود.
اما علاوه بر متغیرهای سراسری، متغیرهای محلی نیز می توانند از رده ذخیره سازی ایستا تعریف شوند. اگر یک متغیر محلی، بصورت ایستا تعریف گردد؛ فقط یکبار و آن هم در شروع اجرای برنامه ایجاد شده و مقدار اولیه خواهد گرفت(البته درصورتیکه به آن مقدار اولیه داده باشیم). پس از آن، با هر بار اجرای تابع، قادر به دسترسی به این متغیر خواهیم بود(چرا که یک متغیر محلی است) ؛ اما با خاتمه تابع این متغیر از بین نرفته و مقدار آن تا فراخوانی بعدی تابع حفظ خواهد شد. متغیرهای محلی ایستا هنگامی از بین می روند که برنامه اصلی خاتمه یابد. از این متغیرها هنگامی استفاده می شود که بخواهیم مقدار یک متغیر محلی در فراخوانیهای

متوالی یک تابع حفظ شود(البته بدون اینکه سایر توابع به آن دسترسی داشته باشند، درغیراینصورت آن را بصورت سراسری تعریف می کردیم).
برای تعریف یک متغیر محلی از رده ذخیره سازی ایستا، از کلمه کلیدی static استفاده می شود. البته استفاده از این کلمه الزامی است، چرا که متغیرهای محلی بطور پیش فرض از رده ذخیره سازی اتوماتیک درنظر گرفته می شوند. مثال زیر تفاوت متغیرهای محلی اتوماتیک و ایستا را نشان می دهد:

```
#include <stdio.h>

void computeSum(int number) {
  int autoSum = 0;
  static int staticSum = 0;

  autoSum += number;
  staticSum += number ;
  printf("autoSum = %d and staticSum = %d \n",autoSum,staticSum);
}

void main() {
  int i;

  for (i=1; i<=5; i++)
     computeSum(i) ;
}
```

```
autoSum = 1 and staticSum = 1
autoSum = 2 and staticSum = 3
autoSum = 3 and staticSum = 6
autoSum = 4 and staticSum = 10
autoSum = 5 and staticSum = 15
```

با دقت در مثال فوق در می یابیم که متغیر محلی ایستای staticSum، فقط یکبار در شروع اجرا مقدار اولیه صفر گرفته است. با هر بار اجرای تابع، این متغیر مقدار قبلی خود را که از اجرای قبلی حفظ نموده است، با ورودی جدید جمع می نماید و در نهایت مجموع کلیه اعداد ارسال شده به تابع را در پنج فراخوانی متوالی چاپ می نماید. اما متغیر محلی اتوماتیک autoSum، با هربار اجرای تابع ایجاد شده و مقدار صفر می گیرد و با پایان یافتن تابع نیز از بین می رود. در نتیجه این متغیر در هربار فقط حاوی مقدار آخرین عددی است که به تابع ارسال شده است. البته از کلمه کلیدی static برای تعریف متغیرهای سراسری نیز می توان استفاده کرد که تاثیر آن در قسمتهای بعدی بررسی خواهد شد.
نوع دیگری از رده ذخیره سازی ایستا، رده extern است. این رده هنگامي بكار مي رود كه برنامه در چندين فايل مختلف قرار داشته باشد. در برنامه هاي بزرگ كه داراي توابع متعددي هستند، براي جلوگيري از بزرگ و پيچيده شدن بيش از حد فايل برنامه، آن را بطور منطقي به چندين فايل تقسيم مي كنند. بدين صورت كه كليه توابع و داده هاي مرتبط با يكديگر را در يك فايل قرار مي دهند. سپس هر فايل بصورت مجزا كامپايل شده و درنهايت تمامي آنها با يكديگر پيوند (link) خورده و تشكيل فايل اجرايي نهايي را مي دهند. اين روش باعث مديريت بهتر پروژه هاي بزرگ مي شود.
اما مشكل هنگامي است كه بخواهيم يك متغير سراسري را در توابع موجود در چند فايل مختلف، مورد استفاده قرار دهيم. مسلما اين متغير سراسري بايد در يكي از فايلها تعريف شود. اما در مورد ساير فايلها چه بايد بكنيم؟ متاسفانه هريك از دو راه زير منجر به شكست مي شود:

- اگر متغير را در ساير فايلها بدون تعريف مجدد استفاده نماييم، كامپايلر اعلام خطا كرده و متغير را نخواهد شناخت.

- اگر متغير را در ساير فايلها نيز تعريف مجدد نماييم، كامپايلر اعلام خطا نخواهد كرد، اما پيوند زننده (linker) در هنگام تركيب فايلها با هم متوجه تعريف چند متغير با نام يكسان شده و اعلام خطا خواهد كرد.

