برنامه نویسی به زبان C++

مهندس امیرحسین شکوهی

معرفی ساختاری زبان C++

پس از نوشتن یک الگوریتم باید آن را با استفاده از یک زبان برنامه نویسی تبدیل به یک برنامه قابل اجرا برای کامپیوتر نماییم. این زبانها به سه دسته کلی تقسیم میگردند :

۱- زبان ماشین (سطح پایین) : این زبان مستقیما با صفر و یک نوشته می شود و بدون هیچ واسطه ای برروی کامپیوتر قابل اجرا است. طراحان سخت افزار هر کامپیوتر، زبان ماشین خاص خود را برای آن ماشین طراحی می نمایند. به همین دلیل هر برنامه ای که به زبان ماشین نوشته شود، فقط برروی همان ماشین خاص کار می کند، بهمین دلیل برنامه های نوشته شده به زبان ماشین را غیر قابل حمل می نامند. از طرف دیگر یادگیری این زبان بسیار مشکل بوده و برنامه نویسی با آن نیز بسیار سخت است و همچنین احتمال بروز خطا نیز در آن زیاد است.

۲- زبان اسمبلی : این زبان شکل ساده تر زبان ماشین است، بدین صورت که برای هر دستورالعمل زبان ماشین، یک اسم نمادین انتخاب شده است (مانند دستور ADD بجای کد دودویی دستورالعمل جمع) که بخاطر سپردن و برنامه نویسی با آنها برای انسانها ساده تر است. اما این برنامه ها برای ماشین قابل فهم نیست و باید قبل از اجرا شدن توسط برنامه مترجمی بنام اسمبلر به زبان ماشین تبدیل شود. توجه کنید که از آنجا که هر دستور زبان اسمبلی معادل یک دستور زبان ماشین است، این زبان نیز وابسته به ماشین می باشد و برنامه های نوشته شده به این زبان فقط برروی همان کامپیوتری که برای آن نوشته شده اند قابل اجرا است. علاوه براین کار با این زبانها هنوز هم نسبتا مشکل بود و فقط متخصصین کامپیوتر قادر به استفاده از آنها بودند.

۳- زبانهای سطح بالا : دستورالعملهای این زبانها بسیار نزدیک به زبان انسانها (بطور مشخص زبان انگلیسی) می باشد و بهمین دلیل برنامه نویسی به آنها بسیار ساده تر بوده و می توان الگوریتمها را به راحتی به این زبانها تبدیل کرد. از آنجا که این زبانها به هیچ ماشین خاصی وابسته نیستند، برنامه های نوشته شده با این زبانها (تا حد زیادی) قابل حمل می باشند. مثالهایی از این زبانها عبارتند از :

- بیسیک (Basic): برای کاربردهای آموزشی
- فرترن (Fortran) : برای کاربردهای علمی و مهندسی
- پاسکال (Pascal) : برای کاربردهای آموزشی و علمی

و بالاخره زبان برنامه نویسی C که درمورد آن بیشتر صحبت خواهیم کرد. البته برنامه های نوشته شده به این زبانها ابتدا باید به زبان ماشین ترجمه شوند تا برروی کامپیوتر قابل اجرا باشند. برای ترجمه این زبانها از کامپایلرها و یا مفسرها استفاده می شود.

تاریخچه C

برای بررسی تاریخچه زبان C باید به سال ۱۹۶۷ بازگردیم که مارتین ریچاردز زبان BCPL را برای نوشتن نرم افزارهای سیستم عامل و کامپایلر در دانشگاه کمبریج ابداع کرد. سپس در سال ۱۹۷۰ کن تامپسون زبان B را بر مبنای ویژگیهای زبان

BCPL نوشت و از آن برای ایجاد اولین نسخه های سیستم عامل Unix در آزمایشگاههای بل استفاده کرد. زبان C در سال ۱۹۷۲ توسط دنیس ریچی از روی زبان B و BCPL در آزمایشگاه بل ساخته شد و ویژگیهای جدیدی همچون نظارت بر نوع داده ها نیز به آن اضافه شد. ریچی از این زبان برای ایجاد سیستم عامل Unix استفاده کرد اما بعدها اکثر سیستم عاملهای دیگر نیز با همین زبان نوشته شدند. این زبان با سرعت بسیاری گسترش یافت و چاپ کتاب "The C Programming Language" در سال ۱۹۷۸ توسط کرنیگان و ریچی باعث رشد روزافزون این زبان در جهان شد.

متاسفانه استفاده گسترده این زبان در انواع کامپیوترها و سخت افزارهای مختلف باعث شد که نسخه های مختلفی از این زبان بوجود آید که با یکدیگر ناسازگار بودند. در سال ۱۹۸۳ انستیتوی ملی استاندارد آمریکا (ANSI) کمیته ای موسوم به X3J11 را را مامور کرد تا یک تعریف فاقد ابهام و مستقل از ماشین را از این زبان تدوین نماید.در سال ۱۹۸۹ این استاندارد تحت عنوان ANSI C به تصویب رسید و سپس در سال ۱۹۹۰، سازمان استانداردهای بین المللی (ISO) نیز این استاندارد را پذیرفت و مستندات مشترک آنها تحت عنوان ANSI/ISO C منتشر گردید.

در سالهای بعد و با ظهور روشهای برنامه نویسی شی گرا نسخه جدیدی از زبان C بنام C++ توسط بیارنه استراوستروپ در اوایل ۱۹۸۰ در آزمایشگاه بل توسعه یافت. در C++ علاوه بر امکانات جدیدی که به زبان C اضافه شده است، خاصیت شی گرایی را نیز به آن اضافه کرده است.

با گسترش شبکه و اینترنت، نیاز به زبانی احساس شد که برنامه های آن بتوانند برروی هر ماشین و هر سیستم عامل دلخواهی اجرا گردد. شرکت سان مایکروسیستمز در سال ۱۹۹۵ میلادی زبان Java را برمبنای C و C++ ایجاد کرد که هم اکنون از آن در سطح وسیعی استفاده می شود و برنامه های نوشته شده به آن برروی هر کامپیوتری که از Java پشتیبانی کند(تقریبا تمام سیستمهای شناخته شده) قابل اجرا می باشد. شرکت مایکروسافت در رقابت با شرکت سان، در سال ۲۰۰۲ زبان جدیدی بنام C# (سی شارپ) را ارائه داد که رقیبی برای Java بشمار می رود.

برنامه نویسی ساخت یافته

در دهه ۱۹۶۰ میلادی توسعه نرم افزار دچار مشکلات عدیده ای شد. در آن زمان سبک خاصی برای برنامه نویسی وجود نداشت و برنامه ها بدون هیچگونه ساختار خاصی نوشته می شدند. وجود دستور پرش (goto) نیز مشکلات بسیاری را برای فهم و درک برنامه توسط افراد دیگر ایجاد می کرد، چرا که جریان اجرای برنامه مرتبا دچار تغییر جهت شده و دنبال کردن آن دشوار می گردید. لذا نوشتن برنامه ها عملی بسیار زمان بر و پرهزینه شده بود و معمولا اشکال زدایی, اعمال تغییرات و گسترش برنامه ها بسیار مشکل بود. فعالیتهای پژوهشی در این دهه باعث بوجود آمدن سبک جدیدی از برنامه نویسی بنام روش ساختیافته گردید؛ روش منظمی که باعث ایجاد برنامه هایی کاملا واضح و خوانا گردید که اشکال زدایی و خطایابی آنها نیز بسیار ساده تر بود.

اصلی ترین نکته در این روش عدم استفاده از دستور پرش (goto) است. تحقیقات بوهم و ژاکوپینی نشان داد که می توان هر برنامه ای را بدون دستور پرش و فقط با استفاده از ۳ ساختار کنترلی ترتیب، انتخاب و تکرار نوشت.

ساختار ترتیب، همان اجرای دستورات بصورت متوالی (یکی پس از دیگری) است که کلیه زبانهای برنامه نویسی در حالت عادی بهمان صورت عمل می کنند.
ساختار انتخاب به برنامه نویس اجازه می دهد که براساس درستی یا نادرستی یک شرط، تصمیم بگیرد کدام مجموعه از دستورات اجرا شود.
ساختار تکرار نیز به برنامه نویسان اجازه می دهد مجموعه خاصی از دستورات را تا زمانیکه شرط خاصی برقرار باشد، تکرار نماید.
هر برنامه ساختیافته از تعدادی بلوک تشکیل می شود که این بلوکها به ترتیب اجرا می شوند تا برنامه خاتمه یابد(ساختار ترتیب). هر بلوک می تواند یک دستورساده مانند خواندن، نوشتن یا تخصیص مقدار به یک متغیر باشد و یا اینکه شامل دستوراتی باشد که یکی از ۳ ساختار فوق را پیاده سازی کنند. نکته مهم اینجاست که درمورد دستورات داخل هر بلوک نیز همین قوانین برقرار است و این دستورات می توانند از تعدادی بلوک به شرح فوق ایجاد شوند و تشکیل ساختارهایی مانند حلقه های تودرتو را دهند.
نکته مهم اینجاست که طبق قوانین فوق یک حلقه تکرار یا بطور کامل داخل حلقه تکرار دیگر است و یا بطور کامل خارج آن قرار می گیرد و هیچگاه حلقه های روی هم افتاده نخواهیم داشت.
از جمله اولین تلاشها در زمینه ساخت زبانهای برنامه نویسی ساختیافته، زبان پاسکال بود که توسط پروفسور نیکلاس ویرث در سال ۱۹۷۱ برای آموزش برنامه نویسی ساختیافته در محیطهای آموزشی ساخته شد و بسرعت در دانشگاهها رواج یافت. اما بدلیل نداشتن بسیاری از ویژگیهای مورد نیاز مراکز صنعتی و تجاری در بیرون دانشگاهها موفقیتی نیافت.
کمی بعد زبان C ارائه گردید که علاوه بر دارا بودن ویژگیهای برنامه نویسی ساختیافته بدلیل سرعت و کارایی بالا مقبولیتی همه گیر یافت و هم اکنون سالهاست که بعنوان بزرگترین زبان برنامه نویسی دنیا شناخته شده است.

مراحل اجرای یک برنامه C

برای اجرای یک برنامه C ابتدا باید آن را نوشت. برای اینکار می توان از هر ویرایشگر متنی موجود استفاده کرد و سپس فایل حاصل را با پسوند .C ذخیره نمود (فایلهای C++ با پسوند .CPP ذخیره می گردند). به این فایل، کد مبدا (source code) گفته می شود. مرحله بعدی تبدیل کد مبدا به زبان ماشین است که به آن کد مقصد (object code) گفته می شود. همانطورکه قبلا نیز گفته شد برای اینکار از یک برنامه مترجم بنام کامپایلر استفاده می شود. کامپایلرهای متعددی برای زبان C توسط شرکتهای مختلف و برای سیستم عاملهای مختلف نوشته شده است که می توانید برحسب نیاز از هریک از آنها استفاده نمایید. اما هنوز برنامه برای اجرا آماده نیست. معمولا برنامه نویسان از در برنامه های خود از یک سری از کدهای از پیش آماده شده برای انجام عملیات متداول (مانند محاسبه جذر و یا سینوس) استفاده می کنند که برنامه آنها قبلا نوشته و ترجمه شده است. این برنامه ها یا در قالب کتابخانه های استاندارد توسط شرکتهای ارائه کننده نرم افزار عرضه شده است و یا توسط دیگر همکاران برنامه نویس اصلی نوشته و در اختیار وی قرار داده شده است. در این مرحله باید کد مقصد برنامه اصلی با کدهای مربوط به این برنامه های کمکی پیوند زده شود. برای اینکار نیاز به یک پیوند زننده (Linker) داریم و نتیجه این عمل یک فایل قابل اجرا خواهد بود (در ویندوز این فایل پسوند .EXE خواهد داشت). مرحله بعدی اجرای برنامه و دادن ورودیهای لازم به آن و اخذ خروجیها می باشد. در شکل زیر این مراحل نشان داده شده اند.

مسلما طی مراحل بالا برای اجرای هر برنامه زمانبر می باشد، بهمین دلیل اکثر تولید کنندگان کامپایلرها، محیطهایی را برای برنامه نویسی ارائه کرده اند که کلیه مراحل بالا را بطور اتوماتیک انجام می دهند.

به این محیطها IDE (Integrated Development Environment) یا محیط مجتمع توسعه نرم افزار گفته می شود. این محیطها دارای یک ویرایشگر متن می باشند که معمولا دارای خواص جالبی همچون استفاده از رنگهای مختلف برای نشان دادن اجزای مختلف برنامه مانند کلمات کلیدی، و یا قابلیت تکمیل اتوماتیک قسمتهای مختلف برنامه می باشد. پس از نوشتن برنامه و با انتخاب گزینه ای مانند Run کلیه عملیات فوق بطور اتوماتیک انجام شده و برنامه اجرا می گردد. علاوه براین، این محیطها معمولا دارای امکانات اشکالزدایی برنامه (Debug) نیز می باشند که شامل مواردی همچون اجرای خط به خط برنامه و یا دیدن محتویات متغیرها در زمان اجرا است. چند محیط معروف برنامه نویسی عبارتند از :

Borland C++ 3.1 برای محیط DOS
Borland C++ از نسخه ۴ به بالا برای Windows
Microsoft Visual C++ برای محیط Windows
Borland C++ Builder برای محیط Windows

برای شروع ما از محیط Borland C++ 3.1 تحت Dos که نحوه کار ساده تری نسبت به سایرین دارد استفاده می کنیم.
پس از نصب این نرم افزار، برنامه BC.exe را اجرا کنید تا وارد محیط borland c شوید
همانطور که می بینید، این محیط از ۳ قسمت اصلی تشکیل شده است :
- **بخش ویرایش برنامه :** بخش آبی رنگ وسط می باشد که در حقیقت یک ویرایشگر است که برنامه در آن تایپ می شود. همانطور که می بینید در این ویرایشگر از رنگهای مختلف برای نشان دادن قسمتهای مختلف برنامه استفاده می شود. مثلا برای کلمات کلیدی از رنگ سفید استفاده شده است.
- **بخش منوهای کاری :** این بخش که در قسمت بالا واقع شده است،- حاوی تعدادی منو (گزینه) برای انجام وظایف مختلف است. خلاصه این عملیات عبارتند از :

- منوی File : عملیاتی مانند باز کردن و یا ذخیره یک برنامه
- منوی Edit : عملیات ویرایش مانند حذف،o کپی و یا چسباندن یک قسمت از برنامه
- منوی Search : جستجوی و یا تعویض یک متن در برنامه
- منوی Run : اجرای برنامه بصورت کامل یا دستور به دستور
- منوی Compile : عملیات مربوط به کامپایل و پیوند برنامه
- منوی Debug : عملیات مربوط به اشکالزدایی مانند دیدن مقادیر متغیرها در زمان اجرا
- منوی Project : عملیات مربوط به مدیریت برنامه هایی که شامل چندین فایل مستقل هستند (پروژه)
- منوی Options : عملیات مربوط به تنظیمات سیستم مانند نحوه کامپایل و یا رنگ پیش فرض محیط
- منوی Windows : عملیات مربوط به پنجره های باز فعلی (مربوط به چندین برنامه یا نمایش متغیرها و ...)

خطاهای برنامه نویسی

بنظر می رسد خطاها جزء جداناپذیر برنامه ها هستند. بندرت می توان برنامه ای نوشت که در همان بار اول بدرستی و بدون هیچگونه خطایی اجرا شود. اما خطاها از لحاظ تاثیری که بر اجرای برنامه ها می گذارند، متفاوتند. گروهی ممکن است باعث شوند که از همان ابتدا برنامه اصلا کامپایل نشود و گروه دیگر ممکن است پس از گذشت مدتها و در اثر دادن یک ورودی خاص به برنامه، باعث یک خروجی نامناسب و یا یک رفتار دور از انتظار (مانند قفل شدن برنامه) شوند. بطور کلی خطاها به دو دسته تقسیم می شوند:

خطاهای نحوی (خطاهای زمان کامپایل): این خطاها در اثر رعایت نکردن قواعد دستورات زبان C و یا تایپ اشتباه یک دستور بوجود می آیند و در همان ابتدا توسط کامپایلر به برنامه نویس اعلام می گردد. برنامه نویس باید این خطا را رفع کرده و سپس برنامه را مجددا کامپایل نماید. لذا معمولا این قبیل خطاها خطر کمتری را در بردارند.

خطاهای منطقی (خطاهای زمان اجرا): این دسته خطاها در اثر اشتباه برنامه نویس در طراحی الگوریتم درست برای برنامه و یا گاهی در اثر درنظر نگرفتن بعضی شرایط خاص در برنامه ایجاد می شوند. متاسفانه این دسته خطاها در زمان کامپایل اعلام نمی شوند و در زمان اجرای برنامه خود را نشان می دهند. بنابراین، این خود برنامه نویس است که پس از نوشتن برنامه باید آن را تست کرده و خطاهای منطقی آن را پیدا کرده و رفع نماید. متاسفانه ممکن است یک برنامه نویس خطای منطقی برنامه خود را تشخیص ندهد و این خطا پس از مدتها و تحت یک شرایط خاص توسط کاربر برنامه کشف شود. بهمین دلیل این دسته از خطاها خطرناکتر هستند. خود این خطاها به دو دسته تقسیم می گردند:

a. خطاهای مهلک: در این دسته خطاها کامپیوتر بلافاصله اجرای برنامه را متوقف کرده و خطا را به کاربر گزارش می کند. مثال معروف این خطاها خطای تقسیم بر صفر می باشد.

b. خطاهای غیرمهلک: در این دسته خطا اجرای برنامه ادامه می یابد ولی برنامه نتایج اشتباه تولید می نماید. بعنوان مثال ممکن است دراثر وجود یک خطای منطقی در یک برنامه حقوق و دستمزد حقوق کارمندان اشتباه محاسبه شود و تا مدتها نیز کسی متوجه این خطا نشود!

با توجه به آنچه گفته شد، در می یابیم که رفع اشکال برنامه ها بخصوص خطاهای منطقی از مهمترین و مشکلترین وظایف یک برنامه نویس بوده و گاهی حتی سخت تر از خود برنامه نویسی است! بهمین دلیل است که بسیاری از شرکتها(همانند مایکروسافت) ابتدا نسخه اولیه نرم افزار خود را در اختیار کاربران قرار می دهند تا اشکالات آن گزارش شده و رفع گردد. بسیار مهم است که در ابتدا سعی کنید برنامه ای بنویسید که حداقل خطاها را داشته باشد، در گام دوم با آزمایش دقیق برنامه خود هرگونه خطای احتمالی را پیدا کنید و در گام سوم بتوانید دلیل بروز خطا را پیدا کرده و آنرا رفع نمایید. هر سه عمل فوق کار سختی بوده و نیاز به تجربه و مهارت دارد.

آخرین نکته اینکه در اصطلاح برنامه نویسی به هر گونه خطا، bug و به رفع خطا debug گفته می شود.

زبان C از قدرتمندترین زبانهاست که سالهاست در صنایع و تجارت و خیلی چیزهای دیگر مورد استفاده قرار گرفته است. زبان C یک زبان سطح میانی میباشد. یعنی نه سطح بالاست(مانند پاسکال)و نه سطح پایین (مانند زبان اسمبلی). بلکه قابلیتهای هر دو اینها را دارد. به همین دلیل هم است که زبان C برای نوشتن برنامه های سیستمی مانند سیستم عامل و کامپایلر و... بسیار مناسب است.

C++ عموماً از سه بخش تشکیل شده است:

- محیطی برای نوشتن برنامه و ویرایش آن.
- کامپایلر C++.
- کتابخانه استاندارد C++.

یک برنامه زبان C++ برای رسیدن به مرحله اجرا از شش مرحله عبور می کند.

مرحله اول : برنامه نویس، برنامه را در محیط ویرایشگر می نویسد و آن را بر روی دیسک ذخیره می کند

مرحله دوم : برنامه پیش پردازنده، خطوط برنامه را از لحاظ ایردات نگارشی بررسی می کند، و در صورت وجود اشکال در برنامه پیغام خطائی داده می شود، تا برنامه نویس نسبت به رفع آن اقدام نماید.

مرحله سوم : کامپایلر، برنامه را به زبان ماشین ترجمه می کند و آن را بر روی دیسک ذخیره می نماید.

مرحله چهارم : پیوند دهنده، کدهای زبان ماشین را، به فایلهای کتابخانه هایی که مورد استفاده قرار گرفته اند پیوند می دهد و یک فایل قابل اجرا بر روی دیسک می سازد.

مرحله پنجم : بار کننده برنامه را در حافظه قرار می دهد.

