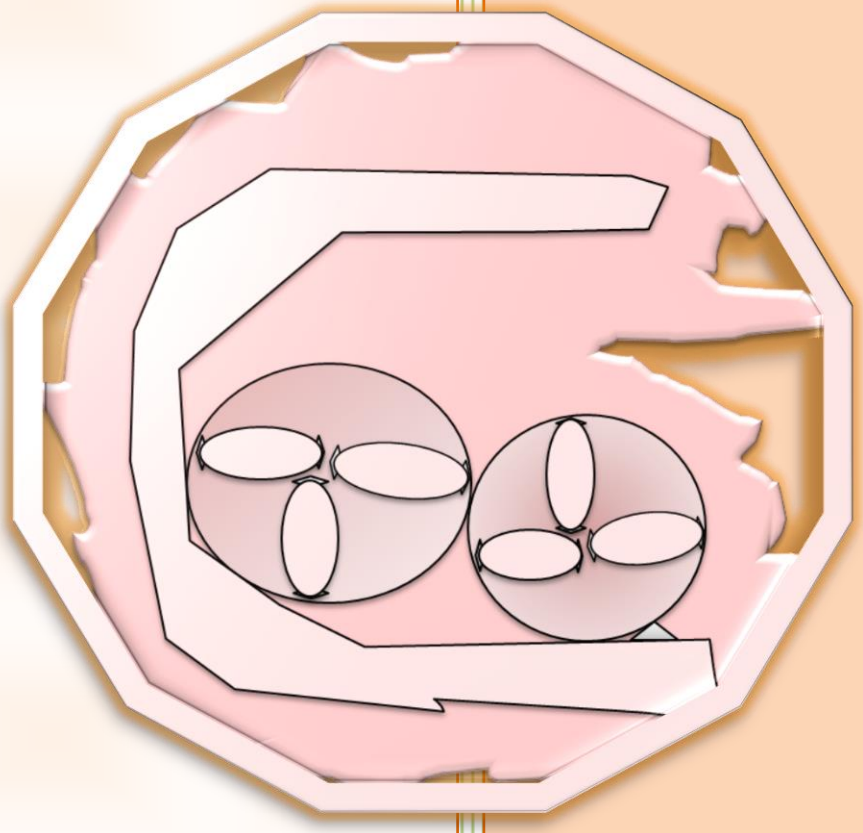
2019

با C++ آشنا شویم – نسخهٔ رایگان

version 5.00



تاریخ توزیع: فروردین ۱۳۹۸

توجه:

این، نسخهٔ رایگان کتاب «با C++ آشنا شویم» است و شامل تنها ۱۴۰ صفحه از نسخهٔ ۹۸۰ صفحه‌ای کامل است. استفادهٔ آموزشی از این نسخه، رایگان است، و به علاوه، توزیع فایل‌های دست‌نخوردهٔ و کامل این نسخه آزاد می‌باشد. نسخهٔ کامل کتاب، رایگان نیست و **برای خرید حق استفاده از نسخهٔ کامل کتاب، و دریافت فایل آن، باید با (مدیر) کانال تلگرامی @cplusplus یا با ایمیل:**

edukadoj@gmail.com

تماس بگیرید تا مقدار هزینهٔ پرداختی و لینک پرداخت در اختیار شما قرار گیرد و پس از تأیید اطلاعات پرداخت، فایل نسخهٔ کامل برای شما ارسال گردد.

تهیه و استفاده از نسخهٔ کامل کتاب با پذیرش شرایط زیر خواهد بود:

۱) به ازای هر شخص حقیقی که قرار است به این نسخه از کتاب، دسترسی داشته باشد و از آن «استفاده» کند، باید هزینه‌ای به حساب‌های مشخص‌شده، که پس از تماس از طریق راه‌های فوق در اختیار درخواست‌کننده قرار می‌گیرد، واریز شده باشد. به چنین هزینه‌ای، «هزینهٔ استفاده» و به شخص حقیقی‌ای که هزینهٔ استفاده، برای استفادهٔ او، پرداخت شده است «پرداخت‌کننده» می‌گوییم. میزان پرداخت پس از تماس معین می‌گردد و ممکن است امکان این پرداخت تنها از داخل ایران وجود داشته باشد.

۲) پس از پرداخت «هزینهٔ استفاده» و مشخص‌شدن نام «پرداخت‌کننده»، نسخهٔ منحصربه‌فردی از این کتاب (شامل نام «پرداخت‌کننده» و کد)، به صورت یک فایل pdf، برای دارندهٔ اطلاعات پرداخت، ارسال خواهد گشت. پرداخت‌کننده، که فایل را دریافت می‌کند، مسئول حفاظت از این فایل، برای ممانعت از نشر غیر مجاز است و موظف به جبران خسارات در صورت نشر غیرمجاز نسخهٔ منحصربه‌فردی است که در اختیار دارد.

۳) پرداخت هزینهٔ استفاده، به معنای خرید محتوا و دریافت اجازهٔ نشر نیست، و تنها، به معنای خرید اجازهٔ «استفادهٔ» یک شخص حقیقی از کتاب است.

۴) معنای «**استفاده**» از این نسخه از کتاب، در این متن، محدود به موارد زیر است:

- مطالعه به منظور آموختن.
- به‌کاربردن یا تغییر کدها برای یادگیری.
- به‌کاربردن کدها، در برنامه‌هایی که شخص می‌نویسد، به شرط آن که هدف، برنامه‌نویسی باشد؛ نه مثلاً توزیع کدها.
- به‌کاربردن کتاب، برای تدریس، به شرط آن که نوعی توزیع غیرمستقیم کتاب، بین کسانی که هزینهٔ استفاده از آن را پرداخت نکرده‌اند، محسوب نگردد؛ برای مثال، عیناً مطالب خود کتاب ارائه نشود به طوری که بخش‌های قابل‌توجهی از کتاب به صورت جزوه درآید؛ همچنین، توزیع اصل این نسخه از کتاب در کلاس و مانند آن، بدون پرداخت «هزینهٔ استفاده» برای هر استفاده‌کننده مجاز نیست.

۵) هرگونه توزیع کل یا بخشی از کتاب، یا هرگونه عملی که نوعی توزیع محسوب گردد (مثل فیلمبرداری از صفحات متعدد کتاب)، مجاز نیست.

۶) پرینت الکترونیک (مثلاً به صورت pdf)، یا تغییر فرمت فایل کتاب، مجاز نیست.

۷) با پرداخت هزینه، هیچ تعهدی برای ارائه نسخهٔ قابل‌پرینت، برای صاحب اثر یا توزیع‌کنندهٔ مجاز آن، به وجود نمی‌آید. با این حال، پرینت (بر کاغذ) تنها برای استفادهٔ آموزشی پرداخت‌کننده، بلامانع است.

۸) با پرداخت هزینهٔ استفاده، هیچ تعهدی برای ارائه نسخهٔ متنی برنامه‌ها، برای صاحب اثر یا توزیع‌کنندهٔ مجاز آن، به وجود نمی‌آید. با این حال، سعی خواهد شد، فایل‌های متنی برنامه‌ها، در اختیار پرداخت‌کننده قرار گیرند؛ اما در چنین صورتی، توزیع چنین فایل‌هایی مجاز نیست و استفاده از آن‌ها، تنها در راستای استفاده از کتاب (و به عنوان بخشی از آن) مجاز خواهد بود.

۹) «پرداخت‌کننده»، در هر زمان، می‌تواند «حق استفاده» را، همراه با تمام فایل‌های مربوط به کتاب، که در اختیار دارد، در اختیار شخص دومی بگذارد به شرط آن که:

- اولاً آن شخص دوم، پایبند به شرایط مذکور در اینجا باشد (به جز این که نیازی به پرداخت هزینهٔ استفاده ندارد) و:
- ثانیاً شخص اول، نسخه‌ای از کتاب (یا بخشی از آن یا فایل‌های متنی برنامه‌ها) را نگه ندارد.
- شخص دوم بپذیرد که مجاز نیست «اجازهٔ استفاده» را در اختیار شخص دیگری بگذارد.

۱۰) در صورتی که پرداخت‌کنندهٔ هزینه، اطلاعات مربوط به پرداخت خود را گم کند و یا به هر علت دیگری، قادر به اثبات پرداخت نباشد، به صاحب اثر، یا توزیع‌کنندهٔ مجاز، اجازهٔ استفاده از آن مبلغ را به عنوان «حمایت مالی» می‌دهد، و در ضمن، با توجه به عدم توانایی اثبات پرداخت، صاحب اثر یا توزیع‌کنندهٔ مجاز، مسئولیتی در قبال آن پرداخت (مانند تحویل فایل کتاب یا بازگرداندن مبلغ) نخواهد داشت. با این حال، «استفادهٔ» پرداخت‌کننده از این نسخهٔ کامل (اگر به شکلی در اختیارش قرار گرفته باشد)، طبق شرایط مذکور، از نظر اخلاقی، بلامانع است.

۱۱) در صورتی که پرداخت‌کنندهٔ هزینه، هزینه را به طور ناقص پرداخت کرده باشد، یا اطلاعات پرداخت را، حداکثر پس دو ماه از زمان پرداخت، به آدرس‌هایی که در ادامه می‌آیند، ارائه نداده باشد، به صاحب اثر یا توزیع‌کنندهٔ مجاز، اجازهٔ استفاده از آن مبلغ را، به عنوان

«حمایت مالی» می‌دهد، و در ضمن، با توجه به عدم پرداخت کامل یا عدم ارائهٔ اطلاعات پرداخت، صاحب اثر یا توزیع‌کنندهٔ مجاز، مسئولیتی در قبال آن پرداخت (مانند تحویل فایل کتاب یا بازگرداندن مبلغ) نخواهد داشت. با این حال، سعی می‌کنیم در صورت امکان و با تماس پرداخت‌کننده، هزینهٔ پرداخت‌شده، بازگردانده شود.

۱۲) پرداخت کننده باید بپذیرد که حق‌استفاده‌ای که می‌خرد، انحصاری نیست و ممکن است افراد دیگری، همان حق را، با پرداخت هزینه یا روش‌های دیگری، از صاحب اثر دریافت کنند.

فهرست مطالب

مقدمه

زبان C++، یکی از دروازه‌های ورود به دنیای برنامه‌نویسی است و بسیاری از برنامه‌نویس‌های حرفه‌ای به C++ مسلطند. شاید بتوان گفت، ساختار و دستورات زبان C++، به نوعی معیار تبدیل شده‌اند و در زبان‌های دیگری مانند C# و جاوا نیز به کار می‌روند.

C++، از زبان C توسعه یافته است؛ یعنی تقریباً هر برنامهٔ C، یک برنامهٔ C++ نیز هست. در زبان C، اگر n، یک متغیر با مقدار 10 باشد، دستور n++، مقدار متغیر را یکی زیاد می‌کند، یعنی، مقدار متغیر، پس از اجرای این دستور، می‌شود 11. نام C++ هم، از همین ویژگی گرفته شده است، یعنی C++، کمی بیشتر از C است. ریک ماشیتی[۱] پیشنهاددهندهٔ این نام بوده است.

زبان C، تحت سیستم عامل یونیکس[۲]، به وسیلهٔ دنیس ریچی[۳]، در دههٔ ۷۰ میلادی (با تکامل زبان‌های ابتدایی‌تر پیشین) به وجود آمد و به تدریج تکامل یافت.

C++، در سال ۱۹۷۹ میلادی، توسط بیارن استروستروپ[۴] دانمارکی توسعه یافت. او، در ابتدا، آن را «C همراه با کلاس» نامید، که بعدها، این نام، به C++ تغییر یافت. «کلاس» یک ابزار بسته‌بندی بسیار قوی است که به C اضافه شده است.

در این کتاب، زبان C++، همراه با برنامه‌هایی که خواننده آن‌ها را، در رایانه، خواهد آزمود، به تدریج، آموزش داده می‌شود. سعی بر این است که تا حد امکان، مطالب ساده بیان شوند تا برای فردی، حتی فردی که با برنامه‌نویسی آشنایی ندارد، نیز، قابل‌فهم باشد. تلاش شده است که برنامه‌هایی که به عنوان مثال ارائه شده‌اند، مینیمال باشند، یعنی تنها هدف برنامه، نشان‌دادن عملکرد یک دستور یا کد C++ باشد و هدف دیگری را دنبال نکند. بنابراین، از برنامه‌هایی که ارائه خواهند شد، نباید انتظار داشت که برای کاری جز آموزش مفید باشند، یا هدفی جز این را دنبال کنند. موضوع اصلی، زبان C++ است، نه آموزش مهارت‌های برنامه‌نویسی با این زبان. نسبت موضوع این کتاب به آموزش مهارت برنامه‌نویسی، مانند نسبت دستور زبان فارسی به ادبیات فارسی است. لزوماً هر کس با دستور زبان فارسی آشنا باشد، نمی‌تواند متن‌های ادبی برجسته بنویسد.

در این کتاب، هدف آموزش سرفایل‌های استاندارد (C یا) C++، توابع، کلاس‌ها و یا تمپلیت‌های آماده نیست مگر آن که:

۱) امکانی را در اختیار برنامه‌نویس بگذارند که خود برنامه‌نویس، نمی‌تواند، به طور مستقل، به آن دست یابد. مانند کلاس‌های مربوط به فایل.
۲) نوشتن معادل برای آن‌ها آسان نباشد و یا دردسرزا باشد؛ مانند تمپلیت vector.
۳) آشنایی با آن، ضروری باشد مثل کلاس string یا vector.

---

[۱] Rick Mascitti
[۲] Unix
[۳] Dennis Ritchie
[۴] Bjarne Stroustrup

در این موارد نیز، گاه، سعی شده است، یک عملکرد مفید یا پرکاربرد، به صورت یک تابع یا کلاس مستقل ارائه گردد بدون آن که جزئیات آن تابع یا کلاس بررسی شوند.

در پایان برخی از بخش‌ها در این کتاب، سؤالات یا تمرین‌هایی قرار گرفته‌اند. سؤالات لزوماً برای پاسخ‌گویی بر اساس آنچه گفته شده است، نیستند؛ بلکه، بیشتر، به منظور برجسته‌کردن برخی نکات در ذهن مخاطب طرح شده‌اند. اما تمرین‌ها، به گونه‌ای هستند که خواننده، باید تلاش کند، با توجه به آنچه در متن قبل از هر تمرین آمده است، به هدف مشخص‌شده در تمرین برسد.

هرچند هدف این کتاب، آموزش زبان C++ است، اما باید توجه داشت که متن آن، مانند کتاب‌های قانون استاندارد C++ نیست و ممکن است برای آموزش آسان‌تر یک مفهوم، از دقت‌های شدید قانونی پرهیز شود.

در نگارش این کتاب، تلاش و دقت زیادی صورت گرفته است تا خطا و اشکالی در بین نباشد، اما بی‌تردید، آنچه در برابر شماست، خالی از اشکالات نگارشی و علمی نیست و از خوانندگان درخواست می‌گردد که این اشکالات را، به آدرس ایمیل edukadoj@gmail.com ارسال نمایند؛ تا در نسخه‌های بعدی تصحیح گردند.

نسخهٔ حاضر از این کتاب رایگان نیست و با هدف توزیع رایگان ویرایش نشده است و در صورتی که این کتاب به صورت رایگان به دست شما رسیده است، برای پرداخت هزینهٔ استفاده از آن باید با edukadoj@gmail.com تماس بگیرید.[۵]

[۵] با گذشت ۴۰ سال از تاریخ انتشار این نسخه، پرداخت هزینه و تماس با ما نیازی نیست.

فصل اول – آشنایی با رایانه و محیط سیستم عامل

در این فصل، مقدماتی در مورد رایانه، سیستم‌عامل و مفاهیم ابتدایی بیان خواهد شد. از آنجا که به نظر می‌رسد، اغلب خوانندگان، با این مباحث ابتدایی آشنایی داشته باشند؛ می‌توانید به سرعت از این فصل بگذرید.

آشنایی ابتدایی

سیستم‌عامل

سیستم‌عامل، نرم‌افزاری است که ارتباط بین سخت‌افزار و نرم‌افزارهای کاربردی را میسّر می‌سازد. آنچه در این نوشته اهمیت دارد، وجود سیستم‌عاملی است که به وسیلهٔ آن بتوان از یک کامپایلر C++ استفاده کرد. کسانی که انتخاب کرده‌اند از سیستم‌عامل‌های لینوکسی مانند اوبونتو[۶] استفاده کنند، به احتمال زیاد، نیازی به معرفی و آشنایی با جزئیات ابتدایی‌ای که در این فصل قرار است به آن‌ها بپردازیم، ندارند. توضیحات این کتاب، بر اساس ویندوز 10 نسخهٔ ۶۴بیتی و اوبونتو 18.04 نسخهٔ ۶۴بیتی خواهد بود.

حافظهٔ رایانه

در رایانه، دو نوع حافظه داریم، حافظهٔ موقت، یا رم[۷]، و حافظهٔ دائم. همهٔ اطلاعات حافظهٔ موقت، به محض قطع برق یا خاموش‌شدن رایانه، پاک می‌شوند؛ اما دسترسی به این حافظه، بسیار سریع‌تر از حافظهٔ دائم است. در برنامه‌نویسی، بیشتر با رم سر و کار داریم؛ زیرا نرم‌افزارها هنگام اجرا باید در رم قرار بگیرند.

حافظهٔ دائم، برای ذخیرهٔ طولانی‌مدت اطلاعات به کار می‌رود. وقتی یک فایل را ذخیره[۸] می‌کنیم، در حقیقت، آن را (از رم) به حافظهٔ دائم می‌فرستیم و در آنجا نگهداری می‌کنیم.

برای این که ببینید اندازهٔ حافظهٔ موقت رایانهٔ شما چه قدر است، در سیستم‌عامل ویندوز ۱۰، بر آیکان This PC در دسکتاپ[۹] راست‌کلیک[۱۰] کنید و از منویی که باز می‌شود، تب Properties را انتخاب کنید:

---

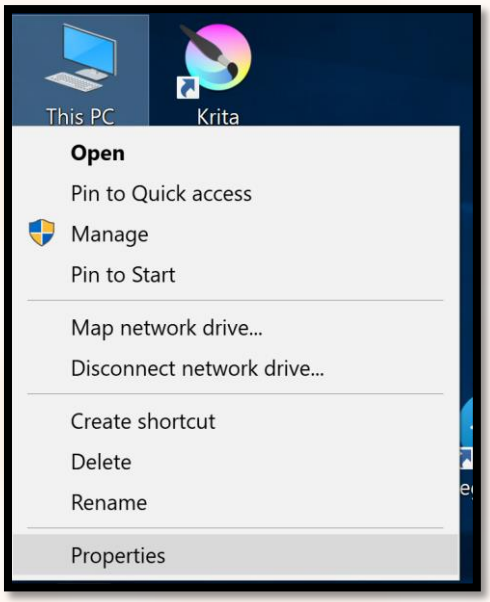
[۶] Ubuntu

[۷] RAM

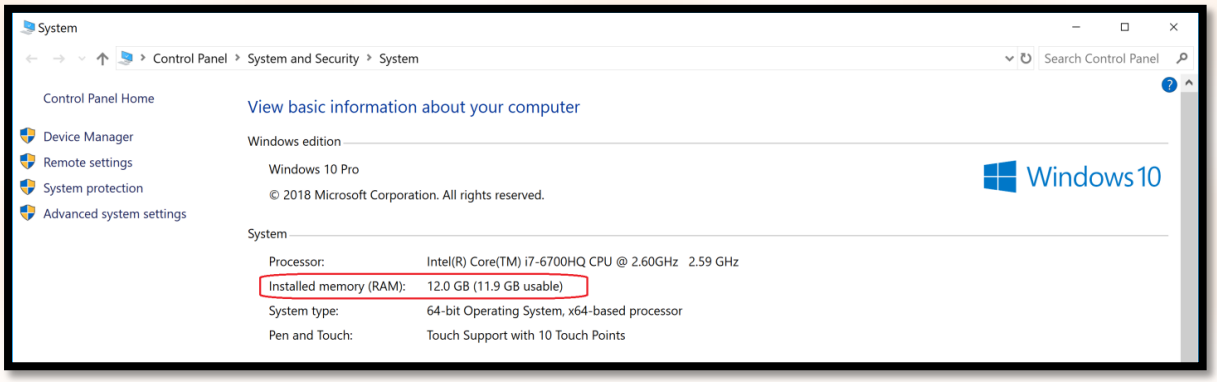
[۸] save

[۹] desktop

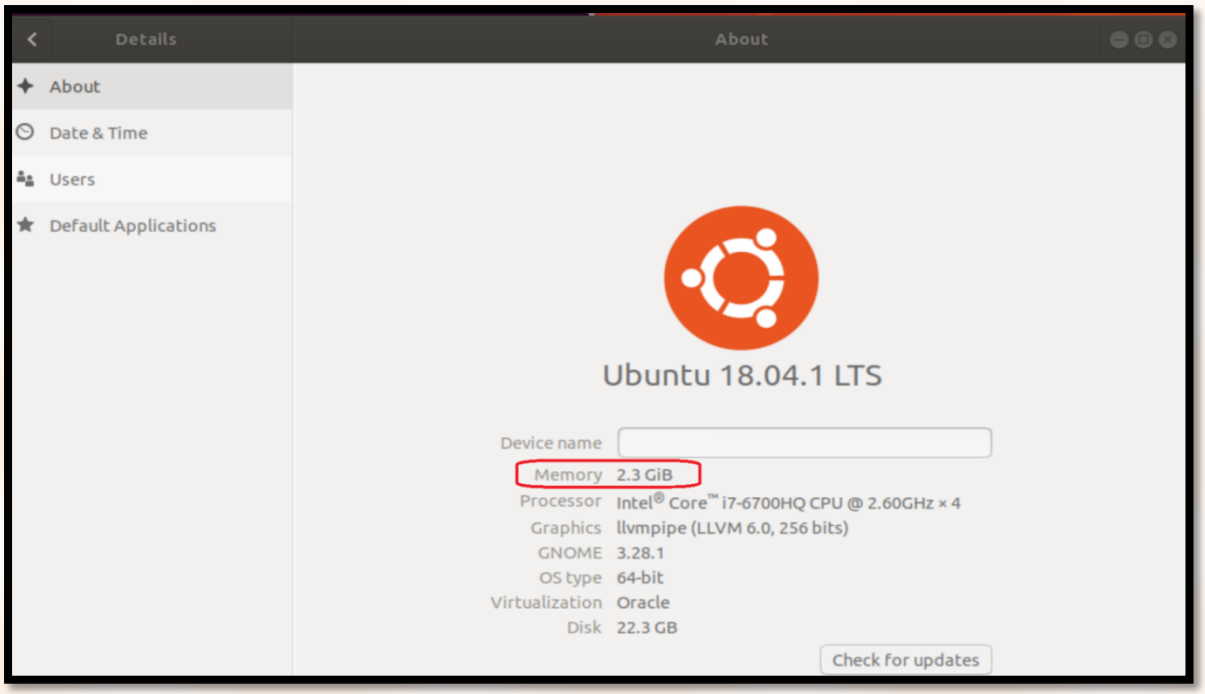
[۱۰] right-click



پنجرهٔ زیر باز می‌شود:



که در این تصویر، میزان حافظه، ۱۲ گیگابایت نشان داده شده است. در اوبونتو، هم با دستور `free -m` و هم به صورت گرافیکی، از قسمت تنظیمات، می‌توان میزان رم رایانه را پیدا کرد:



در تصویر بالا می‌بینید که اندازهٔ رم ۲/۳ گیبی‌بایت است. اما مفهوم این اندازه‌ها و واحدهایشان چیست؟ کوچک‌ترین واحد حافظه بیت[۱۱] است. یک بیت می‌تواند تنها دو مقدار ۰ یا ۱ داشته باشد. از ترکیب هشت بیت، یک بایت[۱۲] ساخته می‌شود. هرچند، بیت، کوچک‌ترین واحد حافظه است، اما معمولاً، حافظه، بر حسب بایت بیان می‌شود و یک برنامه‌نویس، معمولاً، بایت را کوچک‌ترین بخش از حافظه در نظر می‌گیرد. پس هشت بیت، به شکلی، کوچک‌ترین واحد حافظه است. این که چرا هشت بیت و مثلاً شش بیت نه، به استانداردها، تاریخ توسعهٔ نرم‌افزارها، محدودیت‌های سخت‌افزارها و... برمی‌گردد. سیستم‌هایی بوده‌اند که از واحدهای شش‌بیتی استفاده می‌کرده‌اند، و بنابراین، جادوی خاصی در هشت بیت نیست و این، تنها یک قرارداد پذیرفته شده است.

در گذشته، و به صورت تاریخی، به ۱۰۲۴ بایت، یک کیلوبایت[۱۳] گفته می‌شد؛ اما در استانداردهای جدید، یک کیلوبایت، برابر با ۱۰۰۰ بایت است. در عوض، به ۱۰۲۴ بایت، یک کیبی‌بایت[۱۴] گفته می‌شود. جدول زیر واحدهای جدید، برای اندازهٔ حافظه را نشان می‌دهد:

نماد	اندازه بر حسب بایت	نام
KB	۱۰۰۰	kilobyte
KiB	۱۰۲۴	kibibyte
MB	۱۰۰۰ × ۱۰۰۰	megabyte
MiB	۱۰۲۴ × ۱۰۲۴	mebibyte
GB	۱۰۰۰ × ۱۰۰۰ × ۱۰۰۰	gigabyte
GiB	۱۰۲۴ × ۱۰۲۴ × ۱۰۲۴	gibibyte

[۱۱] bit
[۱۲] byte
[۱۳] kilobyte
[۱۴] kibibyte

از آنجا که این استانداردهای جدید، چندان مورد استقبال نبوده‌اند، سیستم‌عامل ویندوز، تاکنون از آن استفاده نکرده است و در ویندوز، همچنان ۱ کیلوبایت، ۱۰۲۴ بایت است. سیستم‌عامل‌های لینوکسی، بیشتر، از استانداردهای جدید استفاده می‌کنند. به هر حال، اگر جایی به «کیلوبایت»، «مگابایت» و... بربخوریم، لازم است بدانیم منظور چیست. برای نمونه، در بیان حجم فلش‌مموری‌ها و هارددیسک‌ها، توسط شرکت‌های سازنده، معمولاً استانداردهای جدید به کار گرفته می‌شوند؛ اما ویندوز، حجم یک فلش‌مموری ۸ گیگابایتی را کم‌تر از ۸ گیگابایت نشان می‌دهد؛ زیرا شرکت‌های سازندهٔ فلش‌مموری و ویندوز، تعریف‌های متفاوتی برای گیگابایت دارند. از آنجا که ۱۰۲۴ توانی از ۲ است:

$$۱۰۲۴ = ۲^{۱۰} = ۲ \times ۲ \times ۲ \times ۲ \times ۲ \times ۲ \times ۲ \times ۲ \times ۲ \times ۲$$

این عدد، برای محاسبات مربوط به رایانه مناسب‌تر است. هرچند هیچ جادوی در توان دهم ۲ وجود ندارد که انتخاب آن را الزامی کند و این، تنها یک قرارداد است.

حافظهٔ دائم رایانه، بسیار بزرگ‌تر و اغلب بسیار کندتر از رم است و این کندی، دلیل اصلی نیاز به حافظهٔ سریع رم است. در آینده، هرگاه بگوییم «حافظه» منظور ما، رم است مگر آن که به صراحت به چیز دیگری تأکید کنیم.

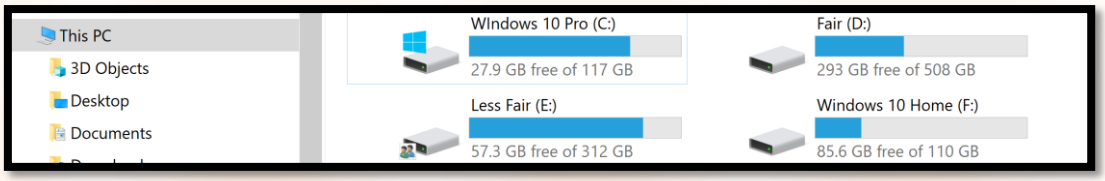
جایگاه سخت‌افزاری حافظهٔ دائم، هارددیسک[۱۵] است. در برنامه‌نویسی، رم، بیش از هارددیسک مورد توجه است، با این حال، هارددیسک، بخش مهمی از رایانه و حافظهٔ دائم، موضوع مهمی در برنامه‌نویسی است. هارددیسک، جایی است که برنامه‌ها و فایل‌ها، در آن نگهداری می‌شوند.

هارددیسک، از چند دیسک انعطاف‌ناپذیر (سخت) تشکیل شده و ماده‌ای آن را پوشانده است که این ماده، می‌تواند اطلاعات را ذخیره کند. ذخیره و بازیابی اطلاعات، به وسیلهٔ هد[۱۶] انجام می‌شود که در فاصلهٔ بسیار نزدیکی به دیسک قرار دارد. هد، چیزی شبیه به سوزن گرامافون است. به هر حال، برای برنامه‌نویسی، به طور طبیعی، نیازی نداریم که جزئیات سخت‌افزاری هارددیسک را بدانیم.

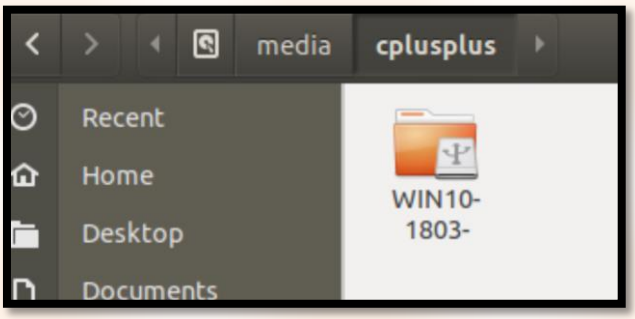
هارددیسک، اغلب، به چند پارتیشن[۱۷] تقسیم می‌شود. هر پارتیشن، مثل یک دیسک جداگانه عمل می‌کند. اگر برای تعریف دقیق کلمات، وسواس به خرج ندهیم، به پارتیشن، درایو[۱۸] یا والیوم[۱۹] نیز گفته می‌شود. هرچند تعریف رسمی درایو، سخت‌افزاری برای ذخیرهٔ اطلاعات است، و در واقع، درایو، شامل کل هارددیسک می‌شود، و نیز، تعریف رسمی والیوم، یک پارتیشن فرمت‌شده[۲۰] است. نرم‌افزارها و حتی ویندوز، به این تعاریف رسمی، بی‌توجه هستند و این کلمات را به جای هم به کار می‌برند. در ویندوز، با فشاردادن کلید پنجره و E (به صورت هم‌زمان)، و انتخاب This PC، پارتیشن‌های هارددیسک نمایش داده می‌شوند:

---

[۱۵] hard disk
[۱۶] head
[۱۷] partition
[۱۸] drive
[۱۹] volume
[۲۰] formatted



در ویندوز، به هر کدام از درایوها، یک حرف از حروف زبان انگلیسی نسبت داده شده است. مثلاً، برای اشاره به یک درایو، می‌گوییم «درایو C» یا «درایو H». در تصویر بالا، حرف مربوط به هر درایو را در کنار آن می‌بینید. ویندوز، در درایو C نصب شده است و به همین سبب، در تصویر فوق، آیکانش کمی متفاوت است. همچنین، با اتصال فلش‌مموری یا هارد اکسترنال به ویندوز، به هر یک از پارتیشن‌های جدید آن، یک حرف اختصاص می‌یابد. این حروف، کمک می‌کنند تا آدرس هر فایل از بین پارتیشن‌ها و فولدرها معلوم شود. برای نمونه، اگر فایلی با نام text.txt، مستقیماً در درایو C قرار داشته باشد، آدرسش، C:\text.txt خواهد بود. در اوبونتو، پارتیشن‌ها، با حروف، مشخص نشده‌اند و نام‌های آزادتری دارند. برای مثال، با اتصال یک فلش‌مموری به اوبونتو، اسمی که برای فلش‌مموری انتخاب شده است، به عنوان اسم درایو مربوط به آن به کار می‌رود:



با دوبار–کلیک[۲۱] بر هر یک از این درایوها، آن درایو، همچون یک فولدر باز می‌شود و فایل‌ها و فولدرهایی که در آن قرار دارند، نمایان می‌گردند.

گفتیم که کوچک‌ترین جزء حافظه، چه از نوع موقت و چه از نوع دائم، بیت است و هیچ اطلاعاتی نمی‌تواند کم‌تر از یک بیت باشد. یک بیت دو وضعیت دارد: وضعیت خاموش و وضعیت روشن. وضعیت خاموش با ۰ نشان داده می‌شود و وضعیت روشن با ۱. از کنارهم‌گذاشتن ۸ بیت، یک بایت به وجود می‌آید؛ پس، یک بایت، می‌تواند به این صورت باشد:

۱	۰	۱	۱	۰	۰	۱	۱

هنگامی که می‌خواهیم ترتیب صفر و یک‌ها را در یک بایت به‌خصوص، بیان کنیم، آن‌ها را به صورت یک عدد در مبنای ۲ ارائه می‌دهیم[۲۲]. به عنوان مثال، برای بایت بالا، ترتیب صفر و یک‌ها، با عدد ۱۰۱۱۰۰۱۱ در مبنای ۲ بیان می‌شود. از آنجا که نمایش اعداد در مبنای ۲، طولانی است، به طور معمول، آن‌ها را در مبناهای دیگری مانند ۸، ۱۰ یا ۱۶ نشان می‌دهیم. ماشین‌حساب‌هایی که به صورت پیش‌فرض، در ویندوز یا اوبونتو نصب شده‌اند، امکان تغییر مبنای اعداد را دارند؛ همین‌طور برخی ماشین‌حساب‌های آنلاین در اینترنت.

---

[۲۱] double-click

[۲۲] عدد درمبنای ۲ یعنی عددی که تنها از دو نماد ۰ و ۱ درست شده است.

مبنای ۱۰، مبنایی است که از آن، در کارهای روزمره، برای نوشتن اعداد استفاده می‌کنیم. از این رو، این مبنا، برای ما معنی‌دارتر از سایر مبناهاست. عدد ۱۰۱۱۰۰۱۱، در مبنای ۱۰، همان عدد ۱۷۹ است. بنابراین، می‌توانیم بگوییم مقدار عددی بایت:

۱	۰	۱	۱	۰	۰	۱	۱

۱۷۹ است. بایت‌ها، می‌توانند هر نوع اطلاعاتی؛ برای نمونه، یک جملهٔ فارسی یا یک تصویر؛ را ذخیره کنند، اما در ابتدا، باید چنین اطلاعاتی به شکل اعداد (در مبنای ۲) در بیایند، تا در بیت‌ها قابل‌ذخیره‌کردن باشند. به این ترتیب، می‌توان گفت، اطلاعات موجود در رم و هارددیسک، چیزی جز دنباله‌هایی طولانی از صفرها و یک‌ها نیستند.

متغیرها

در برنامه‌نویسی، برای آسان‌تر شدن کار، حافظه (رم) را به بسته‌های کوچکی تقسیم می‌کنیم. هر بسته، از کنارهم‌گذاشتن چند بایت به وجود می‌آید. به هریک از این بسته‌ها، یک «**متغیر**[۲۳]» می‌گوییم. متغیرها، با نام‌هایی مانند x، y و... نشان داده می‌شوند. هر متغیر، نمایندهٔ یک یا چند بایت از حافظه است. اگر ch، یک متغیر باشد و نمایندهٔ تنها یک بایت خاص باشد، می‌توانیم بنویسیم:

```
ch = 179;
```

و با این کار، بایت مربوط به متغیر، به صورت زیر در می‌آید:

۱	۰	۱	۱	۰	۰	۱	۱

وقتی با ch کار می‌کنیم، برای ما مهم نیست که بایتی که ch به آن اشاره دارد، در کجای رم مستقر شده است. کار متغیرها همین است؛ با متغیرها به طور غیرمستقیم از رم استفاده می‌کنیم و با وجود آن‌ها، در برنامه‌نویسی؛ لازم نیست اطلاعات زیادی در مورد رم داشته باشیم. به همین دلیل، خیلی از زبان‌های برنامه‌نویسی، امکان دستکاری مستقیم رم را به برنامه‌نویس نمی‌دهند. اما C++، تا حدودی، این امکان را می‌دهد. از آنجا که همهٔ برنامه‌های درحال‌اجرا، از رم استفاده می‌کنند؛ دستکاری مستقیم رم، می‌تواند خطرناک باشد. در آینده، منظور ما از دستکاری رم مشخص خواهد شد.

فایل‌ها

فایل‌ها، بخش‌هایی از اطلاعات هستند که در یک درایو قرار دارند؛ یعنی اطلاعات موجود در هارددیسک، در بسته‌های جداگانه‌ای به نام **فایل**[۲۴] ذخیره می‌شوند. فایل‌ها را می‌توان به دو دستهٔ مهم تقسیم کرد:

۱) فایل اجرایی.