تنها راه حل آن است كه ايت متغير را در يك فايل تعريف كرده و سپس در ساير فايلها آن را بعنوان يك متغير خارجي (extern) اعلان (و نه تعريف) نماييم. مشخصه extern به كامپايلر اعلان مي كند كه اين متغير در جاي ديگري (معمولا يك فايل ديگر) تعريف شده است و بنابراين مي تواند بدون اعلان خطا از آن استفاده كند؛ اما اين مشخصه باعث تعريف مجدد متغير **نمي گردد**. براي تعريف متغير به شكل خارجي بصورت زير عمل مي كنيم:

extern int a;

به مثال زير توجه نماييد:

File1.C

```
int k = 0;

void F1() {
  ….
  k ++; // k دسترسي به متغير سراسري
  …
}

void main() {
  ….
}
```

File2.C

```
extern int k;

void F2() {
  …
  k -- ; // دسترسي به متغير سراسري مشترك
  …
}

void F3() {
  …
}
```

همانگونه كه در مثال بالا ديده مي شود، برنامه داراي دوفايل به نامهاي File1.C و File2.C مي باشد. توجه كنيد كه فقط يكي از اين دو مي تواند داراي تابع main باشد. متغير سراسري k در فايل File1.C تعريف شده است بنابراين توسط توابع آن از جمله F1 قابل دسترسي است. اما اين متغير در فايل File2.C نيز بصورت خارجي تعريف شده است و بنابراين در توابع اين فايل مانند تابع F2 نيز قابل دسترسي است. توجه كنيد كه متغير k كه توسط تابع F2 مورد دسترسي قرار گرفته است، همان متغير سراسري تعريف شده در فايل File1.C است و در حقيقت هر دوفايل از يك متغير k مشترك استفاده مي نمايند.
در نهايت براي رو شنتر شدن موضوع، با ذكر يك مثال كامل بحث را خاتمه مي دهيم. در اين مثال انواع متغيرها و نحوه كار آنها بررسي شده اند. بررسي دقيق اين مثال كمك زيادي در درك مفاهيم خواهد كرد.

```
#include <stdio.h>

int x = 1; // global variable x

void a() {
  int x = 10 ; // local automatic variable x

  printf("x in function a is %d \n",x);
  x ++;
  printf("x in function a is %d \n",x);
}

void b() {
  static int x = 20 ; // local static variable x
```

```
  printf("x in function b is %d \n",x);
  x ++;
  printf("x in function b is %d \n",x);
}

void  c() {
  printf("x in function c is %d \n",x);
  x ++;
  printf("x in function c is %d \n",x);
}

void main() {
  printf("x in function main is %d \n",x);
  a() ;
  b() ;
  c() ;
  printf("x in function main is %d \n",x);
  a() ;
  b() ;
  c() ;
  printf("x in function main is %d \n",x);
}
```

```
x in function main is 1
x in function a is 10
x in function a is 11
x in function b is 20
x in function b is 21
x in function c is 1
x in function c is 2
x in function main is 2
x in function a is 10
x in function a is 11
x in function b is 21
x in function b is 22
x in function c is 2
x in function c is 3
x in function main is 3
```

همانطور كه در مثال فوق ديده مي شود، در شروع برنامه، از آنجا كه در تابع main متغير x وجود ندارد؛ در نتيجه مقدار متغير سراسري يعني 1 چاپ شده است.اما در هنگام فراخواني تابع a، از آنجا كه متغير x بصورت محلي تعريف شده است، هرگونه استفاده از اين متغير به نمونه محلي آن مراجعه مي كند و از نمونه سراسري استفاده نمي شود. لذا مقدار متغير محلي يعني 10 چاپ شده و سپس مقدار آن يك واحد افزايش يافته است. همين مسئله در مورد تابع b نيز برقرار بوده و در نتيجه مقدار متغير محلي يعني 20 چاپ شده و سپس يك واحد افزايش يافته است. اما بدليل تعريف متغير x بصورت ايستا، پس از پايان تابع، مقدار اين متغير حفظ خواهد گرديد. در فراخواني تابع c، از آنجا كه متغير محلي تعريف نشده است، در نتيجه از همان متغير سراسري x، استفاده گرديده و مقدار آن يعني 1 چاپ شده و سپس يك واحد به آن اضافه شده است. پس از بازگشت به main، مجددا مقدار متغير سراسري x چاپ شده است كه برابر 2 است. اين مسئله نشان مي دهد

كه تغيير اعمال شده در تابع c برروي متغير سراسري x، به تابع main نيز منتقل شده است. فراخوانيهاي بعدي اين توابع نيز مشابه حالت قبلي است، تنها نكته جالب آن است كه از آنجا كه متغير x در تابع a بصورت اتوماتيك تعريف شده، در فراخواني دوم مجددا با 10 مقداردهي اوليه شده است و مقدار قبلي از بين رفته است. اما در تابع b كه متغير x بصورت ايستا تعريف شده است، در فراخواني دوم مقدار قبلي يعني 21 حفظ شده است.