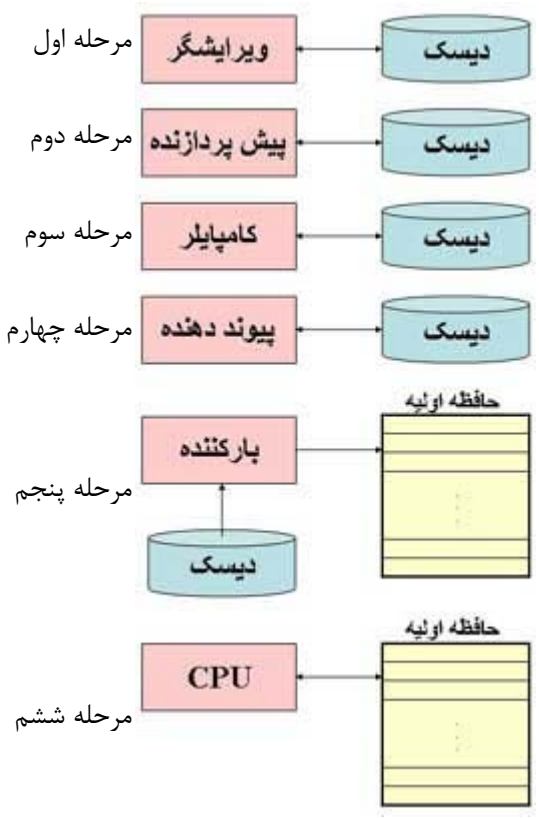
مرحله ششم : واحد پردازش مرکزی کامپیوتر دستورات برنامه را اجرا می کند.

مرحله اول

مرحله دوم

مرحله سوم

مرحله چهارم



نکته : همانطور که گفته شد پیش پردازنده ایرادات برنامه را بررسی می کند و در صورتی که برنامه مشکلی نداشت در نهایت به زبان ماشین ترجمه می شود و قابلیت اجرا پیدا می کند اما در هنگام اجرای برنامه نیز ممکن است خطایی بروز کند به عنوان مثال تقسیم بر صفر بوجود آید. این خطا قابل تشخیص توسط پیش پردازنده نیست و در زمان اجرای برنامه رخ می دهد و باعث خروج ناگهانی از برنامه می شود. به اینگونه خطاها، خطای زمان اجرا گفته می شود. تقسیم بر صفر جزء خطاهای مهلک است. خطای غیر مهلک خطایی است که اجازه اجرای ادامه برنامه را می دهد ولی ممکن است نتایج غیر صحیحی را به ما بدهد.

مفاهیم حافظه و انواع داده

همانطور که در سازمان کامپیوتر گفتیم یکی از واحدهای کامپیوتر، واحد حافظه می باشد.

این واحد که به آن **RAM** (حافظه با دسترسی تصادفی Memory Random Access) نیز می گویند، برای ذخیره موقت داده ها و دستورالعملها تا هنگامی که به آنها احتیاج شود استفاده می شود. اطلاعاتی که در RAM قرار دارند قابل پاک

شدن و جایگزین شدن با داده های دیگر است. فضایی که ما در برنامه نویسی برای متغیرها و داده ها استفاده می کنیم در RAM قرار دارد. برای درک بهتر مطلب ، واحدهای اندازه گیری حافظه را بررسی می کنیم:

Bit بیت: یک بیت عنصری الکترونیکی در کامپیوتر است که دارای دو حالت روشن (۱) و خاموش(۰) می باشد و کوچکترین واحد اطلاعاتی است.

Byte بایت: چون بیتها واحدهای اطلاعاتی کوچکی هستند و فقط می توانند دو حالت را انتقال دهند، بنابراین آنها را در واحدهای بزرگتری سازماندهی می کنند تا اطلاعات بیشتری هر بار قابل انتقال باشد. این واحد بزرگتر بایت است که واحد اصلی اطلاعات در سیستمهای کامپیوتری می باشد. هر ۸ بیت ، یک بایت را تشکیل می دهند.

از واحدهای زیر برای اندازه گیری حافظه استفاده می شود:

یک کیلو بایت $1\ \mathbf{KB} = 1024\ \mathbf{B} = 2^{10}\ \mathbf{B}$

یک مگا بایت $1\ \mathbf{MB} = 1024\ \mathbf{KB} = 2^{20}\ \mathbf{B}$

یک گیگا بایت $1\ \mathbf{GB} = 1024\ \mathbf{MB} = 2^{30}\ \mathbf{B}$

ما در برنامه نویسی نیاز به خانه های حافظه داریم. در تعریف خانه حافظه باید نام و نوع اطلاعاتی که در آن قرار می گیرد معین شود.

```
نام متغییر        نوع داده  ;
int                i1,i2,index;
```

دستور فوق سه خانه حافظه با نامهای i1 وi2 وindex از نوع اعداد صحیح تعیین می کند، یعنی در هر کدام از خانه های حافظه فوق می توان یک عدد صحیح در بازه ۳۲۷۶۷ تا ۳۲۷۶۸- قرار داد. نوع داده int به دو بایت حافظه نیاز دارد.

نکته :

- هر دستور زبان C++ به **;** ختم می شود.
- برای نام گذاری خانه های حافظه فقط می توان از حروف، اعداد و ... استفاده کرد و نیز حرف اول نام یک متغیر باید یک حرف باشد. به عنوان مثال نامهای **test۱** و **test!num** و **mark.1** اسامی غیر مجاز می باشند.

- توضیحات در C بین //(برای یک خط) و /* ..... */ (برای چند خط) قرار می گیرند.

- بین حروف نام متغیر نمی توان از کاراکتر فاصله استفاده کرد.
- زبان C++ دارای تعدادی کلمات کلیدی است که نمی توان از این کلمات به عنوان نام متغیر استفاده کرد. کلمات کلیدی زبان C++ عبارتند از:

char	cdecl	case	break	auto	asm
delete	default	_cs	continue	const	class
_es	enum	else	_ds	double	do
for	float	_fastcall	far	_export	extern
int	inline	if	huge	goto	friend
operator	new	near	long	_loadds	interrupt
return	register	public	protected	private	pascal
_ss	sizeof	signed	short	_seg	_saveregs
typedef	this	template	switch	struct	static
while	volatile	void	virtual	unsigned	union

-
- زبان C++ نسبت به حروف حساس است.(case sensitive) یعنی بین حروف کوچک و بزرگ تفاوت قائل می شود. در این زبان تمام کلمات کلیدی با حروف کوچک نوشته می شوند، به عنوان مثال short یک کلمه کلیدی می باشد ولی SHORT یا shoRT کلمات کلیدی نیستند. توصیه می شود که تمام برنامه های این زبان با حروف کوچک نوشته شوند.

در زبان C++ چهار نوع داده اصلی وجود دارد که عبارتند از :

۱- **char** : این نوع داده برای ذخیره داده های کاراکتری مانند 'a' ، '۱' ، '.' به کار می رود و بازه قابل قبول آن از ۱۲۸- تا ۱۲۷ می باشد. در حقیقت خانه های char نیز از نوع اعداد صحیح می باشند که یک بایت طول دارند و کد اسکی کاراکتر مورد نظر را در خود حفظ می کنند. به عنوان مثال کد اسکی کاراکتر A عدد ۶۵ می باشد.

۲- **int** : این نوع داده برای ذخیره اعداد صحیح مانند ۱۳۰۰، ۳۲۰۰۰ ، ۸۵۰- به کار می رود و بازه قابل قبول آن ۳۲۷۶۸- تا ۳۲۷۶۷ می باشد.

۳- **float** : این نوع داده برای ذخیره اعداد اعشاری مانند ۱۲,۵۲۴۱ ، ۱۵۰۱,۳- ، ۱۴۱۵,۱۲۳۴ به کار می رود و دقت آن تا ۷ رقم اعشاری می باشد.

۴- **double** : این نوع داده برای ذخیره سازی اعداد اعشاری بزرگ به کار می رود و دقت آن از float بیشتر می باشد.

باکلماتی مانند signed (علامت دار) ، unsigned (بدون علامت)، short (کوتاه) و long (بلند) انواع داده های جدیدی می توان ایجاد کرد. نوع int با هر چهار کلمه فوق می تواند مورد استفاده قرار گیرد. نوع char می تواند با signed و unsigned به کار رود و نوع double می تواند با long به کار رود. به جدول زیر توجه کنید:

نوع داده	طول داده	بازه
unsigned char	8 bits	0 to 255
char	8 bits	-128 to 127
enum	16 bits	-32,768 to 32,767
unsigned int	16 bits	0 to 65,535
short int	16 bits	-32,768 to 32,767
int	16 bits	-32,768 to 32,767
unsigned long	32 bits	0 to 4,294,967,295
long	32 bits	-2,147,483,648 to 2,147,483,647
float	32 bits	3.4 * (10**-38) to 3.4 * (10**+38)
double	64 bits	1.7 * (10**-308) to 1.7 * (10**+308)
long double	80 bits	3.4 * (10**-4932) to 1.1 * (10**+4932)

اعمال ریاضی و محاسباتی

در مبحث حافظه با انواع داده و شیوه اختصاص دادن حافظه به متغیرها آشنا شدیم حال می توانیم متغیرها را در محاسبات به کار ببریم. برای نیل به این هدف **C++** عملگرهایی را در اختیار ما قرار داده است.

عملگر انتساب (=)

عملگر تساوی جهت اختصاص دادن یک مقدار به یک متغیر به کار می رود ، مانند **a = 5** که عدد ۵ را به متغیر **a** تخصیص می دهد. جزئی که در سمت چپ تساوی قرار دارد همواره باید نام یک متغیر باشد، وجزء سمت راست تساوی می تواند یک عدد، یک متغیر و یا ترکیبی از هر دو باشد. مانند: **a=b+5** ، که در ایجا حاصل **b + 5** در متغیر **a** قرار می گیرد. توجه داشته باشید که همواره مقدار سمت راست تساوی در مقدار سمت چپ قرار می گیرد. به دستورات زیر توجه کنید.

```
int a,b;
a = 10;
b = 4;
a = b;
b = 7;
```

اگر از دستورات فوق استفاده کنیم در نهایت مقدار **a** برابر **۴** و مقدار **b** برابر **۷** خواهد بود. **C++** قابلیتهای زیادی دارد یکی از این قابلیتها اینست که می توانیم چند دستور را در یک دستور خلاصه کنیم ، به عنوان مثال دستور :

```
a = 2 + (b = 5);
```

برابر است با:

```
b = 5;
a = 2 + b;
```

که هر دو عبارت در نهایت عدد **۷** را در متغیر **a** قرار می دهند.

ضمناً استفاده از عبارت زیر نیز در **C++** مجاز می باشد:

```
a = b = c = 5
```

عبارت فوق عدد **۵** را به سه متغیر **a** و **b** و **c** اختصاص می دهد.

عملگر های محاسباتی

پنج عملگر محاسباتی که قابل استفاده در زبان **C++** هستند عبارتند از:

+	جمع
-	تفریق
*	ضرب
/	تقسیم
%	باقیمانده تقسیم

تنها عملگری که ممکن است برای شما ناشناس باشد عملگر % است. این عملگر باقیمانده تقسیم دو عدد صحیح را به ما می دهد، به عنوان مثال اگر از دستور زیر استفاده کنیم:

```
a = 11 % 3;
```

متغیر **a** حاوی عدد ۲ خواهد شد. چون عدد ۲ باقیمانده تقسیم ۱۱ بر ۳ می باشد.

عملگر های انتساب مرکب

عملگرهای انتسال مرکب عبارتند از += ، -= ، *= ، /= ، %= .این عملگرها دو کار را با هم انجام می دهند و در کم شدن کد نویسی به ما کمک می کنند، به جای توضیح اضافی به مثال های زیر که فهم مطلب را ساده تر می کند توجه کنید:

```
value += increase;«برابر است با»value=value+increase;
a -= 5;           «برابر است با»a = a - 5;
a /= b;           «برابر است با»a = a / b;
price*=units+1;   «برابر است با»price=price*(units+1);
x %= y * z;       «برابر است با»x = x % (y * z);
```

عملگرهای افزایش و کاهش

گونه ای دیگر از عملگرها که در کم شدن کد نویسی به ما کمک می کنند عملگر افزایش(++) و عملگر کاهش(--) می باشند. عملگر افزایش(++) یک واحد به مقدار قبلی که در متغیر بود اضافه می کند و عملگر کاهش(--) یک واحد از مقدار قبلی که در متغیر بود کم می کند.

```
++a;      a++;      a += 1;      a = a + 1;
```

هر چهار دستور فوق یک واحد به مقدار قبلی متغیر اضافه می کنند.

```
--a;      a--;      a -= 1;      a = a - 1;
```

هر چهار دستور فوق یک واحد از مقدار قبلی متغیر کم می کنند.

اگر از دستورات ++**a** و **a**++ به تنهایی استفاده کنیم فرقی ندارد که ++ قبل از متغیر قرار گیرد یا بعد از متغیر. اما اگر از ++ در کنار عملگرهای دیگر استفاده شود، اگر ++ قبل از متغیر قرار گیرد ابتدا یک واحد به متغیر اضافه شده سپس در محاسبه استفاده می شود، ولی اگر ++ بعد از متغیر قرار گیرد ابتدا متغیر در محاسبه استفاده می شود سپس یک واحد به آن اضافه می شود. همین روال برای عملگر -- نیز برقرار است. به مثال های زیر توجه کنید:

b = 3; a = b++;	b = 3; a = ++b;

در مثال سمت چپ ابتدا یک واحد به **b** اضافه می شود، یعنی **b** مقدار **۴** را می گیرد سپس عدد **۴** در **a** قرار می گیرد؛ اما در مثال سمت راست ابتدا مقدار **b** یعنی عدد **۳** در **a** قرار می گیرد سپس یک واحد به **b** اضافه می شود و مقدار ۴ را می گیرد.

در این مثال عدد ۶ در **m** قرار می گیرد:

```
a = 2;
b = 3;
m = ++a + b--;
```

b مقدار **۲** و **a** مقدار ۳ را می گیرد.

حال که با انواع عملگرهای محاسباتی آشنا شدید عبارت زیر را در نظر بگیرید.

y = 5 * 3 + 2 - 1 * 3 / 2;

مقداری که در **y** قرار می گیرد چه عددی می تواند باشد؟ **۳۰** یا **۲۴** یا **۱۵,۵** یا **۱۷,۵** . نظر شما چیست؟ شما مقدار **y** را چگونه حساب می کنید؟

کامپیوتر برای بررسی و محاسبه چنین عبارتی برای اینکه با چندین جواب مواجه نشود قواعدی را در نظر می گیرد و طبق قوانین تقدم عملگرها عمل می کند. این قوانین که مشابه قوانین جبر می باشند به ترتیب عبارتند از:

۱- عملگرهایی که درون پرانتز قرار دارند اول محاسبه می شوند. در صورتی که پرانتزها تودرتو باشند ابتدا داخلی ترین پرانتز مورد بررسی و محاسبه قرار می گیرد.

۲- اگر عبارتی حاوی * ، / و ٪ باشد پس از پرانتز این عملگرها در اولویت قرار دارند. اگر عبارتی حاوی ترکیبی از این عملگرها باشد چون این عملگرها در تقدم یکسانی نسبت به یکدیگر قرار دارند، محاسبه به ترتیب از چپ به راست انجام می شود.

۳- اعمال + و - در انتها انجام می شوند. اگر عبارتی شامل ترکیبی از این دو عملگر باشد چون این دو عملگر در تقدم یکسانی نسبت به هم هستند، محاسبه به ترتیب از چپ به راست انجام می شود.

با توجه به قواعد گفته شده حاصل عبارت فوق عدد ۱۵,۵ خواهد بود.

```
y = 5 * 3 + 2 - 1 * 3 / 2;     ----»  y = 15.5
  6   1   4   5   2   3
```

به مثال های زیر توجه کنید:

```
x = (2 + 1) * 3 + 5;           ----»  x = 14
  4    1    2   3
```

```
z = 5 % 3 * (3 + 1);          ----»  z = 8
  4   2   3    1
y = p * r % q + w / x - y;
  6   1   2   4   3   5
```

عبارات منطقی

یک عبارت منطقی، عبارتی است با مقدار درست یا نادرست. به عنوان مثال ۵ بزرگتر از ۳ است، یک عبارت منطقی است با مقدار درست و ۵ کوچکتر از ۳ است، نیز یک عبارت منطقی است اما با مقدار نادرست. در کامپیوتر نتیجه عبارات منطقی درست عدد یک و نتیجه عبارات منطقی نادرست عدد صفر خواهد بود.

ضمناً کامپیوتر هر عدد مخالف صفر را به عنوان یک عبارت منطقی درست در نظر می گیرد.

عملگرهای رابطه ای

برای مقایسه دو متغیر یا دو عبارت از عملگرهای رابطه ای استفاده می کنیم که همانطور که گفته شد دارای نتیجه درست یا نادرست می باشد. عملگرهای رابطه ای عبارتند از ==(مساوی)، =!(متفاوت)، <(بزرگتر از)، >(کوچکتر از)، =<(بزرگتر مساوی از)، =>(کوچکتر مساوی از). به مثال های زیر توجه کنید.

ضمناً به جای اینکه فقط از اعداد در عبارتهای رابطه ای استفاده کنیم می توانیم از عبارتهایی شامل متغیرها و یا ترکیبی از متغیرها و اعداد استفاده کنیم به عنوان مثال فرض کنید a = 2 و b=3 و c=6 خواهیم داشت:

```
(a==5)          ----->  نادرست
(a*b>=c)        ----->  درست
(b+4<a*c))      ----->  نادرست
((b=2)==a))     ----->  درست
```

توجه کنید که عملگر = همانند عملگر == نمی باشد. اولی عملگر انتساب است که مقدار سمت راست را در متغیر سمت چپ قرار می دهد و دیگری عملگر رابطه ای است که برابر بودن یا نبودن دو مقدار را با هم مقایسه می کند. بنابراین در عبارت ((b=2)==a)) ما ابتدا مقدار ۲ را در متغیر b قرار دادیم سپس آن را با a مقایسه کردیم، لذا نتیجه این مقایسه درست بود.

عملگرهای منطقی

عملگرهای منطقی عبارتند از ! ، || و **&&** . نتیجه عملگر ! (NOT) وقتی درست است که عبارتی که این عملگر بر روی آن عمل می کند نادرست باشد و نتیجه هنگامی نادرست است که عملوند آن درست باشد. ضمناً این عملگر تنها یک عملوند دارد. در حقیقت این عملگر نقیض عملوند خود را به عنوان نتیجه می دهد.

به مثال های زیر توجه کنید:

```
!(5 == 5)     ----> نادرست
!(6 <= 4)     ----> درست
!0            ----> درست
!1            ----> نادرست
```

عملگرهای **&&**(AND) و ||(OR) هنگامی مورد استفاده قرار می گیرند که بخواهیم از دو عبارت یک نتیجه را بدست آوریم. نتیجه این عملگرها بستگی به ارتباط بین دو عملوندشان طبق جدول زیر دارد:

نتیجه a||b	نتیجه a&&b	عملوند دوم a	عملوند اول b
درست	درست	درست	درست
درست	نادرست	نادرست	درست
درست	نادرست	درست	نادرست
نادرست	نادرست	نادرست	نادرست

به مثال های زیر توجه نمائید:

```
((5==5)&&(3>6))     -----> نادرست
((5==5)||(3>6))     -----> درست
((3-3)&&(3<5))      -----> نادرست
((3-3)||(3<5))      -----> درست
```

در مثال های زیر به جای اعداد از متغیر نیز استفاده شده است (فرض کنید a=1 و b=2 وc=3)

```
((b-2*a)&&(c==3))          -----> نادرست
((b==2*a)&&(c!=4))         -----> نادرست
```

```
((c==a+b)||(b<a))          ----->   درست
((b-c==-a)||(b-c==a))      ----->   درست
```

عملگر شرطی

این عملگر یک عبارت را مورد ارزیابی قرار می دهد و براساس عبارت ارزیابی شده مقادیر متفاوتی را به عنوان نتیجه بر می گرداند. ساختار این عملگر به صورت زیر می باشد:

```
شرط ? نتیجه ۱ : نتیجه ۲
```

اگر شرط برقرار باشد نتیجه ۱ به عنوان خروجی خواهد بود در غیر این صورت نتیجه ۲ به عنوان خروجی در نظر گرفته می شود. به مثال های زیر توجه نمایید:

```
7==6?4:3      --->خروجی عدد۳ می باشد چون۷ مساوی ۶ نمی باشد
8==6+2?4:3    --->خروجی عدد۴ می باشد چون۸ مساوی ۶+۲می باشد
6>3?a:b       ---> خروجی aمی باشد چون ۶ از ۳ بزرگتر است
a>b?a:b       ---> خروجی عدد بزرگتر می باشد a یا b
```

همانطور که در عملگرهای محاسباتی دیدیم درک تقدم عملگرها، اهمیت ویژه ای داشت در اینجا نیز دانستن این تقدم از اهمیت خاصی برخوردار می باشد، تقدم عملگرهای رابطه ای ، منطقی و شرطی به ترتیب عبارتند از:

۱- !