---

[۲۳] variable

[۲۴] file

۲) فایل غیراجرایی.

فایل‌های اجرایی همان برنامه‌ها هستند، و می‌توان گفت، بدون آن‌ها، رایانه غیرقابل‌استفاده است. فایل‌های غیراجرایی، اغلب، اطلاعاتی هستند که به وسیلهٔ فایل‌های اجرایی مورد استفاده قرار می‌گیرند.

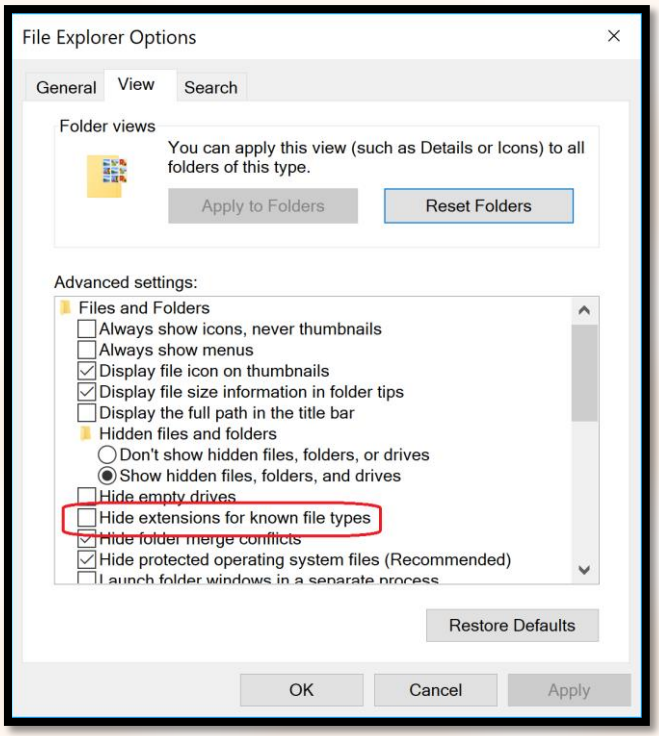
- هر فایل، یک مسیر[۲۵] دارد. مسیر، آدرسی است شامل نام درایو و فولدرها که توسط آن، می‌توان به فایل دسترسی پیدا کرد. برای مثال؛ (بر اساس ویندوز) فرض کنید در درایو C، فولدری به نام hello موجود است و در این فولدر، فایلی با نام bye.txt وجود دارد. مسیر فایل bye.txt، عبارت است از "C:\hello\byte.txt" در اینجا، C:\، یعنی درایو C و \hello، یعنی فولدر hello. تفاوتی بین مسیر نسبی[۲۶] و مسیر کامل[۲۷] هست که در اینجا به آن نمی‌پردازیم و مسیر را همان مسیر کامل تعریف می‌کنیم. گاه، ممکن است به جای واژهٔ «مسیر»، از «آدرس»، یا «اسم و آدرس» استفاده شود. همچنین، به مسیر فولدری که شامل فایل است، مکان/محل فایل[۲۸] گفته می‌شود.
- فایل‌ها، فولدرها، و درایوها، در هر سیستم‌عامل، با یک «برنامهٔ مدیریت فایل» قابل‌مشاهده‌اند. در ویندوز، برنامهٔ مدیریت فایل پیش‌فرض، Windows Explorer نام دارد. در سیستم‌عامل‌های لینوکسی، این برنامه‌ها، گوناگون هستند. مثلاً، در اوبونتو، به صورت پیش‌فرض، برنامهٔ nautilus، برای مدیریت فایل نصب شده است. در یک برنامهٔ مدیریت فایل، می‌توان فایل‌ها را پیدا کرد، حجم (اندازه) آن‌ها را دید یا آن‌ها را با برنامه‌های مناسب باز کرد و اگر فایلی، اجرایی باشند، آن را اجرا کرد. در ویندوز، با فشار کلیدهای پنجره + E، برنامهٔ آشنای مدیریت فایل باز می‌شود.
- تقریباً هر فایلی، یک پسوند[۲۹] دارد. برای نمونه، پسوند فایل bye.txt، txt. است. پسوند، به‌خصوص در ویندوز، معلوم می‌کند که فایل، با چه نرم‌افزاری قابل‌بازشدن است. در ویندوز، ممکن است پسوندهای فایل‌ها، مخفی شوند و در برنامهٔ مدیریت فایل ویندوز، قابل‌مشاهده نباشند. برای این که پسوندها، در ویندوز ۱۰، نشان داده شوند، در File Explorer Options، که می‌توانید آن را با سرچ ویندوز پیدا کنید، مسیر زیر را دنبال کنید:

View → Hide extensions for known file types

و علامت تیک مقابل آن را بردارید:

---

[۲۵] path
[۲۶] relative path
[۲۷] full path
[۲۸] file location
[۲۹] extension



برنامه‌نویسی

برنامه چیست؟

برنامه[۳۰]، نوشته‌ای است که به وسیلهٔ آن به رایانه می‌گوییم که چگونه یک کار خاص را انجام دهد. رایانه، قادر به درک زبان فارسی یا انگلیسی یا حتی اسپرانتو نیست؛ زیرا این زبان‌ها، قواعد پیچیده‌ای دارند و بسیاری از جملات، در زبان‌های انسانی، مبهمند. برای این که بتوانیم به رایانه دستوراتی بدهیم که بتواند آن‌ها را انجام دهد، باید آن دستورات را، به طور دقیق، مطابق با قواعد یکی از زبان‌های برنامه‌نویسی بنویسیم و به رایانه بدهیم تا قادر به اجرای آن باشد. یکی از زبان‌های برنامه‌نویسی، C++ است.

همان‌طور که یک آموزگار، در مدرسه، برای یاددادن ضرب یا تقسیم اعداد چندرقمی، مراحل کار را جزء به جزء، برای دانش‌آموزانی که هیچ پیش‌زمینه‌ای جز جدول ضرب ندارند، توضیح می‌دهد، یک برنامهٔ رایانه‌ای نیز، مراحل انجام یک کار را، به طور دقیق، برای رایانه، توضیح می‌دهد. با یک برنامه، می‌توان شیوهٔ ضرب دو عدد را، که آموزگاران به دانش‌آموزان، آموزش می‌دهند، به رایانه نیز آموزش داد؛ هرچند طراحان زبان برنامه‌نویسی C++، این روش را، برای اعداد نه‌چندان بزرگ، به صورت پیش‌فرض، در آن قرار داده‌اند؛ تا برنامه‌نویس، کم‌تر درگیر مسائل ابتدایی شود.

[۳۰] program

برنامه، یک نوشته است و از تعدادی خط، شامل کلمات و نمادهای عادی تشکیل شده است. وقتی برنامه، برای اجرا، به رایانه داده می‌شود؛ رایانه، برنامه را از ابتدا تا انتها، خط به خط، اجرا می‌کند. برای طراحی و نوشتن یک برنامه، می‌توانیم، در ابتدا، آن را به فارسی بنویسیم، و سپس، سعی کنیم که آن را، به یک زبان برنامه‌نویسی، مانند C++، ترجمه کنیم. برای نمونه، این یک برنامه است:

۱) عدد ۲ را در خودش ضرب کن و حاصل را در متغیر x قرار بده.
۲) x را در ۳ ضرب کن و حاصل را در متغیر y قرار بده.
۳) مقدار y را نمایش بده.

ترجمهٔ این دستورات به زبان برنامه‌نویسی C++، به این صورت است:

```
x = 2 * 2;
y = 3 * x;
cout << y;
```

البته، این برنامه، تنها سه خط دارد. یک برنامه، ممکن است هزاران خط داشته باشد و نیز، شامل حلقه باشد. **حلقه**، دستوری است که سبب می‌شود، خط یا خط‌هایی خاص از برنامه، بارها **اجرا شوند**[۳۱].

کامپایل برنامه

برای این که بتوانیم برنامه‌ای را به رایانه بدهیم، باید از یک کامپایلر استفاده کنیم. کامپایلر، نرم‌افزاری است که برنامه از زبان برنامه‌نویسی (مثلاً C++)، به زبان ماشین[۳۲] تبدیل می‌کند. زبان ماشین، زبانی است که پردازندهٔ مرکزی رایانه[۳۳]، می‌تواند آن را بفهمد و اجرا کند. سطح زبان ماشین، حتی از زبان اسمبلی[۳۴] نیز پایین‌تر است و ممکن است در رایانه‌های مختلف، متفاوت باشد. زبان C++، به گونه‌ای است که برای برنامه‌نویس، قابل فهم است؛ اما، پردازندهٔ رایانه، آن را نمی‌فهمد، و در نتیجه، لازم است به زبان قابل‌فهمی برای پردازنده تبدیل شود که این کار، توسط کامپایلر انجام می‌گردد.

توضیحات بالا، ساده‌شده و کلی هستند و شامل جزئیات نمی‌شوند. در عمل، برای برنامه‌نوشتن به زبان C++، نیازی نداریم که جزئیات عملکرد کامپایلر و اجرای برنامه را بدانیم؛ آنچه لازم است بدانیم، این است که برنامه‌ای که به زبان C++ نوشته‌ایم، باید به یک فایل اجرایی تبدیل شود که توسط سیستم‌عامل، قابل‌اجراست. کامپایل‌شدن برنامه، و به طور دقیق‌تر، تبدیل آن، از متنی که به زبان C++ نوشته شده است، به یک فایل اجرایی، چندین مرحله دارد که توسط قسمت‌های مختلف کامپایلر، انجام می‌شوند:

۱) **پیش‌پردازش**[۳۵]: در این مرحله، عملیات اولیهٔ مربوط به تنظیم متن انجام می‌پذیرد؛ عملیاتی مانند حذف توضیحات و یادداشت‌هایی که برنامه‌نویس در میان دستورات اصلی برنامه، برای مراجعهٔ بعدی خود،

---

[۳۱] در رایانه، به جای «خواندن و انجام دستور» می‌گوییم «اجرای دستور».

[۳۲] machine language
[۳۳] CPU
[۳۴] assembly language
[۳۵] preproccessing

وارد کرده است. در متن برنامه، می‌توان دستوراتی مخصوص پیش‌پردازنده نوشت. برای نمونه، دستوراتی در مورد جایگزین‌کردن یک کلمه با کلمهٔ دیگر. پیش‌پردازش، توسط **پیش‌پردازشگر**[۳۶] انجام می‌شود.

۲) **کامپایل**[۳۷]: این مرحله، یک مرحله از فرایند کلی کامپایل‌شدن است، با این حال، آن را کامپایل می‌نامند.[۳۸] در این مرحله، «**کامپایلر**»، متن آماده‌شده توسط پیش‌پردازشگر را، به زبان اسمبلی ترجمه می‌کند. «کامپایلر» نیز، مانند «کامپایل»، می‌تواند یک نام مشترک باشد و می‌تواند به نرم‌افزاری که تمام این مراحل را انجام می‌دهد، یا قسمتی از آن که تنها مرحلهٔ دوم را انجام می‌دهد، اطلاق گردد.

۳) **اسمبلی**[۳۹]: در این مرحله، **اسمبلر**[۴۰]، متن برنامه را از اسمبلی به آبجکت[۴۱] تبدیل می‌کند. متن آبجکت، در واقع، همان زبان ماشین است، که البته، هنوز کاملاً آمادهٔ اجرا نیست. زبان ماشین، چنان‌که پیش‌تر نیز اشاره شد، زبانی است که پردازندهٔ مرکزی رایانه، می‌تواند آن را بفهمد.

۴) **لینک**: در این مرحله، **لینکر**[۴۲]، متن یا متن‌های آبجکت را، به فایل اجرایی نهایی تبدیل می‌کند. به طور کلی، کار لینکر مرتب و کامل‌کردن متن‌های آبجکت است.

برای این که نام مرحلهٔ دوم، با کل مراحل اشتباه نشود، به جای کامپایل (یا کامپایلر)، وقتی به تمام مراحل اشاره دارد، **بیلد**[۴۳] (یا **بیلدر**[۴۴]) نیز می‌گویند. وقتی یک برنامهٔ C++، بیلد، و سپس، فایل اجرایی تولیدشده، اجرا می‌گردد، برای سادگی، می‌گوییم «برنامه اجرا شده است»؛ در حالی که منظور این است که برنامه بیلد شده و فایل اجرایی حاصل، اجرا شده است.

خوشبختانه، اگر برنامه‌هایمان را در یک **محیط توسعهٔ یک‌پارچه**[۴۵] بنویسیم و کامپایل کنیم، تقریباً، هیچ‌گاه نیازی به آشنایی با تمام مراحل فوق یا انجام تک‌تک آن‌ها به صورت جداگانه نخواهیم داشت. محیط توسعهٔ یک‌پارچه، یا مخفف آن **آی‌دی‌ای**[۴۶]، یک نرم‌افزار برنامه‌نویسی است که تمام امکانات را برای نوشتن یک برنامه و بیلد آن فراهم، و برنامه‌نویسی را، آسان می‌گرداند. برای ادامهٔ این کتاب، بهتر از یکی از دو آی‌دی‌ای رایگان زیر را در سیستم‌عامل خود نصب کنید:

1) Microsoft Visual Studio Community ۲۰۱۷ : Visual C++
2) Qt Creator

هر دو، قابل‌نصب بر ویندوز ۱۰ هستند. دومی، نسخهٔ اوبونتو نیز دارد، و در این کتاب، نسخهٔ اوبونتوی آن، که به صورت پیش‌فرض، از کامپایلر gcc استفاده می‌کند، مد نظر است. هر دو را می‌توان به صورت قانونی و رایگان از طریق پایگاه‌های اینترنتیشان نصب کرد (در اوبونتو، می‌توانید نسخهٔ موجود در مخازن پیش‌فرض آن را نصب

---

[۳۶] preprocessor
[۳۷] compilation
[۳۸] مانند وقتی که مرکز یک استان هم‌نام خود استان است.
[۳۹] Assembly
[۴۰] Assembler
[۴۱] Object Code
[۴۲] linker
[۴۳] build
[۴۴] builder
[۴۵] Integrated Development Environment
[۴۶] IDE

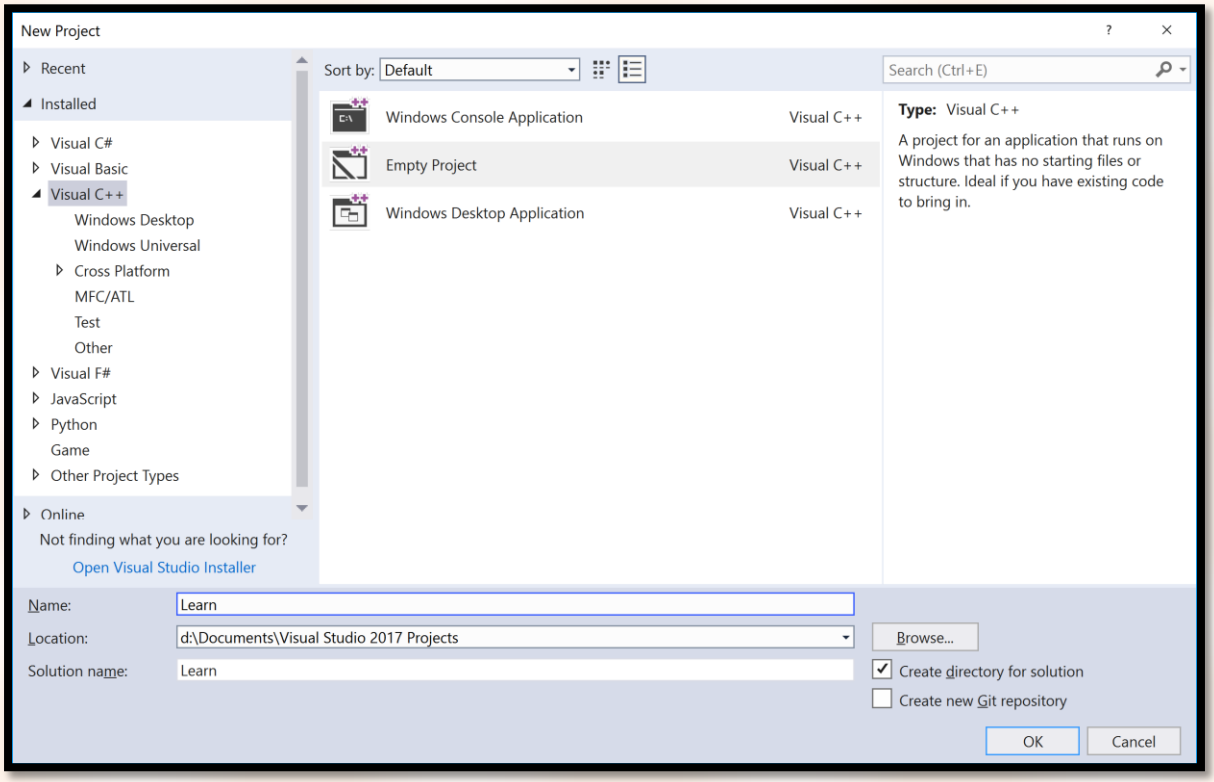
کنید). در این نوشته، هرگاه قصد اشاره به آی‌دی‌ای اول را داشته باشیم، می‌گوییم «ویژوال‌استودیو» و برای اشاره به آی‌دی‌ای دوم، می‌گوییم «کیوت‌کریییتور».

بی‌شک، هر آی‌دی‌ای دیگری نیز برای نوشتن و بیلد برنامه‌ها مناسب است، و حتی برخی کامپایلرهای آنلاین، مانند cpp.sh یا godbolt.org برای اجرای برنامه‌ها کافیند؛ اما توضیحاتی در مورد کار با آن‌ها، در این کتاب موجود نخواهد بود و برنامه‌های این کتاب، لزوماً در آن‌ها آزموده نشده‌اند.

ایجاد پروژه در ویژوال‌استودیو

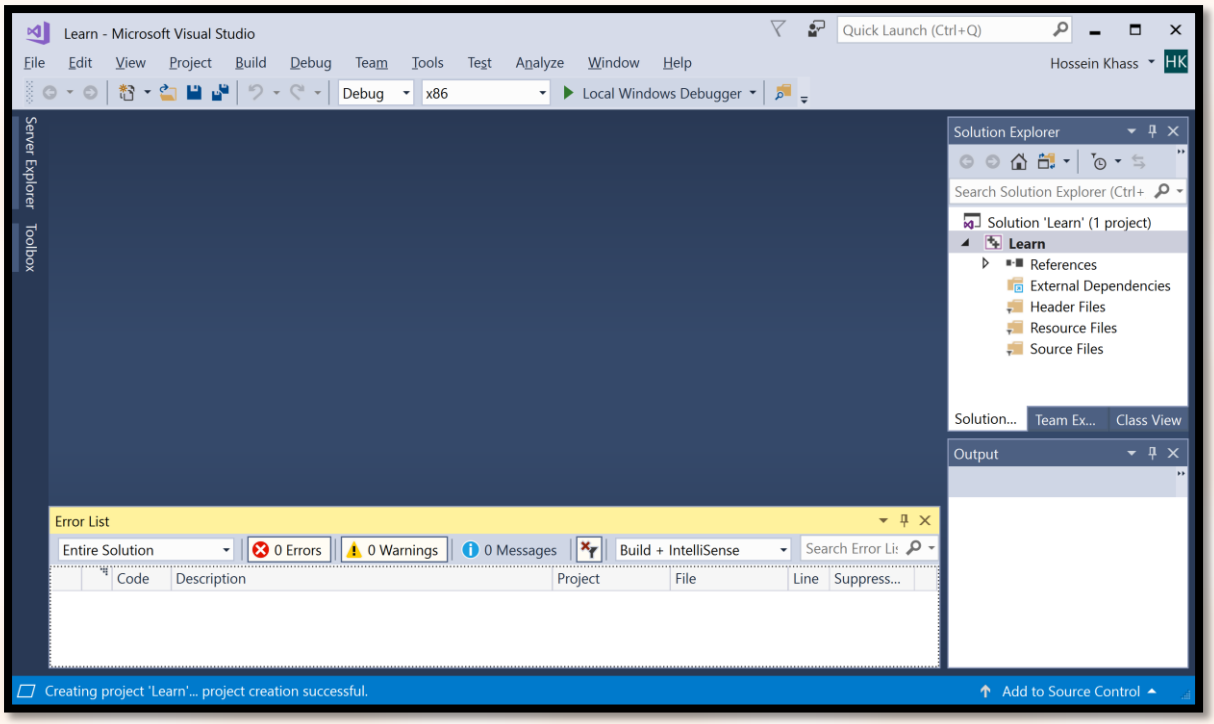
برای نوشتن و اجرای یک برنامهٔ ابتدایی در ویژوال‌استودیو؛ به روشی که در ادامهٔ این بخش توضیح داده می‌شود، یک پروژهٔ خالی (کنسول) بسازید: برنامهٔ ویژوال‌استودیو را باز کنید. ابتدا مسیر زیر را از طریق منوها طی کنید:
File → New → Project... → Visual C++ → Empty Project

پس از انتخاب Empty Project، در قسمت Name بنویسید «Learn»[۴۷]:

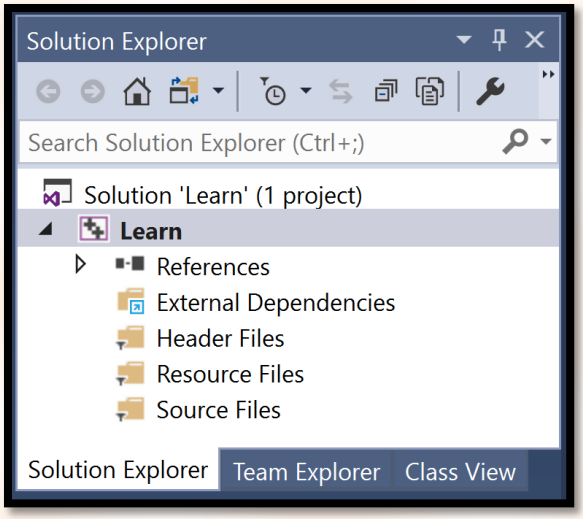


و سپس، OK را فشار بدهید تا پروژه ایجاد شود:

[۴۷] هر نام دیگری هم می‌توان نوشت؛ این نام برای هماهنگی توضیحات کتاب با آنچه خواننده انجام می‌دهد، پیشنهاد شده است.



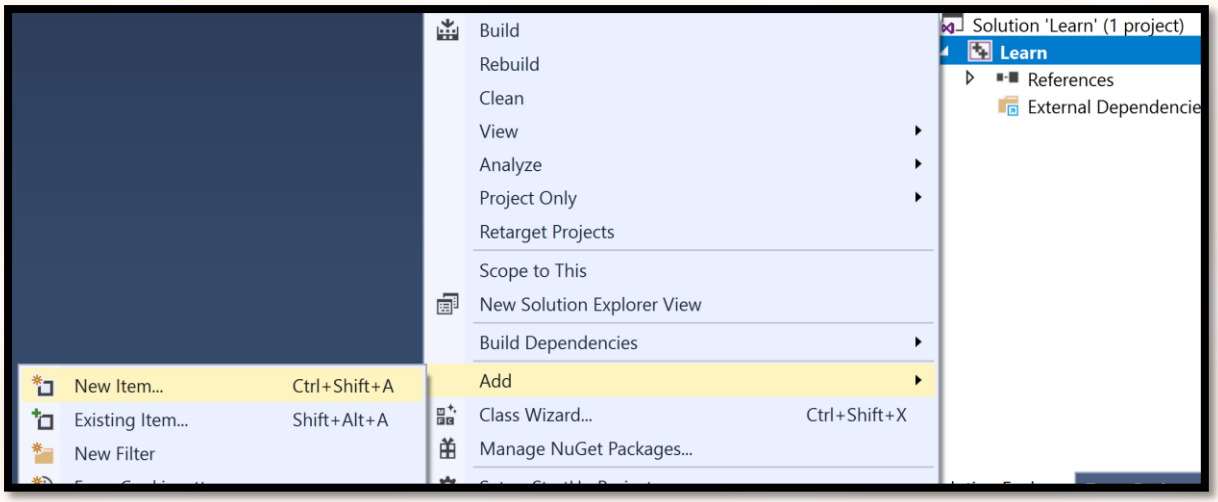
که ممکن است ظاهر این پنجره، برای شما کمی متفاوت باشد؛ زیرا زیرپنجره‌های موجود را می‌توان جابه‌جا کرد. اما زیرپنجرهٔ مهم در این مرحله، Solution Explorer است:



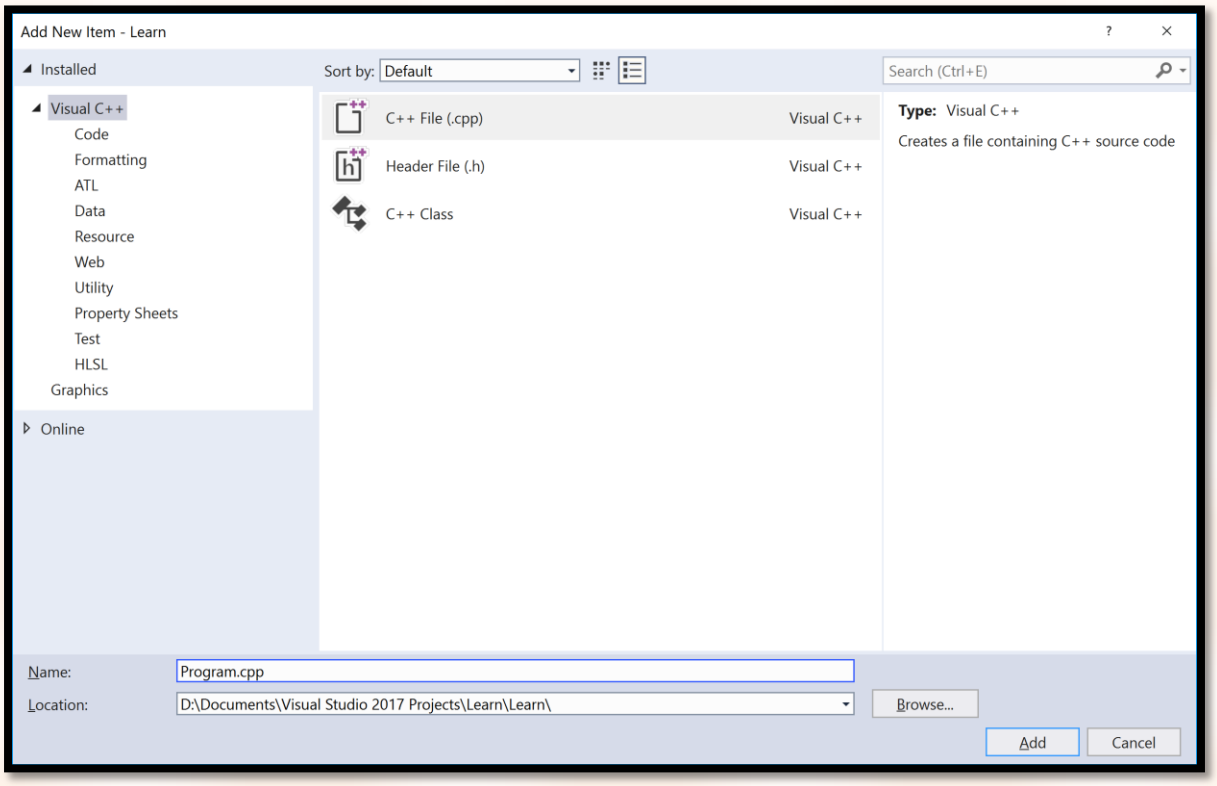
در زیرپنجرهٔ Solution Explorer، سه فیلتر به نام‌های Header Files، Resource Files و Source Files می‌بینید. با انتخاب هر یک از این فیلترها، و فشاردادن کلید Delete از صفحه‌کلید، فیلترها را حذف کنید. بر Learn راست‌کلیک کنید و مسیر زیر را طی کنید:

Add → New Item... → Visual C++ → Code → C++ File(.cpp)

تصاویر مربوط به طی این مسیر را در زیر مشاهده می‌کنید:

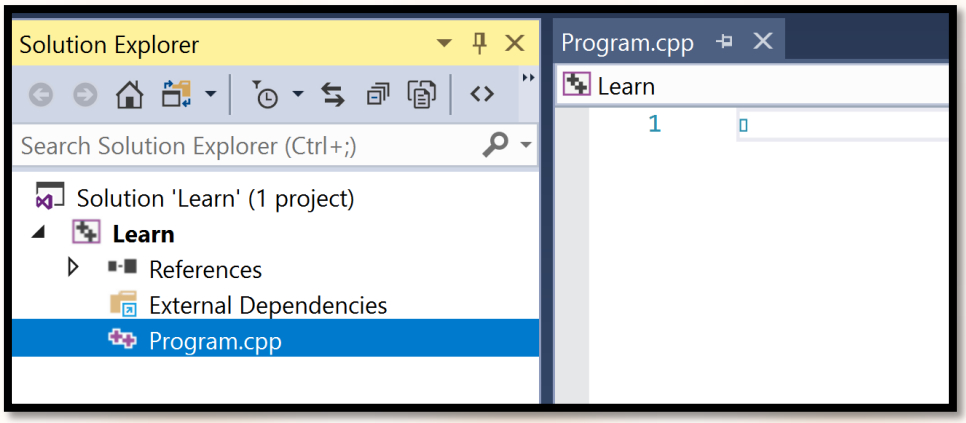


پس از انتخاب C++ File(.cpp)، در نوار Name بنویسید «Program»[۴۸]:



و Add را فشار دهید. حال، زیرپنجرهٔ Solution Explorer باید به این شکل باشد:

[۴۸] هر اسم دیگری هم می‌توان نوشت.

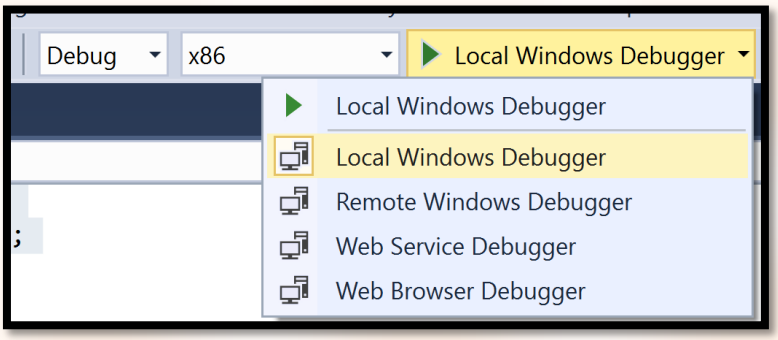


که در آن، فایل Program.cpp اضافه شده است و نیز در ویژوال‌استودیو باز شده و آمادهٔ نوشتن متن برنامه است. اگر این فایل، هنوز در ویژوال‌استودیو باز نشده است، و یا بسته شده است، می‌توانید آن را با دوبار-کلیک بر Program.cpp در زیرپنجرهٔ Solution Explorer، باز کنید. حال، همه چیز برای برنامه‌نوشتن آماده است. در Program.cpp، متن زیر را قرار دهید:

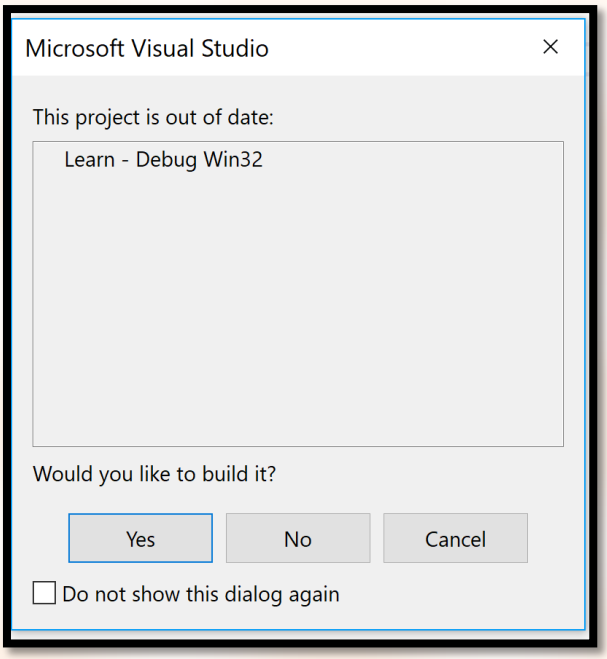
```
// Program ID: 0000
#include <iostream>
using namespace std;

int main()
{
    cout << "This is a test";
    cin.ignore(1);
}
```

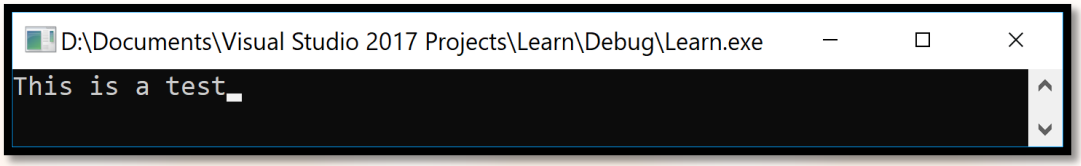
کلید F5 را در صفحه‌کلید فشار دهید تا برنامه اجرا شود؛ توجه کنید که قبل از اجرا، Local Windows Debugger در قسمت بالای ویژوال‌استودیو انتخاب شده باشد:



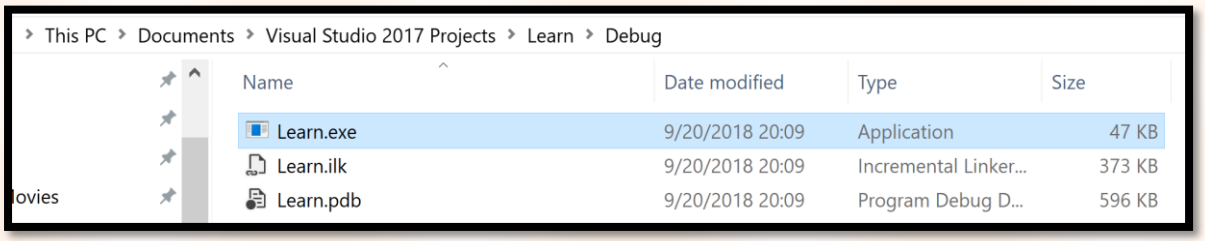
با کلیک بر همین دگمه نیز، می‌توان برنامه را اجرا کرد. ممکن است قبل از شروع اجرا، پنجره‌ای مانند این:



باز شود؛ در این صورت بعد از تیک‌زدن Do not show this dialog again دگمهٔ Yes را فشار دهید. برنامه اجرا می‌گردد و پنجرهٔ زیر باز می‌شود:

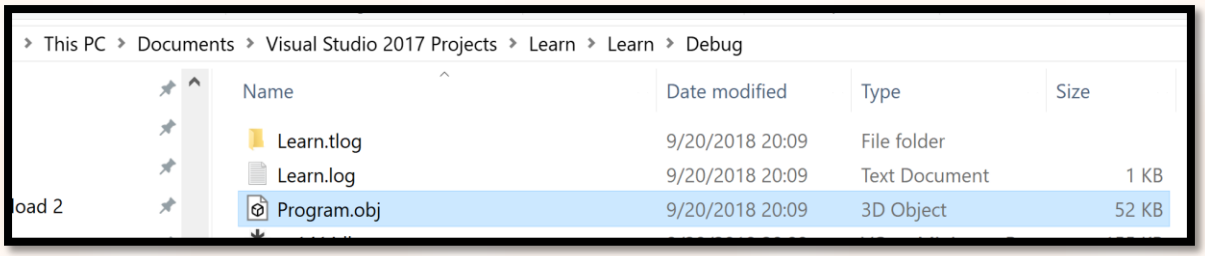


این همان برنامهٔ شماست که اجرا شده است. کار این برنامه، نوشتن یک جملهٔ ساده بود و چنان‌که پیداست این جمله نوشته شده است. حال، با فشاردادن کلید Enter، پنجرهٔ فوق را ببندید. اگر به مکانی در حافظهٔ دائم که مسیرش در پنجرهٔ بالا (و نیز، در پنجره‌ای که نام Program.cpp را وارد کردیم)، مشخص شده است[۴۹]؛ مراجعه کنید، فایل اجرایی برنامه را، که حاصل بیلدشدن برنامه است، می‌بینید:



فایل Learn.exe برای اجرا، دیگر به ویژوال‌استودیو نیاز ندارد. این فایل، نتیجهٔ نهایی نوشتن و بیلد یک برنامه است. البته این برنامه، بسیار ساده است. برنامه‌های پیچیده‌تر نیز، با همین روش ایجاد می‌شوند. چنان که پیش‌تر گفته شد، پس از ایجاد فایل اجرایی نهایی، باید متن آبجکت ساخته شود. فایل مربوط به متن آبجکت را نیز، می‌توان در فولدرهای اطراف فولدر حاوی فایل اجرایی نهایی، پیدا کرد:

[۴۹] این مسیر ممکن است برای شما متفاوت باشد، اما محل درج مسیر در تصاویر مشخص شده است.