10-7 مقادير پيش فرض براي پارامترها

در بسياري از موارد، توابعي داريم كه داراي تعداد زيادي پارامتر هستند كه در هربار فراخواني بايد آرگومانهاي متناظر با هريك را به تابع ارسال كرد. چنانچه تعداد آرگومانهاي ارسالي، با تعداد پارامترها يكسان نباشد(كمتر يا بيشتر)، يك خطاي كامپايل ايجاد مي گردد.
اما در بعضي موارد، مقادير بعضي از اين پارامترها در اكثر موارد مشخص است و فقط در شرايط خاص تغيير مي كند. بعنوان مثال فرض كنيد تابعي نوشته ايد كه يك پنجره را مكان موردنظر ترسيم ميكند. پارامترهاي متداول براي چنين تابعي عبارتند از:
مختصات شروع، رنگ زمينه، رنگ متن، نوع حاشيه (يك خطي، دوخطي و ...)
اما فرض كنيم پنجره هاي متداول در برنامه ما داراي رنگ زمينه آبي و رنگ متن سفيد با حاشيه دو خطي هستند. در اينصورت در اكثر موارد بجز اطلاعات مربوط به مختصات پنجره، بقيه اطلاعات بصورت تكراري ارسال مي گردند. در چنين مواردي مي توان از پارامترهاي پيش فرض استفاده نمود.
چنانچه يك پارامتر از تابع داراي مقدار پيش فرض باشد، آنگاه تابع فراخوانده مي تواند هيچ آرگوماني متناظر با اين پارامتر ارسال ننمايد. در اينصورت تابع فراخوانده شده از مقدار پيش فرض براي آن پارامتر استفاده مي نمايد. براي تعيين مقدار پيش فرض براي پارامتر، كافي است كه در هنگام تعريف پارامتر با استفاده از عملگر نسبت دهي (=) مقدار پيش فرض را به پارامتر نسبت دهيم. بعنوان مثال به نمونه زير دقت نماييد:

```
#include <stdio.h>

void  sum(int a, int b=0, int c=0) {
  return(a+b+c);
}

void main() {

  printf("sum(5,10,20) = %d \n", sum(5,10,20) );
  printf("sum(5,10) = %d \n", sum(5,10) );
  printf("sum(5) = %d \n", sum(5) );
}
```

```
sum(5,10,20) = 35
sum(5,10) = 15
sum(5) = 5
```

همانطور كه ديده مي شود، در اولين فراخواني، تابع sum با 3 آرگومان فراخواني شده و در نتيجه حاصل جمع آنها را بازگردانده است. اما در دومين فراخواني فقط دو آرگومان ارسال شده و در نتيجه پارامتر سوم يعني c، از مقدار پيش فرض خود يعني صفر استفاده نموده است و حاصل جمع دو عدد بعنوان خروجي باز گردانده شده است. در فراخواني سوم، تنها يك آرگومان ارسال شده و پارامترهاي b و c از مقدار پيش فرض خود يعني صفر، استفاده نموده اند و بنابراين تنها خود عدد ارسالي بعنوان خروجي بازگردانده شده است.
بنابراي با توجه به تعريف فوق، مي توان تابع sum را با سه، دو و يا يك آرگومان فراخواني كرد.
البته در نحوه استفاده از پارامترهاي پيش فرض دو محدوديت وجود دارد:

- تعريف پارامترهاي پيش فرض حتما بايد از سمت راست ترين پارامتر آغاز شده و به سمت چپ ادامه يابد. بعنوان نمونه در مثال فوق چنانچه پارامتر c مقدار پيش فرض نداشته باشد،

پارامتر b نيز قادر به اعلام مقدار پيش فرض نيست. علاوه براين پارامتر a نيز فقط درصورتي مي تواند مقدار پيش فرض داشته باشد كه پارامترهاي b و c هر دو داراي مقدار پيش فرض باشند. يعني اعلان زير خطا مي باشد:

```
int sum(int a=0, int b, int c=0) // compile error
```

- در هنگام فراخواني يك تابع كه داراي پارامترهاي پيش فرض است، آرگومانهاي ارسالي از چپ به راست به پارامترها اختصاص مي يابند. يعني در صورتيكه هنگام فراخواني تابع sum دو آرگومان به آن ارسال شود، اولي به a و دومي به b تخصيص خواهد يافت. تحت هيچ شرايطي نمي توان بدون اينكه مقداري براي b ارسال شود، آرگوماني را براي c ارسال نمود.