۲- > >= < <=

۳- == !=

۴- &&

۵- ||

۶- ?:

به عنوان مثال مراحل بررسی عبارت مقابل به صورت زیر می باشد:

```
2 >= 3 && 2 == 2 || 2 != 3
  1     4     2     5     3

1   نادرست
2   درست
```

```
3    درست
4    نادرست  &&  درست    ---->    نادرست
5    درست || نادرست    ---->    درست

جواب نهایی درست می باشد
```

پیشنهاد می شود برای جلوگیری از پیچیدگی فهم عبارتهای منطقی و یا محاسباتی تقدم های مورد نظر را با به کار بردن پرانتز کاملاً مشخص کنیم ، به عنوان مثال عبارت فوق را به صورت زیر مورد استفاده قرار دهیم:

```
(((2 >= 3) && (2 == 2)) || (2 != 3))
```

دستورات ورودی و خروجی

همانطور که در سازمان کامپیوتر گفته شد کامپیوتر دارای واحدهای ورودی و خروجی می باشد. واحد ورودی که ما در اینجا استفاده می کنیم صفحه کلید می باشد و واحد خروجی مورد استفاده نیز صفحه نمایش خواهد بود.

برای دریافت اطلاعات از صفحه کلید ، زبان C++ دستوری به نام **cin** را در اختیار ما قرار داده است، و برای ارسال اطلاعات به صفحه نمایش دستور **cout** موجود می باشد. توسط این دو دستور شما می توانید با نمایش اطلاعات بر روی صفحه نمایش و دریافت اطلاعات از صفحه کلید با کاربری که از برنامه شما استفاده می کند، در ارتباط باشید.

دستور خروجی cout

دستور cout همراه علامت << به کار می رود.

```
cout << "This is a test";
```

دستور فوق عبارت This is a test را بر روی صفحه نمایش چاپ می کند.

```
cout << 5120;
```

دستور فوق عدد ۵۱۲۰ را بر روی صفحه نمایش ظاهر می سازد.

```
cout << x;
```

دستور فوق محتویات متغیر x را به صفحه نمایش می فرستد.

علامت >> با نام عملگر درج شناخته می شود و اطلاعاتی که بعد از این علامت قرار می گیرند به واحد خروجی منتقل می شوند. در مثال های فوق یک عبارت رشته ای (This is a test) یک عدد (۵۱۲۰) و یک متغیر (x) به واحد خروجی ارسال شدند. توجه داشته باشید که در اولین مثال عبارت This is a test بین دو علامت (") قرار گرفت ، چون این عبارت حاوی رشته ای از حروف می باشد؛ هرگاه که بخواهیم رشته ای از حروف را به کار ببریم باید آنها را بین دو علامت (") قرار دهیم تا با نام متغیرها به اشتباه گرفته نشوند. به عنوان مثال، دستور زیر:

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

عبارت Hello را بر روی صفحه نمایش ظاهر می سازد ولی دستور زیر:

```
cout << Hello;
```

محتویات متغیری با نام Hello را بر روی صفحه نمایش چاپ می کند.

عملگر درج ممکن است بیش از یک بار در یک جمله به کار رود، به عنوان مثال دستور زیر:

```
cout << "Hello," << "I am" << "new in C++";
```

پیغام Hello, I am new in C++ را بر روی صفحه نمایش نشان می دهد.

تکرار استفاده از عملگر درج در یک دستور به ما این امکان را می دهد که ترکیبی از متغیر و رشته حروف را در کنار هم استفاده کنیم.

```
cout << "Hello, my code is" << code
     << "and I am" << age << "years old.";
```

به عنوان مثال دستور فوق با فرض اینکه متغیر code حاوی عدد ۱۱۶۲۲۳ و متغیر age حاوی عدد ۱۶ باشد عبارت زیر را در صفحه نمایش ظاهر می سازد:

```
Hello, my code is 116223 and I am 16 years old.
```

توجه داشته باشید که دستور cout عبارات را به صورت خودکار به خط بعد منتقل نمی کند، به عنوان مثال دستورهای زیر:

```
cout << "This is a text.";
cout << "This is another text.";
```

علارقم اینکه از دستور cout در دو خط استفاده شده است، به صورت زیر در صفحه نمایش نشان داده خواهد شد:

```
This is a text. This is another text.
```

برای اینکه عبارتی را در چند خط نمایش دهیم، برای انتقال به هر خط جدید از علامت \n استفاده می کنیم. به عنوان مثال دستورات زیر:

```
cout << "First sentence.\n";
cout << "Second sentence.\n Third sentence.";
```

به شکل زیر در صفحه نمایش دیده خواهد شد:

```
First sentence.
Second sentence.
Third sentence.
```

علاوه بر علامت \n می توان از دستور endl برای انتقال به خط جدید استفاده کرد به عنوان مثال دستورات :

```
cout << "First sentence." << endl;
cout << "Second sentence." << endl;
```

در صفحه نمایش به صورت زیر دیده می شوند:

```
First sentence.
Second sentence.
```

دستور ورودی cin

دستور cin همراه علامت << به کار می رود.

```
int age;
cin >> age;
```

دستورات فوق ابتدا فضایی در حافظه برای متغیر age در نظر می گیرند، سپس برنامه منتظر وارد کردن عددی از صفحه کلید می ماند تا عدد وارد شده را در متغیر age قرار دهد. cin هنگامی قادر به دریافت اطلاعات از صفحه کلید خواهد بود که، کلید Enter بر روی صفحه کلید فشرده شود. به عنوان مثال اگر بخواهیم عدد ۱۶ در متغیر age قرار گیرد ابتدا عدد ۱۶ را تایپ کرده سپس دکمه Enter را فشار می دهیم.

علامت >> با نام عملگر استخراج شناخته می شود، و اطلاعاتی که از واحد ورودی دریافت می شود در متغیری که بعد از این علامت می باشد، قرار می گیرند. ضمناً شما می توانید توسط یک دستور cin بیش از یک متغیر را مقدار دهی کنید.

به عنوان مثال دستورات زیر معادل یکدیگر می باشند:

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

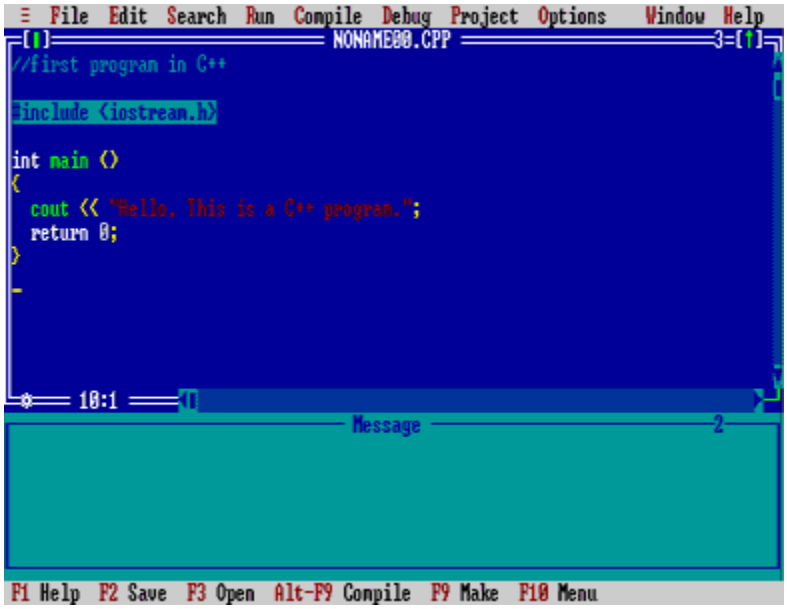
برنامه چاپ یک متن

در مباحث قبلی با مفاهیم حافظه و انواع داده، اعمال محاسباتی، عبارات منطقی ودستورات ورودی وخروجی آشنا شدیم. با این مقدمات می توان نوشتن اولین برنامه را آغاز کرد و با این برنامه نحوه اجرا و سایر جزئیات را مورد بررسی قرار خواهیم داد.

```
//first program in C++

#include <iostream.h>

int main ()
{
  cout << "Hello, This is a C++ program.";
  return 0;
}
```

قبل از هر گونه توضیح اضافی به شیوه نوشتن این برنامه در ویرایشگر زبان **C++** و نحوه اجرای آن می پردازیم.

برای اجرای این برنامه شما به کامپایلر زبان **C**++ نیاز دارید ، که باید آنرا نصب کنید. پس از نصب برنامه به آدرس "c:\tcp\bin" بر روی کامپیوتر خود مراجعه کرده و فایل **tc** را اجرا نمایید. با اجرای این فایل وارد محیط ویرایشگر برنامه **C**++ خواهید شد.



حال در این محیط می توانید برنامه ذکر شده در ابتدای این مبحث را بنویسید. در نوشتن برنامه دقت لازم را به کار ببرید ، چون در صورت وجود اشتباهات تایپی ممکن است اجرای برنامه با مشکل مواجه شود. پس از این که برنامه را نوشتید یک بار دیگر آن را بررسی کنید تا مطمئن شوید اشتباهی ندارد، سپس دکمه **Ctrl** را از صفحه کلید فشار داده و آن را نگه دارید ، سپس همزمان دکمه **F9** از صفحه کلید را فشار دهید(**Ctrl + F9**). با این کار برنامه اجرا خواهد شد. همانطور که دیدید صفحه نمایش به سرعت دوباره به صفحه ویرایشگر زبان **C**++ بر می گردد. برای اینکه خروجی برنامه خود را ببینید دکمه **Alt** را همراه با **F5** را فشار دهید (**Alt + F5**) . با این کار به محیط خروجی برنامه خواهید رفت و پیغام:

```
Hello, This is a C++ program.
```

را در یک صفحه سیاه خواهید دید. تبریک می گوئیم ، شما اولین برنامه خود با زبان **C**++ را با موفقیت به مرحله اجرا گذاشتید؛ به همین سادگی.

اگر به یاد داشته باشید در مبحث معرفی ساختاری زبان **C**++ روال رسیدن به مرحله اجرا را شش مرحله ذکر کردیم، در اینجا مرحله اول را که نوشتن برنامه در ویرایشگر بود، شما به اجرا گذاشتید و با فشار کلید (**Ctrl + F9**) مراحل دوم تا ششم به صورت خودکار انجام گرفت، پس شش مرحله ذکر شده همچنان برقرار است و ما نیز نیازی نداریم خود را با جزئیات مراحل دوم تا ششم درگیر کنیم.

پس برای اجرای هر برنامه کافی است برنامه را در محیط ویرایشگر زبان C++ بنویسیم سپس دکمه (**Ctrl + F9**) را فشار دهیم.

برنامه ای که ما در اینجا نوشتیم یکی از ساده ترین برنامه هایی است که می توانیم به زبان **C**++ بنویسیم ، ضمن اینکه شامل دستوراتی است که تقریباً هر برنامه **C**++ به آنها نیاز دارد. در اینجا به بررسی تک به تک دستورات برنامه فوق می پردازیم.

```
//first program in C++
```

دستور فوق شامل توضیحات می باشد و تأثیری بر نحوه اجرای برنامه نخواهد داشت. هر نوشته ای که بعد از علامت // در زبان **C**++ قرار گیرد به عنوان توضیحات در نظر گرفته می شود. توضیحی که در اینجا مورد استفاده قرار گرفته به ما می گوید که این اولین برنامه ما به زبان **C**++ می باشد. علاوه بر علامت // ، توضیحات را می توان بین */ و /* قرار داد. از شیوه جدید هنگامی استفاده می کنیم که توضیحات ما بیش از یک خط باشد.

```
/* This is a comment line.
   This is another comment line. */
```

قرار دادن دستورات فوق در برنامه تأثیری بر خروجی ندارد و تنها توضیحاتی برای فهم بهتر برنامه می باشد.

```
#include <iostream.h>
```

خطوطی که با علامت پوند # شروع می شوند دستوراتی برای پیش پردازنده می باشند. این دستورات جزء خطوط اجرایی برنامه نمی باشند و نشانه هایی برای کامپایلر می باشند. در اینجا دستور فوق به پیش پردازنده می گوید که تعدادی از دستورات مورد استفاده در این برنامه در فایل کتابخانه ای **iostream.h** قرار دارند. در این مورد خاص فایل **iostream.h** شامل دستورات ورودی و خروجی (مانند **cin** و **cout**) می باشد.

```
int main( )
```

این دستور شروع بدنه اصلی برنامه را مشخص می کند. تابع **main** اولین جایی از برنامه است که زبان **C**++ شروع به اجرای آن می کند. فرقی ندارد که تابع **main** را در کجا مورد استفاده قرار دهیم. ابتدا وسط یا انتهای کدهای برنامه نویسی، در هر کجا که تابع **main** را قرار دهیم ، زبان **C**++ ابتدا به این تابع مراجعه می کند. بعد از کلمه **main** یک جفت پرانتز ( ) قرار می دهیم، چون **main** یک تابع است. در زبان **C**++ تمام توابع دارای یک جفت پرانتز می باشند (در مبحث توابع به طور مفصل در باره نحوه ایجاد تابع و آرگومانها و ... صحبت خواهیم کرد). محتویات تابع **main** همانطور که در برنامه دیدید بین دو علامت { } قرار می گیرند.

```
cout << "Hello, This is a C++ program.";
```

این دستور کار اصلی مورد نظر در این برنامه را که چاپ عبارت داخل کوتیشن " " بر روی صفحه نمایش است را انجام می دهد. همانطور که گفته شد هنگامی که می خواهیم از دستورات ورودی و خروجی استفاده کنیم باید در ابتدای برنامه از دستور #include<iostream.h> استفاده کنیم. توجه داشته باشید که بعد از هر دستور زبان **C**++ ملزم به استفاده از علامت (**;**) می باشیم.

```
return 0;
```

این دستور باعث می شود که تابع **main** به پایان برسد و عدد صفر را به عنوان خروجی تابع بر می گرداند. این مرسوم ترین روش برای پایان دادن به برنامه بدون دریافت هیچگونه پیغام خطا می باشد. همانطور که در برنامه های بعدی خواهید دید، تقریباً همه برنامه های زبان **C**++ با دستوری مشابه دستور فوق به پایان می رسند.

نکته: بهتر است هر دستور زبان **C**++ را در یک خط جداگانه بنویسیم. البته در انجام اینکار الزامی نداریم ولی برای خوانایی بیشتر برنامه توصیه می شود از این شیوه استفاده کنید.

برنامه جمع دو عدد

در مبحث قبلی برنامه ای را نوشتیم که در آن تنها از دستور cout استفاده شده بود و رشته ای را بر روی صفحه نمایش چاپ می کرد. برای اینکه با نحوه کاربرد متغیرها و شیوه مقدار دهی به آنها و نیز دستور cin آشنا شوید، در اینجا برنامه جدیدی را می نویسیم که دو عدد را از ورودی دریافت کرده و سپس آنها را جمع نموده و حاصل را در خروجی نمایش می دهد.

```
// Addition program.
#include <iostream.h>

// function main begins program execution
int main()
{
  int integer1; // first number to be input by user
  int integer2; // second number to be input by user
  int sum; // variable in which sum will be stored

  cout << "Enter first integer\n"; // prompt
  cin >> integer1; // read an integer

  cout << "Enter second integer\n"; // prompt
  cin >> integer2; // read an integer

  sum = integer1 + integer2; //assignment result to sum
```

```
  cout << "Sum is " << sum << endl; // print sum

  return 0; // indicate that program ended successfully

} // end function main
```

همانطور که در این برنامه نیز می بینید، تعدادی از دستورات برنامه قبلی تکرار شده اند. در اینجا به بررسی و توضیح دستورات جدید می پردازیم:

```
  int integer1; // first number to be input by user
  int integer2; // second number to be input by user
  int sum; // variable in which sum will be stored
```

سه دستور فوق وظیفه تخصیص حافظه به سه متغیر integer1 و integer2 و sum از نوع عدد صحیح را دارند. انواع داده در مبحث مفاهیم حافظه و انواع داده توضیح داده شده اند. در ضمن به جای استفاده از سه دستور فوق می توانستیم از دستور زیر نیز استفاده کنیم:

```
int integer1, integer2, sum;
```

نکته :

- بعضی از برنامه نویسان ترجیح می دهند که هر متغیر را در یک خط تعریف کنند و توضیحات لازم را در جلوی آن بنویسند.
- متغیر را می توان در هر جایی از برنامه تعریف کرد، ولی حتماً این کار باید قبل از اولین استفاده از متغیر صورت گیرد، به عنوان مثال برنامه فوق را می توان به صورت زیر نوشت:

```
•   #include <iostream.h>
•
•   int main()
•   {
•     cout << "Enter first integer\n";
•
•     int integer1;
•     cin >> integer1;
•
•     cout << "Enter second integer\n";
•
•     int integer2;
•     cin >> integer2;
•
```

```
•     int sum;
•     sum = integer1 + integer2;
•
•     cout << "Sum is " << sum << endl;
•
•     return 0;
•   }
```

- اگر تعریف متغیر را در بین دستورات اجرایی برنامه انجام می دهید، یک خط خالی قبل از آن بگذارید تا تعریف متغیر مشخص باشد. اینکار به وضوح برنامه کمک می کند.
- اگر تعریف متغیرها را در ابتدای برنامه انجام می دهید، یک خط خالی بعد از آنها بگذارید تا از دستورات اجرایی جدا شوند. اینکار نیز به وضوح برنامه و سهولت اشکال زدایی کمک می کند.

```
cout << "Enter first integer \n";
```

دستور فوق رشته Enter first integer را بر روی صفحه نمایش نشان می دهد و به ابتدای سطر جدید می رود.

```
cin >> integer1;
```

دستور فوق با وارد کردن هر عدد و فشردن کلید Enter عدد وارد شده را در متغیر integer1 قرار می دهد.

```
cout << "Enter second integer \n";
cin >> integer2;
```

دو دستور فوق نیز ابتدا عبارت integer Enter second را بر روی صفحه نمایش چاپ کرده و سپس در خط بعد عدد وارد شده از صفحه کلید را پس از فشردن کلید Enter در متغیر integer2 قرار می دهد.

```
sum = integer1 + integer2;
```

دستور فوق حاصل جمع متغیرهای integer1 و integer2 را محاسبه و نتیجه را توسط عملگر انتساب (=) در متغیر sum قرار می دهد.

```
cout << "Sum is " << sum << endl;
```

و در نهایت دستور فوق باعث نمایش حاصل جمع بر روی صفحه نمایش می شود.

ساختار انتخاب if

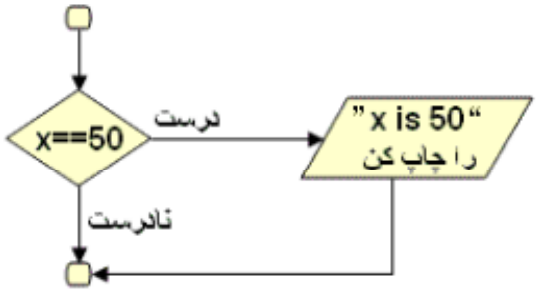
در برنامه نویسی مواردی پیش می آید که بخواهیم دستور یا دستوراتی، هنگامی که شرط خاصی برقرار است، توسط برنامه به اجرا در آید. این مورد در زندگی روزمره نیز دیده می شود؛ به عنوان مثال " اگر فردا باران نیاید، من به کوه خواهم رفت." شرط مورد نظر نیامدن باران است و عملی که قرار است انجام شود رفتن به کوه می باشد. شیوه پیاده سازی ساختار انتخاب **if** به صورت زیر می باشد:

```
if ( شرط مورد نظر )
        دستور مورد نظر ;
```

به مثال زیر توجه کنید:

```
if (x == 50)
    cout << "x is 50";
```

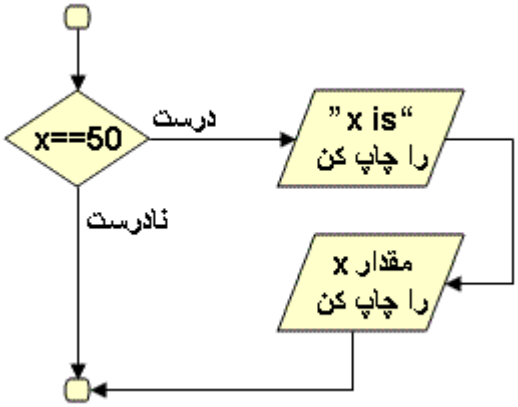
اگر از دستور فوق در برنامه استفاده کنیم، اگر مقدار متغیر **x** قبل از رسیدن به شرط فوق برابر **۵۰** باشد عبارت " **x is 50**" بر روی صفحه نمایش ظاهر خواهد شد وگرنه دستور **cout << "x is 50" ;** نادیده گرفته می شود و برنامه خط بعدی را اجرا می کند.