فایل Program.obj حاوی متن آبجکت مربوط به Program.cpp است. این فایل، تقریباً، هیچ کاربردی برای ما ندارد؛ اما لینکر، برای ایجاد فایل اجرایی نهایی، به آن نیازمند بوده است.

پروژهٔ Learn را نگه دارید و برنامه‌هایی را که در آینده ارائه خواهند شد، در آن آزمایش کنید و یا تمرین‌هایی را که پیشنهاد خواهند شد، در آن بنویسید.

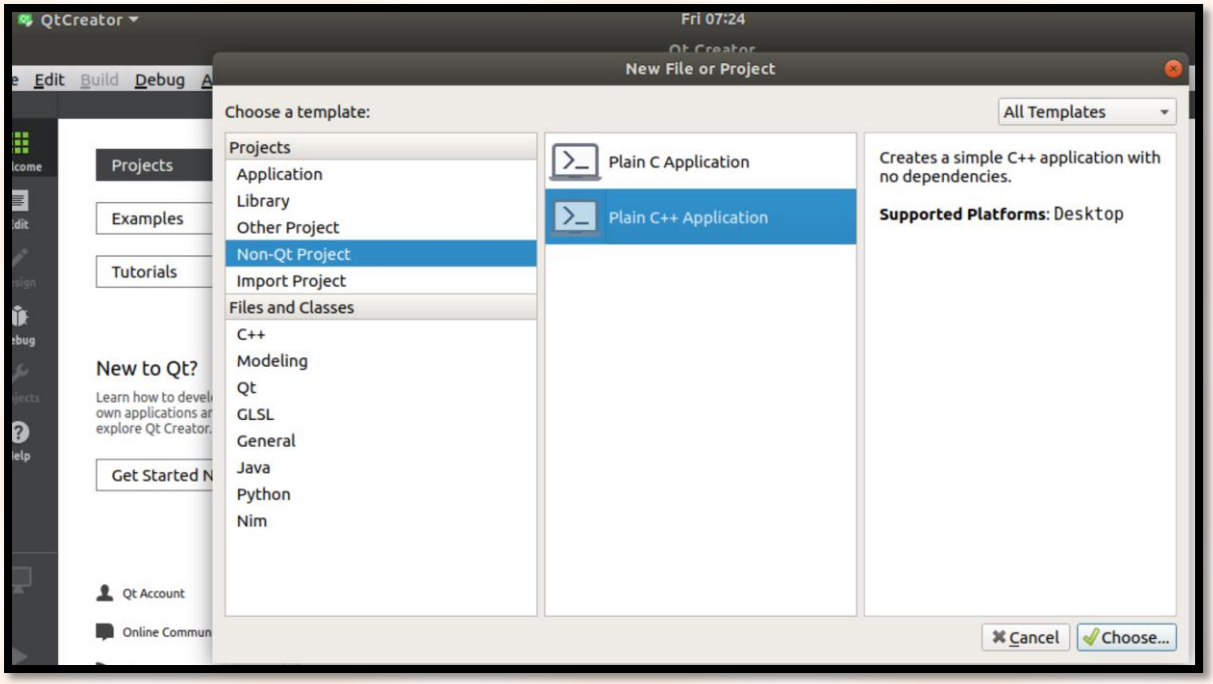
تا به اینجا، مقدمات برنامه‌نویسی آماده شده است و می‌توانیم به تدریج آموزش C++ را شروع کنیم.

ایجاد پروژه در کیوت‌کریییتور

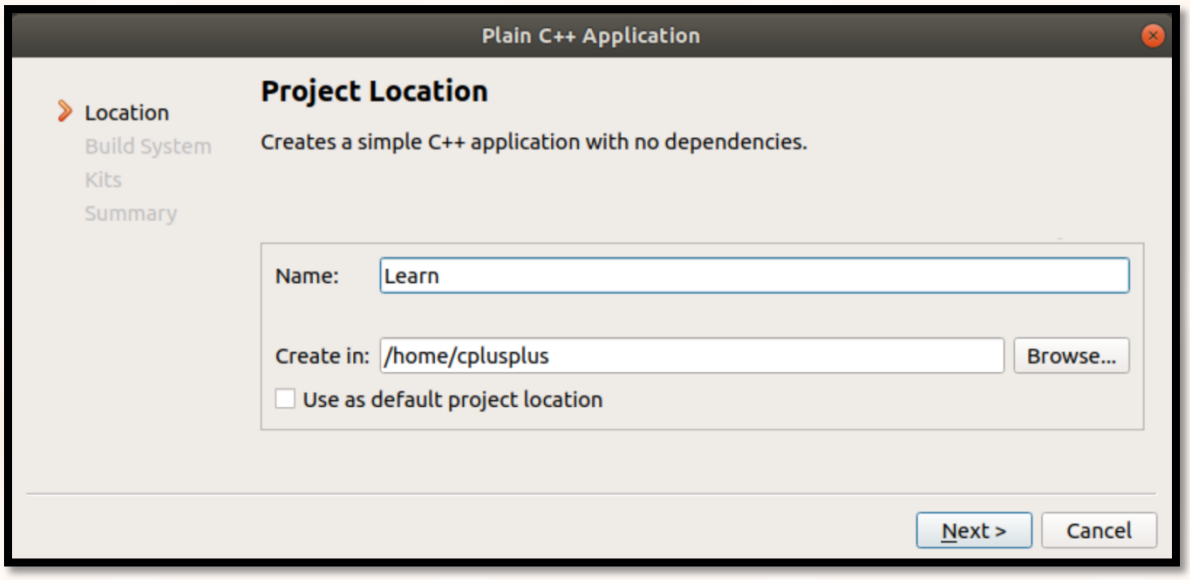
برای ایجاد پروژه در کیوت‌کریییتور، مسیر زیر را طی کنید:

File → New File or Project... → Projects → Non-Qt Project → Plain C++ Application

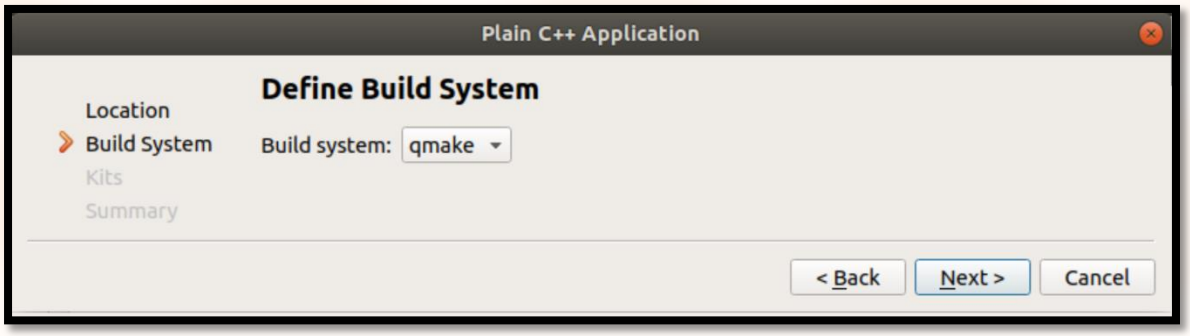
پس از انتخاب Plain C++ Application:



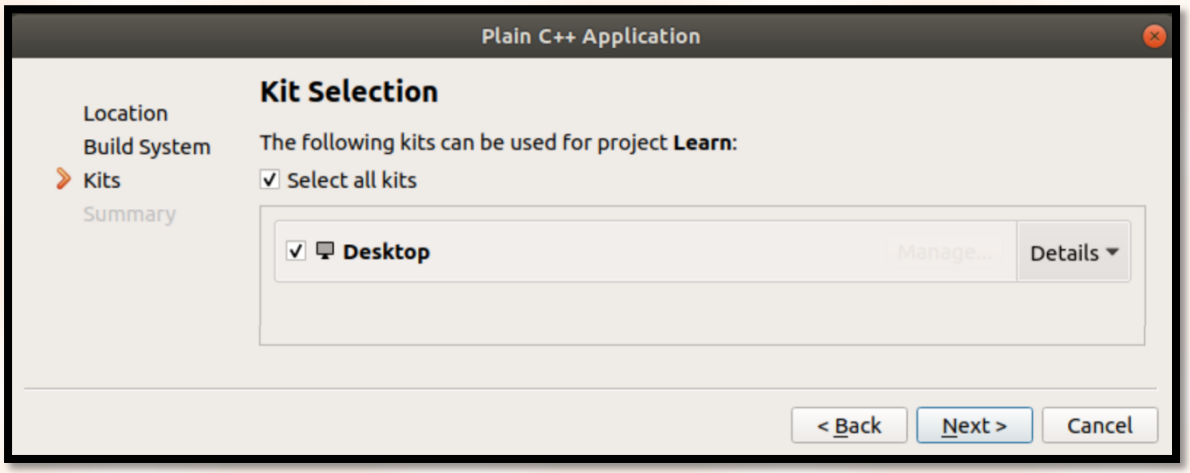
دگمهٔ Choose را فشار دهید. در پنجرهٔ بعد، در نوار Name، بنویسید «Learn»:



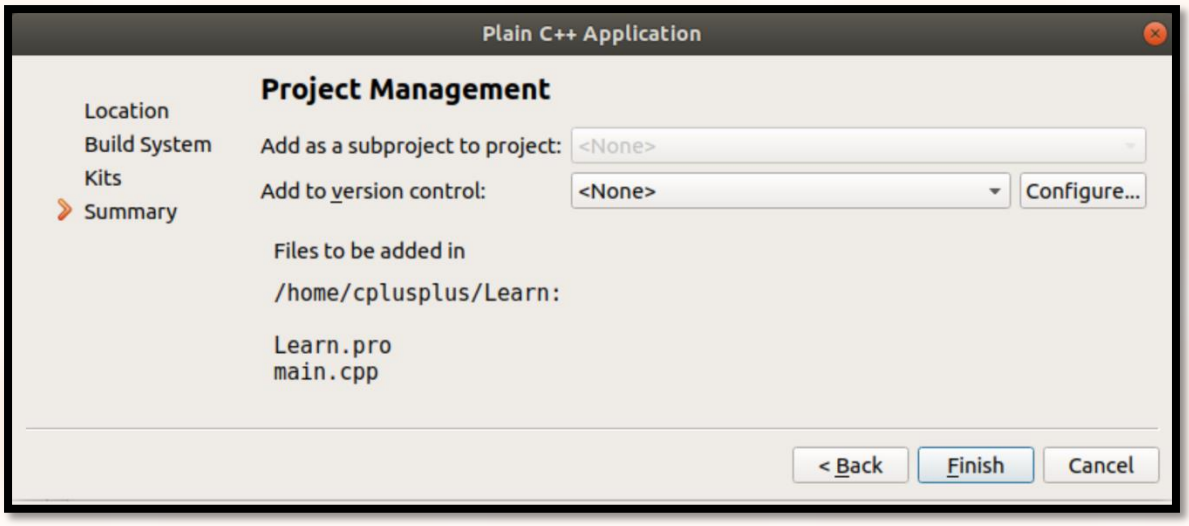
به مسیری که در نوار «Create in:» آمده است دقت کنید. این مسیر ممکن است برای شما متفاوت باشد. پروژه، در این مسیر ذخیره می‌شود. Next را فشار دهید و در پنجرهٔ بعدی نیز:



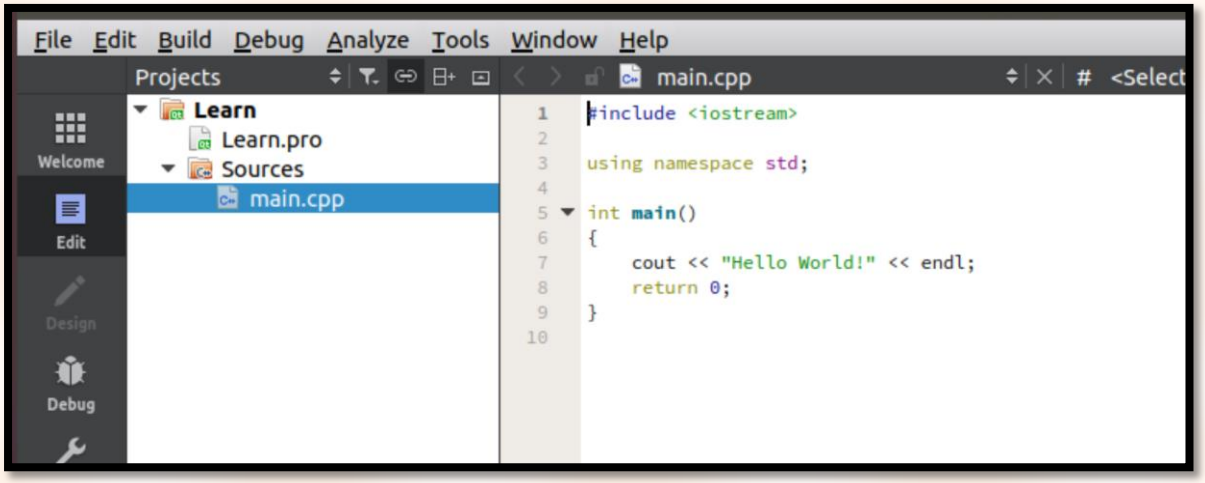
Next را فشار دهید. همین‌طور در پنجرهٔ بعد:



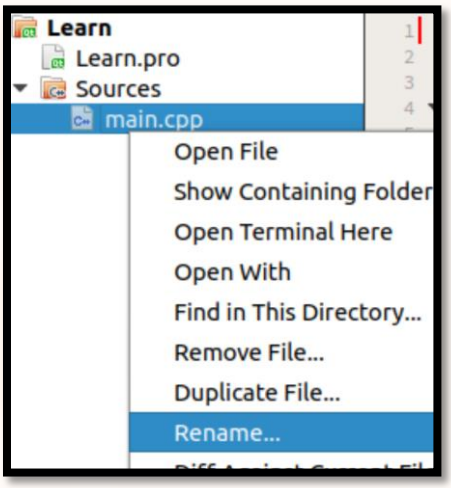
Next را فشار دهید. در نهایت، در:



Finish را فشار دهید تا پروژه ایجاد شود. توجه کنید که تنظیمات پنجره‌ها، در هر مرحله، مطابق تصاویر باشند. با ایجاد پروژه، فایل main.cpp آن، آمادهٔ نوشتن یک برنامه می‌گردد:



بهتر است برای هماهنگی با متن این کتاب، با راست‌کلیک بر main.cpp و انتخاب Rename، اسم فایل main.cpp را به Program.cpp تغییر دهید:

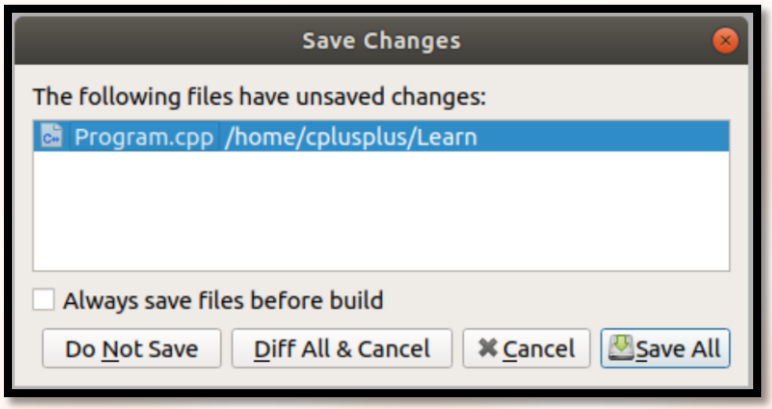


در این فایل، یک برنامهٔ ابتدایی، به صورت پیش‌فرض، قرار داده شده است. این برنامهٔ پیش‌فرض را به طور کامل پاک کنید و به جای آن، برنامهٔ مشابهی را که در بخش پیشین برای ویژوال‌استودیو آورده شده است، قرار دهید:

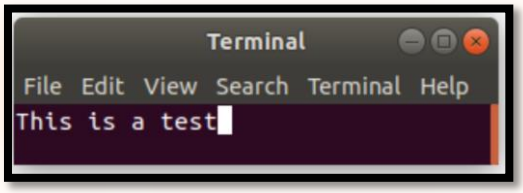
```
// Program ID: 0000
#include <iostream>
using namespace std;

int main()
{
    cout << "This is a test";
    cin.ignore(1);
}
```

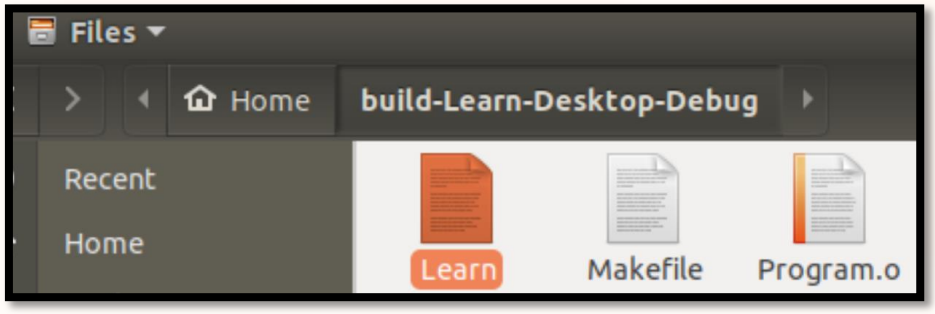
سپس، با فشاردادن کلیدهای Ctrl+R یا فشاردادن کلید F5، یا با کلیک بر مثلث سبزرنگ کنار کیوت‌کریتور، برنامه را اجرا کنید. پیش از اجرای برنامه، ممکن است پنجرهٔ زیر نشان داده شود:



تیک مربوط به Always save files before build را بزنید، و سپس، دگمهٔ Save All را فشار دهید؛ برنامه، اجرا می‌شود:



با دو بار فشاردادن کلید Enter از صفحه‌کلید (در حالی که پنجرهٔ برنامه (یعنی پنجرهٔ فوق) فعال است)، برنامه بسته خواهد شد. حال، در همان مسیری که پروژه ذخیره شده است، فولدری به نام build-Learn-Desktop-debug ایجاد شده است که حاوی فایل اجرایی برنامه است:



فایل بدون‌پسوند Learn، فایل اجرایی است؛ در لینوکس، فایل‌های اجرایی پسوند خاصی ندارند. Program.o نیز متن آبجکت به‌دست‌آمده از Program.cpp است، که البته، کارایی خاصی برای ما ندارد، و تنها، در تهیه فایل اجرایی مورد استفاده بوده است.

تا به اینجا، مقدمات برنامه‌نویسی آماده شده است و می‌توانیم، به‌تدریج، آموزش ++C را آغاز کنیم. اما قبل از شروع:

بر فایل Learn.pro در زیرپنجرهٔ Project دوبار–کلیک کنید تا باز شود. سپس، در متن آن، c++14 را به CONFIGها اضافه کنید و c++11 را، در صورت وجود، حذف کنید. برای مثال اگر متن اولیه به این صورت است:

```
TEMPLATE = app
CONFIG += console c++11
CONFIG -= app_bundle
CONFIG -= qt

SOURCES += \
    Program.cpp
```

آن را به این صورت در آورید:

```
TEMPLATE = app
CONFIG += console
CONFIG -= app_bundle
CONFIG -= qt
CONFIG += c++14

SOURCES += \
    Program.cpp
```

متن را، با فشاردادن Ctrl+S، ذخیره کنید و فایل Learn.pro را ببندید. این تغییر، سبب می‌شود که نسخهٔ C++14 از ++C قابل‌استفاده باشد. اگر در زمان دسترسی به این کتاب، نسخهٔ C++17، قابل‌فعال‌کردن باشد، بهتر است آن را فعال کنید؛ زیرا برخی برنامه‌های فصول پایانی کتاب، بر اساس C++17 نوشته شده‌اند. برای این کار، در حال حاضر، پس از نصب یک نسخهٔ به‌روز از کامپایلر gcc، باید به جای `CONFIG += c++14` کد `QMAKE_CXXFLAGS += -std=c++1z` را قرار داد. اما چون این کار دردسر دارد، می‌توانید از آن صرف‌نظر کنید.

رنگ کدها و پس‌زمینه

به متن برنامه یا بخشی از آن، **کد**[۵۰] می‌گویند. آی‌دی‌ای‌ها، اغلب، برخی کلمات کلیدی کدها را تشخیص، و رنگ آن‌ها را تغییر می‌دهند. رنگ متن و پس‌زمینهٔ متن برنامه، تنها برای کمک به برنامه‌نویس است و تأثیری در عملکرد برنامه ندارد. در بیشتر آی‌دی‌ای‌ها، امکان تغییر رنگ‌آمیزی، در قسمت تنظیمات، وجود دارد. برخی افراد، ممکن است پس‌زمینهٔ تیره با کدهای روشن را ترجیح دهند. با توجه به این که معمولاً، برنامه‌نویسی با تمرکز

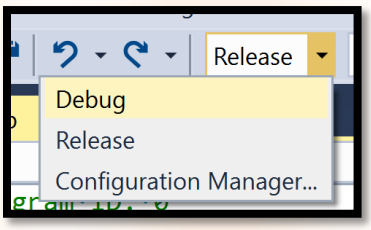
[۵۰] code

طولانی چشم بر متن همراه است، باید توصیه‌های پزشکی در مورد چشم برای انتخاب رنگ پس‌زمینه مورد توجه قرار گیرد.

روش بیلد

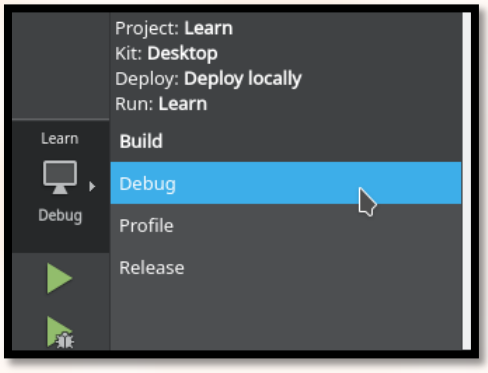
برنامه‌ها را می‌توان بر اساس تنظیمات مختلفی که به کامپایلر داده می‌شوند، بیلد کرد. در آی‌دی‌ای‌ها، معمولاً، دو دسته از تنظیمات به نام‌های Debug و Release در دسترس برنامه‌نویس است. وقتی برنامه، در حالت Debug، بیلد می‌شود، تنظیمات، به گونه‌ای است که بهینه‌سازی‌های کم‌تری انجام می‌گیرند و این حالت، برای پیداکردن خطاهای برنامه، مناسب‌تر است. وقتی برنامه، در حالت Release بیلد می‌شود، بهینه‌سازی‌های بیش‌تری در آن انجام می‌گیرد و فایل اجرایی نهایی، سریع‌تر است. به همین سبب، معمولاً، برای توزیع فایل اجرایی نهایی در بین کاربران، آن را با روش Release بیلد می‌کنیم.

در ویژوال‌استودیو، برای انتخاب بین Debug و Release، در قسمت بالای آی‌دی‌ای، می‌توانیم گزینهٔ مناسب را انتخاب کنیم:



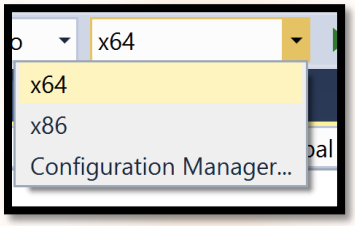
همچنین در قسمت Configuration Manager...، می‌توان انواع جدیدی از بیلد را تعریف کرد که در این کتاب کاری با آن نداریم و تنها، همین دو نوع پیش‌فرض از بیلد را در نظر می‌گیریم.

در کیوت‌کریییتور، برای انتخاب بین Debug و Release، در قسمت چپ آی‌دی‌ای می‌توانیم گزینهٔ مناسب را انتخاب کنیم:



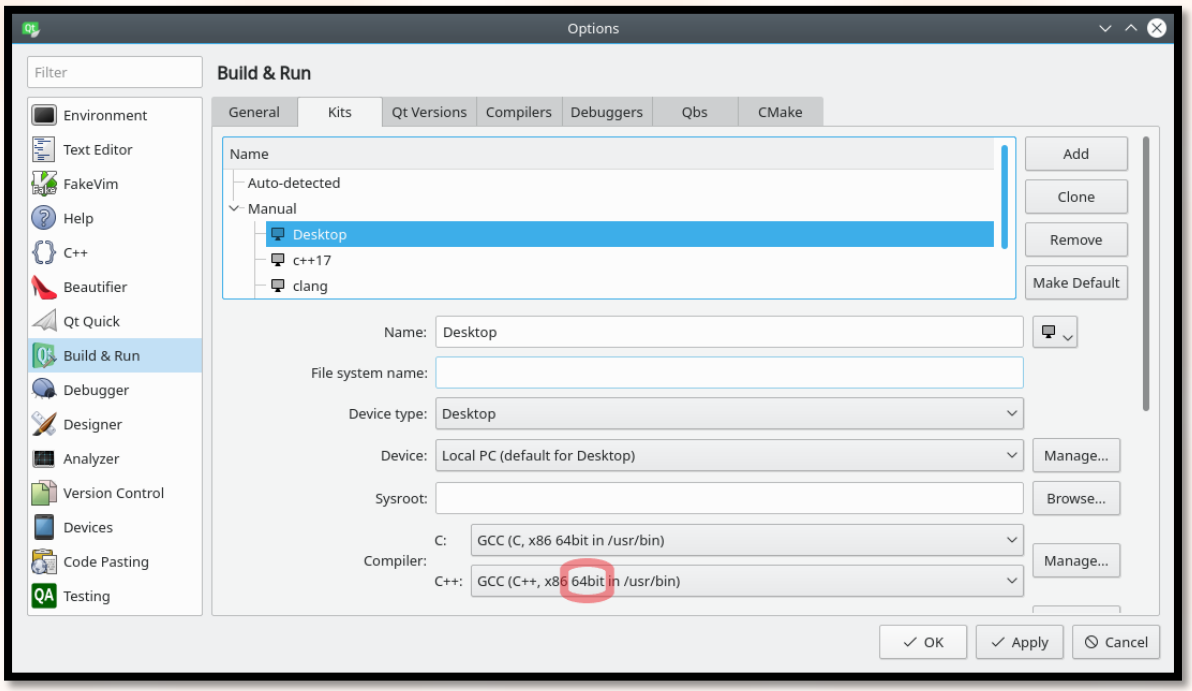
در کیوت‌کریییتور نیز، می‌توان تغییراتی در هر یک از این روش‌های بیلد انجام داد، اما در این کتاب، شکل پیش‌فرض آن‌ها را در نظر می‌گیریم.

برنامه‌ها را می‌توان به صورت ۳۲بیتی یا ۶۴بیتی بیلد کرد. این روش‌ها، تعیین می‌کنند که فایل اجرایی نهایی، ۳۲بیتی باشد یا ۶۴بیتی. فایل‌های اجرایی ۶۴بیتی امکان استفاده از حافظهٔ بیش‌تری را دارند. در ویژوال استودیو، در قسمت بالای آی‌دی‌ای، می‌توان بین این دو روش انتخاب کرد:



در تصویر بالا، x64، به معنی ۶۴بیتی و x86 به معنی ۳۲بیتی است. اغلب سیستم‌عامل‌های امروزی، ۶۴بیتی هستند و به همین دلیل، بهتر است برنامه‌ها را به صورت ۶۴بیتی کامپایل کنیم. در کیوت‌کریییتور، انتخاب بین این دو روش بیلد، در قسمت انتخاب کامپایلر رخ می‌دهد:

Tools → Options... → Build & Run → Kits:



در پنجرهٔ بالا، نام نسخهٔ ۶۴بیتی کامپایلر gcc را مشاهده می‌کنید. با کلید Add، می‌توان Kitهای دیگری اضافه کرد که از نسخهٔ ۳۲بیتی این کامپایلر (یا کامپایلرهای دیگر) استفاده کنند. در کیوت‌کریییتور، همواره، تنظیمات پیش‌فرض را در نظر خواهیم گرفت و کاری به بیلد ۳۲بیتی در آن نداریم. بنابراین، لازم نیست چیزی را عوض کنید.[۵۱]

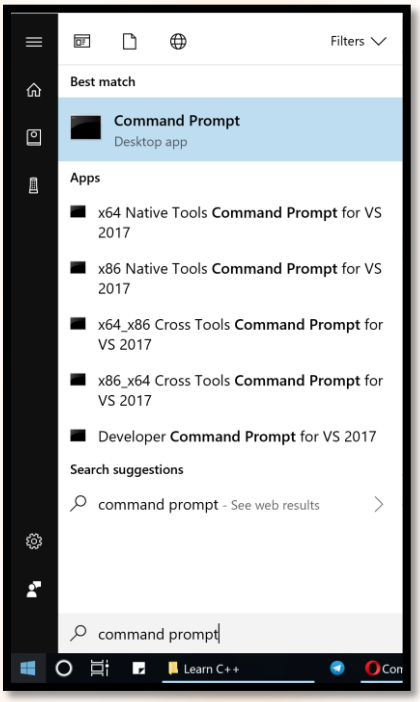
توجه کنید که کیوت‌کریییتور، تنها یک آی‌دی‌ای است؛ کامپایلر اصلی مورد استفادهٔ آن، (معمولاً) gcc است که در پشت‌پرده، کارها را انجام می‌دهد. در این کتاب، ممکن است، کیوت‌کریییتور را، به شکلی معرفی کنیم که گویی همهٔ

[۵۱] اما اگر بتوانید نسخهٔ gcc-8 (یا بالاتر) را نصب و به کامپایلر اضافه کنید بهتر است.

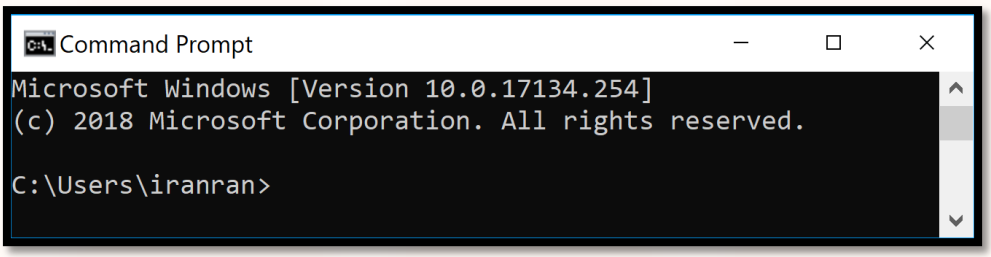
کارها را خودش انجام می‌دهد. این تصور، برای آموزش بهتر است. «ویژوال‌استودیو»، هم‌زمان هم یک آی‌دی‌ای و هم یک کامپایلر است. در مرحلهٔ ابتدایی آموزش، می‌توانید این تصور را در مورد «کیوت‌کریییتور» نیز داشته باشید.

آشنایی با CLI

با استفاده از یک CLI[۵۲]، می‌توان دستوراتی، مثل کپی-پیست، ایجاد فولدر جدید، دانلود یک فایل و... را، به صورت یک دستور متنی به سیستم‌عامل داد. زبان و چگونگی نوشتن دستورات بین سیستم‌عامل‌ها مشترک نیست. در ویندوز CLIای به نام Command Prompt وجود دارد که با جستجو، می‌توانید آن را پیدا کنید:



پنجرهٔ Command Prompt، به این صورت است:

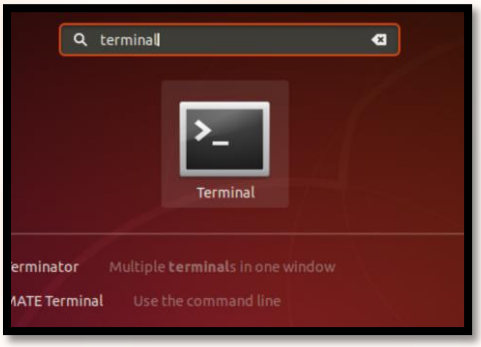


در آن تایپ کنید `CLS` و Enter را فشار دهید؛ می‌بینید که نوشته‌های پنجره، همه، پاک می‌شوند. تایپ کنید `dir` و Enter را فشار دهید؛ می‌بینید نام‌های دسته‌ای از فایل‌ها ظاهر می‌شوند؛ این‌ها همان فایل‌هایی هستند که در

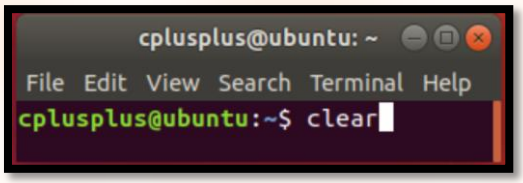
[۵۲] Command-line interface

مکانی از حافظهٔ دائم که **مسیر جاری**[۵۳] نامیده می‌شود، قرار دارند. `CLS` و `dir`، دستورهای command-line برای ویندوز هستند. دستورهای فراوان دیگری نیز وجود دارند که می‌توانید آن‌ها در اینترنت بیابید.

در اوبونتو، برای اجرای CLI، کلمهٔ terminal را جستجو کنید:



و سپس، برنامهٔ Terminal را اجرا کنید:



در این برنامه، می‌توان دستورات مخصوص سیستم‌عامل را تایپ و اجرا کرد؛ برای مثال، دستور `clear` سبب پاک‌شدن محتویات پنجرهٔ CLI می‌گردد.

[53] current path

فصل دوم – اولین برنامه‌ها

در این فصل، با ساختار اولیهٔ زبان C++ آشنا می‌شویم و نوشتن برنامه‌های ابتدایی را می‌آموزیم.

عدد، کاراکتر، رشته

در این قسمت، اشیاء ابتدایی برنامه‌ها معرفی خواهند شد، اشیائی مانند اعداد، که می‌توان آن‌ها را مستقیماً در متن برنامه وارد کرد.

آشنایی با عدد کاراکتر و رشته

این امکان هست که عددها، کاراکترها و رشته‌ها را، مستقیماً، در متن برنامه بنویسیم. برای نمونه، می‌توانید در Program.cpp کدهای زیر را بنویسید:

```
1397
3.1415926535
'A'
"This is a test."
```

1397 یک عدد صحیح است و 3.1415926535 یک عدد اعشاری است[۵۴]. چهار عمل اصلی ریاضی را نیز می‌توان، در متن برنامه، برای اعداد به کار برد:

مثال	توضیح	عمل
6 + 5	جمع	+
6 - 5	تفریق	-
5 * 6	ضرب	*
12 / 5	تقسیم	/

در C++، به جای (اصطلاح ریاضی) **عمل**[۵۵]، می‌گویند **عملگر**[۵۶]. یک نکتهٔ مهم دربارهٔ عملگرهای جدول بالا این است که وقتی یک عملگر، برای دو عدد صحیح به کار می‌رود، حاصل نیز، عددی صحیح است و قسمت اعشاری حذف می‌شود. برای نمونه، می‌دانیم ۱۲ ÷ ۵ = ۲/۴؛ اما حاصل 12 / 5 برابر با ۲ است؛ زیرا با توجه به صحیح‌بودن 12 و 5، قسمت اعشاری ۲/۴ حذف می‌گردد. برای این که قسمت اعشاری حذف نشود، باید عددها را به صورت اعشاری بنویسیم؛ مثلاً بنویسیم:

```
12.0 / 5.0
```

می‌دانید که ۱۲/۰ با ۱۲ تفاوتی ندارد.

---

[۵۴] در فارسی برای ممیز، از خط مورّب (slash) استفاده می‌شود. در انگلیسی برای ممیز از نقطه استفاده می‌شود.

[۵۵] operation

[۵۶] operator

'A'، یک **کاراکتر**[۵۷] است. کاراکتر، یک حرف از زبان انگلیسی یا یک نماد است. می‌بینید که برای واردکردن مستقیم یک کاراکتر در برنامه، آن را درون '' می‌نویسیم. اگر بخواهیم 'A' را دقیق‌تر معرفی کنیم، می‌گوییم 'A' (کد) اَسکی[۵۸] یا عدد یونیکد[۵۹] کاراکتر A است. همین‌طور، '2'، اسکی کاراکتر 2 است. با این که 'A'، اسکی کاراکتر A است، به خود 'A' نیز کاراکتر می‌گوییم. در واقع، یک کاراکتر دو ماهیت دارد، یکی نماد آن و دیگری اسکی آن. اسکی یک کاراکتر، در حقیقت، شمارهٔ آن کاراکتر در جدول اسکی‌هاست. برای مثال، شمارهٔ کاراکتر A در جدول اسکی‌ها، برابر با 'A' است و این شماره، ۶۵ است. اما بهتر است به کاراکترها، به‌خصوص در ابتدای برنامه‌نویسی، به چشم عدد و شماره نگاه نکنیم.