10-8 سربارگذاري توابع

در برخي موارد، تابعي داريم كه يك وظيفه خاص را براي چندين نوع داده مختلف انجام مي دهد. بعنوان مثال تابع max را درنظر بگيريد كه دو داده را بعنوان ورودي دريافت و حداكثر آنها را باز مي گرداند. چنانچه اين تابع را براي دو ورودي از نوع عدد صحيح بنويسيم، آنگاه براي ورودي هاي اعشاري درست عمل نخواهد كرد. بنابراين ممكن است برنامه نويس ترجيح دهد دو نسخه از اين تابع داشته باشد: يكي براي اعداد صحيح و يكي براي اعداد اعشاري. اما از آنجا كه نمي توان دو شناسه همنام تعريف كرد، بنابراين ممكن است مجبور شويم از دو نام مجزا استفاده نماييم، مانند intMax براي اعداد صحيح و doubleMax براي اعداد اعشاري. خوشبختانه با استفاده از سربار گذاري توابع، مي توان اين مشكل را حل كرد.
با استفاده از سربار گذاري، مي توان توابعي تعريف كرد كه داراي نام يكسان باشند، ولي ليست پارامترهاي ورودي آنها متفاوت باشد. در اينصورت در هنگام فراخواني تابع، زبان C با توجه به نوع آرگومانهاي ارسالي، تابع مناسب را انتخاب كرده و فراخواني مي نمايد. تفاوت در ليست پارامترهاي ورودي توابع سربارگذاري شده، مي تواند شامل يك يا هر دو مورد زير باشد:

- تفاوت در تعداد پارامترها
- تفاوت در نوع داده يك يا چند پارامتر

بعنوان مثال، مي توانيم دو تابع max داشته باشيم كه اولي دو عدد صحيح و دومي دو عدد اعشاري دريافت نمايند. علاوه براين مي توان تابع max سومي نيز نوشت كه 3 عدد صحيح دريافت و حداكثر آنها را بازگرداند.

```
#include <stdio.h>

int  max(int a, int b) {
  if (a > b) return(a);
  else return(b);
}

double  max(double a, double b) {
  if (a > b) return(a);
  else return(b);
}

int  max(int a, int b, int c) {
  if (a > b)
    if (a > c) return(a);
    else return(c);
  else
     if (b > c) return(b);
```

```
    else return(c);
}

void main() {
  printf("max(8, 10) = %d \n", max(8, 10) );
  printf("max(4.23, 3.712) = %f \n", max(4.23, 3.712) );
  printf("max(15, 3, 20) = %d \n", max(15, 3, 20) );
}
```

```
max(8, 10) = 10
max(4.23, 3.712) = 4.23
max(15, 3, 20) = 20
```

همانگونه كه در مثال ديده مي شود، 3 تعريف مختلف از تابع max با ورودي هاي مختلف ارائه شده است. با فراخواني تابع max، خود زبان C بسته به نوع آرگومانهاي ارسالي، تابع مناسب را تشخيص داده و آن را فراخواني مي نمايد.
نكته آخر اين كه سربار گذاري نمي تواند بر اساس نوع داده خروجي انجام شود. بعبارت ديگر چنانچه دو تابع همنام با ليست پارامترهاي يكسان داشته باشيم كه تنها نوع مقدار خروجي آن دو متفاوت باشد، يك خطاي كامپايل مبني بر استفاده از شناسه هاي يكسان دريافت خواهيم كرد.

10-9 الگوهاي تابعي

الگوهاي تابعي، در حقيقت شكل پيشرفته تري از سربارگذاري هستند. اگر به تابع max در قسمت قبل دقت كنيد، اين تابع يك بار براي نوع داده int و يك بار نيز براي نوع داده double نوشته شده است، اما الگوريتم هر دو تابع يكسان است. در چنين مواردي مي توان بجاي نوشتن چندين تابع سربارگذاري شده با الگوريتم يكسان، تنها يك الگوي تابعي نوشت و ساخت توابع با نوع داده هاي مختلف را بعهده كامپايلر نهاد.
براي روشن شدن موضوع، بحث را با يك مثال ادامه مي دهيم. تابع swap را در نظر بگيريد كه يك دو متغير را دريافت و مقادير آنها را جابجا مي كند. اين تابع داراي كاربردهاي متعددي بوده و ممكن است با نوع داده هاي مختلفي مورد استفاده قرار گيرد. يك روش سربارگذاري تابع swap براي انواع داده هاي مورد نياز است. اما روش بهتر استفاده از يك الگو است.
تعريف الگو با استفاده از كلمه كليدي template صورت مي پذيرد و پس از آن ليست پارامترهاي الگو در داخل علامت <> قرار مي گيرند. پارامترهاي الگو، مشابه پارامترهاي تابع هستند، با اين تفاوت كه بجاي اينكه حاوي **مقدار** باشند، حاوي **نوع داده** هستند. در هنگام استفاده از يك الگو، نوع داده مورد نظر از طريق اين پارامترها به الگو ارسال مي شود. براي تعريف هر پارامتر الگو، از كلمه كليدي class بعلاوه يك شناسه كه نماينده نوع داده است، استفاده مي شود. در اينجا class به معناي هر نوع داده دلخواه است. پس از كلمه كليدي template و ليست پارامترها، تعريف تابع مربوط به الگو شروع مي شود كه همانند ساير توابع است؛ با اين تفاوت كه مي توان از پارامترهاي الگو، همانند نوع داده هاي عادي در تعريف متغيرهاي تابع استفاده نمود. بعنوان مثال به نمونه زير دقت كنيد:

```
template  <class T>
void swap(T &a, T &b) {
  T  temp;

  temp = a;
  a = b;
  b = temp ;
}

void  main() {
  int  a=5 , b=10;
```

```
  double  c = 1.1 , d = 2.2 ;
  char  e = 'a' , f = 'b' ;

  swap(a,b);
  swap(c,d);
  swap(e,f) ;

  printf("a = %d  and  b = %d \n", a, b);
  printf("c = %f  and  d = %f \n", c, d);
  printf("e = %c  and  f = %c \n", e, f);
}
```

```
a = 10  and  b = 5
c = 2.2  and  d = 1.1
e = b  and  f = a
```