توجه داشته باشید که شرط مورد استفاده در دستور **if** هر عبارت منطقی می تواند باشد. در مبحث عبارات منطقی ، اینگونه عبارات و شیوه کاربرد آنها را به طور کامل بررسی کردیم. اگر بخواهیم هنگامی که شرط برقرار می شود، بیش از یک دستور اجرا شود، باید دستورات مورد نظر را با علامت { } دسته بندی کنیم، به مثال زیر توجه کنید:

```
if ( x==50 )
  {
    cout << "x is ";
    cout << x;
  }
```

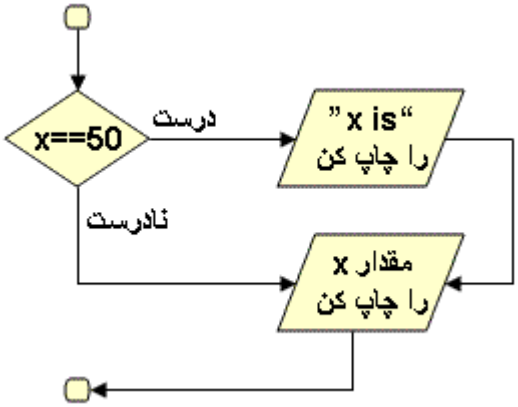
قطعه کد فوق هنگامی که مقدار **x** عدد ۵۰ باشد، عبارت "**x is 50**" را در صفحه نمایش چاپ می کند.



ولی در دستورات زیر:

```
if ( x == 50)
     cout << "x is ";
     cout << x ;
```

خط آخر برنامه به هر جهت اجرا می شود. به عنوان مثال اگر فرض کنیم **x** برابر ۵۰ است برنامه به درستی عبارت " **x is 50**" را چاپ می کند، اما اگر مثلاً **x** برابر ۲۰ باشد عدد ۲۰ بر روی صفحه نمایش ظاهر خواهد شد. چون عبارت **cout; x>>** جز دستورات **if** قرار ندارد و یک دستور مجزا می باشد.



مورد اخیر که توضیح داده شد یکی از مواردی است که بعضی از برنامه نویسان به اشتباه مرتکب آن می شوند. پس در هنگام نوشتن برنامه های خود به دسته بندی دستورات دقت کنید.

ساختار انتخاب if/else

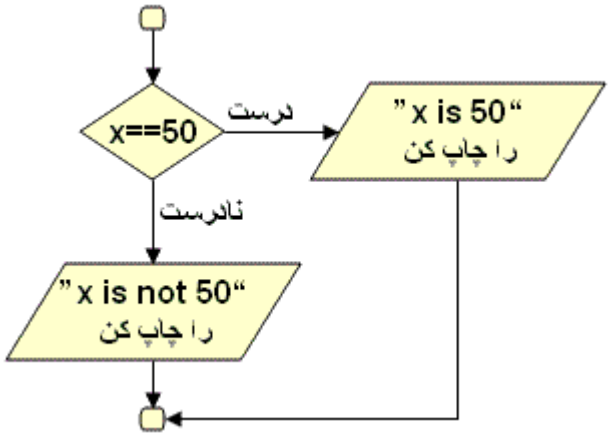
گاهی اوقات نیاز داریم که در صورت برقرار بودن شرط خاصی یک سری دستورات اجرا و در صورت برقرار نبودن شرط دسته ای دیگر از دستورات اجرا گردند. به عنوان مثال اگر فردا باران بیاید من به کوه نمی روم در غیر این صورت من به کوه خواهم رفت؛ زبان C++ برای پیاده سازی چنین ساختاری شیوه زیر را در اختیار ما قرار داده است.

```
if  (شرط مورد نظر)
        دستور۱  ;
else
        دستور۲  ;
```

اگر شرط برقرار باشد دستور۱ اجرا می گردد و در غیر این صورت دستور۲ اجرا می شود. به مثال زیر توجه کنید:

```
if ( x = = 50 )
     cout << "x is 50";
else
     cout << "x is not 50";
```

اگر مقدار متغیر قبل از رسیدن به شرط فوق برابر ۵۰ باشد عبارت "**x is 50**" برروی صفحه نمایش چاپ می شود در غیر این صورت عبارت "**x is not** ۵۰" چاپ می شود.



بیاد داشته باشید اگر می خواهید از بیش از یک دستور استفاده کنید، حتماً آنها را با { } دسته بندی نمایید. به عنوان مثال:

```
if ( x > 50 )
   {
     cout << x;
     cout << "is greater than 50";
```

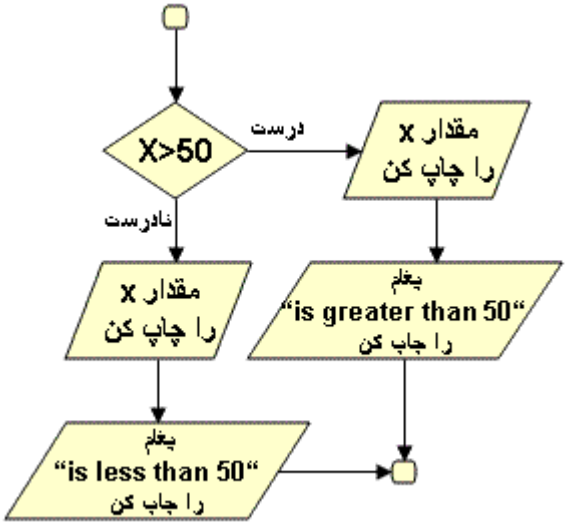
```
   }
else
   {
     cout << x;
     cout << "is less than 50";
   }
```

اگر متغیر **x** حاوی عدد ۱۰۰ باشد خروجی به صورت زیر می باشد:

```
100 is greater than 50
```

و اگر متغیر **x** عدد ۱۰ باشد خروجی به صورت زیر است:

```
10 is less than 50
```



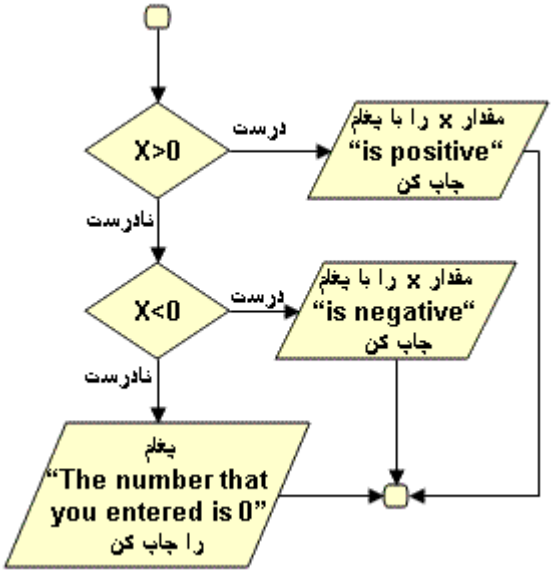
از ساختارهای **if/else** های تو در تو می توان برای بررسی حالتهای چندگانه استفاده کرد. برنامه زیر در همین رابطه است:

```
#include <iostream.h>
int main( )
{
  int x;
  cout << "Please enter a number:";
  cin >> x;

  if ( x > 0 )
     cout << x << "is positive.";
```

```
  else
     if ( x < 0 )
        cout << x << "is negative.";
     else
        cout << "The number that you entered is 0.";
  return 0;
}
```

برنامه فوق را سه بار با سه عدد مختلف اجرا می کنیم. خروجی ها به صورت زیر می باشند:

```
Please enter a number : 10
10 is positive.
Please enter a number : -5
-5 is negative.
Please enter a number : 0
The number that you entered is 0.
```



نکته : برای وضوح برنامه پیشنهاد می شود همانند برنامه فوق هنگام استفاده از **if** یا **if/else** و یا دیگر ساختارهای کنترلی از تورفتگی های مناسب استفاده کنید. یعنی به عنوان مثال دستور **if** را به صورت زیر:

```
if ( x > 0 )
    cout << x << "is positive.";
```

بنویسیم و نه به صورت زیر :

```
if ( x > 0 )
cout << x << "is positive.";
```

ساختار چند انتخابی switch

در دو مبحث قبلی ساختارهای **if** و **if/else** را بررسی کردیم. در برنامه نویسی گاهی به الگوریتمی نیاز پیدا می کنیم که در آن متغیری به ازای هر مقدار صحیح ثابتی، باعث اجرای یک دستور خاص شود و به ازای هر مقدار اعمال مختلف انجام پذیرد. برای نیل به این هدف C++ ساختار چند انتخابی **switch** را که به صورت زیر می باشد در اختیار ما قرار داده است:

```
switch  (عبارتی که باید مورد بررسی قرار گیرد)
   {
      case مقدار ثابت ۱ :
                  مجموعه دستورات ۱
                  break;
      case مقدار ثابت ۲ :
                  مجموعه دستورات ۲
                  break;

      .
      .
      .

      case  مقدار ثابت n :
                  مجموعه دستورات n
                  break;
      default :
           مجموعه دستورات حالت پیش فرض
 }
```

ساختار **switch** به شیوه زیر عمل می کند:

switch ابتدا عبارت داخل پرانتز را مورد ارزیابی قرار می هد و سپس آن را با **مقدار ثابت ۱** مورد مقایسه قرار می دهد. اگر برابر بودند **مجموعه دستورات ۱** را اجرا خواهد شد، تا هنگامی که برنامه به دستور **break** برسد، هنگامی که برنامه به دستور **break** رسید از ساختار چند انتخابی **switch** خارج می شود. اگر عبارت داخل پرانتز با **مقدار ثابت ۱** برابر نبود ساختار **switch** عبارت داخل پرانتز را با **مقدار ثابت ۲** مورد مقایسه قرار می دهد، در صورت برابر بودن **مجموعه**

دستورات ۲ اجرا می گردد. این روال همینطور ادامه پیدا می کند. در صورتی که عبارت داخل پرانتز با هیچ یک از مقادیر ثابت برابر نباشد، مجموعه دستورات حالت **default** (پیش فرض) اجرا می گردد. به برنامه زیر توجه کنید:

```
#include <iostream.h>
int main( )
{
  int x;
  cout << "Please enter a number:";
  cin >> x;

  switch (x) {
    case 1:
      cout << "x is 1";
      break;
    case 2:
      cout << "x is 2";
      break;
    default:
      cout << "Unknown value";
  }
  return 0;
}
```

برنامه فوق را سه بار با سه عدد مختلف اجرا می کنیم. خروجی ها به صورت زیر می باشند:

```
Please enter a number:1
x is 1
Please enter a number:2
x is 2
Please enter a number:5
Unknown value
```

توجه داشته باشید که ساختار **switch** را می توان با ساختار **if/else** نیز پیاده سازی کرد. به عنوان مثال ساختار **switch** به کار رفته در مثال فوف معادل ساختار **if/else** زیر می باشد:

```
if (x == 1)
    cout << "x is 1";
else
    if (x == 2)
        cout << "x is 2";
    else
       cout << Unknown value";
```

ما الزامی به استفاده از حالت **default** در ساختار **switch** نداریم ولی توصیه می شود که حالت پیش فرض را به کار ببریم چون معمولاً امکان دارد که عبارت برابر با هیچ یک از مقادیر ثابت نباشد و با به کار بردن حالت پیش فرض می توانید پیغام مناسبی در این رابطه به صفحه نمایش بفرستید.

توجه داشته باشید اگر دستور **break** بعد از هر مجموعه از دستورات استفاده نکنیم برنامه از ساختار **switch** خارج نخواهد شد و مجموعه دستورات بعدی اجرا می گردد تا به اولین دستور **break** برسد. این مورد به ما امکان ایجاد حالتهای ترکیبی را می دهد. البته در به کار بردن این تکنیک دقت لازم را بکنید.

```
#include <iostream.h>
int main( )
{
  int x;
  cout << "Please enter a number:";
  cin >> x;

  switch (x) {
    case 1:
    case 2:
    case 3:
      cout << "x is (1 or 2 or 3)";
      break;
    default:
      cout << "x is not (1 or 2 or 3)";
    }
  return 0;
}
```

برنامه فوق را سه بار با سه عدد مختلف اجرا می کنیم. خروجی ها به صورت زیر می باشند:

```
Please enter a number:1
x is (1 or 2 or 3)
Please enter a number:2
x is (1 or 2 or 3)
Please enter a number:5
x is not (1 or 2 or 3)
```

ساختار تکرار while

ساختار تکرار (حلقه تکرار) به برنامه نویس این امکان را می دهد که برنامه ، قسمتی از دستورات را تا هنگامی که شرط خاصی برقرار است، را تکرار کند. به عنوان مثال :

تا وقتی که مورد دیگری در لیست خرید من هست.
آن را بخر و از لیست خرید حذفش کن.

مورد فوق روال یک خرید را انجام می دهد. شرط مورد نظر " مورد دیگری در لیست خرید من هست" می باشد، که ممکن است درست یا نادرست باشد. اگر شرط برقرار باشد (یعنی مورد دیگری در لیست خرید باشد) عمل "خرید آن و حذفش از لیست" انجام می گیرد. این عمل تا وقتی که شرط برقرار باشد ادامه می یابد. هنگامی که شرط برقرار نباشد (یعنی تمام موارد لیست خرید حذف شده باشند)، ساختار تکرار به پایان می رسد و اولین دستور بعد از حلقه تکرار، اجرا می گردد. ساختار تکرار **while** به صورت زیر می باشد.

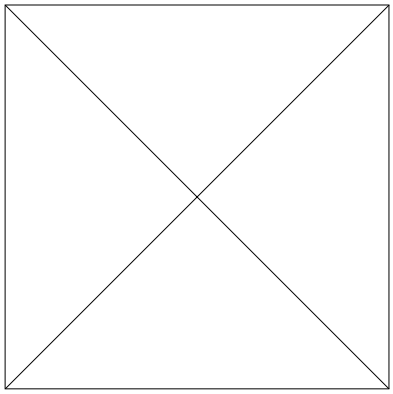
```
while  ( شرط مورد نظر )
       {
           مجموعه دستورات
       }
```

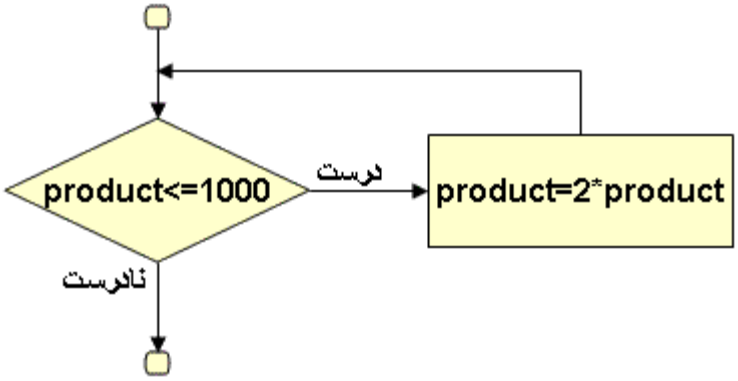
تا وقتی که شرط داخل پرانتز برقرار باشد مجموعه دستورات اجرا خواهند شد. برای درک بهتر شیوه کاربرد حلقه های تکرار فرض کنید می خواهیم اولین توانی از عدد **۲** که بزرگتر از **۱۰۰۰** می باشد را بیابیم. برنامه به صورت زیر خواهد بود.

```
#include <iostream.h>
int main( )
 {
  int product = 2;
  while (product <= 1000)
       product = 2 * product;

  cout << "The first power of 2 larger than 1000 is "
       <<product <<endl;
  return 0;
 }
```

در برنامه فوق ابتدا متغیری به نام **product** را با مقدار اولیه **۲** مقدار دهی کردیم. در حلقه تکرار **while** با هر بار اجرای دستور **product=2*product** مقدار متغیر **product** دو برابر می شود بدین ترتیب با پایان یافتن حلقه متغیر **product** حاوی عدد **۱۰۲۴** یعنی اولین توانی از **۲** که بزرگتر از **۱۰۰۰** می باشد، خواهد بود.

```
The first power of 2 larger than 1000 is 1024
```





نکته : در مثال فوق در حلقه **while** چون تنها از یک دستور استفاده شده بود از {} استفاده نشد، ولی اگر بیش از یک دستور داشتیم ملزم به استفاده از {} بودیم.

مثال : برنامه ای بنویسید تا مجموع اعداد یک تا صد را محاسبه کند.

```
#include <iostream.h>
int main( )
  {
    int n=1, sum=0;
    while (n <= 100)
      {
        sum += n;        // sum = sum + n;
        ++n;             // n = n + 1;
      }
    cout << "1 + 2 + ... + 100 =" <<sum << endl;
    return 0;
  }
```

در مثال فوق حلقه ۱۰۰ بار اجرا می گردد و هر بار عدد **n** به متغیر **sum** اضافه می گردد و عدد **n** نیز یک واحد افزایش می یابد تا در یکصدمین بار اجرای حلقه مقدار متغیر **n** برابر ۱۰۱ می شود و هنگام بررسی شرط (**n** <=۱۰۰) توسط حلقه **while** شرط نادرست می شود و اولین دستور بعد از حلقه یعنی دستور خروجی **cout** اجرا می گردد.

```
1 + 2 + ... + 100 =5050
```

نکته :

- در مثال فوق متغیر **n** به عنوان شمارنده دفعات تکرار حلقه بکار گرفته شد. برحسب مورد شمارنده ها معمولاً با یک یا صفر مقدار دهی اولیه می شوند.
- متغیر **sum** حاوی مجموع حاصلجمع بود. چنین متغیرهایی که برای محاسبه یک حصلجمع به کار می روند معمولاً با صفر مقدار دهی اولیه می شوند.

مثال : برنامه ای بنویسید که تعداد نامشخصی عدد مثبت را از ورودی دریافت نماید و میانگین آنها را محاسبه نماید. عدد ۱- را برای مشخص کردن انتهای لیست اعداد در نظر بگیرید.

```
#include <iostream.h>
int main( )
{ int num, counter = 0;
  float average, sum = 0;

  cout << "Enter a number (-1 to end):";
  cin >>num;

  while (num != -1){
    sum += num ; // sum = sum + sum;
    ++counter;
    cout << "Enter a number (-1 to end):";
    cin >> num;
  }

  if (counter != 0){
    average = sum / counter;
    cout << "The average is " << average << endl;
  }
  else
    cout << "No numbers were entered." << endl;

  return 0;
}
```

خروجی برنامه به صورت زیر خواهد بود.

```
Enter a number (-1 to end):20
Enter a number (-1 to end):50
Enter a number (-1 to end):65
Enter a number (-1 to end):70
Enter a number (-1 to end):90
Enter a number (-1 to end):100
Enter a number (-1 to end):1
Enter a number (-1 to end):6
Enter a number (-1 to end):-1
The average is 50.25
```

در برنامه مثال قبل عدد ۱- به عنوان یک مقدار کنترلی به کار می رود و با وارد کردن این عدد اجرای برنامه به پایان می رسد و میانگین اعداد در خروجی به نمایش در می آید. متغیر **num** اعداد را از ورودی دریافت می کند. متغیر **counter** وظیفه شمارش تعداد اعداد وارد شده را دارا می باشد و متغیر **sum** مجموع حاصلجمع اعداد را در خود قرار می دهد و در نهایت متغیر **average**، میانگین را در خود قرار می دهد. ساختار کنترلی **if** به کار رفته در برنامه، جلوی بروز خطای زمان اجرای تقسیم بر صفر را می گیرد ، یعنی اگر در اولین دستور **cin** به کار رفته عدد ۱- وارد شود خروجی برنامه به صورت زیر خواهد بود :

```
Enter a number (-1 to end): -1
No numbers were entered.
```

ساختار تکرار do/while

ساختار تکرار **do/while** مشابه ساختار تکرار **while** می باشد. در ساختار تکرار **while** شرط حلقه در ابتدا بررسی می شود ولی در ساختار تکرار **do/while** شرط در انتهای حلقه مورد بررسی قرار می گیرد، بدین ترتیب در ساختار تکرار **do/while** دستورات حلقه حداقل یکبار اجرا خواهند شد. ساختار تکرار **do/while** به صورت زیر می باشد:

```
do {
      مجموعه دستورات
    }while ( شرط مورد نظر );
```

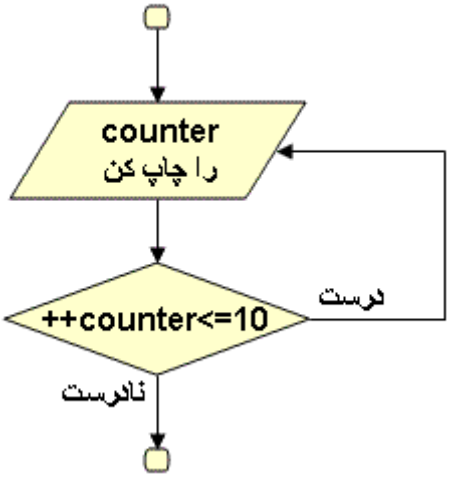
به عنوان مثال به برنامه زیر توجه نمایید:

```
#include <iostream.h>
```

```
int main()
{
   int counter = 1;
   do {
       cout << counter << " ";
      }while ( ++counter <= 10 );
   cout << endl;

   return 0;
}
```