"This is a string" یک رشته است. رشته، دسته‌ای از کاراکترهای کنار هم است. معمولاً، رشته، یک جمله به زبان انگلیسی است. اگر بخواهیم یک رشته را به طور مستقیم در متن برنامه بنویسیم، باید کاراکترهای آن را، درون "" بگذاریم.

ویرایشگر آی‌دی‌ای، به محض تشخیص یک رشته یا کاراکتر، رنگ آن را عوض می‌کند. البته در برنامه‌نویسی، رنگ‌ها نقشی ندارند و می‌توان، در تنظمات، رنگ‌ها را تغییر داد.

سؤال

۱) کدام گزینه درست است؟

- در C++، 2 / 4 می‌شود ۰.۱
- در C++، 3 / 4 می‌شود ۰.۲
- در C++، 4 / 4 می‌شود ۰.۱

تمرین

۱) کدام یک از گزینه‌های زیر، کاراکتر نیست؟

- '"'
- '\'
- 'ab'
- '"a"'

راهنمایی: فقط یکی از گزینه‌ها کاراکتر نیست. برای تشخیص گزینهٔ درست آن‌ها را در Program.cpp بنویسید و با توجه به تغییر رنگ، گزینهٔ صحیح را پیدا کنید.

توجه: گزینه‌های سوال کمی غیر عادی هستند، نباید با دیدن آن‌ها تصور شما از کاراکتر به هم بریزد.

۲) کدام یک از گزینه‌های زیر رشته است؟

- "\\\"
- "\\\\\"
- "این یک رشته نیست"
- "'''''"

[۵۷] character

[۵۸] ASCII: American Standard Code for Information Interchange

[۵۹] unicode

آشنایی با متغیرها

در ریاضیات، از **نمادها**، برای نشان‌دادن اعداد، تابع‌ها، مجموعه‌ها و چیزایی مانند آن‌ها استفاده می‌کنیم. برای نمونه، می‌نویسیم:

$$x = ۲$$

یا:

$$x^۲ = ۴$$

و گاه، می‌گوییم برای هر عدد x:

$$۰ \leq x^۲$$

به x، که نمایندهٔ یک عدد است، **متغیر** گفته می‌شود.

در برنامه‌های رایانه‌ای نیز، می‌توان از متغیرها استفاده کرد. اما یک تفاوت مهم وجود دارد. در ریاضیات، همه‌چیز با ذهن انعطاف‌پذیر انسان سروکار دارد. اما در رایانه، ذهنی مثل ذهن انسان وجود ندارد. وقتی قرار است رایانه از یک متغیر استفاده کند، باید ابتدا به رایانه بگوییم که این متغیر، چند بایت از حافظه را اشغال می‌کند و سپس، باید، مکان مناسبی از حافظه، برای آن متغیر، مشخص گردد. اما برای ذهن انسان، توجه به این جزئیات، لازم نیست. به این ترتیب، برای تعریف و استفاده از یک متغیر در یک برنامهٔ رایانه‌ای، باید دو چیز مشخص باشد:

۱) مقدار حافظه‌ای که متغیر نیاز دارد.
۲) آدرس مکانی از رَم (حافظهٔ موقت) که قرار است متغیر در آن قرار بگیرد. به این آدرس، **آدرس**[۶۰] آن متغیر می‌گویند.

متغیر، پیش از استفاده، باید **تعریف**[۶۱] شود. با تعریف یک متغیر، اندازه و آدرس آن متغیر مشخص می‌گردد.

تصور کنید که شخصی می‌خواهد اتومبیل خود را در یک پارکینگ پارک کند؛ برای این کار، باید محلی آزاد از پارکینگ، متناسب با اندازهٔ اتومبیل مشخص گردد و اتومبیل در آنجا پارک شود و پس از پارک، آدرس محل پارک باید معلوم باشد تا بتوان اتومبیل را پیدا کرد. فرض کنید این شخص پس از پارک، از پارکینگ خارج می‌شود و از شخص دیگری می‌خواهد که کیف او را، که در اتومبیل جا مانده است، بیاورد. برای این کار، محل پارک را در کاغذی یادداشت می‌کند و به او می‌دهد. به همین صورت، وقتی قرار است متغیری را تعریف کنیم، باید اندازه و محل استقرار آن در رم مشخص شود. نام متغیر، مانند کاغذ یادداشتی است که با آن، می‌توان به محلی از حافظه که متغیر در آنجاست، دسترسی پیدا کرد. البته خود کاغذ یادداشت، آدرس نیست، بلکه آدرس با استفاده از آن پیدا می‌شود.

برای این که اندازهٔ یک متغیر را مشخص کنیم، از **نوع**[۶۲]، استفاده می‌کنیم. مثلاً، اگر در C++، بخواهیم متغیری با نام x داشته باشیم که چهار بایت از حافظه را می‌گیرد، می‌نویسیم:

---

۶۰ address
۶۱ define

```
int x;
```

که int یک نوع است که برای ایجاد متغیرهای چهاربایتی به کار می‌رود. خط بالا، متغیر x را تعریف می‌کند؛ و علاوه بر آن، آدرسش را نیز مشخص می‌کند و این آدرس، از طریق نام متغیر، یعنی x، قابل‌دستیابی است. ما، به عنوان برنامه‌نویس، معمولاً، در تعیین آدرس متغیر در رم (حافظهٔ موقت) نقشی نداریم. آدرس، خود به خود تعیین می‌شود؛ همان‌گونه که ممکن است مسئول پارکینگ، خود، محل پارک اتومبیل را معین کند، کامپایلر، خود، محل ایجاد متغیر را تعیین می‌کند. **متغیر، نمایندهٔ بخشی از حافظه است و با مقداردادن به متغیر، به حافظه‌ای که متغیر نمایندهٔ آن است، مقدار می‌دهیم.**

یک ساختمان ده‌طبقه، یک ساختمان است که از ده طبقه (آپارتمان) تشکیل شده است. متغیرها را نیز، می‌توان به هم چسباند و یک ساختمان چندین‌طبقه از متغیرها ساخت که به آن **آرایه**[۶۳] می‌گویند. اما همان‌طور که طبقه‌های ساختمان، همه، هم‌اندازه هستند؛ متغیرهای تشکیل‌دهندهٔ یک آرایه نیز، باید هم‌اندازه باشند. در آینده، آرایه‌ها به طور کامل معرفی خواهند شد.

سوال

۱) گزینهٔ نادرست کدام است؟

- یک متغیر در برنامه‌نویسی، نمایندهٔ بخشی از رم است.
- نوع، مکان متغیر را در رم مشخص می‌کند.
- مقدار یک متغیر، در برنامه‌نویسی، بر خلاف متغیرهای ریاضی قابل‌تغییر است.
- نام یک متغیر می‌تواند abcdefghijklmopqrstuvwxyz باشد.

نوع‌ها و مقداردادن به متغیرها

هنگامی که می‌خواهیم یک متغیر جدید تعریف کنیم، باید مشخص کنیم که این متغیر، به چند بایت از حافظه نیاز دارد. همان‌طور که پیش‌تر گفته شد، برای مشخص‌کردن تعداد بایت‌های مورد نیاز، از **نوع** استفاده می‌کنیم. نوع یک متغیر، تعداد بایت‌های حافظهٔ آن را معین می‌کند. البته این، تنها کارکرد نوع نیست. نوع، به علاوه، مشخص می‌کند که متغیر، قرار است برای ذخیرهٔ چه نوع اطلاعاتی به کار رود؛ به عنوان نمونه، نوع مشخص می‌کند که آیا متغیر، قرار است اعداد صحیح را ذخیره کند یا اعداد اعشاری یا کاراکترها یا چیز دیگری را. برخی از نوع‌های موجود در C++، در جدول زیر آمده‌اند:

نوع	تعداد بایت‌ها	متغیر ایجاد شده توسط آن	توضیح
bool	۱ بایت	عبارت بولی	عبارت بولی، یعنی عبارتی که یا درست است یا نادرست
char	۱ بایت	کاراکتر	نماد یک حرف یا نماد یک عدد یا نماد یک علامت
const char*	۴ بایت	رشته	دسته‌ای از کاراکترها که یک جمله و... تشکیل می‌دهند
double	۸ بایت	عدد اعشاری	-
int	۴ بایت	عدد صحیح	عدد بدون قسمت اعشاری که می‌تواند منفی هم باشد

[۶۲] type

[۶۳] array

عدد بدون قسمت اعشاری که نمی‌تواند منفی باشد	عدد صحیح نامنفی	۴ بایت	unsigned

چند مثال که روش استفاده از نوع را نشان می‌دهند:

مثال: اگر بخواهیم متغیری داشته باشیم که اعداد صحیح، یعنی اعداد بدون قسمت اعشاری، مانند عدد ۳، را در خود جا دهد، می‌نویسیم:

```
int a;
```

حال، a، می‌تواند هر عدد صحیحی را در خود ذخیره کند، به شرط آن که بتوان عدد را در چهار بایت جا داد. برای قراردادن عدد ۲ در این متغیر، می‌نویسیم:

```
a = 2;
```

حال، a، یک متغیر است و مقدار آن ۲ است.

در انتهای دستورات C++، علامت «;» را قرار می‌دهیم تا پایان دستور مشخص باشد. قرارندادن آن، اغلب، سبب ایجاد خطا در برنامه می‌شود. این علامت سمی‌کولون[۶۴] نام دارد.

مثال: "Donald Duck" یک رشته است. برای قراردادن این رشته در یک متغیر می‌نویسیم:

```
const char* c = "Donald Duck";
```

پس از اجرای این خط، c یک متغیر است و مقدار آن، "Donald Duck" است.

مثال: 'g' یک کاراکتر است و می‌توانیم بنویسیم:

```
char ch = 'g';
```

پس از اجرای این خط، ch یک متغیر و مقدار آن 'g' است.

در مثال‌های بالا، همراه با تعریف متغیرها، به آن‌ها مقدار (اولیه) هم داده شده است. در مثال اول، با نوشتن:

```
a = 2;
```

مقدار ۲ را به متغیر a داده‌ایم. می‌توان به یک متغیر، بیش از یک بار مقدار داد. برای مثال، می‌توان نوشت:

```
double r = 2.3;
r = 5;
r = 7.91;
```

[۶۴] semicolon

پس از انجام این دستورات، مقدار r، ۷/۹۱ است. یعنی وقتی دوباره به یک متغیر مقدار می‌دهیم، مقدار قبلی پاک می‌شود و مقدار جدید در آن قرار می‌گیرد و همواره، مقدار متغیر، برابر با مقداری است که آخرین بار به آن داده شده است.

در ریاضیات، به عبارت‌هایی مانند $x \leq 0$ یا $12 > 10$ زیاد برخورد می‌کنیم. این عبارت‌ها، در برنامه‌نویسی نیز ظاهر می‌شوند. به این عبارات در ریاضیات، **گزاره**[۶۵] و در برنامه‌نویسی، **عبارت بولی**[۶۶] می‌گویند. یک عبارت بولی، می‌تواند درست یا نادرست باشد؛ یعنی تنها دو حالت «درست» یا «نادرست» را به خود می‌پذیرد. برای مثال، «10 < 12»، یک عبارت بولی درست است و «2 < 3»، یک عبارت بولی نادرست است و نیز، «x < 0»، یک عبارت بولی است که مقدارش (یعنی درست یا نادرست‌بودنش)، پس از تعیین مقدار x معلوم می‌گردد. در C++، متغیرهای دارای نوع bool، می‌توانند عبارت‌های بولی را به عنوان مقدار، در خود ذخیره کنند. برای مثال، می‌توانیم بنویسیم:

```
bool b = (10 < 12);
```

const char*، یک نوع ستاره‌دار است؛ یعنی نوعی است که در انتهایش «*» قرارگرفته است. نوع‌های ستاره‌دار، با بقیهٔ نوع‌ها فرق دارند و باید با آن‌ها، به شکل دیگری برخورد کرد. به یک متغیر که نوع ستاره‌دار دارد، «اشاره‌گر[۶۷]» می‌گویند. تفاوت اشاره‌گرها با سایر متغیرها این است که متغیرهای معمولی؛ نمایندهٔ بخشی از حافظه هستند، اما اشاره‌گرها، تنها می‌گویند که مکان مورد نظر از حافظه کجاست؛ یعنی اشاره‌گر، نمایندهٔ متغیری است که حاوی آدرس یک محل از حافظه است و با متغیرهای عادی، که مستقیماً نمایندهٔ آن بخش از حافظه‌اند، تفاوت دارد. با تغییر مقدار یک متغیر عادی، مقدار آن بخش حافظه که متغیر نماینده‌اش است، تغییر می‌کند؛ اما با تغییر مقدار اشاره‌گر، مقدار حافظه‌ای که اشاره‌گر به آن اشاره داشته است، تغییر نمی‌کند. شاید این مقایسه برای درک این تفاوت مناسب باشد: یک اشاره‌گر مانند نقشهٔ گنج است، اما یک متغیر عادی، مثل صندوق گنج.

اشاره‌گرها، معمولاً، در مواردی به کار می‌روند که حافظهٔ مورد نظر زیاد باشد. برای نمونه، از آنجا که یک رشته، حافظهٔ نسبتاً زیادی می‌خواهد، متغیر مناسب برای آن، باید از نوع اشاره‌گر باشد. در فصل مجزّایی، اشاره‌گرها، به طور کامل، معرفی خواهند شد. تا به اینجا، می‌توانید const char* را، مانند بقیهٔ نوع‌ها به حساب بیاورید.

مثال: به این کد توجه کنید:

```
int a;
int b;
a = 100;
b = a;
```

در اینجا، ابتدا دو متغیر a و b تعریف شده‌اند. سپس، مقدار ۱۰۰ به a داده شده است، و در نهایت، مقدار a (که ۱۰۰ است) به b داده شده است؛ یعنی، پس از اجرای این خطوط برنامه، مقدار b، ۱۰۰ خواهد بود.

---

[۶۵] proposition

[۶۶] Boolean expression

[۶۷] pointer

مثال بالا می‌گوید که می‌توان نام متغیر را، به جای مقدارش به کار برد؛ یعنی وقتی مقدار a برابر با ۱۰۰ است، می‌توان با a، (در بسیاری از موارد،) مانند 100 برخورد کرد. همان‌طور که در ریاضیات وقتی می‌نویسیم x = ۱۰۰، بین x و ۱۰۰ هیچ تفاوتی قائل نمی‌شویم؛ در کد بالا نیز، وقتی مقدار a، برابر ۱۰۰ است، (در بسیاری از موارد،) بین a و 100، تفاوتی قائل نمی‌شویم. اما باید توجه داشت که متغیرها، در برنامه‌نویسی، با نمادهای ریاضی فرق دارند. متغیرها، حافظه‌ای مشخص و فیزیکی دارند و ممکن است مقدارشان با تغییر بیت‌های حافظه، تغییر کند و هیچ یک از این‌ها، در ریاضیات و نمادهای آن مطرح نیستند؛ با این حال، سعی شده است که کارکرد آن‌ها، شبیه به کارکرد نمادهایی مانند x در ریاضیات باشد.

مثال: کمی پیش می‌رویم و از عملگر جمع استفاده می‌کنیم:

```
int a = 8;
int b = 3;
int c = a + b;
```

در اینجا، ابتدا، مقدار a، ۸ قرار داده شده و مقدار b، ۳ گذاشته شده است. سپس، مقدار c، حاصل جمع مقدارهای a و b، تعیین شده است. پس از اجرای این کدها، مقدار c، ۱۱ خواهد بود.

سوال

۱) به نظر شما، آیا بین کد:

```
int a;
a = 1111;
```

و کد:

```
int a = 1111;
```

تفاوتی هست؟

۲) هر کاراکتر، ۱ بایت حافظه می‌خواهد. رشته، مجموعه‌ای از کاراکترهاست. پس، "hello"، (حداقل) به ۵ بایت حافظه نیاز دارد. با توجه به جدول بالا، می‌دانیم که اگر تعریف کنیم:

```
const char* c;
```

c، تنها ۴ بایت حافظه خواهد داشت. با این حال، می‌توانیم بنویسیم:

```
c = "hello";
```

چگونه ۵ بایت در ۴ بایت جا می‌شود؟

شروع برنامه‌نویسی

قالب اولیهٔ همهٔ برنامه‌ها

کدهای مشترکی هستند که تقریباً در همهٔ برنامه‌های این کتاب، وجود دارند. این کدها عبارتند از:

```
#include <iostream>
using namespace std;

int main()
{
    /////////
    cin.ignore(1);
}
```

دستورات اصلی برنامه، در جایی که با ///////// نشان داده‌ایم، نوشته خواهند شد. برای مثال، می‌توان کدهایی را که، پیش‌تر، دربارهٔ متغیرها آوردیم، در محل ///////// وارد کرد:

```
#include <iostream>
using namespace std;

int main()
{
    int a = 5;
    int b = 6;
    int c = a + b;
    cin.ignore(1);
}
```

چنان که پیداست، دستورات را، از ابتدای هر خط ننوشته‌ایم و آن‌ها را کمی جلوتر برده‌ایم. این کار، تنها برای زیبایی برنامه است و در نحوهٔ اجرای برنامه، هیچ اثر و اهمیتی ندارد. شما می‌توانید همهٔ دستورات را از ابتدای خطوط بنویسید. بنابراین، می‌توان گفت که کدهای:

```
#include <iostream>
using namespace std;

int main()
{
    /////////
    cin.ignore(1);
}
```

یک **قالب** را به وجود می‌آورند که در همهٔ برنامه‌ها وجود دارد. نوشته‌های فایل Program.cpp را کاملاً پاک کنید و این قالب را در آن بنویسید و آن را ذخیره[۶۸] کنید تا در ادامه، بتوانید، از این قالب، برای نوشتن برنامه‌ها استفاده کنید. ممکن است، در برخی از برنامه‌ها، به دلیلی، به جای:

```
cin.ignore(1);
```

بنویسیم:

```
cin.ignore(2);
```

[۶۸] save

که اثری بر کدهای اصلی برنامه ندارد. اما چرا C++ به قالب احتیاج دارد؟ و چرا برنامه را بدون هیچ قالبی از ابتدای فایل نمی‌نویسیم؟ به طور کلی، می‌توان گفت که علت این است که قالب، به برنامه، نظم منطقی می‌دهد. در واقع، منطقی ساده، پشت این قالب است که به تدریج، با آن آشنا خواهید گشت. در بخش‌های بعد، خواهید دید که قالب، قابل‌تغییر و گسترش است و این، سبب انعطاف‌پذیری برنامه می‌گردد. در بعضی زبان‌ها، مثل TELCOMP، قالبی در کار نیست و برنامه، از اول فایل، بی هیچ مقدمه‌ای، آغاز می‌شود. به چنین زبان‌هایی، غیرساخت‌یافته[۶۹] می‌گویند.

اولین برنامهٔ شما

مرسوم است که اولین برنامه، برنامه‌ای باشد که پیام:

```
Hello World!
```

را بر صفحهٔ نمایش چاپ می‌کند[۷۰]؛ که نوعی خوشامدگویی به دنیای برنامه‌نویسی است. برای چاپ‌کردن، از دستور cout << استفاده می‌کنیم[۷۱]. برای چاپ هر چیزی (مثل عدد، رشته و...) آن را پس از دستور مذکور می‌نویسیم. به عنوان نمونه، خط:

```
cout << 2;
```

عدد 2 را چاپ می‌کند. برنامهٔ زیر، اولین برنامهٔ شماست:

```
// Program ID: 0010
#include <iostream>
using namespace std;

int main()
{
    cout << "Hello World!";
    cin.ignore(1);
}
```

این برنامه را در Program.cpp قرار دهید و آن را اجرا کنید. خروجی زیر را دریافت خواهید کرد:

خروجی:

```
Hello World!
```

نحوهٔ اجرای یک برنامه، در ویژوال‌استودیو و در کیوت‌کریتور، در فصل اول توضیح داده شده است. خط مهم برنامهٔ فوق، این خط است:

```
cout << "Hello World!";
```

"Hello World!" یک رشته است که با cout << چاپ شده است. در برنامهٔ فوق، خط:

```
// Program ID: 0010
```

---

[۶۹] Non-structured

[۷۰] «چاپ کردن» در اینجا به معنی نوشتن بر صفحهٔ مانیتور است نه پرینت گرفتن روی کاغذ.

[۷۱] cout را see-out تلفظ کنید.

جزء برنامه محسوب نمی‌شود و تنها شمارهٔ برنامه را، که در اینجا 10 است، مشخص می‌کند. با این شماره، در این کتاب، می‌توانیم به برنامه‌ها اشاره کنیم. برای نمونه، برنامهٔ فوق برنامهٔ (شمارهٔ) 10 است. در نسخهٔ فعلی از کتاب، از این شماره‌ها در متن استفاده نخواهد شد و آن‌ها تنها برای یاری خواننده در پیداکردن برنامه‌ها مفیدند.

چاپ‌کردن بر صفحهٔ نمایش

چاپ کردن، همان نوشتن در پنجره‌ای است که هنگام اجرای برنامه ظاهر می‌شود. به این پنجره، **پنجرهٔ کنسول** یا **پنجرهٔ خروجی** می‌گوییم، و در واقع، یک پنجرهٔ CLI است. در ویندوز ۱۰، این پنجره، پس‌زمینهٔ سیاه با کاراکترهای سفید دارد. در اوبونتو، پس‌زمینه، بسته به تنظیمات و محیط، می‌تواند سفید یا سیاه باشد.

در بخش قبل، دیدید که با cout >>، می‌توان رشته‌ها را در پنجرهٔ کنسول چاپ کرد. با این دستور، تقریباً هر چیزی را می‌شود چاپ کرد؛ از جمله مقدار متغیرها را. برای مثال، اگر تعریف کنیم:

```
const char* c = "This is a test";
```

می‌توانیم مقدار c را، با cout >>، در پنجرهٔ خروجی چاپ کنیم؛ برای چاپ، می‌نویسیم:

```
cout << c;
```

مثال: برنامهٔ بخش قبل را، می‌توان به صورت زیر تغییر داد، به طوری که خروجی تغییر نکند:

```
// Program ID: 0020
#include <iostream>
using namespace std;

int main()
{
    const char* c = "Hello World!";
    cout << c;
    cin.ignore(1);
}
```

خروجی:

```
Hello World!
```

می‌بینید که خروجی این برنامه، با برنامهٔ قبل، هیچ تفاوتی ندارد. همان‌طور که پیش‌تر گفتیم، در پایان همهٔ دستورات، در C++، «;» قرار می‌گیرد؛ به جز موارد خاصی که در آینده به آن‌ها اشاره خواهیم کرد.

با cout >>، می‌توان هر چیز دیگری را نیز چاپ کرد؛ مثلاً یک عدد یا یک کاراکتر.

مثال: برنامهٔ زیر دو عدد را با هم جمع و حاصل را چاپ می‌کند:

```
// Program ID: 0030
#include <iostream>
using namespace std;

int main()
{
```

```
    int a;
    int b;
    int c;
    a = 2;
    b = 3;
    c = a + b;
    cout << c;
    cin.ignore(1);
}
```

خروجی:

```
5
```

مثال: برنامهٔ مثال قبل را می‌توان جمع‌وجورتر هم نوشت:

```
// Program ID: 0040
#include <iostream>
using namespace std;

int main()
{
    int a = 2, b = 3, c = a + b;
    cout << c;
    cin.ignore(1);
}
```

خروجی:

```
5
```

در واقع، به جای:

```
int a;
a = 2;
```

می‌توانیم بنویسیم:

```
int a = 2;
```

و به جای:

```
int a = 2;
int b = 3;
```

می‌توان نوشت:

```
int a = 2, b = 3;
```

و در آخر، به جای:

```
int a = 2, b = 3;
int c = a + b;
```

می‌شود نوشت:

```
int a = 2, b = 3, c = a + b;
```

و به این ترتیب، از تکرار int جلوگیری می‌شود. استفاده از این مختصرنویسی، به انتخاب و سلیقهٔ برنامه‌نویس بستگی دارد.

دریافت مقدار از پنجرهٔ کنسول

برای این که بتوانیم یک مقدار را از پنجرهٔ کنسول بگیریم و آن را به یک متغیر بدهیم، از دستور cin >>، استفاده می‌کنیم. cin >>، بر عکس cout << عمل می‌کند. cout << مقدار یک متغیر را در پنجرهٔ کنسول نشان می‌دهد، در حالی که cin >>، یک مقدار را از پنجرهٔ خروجی می‌گیرد و در متغیر قرار می‌دهد.[۷۲]

مثال: برنامهٔ زیر، مقداری را از پنجرهٔ خروجی می‌گیرد، این مقدار را به توان ۲ می‌رساند و چاپ می‌کند. شاید لازم به یادآوری باشد که، در C++، برای ضرب‌کردن دو عدد از «*» استفاده می‌شود نه از «×».

```
// Program ID: 0050
#include <iostream>
using namespace std;

int main()
{
    int a = 0;
    cin >> a;
    cout << a * a;
    cin.ignore(2);
}
```

خروجی:

```
9
81
```

وقتی این برنامه را اجرا می‌کنیم، برنامه، ابتدا، منتظر یک عدد صحیح می‌ماند. عدد ۹ را وارد کرده‌ایم و **کلید Enter را فشار داده‌ایم**. با این کار، مقدار a، برابر با ۹ می‌شود. سپس، توان دوم آن چاپ می‌گردد. همان‌طور که می‌بینید عدد ۸۱ چاپ شده است.

شاید این سوال پیش بیاید که چرا، در برنامهٔ فوق، به جای ضرب عدد در خودش، آن را به توان ۲ نرساندیم؟ پاسخ روشن است؛ زیرا در C++، برخلاف برخی از زبان‌ها، عملگر توان نداریم. در C++ عملگر «^» هست اما معنای دیگری دارد و برای توان از آن استفاده نمی‌شود. البته در آینده، خواهید دید که C++، آن‌قدر انعطاف دارد که برنامه‌نویس، می‌تواند قابلیت «به توان رساندن» را به برنامهٔ خود اضافه کند.

[۷۲] cin را see-in تلفظ کنید.

می‌توان از cin >> ، برای گرفتن تقریباً هر نوع مقداری، از جمله، عدد صحیح، عدد اعشاری، کاراکتر یا رشته استفاده کرد. اما گرفتن رشته با آن، جزئیاتی دارد، که در آینده گفته خواهند شد؛ و قطعه‌کد زیر خطا دارد:

```
const char* c;
cin >> c;
```

در مورد رشته، چیزهای زیادی هستند که هنوز تبیین نشده‌اند.

اما چرا اصلاً مقدار متغیر را از پنجرهٔ کنسول بگیریم؟ چرا نباید آن را در خود برنامه به متغیر بدهیم؟ پاسخ این است که هر برنامه، یک فایل اجرایی می‌سازد. فایل اجرایی، به صورت نرم‌افزار فروخته یا منتشر می‌شود. اما متن برنامه، همراه با فایل اجرایی منتشر نمی‌گردد. از این رو، همیشه نمی‌توان مقدار مناسب را در متن برنامه، به متغیر داد؛ و در مواردی، لازم است مقدارهایی را، کاربر، یعنی کسی که تنها به فایل اجرایی برنامه دسترسی دارد، به برنامه بدهد. پس مقدار بعضی متغیرها، باید حتماً از کاربر گرفته شوند. برای نمونه، ممکن در جایی لازم باشد، کاربر، اسم خود را وارد کند. برنامه‌نویس نمی‌تواند اسم کاربری را که نمی‌شناسد در متن برنامه قرار دهد.

در نهایت، لازم به تذکر است که وقتی از cin >> استفاده می‌کنیم، در قالب برنامه، به جای:

```
cin.ignore(1);
```

از:

```
cin.ignore(2);
```

استفاده خواهیم تا پنجرهٔ کنسول، (به خصوص در ویژوال‌استودیو،) خود به خود بسته نشود.

عملگرها

پیش‌تر، با عملگرهایی مثل جمع و ضرب آشنا شدید. در C++، عملگرهای زیادی وجود دارند، که برخی از آن‌ها را در جدول زیر می‌بینید:

مثال	توضیح	عملگر
a + b	جمع	+
a - b	تفریق	-
-a	منفی	-
a * b	ضرب	*
a / b	تقسیم	/
a++ یا ++a	اضافه کردن یک واحد	++
a-- یا --a	کم کردن یک واحد	--
a % b	باقی‌ماندهٔ تقسیم دو عدد	%

می‌دانید که عملگرهای جمع، تفریق، ضرب و تقسیم را، می‌توان برای اعدادی که در متن برنامه ظاهر می‌شوند، مانند 2 و 3، به کار برد:

```
2 + 3
```

آن‌ها را برای متغیرها نیز می‌توان به کار برد:

```
a + b
```

که در آن، **مقدار** a و b، به عنوان عدد در نظر گرفته می‌شود و مقدارها جمع می‌شوند. مثال زیر، این را نشان می‌دهد:

مثال: جمع متغیرها و اعداد:

```
// Program ID: 0060
#include <iostream>
using namespace std;

int main()
{
    int a = 2, b = 3;
    cout << a + b + 5;
    cin.ignore(1);
}
```

خروجی:

```
10
```

برنامهٔ فوق، حاصل جمع عددهای ۲، ۳ و ۵ را چاپ می‌کند. a، مقدار ۲ و b، مقدار ۳ دارد، پس:

```
a + b + 5
```

با:

```
2 + 3 + 5
```

تفاوتی ندارد و حاصل آن، برابر با ۱۰ است که این مقدار، در پنجرهٔ خروجی چاپ شده است.

هدف برنامهٔ بالا این است که نشان دهد که وقتی دو متغیر عددی را، با هم جمع می‌کنیم، مقادیر متغیرها، به عنوان اعداد، برای محاسبات در نظر گرفته می‌شوند (نه خود متغیرها). چیز دیگری که می‌توان از این برنامه دریافت، این است که با cout >>، می‌توان مستقیماً a + b + 5 را چاپ کرد.

عملگرهای ++ و --، فقط بر متغیرها عمل می‌کنند. یعنی می‌توانیم بنویسیم:

```
int a = 3;
a++;
```

اما نمی‌توانیم بنویسیم:

```
3++;
```

زیرا 3 یک عدد است[۷۳] نه یک متغیر. کارکرد «++» این است که مقدار متغیر، را یکی زیاد می‌کند. برای مثال، اگر مقدار متغیر عددی a، ۳ باشد با نوشتن:

```
a++;
```

مقدار a می‌شود ۴.

کارکرد «--» این است که مقدار متغیر را یک واحد کاهش دهد. برای مثال، اگر مقدار متغیر عددی a، ۳ باشد با نوشتن:

```
a--;
```

مقدار a می‌شود ۲.

مثال: به‌کاربردن عملگر ++ بر یک متغیر:

```
// Program ID: 0070
#include <iostream>
using namespace std;

int main()
{
    double a = 9.460;
    a++;
    cout << a;
    cin.ignore(1);
}
```

خروجی:

```
10.46
```

در این برنامه، عدد اعشاری a، تعریف شده و به آن مقدار ۹/۴۶ داده شده است. با اجرای:

```
a++;
```

مقدار a، به ۱۰/۴۶ تغییر یافته، و در آخر، مقدار نهایی a، چاپ شده است.

عملگر «-» دو کاربرد دارد: یکی برای تقریق و دیگری برای قرینه‌کردن اعداد. در مثال زیر، هر دو کاربرد این عملگر نشان داده شده‌اند.

مثال: برنامهٔ زیر قرینهٔ تفاضل دو عدد را چاپ می‌کند.

```
// Program ID: 0080
#include <iostream>
using namespace std;

int main()
{
    double a = 1.4142;
    int b = 1;
```

[۷۳] در واقع، یک «ثابت لیترال» است و لیترال‌ها در آینده بیشتر معرفی خواهند شد.

```
    cout << -(a - b);
    cin.ignore(1);
}
```

خروجی:

```
-0.4142
```

a و b دو متغیر عددی هستند، تفاضل آن‌ها، a - b است و قرینهٔ این تفاضل، (a - b)- است. در اینجا، عملگر «-» دو نقش مختلف دارد.

برنامهٔ بالا، حاوی چند نکته است: اول این که a و b، دو متغیر عددی با نوع‌های متفاوتند، اما چون عددی هستند، هرچند نوع‌های متفاوت دارند، می‌توان با عملگر - تفاضلشان را به دست آورد. دوم این که پرانتز، همان‌گونه که در ریاضیات استفاده می‌شود، در متن برنامهٔ C++ نیز قابل‌استفاده است و همان کاربرد را دارد.

مانند «-»، عملگر «+» نیز ماهیت دوگانه دارد. یعنی اگر a یک عدد باشد، می‌توان نوشت a+، که البته این هیچ تغییری در مقدار به وجود نمی‌آورد؛ یعنی مقدار a+ و a یکسان است؛ هرچند ماهیت آن‌ها لزوماً یکسان نیست؛ برای مثال، اگر a یک متغیر باشد، نمی‌توان نوشت:

```
+a = 2;
```

هرچند که می‌توان نوشت:

```
a = 2;
```

مثال: برنامهٔ زیر:

۱) یک عدد صحیح را از پنجرهٔ خروجی می‌گیرد.
۲) یک واحد به آن می‌افزاید.
۳) حاصل را به توان ۲ می‌رساند.
۴) از حاصل یکی کم می‌کند.
۵) و نتیجه را چاپ می‌کند.

اگر بخواهیم ساده‌تر بگوییم؛ برنامه، x را می‌گیرد و $(x+1)^2 - 1$ را چاپ می‌کند:

```
// Program ID: 0090
#include <iostream>
using namespace std;

int main()
{
    int a;
    cin >> a;
    a++;
    int b = a * a;
    b--;
    cout << b;
    cin.ignore(2);
}
```

خروجی:

```
5
35
```

خط‌های برنامه را با پنج مرحلهٔ بالا، مقایسه کنید. این برنامه، همان مراحل را انجام می‌دهد.

برنامهٔ فوق؛ تنها برای آموزش به این شکل نوشته شده است. اگر این قصد نبود، می‌توانستیم آن را به صورت زیر بنویسیم:

```
// Program ID: 0100
#include <iostream>
using namespace std;

int main()
{
    int a;
    cin >> a;
    cout << (a + 1) * (a + 1) - 1;
    cin.ignore(2);
}
```

خروجی:

```
5
35
```

در موارد متعددی در برنامه‌نویسی، فرمول‌های ریاضی، برنامه‌نویسی را آسان می‌کنند. مثال زیر، یکی از این موارد را نشان می‌دهد.

مثال: برنامه‌ای می‌نویسیم که طول و عرض یک مستطیل را بگیرد و مساحت آن را حساب کند:

```
// Program ID: 0110
#include <iostream>
using namespace std;

int main()
{
    double width, length;
    cout << "Enter width: ";     // عرض را وارد کن
    cin >> width;                // وارد شدن عرض توسط کاربر
    cout << "Enter length: ";    // طول را وارد کن
    cin >> length;               // وارد شدن طول توسط کاربر
    cout << "the area is: ";     // مساحت برابر است با
    cout << length * width;      // چاپ مساحت مستطیل
    cin.ignore(1);
}
```

خروجی:

```
Enter width: 3
Enter length: 4
the area is: 12
```

عدد ۳ را به عنوان عرض مستطیل، و عدد ۴ را به عنوان طول مستطیل وارد کرده‌ایم. ۱۲، به عنوان مساحت چاپ شده است. در متن برنامه، در کنار هر خط، توضیح مربوط به آن نوشته شده است. توجه کنید که اگر ویرایشگر آی‌دی‌ای شما، حروف فارسی را نمی‌پذیرد، باید توضیح‌های فارسی را حذف کنید.

مثال: برنامه‌ای می‌نویسیم که یک عدد صحیح را بگیرد و باقی‌ماندهٔ تقسیم آن بر ۲ را چاپ کند.

```
// Program ID: 0120
#include <iostream>
using namespace std;

int main()
{
    int a;
    cin >> a;
    cout << a % 2;
    cin.ignore(1);
}
```

خروجی:

```
5
1
```

عدد ۵ را وارد کرده‌ایم و باقی‌ماندهٔ تقسیم ۵ بر ۲ چاپ شده است.

عملگر «%» که عملگر (**تقسیم**) **پیمانه‌ای**[۷۴] نام دارد، با فرمول زیر تعریف می‌شود:

$$a\%b = a - b\left[\frac{a}{b}\right]$$

که در آن، $\left[\frac{a}{b}\right]$، قسمت صحیح از عدد اعشاری $\frac{a}{b}$ است؛ توجه کنید که $\left[\frac{a}{b}\right]$، جزءصحیح $\frac{a}{b}$ با تعریف ریاضی «جزءصحیح» نیست، بلکه تنها بخش غیر اعشاری است؛ برای مثال اگر $\frac{a}{b} = -۱/۱۲$ آنگاه $\left[\frac{a}{b}\right] = -۱$. برای خواننده‌ای که با ریاضیات آشنایی بیشتری دارد، می‌گوییم که فرمول بالا را، می‌توان به صورت زیر هم نوشت:

$$a\%b = a - b\,\mathrm{sgn}\left(\frac{a}{b}\right)\left|\left[\frac{a}{b}\right]\right|$$

که در آن، sgn، تابع علامت[۷۵]، $|.|$ تابع قدرمطلق[۷۶] و $[.]$ تابع جزءصحیح[۷۷] است. در C++، عملگر تقسیم پیمانه‌ای را نمی‌توان برای اعداد اعشاری به کار برد (اما در C# می‌توان آن را برای اعداد اعشاری نیز به کار برد).