در مثال فوق، از شناسه Tبعنوان يك پارامتر الگو استفاده شده است. اين پارامتر نوع داده اي كه تابع swap بايد با آن كار كند را تعيين مي نمايد. هنگامي كه برنامه به فراخواني اول swap مي رسد، ابتدا به دنبال تابعي بنام swap با پارامترهاي int جستجو مي كند. از آنجا كه چنين تابعي وجود ندارد، بدنبال الگويي با نام swap جستجو مي كند. پس از پيدا شدن آن، نوع داده ارسالي به swap يعني int را جايگزين پارامتر T مي نمايد و از روي الگو، تابعي مي سازد كه بجاي تمام پارامترهاي T آن، نوع داده int قرار گرفته است. سپس تابع ساخته شده كامپايل مي گردد. همين عمل براي فراخوانيهاي بعدي swap نيز صورت مي پذيرد و در حقيقت 3 نسخه كامل از تابع swap با پارامترهاي int، double و char توسط كامپايلر ساخته مي شود و هريك در زمان مناسب فراخواني مي گردد.
نكته جالب اين است كه چنانچه بخواهيم تابع swap براي يك نوع داده خاص بگونه اي ديگر عمل كند، كافي است يك تابع (و نه يك الگو) swap براي آن نوع داده خاص بنويسيم. در هنگام فراخواني swap، چنانچه تابعي براي يك نوع داده وجود داشته باشد، هيچگاه از الگو استفاده نمي شود.
الگوها كاربرد بسيار زيادي براي تعريف توابع كلي دارند كه با نوع داده هاي مختلفي عمل مي كنند. در فصلهاي بعدي نمونه هاي ديگري از اين كاربرد را خواهيم ديد.

10-10 توابع درون برنامه اي

فراخواني يك تابع داراي سربار زماني و حافظه اي مي باشد. با فراخواني هر تابع بايد براي متغيرهاي محلي و پارامترهاي آن حافظه تخصيص داده شود، آرگومانهاي ارسالي در پارامترهاي تابع كپي شوند و ... كه همگي اين موارد باعث صرف زمان و حافظه مي شوند. به همين دليل براي انجام عمليات كوتاه، استفاده از تابع ايده چندان خوبي نيست؛ چراكه گرچه خوانايي برنامه بالا مي رود، اما براي انجام يك كار كوچك، هزينه زيادي به سيستم تحميل مي گردد. براي رفع اين مشكل، زبان C از توابع درون برنامه اي يا inline استفاده مي كند. اگر قبل از شروع تعريف تابع (يعني قبل از نوع مقدار بازگشتي) از كلمه كليدي inline استفاده شود، به كامپايلر توصيه مي شود كه هر فراخواني تابع مورد نظر را با يك كپي صريح از كد تابع جايگزين نمايد، تا ديگر نيازي به صرف زمان براي فراخواني تابع نباشد. بعنوان مثال به نمونه زير دقت كنيد:

```
inline  int  square(int n) {
  return (n*n) ;
}

void main() {
  int  a, b ;

  a = square(5) ;
  b = square(2*a+3) ;
```

}

در اينحالت كامپايلر تابع mainرا بصورت زير تغيير مي دهد:

```
void main() {
  int  a,b ;

  a = (5) * (5) ;
  b = (2*a+3) * (2*a+3) ;
}
```

البته استفاده از inline، گرچه باعث بالا رفتن سرعت اجراي برنامه مي شود، اما حجم آن را بالا مي برد؛ چرا كه بجاي هر فراخواني تابع، يك كپي از آن را قرار مي دهد. در نتيجه استفاده از اين مشخصه، فقط در توابع كوچك قابل قبول است. در حقيقت C از اين مشخصه براي توابع بزرگ صرفنظر مي كند.