در این برنامه اعداد ۱ تا ۱۰ با فاصله بر روی صفحه نمایش چاپ خواهند شد. دقت کنید که متغیر **counter** در قسمت شرط حلقه ، یک واحد اضافه می گردد سپس مقدارش با عدد ۱۰ مقایسه می گردد.

```
1 2 3 4 5 6 7 8 9 10
```



مثال: برنامه ای بنویسید که لیست نمرات یک کلاس را دریافت کرده و تعداد قبولی ها و مردودی ها را مشخص کند. ضمنا در ابتدای برنامه تعداد نمرات لیست پرسیده شود.

```
#include <iostream.h>
int main( )
{
  float mark;
  int howmany,counter=1;
  int passes=0,failures=0;

  cout << "How many marks : ";
  cin >> howmany;
```

```
  do {
      cout << "Enter mark "<<counter<<" : ";
      cin>>mark;
      if (mark>=10)
         ++passes;
      else
         ++failures;
     }while (++counter <= howmany);

  cout<<"Passed : "<<passes<<endl;
  cout<<"Failed : "<<failures<<endl;

  return 0;
}
```

خروجی برنامه به صورت زیر می باشد :

```
How many marks : 10
Enter mark 1 : 18
Enter mark 2 : 15
Enter mark 3 : 9
Enter mark 4 : 17.5
Enter mark 5 : 9.75
Enter mark 6 : 8
Enter mark 7 : 11
Enter mark 8 : 13
Enter mark 9 : 5
Enter mark 10 : 13
Passed : 6
Failed : 4
```

ساختار تکرار for

ساختار تکرار **for** نیز مانند دو ساختار قبلی یک حلقه تکرار می سازد. از ساختار تکرار **for** معمولاً هنگامی که دفعات تکرار حلقه براساس یک شمارنده می باشد استفاده می شود. ساختار تکرار **for** به صورت زیر می باشد:

```
for (  افزایش یا کاهش  ;  شرط حلقه  ;  تعریف متغیر  )
       مقدار شمارنده                  شمارنده و تعیین
       مقدار اولیه
```

```
      {
            مجموعه دستورات
      }
```

ساختار تکرار **for** را با ساختار تکرار **while** نیز می توان پیاده سازی کرد به عنوان مثال دو برنامه زیر اعداد **۱** تا **۵** را بر روی صفحه نمایش چاپ می کنند:

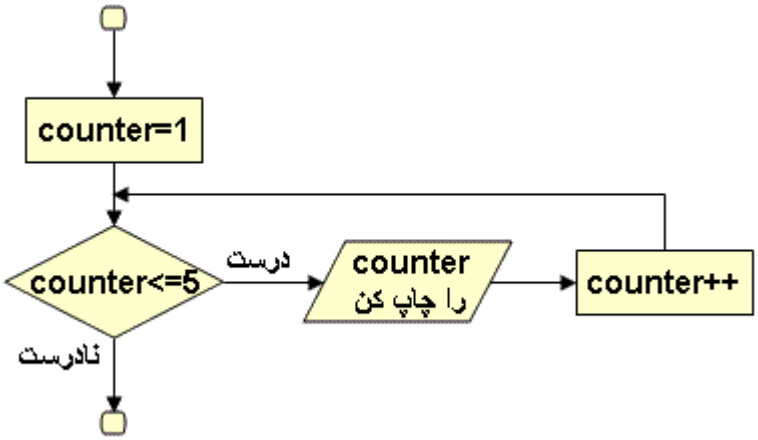
```
#include <iostream>

int main()
{
  int counter = 1;

  while ( counter <= 5 ) {
     cout << counter << endl;
     ++counter;
  }
  return 0;
}
```

برنامه فوق با حلقه **while** نوشته شده بود. در برنامه زیر معادل حلقه **while** فوق را با حلقه **for** پیاده سازی می کنیم:

```
#include <iostream>

int main()
{

  for ( int counter = 1; counter <= 5; counter++ )
     cout << counter << endl;

return 0;
}
```

در برنامه فوق هنگامی که دستور **for** اجرا می شود متغیر کنترلی **counter** تعریف می گردد و عدد **۱** در آن قرار می گیرد. سپس شرط حلقه مورد بررسی قرار می گیرد (**counter**<=۵) چون مقدار **counter** ، عدد **۱** می باشد پس شرط درست است و دستور حلقه، یعنی دستور **cout** اجرا می گردد و اولین عدد یعنی **۱** نمایش داده می شود. پس از آن دستور ++**counter** اجرا می گردد و مقدار متغیر **counter** یک واحد اضافه می شود. سپس مجدداً شرط حلقه بررسی و در صورت برقرار بودن شرط دستور **cout** اجرا می گردد. این روال تا وقتی که شرط برقرار باشد ادامه می یابد و به محض برقرار نبودن شرط یعنی هنگامی که **counter** حاوی عدد **۶** شود خاتمه می یابد و برنامه به پایان می رسد.

```
1 2 3 4 5
```



در برنامه قبلی اگر حلقه **for** را به صورت زیر بازنویسی کنیم:

```
for(int counter=10; counter>=1; counter=counter-2)
```

خروجی برنامه اعداد زوج **۱۰** تا **۱** به صورت معکوس می باشد، یعنی :

```
10 8 6 4 2
```

توجه داشته باشید که در حلقه فوق به جای استفاده از دستور **counter=counter-1** می توانستیم از دستور **counter -= 2** استفاده کنیم.

مثال : برنامه ای بنویسید که مجموع اعداد زوج **۱** تا **۱۰۰** را محاسبه کند.

```
#include <iostream.h>

int main ( )
{ int sum = 0;

  for (int num = 2; num <= 100; num += 2)
     sum += num;
  cout << "2 + 4 + 6 + ... + 100 =" <<sum<<endl;

  return 0;
}
```

```
2 + 4 + 6 + ... + 100 =2550
```

توجه داشته باشید که حلقه for در برنامه فوق را با کمک عملگر کاما (,) می توانیم به صورت زیر نیز بنویسیم:

```
for (int num = 2;
     num <= 100;
     sum += num, num +=2);
```

ضمناً شکستن حلقه به چند خط نیز مشکلی ایجاد نمی کند. البته دو مورد اخیر توصیه نمی شوند، چون از خوانایی برنامه می کاهند.

مثال : برنامه ای بنویسید که عددی را از ورودی دریافت کرده و **۲** به توان آن عدد را محاسبه و در خروجی چاپ نماید.

```
#include <iostream.h>
int main( )
{
  unsigned long int x=1;
  int power;

  cout << "Enter power:";
  cin >>power;

  for (int counter=1;counter<=power;counter++)
    x*=2;

  cout << "2 ^ " << power << " = " << x <<endl;

return 0;
}
```

```
Enter power:25
2 ^ 25 = 33554432
```

در مثال های فوق، دستورات حلقه **for** را داخل { } قرار ندادیم چون حلقه **for** تنها شامل یک دستور بود، توجه داشته باشید که اگر بیش از یک دستور در حلقه به کار رود ملزم به استفاده از { } می باشیم.

مثال : برنامه ای بنویسید که جدول ضرب 5X5 را ایجاد کند.

```
#include <iostream.h>
int main( )
{

  for (int x=1;x<=5;x++)
   {
```

```
      for (int y=1;y<=5;y++)
         cout <<x*y<<"\t";
      cout<<endl;
    }

  return 0;
}
```

خروجی برنامه به صورت زیر خواهد بود:

```
1     2     3     4     5
2     4     6     8     10
3     6     9     12    15
4     8     12    16    20
5     10    15    20    25
```

در برنامه فوق حلقه شامل متغیر **x** ، دارای دو دستور **for** و **cout** بود، به همین علت از { } استفاده شد. اما حلقه شامل متغیر y تنها دارای یک دستور **cout** بود. اگر دقت کرده باشید دستور **t\">>x*y>>cout";** دارای علامت **\t** بود. به کار بردن **\t** باعث جدول بندی و مرتب شدن خروجی می شود. در حقیقت مکان نمای صفحه نمایش را در محل جدول بندی قرار می دهد. ضمناً در مثال فوق یک ساختار **for** در دل ساختار **for** دیگری استفاده شد به این شیوه استفاده حلقه های تودرتو گفته می شود که در برنامه نویسی ها به کرات از آنها استفاده می شود. در ضمن توجه داشته باشید که اگر از دستور **endl>> cout;** استفاده نشود، خروجی به صورت نا مرتب زیر خواهد بود:

```
1     2     3     4     5     2     4     6     8     10
3     6     9     12    15    4     8     12    16    20
5     10    15    20    25
```

نکته : در حلقه های تکرار ممکن است شرط حلقه را به اشتباه بنویسیم یا به عنوان مثال شمارنده حلقه را افزایش ندهیم در چنین حالاتی ممکن است پس از اجرای برنامه، برنامه به اتمام نرسد و حلقه همچنان تکرار شود. در صورت بروز چنین مشکلی با فشردن کلید **Ctrl** همراه با **Break** (**Ctrl+Break**) اجرای برنامه به صورت ناگهانی قطع می شود و به محیط ویرایشگر C++ باز می گردید و می توانید کد اشتباه را درست کنید. سپس برنامه را مجدداً اجرا کنید.

دستور های break و continue

دستور **break** هرگاه که در ساختارهای **while** و **do/while** و **for** یا **switch** اجرا گردد، باعث خروج فوری برنامه از آن ساختار خواهد شد و برنامه اولین دستور بعد از آن ساختار را اجرا خواهد کرد. به برنامه زیر توجه کنید:

```
#include <iostream.h>
```

```
int main()
{
  int n;
  for (n=10; n>0; n--)
  {
     cout << n << ",";
     if (n==3)
     {
       cout << "countdown aborted!";
       break;
     }
  }
  return 0;
}
```

خروجی برنامه به صورت زیر می باشد:

```
10,9,8,7,6,5,4,3,countdown aborted!
```

برنامه فوق اعداد **۱۰** تا **۴** را چاپ خواهد کرد و هنگامی که متغیر **n** عدد **۳** می شود، شمارش معکوس به پایان می رسد.

نکته : در برنامه فوق شمارنده حلقه یعنی **n** در خارج از دستور **for** تعریف شد. در چنین حالتی ، این متغیر خارج از حلقه نیز می تواند مورد استفاده قرار گیرد ولی اگر تنها در داخل حلقه تعریف شده بود ، تنها آنجا می توانستیم از آن استفاده کنیم و خارج حلقه تعریف نشده بود.

دستور **continue** هرگاه در ساختارهای **while** و **do/while** یا **for** اجرا گردد دستورات بعدی آن ساختار نادیده گرفته می شود و بار بعدی حلقه تکرار اجرا می شود. در دو ساختار **while** و **do/while** پس از اجرای دستور **continue** شرط حلقه مورد بررسی قرار می گیرد، اما در ساختار **for** ابتدا مقدار شمارنده افزایش یا کاهش می یابد، سپس شرط حلقه بررسی می شود. توجه داشته باشید که در حلقه **while** و**do/while** دستور **continue** همواره بعد از افزایش یا کاهش شمارنده به کار رود. به عنوان مثال برنامه زیر مجموع اعداد **۱** تا **۲۰** به جز **۱۰** را محاسبه می کند.

```
#include <iostream.h>

int main( )
{
  int n=0, sum=0;
  while (n < 20)
  {
    ++n; // n = n + 1;
    if (n==10) continue;
    sum += n; // sum = sum + n;
```

```
  }
  cout << "1+2+ ...(except 10)...+20=" <<sum << endl;

  return 0;
}
```

خروجی برنامه به صورت زیر می باشد.

```
1+2+ ...(except 10)...+20=200
```

برنامه عمل جمع را تا رسیدن به عدد ۱۰ ادامه می دهد به محض اینکه **n** برابر ۱۰ می شود دوباره به شرط حلقه منتقل می شود و چون شرط همچنان برقرار است وارد حلقه شده و **n** یک واحد افزایش می یابد و جمع اعداد ادامه می یابد.

توابع ریاضی

مفهوم تابع یکی از مهمترین مفاهیم در ریاضیات و علوم کامپیوتر و نیز سایر علوم می باشد. تابع را می توان به عنوان دستگاهی در نظر گرفت که ورودیهای مجازش را با تغییراتی که وظیفه آن دستگاه می باشد به خروجی متناظر با ورودی تبدیل می کند. معادله خط **y = 2x + 1** را در نظر بگیرید، اگر به جای **f(x**)، **y** را قرار دهیم معادله خط فوق به صورت **f(x) = 2x + 1** در خواهد آمد. در اینجا دستگاه ما تابع **f** خواهد بود که هر ورودی (هر عدد حقیقی) را در ۲ ضرب می کند و سپس یک واحد به آن اضافه می کند. شکل دستگاه به صورت زیر می باشد.

به عنوان مثال :

```
f(0) = 2 * (0) + 1 = 1
f(1) = 2 * (0) + 1 = 3
f(-1) = 2 * (-1) + 1 = -1
```

همانطور که در مثالهای فوق دیدید تابع **f** به هر ورودی تنها یک خروجی را نظیر می کند.

مثال : تابعی بنویسید که شعاع یک دایره به عنوان ورودی باشد و خروجی ، مساحت دایره باشد.

می دانیم که فرمول مساحت دایره $s = 3.14*r^2$ می باشد پس تابع را به صورت زیر تعریف می کنیم:

$$s(r) = 3.14*r^2$$

بعضی از توابع ممکن است بر اساس شرط خاصی خروجی متفاوتی داشته باشند، اینگونه توابع معمولاً به صورت چند ضابطه ای تعریف می شوند. به عنوان مثال تابع قدر مطلق به صورت زیر می باشد:

$$abs(x) = |x| = \begin{cases} x & , x >= 0 \text{ اگر} \\ -x & , x < 0 \text{ اگر} \end{cases}$$

abs(1)= محاسبه خروجی

به عنوان نمونه :

```
abs(1) = 1
abs(-1) = 1
abs(0) = 0
```

توابع همواره یک مقدار را به عنوان ورودی دریافت نمی کنند، بلکه ممکن است بیش از یک مقدار را به عنوان ورودی دریافت کنند. به عنوان مثال تابع زیر را در نظر بگیرید :

$$f(x,y) = x + y$$

تابع فوق هر دو عددی که به عنوان ورودی به آن داده می شوند را با هم جمع کرده و به عنوان خروجی بر می گرداند. به عنوان مثال :

f(1 , 1)=x+y= محاسبه خروجی

```
f(1,2) = 1+2 = 3
f(2,3) = 2+3 = 5
f(1,-1) = 1+(-1) = 0
```

به توابعی مانند تابع فوق توابع دو متغیره گفته می شود و اگر تعداد متغیرها سه تا باشد، تابع سه متغیره و ... و چند متغیره گفته می شود. به متغیرهای ورودی تابع آرگومان نیز گفته می شود.

زبان **C++** برای انجام محاسبات ریاضیاتی ، توابع کاربردی فراوانی را در اختیار ما قرار داده است ، به عنوان مثال فرض کنید که می خواهید جذر یک عدد را بدست آورید، تابعی که زبان **C++** برای اینکار در اختیار ما قرار داده است، تابع **sqrt** می باشد. به عنوان مثال دستور زیر :

```
cout << sqrt (900);
```

عدد ۳۰ را چاپ خواهد کرد. در اینجا عدد ۹۰۰ آرگومان تابع **sqrt** می باشد. برای استفاده از توابع ریاضی در برنامه ملزم به استفاده از دستور:

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

در ابتدای برنامه می باشیم، چون توابع ریاضی در فایل کتابخانه ای **math.h** قرار دارند. آرگومانهای توابع می توانند شامل اعداد ثابت، متغیرها و یا ترکیبی از آنها باشند؛ به عنوان مثال به برنامه زیر توجه کنید :

```
#include <iostream.h>
#include <math.h>
int main ( )
{
    int x = 30;
    double y = 5;
    cout << sqrt (x+2*y+9)<<endl;
    return 0;
}
```

خروجی برنامه فوق عدد ۷ خواهد بود چون تابع **sqrt** جذر عبارت **30+2*5+9=49** را محاسبه خواهد کرد.

تعدادی از توابع ریاضی مورد استفاده در جدول زیر توضیح داده شده اند.

fabs(x)	این تابع، قدر مطلق **x** را حساب می کند. **fabs(1.2) = 1.2** **fabs(-1.2) = 1.2** **fabs(0) = 0** fabs(-1)= [ ] محاسبه خروجی
sqrt(x)	این تابع ، جذر **x** را محاسبه می کند. **sqrt(9) = 3** **sqrt(2) = 1.414214** sqrt(9)= [ ] محاسبه خروجی
pow(x,y)	این تابع ، **x** به توان **y** را محاسبه می کند. **pow(2, 5) = 32** **pow(2, 0.5) = 1.414214** pow(2 , 3)= [ ] محاسبه خروجی
exp(x)	این تابع ، مقدار عبارت **e** را محاسبه می کند.**e** یک ثابت ریاضی با مقدار تقریبی ۲,۷۱۸۲۸ می باشد. **exp(2) = 7.38906** exp(2)= [ ] محاسبه خروجی
log(x)	این تابع ، لگاریتم طبیعی عدد **x** را حساب می کند . لگاریتم طبیعی، لگاریتم بر مبنای **e** می باشد. این تابع بر عکس تابع **exp** عمل می کند. **log(2.71828) = 1** **log(7.389050) = 2** log(2.718)= [ ] محاسبه خروجی
log10(x)	این تابع ، لگاریتم بر مبنای ۱۰ عدد **x** را محاسبه می کند.مثلاً **log10(100)** یعنی ۱۰ به توان چه عددی برسد تا حاصل برابر ۱۰۰ شود که جواب ۲ خواهد بود پس **log10(100)=2** **log10(10) = 1** **log10(1000) = 3** log10(10)= [ ] محاسبه خروجی

fmod(x,y)	این تابع ، باقیمانده تقسیم دو عدد اعشاری را حساب می کند. **x/y** **fmod(13.657,2.333) = 1.992** fmod(5.1 , 4.2)= محاسبه خروجی
ceil(x)	این تابع ، کوچکترین عدد صحیح بزرگتر مساوی **x** را به عنوان خروجی بر می گرداند. **ceil(9) = 9** **ceil(9.2) = 9** **ceil(-9.8) = -9** ceil(2.4)= محاسبه خروجی
floor(x)	این تابع ، همانند تابع جزء صحیح ریاضیات عمل می کند.یعنی بزرگترین عدد صحیح کوچکتر مساوی **x** را بر می گرداند. **floor(7) = 7** **floor(7.3) = 7** **floor(-7.2) = -8** floor(2.4)= محاسبه خروجی
sin(x)	این تابع ، **sin** عدد **x** را حساب می کند.(**x** باید بر اساس رادیان باشد) **sin(0) = 0** **sin(3.14/2) = 1** sin(1.57( )= محاسبه خروجی
cos(x)	این تابع ، **cos** عدد **x** را محاسبه می کند.(**x** باید بر اساس رادیان باشد) **cos(0) = 1** **cos(3.14/2) = 0** cos(1.57( )= محاسبه خروجی
tan(x)	این تابع ، **tan** زاویه **x** را محاسبه می کند.(**x** باید بر اساس رادیان باشد) **tan(0) = 0** **tan(3.14/4) = 1**

tan(0.785)= محاسبه خروجی	
این تابع ، **arcsin** عدد **x** را محاسبه می کند. توجه داشته باشید که عدد **x** بین ۱- تا ۱ باشد. ضمناً خروجی تابع زاویه ای بر حسب رادیان می باشد.	**asin(x)**
asin(0) = 0 **asin(1) = 1.570796** asin(1)= محاسبه خروجی	
این تابع ، **arccos** عدد **x** را محاسبه می کند. توجه داشته باشید که عدد **x** بین ۱- تا ۱ باشد. ضمناً خروجی تابع زاویه ای بر حسب رادیان می باشد.	**acos(x)**
acos(1) = 0 **acos(0) = 1.570796** acos(1)= محاسبه خروجی	
این تابع ، **arctan** عدد **x** را محاسبه می کند. خروجی تابع زاویه ای بر حسب رادیان می باشد.	**atan(x)**
atan(1) = 0.785398 **atan(0) = 0** **atan(21584.891) = 1.570795** atan(1)= محاسبه خروجی	

نکته : در جدول فوق گفتیم که زاویه ها بر حسب رادیان می باشند. رادیان یکی از واحدهای اندازه گیری زاویه می باشد و با یک تناسب ساده می توان هر زاویه ای که بر حسب درجه می باشد را بر حسب رادیان محاسبه کرد.

$$\frac{R}{3.14} = \frac{D}{180^\circ}$$

در تناسب فوق به جای **D** اندازه زاویه مورد نظر را بنویسید و سپس با حل تناسب، **R** حاوی اندازه درجه بر حسب رادیان می باشد و بر عکس. به مثال های زیر توجه کنید :

$$R \quad 45^\circ$$

```
———— = ————    ====> R=0.785
3.14    180°

1.57     D
———— = ————    ====> D=90°
3.14    180°
```

مثال : برنامه ای بنویسید که **sin** و **cos** و**tan** زاویه های زوج **۱** تا **۹۰** درجه را در خروجی به صورت جدول بندی شده تا سه رقم اعشار چاپ نماید.