سوال

۱) آیا اگر عملگر تقسیم پیمانه‌ای نبود، چیزی از قدرت برنامه‌نویسی ما کم می‌شد؟

۲) آیا تفاوتی بین مقدار `3 / 4` و `3 / 4.0` هست؟

تمرین

۱) برنامه‌ای بنویسید که شعاع یک دایره را بگیرد و محیط و مساحت آن را چاپ کند.

---

[۷۴] modulus (division) operator

[۷۵] sign function

[۷۶] absolute value

[۷۷] floor

راهنمایی: ۳/۱۴۱۵۹۲ و $2\pi r$ و πr^2.

۲) فرض کنید n تخم‌مرغ را می‌خواهیم در بسته‌های mتایی بگذاریم. برنامه‌ای بنویسید که n و m را بگیرد و تعداد بسته‌ها را چاپ کند.

راهنمایی: از عملگر تقسیم پیمانه‌ای استفاده کنید.

عبارت‌های بولی

در ریاضیات، گزاره، عبارتی است که یا درست است یا نادرست. برای مثال، «هر عدد زوج، حاصل جمع دو عدد اول است»، یک گزاره است. همچنین $x > 0$ یک گزاره است. چنان‌که پیش‌تر گفته شد، در برنامه‌نویسی، به جای «گزاره» می‌گوییم «عبارت بولی».

دستور if

فرض کنید می‌خواهیم برنامه‌ای بنویسیم که یک عدد صحیح را از کاربر (یعنی از پنجرهٔ کنسول) دریافت کند و اگر این عدد، مثبت بود، پیام `"It is positive"` را چاپ کند. برای این کار، باید از دستور `if` استفاده کنیم و بنویسیم:

```
if (a > 0)
    cout << "It is positive";
```

`if (a > 0)` یعنی « اگر a از 0 بزرگ‌تر باشد». اگر این طور باشد، دستور پس از `if`، یعنی:

```
cout << "It is positive";
```

اجرا می‌شود؛ وگرنه، اجرا نمی‌شود. توجه کنید که عبارت بولی پس از `if`، حتماً باید درون پرانتز باشد. برنامهٔ مورد نظر به این صورت است:

```
// Program ID: 0130
#include <iostream>
using namespace std;

int main()
{
    int a;
    cin >> a;
    if (a > 0)
        cout << "It is positive";
    cin.ignore(2);
}
```

خروجی:

```
4
It is positive
```

عدد ۴ وارد شده است. چون ۴ از صفر بیشتر است، خط:

```
cout << "It is positive";
```

اجرا شده و `It is positive` را چاپ کرده است.

`a > 0`، یک عبارت بولی است، زیرا با توجه به مقدار a، یا درست است یا نادرست. `if`، با توجه به درست یا نادرست‌بودن عبارت بولی `a > 0`، تصمیم می‌گیرد که دستور پس از خود را، اجرا کند یا اجرا نکند.

تمرین

۱) برنامه‌ای بنویسید که:
   ۱) عدد اعشاری x را از کاربر بگیرد.
   ۲) از بین x و x^2، هر کدام بیشتر بود، آن را چاپ کند.
۲) برنامه‌ای بنویسید که:
   ۱) یک عدد صحیح را دریافت کند.
   ۲) اگر یک رقمی بود پیام `"One digit"`،
   ۳) و اگر بیش از یک رقم داشت، پیام `"More than one digit"` را چاپ کند.
۳) برنامه‌ای بنویسید که یک عدد صحیح را بگیرد و قدرمطلق آن را چاپ کند.

راهنمایی: قدر مطلق یک عدد، همان عدد، بدون علامت منفی است. برای مثال، قدر مطلق ۲-، برابر با ۲ و قدر مطلق ۳، برابر با خود ۳ است.

عملگر تساوی

می‌خواهیم برنامه‌ای بنویسیم که عددی را از پنجرهٔ خروجی بگیرد و اگر این عدد، برابر با ۲ یا ۳ باشد، پیام مناسبی را چاپ کند. برای این کار، به **عملگر تساوی**[۷۸] نیاز داریم. عملگر تساوی، «==» است که از دو علامت مساوی پشت‌سرهم تشکیل شده است. می‌توان گفت این عملگر، هیچ ربطی به **عملگر انتساب**[۷۹] یعنی «=»، که برای مقداردادن به متغیرها استفاده می‌شود، ندارد. برنامهٔ یادشده، به این صورت است:

```
// Program ID: 0140
#include <iostream>
using namespace std;

int main()
{
    int a;
    cin >> a;
    if (a == 2)
        cout << "It is 2";
    if (a == 3)
        cout << "It is 3";
    cin.ignore(2);
}
```

[۷۸] equality operator

[۷۹] assignment operator

خروجی:

```
3
It is 3
```

عدد ۳ را وارد کرده‌ایم. در:

```
if (a == 2)
    cout << "It is 2";
```

چون مقدار a، برابر با ۲ نیست، دستور:

```
cout << "It is 2";
```

اجرا نشده است. در:

```
if (a == 3)
    cout << "It is 3";
```

چون مقدار a، برابر با ۳ است، دستور:

```
cout << "It is 3";
```

اجرا شده و `It is 3` را چاپ کرده است.

سوال

۱) چرا مثل ریاضیات یا برخی زبان‌های برنامه‌نویسی، عملگر «=» همزمان، هم برای انتساب و هم برای تساوی به کار نمی‌رود؟

۲) چرا برای عملگر «-»، که ماهیت دوگانهٔ تفریق و قرینه‌کردن دارد، دو علامت متفاوت تعریف نشده است؟ همان‌گونه که برای تساوی و انتساب، که دو ماهیت متفاوتند، دو علامت در نظر گرفته شده است؟

عملگرهای ≤ و ≥

در C++،برای گزارهٔ $x < 2$، می‌نویسیم `x < 2`. اما برای $x \leq 2$ چه باید بنویسیم؟ برای آن، از عملگر «<=» استفاده می‌کنیم.

مثال: برنامه‌ای می‌نویسیم که عددهای بزرگ‌تر از ۲ یا مساوی با آن را تشخیص دهد:

```
// Program ID: 0150
#include <iostream>
using namespace std;

int main()
{
    int a;
    cin >> a;
    if (2 <= a)
        cout << "It is greater than or equal to 2";
    if (a < 2)
```

```
        cout << "It is less than 2";
    cin.ignore(2);
}
```

خروجی:

```
2
it is geater than or equal to 2
```

عدد ۲ را به عنوان ورودی وارد کرده‌ایم. از آنجا که $۲ \leq ۲$، یک گزارهٔ درست است، خط:

```
cout << "It is greater than or equal to 2";
```

اجرا شده است.

برای $x \geq ۲$ نیز می‌نویسیم x =< 2. توجه کنید که در «=<» و «=>»، «=» در طرف راست قرار دارد.

سوال

۱) کدام گزینه درست است؟

- $۳ \leq ۴$
- $۳ \leq ۳ - ۱$
- $۲ < ۲$
- $۲ > ۲$

۲) آیا دستور:

```
if (1 <= 2)
    cout << "t.me/cplusplus";
```

با:

```
cout << "t.me/cplusplus";
```

تفاوتی دارد؟

تمرین

۱) بدون استفاده از عملگر تساوی، برنامه‌ای بنویسید که دو عدد را بگیرد و اگر مساوی بودند، پیامی را چاپ کند.

راهنمایی: دو راه حل دارید:

۱) استفاده از نامساوی $(x - y)^۲ \leq ۰$.

۲) استفاده از دو if تو در تو مانند این:

```
if (x <= y)
    if (x >= y)
        cout << "t.me/cplusplus";
```

۲) برنامه‌ای بنویسید که با گرفتن ضرایب، تعیین کند که معادلهٔ $ax^۲ + bx + c = ۰$ ریشه دارد یا نه.

راهنمایی: اگر $\Delta = b^۲ - ۴ac$ منفی نباشد، معادله، ریشه دارد و در غیر این صورت، ریشه ندارد.

ترکیب گزاره‌ها

می‌خواهیم برنامه‌ای بنویسیم که دو عدد را بگیرد و اگر **هر دو** از ۱۰ بیشتر بودند، پیامی را چاپ کند. برای این کار، به عملگر «وَ» یعنی «&&» نیاز داریم. دستور:

```
if ((a > 10) && (b > 10))
    cout << "hello";
```

یعنی «اگر a از ۱۰ بیشتر باشد **و** b نیز از ۱۰ بیشتر باشد، "hello" را چاپ کن». برنامهٔ زیر، برنامه‌ای است که می‌خواستیم بنویسیم:

```
// Program ID: 0160
#include <iostream>
using namespace std;

int main()
{
    int a, b;
    cin >> a;
    cin >> b;
    if ((a > 10) && (b > 10))
        cout << "hello";
    cin.ignore(2);
}
```

خروجی:

```
11
19
hello
```

اعداد ۱۱ و ۱۹ را وارد کرده‌ایم که چون هر دو از ۱۰ بیشترند، رشتهٔ "hello" چاپ شده است.

اگر بخواهیم برنامه‌ای بنویسیم که دو عدد را بگیرد و اگر **حداقل یکی** از آن‌ها از ۱۰ بیشتر بود، پیامی را چاپ کند؛ از عملگر «**یا**» یعنی «||» استفاده می‌کنیم[۸۰]. دستور:

```
if ((a > 10) || (b > 10))
    cout << "hello";
```

یعنی «اگر حداقل یکی از a یا b از ۱۰ بیشتر باشد، "hello" را چاپ کن». برنامهٔ مورد نظر، به این صورت است:

```
// Program ID: 0170
#include <iostream>
using namespace std;
```

[۸۰] برای تایپ کردن || باید کلید Shift را بگیرید و دو بار کلید مناسب از صفحه‌کلید را فشار بدهید. این کلید مناسب، معمولاً در اطراف کلید Enter قرار دارد.

```
int main()
{
    int a, b;
    cin >> a;
    cin >> b;
    if ((a > 10) || (b > 10))
        cout << "hello";
    cin.ignore(2);
}
```

خروجی:

```
15
6
hello
```

عددهای ۱۵ و ۶ را وارد کرده‌ایم، چون حداقل یکی از آن‌ها، از ۱۰ بیشتر است، رشتهٔ "hello" چاپ شده است.

مثال: برنامه‌ای می‌نویسیم که عدد صحیحی را دریافت کند و اگر این عدد ۱ یا ۲ یا ۳ بود، یک پیام و اگر این گونه نبود، پیام دیگری را چاپ کند.

```
// Program ID: 0180
#include <iostream>
using namespace std;

int main()
{
    int a;
    cin >> a;
    if ((a == 1) || (a == 2) || (a == 3)) // اگر ۱ یا ۲ یا ۳ بود
        cout << "1 or 2 or 3";
    if ((a < 1) || (3 < a))               // اگر کمتر از ۱ یا بیشتر از ۳ بود
        cout << "Not 1 nor 2 nor 3";
    cin.ignore(2);
}
```

خروجی:

```
2
1 or 2 or 3
```

2 را وارد کرده‌ایم؛ مقدار a، ۲ شده است و چون **حداقل یکی** از عبارت‌های بولی a == 1، a == 2 یا a == 3 درست درآمده، پیام 1 or 2 or 3 چاپ شده است.

از آنجا که هیچ‌یک از a < 1 و 3 < a درست در نیامده، رشتهٔ "Not 1 nor 2 nor 3" چاپ نشده است. در خط:

```
if ((a == 1) || (a == 2) || (a == 3))
```

و نیز خط:

```
if ((a < 1) || (3 < a))
```

پرانتزهای درونی ضروری نیستند و می‌توان آن‌ها را حذف کرد (چون اولویت عملگر || کم‌تر از «>» است)؛ مثلاً در مورد اول می‌توان نوشت:

```
if (a == 1 || a == 2 || a == 3)
```

پرانتز پس از if، هرگز قابل‌حذف نیست.

سوال

۱) اگر عملگر => در C++ نبود، چگونه می‌شد با > و == آن را ساخت؟

۲) اگر عملگر == در C++ نبود، چگونه می‌شد با => آن را ساخت؟

۳) آیا دستور:

```
if (a > 2)
    if (a < 10)
        cout << "hello";
```

با دستور:

```
if ((a > 2) && (a < 10))
    cout << "hello";
```

معادل است؟ یعنی آیا در عملکرد با هم تفاوتی دارند؟ و همچنین آیا از نظر مدت‌زمان اجرا، با هم متفاوتند؟

۴) اگر a > 20 درست باشد، آیا کامپایلر در بررسی دستور:

```
if ((a > 20) || (a < 10))
    cout << "hello";
```

به بررسی a < 10 نیاز دارد؟

تمرین

۱) برنامه‌ای بنویسید که سه عدد را بگیرد و بزرگ‌ترین آن‌ها را چاپ کند.

عملگر نقیض

در بخش‌های قبلی گفتیم که چگونه می‌توان دستوری با الگوی «اگر این طور باشد، آن کار را بکن» نوشت. در این بخش، قرار است بگوییم چگونه می‌شود دستوری نوشت با الگوی «اگر این طور **نباشد**، آن کار را بکن». برای نوشتن چنین دستوری، از عملگر نقیض[۸۱]، یعنی «!»، استفاده می‌کنیم[۸۲]. به عنوان نمونه، در صورتی که بخواهیم بنویسیم «اگر a از ۱۰ کم‌تر **نبود**، "hello" را چاپ کن» می‌نویسیم:

```
if (!(a < 10))
```

[۸۱] negation operator

[۸۲] در ریاضیات، برای عملگر نقیض از «¬» یا «~» استفاده می‌شود اما در C++، از علامت تعجب استفاده می‌شود.

```
    cout << "hello";
```

مثال: برنامه‌ای که عددی را می‌گیرد و اگر مثبت باشد، "positive" و در غیر این صورت، "negative" را چاپ می‌کند:

```
// Program ID: 0190
#include <iostream>
using namespace std;

int main()
{
    int a;
    cin >> a;
    if (a > 0)
        cout << "positive";
    if (!(a > 0))
        cout << "negative";
    cin.ignore(2);
}
```

خروجی:

```
-9
negative
```

عدد -9 را وارد کرده‌ایم و چون a > 0 درست نبوده، negative چاپ شده است.

سوال

۱) در ریاضیات می‌گویند گزارهٔ «اگر p، آنگاه q» معادل است با «اگر نقیض q، آنگاه نقیض p». به عنوان مثال، «اگر باران ببارد، زمین تر می‌شود» معادل است با این گزاره که «اگر زمین تر نباشد، باران نباریده است». آیا در مورد دستور if در C++ نیز این قاعده درست یا قابل‌استفاده است؟

دستور else

«اگر x مثبت بود خود x را چاپ کن **وگرنه** x- را چاپ کن». چنین دستوری را چگونه می‌توان نوشت؟ برای این کار، باید پس از دستور if، از دستور else استفاده کنیم که به معنی «وگرنه» است. برنامهٔ مثال قبل را می‌شود با استفاده از دستور else، راحت‌تر نوشت:

```
// Program ID: 0200
#include <iostream>
using namespace std;

int main()
{
    int a;
    cin >> a;
    if (a > 0)
        cout << "positive";
    else
        cout << "negative";
    cin.ignore(2);
}
```

خروجی:

```
8
positive
```

در برنامهٔ بالا، کد:

```
if (a > 0)
    cout << "positive";
else
    cout << "negative";
```

یعنی «اگر a مثبت بود، "positive" را چاپ کن وگرنه، "negative" را چاپ کن».

دستور else، باید بلافاصله پس از دستور if آورده شود.

سوال

۱) آیا else، نیاز به عملگر نقیض را مرتفع می‌کند؟

نوع bool

چنان‌که پیش‌تر نیز اشاره شده است، می‌توان یک عبارت بولی، مانند a > 0، را در یک متغیر ذخیره کرد که آن متغیر، باید نوع bool داشته باشد. برای مثال، به جای:

```
if (a > 0)
    cout << "positive";
```

می‌توانیم بنویسیم:

```
bool b = (a > 0);
if (b)
    cout << "positive";
```

در اینجا، b همان a > 0 است، یا به طور دقیق‌تر، b، متغیری با مقداری برابر با مقدار a > 0 است.

مثال: برنامه‌ای که یک عدد را می‌گیرد و اگر سه رقمی بود، پیامی را چاپ می‌کند:

```
// Program ID: 0210
#include <iostream>
using namespace std;

int main()
{
    int a;
    cin >> a;
    bool p1 = (100 <= a);
    bool p2 = (a <= 999);
    if (p1 && p2)
        cout << "OK";
    else
        cout << "Access Denied";
    cin.ignore(2);
```

```
}
```

خروجی:

```
256
OK
```

عدد 256 را وارد کرده‌ایم، چون p1 و p2 هر دو درست بوده‌اند، OK چاپ شده است.

می‌دانیم که به عبارت‌هایی مانند a > 0، که یا درست یا نادرست هستند، عبارت بولی گفته می‌شود. پس، یک عبارت بولی را می‌توان در یک متغیر با نوع bool، ذخیره کرد. در حقیقت، یک عبارت بولی، می‌تواند دو مقدار داشته باشد: true (درست) یا false (غلط). برای نمونه، مقدار 1 < 2، true است، زیرا درست است و مقدار 4 < 3، false است، چون نادرست است.

مثال: برنامهٔ زیر ثابت می‌کند که مقدار 1 < 2، true است:

```
// Program ID: 0220
#include <iostream>
using namespace std;

int main()
{
    if ((1 < 2) == true)
        cout << "proved";
    cin.ignore(1);
}
```

خروجی:

```
proved
```

چون رشتهٔ "proved" چاپ شده است؛ می‌توانیم نتیجه بگیریم که 1 < 2 و true مساوی هستند. توجه کنید که عبارت (1 < 2) == true، خود، یک عبارت بولی است. از این رو، می‌توانیم بنویسیم:

```
bool b = ((1 < 2) == true);
```

که شاید کمی گیج‌کننده باشد.

تمرین

۱) آیا گزاره‌های زیر مساوی هستند؟:

a) `1 < 2`
b) `(1 < 2) == true`
c) `((1 < 2) == true) == true`
d) `(((1 < 2) == true) == true) == true`

۲) فرض کنید b، متغیری با نوع bool است. آیا دو گزارهٔ زیر، همواره معادل (یعنی مساوی) هستند؟:

a) `p`
b) `p == true`

۳) فرض کنید b، متغیری با نوع bool است. آیا دو گزارهٔ زیر، همواره معادل هستند؟:

a) `!p`
b) `p == false`

۴) آیا عبارت‌های بولی `true` و `false` نقیض یکدیگرند؟

ترکیب دستورها

بلوک‌ها

بلوک‌ها

به هریک از علامت‌های «{}» و «{»، آکلاد[۸۳] می‌گویند. با استفاده از آکلادها، می‌توان چندین دستور را بسته‌بندی کرد تا همگی، **یک** دستور به حساب آیند؛ نه چند دستور مجزا. اما کاربردش چیست؟ فرض کنید می‌خواهیم برنامه‌ای بنویسیم که یک عدد را بگیرد و اگر مثبت بود، چندین دستور را انجام دهد؛ اما پس از `if`، تنها یک دستور، متعلق به `if` به حساب می‌آید؛ برای مثال در:

```
if (a > 0)
    cout << "hello";
cout << "bye";
```

تنها `cout << "hello";` مربوط به `if` است. حتی اگر بنویسیم:

```
if (a > 0)
    cout << "hello";
    cout << "bye";
```

باز هم تنها `cout << "hello";` مربوط به `if` است. برای این که هر دو دستور به `if` مربوط گردند، آن‌ها را، با استفاده از آکلاد، ترکیب می‌کنیم:

```
if (a > 0)
{
    cout << "hello";
    cout << "bye";
}
```

در اینجا:

```
{
    cout << "hello";
    cout << "bye";
}
```

تنها یک دستور به حساب می‌آید؛ و همین یک دستور است که پس از `if` قرار گرفته است. حال، اگر شرط `if` درست باشد، هم `cout << "hello";` و هم `cout << "bye";` اجرا می‌شوند؛ وگرنه هیچ‌کدام اجرا نمی‌شوند.

---

[۸۳] brace

مثال: برنامه‌ای می‌نویسیم که عددی را بگیرد، اگر مثبت بود یک عدد دیگر را بگیرد و مجموع دو عدد را چاپ کند. ولی اگر عدد اول مثبت نبود، پیام "Access is denied: " را همراه با خود عدد چاپ کند.

```
// Program ID: 0230
#include <iostream>
using namespace std;

int main()
{
    int a;
    cin >> a;
    if (a > 0)
    {
        int b;
        cin >> b;
        cout << a + b;
    }
    else
    {
        cout << "Access is denied: ";
        cout << a;
    }
    cin.ignore(2);
}
```

خروجی:

```
10
13
23
```

عدد ۱۰ را وارد کرده‌ایم، چون مثبت بوده است، خطوط:

```
{
    int b;
    cin >> b;
    cout << a + b;
}
```

اجرا شده‌اند. یعنی ابتدا، متغیر b تعریف شده است، و سپس، مقدار آن، از پنجرهٔ خروجی گرفته شده است (به آن، مقدار ۱۳ را داده‌ایم)؛ در نهایت، مجموع a و b چاپ شده است. می‌بینید که در مورد else نیز، می‌توان از آکلاد استفاده کرد.

به دستورهایی که با آکلاد ترکیب شده‌اند، **بلوک**[۸۴] می‌گوییم. بلوک‌ها، تنها مربوط به if، else و مثل این‌ها نیستند. به عنوان مثال، در برنامهٔ زیر، چند دستور، با آکلاد ترکیب شده‌اند، بدون این که if یا else‌ی در کار باشد:

```
// Program ID: 0240
#include <iostream>
using namespace std;

int main()
{
```

[۸۴] block

```
    {
        int a = 5;
        cout << "hello ";
        cout << a;
    }
    cin.ignore(1);
}
```

خروجی:

```
hello 5
```

در برنامهٔ بالا، خطوط:

```
{
    int a = 5;
    cout << "hello ";
    cout << a;
}
```

یک بلوک تشکیل می‌دهند. البته، در اینجا، نیازی به ساختن بلوک نبود؛ اما ساختن آن هم مشکلی ایجاد نمی‌کند.

مثال: در برنامهٔ زیر، نشان می‌دهیم که در درون یک بلوک، می‌توان بلوک‌های دیگری ساخت:

```
// Program ID: 0250
#include <iostream>
using namespace std;

int main()
{
    {                               // شروع یک بلوک
        {                           // شروع یک بلوک
            cout << "hello ";
        }                           // آخر بلوک

        {                           // شروع یک بلوک
            if (true)
            {                       // شروع یک بلوک
                cout << "if";
            }                       // آخر بلوک
        }                           // آخر بلوک
    }                               // آخر بلوک
    cin.ignore(1);
}
```

خروجی:

```
hello if
```

در برنامهٔ فوق، خطوط:

```
{
    {
        cout << "hello ";
    }

    {
        if (true)
        {
            cout << "if";
        }
```

```
    }
}
```

یک بلوک می‌سازند که شامل دو بلوک دیگر است؛ یعنی بلوک:

```
{
    cout << "hello ";
}
```

که یک بلوک است با فقط یک دستور و بلوک:

```
{
    if (true)
    {
        cout << "if";
    }
}
```

و در این بلوک:

```
if (true)
{
    cout << "if";
}
```

از نظر نتیجهٔ اجرا، با:

```
cout << "if";
```

تفاوتی ندارد؛ زیرا true یک گزارهٔ همیشه‌درست است، و بنابراین، دستور پس از if، همیشه اجرا می‌شود.

دسترسی به متغیرها در بلوک‌ها

وقتی یک متغیر، در درون یک بلوک تعریف می‌شود، با خروج از آن، از بین می‌رود؛ یعنی دیگر قابل‌استفاده نیست. برای مثال، کد:

```
{
    int a = 5;
}
cout << a;
```

خطا دارد. علت خطا این است که a، تنها در درون بلوک، قابل‌استفاده است و با خروج از آن، از بین می‌رود. می‌توان نام متغیری را که قبل از یک بلوک به کار رفته است، در درون آن، برای متغیر دیگری به کار برد. در این صورت، متغیر بیرونی، که پیش‌تر تعریف شده است، موقتاً، **محو** می‌گردد.

مثال: در برنامهٔ زیر، یک متغیر، در یک بلوک درونی‌تر، محو می‌شود:

```
// Program ID: 0260
#include <iostream>
```

```
using namespace std;

int main()
{
    int a = 1;
    {
        int a = 5;
        cout << a;
    }
    cout << "   ";
    cout << a;

    cin.ignore(1);
}
```

خروجی:

```
5   1
```

متغیر aی قبل از بلوک، با مقدار ۱ و aی درون بلوک، با مقدار ۵، تعریف شده است. با:

```
cout << a;
```

مقدار aی درون بلوک و مقدار aی بیرون بلوک، چاپ شده‌اند. می‌بینید که مقدار aی درون بلوک با مقدار aی بیرون از آن تفاوت دارد. علت این است که وقتی درون بلوک هستیم، با تعریف a جدید، aی قبلی، موقتاً محو می‌گردد، انگار تعریف نشده باشد؛ و aی جدید، جایگزین آن می‌شود. اما طول عمر این aی جدید، تنها تا پایان بلوک است و با خروج از بلوک، aی قبلی دوباره ظاهر می‌شود. در برنامهٔ فوق، خط:

```
cout << "   ";
```

تنها برای ایجاد فاصلهٔ بین دو عدد در خروجی، گذاشته شده است.

اگر در مثال بالا، در درون بلوک، متغیری با نام a تعریف نمی‌شد، متغیر قبلی نیز محو نمی‌شد:

```
// Program ID: 0270
#include <iostream>
using namespace std;

int main()
{
    int a = 1;
    {
        cout << a;
    }
    cout << "   ";
    cout << a;
    cin.ignore(1);
}
```

خروجی:

```
1   1
```

این برنامه، همان برنامهٔ قبلی است با این فرق که دیگر درون بلوک، aی جدیدی تعریف نشده است و به همین سبب، a، در درون و بیرون بلوک یکسان است.

یک نکتهٔ مهم این است که متغیر، از محل تعریف به بعد، محو می‌شود، مثلاً در:

```
int a = 1;
{
    cout << a;
    int a = 5;
    cout << a;
}
```

cout << اول، مقدار ۱ و cout << دوم، مقدار ۵ را چاپ می‌کند.

دستورهای ساده

دستورهای ساده، دستورهایی مثل:

```
a = b + c
```

یا:

```
cout << "hello"
```

هستند؛ اما بلوک‌ها و دستورهای تعریف متغیر، دستورهای ساده محسوب نمی‌شوند؛ برای مثال:

```
int a = 2
```

یک دستور ساده نیست؛ زیرا در آن یک متغیر تعریف می‌شود.

ویژگی مهمی که به وسیلهٔ آن، دستورهای ساده مشخص می‌شوند، این است که می‌توان دور آن‌ها پرانتز گذاشت؛ بدون این که خطایی در برنامه پیش بیاید. برای نمونه، به جای:

```
cout << "hello";
```

می‌شود نوشت:

```
(cout << "hello");
```

دور دستورهای غیرساده نمی‌توان پرانتز گذاشت؛ مثلاً کد:

```
({
    cout << "hello";
})
```

خطا دارد. همین‌طور:

```
(int a);
```

خطا دارد.

ترکیب دستورهای ساده با کاما

می‌دانید که به جای:

```
int a = 1;
int b = 2;
```

می‌توانیم بنویسیم:

```
int a = 1, b = 2;
```

این ترکیب تعریف‌ها، با استفاده از علامت «,» است، که این علامت، کاما[۸۵] نامیده می‌شود. دستورهای ساده را نیز می‌توان با استفاده از کاما ترکیب کرد. برای مثال، به جای:

```
cout << "hello ";
cout << "bye";
a = 2;
```

می‌توان نوشت:

```
cout << "hello ", cout << "bye", a = 2;
```

توجه کنید که تنها دستورهای ساده را می‌توان به این شکل ترکیب کرد؛ مثلاً نوشتن:

```
{cout << "hello";}, cout << "bye";
```

مسبب خطا است؛ زیرا {cout << "hello";} یک بلوک است و دستور ساده محسوب نمی‌گردد. اگر دستورهای ساده را با کاما ترکیب کنیم، حاصل نیز، یک دستور ساده است. برای نمونه:

```
cout << "hello ", cout << "bye", a = 2
```

خود، یک دستور ساده است. پس می‌توانیم دور آن پرانتز بگذاریم:

```
(cout << "hello ", cout << "bye", a = 2);
```

سوال

۱) آیا برنامهٔ زیر خطا دارد؟ اگر دارد چگونه باید آن را رفع کرد و اگر ندارد خروجی برنامه چیست؟:

```
// Program ID: 0280
#include <iostream>
using namespace std;

int main()
{
    int a = 2;
    if (a == 1)
        cout << "1",
        cout << "2";
    else
        cout << "3",
        cout << "4";
```

[۸۵] comma

```
    cin.ignore(1);
}
```

ترکیب >> coutها و << cinها

برای دستورات ورودی و خروجی، شکل جدیدی از ادغام ممکن است؛ یعنی این امکان وجود دارد که دو یا چند cout << پشت‌سرهم یا چند دستور cin >> پشت‌سرهم را، به شکل خاصی، با یکدیگر ادغام کنیم. این شکل خاص، بدین گونه است که، به جای:

```
cout << a;
cout << "hello";
cout << b;
```

می‌نویسیم:

```
cout << a << "hello" << b;
```

و به جای:

```
cin >> a;
cin >> b;
cin >> c;
```

می‌نویسیم:

```
cin >> a >> b >> c;
```

مثال: برنامه‌ای می‌نویسیم که دو عدد صحیح را بگیرد و مجموع و حاصل‌ضرب آن‌ها را چاپ کند.

```
// Program ID: 0290
#include <iostream>
using namespace std;

int main()
{
    int a, b;
    cin >> a >> b;
    cout << a + b << "      " << a * b;
    cin.ignore(2);
}
```

خروجی:

```
2
5
7      10
```

در اینجا، با cin >>، به دو متغیر، مقدار داده می‌شود. cout <<، حاصل‌جمع و حاصل‌ضرب را، همراه با یک فاصله در بینشان، چاپ می‌کند. به عنوان ورودی، اعداد ۲ و ۵ را به برنامه داده‌ایم.

ترکیب انتساب‌ها

به برنامهٔ زیر توجه کنید:

```
// Program ID: 0300
#include <iostream>
using namespace std;

int main()
{
    int a, b;
    a = b = 2018;
    cout << a << "  " << b;
    cin.ignore(1);
}
```

خروجی:

```
2018  2018
```

در این برنامه، خط:

```
a = b = 2018;
```

مقدارهای a و b را همزمان ۲۰۱۸ می‌کند. این شکل از به‌کاربردن تساوی‌ها را می‌توانیم نوعی ترکیب برای تساوی‌ها به حساب بیاوریم. اما واقعیت، آن است که در اینجا، چیز جدیدی وجود ندارد که آن را ترکیب بنامیم. شاید، این فصل جای مناسبی برای تحلیل این گونه از کدها نباشد؛ اما از آنجا که، در آینده، تحلیل‌های بیشتری مورد نیاز خواهند بود، خوب است کمی با چنین تحلیل‌هایی آشنا شویم: همان‌طور که b + 2018 یک مقدار دارد که حاصل‌جمع b و 2018 است، b = 2018 نیز یک مقدار دارد و این مقدار، 2018 است. پس می‌توانیم بنویسیم:

```
a = (b = 2018);
```

و می‌توان پرانتزها را هم برداشت و نوشت:

```
a = b = 2018;
```

که این کار، تفاوتی ایجاد نمی‌کند. اما محل قرارگرفتن پرانتز مهم است: a = (b = 2018); (و معادل بدون پرانتزش) با:

```
(a = b) = 2018;
```

تفاوت دارد. به خروجی برنامهٔ زیر نگاه کنید:

```
// Program ID: 0310
#include <iostream>
using namespace std;

int main()
{
    int a, b = 1397;
```

```
    (a = b) = 2018;
    cout << a << "  " << b;
    cin.ignore(1);
}
```

خروجی:

```
2018  1397
```

خروجی این برنامه را، با خروجی برنامهٔ پیشین، مقایسه کنید. می‌بینید که گذاشتن پرانتز حول تساوی اول، خروجی را تغییر داده است. می‌توان گفت: دستور:

```
a = b = 2018;
```

معادل است با قطعه‌کد:

```
b = 2018;
a = b;
```

و دستور:

```
(a = b) = 2018;
```

معادل است با:

```
a = b;
a = 2018;
```

تمرین

۱) پس از اجرای:

```
int a = 10;
(a = 11)++;
```

مقدار a چه قدر است؟

تابع‌ها

تابع در ریاضی و در C++

حتماً در ریاضی دورهٔ دبیرستان با مفهوم تابع آشنا شده‌اید. به طور ساده، تابع، یعنی فرمول[۸۶]. تابع:

$$f(x) = x^2 - x + 1$$

را در نظر بگیرید. برای محاسبهٔ f(۲)، کافی است در فرمول بالا، به جای x، بگذاریم ۲:

[۸۶] به تعریف دقیق ریاضی تابع کاری نداریم.

$$f(۲) = ۲^{۲} - ۲ + ۱ = ۴ - ۲ + ۱ = ۳$$

پس f(۲) برابر با ۳ است. به ۲، آرگومان[۸۷] می‌گوییم. در C++ نیز، می‌توان تابع‌هایی مانند این تابع داشت. تابع بالا، در C++ به این شکل نوشته می‌شود:

```
double f(double x)
{
    return x * x - x + 1;
}
```

با اضافه‌کردن این تابع به برنامه، می‌بینیم که همانند محاسبات فوق، f(2) می‌شود ۳:

```
// Program ID: 0320
#include <iostream>
using namespace std;

double f(double x)
{
    return x * x - x + 1;
}
int main()
{
    cout << f(2);
    cin.ignore(1);
}
```

خروجی:

```
3
```

این برنامه f(2) را چاپ کرده است. به مقدار f(2)، مقدار تابع در نقطهٔ 2 یا **خروجی** گفته می‌شود. می‌توان مقدار تابع را، در هر نقطهٔ دیگر نیز به همین شکل به دست آورد.

مثال: برنامه‌ای می‌نویسیم که یک عدد را بگیرد و توان دومش را چاپ کند.

```
// Program ID: 0330
#include <iostream>
using namespace std;

double square(double x)
{
    return x * x;
}
int main()
{
    double a;
    cin >> a;
    cout << square(a);
    cin.ignore(2);
}
```

خروجی:

```
3.16
9.9856
```

[۸۷] argument

از خروجی برنامهٔ بالا نتیجه می‌گیریم که:

$$۳/۱۶^{۲} = ۹/۹۸۵۶$$

square، تابعی است که خروجی آن، توان دوم آرگومانش است؛ یعنی، مقدار square(x) برابر با x * x است. معادل ریاضی تابع square به این شکل است:

$$\text{square}(x) = x^{۲}$$

حال، دربارهٔ تابع:

```
double square(double x)
{
    return x * x;
}
```

و بخش‌های مختلف آن توضیح می‌دهیم. اولین double، **نوع** خروجی تابع را مشخص می‌کند. خروجی تابع، ربطی به خروجی برنامه ندارد؛ خروجی، در اینجا، یعنی مقداری که تابع در هر نقطه دارد؛ به طور ساده‌تر، یعنی f(1.1)، f(2)، f(-1) و مانند این‌ها؛ همهٔ این‌ها، مقدارهایی با نوع double هستند؛ یعنی عدد اعشارند.

دومین double می‌گوید که ورودی تابع، که آرگومان نامیده، می‌شود، باید نوع double داشته باشد. پس می‌توانیم بنویسیم f(2.32) ولی نمی‌توانیم بنویسیم f("hello")؛ زیرا "hello" یک رشته است نه یک عدد اعشاری. کلمهٔ return، مقداری را که f(x) (به ازای x داده شده) باید داشته باشد، معین می‌کند. از آنجا که return، به معنی «برگرداندن» است، در مورد تابع بالا می‌گوییم تابع square، مقدار x * x را **برمی‌گرداند** (یا آن را return می‌کند).

ورودی و خروجی یک تابع، ممکن است هر چیزی باشند. به علاوه، یک تابع؛ ممکن است دو یا چند ورودی داشته باشد.