10-11 توابع بازگشتي

يكي از مهمترين مفاهيم علم كامپيوتر، توابع بازگشتي هستند كه كاربرد بسيار زيادي در حل مسائل دارند. توابع بازگشتي، توابعي هستند كه خود را مجددا فراخواني مي كنند. اين توابع به دو دسته تقسيم مي شوند:

- توابع بازگشتي مستقيم : تابعي مانند A، خودش را فراخواني نمايد.
- توابع بازگشتي غير مستقيم : تابعي مانند A، تابع ديگري مانند B را فراخواني نمايد و سپس B مجددا تابع A را فراخواني نمايد.

ما در اين فصل، منحصرا به بررسي توابع بازگشتي مستقيم خواهيم پرداخت.
اما چرا يك تابع بايد خودش را فراخواني نمايد؟ در بعضي مسائل (كه عموما مسائل پيچيده اي نيز هستند)، مي توان يك نمونه از مسئله را با استفاده از نمونه ساده تر يا كوچكتري از همان مسئله حل كرد. بنابراين، تابع مي تواند براي حل مسئله بزرگتر، نسخه جديدي از خودش را براي حل مسئله كوچكتر فراخواني نمايد و سپس با استفاده از نتايج بدست آمده، مسئله بزرگ را حل نمايد. مسلم است كه تابعي كه براي حل مسئله كوچك فراخواني شده است نيز از همين مكانيزم استفاده كرده و مجددا خودش را براي حل يك مسئله كوچكتر فراخواني خواهد كرد. اما اين فراخوانيها تا چه زماني ادامه پيدا مي كند؟ هر مسئله بازگشتي داراي يك حالت **پايه** است كه حل آن بديهي بوده و نيازي به حل يك مسئله كوچكتر ندارد. بنابراين تابع بازگشتي حتما بايد داراي يك شرط براي بررسي حالت پايه باشد، كه به آن شرط توقف گفته مي شود. هنگامي كه تابع به حالت پايه مي رسد، فراخواني بازگشتي را متوقف كرده و حل بديهي مسئله را براي فراخواني قبلي تابع باز مي گرداند. سپس اين بازگرداندن مقادير رو به عقب تكرار مي شود تا هنگامي كه به نسخه اوليه تابع برسد و در نهايت جواب نهايي به تابع اصلي بازگردانده شود.
اكنون براي روشن شدن موضوع به بررسي يك مثال كلاسيك در زمينه توابع بازگشتي، يعني محاسبه فاكتوريال مي پردازيم. قبلا نحوه محاسبه فاكتوريال يك عدد را بصورت زير بيان كرديم :

$n! = 1 \times 2 \times \ldots \times n$

تابع مربوط به فاكتوريال نيز با استفاده از يك حلقه تكرار كه اعداد 1 تا n را در يكديگر ضرب مي كرد، نوشته شد. اما اگر به تعريف فاكتوريال دقت كنيم، درمي يابيم كه فاكتوريال عددي مانند 5 را مي توان از ضرب عدد 5 در فاكتوريال 4 بدست آورد. بعبارت بهتر $5! = 5 \times 4!$. استدلال مشابهي براي $4!$ برقرار است. بطور كلي مي توان گفت:

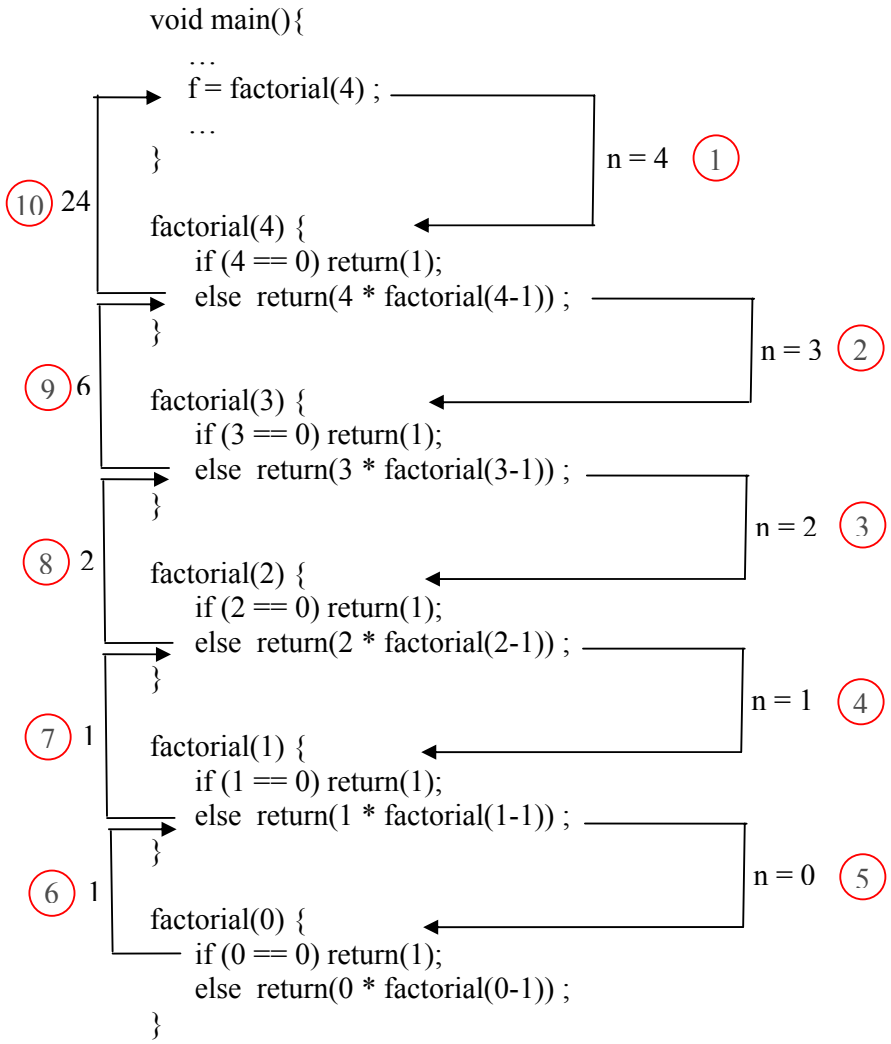
$$n! = \begin{cases} n \times (n-1)! & n > 0 \\ 1 & n = 0 \end{cases}$$