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

int main( )
{
  float r;
  for (int d=2;d<=90;d+=2)
  {
    r = 3.1415 * d / 180;
    cout<<"sin("<<d<<")="
        <<floor(sin(r)*1000 + 0.5)/1000;
    cout<<"\tcos("<<d<<")="
        <<floor(cos(r)*1000 + 0.5)/1000;
    cout<<"\ttan("<<d<<")="
        <<floor(tan(r)*1000 + 0.5)/1000;
    cout<<endl;
  }
  return 0;
}
```

خروجی برنامه به صورت زیر می باشد:

```
sin(2)=0.035    cos(2)=0.999    tan(2)=0.035
sin(4)=0.07     cos(4)=0.998    tan(4)=0.07
sin(6)=0.105    cos(6)=0.995    tan(6)=0.105
sin(8)=0.139    cos(8)=0.99     tan(8)=0.141
sin(10)=0.174   cos(10)=0.985   tan(10)=0.176
sin(12)=0.208   cos(12)=0.978   tan(12)=0.213
sin(14)=0.242   cos(14)=0.97    tan(14)=0.249

       .               .               .
       .               .               .
       .               .               .
```

```
sin(86)=0.998    cos(86)=0.07     tan(86)=14.292
sin(88)=0.999    cos(88)=0.035    tan(88)=28.599
sin(90)=1        cos(90)=0        tan(90)=21584.891
```

در برنامه فوق توسط فرمول **r = 3.1415 * d / 180** زاویه بر حسب درجه را به رادیان تبدیل کردیم و توسط فرمول **floor(sin(r)*1000 + 0.5)/1000** خروجی را تا سه رقم اعشار محاسبه کردیم. همانطور که می بینید ورودی تابع، علاوه بر متغیر و عدد ثابت خروجی تابع دیگری می باشد. به عنوان مثال **sqrt(pow(2,2))** برابر با **۲** خواهد بود و ترکیب توابع به این شکل کاملاً مجاز می باشد و بر حسب نیاز می توانید از این شیوه استفاده کنید.

تعریف توابع

اکثر برنامه های کاربردی ، خیلی بزرگتر از برنامه هایی می باشند که ما تا اکنون در مباحث قبلی نوشته ایم. تجربه نشان داده است که بهترین راه برای گسترش و نگهداری و ارتقاء برنامه های بزرگ، تقسیم کردن آنها به قطعات کوچکتر می باشد. هر کدام از قطعات راحتتر از برنامه اصلی مدیریت می شوند و اعمال تغییرات و خطایابی در آنها نیز ساده تر می باشد. قطعات مورد نظر ما در زبان **C++** همان توابع می باشند. اگر بیاد داشته باشید در مباحث قبلی گفتیم که **main** نیز یک تابع می باشد و این تابع نقطه ای است که برنامه اجرای دستورات را از آن آغاز می کند. زبان **C++** توابع آماده زیادی را در اختیار ما قرار داده که در فایلهای کتابخانه ای این زبان موجود می باشند، که گوشه ای از آنها را در مبحث قبل دیدید. در برنامه نویسی ممکن است که نیاز داشته باشیم مجموعه دستوراتی را عیناً در چند جای برنامه استفاده کنیم، در چنین حالتی بهتر است این مجموعه دستورات را در تابعی قرار دهیم و تابع را در برنامه چندین باز صدا بزنیم و از تکرار دستورات که تنها حجم برنامه اصلی را زیاد می کنند و از خوانایی آن می کاهند خودداری کنیم.

در زبان **C++** توابع به شیوه زیر تعریف می شوند :

```
نوع داده خروجی    نام تابع    (آرگومانهای تابع)
        {
            تعریف متغیرها
            دستورات تابع
        }
```

نام تابع از قواعد نام گذاری متغیرها پیروی می کند. برای آشنا شدن با نحوه تعریف توابع و شیوه به کار گیری آنها به برنامه زیر توجه کنید:

```
#include <iostream.h>

long int power2 (int x)
{
  long int y;
```

```
  y=x*x;

  return y;
}

int main ()
{
  for (int i=1;i<=10;i++)
    cout<<power2(i)<<" ";

  return 0;
}
```

خروجی برنامه مربع اعداد ۱ تا ۱۰ می باشد.

```
1 4 9 16 25 36 49 64 81 100
```

تابع **power2(x)** که در این برنامه نوشتیم تقریباً شبیه تابع **pow(x,2)** از توابع کتابخانه ای **C++** عمل می کند. ضابطه ریاضیاتی این تابع $f(x) = x^2$ می باشد. ورودی این تابع اعداد صحیح (**int**) و خروجی آن اعداد بزرگ (**long int**) می باشد. هنگامی که برنامه به تابع **power2(i)** می رسد تابع فراخوانی می شود و مقدار آرگومان **i** را دریافت می کند و در متغیر**x** قرار می دهد. سپس متغیر **y** تعریف می گردد و مقدار **x*x** در **y** قرار می گیرد و سرانجام مقدار **y** به عنوان خروجی تابع برگردانده می شود و توسط دستور **cout** چاپ می گردد. توجه داشته باشید که تابع تغییری در مقدار متغیر **i** ایجاد نمی کند و حلقه **for** تابع را ۱۰ بار فراخوانی می کند، ضمناً متغیرهای **x** و**y** در تابع **main** قابل استفاده نمی باشند و نیز متغیر **i** در تابع **power2** تعریف نشده است.

نکته : توجه داشده باشید که توابع داخل یکدیگر قابل تعریف نمی باشند و جدا از هم باید تعریف گردند.

مثال : تابعی بنویسید که سه عدد را به عنوان ورودی دریافت کرده و بزرگترین آنها را به عنوان خروجی برگرداند. این تابع را در برنامه ای به کار ببرید.

```
#include <iostream.h>

int maximum (int x,int y, int z)
{
  int max=x;

  if (y>max)
     max=y;
  if (z>max)
```

```
     max=z;

  return max;
}

int main ()
{
  int num1,num2,num3;

  cout << "Enter three numbers: ";
  cin >> num1 >> num2 >> num3;
  cout << "Max is :"
       << maximum(num1,num2,num3)<<endl;
  cout << "Max of 5,20,1 is "
       << maximum(5,20,1)<<endl;

  return 0;
}
```

```
Enter three numbers: -5 20 150
Max is :150
Max of 5,20,1 is 20
```

تابع **maximum** دارای سه آرگومان بود. هنگامی که اعداد **num1** و **num2** و **num3** در برنامه از ورودی دریافت می شوند با فراخوانی تابع **maximun(num1,num2,num3)** اعداد آرگومانهای **num1** و **num2** و **num3** به ترتیب در متغیر های **x** و **y** و **z** قرار می گیرند و مقادیر توسط تابع مقایسه می شوند و نهایتاً بزرگترین عدد در متغیر **max** قرار می گیرد که توسط دستور **return max;** به عنوان خروجی تابع برگردانده می شود. سپس دستور **cout** خروجی تابع را بر روی صفحه نمایش چاپ می کند.

پیش تعریف توابع

تا به حال توابع مورد استفاده در برنامه هایمان را قبل از اولین فراخوانی آنها تعریف کردیم و این فراخوانی معمولا در تابع **main** بود. لذا تابع **main** را به عنوان آخرین تابع در برنامه نوشتیم. اگر بخواهید که تابع **main** را قبل از هر تابع دیگری در برنامه بنویسید. هنگام اجرای برنامه یک پیغام خطا دریافت خواهید کرد. دلیل وقوع خطا این است که هنگامی که تابعی فراخوانی می شود باید قبلا تعریف شده باشد، مانند شیوه ای که ما در برنامه های قبلی استفاده کردیم.

یک راه چاره برای اجتناب از نوشتن کد همه توابع قبل از استفاده آنها در تابع **main** یا سایر توابع وجود دارد. این راهکار پیش تعریف توابع می باشد. پیش تعریف تابع به صورت زیر می باشد:

```
نوع داده خروجی    نام تابع    ( نوع آرگومانهای تابع ) ;
```

توجه داشته باشید که پیش تعریف تابع شامل دستورات تابع نمی شود و تنها شامل نوع داده خروجی ، نام تابع و نوع آرگومانها می باشد و در پایان نیاز به علامت (;) دارد. به عنوان مثال پیش تعریف تابع **power2** در مبحث قبلی به صورت زیر می باشد:

```
long int power2( int );
```

ویا پیش تعریف تابع **maximum** به صورت زیر است :

```
int maximum( int, int, int );
```

در اینجا برنامه تابع **maximum** در مبحث قبلی را با روش پیش تعریف تابع باز نویسی می کنیم :

```
#include <iostream.h>

int maximum (int ,int,int);

int main ()
{
  int num1,num2,num3;

  cout << "Enter three numbers: ";
  cin >> num1>>num2>>num3;
  cout << "Max is :"
       << maximum(num1,num2,num3)<<endl;
  cout << "Max of 5,20,1 is "
       << maximum(5,20,1)<<endl;

  return 0;
}

int maximum (int x,int y, int z)
{
  int max=x;

  if (y>max)
      max=y;
```

```
  if (z>max)
      max=z;

  return max;
}
```

```
Enter three numbers: -5 20 150
Max is :150
Max of 5,20,1 is 20
```

همانطور که در برنامه می بینید، تابع **main** قبل از تابع **maximum** نوشته شده است و این امکانی است که پیش تعریف تابع **maximum** به ما داده است.

نکته: در بعضی از برنامه ها، نیاز پیدا می کنیم که دو تابع یکدیگر را فراخوانی کنند. در چنین حالتی ملزم به استفاده از پیش تعریف تابع می باشیم. اما به عنوان یک توصیه برنامه نویسی همواره از پیش تعریف توابع استفاده کنید ، حتی اگر ملزم به استفاده از آن نبودید.

توابع بازگشتی

برنامه هایی که تا کنون نوشتیم یک تابع، تابع دیگری را فراخوانی می کرد. در برنامه نویسی ممکن است نیاز پیدا کنیم که تابعی خودش را به صورت مستقیم یا غیر مستقیم فراخوانی کند. به چنین توابعی، توابع بازگشتی گفته می شود . ابتدا از دید ریاضیاتی توابع بازگشتی را بررسی می کنیم. دنباله اعداد زیر را در نظر بگیرید :

2 , 5 , 11 , 23 , ...

جمله پنجم دنباله اعداد فوق چه عددی می باشد؟ حدس شما چیست؟ اگر کمی دقت کنید متوجه خواهید شد که هر جمله از دنباله فوق برابر است با دو برابر جمله قبلی بعلاوه یک. پس جمله پنجم برابر است با 2*23+1=47 دنباله فوق را توسط فرمول زیر نیز می توان مشخص کرد :

$$d_1 = 2$$
$$d_n = 2*d_{n-1}+1$$

همانطور که متوجه شده اید در این دنباله هر جمله به جملات قبلی خود وابسته است و برای بدست آوردن آن نیاز به بازگشت روی جملات قبلی داریم تا اینکه سرانجام به جمله اول که عدد ۲ می باشد برسیم. فرمول فوق را به صورت تابعی زیر بازنویسی می کنیم :

$$d(1) = 2$$
$$d(n) = 2*d(n-1)+1$$

همانطور که در تابع فوق می بینید یک حالت پایه وجود دارد که همان **d(1)=2** می باشد و یکه حالت بازگشتی که تابع با یک واحد کمتر دوباره فراخوانی می شود **d(n) = 2*d(n-1)+1** . توابع بازگشتی به طور کلی دارای یک یا چند حالت پایه و یک بخش بازگشتی می باشند. که معمولاً در بخش بازگشتی تابع با مقداری کمتر مجدداً فراخوانی می شود. تابع بازگشتی فوق به زبان **C++** به صورت زیر می باشد :

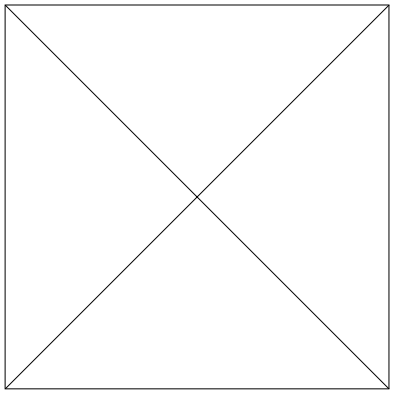
```
long int d(long int n)
{
   if (n == 1)
      return 2;
   else
      return 2*d(n-1)+1;
}
```

در زیر برنامه ای می نویسیم تا با استفاده از تابع فوق ۲۰ جمله اول دنباله مذکور را نمایش دهد.

```
#include <iostream.h>

long int d(long int);
int main( )
{
   for (int i=1;i<=20;i++)
     {
      cout<<d(i)<<"\t";
      if (i%5==0) cout<<endl;
     }

   return 0;
}

long int d(long int n)
{
   if (n == 1)
      return 2;
   else
      return 2*d(n-1)+1;
}
```

```
2       5       11      23      47
95      191     383     767     1535
3071    6143    12287   24575   49151
98303   196607  393215  786431  1572863
```

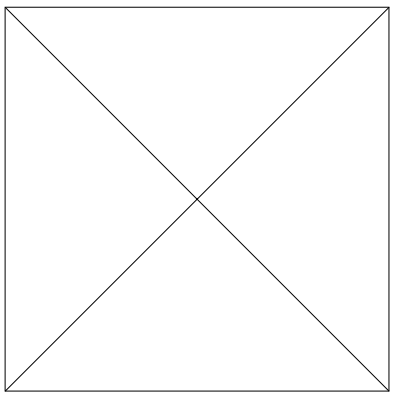
به عنوان مثال برنامه فوق جمله پنجم را به شیوه زیر محاسبه می کند :



تابع، بازگشت را تا رسیدن به حالت پایه ادامه می دهد و به محض رسیدن به حالت پایه محاسبات بازگشتی پی در پی موجب رسیدن به جواب مورد نظر می شود.

مسئله برجهای هانوی (Towers of Hanoi)

هر برنامه نویسی باید به نحوی با مسائل کلاسیک دست وپنجه نرم کرده باشد . یکی از معروفترین مسائل کلاسیک ، مسئله برجهای هانوی می باشد. طبق افسانه ای در معبدی در شرق دور، کاهنان معبدی تعدادی دیسک را از یک ستون به ستون دیگر جا به جا می کردند . ستون اول در ابتدا دارای ۶۴ دیسک با اندازه های مختلف می باشد، که بزرگترین دیسک در پایین ستون و کوچکترین دیسک در بالای ستون قرار دارد. کاهنان باید همه دیسکها را از یک ستون به ستون دوم منتقل می کردند. با این شرط که در هر بار جا به جایی تنها یک دیسک منتقل شود و نیز دیسک بزرگتری روی دیسک کوچکتر قرار نگیرد. ضمناً ستون سومی به عنوان ستون کمکی در اختیار آنها می باشد. گویند هنگامی که کاهنان معبد همه ۶۴ دیسک را با روش گفته شده از ستون اول به ستون دوم منتقل کنند جهان به پایان می رسد. شکل زیر نحوه انتقال سه دیسک را نشان می دهد.



برای راحتی کار کاهنان و برای اینکه دچار اشتباه و دوباره کاری در انتقال نشوند می خواهیم برنامه ای بنویسیم که ترتیب انتقال دیسکها را چاپ کند.

برای نوشتن این برنامه ، مسئله را باید با دید بازگشتی نگاه کنیم . انتقال **n** دیسک را به شیوه زیر انجام می دهیم :

۱- ابتدا **n-1** دیسک را از ستون اول به ستون دوم به کمک ستون سوم منتقل کن.
۲- دیسک آخر (بزرگترین دیسک) را از ستون اول به ستون سوم منتقل کن.
۳- **n-1** دیسک قرار گرفته در ستون دوم را به کمک ستون اول به ستون سوم منتقل کن.

مراحل انجام کار هنگام انتقال آخرین دیسک یعنی وقتی که **n=1** می باشد، یعنی حالت پایه به اتمام می رسد. در حالت **n=1** یک دیسک بدون کمک ستون کمکی به ستون دیگر منتقل می شود.

تابع بازگشتی مورد استفاده برای حل مسئله برجهای هانوی را با چهار آرگومان می نویسیم.

۱-	تعداد	دیسکها
۲-	ستون	مبدأ
۳-	ستون	کمکی
۴- ستون مقصد		

تابع هانوی و برنامه ای که در آن این تابع مورد استفاده قرار گرفته است به صورت زیر می باشد :

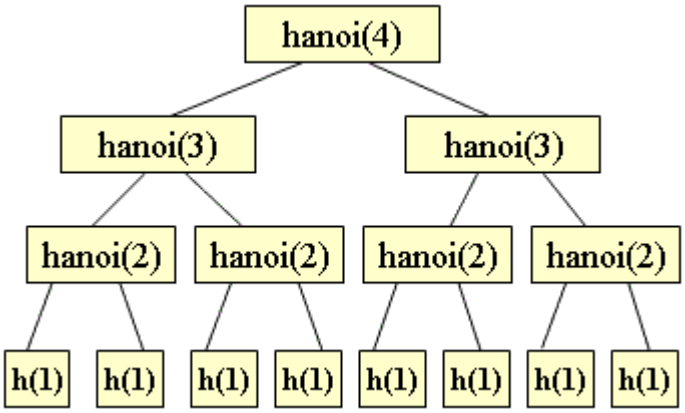
```
#include <iostream.h>

int hanoi(int, char, char, char);

int main( )
{ int disks;

  cout<<"Moving disks form tower A to C."<<endl;
  cout<<"How many disks do you want to move?";
  cin>>disks;
  cout<<hanoi(disks,'A','B','C')<<endl;

  return 0;
}

int hanoi(int n, char first, char help, char second)
{
  if (n == 1) {
    cout << "Disk " << n << " from tower " << first
         << " to tower " << second << endl;
  }
  else {
     hanoi(n-1, first, second, help);
     cout << "Disk " << n << " from tower " << first
          << " to tower " << second << endl;
     hanoi(n-1, help, first, second);
```

```
  }
  return 0;
}
```

خروجی برنامه با فرض اینکه می خواهیم مراحل انتقال چهار دیسک را ببینیم به صورت زیر می باشد :

```
Moving disks form tower A to C.
How many disks do you want to move?4
Disk 1 from tower A to tower B
Disk 2 from tower A to tower C
Disk 1 from tower B to tower C
Disk 3 from tower A to tower B
Disk 1 from tower C to tower A
Disk 2 from tower C to tower B
Disk 1 from tower A to tower B
Disk 4 from tower A to tower C
Disk 1 from tower B to tower C
Disk 2 from tower B to tower A
Disk 1 from tower C to tower A
Disk 3 from tower B to tower C
Disk 1 from tower A to tower B
Disk 2 from tower A to tower C
Disk 1 from tower B to tower C
0
```

روال فراخوانی تابع هانوی به صورت شکل زیر می باشد:



تولید اعداد تصادفی

یکی از کاربردهای اساسی کامپیوتر، استفاده از آن در کارهای شبیه سازی می باشد. در اینجا به بررسی نحوه تولید اعداد تصادفی می پردازیم. اعداد تصادفی در مسائل شبیه سازی کاربرد فراوانی دارند، به عنوان مثال شبیه سازی پرتاب سکه، پرتاب تاس و مسائلی از این قبیل.

برای تولید اعداد تصادفی زبان C++ تابعی با نام **rand()** را که در فایل کتابخانه ای **stdlib.h** قرار دارد، در اختیار ما گذاشته است. به عنوان مثال دستور زیر :

```
i = rand();
```

یک عدد تصادفی بین **۱** تا **۳۲۷۶۷** را در متغیر **i** قرار می دهد . تابع **rand()** اعداد را با احتمال مساوی در این بازه انتخاب می کند پس شانس انتخاب هر عددی در این بازه با اعداد دیگر برابر است.

معمولاً بازه اعدادی که توسط تابع **rand** تولید می شود، با آنچه که مورد نیاز ماست متفاوت می باشد. به عنوان مثال برای شبیه سازی پرتاب سکه به دو عدد تصادفی و برای تاس به شش عدد تصادفی نیاز داریم. فرض کنید که می خواهید عدد **۳۱۲۵۰** را به عددی بین **۱** تا **۶** تبدیل کنید. چه راه کاری را در نظر می گیرید؟ راهی که برای این تبدیل وجود دارد استفاده از باقیمانده صحیح می باشد، همانطور که می دانید باقیمانده صحیح تقسیم هر عددی بر **۶** یکی از اعداد **۰** تا **۵** می باشد. پس با اضافه کردن **۱** واحد به باقیمانده ، عددی بین **۱** تا **۶** خواهیم داشت. به عنوان مثال در کادر زیر عددی بین **۱** تا **۳۲۷۶۷** وارد کنید و با کلیک بر روی دکمه محاسبه خروجی ، نتیجه محاسبه را ببینید :

a= 31250

a%6+1= محاسبه خروجی

عددی تصادفی بین **۱** تا **۶** به ما می **rand()%6+1** را قرار دهیم عبارت **rand()** ، تابع **a**حال اگر به جای متغیر دهد به طور کلی برای بدست آوردن عددی تصادفی در بازه **[a,b]** می توانیم از فرمول زیر استفاده کنیم.