مثال: تابعی می‌نویسیم که دو عدد صحیح را می‌گیرد و باقی‌ماندهٔ تقسیم اولی بر دومی را برمی‌گرداند. این تابع را در برنامهٔ زیر می‌بینید:

```
// Program ID: 0340
#include <iostream>
using namespace std;

int residue(int a, int b)
{
    return a % b;
}
int main()
{
    cout << residue(15, 4);
    cin.ignore(1);
}
```

خروجی:

```
3
```

می‌دانید که a % b باقی‌ماندهٔ تقسیم a بر b است[۸۸]. نکات جدید در این برنامه، یکی این است که تابع residue دو ورودی دارد، و دیگر این که نوع ورودی‌ها و خروجی int است. در ریاضیات نیز تابع‌های دو یا چند متغیره داریم.

در مثال بالا، تابع residue، در حقیقت، همان عملگر % است، و تنها، این عملگر بازنویسی شده است. مثال بعدی، عملگر «||» را به شکل تابع در می‌آورد. در آینده، خواهید دید که خود عملگرها نیز نوعی تابع هستند.

مثال: تابعی می‌نویسیم که بتوان از آن، به جای عملگر «||»، استفاده کرد.

```
// Program ID: 0350
#include <iostream>
using namespace std;

bool Or(bool p, bool q)
{
    return p || q;
}
int main()
{
    bool p = (1 < 2);
    bool q = (2 < 1);
    if (Or(p, q))
        cout << "Or";
    cin.ignore(1);
}
```

خروجی:

```
Or
```

در این برنامه، دستور:

```
if (Or(p, q))
    cout << "Or";
```

با دستور:

```
if (p || q)
    cout << "Or";
```

در عملکرد، فرقی ندارد. p || q در اینجا، یعنی «یا ۱ از ۲ کم‌تر است یا ۲ از ۱»؛ یا به طور دقیق‌تر، یعنی حداقل یکی از دو گزارهٔ p یا q درست است.

مثال بعدی، شاید کمی گیج‌کننده به نظر برسد، مگر آن که با منطق ریاضی آشنایی داشته باشید:

مثال: می‌دانید که عملگر ترکیبی || معنی «حداقل یکی» دارد. حال، می‌خواهیم تابعی مشابه آن بنویسیم که معنی «دقیقاً یکی» داشته باشد.

```
// Program ID: 0360
#include <iostream>
using namespace std;
```

[۸۸] البته اگر a یا b منفی باشند این تعریف منطبق بر تعریف باقی‌مانده در نظریهٔ اعداد نیست.

```
bool Xor(bool p, bool q)
{
    return (p || q) && !(p && q);
}
int main()
{
    bool p = (1 < 2);
    bool q = (1 < 3);
    if (Xor(p, q))
        cout << "Xor";
    else
        cout << "else";
    cin.ignore(1);
}
```

خروجی:

```
else
```

(p || q) && !(p && q) یعنی «حداقل یکی از p و q درست باشد ولی هر دو درست نباشند» و این یعنی «دقیقاً یکی درست باشد». (p || q) یعنی «حداقل یکی از p و q درست باشد» و !(p && q) یعنی «این طور نباشد که هم p و هم q درست باشد». این جور استدلال‌ها به منطق گزاره‌ها مربوط می‌شوند و ممکن است برای فرد ناآشنا مبهم به نظر برسند.

ورودی‌ها و خروجی تابع‌ها می‌توانند از هر نوعی باشند و لازم نیست مانند مثال‌های بالا، یکسان باشند. به تابع زیر نگاه کنید:

```
int f(int y)
{
    return y - 1;
}
```

برای فراخواندن آن، می‌نویسیم:

```
f(1397);
```

در این حالت، مقدار 1397 در y می‌گیرد؛ اما 1397 و y دو ماهیت متفاوتند که تنها مقدار یکسانی دارند. به y **پارامتر** تابع f و به 1397 **آرگومان** تابع f خواهیم گفت. بنابراین، مقدار آرگومان یک تابع، قبل از اجرای تابع، در پارامترهای آن قرار می‌گیرند و سپس، اجرا آغاز می‌گردد. ویژوال‌استودیو پارامترها را با رنگ متمایزی نشان می‌دهد.

تمرین

۱) تابعی بنویسید و در یک برنامه از آن استفاده کنید به گونه‌ای که پارامتر و آرگومان تابع هم‌نام باشند.

تابع و انجام دستورات

تابع‌های C++، بر خلاف توابع ریاضی، می‌توانند علاوه بر برگرداندن مقدار؛ هر کار دیگری نیز انجام دهند. برای نمونه، یک تابع، می‌تواند مقداری را که قرار است برگرداند، چاپ نیز بکند:

```
// Program ID: 0370
#include <iostream>
using namespace std;

double f(double x)
{
    double a = x * x - x + 1;
    cout << a;
    return a;
}
int main()
{
    f(2);
    cin.ignore(1);
}
```

خروجی:

```
3
```

در این برنامه، چند نکتهٔ مهم وجود دارد: اول این که، دستورات دیگری درون تابع قرار داده شده‌اند. دوم این که return، مقدار یک متغیر را به عنوان خروجی برگردانده است؛ در حالی که در برنامه‌های پیشین، مقدار عبارت، مستقیماً برگردانده می‌شد؛ مثلاً نوشته می‌شد:

```
return x * x - x + 1;
```

سوم این که قبل از cin.ignore(1); این خط نوشته شده است:

```
f(2);
```

در حالی که در برنامه‌های قبلی مثلاً می‌نوشتیم:

```
cout << f(2);
```

اینجا هم می‌شد این طور بنویسیم، اما لازم نبود؛ زیرا این مقدار، در خود تابع f چاپ می‌شود. نوشتن f(2) در هریک از دو صورت:

```
f(2);
```

یا:

```
cout << f(2);
```

سبب اجرای دستورات داخل تابع f می‌گردد؛ یعنی چه بنویسیم f(2); و چه بنویسیم cout << f(2);، دستورات تابع حتماً به طور کامل اجرا می‌شوند. از آنجا که دستورات تابع f، اجرا می‌شوند می‌گوییم تابع f **فراخوانده شده است**.


تمرین

۱) تابعی بنویسید که تعداد روزها را بگیرد و تعداد ثانیه‌های موجود در آن تعداد از روزها را چاپ کند و تعداد ساعات موجود در آن روزها را به عنوان خروجی برگرداند.

۲) بر اساس فرمول $a\%b = a - b\,\mathrm{sgn}\left(\frac{a}{b}\right)\left\lfloor\left|\frac{a}{b}\right|\right\rfloor$، تابعی بنویسید که جایگزین عملگر «%» باشد. سپس، یکسان‌بودن عملکرد تابعی را که نوشته‌اید با «%»، با اعداد مثبت و منفی مختلف بیازمایید.

اجرای قدم‌به‌قدم برنامه

می‌توان در آی‌دی‌ای، برنامه‌ها را گام‌به‌گام و آرام اجرا کرد. اجرای آرام، در برطرف‌کردن خطاهای برنامه کمک می‌کند، اما هدف اول ما در این بخش، آشنایی با نحوهٔ فراخوانی تابع‌هاست.

برنامهٔ زیر را در Program.cpp وارد کنید و یک بار به طور طبیعی اجرا کنید تا برای اجرای قدم‌به‌قدم آماده شود:

```
// Program ID: 0380
#include <iostream>
using namespace std;

int main()
{
    cout << 1;
    cout << 2;
    cout << 3;
    cout << 4;
    cin.ignore(1);
}
```

خروجی:

```
1234
```

برای اجرای قدم‌به‌قدم این برنامه، کلید F10 را فشار دهید. با هر بار فشار این کلید، یک خط از برنامه اجرا می‌شود. پس از هر بار فشار کلید، پنجرهٔ خروجی را چک کنید تا تغییرات را ببینید. آنگاه که به خط:

```
cin.ignore(1);
```

رسیدید و خواستید از آن عبور کنید، باید حتماً پنجرهٔ کنسول را (اگر خودبه‌خود فعال نشود) فعال کنید[۸۹] و کلید Enter را فشار دهید تا بتوانید از این خط بگذرید.

اجرای قدم‌به‌قدم به دو صورت قابل‌انجام است:

۱) step over

۲) step into (یا trace into)

step over از روی تابع‌ها می‌پرد و به درون تابع نمی‌رود. اما step into وارد تابع هم می‌شود.

[۸۹] فعال شدن یک پنجره در ویندوز یعنی به رو آمدن آن پنجره.

کلید مربوط step over:

F10

و کلید مربوط به step into:

F11

است.

از طریق منوها نیز می‌شود step over و step into انجام داد. برای دیدن تفاوت بین step over و step into مثال‌های بعدی را ببینید و آن‌ها را آزمایش کنید. در مثال‌های بعد، برنامهٔ زیر را قدم‌به‌قدم اجرا می‌کنیم:

```
// Program ID: 0390
#include <iostream>
using namespace std;

double f(int x)

{
    double a = x * x;
    double b = x;
    double c = 1;
    double ret = a - b + c;
    return ret;
}
int main()
{
    int a = 2;
    f(a);
    cin.ignore(1);
}
```

خروجی: ندارد

برنامهٔ بالا، هیچ خروجی‌ای ندارد؛ زیرا درون آن، جایی از cout << استفاده نشده است. آن را در Program.cpp بنویسید و یک بار به طور عادی اجرا کنید تا برای اجرای قدم‌به‌قدم آماده باشد.

مثال (اجرا به صورت step over): برنامهٔ بالا را به صورت step over، قدم‌به‌قدم، اجرا می‌کنیم. خط درحال‌اجرا، با یک فلش زردرنگ () در کنار خط، مشخص می‌شود؛ اما در زیر با تغییر رنگ زمینهٔ خط، آن را نشان خواهیم داد.

قدم اول:

```
// Program ID: 0390
#include <iostream>
using namespace std;

double f(int x)
{
    double a = x * x;
    double b = x;
```

```
    double c = 1;
    double ret = a - b + c;
    return ret;
}
int main()
{
    int a = 2;
    f(a);
    cin.ignore(1);
}
```

اینجا نقطهٔ شروع است، این که دقیقاً در این مرحله چه اتفاقاتی می‌افتد برای ما مهم نیست. برای مثال، ممکن است در این مرحله، حافظهٔ لازم از رم گرفته شود. برای این که ببینیم واقعاً در این خط چه اتفاقاتی می‌افتد باید کد اسمبلی برنامه را ببینیم. این مرحله در کیوت‌کریییتور نشان داده نمی‌شود.

قدم دوم:

```
// Program ID: 0390
#include <iostream>
using namespace std;

double f(int x)
{
    double a = x * x;
    double b = x;
    double c = 1;
    double ret = a - b + c;
    return ret;
}
int main()
{
    int a = 2;
    f(a);
    cin.ignore(1);
}
```

در این مرحله، مقدار 2 در متغیر a قرار می‌گیرد.

قدم سوم:

```
// Program ID: 0390
#include <iostream>
using namespace std;

double f(int x)
{
    double a = x * x;
    double b = x;
    double c = 1;
    double ret = a - b + c;
    return ret;
}
int main()
{
    int a = 2;
    f(a);
    cin.ignore(1);
}
```

در این مرحله تابع f فراخوانده می‌شود (یعنی دستوراتش اجرا می‌شوند).

قدم چهارم:

```
// Program ID: 0390
#include <iostream>
using namespace std;

double f(int x)
{
    double a = x * x;
    double b = x;
    double c = 1;
    double ret = a - b + c;
    return ret;
}
int main()
{
    int a = 2;
    f(a);
    cin.ignore(1);
}
```

اجرای این خط سبب می‌شود که برنامه، منتظر فشار کلید Enter صفحه‌کلید بماند. هنگام عبور از این خط، پنجرهٔ خروجی، معمولاً، خودبه‌خود، فعال می‌شود (و اگر نشد باید آن را فعال کرد)، در این هنگام، کلید Enter را فشار دهید و دوباره پنجرهٔ آی‌دی‌ای را فعال کنید.

قدم پنجم:

```
// Program ID: 0390
#include <iostream>
using namespace std;

double f(int x)
{
    double a = x * x;
    double b = x;
    double c = 1;
    double ret = a - b + c;
    return ret;
}
int main()
{
    int a = 2;
    f(a);
    cin.ignore(1);
}
```

این خط، خط پایانی برنامه است و اگر آن را اجرا کنید، اجرای برنامه پایان می‌یابد.

مثال (اجرا به صورت step into): برنامهٔ بالا را، با زدن کلید F11 به صورت step into، قدم به قدم، اجرا می‌کنیم.

قدم اول:

```
// Program ID: 0390
#include <iostream>
using namespace std;

double f(int x)
{
    double a = x * x;
    double b = x;
    double c = 1;
    double ret = a - b + c;
    return ret;
}
int main()
{
    int a = 2;
    f(a);
    cin.ignore(1);
}
```

اینجا، نقطهٔ شروع در ویژوال‌استودیو است. در کیوت‌کریییتور، این مرحله نشان داده نمی‌شود.

قدم دوم:

```
// Program ID: 0390
#include <iostream>
using namespace std;

double f(int x)
{
    double a = x * x;
    double b = x;
    double c = 1;
    double ret = a - b + c;
    return ret;
}
int main()
{
    int a = 2;
    f(a);
    cin.ignore(1);
}
```

در این خط قرار است مقدار 2 در متغیر a قرار گیرد.

قدم سوم:

```
// Program ID: 0390
#include <iostream>
using namespace std;

double f(int x)
{
    double a = x * x;
    double b = x;
    double c = 1;
    double ret = a - b + c;
    return ret;
}
int main()
```

```
{
    int a = 2;
    f(a);
    cin.ignore(1);
}
```

در این خط، قرار است تابع f فراخوانده شود.[۹۰] توجه کنید که فراخوانی تابع f، با آرگومان[۹۱] a انجام شده است که در این خط، مقدار ۲ دارد. پس، در حقیقت، تابع f با آرگومان ۲ فراخوانده شده است. با فشار F11، وارد تابع f می‌شویم و خطوط آن را اجرا می‌کنیم. اما اگر به:

```
double f(int x)
{
    double a = x * x;
    double b = x;
    double c = 1;
    double ret = a - b + c;
    return ret;
}
```

نگاه کنید؛ می‌بینید که قبل از اجرای دستورات این تابع، باید مقدار x مشخص باشد. مقدار x، همان ۲ است که به عنوان آرگومان به تابع دادیم. می‌توان تصور کرد که با نوشتن:

```
f(a);
```

گویی دستور زیر، به طور پنهانی انجام می‌شود:

```
x = a;
```

یا مناسب‌تر است بگوییم دستور:

```
x = 2;
```

به طور پنهانی انجام می‌شود.

قدم چهارم:

```
// Program ID: 0390
#include <iostream>
using namespace std;

double f(int x)
{
    double a = x * x;
    double b = x;
    double c = 1;
    double ret = a - b + c;
    return ret;
}
int main()
{
    int a = 2;
    f(a);
```

[۹۰] یعنی دستوراتش اجرا شود.

[۹۱] یعنی ورودی.

```
    cin.ignore(1);
}
```

چون روش step into را به کار بردیم، وارد تابع f شده‌ایم. پیش از هر چیز، باید بدانیم که x، یک متغیر با نوع int است و مقدار آن، ۲ است؛ زیرا هنگام فراخواندن تابع، مقدار ۲ را به عنوان آرگومان به آن داده‌ایم. این خط، خط اول اجرای تابع است. در این خط، کارهایی (مثل گرفتن حافظهٔ لازم) انجام می‌شود که زیاد برای ما (به عنوان برنامه‌نویس) مهم نیست. البته این مرحله در کیوت‌کریییتور نشان داده نمی‌شود.

قدم پنجم:

```
// Program ID: 0390
#include <iostream>
using namespace std;

double f(int x)
{
    double a = x * x;
    double b = x;
    double c = 1;
    double ret = a - b + c;
    return ret;
}
int main()
{
    int a = 2;
    f(a);
    cin.ignore(1);
}
```

در این خط، قرار است مقدار x * x در a قرار گیرد. توجه کنید که تابع f، با آرگومان ۲ فراخوانده شده است. از این رو، مقدار x برابر با ۲ است. بنابراین، x * x می‌شود ۴. یعنی با تمام‌شدن اجرای این خط، مقدار a برابر با ۴ خواهد بود.

قدم ششم:

```
// Program ID: 0390
#include <iostream>
using namespace std;

double f(int x)
{
    double a = x * x;
    double b = x;
    double c = 1;
    double ret = a - b + c;
    return ret;
}
int main()
{
    int a = 2;
    f(a);
    cin.ignore(1);
}
```

در این خط، قرار است مقدار b برابر با ۲ شود.

قدم هفتم:

```
// Program ID: 0390
#include <iostream>
using namespace std;

double f(int x)
{
    double a = x * x;
    double b = x;
    double c = 1;
    double ret = a - b + c;
    return ret;
}
int main()
{
    int a = 2;
    f(a);
    cin.ignore(1);
}
```

در این خط، قرار است مقدار c برابر با ۱ گردد.

قدم هشتم:

```
// Program ID: 0390
#include <iostream>
using namespace std;

double f(int x)
{
    double a = x * x;
    double b = x;
    double c = 1;
    double ret = a - b + c;
    return ret;
}
int main()
{
    int a = 2;
    f(a);
    cin.ignore(1);
}
```

وقتی به این خط می‌رسیم؛

مقدار a، ۴

مقدار b، ۲

و مقدار c، ۱ است.

پس، مقدار a - b + c می‌شود ۳.

بنابراین، در این خط، مقدار ret، برابر با ۳ می‌گردد.

قدم نهم:

```
// Program ID: 0390
#include <iostream>
using namespace std;
```

```
double f(int x)
{
    double a = x * x;
    double b = x;
    double c = 1;
    double ret = a - b + c;
    return ret;
}
int main()
{
    int a = 2;
    f(a);
    cin.ignore(1);
}
```

در این خط، مقدار ret به عنوان خروجی برگردانده می‌شود. در واقع، این همان مقداری است که f(a) خواهد داشت. البته از این مقدار، در این برنامه، استفاده نمی‌شود.

قدم دهم:

```
// Program ID: 0390
#include <iostream>
using namespace std;

double f(int x)
{
    double a = x * x;
    double b = x;
    double c = 1;
    double ret = a - b + c;
    return ret;
}
int main()
{
    int a = 2;
    f(a);
    cin.ignore(1);
}
```

اینجا، خط پایانی تابع f است.

قدم یازدهم:

```
// Program ID: 0390
#include <iostream>
using namespace std;

double f(int x)
{
    double a = x * x;
    double b = x;
    double c = 1;
    double ret = a - b + c;
    return ret;
}
int main()
{
    int a = 2;
```

```
    f(a);
    cin.ignore(1);
}
```

اجرای این خط، سبب می‌شود، برنامه، منتظر فشار کلید Enter بماند. هنگام عبور از این خط، پنجرهٔ خروجی، به طور معمول، خود به خود فعال می‌گردد (و اگر نشد باید آن را فعال کرد)؛ در این هنگام، کلید Enter را فشار دهید.

قدم دوازدهم:

```
// Program ID: 0390
#include <iostream>
using namespace std;

double f(int x)
{
    double a = x * x;
    double b = x;
    double c = 1;
    double ret = a - b + c;
    return ret;
}
int main()
{
    int a = 2;
    f(a);
    cin.ignore(1);
}
```

این خط، خط پایانی برنامه است. با چند بار فشاردادن F10، اجرای برنامه را تمام کنید.

مثال‌های قبل، نحوهٔ اجرای خطبه‌خط برنامه را نشان می‌دهند. وقتی یک برنامه می‌نویسیم، باید برنامه را با توجه به اجرای خطبه‌خط آن بنویسیم.

برنامه‌نویس‌ها، تصور می‌کنند که یک موجود خیالی، به نام «**کنترل**[۹۲] **برنامه**» وجود دارد، که برنامه را، خطبه‌خط، اجرا می‌کند و پیش می‌رود. آموزنده است اگر تصور کنیم که «کنترل برنامه» خطوط برنامه را، با همان ترتیبی که در دو مثال قبلی دیدیم اجرا می‌کند. بدون تصور وجود «کنترل برنامه» و پیشبینی عملکرد آن، برنامه‌نوشتن ممکن نیست.

هنگامی که کنترل برنامه، به فراخوانی یک تابع می‌رسد، وارد تابع می‌شود و دستورات آن را اجرا می‌کند و سپس، به محل اولیه (که در آنجا، تابع، فراخوانده شده بود) برمی‌گردد.

لازم نیست از ابتدا تا انتهای اجرای قدم‌به‌قدم، تنها یکی از دو روش step over و step into را به کار بریم. در واقع، هر جا خواستیم وارد تابعی شویم و اجرای دستورات آن را ببینیم، از کلید مربوط به step into استفاده می‌کنیم و هر جا خواستیم از روی یک تابع رد شویم، از کلید مربوط به step over استفاده می‌کنیم.

[۹۲] control (flow)

به یاد داشته باشید که وقتی از روی تابع‌هایی که خودتان تعریف نکرده‌اید رد می‌شوید، از کلید step over استفاده کنید تا وارد تعریف نشوید؛ به‌خصوص هنگام عبور از روی cout << ، در برخی آی‌دی‌ای‌ها.

تمرین

۱) فرض کنید در یک آی‌دی‌ای، امکان اجرای قدم‌به‌قدم نیست. یک برنامه بنویسید که بتوان ترتیب اجرای دستورات را، هنگام فراخوانی یک تابع، از خروجی برنامه تشخیص داد.

تابع‌های بدون خروجی

یک تابع، می‌تواند خروجی نداشته باشد. هنگامی که به خروجی نیاز نداشته باشیم، می‌توانیم خروجی را تهی بگذاریم. برای این کار، در تعریف تابع، نوع خروجی را void می‌گذاریم.

مثال: برنامهٔ زیر، مشابه یکی از برنامه‌های پیشین است که چون به خروجی تابع، در برنامه نیاز نیست؛ خروجی آن را void گذاشته‌ایم:

```
// Program ID: 0400
#include <iostream>
using namespace std;

void f(double x)
{
    double a = x * x - x + 1;
    cout << a;
}
int main()
{
    f(2);
    cin.ignore(1);
}
```

خروجی:

```
3
```

void قبل از f، یعنی این که تابع f خروجی ندارد. از آنجا که تابع f خروجی ندارد، در تعریف آن، از دستور return استفاده نشده است؛ زیرا این دستور، مقدار خروجی را مشخص می‌کرد در حالی که اینجا خروجی نداریم.

اگر یک تابع مانند f در مثال بالا، خروجی نداشته باشد، نمی‌توانیم مقدارش را چاپ کنیم؛ مثلاً نمی‌توانیم بنویسیم:

```
cout << f(0);
```

زیرا f(0) چیز قابل‌چاپی نیست.

تمرین

۱) تابعی بدون خروجی بنویسید که مقدار یک عدد صحیح را بگیرد و چاپ کند.

تابع‌های بدون‌ورودی

یک تابع، می‌تواند هیچ ورودی‌ای نداشته باشد. برای تعریف چنین تابعی، لیست پارامترهای آن[۹۳] را، خالی می‌گذاریم:

```
int f()
{
    return 2;
}
```

مثال: برنامه‌ای می‌نویسیم که، درون یک تابع، یک عدد را بگیرد و برگرداند و حاصل را چاپ کند.

```
// Program ID: 0410
#include <iostream>
using namespace std;

int f()
{
    int a;
    cin >> a;
    return a;
}
int main()
{
    cout << f();
    cin.ignore(2);
}
```

خروجی:

```
25
25
```

کنترل برنامه، وارد تابع f می‌شود و با cin >> یک عدد صحیح را، که در اینجا ۲۵ است، می‌گیرد و برمی‌گرداند. مقدار برگردانده‌شده، با cout << چاپ می‌شود.

نکتهٔ قابل‌توجه، نحوهٔ فراخوانی تابع بدون‌ورودی است. برای فراخوانی نوشته‌ایم f(). که در آن پرانتزها وجود دارند، اما درونشان خالی است.

هنگام تعریف یک تابع بدون‌پارامتر، می‌توان در لیست ورودی‌ها (به جای خالی گذاشتن آن) نوشت void، برای نمونه، تابع f در مثال قبل را می‌توان به صورت زیر نوشت:

```
int f(void)
{
    int a;
    cin >> a;
    return a;
}
```

اما هنگام فراخوانی تابع، نمی‌شود در لیست آرگومان‌ها void نوشت؛ مثلاً نوشتن:

```
cout << f(void);
```

---

[۹۳] یعنی ورودی‌های آن.

در مثال بالا خطا دارد.

مثال: تابعی می‌نویسیم که ورودی و خروجی ندارد و تنها پیامی را چاپ می‌کند.

```
// Program ID: 0420
#include <iostream>
using namespace std;

void f()
{
    cout << "hello";
}
int main()
{
    f();
    cin.ignore(1);
}
```

خروجی:

```
hello
```

تمرین

۱) تابعی بدون ورودی و خروجی بنویسید که جمع اعداد ۱ تا ۱۰۰ را چاپ می‌کند.

تابع main

پیش از اولین برنامه، گفتیم که:

```
#include <iostream>
using namespace std;

int main()
{

    cin.ignore(1);
}
```

یک قالب برای برنامه‌نویسی است. اما واقعیت آن است که ما چیزی به نام قالب نداریم، و آنچه گفته شد، تنها برای ساده‌شدن مطالب بود. main، یک تابع است. این تابع، هنگام اجرای برنامه، خودبه‌خود فراخوانده می‌شود و دستوراتش اجرا می‌شوند. cin.ignore نیز نوعی تابع است که از آن، برای جلوگیری از بسته‌شدن خودبه‌خودی برنامه استفاده کرده‌ایم.

پس، یک برنامه، به این شکل اجرا می‌شود که: تابع main (خودبه‌خود) فراخوانده می‌شود و کنترل برنامه، متولد می‌شود. کنترل برنامه، دستورات تابع main را اجرا می‌کند و اگر بین این دستورات، تابع دیگری فراخوانده شود، دستورات آن تابع را نیز اجرا می‌کند، تا اجرای همهٔ دستورات تمام شوند. در این لحظه، اجرای برنامه پایان می‌یابد.

به کنترل برنامه، **ریسمان**[۹۴] نیز می‌گویند. برنامه‌هایی هستند که بیش از یک ریسمان دارند که بحث آن را به اواخر کتاب موکول می‌کنیم. برنامه‌های چندریسمانی، همزمان چند «کنترل برنامه» دارند. بحث ما بر برنامه‌های تک‌ریسمانی متمرکز است و برنامه‌ها، تک‌ریسمانی هستند مگر آن که خود ما با دستوری موجب تولد ریسمان دیگری شویم.

فراخواندن یک تابع توسط تابعی دیگر

این تنها تابع main نیست که می‌تواند توابع دیگر را فرابخواند، هر تابعی می‌تواند هر تابع دیگر (و حتی خودش را) فرابخواند[۹۵].

مثال: فراخواندن یک تابع توسط تابعی دیگر:

```
// Program ID: 0430
#include <iostream>
using namespace std;

void g()
{
    cout << "g() was called  ";
}
void f()
{
    cout << "f() was called  ";
    g();
}
int main()
{
    f();
    cin.ignore(1);
}
```

خروجی:

```
f() was called  g() was called
```

در برنامهٔ بالا، تابع main، تابع f را فرامی‌خواند و تابع f نیز، تابع g را فرامی‌خواند. این برنامه را قدم‌به‌قدم (با روش step into هنگام فراخوانی تابع‌ها) اجرا کنید تا نحوهٔ فراخوانی تابع‌ها و اجرای دستورات، کاملاً برای شما روشن گردد.

تمرین

۱) برنامه‌ای بنویسید که در آن، تابعی خودش را فرابخواند؛ اما اجرای برنامه به صورت طبیعی پایان یابد نه با خطای حین-اجرا.

---

[۹۴] thread

[۹۵] اما تابع استثنایی main را نباید در متن برنامه فرابخوانیم.

اهمیت ترتیب

هنگام فراخوانی یک تابع، باید آن تابع را قبلاً تعریف کرده باشیم. در مثال بخش پیش، می‌بینید تابع `f` تابع `g` را فراخوانده و `g` قبل از `f` تعریف شده است.

به همین سبب است که همهٔ تابع‌ها را قبل از تابع `main` تعریف می‌کنیم. اما مواردی پیش می‌آیند که لازم است یک تابع، تابعی را فرابخواند که پس از خودش تعریف شده است. راهکاری وجود دارد که اهمیت ترتیب تابع‌ها را از بین می‌برد و با آن راهکار، حتی تعریف تابع‌ها را می‌توانیم به بعد از `main` منتقل کنیم. این راهکار را در آینده معرفی خواهیم کرد. این بخش، تنها در مورد اهمیت ترتیب است.

اهمیت ترتیب تعریف، فقط در مورد تابع‌ها نیست. درون یک تابع نیز، ابتدا باید متغیر را تعریف کنیم و سپس، از آن استفاده کنیم و مثلاً نمی‌توان ابتدا استفاده کرد و سپس تعریف.

همچنین، ترتیب اجرای دستورات، به همان شکلی است که در اجرای قدم‌به‌قدم دیدید. ترتیب دستورات نیز مهم است. ممکن است با تعویض جای دو دستور، نتیجهٔ کار عوض شود.

تمرین

۱) برنامه‌ای بنویسید که `"hello"` را چاپ کند؛ اما اگر دو خط متوالی از آن را جابه‌جا کنیم، به جای `"hello"`، `"bye"` را چاپ کند.

پارامتر و آرگومان

در این قسمت، دربارهٔ مقادیر راست و چپ و ارتباط آرگومان و پارامتر توابع سخن خواهیم گفت.

مقدار راست و چپ

اگر تعریف کنیم:

```
int x = 9;
```

می‌توانیم بنویسیم:

```
x = 2;
```

این کد، مقدار `x` را (که در ابتدا، ۹ بوده است) تغییر می‌دهد. اما نمی‌توانیم بنویسیم:

```
9 = 2;
```

زیرا 9 یک متغیر نیست که بتوانیم مقدارش را تغییر دهیم. در حقیقت، هنگامی که می‌نویسیم:

```
int x = 9;
```

x را می‌توان به جای 9 به کار برد؛ اما 9 را نمی‌شود به جای x به کار برد. x، چیزی بیش از 9 دارد و آن حافظهٔ **قابل‌تغییر** است. از این روست که به آن می‌گوییم «**متغیر**». 9؛ یک متغیر نیست، زیرا نمی‌توان آن را تغییر داد.

x و 9، هر دو، عددهایی صحیح هستند، یا اگر دقیق‌تر بگوییم، x و 9 عبارت‌هایی با نوع int هستند و هر دو، مقدار ۹ دارند. هر دو، چهار بایت از رم را در اختیار دارند و مقدار مشترک این دو عبارت (یعنی ۹)، با ترکیب صفر و یک‌های این چهار بایت تعیین می‌گردد؛ اما تفاوت x با 9 در این است که می‌توان نوشت:

```
x = 2;
```

و مقدار x را تغییر داد ولی نمی‌شود نوشت:

```
9 = 2;
```

زیرا مقدار 9، که همان ترکیب صفر و یک‌های حافظهٔ تحت اختیار 9 است، غیرقابل‌تغییر است.

به x، **مقدار چپ**[۹۶] می‌گویند، زیرا می‌توان آن را در طرف چپ عملگر انتساب (یعنی «=») قرار داد. به 9، **مقدار راست**[۹۷] می‌گویند، چون نمی‌توان آن را در طرف چپ عملگر انتساب قرار داد و تنها می‌تواند طرف راست آن قرار بگیرد. پس، هر متغیر، یک مقدار چپ است و می‌توان مقدار آن را تغییر داد.

یک مقدار چپ دو ویژگی دارد:

۱) مقدار.

۲) قابلیت تغییر مقدار.

اما یک مقدار راست تنها ویژگی اول را دارد.

توجه که مقدارهای چپ و راست در این بخش، مطابق تعاریف قدیمی C++، معرفی شده‌اند. در استانداردهای جدید، این تعاریف متفاوت و پیچیده شده‌اند. تعاریف جدید را در اواخر این کتاب خواهیم آورد؛ اما آنچه در اینجا اهمیت دارد، فهمیدن کدهاست به گونه‌ای است که امکان نوشتن کدهای جدید فراهم شود. تعاریف قدیمی در این مرحله از معرفی C++، مناسب‌ترند.

تمرین

۱) آیا "hello" یک مقدار چپ است؟

[۹۶] LValue

[۹۷] RValue

ارتباط بین پارامتر و آرگومان تابع

در تعریف تابع:

```
void f(int a, int b)
{
    cout << a + b;
}
```

به a و b **پارامتر** می‌گویند. هنگام فراخوانی این تابع:

```
int x = 2;
f(x, 3);
```

به x و 3 **آرگومان** می‌گویند. البته، گاه، بین اصطلاح‌های «پارامتر» و «آرگومان» تفاوتی قائل نمی‌شوند. اما در این کتاب، این دو اصطلاح را یکسان **نمی‌دانیم**.

هنگامی که یک تابع فراخوانده می‌شود، کنترل برنامه[۹۸]، پیش از هر کاری، مقدار آرگومان‌ها را در پارامترها، **کپی**[۹۹] می‌کند. برای مثال، اگر تعریف کنیم:

```
void f(int a, int b)
{
    cout << a + b;
}
```

و در تابع main بنویسیم:

```
int x = 2;
f(x, 3);
```

هنگام اجرای خط:

```
f(x, 3);
```

مقدار x (یعنی ۲) در a و مقدار 3 در b **کپی** می‌شود. می‌توان تصور کرد که گویا خطوط:

```
a = x;
b = 3;
```

به طور پنهانی اجرا می‌شوند. پس از این کپی‌شدن، اجرای دستورات تابع f آغاز می‌گردد.

پارامترهای a و b، متغیرهای جدیدی هستند و ربطی به آرگومان‌ها ندارند؛ تنها، **مقدار اولیهٔ** آن‌ها، به وسیلهٔ آرگومان‌ها تعیین می‌گردد.

مثال: در برنامهٔ زیر، نشان می‌دهیم که پارامترها نیز، همانند متغیرهای دیگر عمل می‌کنند.

---

[۹۸] یعنی همان خط درحال‌اجرا.

[۹۹] copy

```
// Program ID: 0440
#include <iostream>
using namespace std;

void f(int a)
{
    a = 5;
    cout << a;
}
int main()
{
    f(0);
    cin.ignore(1);
}
```

خروجی:

```
5
```

می‌بینید که مقدار پارامتر a را، در خط:

```
a = 5;
```

مانند هر متغیر دیگری، تغییر داده‌ایم و مقدارش را چاپ کرده‌ایم.

هر تابع، متغیرهای مخصوص به خودش را دارد و **متغیرهای یک تابع، ارتباطی به تابع‌های دیگر ندارند؛ حتی اگر متغیرها، هم‌نام باشند**. نکتهٔ قابل‌توجه دیگر در مورد تابع‌ها، این است که وقتی یک تابع فراخوانده شد، و همهٔ دستورات آن، انجام و اجرای آن تمام شد، همهٔ متغیرها و پارامترهای تابع، پاک می‌شوند و حافظه‌هایی که به آن‌ها داده شده بود، از دسترس خارج می‌شوند.

مثال: پارامتر هم‌نام با آرگومان و تغییر مقدار پارامتر:

```
// Program ID: 0450
#include <iostream>
using namespace std;

void f(int a)
{
    a = 2;
}
int main()
{
    int a = 1;
    f(a);
    cout << a;
    cin.ignore(1);
}
```

خروجی:

```
1
```

a، در تابع main، ارتباطی به a در تابع f ندارد. وقتی کنترل برنامه وارد تابع f می‌شود، مکان جدیدی از حافظه را در اختیار می‌گیرد، نام آن را a می‌گذارد و مقدار aی تابع main را در aی جدید کپی می‌کند. هنگام خروج از تابع

f، aی جدید از بین می‌رود؛ یعنی حافظه‌ای که با نام a (جدید) در اختیار برنامه بود، از اختیار برنامه خارج می‌شود، و به اصطلاح آزاد می‌شود؛ اما، با این که aی جدید از بین می‌رود، aی قبلی همچنان موجود است.

a در main، آرگومان تابع f و a در f، پارامتر تابع f است. برای این که مطمئن باشیم aهای دو تابع، به یکدیگر ربطی ندارند، در برنامهٔ بالا، مقدار a در تابع main را، پس از اجرای تابع f چاپ کرده‌ایم، تا آشکار گردد که مقدارش تغییری نکرده است. اما، اگر واقعاً می‌خواستیم تغییر کند باید چه می‌کردیم؟ پاسخ ساده است. کافی بود تنها یک علامت & قبل از پارامتر a بگذاریم. در بخش آینده، در این باره سخن خواهیم گفت.