همانطور كه ديده مي شود، حالت پايه در اين مسئله، حالت n=0 است؛ چرا كه در اين حالت حل مسئله بديهي بوده و جواب برابر 1 است.
تابع بازگشتي فاكتوريال بصورت زير نوشته مي شود:

```
long int factorial(int  n) {
```

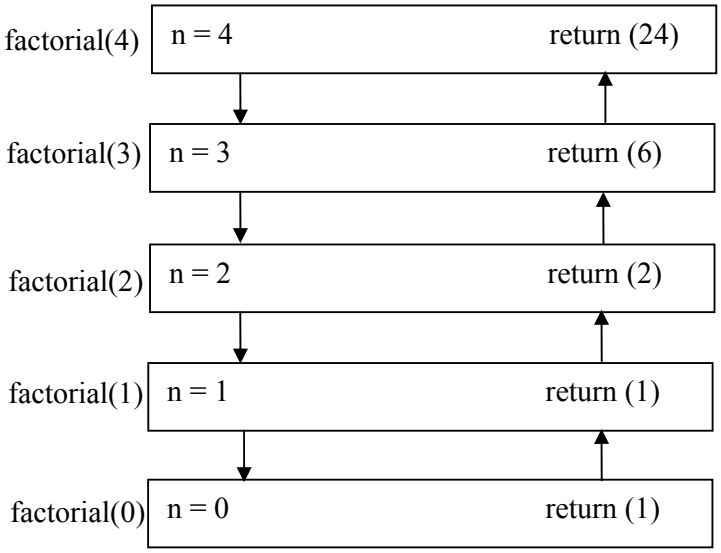
```
  if (n ==0) return(1) ;
  else return(n * factorial(n-1));
}
```

همانطور كه ديده مي شود، توابع باز گشتي بسيار ساده و كوچك مي باشند، اما درك نحوه كار آنها كمي مشكل است. قبل از توضيح نحوه كار اين تابع، توجه كنيد كه هنگامي كه يك تابع خودش را فراخواني مي كند، يك نسخه جديد از آن تابع بوجود مي آيد. بدين معنا كه يك نسخه جديد از كليه پارامترها و متغيرهاي محلي تابع ايجاد مي شود و عمليات برروي اين متغيرهاي جديد انجام مي شود(بدون اينكه تغييري در متغيرهاي فراخواني قبلي ايجاد شود). البته براي صرفه جويي در حافظه، نسخه جديدي از دستورات برنامه ايجاد نمي شود.
شكل زير نحوه انجام عمليات را براي فراخواني factorial(4) نشان مي دهد. اعداد روي فلشها، مقدار ارسالي به تابع و يا مقدار بازگشتي از تابع رانشان مي دهند، و اعداد داخل دايره، ترتيب فراخواني و بازگشت را مشخص مي نمايند.



همانطور كه ديده مي شود، دنبال كردن نحوه كار توابع بازگشتي قدري دشوار است و بهمين دليل، اشكال زدايي آنها نيز نسبتا مشكل است. اما معمولا خود توابع ساده تر و واضح تر از نمونه غير بازگشتي خود هستند.

روش ديگري نيز براي دنبال كردن توابع باز گشتي وجود دارد. در اين روش هر بار فراخواني تابع، با يك مستطيل نشان داده مي شود كه در داخل آن مقادير پارامترها و متغيرهاي محلي و همچنين مقادير بازگشتي تابع نوشته مي شود. هر بار فراخواني مجدد با يك فلش به مستطيل بعدي نشان داده مي شود و در هنگام بازگشت نيز يك فلش به مستطيل قبلي رسم مي شود. اين روش به افراد كمك مي كند مقادير متغيرهاي محلي و پارامترها را در حين فراخوانيهاي متعدد مشاهده نمايند. به مثال زير توجه كنيد:



اكنون براي روشنتر شدن موضوع به چند مثال ديگر توجه نمايد:

برنامه 7) تابعي بنويسيد كه تركيب n به k را محاسبه نمايد.