```
rand()%(b-a+1)+a
```

به عنوان مثال خروجی قطعه برنامه زیر عدد صحیحی در بازه **[۲۰،۱۰۰]** می باشد.

```
int a = 20 , b = 100;
cout<< rand()%(b-a+1)+a;
```

برنامه زیر **۲۰** عدد تصادفی بین **۱** تا **۶** را ایجاد می کند. که این برنامه را می توان **۲۰** بار پرتاب یک تاس در نظر گرفت :

```
#include <iostream.h>
```

```
#include <stdlib.h>

int main()
{
   for (int i = 1; i<= 20; i++ )
    {
     cout << rand() % 6 + 1<<"\t";

     if ( i % 5 == 0 )
        cout << endl;
    }
   return 0;
}
```

خروجی برنامه فوق به صورت زیر می باشد :

```
5       5       3       5       5
2       4       2       5       5
5       3       2       2       1
5       1       4       6       4
```

یک بار دیگر برنامه فوق را اجرا کنید و خروجی را مجدداً بررسی کنید. خواهید دید خروجی دقیقاً همان اعداد قبلی می باشد. خروجی تابع **rand()** اعداد تصادفی می باشد ولی با اجرای دوباره برنامه همان اعداد مجدداً به همان ترتیب قبلی تکرار می شوند. این تکرار یکی از قابلیتهای تابع می باشد ودر اشکال زدایی برنامه کاربرد دارد.

اگر بخواهیم که تابع **rand()** اعداد کاملاً تصادفی ایجاد کند باید از تابع **srand()** استفاده کنیم. این تابع ورودی از نوع اعداد صحیح بدون علامت می گیرد و باعث تصادفی شدن تابع **rand()** بر اساس مقدار ورودی تابع **srand()** می شود. تابع**srand()** نیز در فایل کتابخانه ای **stdlib.h** قرار دارد. در برنامه زیر به نحوه استفاده از تابع **srand()** پی خواهید برد.

```
#include <iostream.h>
#include <stdlib.h>

int main()
{
  unsigned int num;

  cout<<"Enter a number: ";
  cin>>num;

  srand(num);
```

```
  for (int i = 1; i<= 20; i++ )
   {
    cout << rand() % 6 + 1<<"\t";

    if ( i % 5 == 0 )
      cout << endl;
   }
  return 0;
}
```

خروجی برنامه به صورت زیر می باشد.

```
Enter a number: 251
3       4       1       4       6
6       4       6       2       5
5       3       1       4       5
1       6       6       6       1
Enter a number: 350
1       4       3       4       1
2       6       2       6       2
4       2       5       3       5
4       4       5       2       3
Enter a number: 251
3       4       1       4       6
6       4       6       2       5
5       3       1       4       5
1       6       6       6       1
```

همانطور که می بینید بر اساس ورودی های مختلف خروجی نیز تغییر می کند. توجه داشته باشید که ورودی های یکسان خروجی های یکسانی دارند.

اگر بخواهیم بدون نیاز به وارد کردن عددی توسط کاربر، اعداد تصادفی داشته باشیم می توانیم از تابع **time** که در فایل کتابخانه ای **time.h** قرار دارد استفاده کنیم . تابع **time** ساعت کامپیوتر را می خواند و زمان را بر حسب ثانیه بر می گرداند ، به این ترتیب دستور زیر:

```
srand(time(0));
```

باعث می شود که تابع **rand()** در هر بار اجرای برنامه اعداد متفاوتی را ایجاد کند. اگر برنامه فوق را به صورت زیر باز نویسی کنیم با هر بار اجرای برنامه اعداد تصادفی متفاوتی خواهیم داشت.

```
#include <iostream.h>
```

```
#include <stdlib.h>
#include <time.h>

int main()
{
  srand(time(0));

  for (int i = 1; i<= 20; i++ )
   {
    cout << rand() % 6 + 1<<"\t";

    if ( i % 5 == 0 )
      cout << endl;
   }
  return 0;
}
```

مثال : برنامه ای بنویسید که پرتاب سکه ای را شبیه سازی کند ، بدین صورت که سکه را **۲۰۰۰** بار پرتاب کند و دفعات رو یا پشت آمدن سکه را چاپ کند.

```
#include <iostream.h>
#include <stdlib.h>
#include <time.h>

int main()
{
  int back=0,front=0,face;

  srand(time(0));

  for (int i = 1; i<= 2000; i++ )
    {
      face=rand()%2+1;
      switch(face)
        {
          case 1:
            ++back;
            break;
          case 2:
            ++front;
            break;
          default:
            cout<<"Error!";
        }
    }
```

```
  cout<<"Front: "<< front<<endl;
  cout<<"Back : "<< back<<endl;

  return 0;
}
```

نوع داده ای enum

enum یک نوع داده ای تعریف شده توسط برنامه نویس را که به آن نوع داده شمارش می گویند، ایجاد می کند. نحوه ایجاد یک نوع داده شمارش به صورت زیر می باشد.

```
enum  نام نوع داده{ثابت ۱ و ثابت ۲ و ... و ثابتn}
```

این دستور به ترتیب در ثابت ۱ ، ثابت ۲ و ... و ثابتn اعداد صحیح متوالی ۰ تا **n** را قرار می دهد . به صورت پیش فرض مقداردهی متغیرها در این دستور از صفر شروع می شود.

```
enum TrueFalse {FALSE , TRUE}
```

دستور فوق به ثابت **FALSE** ، عدد **صفر** و به ثابت **TRUE** عدد **۱** را تخصیص می دهد. حال اگر بخواهیم مقداردهی از عددی غیر از صفر شروع شود باید عدد مورد نظر را مشخص کنیم :

```
enum Days {SAT = 1, SUN, MON, TUE, WED, THU, FRI}
```

دستور فوق به روزهای هفته به ترتیب اعداد **۱** تا **۷** را نسبت می دهد. توصیه می شود که نام ثابت های شمارشی را با حروف بزرگ بنویسید ، بدین صورت این ثابتها با متغیرهای برنامه ، اشتباه نمی شوند. ضمناً **enum** را در ابتدای برنامه به کار ببرید.

در حقیقت این نوع داده به هر یک از موارد لیستی از اعداد نامی را نسبت می دهد. به عنوان نمونه در مثال روزهای هفته هر یک از اعداد **۱** تا **۷** را با یکی از روزهای هفته نام گذاری کردیم.

مقدار دهی موارد لیست به صورت های مختلف امکان پذیر می باشد.

```
enum Days { MON, TUE, WED, THU, FRI, SAT , SUN = 0}
```

دستور فوق **SUN** را با عدد **صفر** و **SAT** را با عدد **۱**- و ... و **MON** را با عدد **6-** مقدار دهی می کند.

```
enum Colors {BLACK = 2, GREEN = 4, RED = 3,
             BLUE = 5, GRAY,WHITE = 0}
```

در دستور فوق هر یک از موارد با عدد نسبت داده شده مقدار دهی می شوند و **GRAY** با عدد **۶** مقدار دهی می شود چون بعد از **BLUE = 5** آمده است.

به محض ساخته شدن لیست ، نوع داده نوشته شده توسط برنامه نویس قابل استفاده می گردد و می توان متغیرهایی را از نوع داده نوشته شده توسط برنامه نویس به همان شیوه ای که سایر متغیرها را تعریف می کردیم، تعریف کرد. به عنوان مثال :

```
TrueFalse  tf1,tf2;
Days       day1, day2 = SUN;
Colors     color1 = BLACK , color2 = GRAY;
```

همچنین متغیرها را می توان هنگام ایجاد نوع داده، تعریف کرد. به عنوان مثال :

```
TrueFalse {FALSE, TRUE} tf1 ,tf2;
```

نکته : تبدیل داده ای از نوع **enum** به عدد صحیح مجاز می باشد ولی بر عکس این عمل غیر مجاز است. به عنوان مثال :

```
enum MyEnum {ALPHA, BETA, GAMMA};
int   i = BETA;
int   j = 3+GAMMA;
```

دستورات فوق مجاز می باشند، و این دستورات عدد **۱** را در **i** و **۵** را در **j** قرار می دهند.

```
enum MyEnum {ALPHA, BETA, GAMMA};
MyEnum  x = 2;
MyEnum  y = 123;
```

ولی دستورات فوق غیر مجاز می باشند. البته بعضی از کامپایلرها این موضوع را نادیده می گیرند و تنها یک پیغام اخطار می دهند ولی توصیه می شود که برای پیشگیری از وقوع خطاهای منطقی در برنامه از به کار بردن دستوراتی مانند کدهای فوق خودداری کنید.

برنامه زیر نحوه کاربرد نوع داده **enum** را نشان می دهد.

```
#include <iostream.h>
int main()
{
  enum PizzaSize{SMALL,MEDIUM,LARGE,EXTRALARGE};
  PizzaSize size;
  size=LARGE;

  cout<<"The small pizza has a value of "<<SMALL;
  cout<<"\nThe medium pizza has a value of "<<MEDIUM;
  cout<<"\nThe large pizza has a value of "<<size;

  return 0;
}
```

خروجی برنامه به صورت زیر می باشد:

```
The small pizza has a value of 0
The medium pizza has a value of 1
The large pizza has a value of 2
```

توابع بدون خروجی و آرگومان

در برنامه نویسی به توابعی نیاز پیدا می کنیم که نیاز ندارند چیزی را به عنوان خروجی تابع برگردانند و یا توابعی که نیاز به آرگومان و ورودی ندارند ویا هر دو. زبان C++ برای امکان استفاده از چنین توابعی، کلمه **void** را در اختیار ما قرار داده است. اگر بخواهیم تابعی بدون خروجی ایجاد کنیم کافی است به جای نوع داده خروجی تابع کلمه **void** را قرار دهیم. به تابع زیر توجه کنید.

```
void function (int num)
{
   cout << "My input is" << num << endl;
}
```

همانطور که می بینید این تابع نیازی به استفاده از دستور **return** ندارد چون قرار نیست چیزی را به عنوان خروجی تابع برگرداند. تابع فوق بر اساس مقدار داده ورودی، پیغامی را بر روی صفحه نمایش چاپ می کند.

حال که با **void** آشنا شدید می توانیم از این به بعد تابع **main** را از نوع **void** تعریف کنیم. در این صورت دیگر نیازی به استفاده از دستور **return 0;** در انتهای برنامه نداریم :

```
void main()
{
    دستورات برنامه
}
```

به عنوان مثال برنامه برج هانوی را که در مبحث توابع بازگشتی نوشتیم با استفاده از نوع **void** بازنویسی می کنیم.

```
#include <iostream.h>

void hanoi(int, char, char, char);

void main( )
{
   cout<<"Moving 4 disks form tower A to C."<<endl;
   hanoi(4,'A','B','C');
}

void hanoi(int n, char first, char help, char second) {
   if (n == 1) {
       cout << "Disk " << n << " from tower " << first
            << " to tower " << second << endl;
   } else {
       hanoi(n-1, first, second, help);
       cout << "Disk " << n << " from tower " << first
            << " to tower " << second << endl;
       hanoi(n-1, help, first, second);
   }
}
```

همانطور که در برنامه فوق می بینید تابع **hanoi** بدون دستور **return** نوشته شده است، زیرا نوع تابع **void** می باشد، یعنی تابع بدون خروجی است و توابع بدون خروجی را می توانیم مستقیماً همانند برنامه فوق فراخوانی کنید. یعنی کافی است نام تابع را همراه آرگومانهای مورد نظر بنویسیم.

برای ایجاد توابع بدون آرگومان می توانید در پرانتز تابع کلمه **void** را بنویسید یا اینکه این پرانتز را خالی گذاشته و در آن چیزی ننویسید.

```
void function1();
int  function2(void);
```

در دو دستور فوق تابع **function1** از نوع توابع بدون خروجی و بدون آرگومان ایجاد می شود. و تابع **function2** از نوع توابع بدون آرگومان و با خروجی از نوع عدد صحیح می باشد، برای آشنایی با نحوه کاربرد توابع بدون آرگومان به برنامه زیر توجه کنید :

```
#include <iostream.h>

int function(void);

void main( )
{
   int num, counter = 0;
   float average, sum = 0;

   num=function();

   while (num != -1){
      sum += num ;
      ++counter;
      num=function();
   }

   if (counter != 0){
      average = sum / counter;
      cout << "The average is " << average << endl;
   }
   else
      cout << "No numbers were entered." << endl;
}

int function(void){
  int x;
  cout << "Enter a number (-1 to end):";
  cin >>x;
  return x;
}
```

همانطور که در برنامه فوق مشاهده می کنید تابع **function** از نوع توابع بدون آرگومان می باشد و دارای خروجی صحیح می باشد. این تابع در برنامه دو بار فراخوانی شده است و وظیفه این تابع دریافت متغیری از صفحه کلید و برگرداندن آن متغیر به عنوان خروجی تابع می باشدو دستور **num=function()** عدد دریافت شده از صفحه کلید را در متغیر **num** قرار می دهد اگر به یاد داشته باشید این برنامه قبلاً بدون استفاده از تابع در مبحث ساختار تکرار while نوشته بودیم. برای درک بهتر این برنامه توصیه می شود آن را با برنامه موجود در مبحث ساختار تکرار while مقایسه کنید، و متوجه خواهید شد که تابع **function** ما را از دوباره نویسی بعضی از دستورات بی نیاز کرده است و نیز برنامه خلاصه تر و مفهوم تر شده است.

قوانین حوزه

قسمتی از برنامه که در آن متغیری تعریف شده و قابل استفاده می باشد، حوزه آن متغیر گفته می شود. در زبان C++ به قسمتی از برنامه که با یک علامت ({) شروع شده و با علامت (}) به پایان می رسد یک بلوک می گویند. به عنوان مثال هنگامی که متغیری را در یک بلوک تعریف می کنیم، متغیر فقط در آن بلوک قابل دسترسی می باشد ولذا حوزه آن متغیر بلوکی که در آن تعریف شده است ، می باشد. به مثال زیر توجه کنید :

```
#include <iostream.h>

void main( )
{
   {
     int x= 1;
     cout << x;
   }
   cout << x;
}
```

اگر برنامه فوق را بنویسیم و بخواهیم اجرا کنیم پیغام خطای **Undefined symbol 'x'** را دریافت خواهیم کرد ودلیل این امر این است که متغیر **x** فقط در بلوک درونی تابع **main** تعریف شده است، لذا در خود تابع قابل دسترسی نمی باشد. در این مبحث به بررسی حوزه تابع، حوزه فایل و حوزه بلوک می پردازیم.

متغیری که خارج از همه توابع تعریف می شود، دارای حوزه فایل می باشد و چنین متغیری برای تمام توابع، شناخته شده وقابل استفاده می باشد. به مثال زیر توجه کنید:

```
#include <iostream.h>

int x=1;
int f();

void main( )
{
   cout << x;
   cout << f();
   cout << x;
}
int f(){
  return 2*x;
}
```

متغیر **x** دارای حوزه فایل می باشد. لذا در تابع **main** و تابع **f** قابل استفاده می باشد. خروجی برنامه فوق به صورت زیر می باشد.

```
121
```

متغیری که درون توابع و یا به عنوان آرگومان تابع تعریف می گردد، دارای حوزه تابع می باشد و از نوع متغیرهای محلی است و خارج ازتابع قابل استفاده و دسترسی نمی باشد. توابعی که تا کنون نوشتیم ومتغیرهایی که در آن ها تعریف کردیم، همگی دارای حوزه تابع بودند. ضمنا این متغیرها، هنگامی که برنامه از آن تابع خارج می شود، مقادیر خود را از دست می دهند. حال اگر بخواهیم یک متغیر محلی تابع، مقدار خود را حفظ کرده و برای دفعات بعدی فراخوانی تابع نیز نگه دارد، زبان C++ کلمه **static** را در اختیار ما قرار داده است. کلمه **static** را باید قبل از نوع متغیر قرار دهیم. مانند:

```
static int x=1;
```

دستور فوق متغیر **x** را از نوع عدد صحیح تعریف می کند و این متغیر با اولین فراخوانی تابع مقدار دهی می شود و در دفعات بعدی فراخوانی تابع مقدار قبلی خود را حفظ می کند. به مثال زیر توجه کنید:

```
#include <iostream.h>

int f();

void main( )
{
   cout << f();
   cout << f();
   cout << f();
}
int f(){
  static int x=0;
  x++;
  return x;
}
```

خروجی برنامه فوق به صورت زیر می باشد:

```
123
```

برنامه با اولین فراخوانی تابع f به متغیر محلی **x** مقدار ۰ را می دهد، سپس به **x** یک واحد اضافه می شود و به عنوان خروجی برگردانده می شود. پس ابتدا عدد۱ چاپ می گردد. در بار دوم فراخوانی تابع **f** ، متغیر **x** دوباره مقداردهی نمی شود، بلکه به مقدار قبلی آن که عدد **۱** است، یک واحد اضافه گشته و به عنوان خروجی برگردانده می شود. پس این بار عدد **۲**

چاپ می گردد و در نهایت با فراخوانی تابع **f** برای بار سوم عدد **۳** چاپ خواهد شد. اگر برنامه فوق را بدون کلمه **static** بنویسیم، خروجی **۱۱۱** خواهد بود.

یکی از نکاتی که می توان در قوانین حوزه بررسی کرد، متغیرهای همنام در بلوک های تو در تو می باشد. به مثال زیر توجه کنید:

```
#include <iostream.h>

void main( )
{
   int x=1;
   cout << x;
   {
     int x= 2;
     cout << x;
   }
   cout << x;
}
```

در برنامه فوق متغیر **x** یک بار در تابع **main** تعریف شده است و با دیگر در بلوک درونی. خروجی برنامه به صورت زیر می باشد:

```
121
```

هنگامی که در بلوک درونی متغیری با نام **x** را مجددا تعریف می کنیم، متغیر خارج از بلوک از دید برنامه پنهان می گردد و تنها متغیر داخل بلوک قابل استفاده می شود. همچنین هنگامی که برنامه از بلوک خارج می گردد، متغیر **x** بیرونی دوباره قابل استفاده می گردد. ضمنا توجه داشته باشید که مقدار متغیر x تابع **main** تغییری نکرده است، یعنی با وجود استفاده از متغیری همنام و نیز مقداردهی آن، تاثیری روی متغیر **x** تابع ایجاد نشده است. چون حوزه متغیر **x** بلوک درونی تنها داخل آن بلوک می باشد.

برنامه زیر تمام موارد ذکر شده در این مبحث را شامل می شود. بررسی آن و خروجی برنامه شما را در فهم بهتر این مبحث یاری می نماید.