تغییر مقدار یک متغیر با یک تابع

فرض کنید یک متغیر داریم و می‌خواهیم آن را به درون یک تابع بفرستیم تا مقدارش را تغییر دهد. با مباحثی که تا به اینجا گفته شده‌اند، این کار امکان ندارد؛ زیرا پارامتر یک تابع، تنها مقدار آرگومان را می‌گیرد؛ نه خود آرگومان را؛ و پارامتر، یک متغیر جداست. مثال آخر بخش قبل نشان می‌دهد که هر تغییری در پارامتر، هیچ تأثیری در آرگومان ندارد؛ اما در اینجا می‌خواهیم تأثیر داشته باشد. برای این که تأثیر داشته باشد، درست قبل از پارامتر، یک امپرسند[۱۰۰] یعنی علامت «&» می‌گذاریم.

مثال: مثال آخر بخش قبل را طوری اصلاح می‌کنیم که تابع، بتواند مقدار آرگومان خود را تغییر دهد:

```
// Program ID: 0460
#include <iostream>
using namespace std;

void f(int& x)
{
    x = 2;
}
int main()
{
    int a = 1;
    f(a);
    cout << a;
    cin.ignore(1);
}
```

خروجی:

```
2
```

در برنامهٔ بالا، متغیر a، با مقدار اولیهٔ ۱ تعریف شده است. سپس، این متغیر، به درون تابع f فرستاده شده، اسم جدید x را پیدا کرده و مقدارش تغییر کرده است. توجه کنید که تابع:

```
void f(int& x)
{
    x = 2;
}
```

[۱۰۰] ampersand

یک متغیر را، به عنوان آرگومان می‌خواهد، نه فقط مقدار را و کل متغیر را می‌گیرد و چنین تابعی، اغلب، مانند برنامهٔ بالا، اسم متغیر را نیز عوض می‌کند؛ در برنامهٔ بالا، x همان a است که موقتاً اسمش عوض شده است.

a، به درون تابع فرستاده شده و مقدارش تغییر کرده و برابر با ۲ شده است. پس از اجرای تابع f، مقدار a را چاپ کرده‌ایم.

فرق برنامهٔ بالا، با برنامهٔ آخر بخش قبل این است که وقتی کنترل برنامه، وارد f می‌شود، پارامتر، حافظهٔ جدیدی پیدا نمی‌کند و از همان حافظهٔ قبلی آرگومان استفاده می‌کند. به همین سبب، هنگامی که مقدار پارامتر، عوض می‌شود، مقدار آرگومان نیز عوض می‌شود. توضیح بیشتر این که **مقدار یک متغیر، یعنی مقداری که در حافظهٔ آن ذخیره شده است**. وقتی a و x هر دو از یک حافظه استفاده کنند، مقدار هر دو، همواره یکسان است.

تابع:

```
void f(int& x)
{
    x = 2;
}
```

یک متغیر را به عنوان آرگومان می‌خواهد. به همین دلیل، به پارامتر آن، **رفرنس**[۱۰۱] می‌گویند. به طور دقیق‌تر، **آرگومان f بالا، باید یک مقدار چپ باشد**. در حالی که برای:

```
void f(int x)
{
    x = 2;
}
```

آرگومان، مقدار راست هم می‌تواند باشد.

مثال: برنامهٔ زیر، در ویژوال‌استودیو و کیوت‌کریتور اجرا نمی‌شود و خطا دارد:

```
// Program ID: 0470
#include <iostream>
using namespace std;

void f(int& b)
{
    b = 2;
    cout << b;
}
int main()
{
    f(1);
    cin.ignore(1);
}
```

خطا:

```
'void f(int &)': cannot convert argument 1 from 'int' to 'int &'
```

در مستطیل قرمز رنگ، پیام خطایی که ویژوال‌استودیو داده است، آورده شده است.[۱۰۲]

[۱۰۱] reference

چون 1، یک مقدار چپ نیست، برنامهٔ بالا خطا دارد. f، یک مقدار چپ، یعنی یک متغیر را به عنوان پارامتر می‌خواهد.

اشتباه در کجاست که خطا گرفته می‌شود؟ قبلاً گفتیم که وقتی به جای:

```
void f(int b) { … }
```

می‌نویسیم:

```
void f(int& b) { … }
```

حافظهٔ جدید قابل‌تغییری برای b در نظر گرفته نمی‌شود. اما باید یک حافظهٔ قدیمی موجود باشد تا جدیدش به وجود نیاید؛ ولی ما در برنامه نوشته‌ایم ;f(1)، در حالی که 1، حتی اگر حافظه‌ای داشته باشد نیز، این حافظه خارج‌ازدسترس است و حتی اگر در دسترس بود، قابل‌تغییر نبود. تابع f، به مکانی از حافظه نیاز دارد؛ در حالی که ما با نوشتن int& (به جای int) به آن اجازه نداده‌ایم که خودش حافظه‌ای تهیه کند، و از طرف دیگر، حافظهٔ قابل‌تغییری نیز، از طریق آرگومان، در اختیارش نگذاشته‌ایم. اینجاست که کامپایلر تصمیم می‌گیرد کامپایل‌کردن برنامه را متوقف کند و پیغام خطایی نشان دهد.

البته کامپایلرهایی هستند که کامپایل را متوقف نمی‌کنند و تنها یک **اخطار**[۱۰۳] نشان می‌دهند. این گونه کامپایلرها احتمالاً از طریق دیگری حافظهٔ مورد نیاز را تهیه می‌کنند.

پارامتر به عنوان خروجی

می‌دانید که هر تابع، تنها یک خروجی دارد؛ اما می‌تواند چند ورودی داشته باشد. حال، اگر بخواهیم چند خروجی از یک تابع بگیریم، راه حل چیست؟ یک راه، استفاده از پارامترهای رفرنس است. پارامترهایی را که رفرنس هستند، می‌توان به عنوان خروجی به کار برد. به این ترتیب، می‌توانیم چند خروجی داشته باشیم؛ نه فقط یک خروجی.

مثال: تابعی می‌نویسیم که دو عدد را بگیرد و حاصل‌ضرب و حاصل‌جمع آن‌ها را بنویسد.

```
// Program ID: 0480
#include <iostream>
using namespace std;

void f(int a, int b, int& haasele_jam, int& haasele_zarb)
{
    haasele_jam = a + b;
    haasele_zarb = a * b;
}
int main()
```

---

[۱۰۲] برای دیدن این پیام، باید پنجرهٔ Error List فعال باشد. برای فعال‌کردن آن می‌توانید این مسیر را طی کنید:

View → Error List

پیغام‌های خطا در این پنجره قابل‌مشاهده است.

[۱۰۳] warning

```
{
    int a = 6;
    int b = 5;
    int haasele_jam, haasele_zarb;
    f(a, b, haasele_jam, haasele_zarb);
    cout << haasele_jam << "    " << haasele_zarb;
    cin.ignore(1);
}
```

خروجی:

```
11    30
```

در این برنامه، haasele_jam و haasele_zarb به عنوان خروجی مورد استفاده قرار گرفته‌اند. مقدار هر دو را، تابع f مشخص کرده است. پس می‌توانیم بگوییم آن‌ها نوعی خروجی برای تابع f هستند.

یک پارامتر رفرنس، می‌تواند هم نقش ورودی داشته باشد هم نقش خروجی:

مثال: تابعی می‌نویسیم که یک عدد را بگیرد و به توان ۲ برساند.

```
// Program ID: 0490
#include <iostream>
using namespace std;

void f(int& a)
{
    a = a * a;
}
int main()
{
    int a = 5;
    f(a);
    cout << a;
    cin.ignore(1);
}
```

خروجی:

```
25
```

در اینجا، پارامتر a، هم نقش ورودی دارد هم نقش خروجی. در مورد:

```
a = a * a;
```

ابتدا، مقدار طرف راست حساب می‌شود و مقداری به دست می‌آید، و سپس، این مقدار، در متغیر طرف چپ قرار می‌گیرد. برای نمونه، اگر مقدار a، ۵ باشد، پس از اجرای این خط، مقدار آن می‌شود ۲۵.

راه قدیمی تغییر مقدار یک متغیر با تابع

راه قدیمی‌تری نیز برای برای تغییر یک متغیر در یک تابع وجود دارد. این روش، در زبان C که از C++ امکانات کم‌تری دارد، به کار می‌رود. از آنجایی که خیلی از تابع‌های مهم، برای هر دو زبان C و C++ نوشته می‌شوند، دانستن این روش اهمیت دارد. این روش، به این صورت است که به جای:

```
void f(int a, int b, int& haasele_jam, int& haasele_zarb)
{
    haasele_jam = a + b;
    haasele_zarb = a * b;
}
```

می‌نویسیم:

```
void f(int a, int b, int* haasele_jam, int* haasele_zarb)
{
    *haasele_jam = a + b;
    *haasele_zarb = a * b;
}
```

یعنی جلوی پارامترها، به جای امپرسند (&)، ستاره (*) می‌گذاریم و درون تابع، هر جا که پارامترها ظاهر می‌شوند، جلوی آن‌ها ستاره می‌گذاریم. به پارامترهای haasele_jam و haasele_zarb، در این شکل جدید از تابع، **اشاره‌گر**[۱۰۴] می‌گویند.

شیوهٔ فراخوانی تابع هم کمی متفاوت است؛ به جای:

```
f(a, b, haasele_jam, haasele_zarb);
```

باید نوشت:

```
f(a, b, &haasele_jam, &haasele_zarb);
```

یعنی جلوی آرگومان‌ها باید امپرسند گذاشت.

مثال: با استفاده از پارامترهای اشاره‌گر، تابعی می‌نویسیم که یک عدد را بگیرد و به توان دو برساند.

```
// Program ID: 0500
#include <iostream>
using namespace std;

void f(int* a)
{
    *a = (*a)*(*a);
}
int main()
{
    int a = 5;
    f(&a);
    cout << a;
    cin.ignore(1);
}
```

خروجی:

[۱۰۴] pointer

```
25
```

در اینجا، پارامتر a، هم نقش ورودی دارد هم نقش خروجی. به خط‌های:

```
*a = (*a)*(*a);
```

و:

```
f(&a);
```

توجه کنید و این خط‌ها را با آخرین مثال بخش قبل مقایسه کنید.

این شیوهٔ قدیمی معمولاً در توابع API در برنامه‌نویسی ویندوز زیاد به کار می‌رود. البته با وجود شیوهٔ جدید که در بخش‌های قبلی گفته شد، نیازی به استفاده از این شیوه برای ما وجود ندارد؛ مگر وقتی که به زبان C برنامه می‌نویسیم.

دستور return

دستور return می‌تواند چندین بار در یک تابع ظاهر شود؛ و در هر جای تابع که باشد، سبب خروج فوری از تابع می‌شود.

مثال: تابعی که اعداد زوج و فرد را تشخیص می‌دهد:

```
// Program ID: 0510
#include <iostream>
using namespace std;

bool isEven(int a)
{
    if (a % 2 == 0)
        return true;
    else
        return false;
}
int main()
{
    cout << isEven(10);
    cin.ignore(1);
}
```

خروجی:

```
1
```

اگر دستور return true اجرا شود، اجرای تابع تمام می‌شود و خروجی تابع می‌شود true. همین‌طور در مورد return false.

مقدار تابع isEven، به ازای پارامتر ۱۰ چاپ شده است. cout << **مقدار یک گزاره (با نوع bool) را اگر درست باشد، ۱ و اگر نادرست باشد، ۰ چاپ می‌کند.** چون ۱۰ زوج است، ۱ (یعنی درست) چاپ شده است.

به محض اجرای دستور return، کنترل برنامه، از تابع خارج می‌شود و دستورهای پس از آن در تابع، اجرا **نمی‌شوند**. برای مثال، در:

```
int f()
{
    cout << "hello";
    return 1;
    cout << "bye";
}
```

خط:

```
cout << "bye";
```

نمی‌تواند اجرا شود؛ از این رو، به‌کاربردن این تابع در بعضی کامپایلرها، سبب صدور اخطار می‌گردد.

در تابع‌های بدون خروجی نیز، می‌توانیم از return استفاده کنیم؛ تنها جلوی آن، نباید عدد یا چیز دیگری بگذاریم[۱۰۵]:

```
return;
```

مثال: برنامه‌ای می‌نویسیم که از کاربر، عددی بگیرد و اگر این عدد ۱۰ یا بیشتر بود، هیچ کاری نکند و اگر این‌طور نبود، پیام "mardood!" را چاپ کند:

```
// Program ID: 0520
#include <iostream>
using namespace std;

void f(double nomreh)
{
    if (10 <= nomreh)
        return;
    cout << "mardood!";
}
int main()
{
    double mo_addel;
    cin >> mo_addel;
    f(mo_addel);
    cin.ignore(2);
}
```

خروجی:

```
9.75
mardood!
```

[۱۰۵] اما می‌شود نوشت return g(); به شرط این که خروجی تابع g، void باشد.

عدد ۹/۷۵ را وارد کرده‌ایم و چون از ۱۰ کم‌تر بوده است، دستور return; اجرا نشده و "mardood!" چاپ شده است. تابع f بالا را، با اضافه‌کردن else، می‌توان به این صورت هم نوشت:

```
void f(double nomreh)
{
    if (10 <= nomreh)
        return;
    else
        cout << "mardood!";
}
```

اما به else نیازی نیست؛ زیرا اگر nomreh <= 10 درست باشد، اجرای تابع تمام می‌شود و خط:

```
cout << "mardood!";
```

اصلاً، اجرا نمی‌شود. اگر هم درست نباشد، کنترل برنامه به خط بعدی می‌رود و:

```
cout << "mardood!";
```

را اجرا می‌کند. پس، نیازی به else نیست.

در تابع main نیز می‌توان از دستور return استفاده کرد و حتی معمول است که در آخر تابع main خط:

```
return 0;
```

را بگذارند. main (وقتی خروجیش int باشد)، تنها تابع با خروجی غیر void است که در آن، می‌توان از return استفاده نکرد؛ یعنی **هر تابعی که خروجی دارد، باید return هم داشته باشد مگر** main.

حلقه‌ها

حلقه‌ها هنگامی به کار می‌آیند که بخواهیم یک یا چند دستور خاص، چندین بار اجرا شوند؛ مثلاً ۱۰ بار اجرا شوند، یا ۱۰۰۰ بار اجرا شوند. ممکن است تعداد اجراها به صورت حین-اجرا[۱۰۶] مشخص شود؛ یعنی قبل از اجرا، تعداد معلوم نباشد و در حین-اجرا معلوم شود؛ به عنوان مثال، ممکن است تعداد را کاربر وارد کند. با آنچه، تا به حال، گفته‌ایم، انجام چنین کاری ممکن نیست. حلقه‌ها این کار را ممکن می‌کنند.

شاید بتوان گفت، برنامه‌نویسی واقعی پس از آشنایی با حلقه‌ها آغاز می‌گردد؛ یعنی تا به حال، همه چیز مقدماتی بود؛ و برنامه‌نویسی از این به بعد شروع می‌شود. می‌توان گفت بدون استفاده از حلقه‌ها، هیچ برنامهٔ مهمی نمی‌توان نوشت. البته ابزار قوی‌تری وجود دارد که هر حلقه‌ای را می‌توان با استفاده از آن ساخت و آن ابزار، تابع بازگشتی است که در آینده معرفی خواهد شد.

[۱۰۶] run-time

حلقهٔ for

فرض کنید می‌خواهیم ۱۰ بار "hello" را چاپ کنیم. یک راه این است که ۱۰ بار بنویسیم:

```
cout << "hello";
```

اما راه بهتر، استفاده از حلقهٔ for است. برای این که ۱۰ بار یک دستور را انجام دهیم، باید قبل از آن بنویسیم:

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

به عنوان مثال، برای ۱۰ بار چاپ "hello"، می‌نویسیم:

```
for (int i = 1; i <= 10; i++)
    cout << "hello";
```

اگر بخواهیم ۵ بار "hello" چاپ شود، می‌نویسیم:

```
for (int i = 1; i <= 5; i++)
    cout << "hello";
```

مثال: برنامه‌ای می‌نویسیم که ۷ بار "hello " را چاپ کند.

```
// Program ID: 0530
#include <iostream>
using namespace std;

int main()
{
    for (int i = 1; i <= 7; i++)
        cout << "hello ";
    cin.ignore(1);
}
```

خروجی:

```
hello hello hello hello hello hello hello
```

در هر بار اجرای یک دستور، به وسیلهٔ for (int i = 1; i <= 10; i++)، مقدار i یکی زیاد می‌شود. برنامهٔ زیر، این گفته را ثابت می‌کند:

```
// Program ID: 0540
#include <iostream>
using namespace std;

int main()
{
    for (int i = 1; i <= 6; i++)
        cout << i;
    cin.ignore(1);
}
```

خروجی:

```
123456
```

در برنامهٔ فوق، خط:

```
cout << i;
```

۶ بار اجرا شده، و در هر بار اجرا، مقدار i چاپ شده است. می‌بینید مقدار i، از ۱ تا ۶ تغییر می‌کند. اگر با ریاضیات آشنا باشید، ممکن است این رفتار for، شما را به یاد:

$$\sum_{i=۱}^{n} a_i$$

یا:

$$\bigcup_{i=۱}^{n} A_i$$

بیندازد.

مثال: برنامه‌ای می‌نویسیم که اعداد ۱ تا ۱۰۰ را با هم جمع کند و حاصل را بنویسد:

```
// Program ID: 0550
#include <iostream>
using namespace std;

int main()
{
    int sum = 0;
    for (int i = 1; i <= 100; i++)
        sum = sum + i;
    cout << sum;
    cin.ignore(1);
}
```

خروجی:

```
5050
```

در این برنامه، sum، در ابتدا، مقدار صفر دارد. با هر بار اجرای:

```
sum = sum + i;
```

مقدار آن به اندازهٔ i تا زیاد می‌شود. sum + i یعنی مقدار قبلی، به‌اضافهٔ i. اما، i، در ابتدا، مقدار ۱ دارد، سپس، مقدار ۲ پیدا می‌کند؛ بعد، مقدار ۳ پیدا می‌کند و همین‌طور تا ۱۰۰. پس sum، در ابتدا، صفر است؛ سپس، یکی به آن اضافه می‌شود؛ سپس، دو تا به آن اضافه می‌شود؛ سپس، سه تا و همین‌طور تا صد. هنگامی که کار، پایان می‌یابد؛ حاصل‌جمع ۱ تا ۱۰۰ در sum قرار گرفته است. برنامهٔ فوق، این حاصل‌جمع را چاپ کرده است.

sum را می‌توان همانند یک کیسهٔ خالی در نظر گرفت که ۱۰۰ نفر در آن تعدادی توپ می‌ریزند. نفر اول، ۱ توپ می‌اندازد، نفر دوم، ۲ توپ و همین‌طور تا نفر صدم که ۱۰۰ توپ می‌اندازد. در پایان:

$۱ + ۲ + ۳ + \cdots + ۱۰۰$

توپ در کیسه است.

اگر بخواهیم i از ۱۱ تا ۲۰ تغییر کند، می‌نویسیم:

```
for (int i = 11; i <= 20; i++)
```

و در ضمن، به جای i، می‌توان هر نام دیگری نیز انتخاب کرد، به‌خصوص اگر قبلاً متغیری به نام i موجود باشد و استفادهٔ مجدد از شناسهٔ i ممکن نباشد.

مثال: برنامه‌ای می‌نویسیم که حاصل‌ضرب و حاصل‌جمع اعداد ۱۰ تا ۱۵ را پیدا کند:

```
// Program ID: 0560
#include <iostream>
using namespace std;

int main()
{
    int sum = 0;
    int product = 1;
    for (int a = 10; a <= 15; a++)
    {
        sum = sum + a;
        product = product * a;
    }
    cout << "sum is: " << sum;
    cout << " product is: " << product;
    cin.ignore(1);
}
```

خروجی:

```
sum is: 75 product is: 3603600
```

در این برنامه، پس از for، یک بلوک قرار گرفته است. علت استفاده از بلوک این است که دستورات بیشتری به for وابسته و هر بار، اجرا شوند[۱۰۷].

در مورد sum، در مثال پیشین توضیح داده شد. product هم مشابه sum است. ابتدا، product مقدار ۱ دارد و با ضرب اعداد ۱۰ تا ۱۵ در آن، در نهایت، مقدارش برابر می‌شود با ۱۵ × ۱۴ × ۱۳ × ۱۲ × ۱۱ × ۱۰. در حلقهٔ for، به جای حرف i، از حرف a استفاده کرده‌ایم.

پس از for، باید یک دستور قرار بگیرد و مهم نیست آن دستور چه باشد. به همین سبب، بعد از for، می‌توان دستور for دیگری گذاشت؛ مانند این برنامه:

```
// Program ID: 0570
#include <iostream>
using namespace std;

int main()
{
    for (int a = 1; a <= 5; a++)
        for (int b = 1; b <= 5; b++)
            cout << "(" << a << "," << b << ") ";
    cin.ignore(1);
}
```

خروجی:

[۱۰۷] این دقیقاً مثل همان کاری است که با if می‌کردیم تا دستورات بیشتری را به یک if وابسته کنیم.

```
(1,1) (1,2) (1,3) (1,4) (1,5) (2,1) (2,2) (2,3) (2,4) (2,5) (3,1) (3,2)
(3,3) (3,4) (3,5) (4,1) (4,2) (4,3) (4,4) (4,5) (5,1) (5,2) (5,3) (5,4)
(5,5)
```

دستور:

```
cout << "(" << a << "," << b << ") ";
```

۲۵ بار اجرا می‌شود و تمام زوج‌های مرتب (a,b) را، که $۱ \le a \le ۵$ و $۱ \le b \le ۵$؛ چاپ می‌کند. در واقع، مقدار a از ۱ تا ۵ تغییر می‌کند، و در هر تغییر، یک بار:

```
for (int b = 1; b <= 5; b++)
    cout << "(" << a << "," << b << ") ";
```

اجرا می‌شود که خود، ۵ بار خط:

```
cout << "(" << a << "," << b << ") ";
```

را به ازای b از ۱ تا ۵ اجرا می‌کند. ۵ بار اجرا، و در هر اجرا ۵ بار چاپ، می‌شود ۲۵ بار چاپ.

مثال طولانی زیر، سعی دارد به شما روش حل یک مسألهٔ برنامه‌نویسی را آموزش دهد:

مثال: برنامه‌ای می‌نویسیم که اعداد اول بین ۱ و ۵۰۰ را چاپ کند. عدد صحیح n اول است اگر حاصل‌ضرب دو عدد صحیح بزرگ‌تر از ۱ **نباشد**. پس اگر **برای هر** $i \ge ۲$ و **هر** $j \ge ۲$، داشته باشیم:

$$n \neq ij$$

آنگاه n اول است. اما، اگر $i \le n$ یا $j \le n$، نابرابری:

$$n < ij$$

برقرار است؛ در نتیجه، برای اثبات اول‌بودن n، کافی است ثابت کنیم برای هر $i = ۲, ۳, \dots, n-۱$ و هر $j = ۲, ۳, \dots, n-۱$، نامساوی:

$$n \neq ij$$

برقرار است. با استفاده از این نتیجه، می‌توانیم تابعی بنویسیم که اول‌بودن یک عدد را مشخص کند:

```
bool isPrime(unsigned n)
{
    bool ret = true;
    for (int i = 2; i <= n - 1; i++)
        for (int j = 2; j <= n - 1; j++)
            if (n == i * j)
                ret = false;// عدد اول نیست
    return ret;
}
```

در این تابع، ret با مقدار اولیهٔ true (درست)، تعریف می‌شود؛ یعنی، در ابتدای کار، فرض می‌کنیم عدد n، اول باشد، سپس، با دو حلقهٔ for تودرتو، بررسی می‌کنیم که عبارت n == i * j برقرار نباشد (یعنی درست نباشد)، اما اگر برقرار باشد، درمی‌یابیم که عدد n، اول نیست و قرار می‌دهیم:

```
ret = false;
```

در پایان، ret برگردانده می‌شود. پس، وقتی n اول است، مقدار true و وقتی اول نیست، مقدار false برگردانده می‌شود. حال، بگذارید به سبک ریاضیدانان، دوباره به طور دقیق‌تر تابع را بررسی کنیم، تا از عملکرد درست آن مطمئن شویم. این کار، برای همهٔ موارد، بین برنامه‌نویس‌ها رایج نیست، اما انجام آن تا حد زیادی بی‌نقص‌بودن تابع را تضمین می‌کند:

قضیه: خروجی تابع bool isPrime(unsigned n) اگر مقدار n عدد اول باشد، true، و اگر اول نباشد، false است.

اثبات: الف) فرض کنید مقدار n اول است. با اجرای:

```
bool ret = true;
```

مقدار ret می‌شود true. باید ثابت کنیم این مقدار تا پایان تابع تغییر نمی‌کند. یعنی باید ثابت کنیم خط:

```
ret = false;
```

اجرا نمی‌شود. از آنجا که این خط، پس از یک if، قرار دارد، باید ثابت کنیم شرط n == i * j، هیچگاه برقرار نخواهد شد. اما از آنجا که n اول است و i و j همواره از ۱ بزرگ‌ترند، طبق تعریف اول‌بودن، n == i * j هیچگاه برقرار نمی‌شود. به این ترتیب، ثابت شد که مقدار ret، تا پایان کار تابع، true باقی می‌ماند و همین مقدار برگردانده می‌شود. یعنی خروجی true است.

ب) فرض کنید مقدار n اول نباشد، با اجرای:

```
bool ret = true;
```

مقدار ret می‌شود true. اما چون n اول نیست، عددهای a و b بین اعداد $۱ - n, \dots, ۳, ۲$، پیدا می‌شوند که:

$$n = ab$$

پس، با توجه این که i و j در دو حلقهٔ:

```
for (int i = 2; i <= n - 1; i++)
    for (int j = 2; j <= n - 1; j++)
```

همهٔ مقدارهای $۱ - n, \dots, ۳, ۲$ را اخذ می‌کنند؛ پس حتماً، مقدار a و b را نیز اخذ می‌کنند. از این رو، حداقل یک بار شرط n == i * j برقرار می‌شود و خط:

```
ret = false;
```

اجرا و مقدار ret، false می‌شود. در نهایت، همین مقدار برگردانده می‌شود؛ یعنی خروجی، false است.

به این ترتیب، ثابت شد که تابع isPrime درست کار می‌کند. شاید همین «درست کار کردن»، از نظر یک ریاضیدان کافی باشد، اما در برنامه‌نویسی، علاوه بر درست‌کارکردن تابع، چیزهایی دیگری نیز اهمیت دارند؛ از

جمله، سرعت اجرای تابع. تابع بالا، از نظر سرعت، به‌هیچ‌وجه خوب نیست و می‌توان توابع بسیار سریع‌تری، با عملکرد یکسان نوشت که آن‌ها هم درست کار می‌کنند.

برنامه‌نویس‌ها، معمولاً، برای اثبات درست‌کارکردن تابع، آن را تنها در حالات مختلف آزمایش می‌کنند. مثلاً به این صورت:

```
// Program ID: 0580
#include <iostream>
using namespace std;

bool isPrime(unsigned n)
{
    bool ret = true;
    for (int i = 2; i <= n - 1; i++)
        for (int j = 2; j <= n - 1; j++)
            if (n == i * j)
                ret = false;
    return ret;
}
int main()
{
    if (isPrime(15) == true)
        cout << "prime";
    else
        cout << "not prime";
    cin.ignore(1);
}
```

خروجی:

```
not prime
```

البته این واقعاً یک اثبات نیست؛ و تنها یک آزمون است؛ اما با همین آزمون‌ها برنامه‌نویس می‌تواند مطمئن شود که منطق ابتدایی ذهن او هنگام نوشتن برنامه، خالی از اشکال بوده است؛ و نیازی نیست که جزئیات آن منطق، به صورت ریاضی بازنویسی و بررسی شود.

حال، می‌توانیم به آسانی، تابعی بنویسیم که اعداد اول بین ۱ و ۵۰۰ را چاپ کند:

```
// Program ID: 0590
#include <iostream>
using namespace std;

bool isPrime(unsigned n)
{
    bool ret = true;
    for (unsigned i = 2; i <= n - 1; i++)
        for (unsigned j = 2; j <= n - 1; j++)
            if (n == i * j)
                ret = false;
    return ret;
}
int main()
{
    for (unsigned i = 1; i <= 500; i++)
        if (isPrime(i) == true)
        {
            cout << i;
            cout << " ";
```

```
        }
    cin.ignore(1);
}
```

خروجی:

```
1 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
101 103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191
193 197 199 211 223 227 229 233 239 241 251 257 263 269 271 277 281 283
293 307 311 313 317 331 337 347 349 353 359 367 373 379 383 389 397 401
409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499
```

طبق تعریفی که ما از عدد اول گفتیم، عدد ۱ نیز اول است، و به همین سبب، در خروجی، ۱ نیز چاپ شده است. اما در ریاضیات، ۱ را عدد اول به حساب نمی‌آورند. از آنجا که، در برنامهٔ بالا، کاری با اعداد منفی نداریم، در حلقه‌های for، به جای int، unsigned گذاشته‌ایم:

```
for (unsigned i = 2; i <= n - 1; i++)
    for (unsigned j = 2; j <= n - 1; j++)
```

تمرین

۱) می‌گوییم دو عدد صحیح m و n، نسبت به هم، اولند، اگر هیچ مقسوم‌علیه مشترکی نداشته باشند. تابعی به شکل bool coprime(int m, int n) بنویسید که اول‌بودن دو عدد صحیح نسبت به هم را مشخص کند.

۲) اگر n یک عدد صحیح بزرگ‌تر از ۱ باشد، تعداد اعداد صحیح k که $1 \le k \le n$ و k نسبت به n اول است، با $\phi(n)$ نشان داده می‌شود. تابعی به شکل int phi(int n) بنویسید که $\phi(n)$ را برگرداند.

حلقهٔ while

حلقهٔ while، یک دستور یا یک بلوک از دستورات را تا زمانی که یک شرط به‌خصوص برقرار است، اجرا می‌کند. برنامهٔ زیر را ببینید:

```
// Program ID: 0600
#include <iostream>
using namespace std;

int main()
{
    int n = 0;
    while (n < 10)
    {
        cout << n;
        n++;
    }
    cin.ignore(1);
}
```

خروجی:

```
0123456789
```

در این برنامه، هنگامی که کنترل برنامه به:

```
while (n < 10)
{
    cout << n;
    n++;
}
```

می‌رسد، اگر n < 10 درست باشد، خطوط:

```
{
    cout << n;
    n++;
}
```

را اجرا می‌کند؛ در صورتی که پس از اجرای این خطوط، هنوز n < 10 درست باشد، همین خطوط را دوباره اجرا می‌کند؛ و این روند آن‌قدر ادامه پیدا می‌کند تا دیگر n < 10 درست نباشد. while در زبان انگلیسی یعنی «تا زمانی که» و:

```
while (n < 10)
```

یعنی تا زمانی که n از ۱۰ کم‌تر است.

مثال: برنامه‌ای می‌نویسیم که یک عدد از کاربر بگیرد، اگر این عدد ۱۰ نبود، دوباره یک عدد بگیرد و آن‌قدر این کار تکرار شود تا کاربر عدد ۱۰ را وارد کند.

```
// Program ID: 0610
#include <iostream>
using namespace std;

int main()
{
    int n = 0;
    while (n != 10)
        cin >> n;
}
```

خروجی:

```
2
5
3
125
10
```

اعداد ۲، ۵، ۳، ۱۲۵ و ۱۰ وارد کردیم و اجرای برنامه هنگامی تمام شد که ۱۰ را وارد کردیم. در حقیقت:

```
while (n != 10)
    cin >> n;
```

یعنی تا زمانی که مقدار n، ۱۰ نیست، cin >> n; را اجرا کن.

این برنامه، cin.ignore(2); ندارد. دستور cin.ignore(1); در پایان برنامه‌ها، سبب می‌شود که برنامه، منتظر فشار کلید Enter صفحه‌کلید از سوی کاربر بماند و کاربر، با اختیار خود، پنجرهٔ خروجی را ببندد. به این ترتیب،

این دستور مانع از بسته‌شدن فوری پنجرهٔ کنسول می‌گردد. اما در این برنامه، چنین توقفی نیاز نیست، زیرا کاربر، با اختیار خود، ۱۰ را وارد می‌کند تا برنامه بسته شود.

برنامهٔ مثال بالا را می‌توان نوعی برنامه دانست که تا رمز خاصی را به آن ندهیم، تمام نمی‌شود. رمز ما ۱۰ است. اگر کاربر این رمز را نداند، باید با آزمایش و خطا آن را پیدا کند. با استفاده از فرایندی که در برنامهٔ پیشین موجود است، می‌توانیم برای برنامه‌های خود رمز بگذاریم، تا هر کسی نتواند آن‌ها را اجرا کند.

تمرین

۱) در کد زیر، بدون این که عملکرد کد تغییر کند، حلقهٔ for را به حلقهٔ while تبدیل کنید:

```
int sum = 0;
for (int i = 1; i <= 10; i++)
    sum += i;
```

دستور break

دستور break را می‌توان در درون یک حلقه به کار برد. هنگامی که این دستور اجرا می‌شود، اجرای حلقه، فوراً پایان می‌یابد، و به‌اصطلاح، حلقه شکسته می‌شود. break در زبان انگلیسی به معنی «شکستن» است.

مثال: برنامه‌ای می‌نویسیم که از کاربر آن‌قدر عدد بگیرد تا کاربر، عدد ۱۹۸۵ را وارد کند و هنگامی که چنین کرد، پیامی چاپ شود:

```
// Program ID: 0620
#include <iostream>
using namespace std;

int main()
{
    while (true)
    {
        int n;
        cin >> n;
        if (n == 1985)
            break;
    }
    cout << "Election comes next Saturday, and ";
    cout << "we're going to have a torchlight procession in the evening, ";
    cout << "no matter who wins.";
    cin.ignore(2);
    return 0;
}
```

خروجی:

```
45
1900
1980
1985
Election comes next Saturday, and we're going to have a torchlight
procession in the evening, no matter who wins.
```

چندین عدد را وارد کرده‌ایم. با وارد کردن ۱۹۸۵، پیامی چاپ شده است که گویی قرار است محرمانه باشد. حلقهٔ:

```
while (true)
{
    int n;
    cin >> n;
    if (n == 1985)
        break;
}
```

یک حلقهٔ بی‌نهایت است، یعنی دستورات:

```
int n;
cin >> n;
if (n == 1985)
    break;
```

را بی‌نهایت بار اجرا می‌کند؛ مگر این که شکسته شود. در حقیقت، طبق آنچه در بخش‌های پیشین گفتیم، اجرای دستورات حلقه، وقتی متوقف می‌گردد که گزارهٔ پس از while، که در اینجا true است، نادرست شود (یا باشد). اما در اینجا، گزارهٔ پس از while، هیچ‌گاه نادرست نمی‌شود. از این رو، اجرای دستورات حلقه نیز، هیچ‌گاه پایان پیدا نمی‌کند، مگر این که حلقه شکسته شود. break، حلقه را می‌شکند. اگر کاربر عدد ۱۹۸۵ را وارد کند، دستور break اجرا و حلقه شکسته می‌شود. پس از شکسته‌شدن حلقه، دستورهای پس از آن، یعنی:

```
cout << "Election comes next Saturday, and ";
cout << "we're going to have a torchlight procession in the evening, ";
cout << "no matter who wins.";
```

اجرا می‌شوند. اگر کاربر نداند چه عددی باید وارد کند، ممکن است مدت‌ها معطل شود. بدون شکسته‌شدن حلقه، کاربر نمی‌تواند به طور معمول، پیام سرّی‌ای را که cout << ها چاپ می‌کنند، ببیند. به این ترتیب، چنین برنامه‌ای را می‌توان برای ارسال یک پیام محرمانه به کار گرفت.

البته برنامهٔ بالا، امنیت کافی ندارد. ممکن است حتی با بازکردن **فایل اجرایی** آن، در یک ویرایشگر ساده بتوان پیغام محرمانه را پیدا کرد:

برای بالابردن امنیت برنامه، باید متن به صورتی غیر مستقیم ساخته و نمایش داده شود تا مستقیماً درون فایل اجرایی قرار نگیرد. حتی در این صورت نیز، ممکن است بتوان، با دستکاری فایل اجرایی نهایی، به روشی، از حلقهٔ بی‌نهایت عبور کرد تا پیغام محرمانه چاپ گردد. بنابراین، برای درج پیام‌های محرمانه، یک ساختار ساده جوابگو نیست.

دستور `break` را درون حلقهٔ `for` نیز می‌توان به کار برد.