حل) قبلا اين تابع را به روش عادي و با استفاده از فاكتوريال نوشته ايم. اما متاسفانه اين روش حل چندان مناسب نيست، چرا كه در حالاتي مانند n=20 و k=18، گرچه حاصل نهايي عدد چندان بزرگي نيست، ولي از آنجا كه !20 و !18 اعداد بسيار بزرگي هستند، محاسبه امكان پذير نيست. در رياضيات يك فرمول رياضي بازگشتي براي محاسبه تركيب وجود دارد كه بصورت زير است:

$$\binom{n}{k} = \begin{cases} 1 & k = 0 \ or \ n = k \\ \binom{n-1}{k-1} + \binom{n-1}{k} & otherwise \end{cases}$$

بنابراين مي توان تابع بازگشتي آن را بصورت زير نوشت:

```
int  combination(int n, int k) {
  if (k==0  ||  n==k)  return(1);
  else return( combination(n-1,k-1) + combination(n-1,k) ) ;
}
```

برنامه 8) تابعي بنويسيد كه nامين عدد فيبوناچي را باز گرداند.
حل) دنباله فيبو ناچي بصورت زير تعريف مي شود:

1 1 2 3 5 8 13 21 …

دو عدد اول فيبو ناچي برابر 1 هستند و از آن پس هر عدد برابر جمع دو عدد قبلي است. بنابراين تابع آن به راحتي و به شكل زير قابل تعريف است:

```
int fibonacci(int n) {
  if (n==1 || n==2) return(1);
  else return( fibonacci(n-1) + fibonacci(n-2) );
}
```

برنامه 9) تابعي بنويسيد كه دو عدد را دريافت و بزرگترين مقسوم عليه مشترك آن دو را بازگرداند.

حل) حتما با روش محاسبه بزرگترين مقسوم عليه مشترك به روش نردباني آشنايي داريد. اين روش، يك روش بازگشتي بصورت زير است:

$$\gcd(x,y)=\begin{cases} x & y=0 \\ \gcd(y, x\%y) & y\neq 0 \end{cases}$$

بنابراين تابع نهايي بصورت زير خواهد شد:

```
int gcd(int x, int y) {
  if (y == 0)  return(x);
  else return( gcd(y, x%y) );
}
```

10-11-1 مقايسه روشهاي تكراري و بازگشتي

همانطور كه قبلا ديده شد، الگوريتم فاكتوريال را مي توان به دوصورت تكراري (با حلقه تكرار) و بازگشتي نوشت. آيا اين مسئله براي ساير الگوريتمهاي بازگشتي نيز برقرار است؟ جواب مثبت است. هر الگوريتم بازگشتي را مي توان بصورت غير بازگشتي يا تكراري نيز نوشت، گرچه ممكن است راه حل آن پيچيده تر و وضوح آن نيز كمتر باشد. اما كدام روش بهتر است؟
همانطور كه ديديد، در الگوريتمهاي بازگشتي، با هربار فراخواني تابع يك نسخه جديد از متغيرها ايجاد مي شود. به همين دليل معمولا الگوريتمهاي بازگشتي حافظه بيشتري را نسبت به الگوريتمهاي تكراري مصرف مي نمايند. علاوه براين الگوريتمهاي بازگشتي معمولا از نظر زمان اجرا نيز كندتر هستند. دليل اين مسئله سربار ناشي از فراخواني مكرر تابع است. هربار فراخواني يك تابع، باعث مي شود كه عمليات خاصي (همانند ذخيره آدرس برگشت از تابع) انجام گيرد كه سربار زماني كمي را ايجاد مي كند. اما فراخوانيهاي متعدد، باعث ايجاد سربار زماني قابل توجهي مي گردد.
با توجه به اين نكات منفي، چرا از توابع بازگشتي استفاده كنيم؟ تنها هنگامي از توابع بازگشتي استفاده نماييد كه طبيعت مساله بگونه اي باشد كه توسط روشهاي بازگشتي راحتتر حل شود و الگوريتم حاصل ساده تر و واضحتر باشد. در بسياري از مسائل، روش بازگشتي بسيار ساده و آسان بوده و درك و اشكالزدايي آن بسيار ساده است، درحاليكه راه حل غيربازگشتي به راحتي به ذهن نمي رسد و يا بسيار پيچيده است. اما در مواردي كه كارايي اهميت زيادي دارد، هرگز از بازگشت استفاده ننماييد.

با آرزوی موفقیت
سعید ابریشمی