سوال

۱) آیا می‌توان با `break` یا ترکیبی از چند `break`، از درون دو حلقهٔ بی‌نهایت تو در تو مانند:

```
while (true)
    while (true)
    { ... }
```

بیرون آمد (با شکستن هر دو حلقه)؟

مهارت برنامه‌نویسی

متن برنامه

هنگامی که یک برنامه را می‌نویسیم و سپس آن را بیلد (کامپایل) می‌کنیم، یک فایل اجرایی (در ویندوز با پسوند exe.) ساخته می‌شود. این فایل، مستقل است؛ یعنی برای اجراشدن، به کامپایلر نیاز ندارد. همین فایل است که به عنوان برنامهٔ ما، در بین کاربران توزیع خواهد شد.

«برنامه» دو معنا دارد:

۱) فایل اجرایی (در ویندوز با پسوند exe.)، که از کامپایل‌شدن متن ++C به وجود می‌آید.
۲) متن ++C برنامه، که همان خطوط و دستورات برنامه است و به آن **سورس‌کد**[۱۰۸] (یا فقط کد) نیز می‌گویند.

در این کتاب، بیشتر معنی دوم مورد نظر است؛ زیرا بحث ما در مورد متن ++C است؛ اما در جاهایی که صحبت از برنامه‌نویسی نیست، معنای اول مورد نظر است، و البته، در این موارد، به جای «برنامه»، ممکن است از اصطلاحات «برنامهٔ کاربردی»، «اپلیکیشن[۱۰۹]»، یا «نرم‌افزار[۱۱۰]» نیز استفاده شود.

با استفاده از یک کامپایلر، می‌توان کد را به فایل اجرایی تبدیل کرد. اما تبدیل‌کردن فایل اجرایی به کد، تقریباً ناممکن است.

کلمات کلیدی

برخی کلمات در زبان ++C، کلیدی هستند. **کلمات کلیدی**[۱۱۱]، به طور معمول، در متن ویرایشگر آی‌دی‌ای، رنگ متفاوتی دارند. در زیر، لیستی از کلمه‌های کلیدی آورده شده است:

```
int
float
bool
if
while
for
class
template
__asm
this
new
```

کلمه‌های کلیدی دیگری نیز وجود دارند که کم‌کم با آن‌ها آشنا خواهیم شد.

در آی‌دی‌ای ویژوال‌استودیو، تعدادی کلمهٔ غیر کلیدی هست که رنگ کلمه‌های کلیدی را به خود می‌گیرند. برای مثال، `array`، کلمهٔ کلیدی نیست؛ اما به رنگ آن‌هاست. علت این تغییر رنگ این است که در پروژه‌هایی با

---

[۱۰۸] source code
[۱۰۹] application
[۱۱۰] software
[۱۱۱] keywords

نوع‌های دیگر، که مخصوص ویژوال‌استودیو هستند، این کلمات، کلیدی محسوب می‌شوند. اما در C++ استاندارد، این کلمات، کلیدی به حساب نمی‌آیند. همچنین، برخی کلمات کلیدی در C++ هستند که ویژوال‌استودیو رنگ آن‌ها را مانند کلمات کلیدی دیگر نشان نمی‌دهد؛ مانند xor. کلماتی هم هستند که در گذشته کلمهٔ کلیدی بوده‌اند، اما در استانداردهای امروزی، دیگر کلمهٔ کلیدی نیستند؛ برای نمونه، می‌توان به این کلمات اشاره کرد:

```
far
near
huge
```

گرچه این‌ها از C++ حذف شده‌اند، اما بهتر است از آن‌ها استفاده نکنیم. علت این است که ممکن است، در مواردی، بعضی از دستورات پیش‌پردازنده[۱۱۲] آن‌ها را حذف کنند. در فصل‌های آینده، دستورات پیش‌پردازنده معرفی خواهند شد.

برنامهٔ زیر، بدون هیچ خطایی در ویژوال‌استودیو اجرا می‌شود:

```
// Program ID: 0630
#include <iostream>
using namespace std;

int main()
{
    int array = 1;
    int gcnew = 2;
    int pascal = 3;
    int __pascal = 4;
    int far = 5;
    int near = 6;
    int huge = 7;
    int property = 8;
    cout << array << gcnew << pascal << far << near << huge << property;
    cin.ignore(1);
}
```

خروجی (ویژوال‌استودیو):

```
1235678
```

خروجی (کیوت‌کریپتور): خطا دارد.

کیوت‌کریپتور در:

```
int __pascal = 4;
```

خطا می‌گیرد و برنامه اجرا نمی‌شود. به یاد داشته باشید که هیچ‌گاه نباید از کلمه‌های کلیدی، به عنوان متغیر استفاده کنید.[۱۱۳]

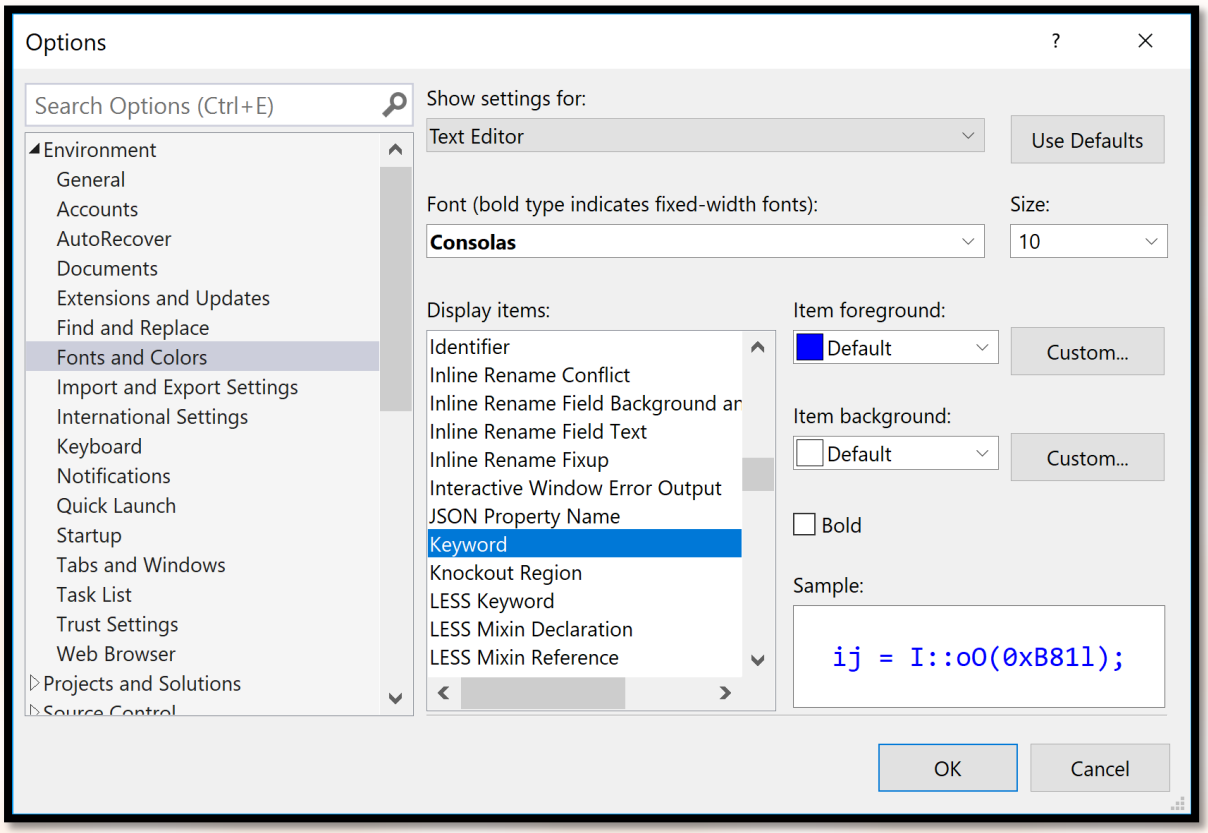
[۱۱۲] preprocessor

[۱۱۳] Visual C++ 2010 Express از به‌کاربردن کلمهٔ کلیدی default به عنوان متغیر ایرادی نمی‌گیرد؛ که ایرادی در این کامپایلر است و در نسخه‌های جدیدتر برطرف شده است. ممکن است در کامپایلرها، ایرادات مشابهی موجود باشند.

رنگ متن برنامه

رنگ، در متن برنامه، اهمیت اجرایی ندارد اما می‌تواند در نوشتن برنامه، به برنامه‌نویس، کمک کند. اغلب آی‌دی‌ای‌ها، به کاربر اجازه می‌دهند که رنگ‌آمیزی کدها را تغییر دهد. برای تنظیمات مربوط به رنگ، در ویژوال‌استودیو، از طریق منوها، این مسیر را طی کنید:

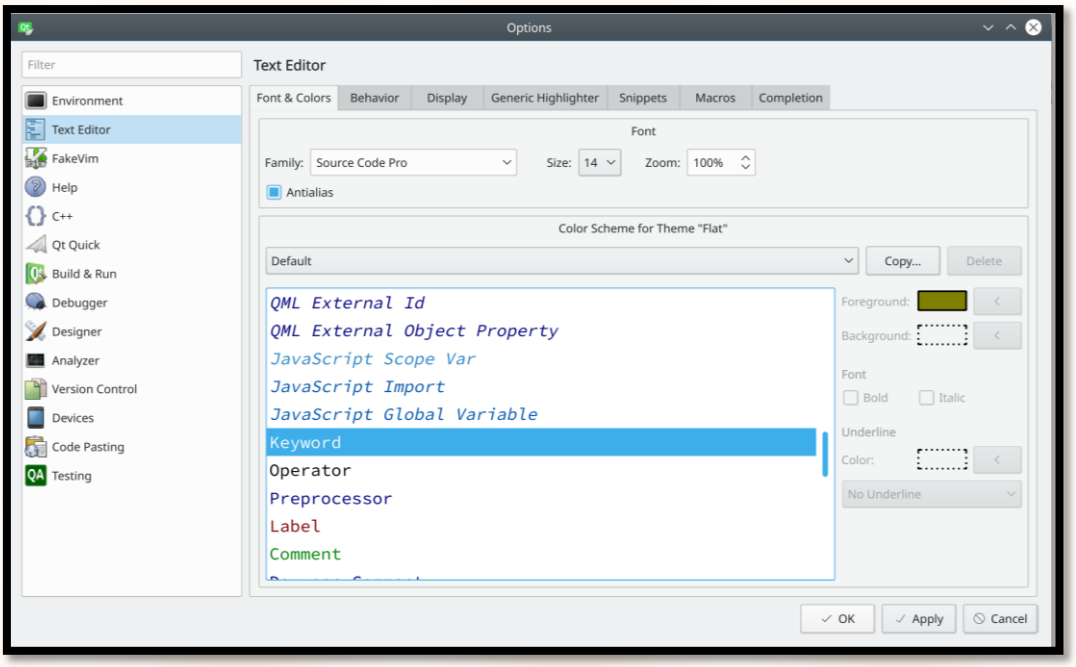
Tools → Options... → Environment ▱ Fonts and Colors:



برای مثال، در تصویر بالا، رنگ Keyword (یعنی کلمهٔ کلیدی) نشان داده شده است. در این پنجره، می‌توانید این رنگ را تغییر دهید. بالای Keyword، Identifier (یعنی شناسه) قرار دارد. **شناسه، به نام تابع‌ها، متغیرها و... که خود ما انتخاب می‌کنیم، اطلاق می‌شود.** همهٔ این رنگ‌ها را می‌توان تغییر داد.

در کیوت‌کریتور، برای تنظیمات مربوط به رنگ، مسیر زیر را طی کنید:

Tools → Options... → Text Editor:



در آی‌دی‌ای‌ها رنگ پس‌زمینهٔ ویرایشگر را نیز می‌توان تغییر داد. وقتی زمینه، تیره باشد، مردمک چشم بازتر است و بعضی افراد ممکن است (به دلیل عیوب انکساری چشم) در زیر نوشته‌ها سایه ببینند. هنگامی که پس‌زمینه روشن‌تر باشد، مردمک جمع‌تر می‌شود و نور از دریچهٔ باریک‌تری وارد چشم می‌شود. در آی‌دی‌ای‌های قدیمی، پس‌زمینه، به طور پیش‌فرض، تیره بود؛ اما در کامپایلرهای جدید، پس‌زمینه به طور پیش‌فرض روشن است. در مورد رنگ پس‌زمینه باید توصیه‌های پزشکی را در نظر داشت.

مرتب‌نوشتن برنامه‌ها

از موارد مهم در برنامه‌نویسی، مرتب‌نوشتن و داشتن‌یک‌سبک‌مشخص است. لازم است که متن برنامه، هم در ظاهر، و هم در محتوا، به گونه‌ای باشد که امکان خطا را کمینه و درک کدها را، برای برنامه‌نویس، در مراجعات بعدی یا برای برنامه‌نویسان دیگر، بیشینه کند. چند پیشنهاد در مورد نوشتن برنامه:

۱) آکلادها باید کاملاً در یک ستون قرار بگیرند و آنچه بین آن‌هاست، باید به اندازهٔ یک تب[۱۱۴] یا چهار فاصله، جلوتر باشد. مانند این کد:

```
void f(int b)
{
    int a = 2;
    cout << a + b;
}
```

در آی‌دی‌ای‌ها، اغلب، عرض یک تب، به طور پیش‌فرض، به اندازهٔ چهار فاصلهٔ خالی[۱۱۵] است. برخی آی‌دی‌ای‌ها، اجازهٔ تغییر عرض تب را به برنامه‌نویس می‌دهند و بعضی، اجازهٔ جایگزینی خودکار آن با ۴ فاصلهٔ خالی.

[۱۱۴] tab

۲) خط پس از دستوراتی مانند if، باید به اندازهٔ یک تب جلوتر شروع شود، مانند این کد:

```
if (a > 2)
    a = 2;
```

۳) گاه، برای ساده‌ترپیداکردن اجزای کد، می‌توانیم طرح‌های خلاقانه‌ای به کار ببریم:

```
int     limo = 2;
char    limousine = 4;
double  RedLimousine = 8;
```

۴) در دو طرف عملگرهایی مثل «=»، «+» و...، یک فاصلهٔ خالی قرار دهید:

```
int c = a + b;
```

ویرایشگرهای آی‌دی‌ای‌ها ممکن است، خودبه‌خود، برخی از این پیشنهادها را اعمال کنند. توجه داشته باشید که هرچند مرتب‌نوشتن کدها، برای درست‌انجام‌دادن برنامه‌نویسی ضروری است، اما اثری در فایل اجرایی نهایی ندارد، و تنها، یک نظم انسانی و بی‌اثر در ماشین است. ماشین، نه خطا می‌کند و نه نیازی به فهم کد از روی آرایش آن دارد.

کامنت‌ها

گاه لازم است در کنار برخی کدهای برنامه، توضیحاتی بنویسیم تا در مراجعات بعدی، به کمک آن توضیحات، راحت‌تر کدها را درک کنیم. هنگامی که یک برنامه یا بخشی از آن را کامل کردیم، تا مدت کوتاهی، همهٔ جزئیات آن را درک خواهیم کرد. اما اگر مدتی بگذرد، جزئیات و حتی کلیات فراموش خواهند شد[۱۱۶] و هنگام بازگشت، به سختی آن را درک خواهیم کرد. در اینجاست که توضیحات به زبان فارسی یا انگلیسی، می‌توانند کمک بزرگی باشند.

به توضیحاتی که در یک برنامه نوشته می‌شوند و کامپایلر آن‌ها را نادیده می‌گیرد، **کامنت**[۱۱۷] می‌گوییم. در C++، به دو شکل می‌توان کامنت گذاشت: شکل قدیمی، که از C به ارث رسیده است[۱۱۸]؛ و شکل جدید، که مخصوص C++ است.[۱۱۹] در شکل قدیمی، توضیحات بین «/*» و «*/» نوشته می‌شوند. برای مثال:

```
cout << b; /* This is
              a comment */
```

کامپایلر، کل:

```
           /* This is
           a comment */
```

---

[۱۱۵] space

[۱۱۶] می‌توان این فراموشی را در ابعاد دیگر هم در نظر داشت: وقتی برای بار اول یک زبان برنامه‌نویسی جدید را می‌آموزیم با چند ماه دوری از آن، نوشتن برنامه‌های ساده را هم فراموش می‌کنیم. به همین سبب، بدون برنامه‌نویسی مستمر نمی‌توان یک زبان را در ذهن تثبیت کرد.

[۱۱۷] comment

[۱۱۸] C Style Comment

[۱۱۹] C++ Style Comment

را نادیده می‌گیرد؛ انگار اصلاً نوشته نشده باشد. شکل جدید، یک خط را از جایی به بعد حذف می‌کند؛ مانند:

```
cout << b; // This is a comment
```

می‌توانید رنگ کامنت‌ها را تغییر دهید؛ اما رنگ مناسب، رنگی است که جزئی‌ازمتن‌اصلی‌نبودن آن‌ها را نشان دهد؛ برای نمونه، خاکستری کم رنگ.

از کامنت‌ها، می‌توان برای حذف موقت بخش‌هایی از برنامه استفاده کرد. به این کار، **حذف کامنتی**[۱۲۰] می‌گوییم. ریزه‌کاری‌هایی در مورد کامنت‌های تودرتو هست که در هنگام نیاز با آزمایش و خطا با آن‌ها آشنا می‌شوید. حال، به مثال زیر دقت کنید:

```
// Program ID: 0640
#include <iostream>    // a comment
using namespace std;

//--------------------------------------------------
/*
void f(int& b)
{
    b = 2;
    // cout<< b; // This is a comment
}
   */
   //@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
int main()
{
    // //f(1);
    // /* ff */
    ////////////////////////////
    ///
    cout << "hello and /*hallo*/ and //hello";
    cin.ignore(1);
}
```

خروجی:

```
hello and /*hallo*/ and //hello
```

می‌بینید که تابع f، موقتاً حذف شده است؛ با کامنتگذاری می‌توان بخش‌هایی از برنامه را، موقتاً حذف کرد. همچنین، در برنامهٔ بالا، سه خط /// را با رنگ کمی متمایز می‌بینید[۱۲۱]. آی‌دی‌ای ویژوال‌استودیو، کامنت‌هایی را که دقیقاً با سه «/» شروع می‌شوند، نوع جدیدی از کامنت به حساب می‌آورد، و به همین سبب، رنگشان را (کمی) متمایز می‌کند.

نکتهٔ مهم این است که در رشتهٔ `"hello and /*hallo*/ and //hello"` هیچ قسمتی، کامنت به حساب نمی‌آید. برای این که امکان چاپ */ و // از بین نرود، امکان حذف‌کامنتی‌کردن درون رشته‌ها نیست.

در بسیاری از آی‌دی‌ای‌های جدید، می‌توان کامنت‌ها را فارسی نیز نوشت؛ اما اگر چنین کنیم، لازم است فایل را در فرمت یونیکد ذخیره کنیم، که به طور معمول، هنگام ذخیرهٔ متن برنامه، تذکری در این مورد، از سوی آی‌دی‌ای نشان داده می‌شود.

---

۱۲۰ comment out

۱۲۱ تشخیص تمایز سخت است.

خطا و اخطار

در بسیاری از موارد، در اولین اجرای یک برنامه، که به تازگی آن را تکمیل کرده‌ایم، کامپایلر، ایرادهایی می‌گیرد. این ایرادها، دو صورت دارند:

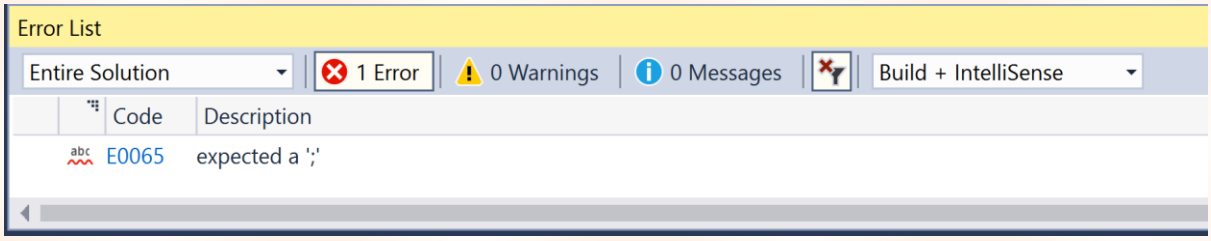
۱) خطا[۱۲۲] که مانع کامپایل‌شدن و اجرای برنامه می‌گردد.

۲) اخطار[۱۲۳] که مانع اجرای برنامه نیست و تنها تذکر است.

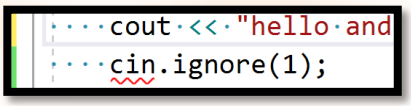
کامپایلرها، معمولاً لیست خطاها و اخطارهای موجود را نشان می‌دهند. اگر در ویژوال‌استودیو، این لیست قابل‌مشاهده یا فعال نیست، برای فعال کردن آن؛ مسی زیرر را از طریق منوها طی کنید:

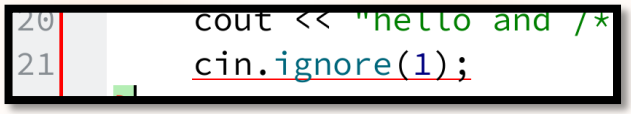
View → Error List

این پنجره اضافه می‌شود:



در تصویر بالا، تنها، تب خطاها فعال است. با کلیک بر تب اخطارها، می‌توانید آن‌ها را نیز فعال کنید. در ویژوال‌استودیو و کیوت‌کریتور، برخی خطاها، (توسط آنچه در ویژوال‌استودیو IntelliSense نامیده می‌شود)، قبل از بیلد یا اجرای برنامه، تشخیص داده می‌شوند. این یک امکان در این دو آی‌دی‌ای است و ممکن است در همهٔ آی‌دی‌ای‌ها، مشابه آن، موجود نباشد. ویژوال‌استودیو و کیوت‌کریتور خطاهای موجود را در هنگام تایپ متن برنامه، و قبل از کامپایل، به طور خودکار پیدا می‌کنند و زیر آن‌ها خط قرمز می‌کشند:





چنان‌که گفته شد، این خطاها، قبل از بیلد و اجرا، ظاهر می‌شوند؛ و در ویژوال‌استودیو، (اگر گزینهٔ نمایش آن‌ها را انتخاب کرده باشید) در پنجرهٔ Error List نیز آورده می‌شوند. خطاهای IntelliSense، و مشابه آن در کیوت‌کریتور، پیشبینی خطاهای کامپایلر در زمان کامپایل‌کردن هستند و ممکن است در مواردی، اشتباه هم باشند؛ یعنی چیزهایی به عنوان خطا مشخص شوند که کامپایلر، آن‌ها را خطا نمی‌داند؛ و یا بر عکس، خطاهایی از قلم بیفتند.

---

[۱۲۲] error

[۱۲۳] warning

اجرای قدم‌به‌قدم

پیش‌تر گفتیم که چگونه می‌توان یک برنامه را به صورت گام‌به‌گام اجرا کرد. هدف ما این بود که مشخص شود چگونه خطوط برنامه، به ترتیب، اجرا می‌شوند؛ تا تصوری از نحوهٔ اجرای برنامه پیدا کنید. اما هدف اصلی از اجرای قدم‌به‌قدم، برطرف‌کردن اشتباهات **منطقی** برنامه است:

خطا یا اخطار وقتی داده می‌شود که از نظر دستور زبان ++C، اشتباهی کرده باشیم. اما در بسیاری از موارد، از نظر دستوری، اشتباهی نکرده‌ایم؛ اما برنامه آن طور که می‌خواهیم کار نمی‌کند. برای مثال، تصور کنید که یک برنامه نوشته‌ایم که قرار است اعداد اول بین ۱ تا ۱۰۰ را چاپ کند؛ اما عدد ۹ را نیز چاپ می‌کند، که اول نیست. پس برنامه، اشکال دارد. به این گونه از اشکال‌ها، **اشکالات منطقی** می‌گویند و کامپایلر، آن‌ها تشخیص نمی‌دهد. یکی از راه‌های خوب، برای برطرف کردن خیلی از اشکالات منطقی، اجرای قدم‌به‌قدم است.

در آی‌دی‌ای‌های امروزی، هنگام اجرای قدم‌به‌قدم، با نگه‌داشتن‌ماوس بر یک متغیر، می‌توان مقدار آن را دید. برای نمونه، در تصویر زیر، اشاره‌گر ماوس بر یک متغیر قرار گرفته است:

می‌بینید که مقدار متغیر، نشان داده شده است. اگر بر علامت (یا مشابهش در کیوت‌کریتور) کلیک کنید، مقدار متغیر، همواره به طور ثابت نشان داده می‌شود.

ویژوال‌استودیو و کیوت‌کریتور، پنجره‌هایی مثل Locals و Locals and Expressions نیز دارند که مقدار متغیرها را نشان می‌دهند.

نام‌گذاری متغیرها و تابع‌ها

چنان‌که پیش‌تر نیز اشاره شد، به نامی که برای تابع و متغیر و... توسط برنامه‌نویس انتخاب می‌شود، شناسه[۱۲۴] می‌گویند. لازم نیست همهٔ قوانین حاکم بر انتخاب شناسه‌ها را بگوییم. قوانینی مثل این که نباید با عدد شروع شوند، نباید تکراری باشند، نباید کلمهٔ کلیدی باشند و...؛ زیرا در صورت نقض این قوانین، کامپایلر، با خطاگرفتن، تذکر می‌دهد. ++C، بر خلاف زبان‌هایی مانند BASIC، حساس به بزرگی یا کوچکی حروف[۱۲۵] است؛ یعنی در ++C، شناسه‌های `Id` و `id` و `ID` و `iD` کاملاً متفاوت هستند؛ و برای نمونه، بر عکس `if`، `If` یک کلمهٔ کلیدی نیست.

---

[۱۲۴] identifier

[۱۲۵] case-sensitive

شناسه‌هایی که تا به حال در مثال‌ها به کار برده‌ایم، معمولاً، یک حرف بیشتر نداشتند. اما یک شناسه، می‌تواند طولانی‌تر باشد. در گذشته، نام‌های انتخابی، کوتاه بودند و معنی آن‌ها، برای کسی که بار اول آن‌ها را می‌دید، روشن نبود مانند getch.

نام‌هایی که امروزه به عنوان شناسه انتخاب می‌شوند، کامل‌تر هستند، مانند SetFilePointer یا CreateWindow یا is_file_created.

چند روش برای ساختن و انتخاب شناسه وجود دارد:

۱) **بزرگ‌گرفتن حرف اول کلمات**: مثل SetFilePointer یا MinimizeAllWindows. این روش بیشتر در مورد تابع‌ها به کار می‌رود.

۲) **نمادگذاری شترمانند**[۱۲۶]: در این روش، اولین کلمه، با حروف کوچک و حرف اول مابقی کلمه‌ها، با حروف بزرگ نوشته می‌شود؛ مانند newString یا myCarClass یا itsLength.

۳) **استفاده از خط زیرین**[۱۲۷]: نام‌هایی که توضیح بیشتر می‌خواهند، و کاربرد زیادی ندارند، با این روش انتخاب می‌شوند. در این روش، چند کلمه که با حروف کوچک نوشته شده‌اند، با خط زیرین، یعنی «_»، به هم متصل می‌گردند؛ مانند is_the_window_maximized یا number_of_instances.

۴) **نمادگذاری مجارستانی**: در این روش نمادگذاری، متغیرهای دارای نوع int، با i شروع می‌شوند، متغیرهای دارای نوع long، با l شروع می‌شوند و... . برای مثال:

```
int ia;
bool bx;
long lr;
```

این روش برای C++ چندان مناسب نیست، چون کسی که با برنامه آشنایی کمی دارد، با دیدن چند متغیر که مثلاً با i شروع می‌شوند، سردرگم می‌گردد.

شاید در این کتاب، کمتر از این روش‌ها استفاده کنیم. برنامه‌های این کتاب، با هدف آموزش زبان C++ نوشته شده‌اند و کوتاه و ساده‌اند. اما نداشتن روش نام‌گذاری مشخص، در برنامه‌های طولانی، بدون شک مسأله‌ساز خواهد بود.

برنامه بنویسیم

[۱۲۶] camel-notation

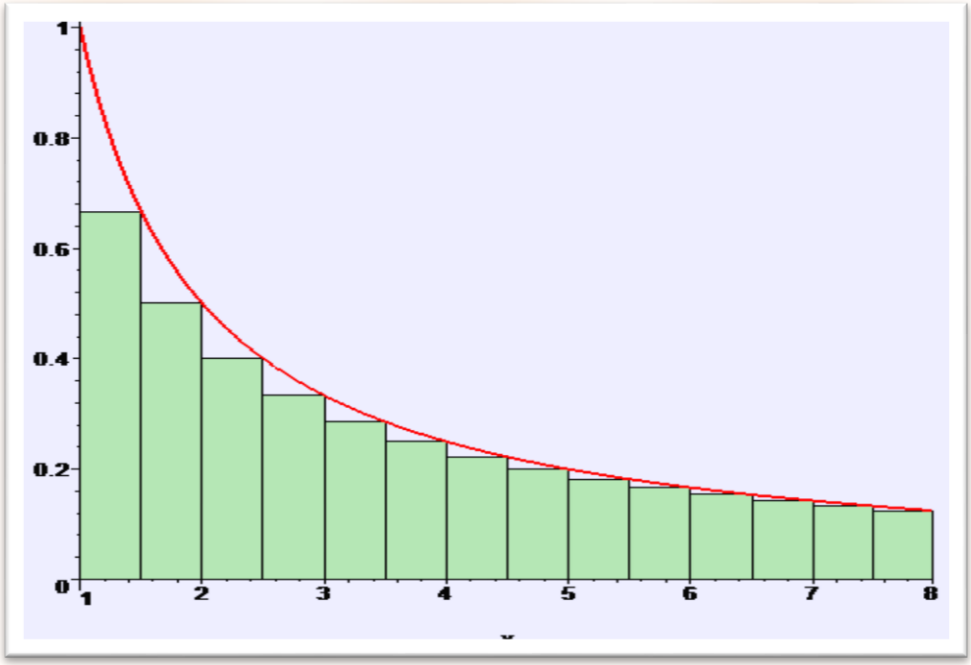
[۱۲۷] underscore

- یک جدول را که مانند:

۲	۷	۶
۹	۵	۱
۴	۳	۸

در خانه‌های آن، اعداد ۱ تا ۹ قرار گرفته‌اند و جمع اعداد، از همه طرف مساوی است، مربع نقره‌ای می‌نامیم. برنامه‌ای بنویسید که یک مربع نقره‌ای ۴ در ۴ را مشخص کند. روشی که برای این برنامه به کار می‌گیرید، باید کلی باشد؛ به طوری که، با تغییراتی، بتوانید برنامه را برای پیداکردن یک مربع نقره‌ای ۵ در ۵ نیز به کار گیرید.

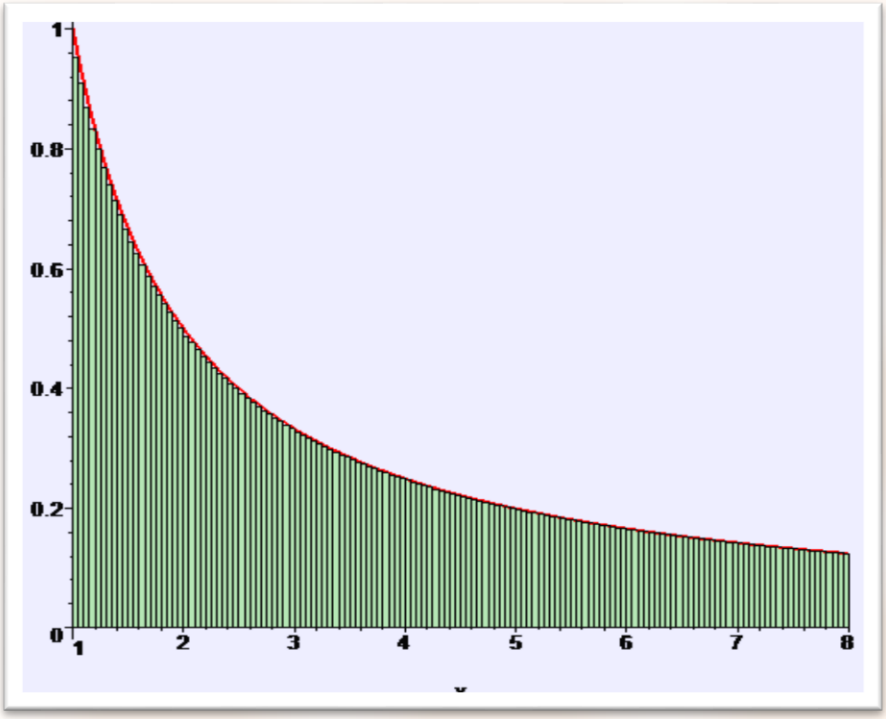
- می‌خواهیم برنامه‌ای بنویسیم که، با تقریب خوبی، $\ln(8)$ را حساب کند. به تصویر زیر نگاه کنید:



در این تصویر، نمودار $y = \frac{1}{x}$، از $x = 1$ تا $x = 8$ رسم شده است. $\ln(8)$، مساحت زیر این نمودار است. می‌خواهیم این مساحت را، با مجموع مساحت مستطیل‌هایی که در زیر نمودار رسم شده‌اند، تقریب بزنیم. در زیر نمودار، ۱۴ مستطیل قرار دارد.

الف) برنامه‌ای بنویسید که مجموع مساحت این ۱۴ مستطیل را حساب کند.

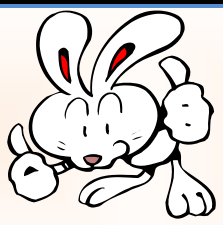
ب) فرض کنید مساحت زیر نمودار را، به جای ۱۴ مستطیل، با ۱۴۰ مستطیل بپوشانیم:



برنامه‌ای بنویسید که مجموع مساحت‌های این ۱۴۰ مستطیل را حساب و چاپ کند. ln(۸) را با ماشین‌حساب (مثلاً ماشین‌حساب موجود در ویندوز/اوبونتو)، حساب و با مقدار چاپ‌شده مقایسه کنید.

- هر عدد صحیح، مانند n، را می‌توان به صورت حاصل‌ضرب اعداد اول نوشت. برنامه‌ای بنویسید که n را بگیرد و آن را، به صورت حاصل‌ضرب اعداد اول بنویسد. برای مثال، ۱۲ را بگیرد و بنویسد:

```
12 = 2*2*3
```



فصل سوم – نوع‌ها و متغیرها

در این فصل، به طور تخصصی، به نوع‌ها، متغیرها[۱۲۸]، ثابت‌ها[۱۲۹] و ناپایاها[۱۳۰] می‌پردازیم. از آنجا که وارد جزئیات بیتی حافظهٔ اشیاء و نیز، معرفی ساختارها، خواهیم شد، ممکن است درک مطالب برای خواننده سخت‌تر و خسته‌کننده‌تر باشد. اما این فصل، یک فصل پایه‌ای و اساسی است و فهم دقیق مطالب آن ضروری است.

نوع‌های اصلی و آرایش بیتی

نوع‌های اصلی

به نوع‌های ابتدایی[۱۳۱] که در C++، به صورت کلمات کلیدی ارائه شده‌اند، **نوع‌های اصلی**[۱۳۲] می‌گوییم. بسیاری از نوع‌های اصلی را در این جدول می‌بینید:

کاربرد	تعداد بایت‌ها	معادل	نوع
عبارت بولی	۱		bool
کاراکتر	۱	__int8	char
عدد اعشاری	۸	long float	double
عدد اعشاری	۴		float
عدد صحیح	۴	signed signed int __int32 signed bool	int
عدد صحیح	۴	long int signed long signed long int	long
عدد اعشاری	۸ یا ۱۲[۱۳۳]		long double
عدد صحیح	۸	signed long long signed long long int __int64	long long
عدد صحیح	۲	short int signed short signed short int __int16	short
کاراکتر	۱	signed __int8	signed char
کاراکتر بی علامت	۱	unsigned __int8	unsigned char
عدد صحیح نامنفی	۴	unsigned unsigned bool unsigned __int32	unsigned int

[۱۲۸] variables
[۱۲۹] constants
[۱۳۰] temporaries
[۱۳۱] primitive data type
[۱۳۲] basic types

[۱۳۳] در ویژوال‌استودیو ۸ و در کیوت‌کریتور ۱۲.

عدد صحیح نامنفی	۴	unsigned long int	unsigned long
عدد صحیح نامنفی	۸	unsigned long long int unsigned __int64	unsigned long long
عدد صحیح نامنفی	۲	unsigned short int unsigned __int16	unsigned short

با بعضی از این نوع‌ها در گذشته آشنا شده‌اید. این جدول چیز زیادی بیشتر از آنچه در گذشته گفته شد، ندارد. می‌بینید که بیشتر نوع‌ها، «عدد صحیح» هستند. در این جدول، نوع‌هایی که با رنگ نارنجی نشان داده شده‌اند، تنها در ویژوال‌استودیو قابل‌استفاده‌اند و نمی‌توان آن‌ها را در کیوت‌کریتور به کار برد. (ادامه دارد ...)