```
#include <iostream.h>

void useLocal( void ); // function prototype
void useStaticLocal( void ); // function prototype
void useGlobal( void ); // function prototype

int x = 1; // global variable
```

```
void main()
{
   int x = 5; // local variable to main

   cout <<"local x in main's outer scope is "<<x<<endl;

   { // start new scope

     int x = 7;

     cout <<"local x in main's inner scope is "<<x<<endl;

   } // end new scope

   cout <<"local x in main's outer scope is "<<x<< endl;

 useLocal(); //useLocal has local x
 useStaticLocal(); //useStaticLocal has static local x
 useGlobal(); //useGlobal uses global x
 useLocal(); //useLocal reinitializes its local x
 useStaticLocal();//static local x retains its prior value
 useGlobal(); //global x also retains its value

   cout << "\nlocal x in main is " << x << endl;

} // end main

//useLocal reinitializes local variable x during each call
void useLocal( void )
{
   int x = 25; //initialized each time useLocal is called

   cout << endl << "local x is " << x
        << " on entering useLocal" << endl;
   ++x;
   cout << "local x is " << x
        << " on exiting useLocal" << endl;

} // end function useLocal

// useStaticLocal initializes static local variable x
// only the first time the function is called; value
// of x is saved between calls to this function
void useStaticLocal( void )
{
// initialized first time useStaticLocal is called.
```

```
   static int x = 50;

   cout << endl << "local static x is " << x
        << " on entering useStaticLocal" << endl;
   ++x;
   cout << "local static x is " << x
        << " on exiting useStaticLocal" << endl;

} // end function useStaticLocal

// useGlobal modifies global variable x during each call
void useGlobal( void )
{
   cout << endl << "global x is " << x
        << " on entering useGlobal" << endl;
   x *= 10;
   cout << "global x is " << x
        << " on exiting useGlobal" << endl;

} // end function useGlobal
```

خروجی برنامه به صورت زیر می باشد:

```
local x in main's outer scope is 5
local x in main's inner scope is 7
local x in main's outer scope is 5

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 50 on entering useStaticLocal
local static x is 51 on exiting useStaticLocal

global x is 1 on entering useGlobal
global x is 10 on exiting useGlobal

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 51 on entering useStaticLocal
local static x is 52 on exiting useStaticLocal

global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal
```

```
local x in main is 5
```

در برنامه فوق سه تابع با نام های **useStaticLocal**, **useGlobal**, **useLocal** داریم. متغیر x تعریف شده در ابتدای برنامه با مقدار **۱** به عنوان یک متغیر عمومی می باشد و دارای حوزه فایل است. در تابع **main**، متغیر **x** با مقدار **۵** تعریف شده است. لذا متغیر عمومی **x** با مقدار **۱** نادیده گرفته می شود و هنگام اجرای دستور **cout** ، متغیر **x** با مقدار **۵** در خروجی چاپ می شود. در بلوک درونی، متغیر **x** با مقدار **۷** تعریف شده است. لذا **x** عمومی و **x** محلی تابع **main** نادیده گرفته می شوند و تنها **x** با مقدار **۷** توسط دستور **cout** چاپ می گردد. پس از آنکه بلوک حوزه **x** با مقدار **۷** به اتمام می رسد، دوباره **x** محلی تابع **main** با مقدار **۵** نمایان می گردد.

تابع **useLocal** متغیر محلی **x** را با مقدار **۲۵** در خود تعریف می کند. هنگامی که این تابع در برنامه فراخوانی می شود، تابع ، متغیر **x** را چاپ می کند، سپس یک واحد به آن اضافه کرده و دوباره **x** را چاپ می کند. هر بار که این تابع فراخوانی می شود، متغیر **x** با مقدار **۲۵** در آن تعریف می شود و هنگام خروج از تابع از بین می رود.

تابع **useStaticLocal** متغیرمحلی **x** را از نوع **static** تعریف کرده و با عدد **۵۰** مقداردهی می کند و سپس آن را چاپ کرده و یک واحد به آن اضافه می کند و دوباره چاپش می کند. اما هنگام خروج از تابع مقدار **x** از بین نمی رود. و با فراخوانی مجدد تابع ، مقدار قبلی متغیر **x** محلی ، برای این تابع موجود می باشد و دوباره از نو مقداردهی نمی شود. در اینجا هنگامی که تابع دوباره فراخوانی می شود، **x** حاوی **۵۱** خواهد بود.

تابع **useGlobal** هیچ متغیری را در خود تعریف نمی کند. لذا هنگامی که به متغیر **x** مراجعه می کند ، متغیر**x** عمومی مورد استفاده قرار می گیرد. هنگامی که این تابع فراخوانی می شود مقدار متغیر **x** عمومی چاپ می شود. سپس در **۱۰** ضرب شده و دوباره چاپ می گردد. هنگامی که برای بار دوم تابع **useGlobal** فراخوانی می شود **x** عمومی حاوی عدد **۱۰۰** می باشد.

پس از اینکه برنامه هر یک از توابع فوق را دوبار فراخوانی کرد ، مجددا متغیر **x** تابع **main** با مقدار **۵** چاپ می گردد و این نشان می دهد که هیچ یک از توابع ، تاثیری روی متغیر محلی تابع **main** نداشتند.

آرگومانهای پیش فرض توابع

در برنامه نویسی ممکن است تابعی را به دفعات با آرگومانهای یکسانی صدا بزنیم . در چنین حالتی ، برنامه نویس می تواند برای آرگومانهای تابع ، مقداری را به عنوان پیش فرض قرار دهد . هنگامی که در فراخوانی توابع ، آرگومان دارای مقدار پیش فرض حذف شده باشد ، کامپایلر مقدار پیش فرض آن آرگومان را به تابع خواهد فرستاد .

آرگومان های پیش فرض باید سمت راستی ترین آرگومان های تابع باشند . هنگامی که تابعی با بیش از یک آرگومان فراخوانی می شود ، اگر آرگومان حذف شده سمت راستی ترین آرگومان نباشد ، آنگاه همه آرگومانهای سمت راست آن

آرگومان نیز باید حذف شوند . آرگومان های پیش فرض باید در اولین جایی که نام تابع آورده می شود (که معمولاً در پیش تعریف تابع است) مشخص شوند .

مقادیر پیش فرض می توانند اعداد ، مقادیر ثابت ، متغیرهای عمومی و یا خروجی تابع دیگر باشند .

برنامه زیر نحوه مقدار دهی به آرگومان های پیش فرض و نیز نحوه فراخوانی تابع با مقدار پیش فرض را نشان می دهد . در این برنامه حجم جعبه ای محاسبه می شود .

```
#include <iostream.h>

// function prototype that specifies default arguments
int boxVolume(int length=1, int width=1, int height=1);

int main()
{
  //no arguments--use default values for all dimensions
  cout <<"The default box volume is: "<<boxVolume();

  //specify length; default width and height
  cout <<"\n\nThe volume of a box with length 10,\n"
       <<"width 1 and height 1 is: "<<boxVolume(10);

  //specify length and width; default height
  cout <<"\n\nThe volume of a box with length 10,\n"
       <<"width 5 and height 1 is: "<<boxVolume(10,5);

  //specify all arguments
  cout <<"\n\nThe volume of a box with length 10,\n"
       <<"width 5 and height 2 is: "<<boxVolume(10,5,2)
       <<endl;

  return 0; // indicates successful termination

} //end main

// function boxVolume calculates the volume of a box
int boxVolume( int length, int width, int height )
{
  return length * width * height;
} // end function boxVolume
```

خروجی برنامه فوق به صورت زیر می باشد .

```
The default box volume is: 1
```

```
The volume of a box with length 10,
width 1 and height 1 is: 10

The volume of a box with length 10,
width 5 and height 1 is: 50

The volume of a box with length 10,
width 5 and height 2 is: 100
```

در پیش تعریف تابع **boxVolume** به هر یک از سه آرگومان تابع مقدار پیش فرض ۱ داده شده است . توجه داشته باشید که مقادیر پیش فرض باید در پیش تعریف تابع نوشته شوند ، ضمناً نوشتن نام آرگومان های تابع در پیش تعریف الزامی نیست و در برنامه فوق اینکار تنها برای خوانایی بیشتر انجام گرفته است ، البته توصیه می شود که شما نیز از این شیوه استفاده کنید . به عنوان مثال پیش فرض تابع **boxVolume** در برنامه فوق را می توانستیم به صورت زیر نیز بنویسیم :

```
int boxVolume (int = 1 , int = 1 , int = 1 );
```

در اولین فراخوانی تابع **boxVolume** در برنامه فوق هیچ آرگومانی به آن داده نشده است لذا هر سه مقدار پیش فرض آرگومان ها مورد استفاده قرار می گیرد و حجم جعبه عدد ۱ می شود . در دومین فراخوانی آرگومان **length** ارسال می گردد ، لذا مقادیر پیش فرض آرگومان های **width** و **height** استفاده می شوند . در سومین فراخوانی آرگومان های **width** و **length** ارسال می گردند لذا مقادیر پیش فرض آرگومان **height** مورد استفاده قرار می گیرد . در آخرین فراخوانی هر سه آرگومان ارسال می شوند لذا از هیچ مقدار پیش فرضی استفاده نمی شود .

پس هنگامی که یک آرگومان به تابع فرستاده می شود ، آن آرگومان به عنوان **length** در نظر گرفته می شود و هنگامی که دو آرگومان به تابع **boxVolume** فرستاده می شود تابع آنها را به ترتیب از سمت چپ به عنوان آرگومان **length** و سپس **width** در نظر می گیرد و سرانجام هنگامی که هرسه آرگومان فرستاده می شود به ترتیب از سمت چپ در **length** و **width** و **height** قرار می گیرند .

عملگر یگانی تفکیک حوزه

همانطور که در مبحث قوانین حوزه دیدید تعریف متغیرهای محلی و عمومی با یک نام در برنامه امکان پذیر می باشد . زبان **C++** عملگر یگانی تفکیک دامنه (::) را برای امکان دستیابی به متغیر عمومی همنام با متغیر محلی ، در اختیار ما قرار داده است . توجه داشته باشید که این عملگر تنها قادر به دستیابی به متغیر عمومی در حوزه فایل می باشد . ضمناً متغیر عمومی بدون نیاز به این عملگر نیز قابل دستیابی می باشد ؛ به شرط آنکه متغیرمحلی همنام با متغیر عمومی ، در برنامه به کار برده نشود . استفاده از عملگر (::) همراه نام متغیر عمومی ، در صورتی که نام متغیر عمومی برای متغیر دیگری به کار برده نشده باشد ، اختیاری است . اما توصیه می شود که برای اینکه بدانید از متغیر عمومی استفاده می کنید از این عملگر همواره در کنار نام متغیر عمومی استفاده کنید . برنامه زیر نحوه کاربرد عملگر (::) را نشان می دهد .

```
#include <iostream.h>

float pi=3.14159;

void main( )
{
     int pi=::pi;
     cout << "Local pi is : " << pi << endl;
     cout << "Global pi is : " << ::pi << endl;
}
```

خروجی برنامه به صورت زیر می باشد :

```
Local pi is : 3
Global pi is : 3.14159
```

در برنامه فوق متغیر عمومی **pi** از نوع **float** تعریف شده است و در تابع متغیر محلی **pi** از نوع **int** با مقدار اولیه **pi** عمومی مقدار دهی می شود . توجه داشته باشید که برای دستیابی به مقدار **pi** عمومی از عملگر یگانی تفکیک حوزه (::) استفاده شد . پس از مقدار دهی به **pi** محلی ، توسط دستور **cout** ، متغیر **pi** محلی که حاوی عدد **۳** است چاپ می گردد و در خط بعدی متغیر **pi** عمومی که حاوی **۳,۱۴۱۵۹** می باشد چاپ خواهد شد .

ارسال آرگومان ها به تابع ، با ارجاع

تا به حال ، در تمام توابعی که نوشتیم آرگومان ها با مقدار به توابع فرستاده می شدند . این بدان معناست که هنگامی که توابع با آرگومانها فرا خوانی می شدند ، چیزی که ما به عنوان ورودی تابع ارسال می کردیم مقدار یا عدد بود و هرگز خود متغیر به تابع فرستاده نشد ، به عنوان مثال تابع **maximum** در مبحث تعریف توابع را به صورت زیر فراخوانی می کنیم :

```
int a=5, b=6, c=7, max;
max = maximum(a,b,c);
```

کاری که در اینجا صورت می گیرد فراخوانی تابع و فرستادن مقادیر موجود در **a** و**b** و**c** یعنی ۵ و ۶ و ۷ به تابع می باشد . و خود متغیرها فرستاده نمی شوند .

```
int maximum (int x   , int y   , int z)
              5↑       6↑       7↑
max = maximum (  a   ,    b   ,    c )
```

بدین صورت هنگامی که تابع **maximum** فراخوانی می شود ، مقدار متغیرهای **x** و**y** و**z** به ترتیب برابر ۵ و ۶ و ۷خواهند شد و هرگونه تغییری روی متغیرهای **x** و**y** و**z** در تابع ، تأثیری روی متغیرهای **a** و**b** و**c** نخواهد داشت . زیرا خود متغیرهای **a** و**b** و**c** به تابع فرستاده نشده اند بلکه مقادیر موجود در آنها به تابع ارسال گشته اند .

در برنامه نویسی مواردی پیش می آید که بخواهید از داخل تابع ، مقادیر متغیرهای خارجی را تغییر دهیم ، به عنوان مثال در تابع **maximum** مقدار متغیر **a** را از داخل تابع تغییر دهیم . برای نیل به این هدف باید از روش ارسال آرگومان ها با ارجاع استفاده کنیم . برای آنکه آرگومان تابعی با ارجاع فرستاده شود ، کافی است در پیش تعریف تابع بعد از تعیین نوع آرگومان یک علامت (**&**) بگذاریم و نیز در تعریف تابع قبل از نام آرگومان یک علامت (**&**) قرار دهیم . برای آشنایی با نحوه ارسال آرگومان ها با ارجاع به برنامه زیر توجه کنید .

```
#include <iostream.h>

void duplicate (int & , int & );

void main ( )
{
    int a=1 , b=2 ;
    cout << "a = " << a << " and b = " << b << endl;
    duplicate (a,b);
    cout << "a = " << a << " and b = " << b << endl;
}

void duplicate (int &x , int &y)
{
    x*=2;
    y*=2;
}
```

خروجی برنامه به صورت زیر می باشد .

```
a = 1 and b = 2
a = 2 and b = 4
```

در برنامه فوق متغیرهای **a** و**b** به تابع ارسال می گردند و سپس در دو ضرب می شوند. در این برنامه مقدار متغیرهای **a** و**b** فرستاده نمی شود بلکه خود متغیر فرستاده می شود و لذا هنگامی که دستورهای

```
x*=2;
y*=2;
```

اجرا می گردند مقادیر دو متغیر**a** و**b** دو برابر می شود . در حقیقت **x** و**y** مانند نام مستعاری برای **a** و**b** می باشند .

```
void duplicate (int &x , int &y)
                a↕      b↕
     duplicate (    a ,     b)
```

هنگامی که متغیری با ارجاع فرستاده می شود هر گونه تغییری که در متغیر معادل آن در تابع صورت گیرد عیناً آن تغییر بر روی متغیر ارسالی نیز اعمال می گردد .

مثال : تابعی بنویسید که دو متغیر را به عنوان ورودی دریافت کرده و مقادیر آنها را جابه جا کند . از این تابع در برنامه ای استفاده کنید .

```
#include <iostream.h>
void change (int & , int &);
int main ( )
 {
      int a=1 , b=2 ;
      cout << "a is " << a << " and b is " << b << endl;
      change (a,b);
      cout << "a is " << a << " and b is " << b << endl;
      return 0;
  }
void change (int &x , int &y)
 {
      int temp;
      temp = y;
      y = x;
      x = temp;
 }
```

خروجی برنامه به صورت زیر است :

```
a is 1 and b is 2
a is 2 and b is 1
```

برنامه فوق مقادیر دو متغیر **a** و**b** را توسط **change** با شیوه ارسال آرگومان با ارجاع جابه جا می کند .

یکی دیگر از کاربردهای ارسال با ارجاع ، دریافت بیش از یک خروجی از تابع می باشد ، به عنوان مثال تابع **prevnext** در برنامه زیر مقادیر صحیح قبل و بعد از اولین آرگومان را به عنوان خروجی بر می گرداند .

```
#include <iostream.h>
```

```
void prevnext (int ,int & , int &);

void main ( )
{
    int x = 100 , y , z ;
    cout << "The input of prevnext function is "
         << x << endl;
    prevnext (x,y,z) ;
    cout << "previous =" << y <<",Next =" << z;
}

void prevnext (int input , int & prev , int & next)
{
    prev = input - 1 ;
    next = input + 1 ;
}
```

خروجی برنامه فوق به صورت زیر می باشد .

```
The input of prevnext function is 100
previous =99,Next =101
```

همانطور که مشاهده می کنید آرگومان **input** مقدار داده موجود در متغیر **x** را دریافت می کند ولی آرگومان های **prev** و**next** خود متغیرهای **y** و**z** را دریافت می کنند . لذا تغییرات روی متغیر**prev** و**next** بر روی **y** و**z** انجام می گیرد و توسط تابع مقدار دهی می شوند .

گرانبار کردن توابع (استفاده از یک نام برای چند تابع)

C++ استفاده از یک نام را برای چند تابع ، هنگامی که توابع از نظر نوع آرگومان ها ، تعداد آرگومان ها یا ترتیب قرار گرفتن نوع آرگومان ها با هم متفاوت باشند را امکان پذیر کرده است این قابلیت ، گرانبار کردن توابع نامیده می شود . هنگامی که یک تابع گرانبار شده فراخوانی می شود کامپایلر با مقایسه نوع ، تعداد و ترتیب آرگومان ها تابع درست را انتخاب می کند . معمولاً از توابع گرانبار شده برای ایجاد چند تابع با نامهای یکسان که کار یکسانی را بر روی انواع داده ای متفاوتی انجام می دهند استفاده می شود . به عنوان مثال اکثر توابع ریاضی زبان **C**++ برای انواع داده ای متفاوت گرانبار شده اند . گرانبار کردن توابعی که کار یکسانی را انجام می دهند برنامه را قابل فهم تر و خواناتر می سازد . برنامه زیر نحوه به کار گیری توابع گرانبار شده را نشان می دهد .

```
#include <iostream.h>

int square( int );
double square( double );
```

```
void main()
{
  // calls int version
  int intResult = square( 7 );
  // calls double version
  double doubleResult = square( 7.5 );

  cout << "\nThe square of integer 7 is "
       << intResult
       << "\nThe square of double 7.5 is "
       << doubleResult
       << endl;

} // end main

// function square for int values
int square( int x )
{
  cout <<"Called square with int argument: "
       << x << endl;
  return x * x;
} // end int version of function square

// function square for double values
double square( double y )
{
  cout <<"Called square with double argument: "
       << y << endl;
  return y * y;
} // end double version of function square
```

خروجی برنامه به صورت زیر می باشد .

```
Called square with int argument: 7
Called square with double argument: 7.5

The square of integer 7 is 49
The square of double 7.5 is 56.25
```

برنامه فوق برای محاسبه مربع یک عدد صحیح (**int**) و یک عدد اعشاری (**double**) از تابع گرانبارشده **square** استفاده می کند .هنگامی که دستور:

```
int intResult = square (7) ;
```

اجرا می گردد تابع **square** با پیش تعریف :

```
int square (int) ;
```

فراخوانی می شود و هنگامی که دستور :

```
double doubleResult = square (7.5) ;
```

اجرا می گردد تابع **square** با پیش تعریف :

```
double square (double );
```

فراخوانی می شود .

نکته : توجه داشته باشید که توابع گرانبار شده الزامی ندارند که وظیفه یکسانی را انجام دهند . و ممکن است کاملاً با هم تفاوت داشته باشند ، ولی توصیه می شود که توابعی را گرانبار کنید که یک کار را انجام می دهند .

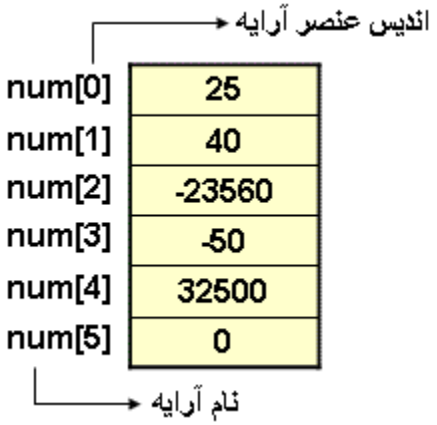
اعلان آرایه ها

یک آرایه مجموعه ای از خانه ها متوالی حافظه می باشد که دارای یک نام و یک نوع می باشند . به هر یک از این خانه ها یک عنصر آرایه گفته می شود . برای دستیابی به یک عنصر آرایه ، باید نام آرایه و شمارنده آن خانه را مشخص کنیم . لذا عناصر آرایه توسط متغیری به نام اندیس مشخص می شوند به همین دلیل، آرایه ها را متغیرهای اندیس دار نیز می گویند . نام آرایه ، از قواعد نام گذاری متغیرها پیروی می کند . نوع آرایه نیز یکی از انواع داده ذکر شده در مبحث مفاهیم حافظه و انواع داده ای می باشد . اعلان آرایه ها به صورت زیر است :

```
نام آرایه [طول آرایه]  نوع داده آرایه ;
```

به عنوان مثال دستور زیر آرایه ای به طول ۶ ، با نام **num** را از نوع **int** ایجاد می کند .

```
int num [6];
```



توجه داشته باشید که تمام عناصر دارای نام یکسانی می باشند و تنها با اندیس از هم تفکیک می شوند . به عنوان مثال عنصر با اندیس **۲** دارای مقدار **۲۳۵۶۰**- می باشد ، ضمناً اندیس عناصر از **۰** شروع می شود .

استفاده از یک عبارت محاسباتی به جای اندیس عناصر امکان پذیر می باشد ، به عنوان مثال با فرض اینکه متغیر **a** حاوی **۲** و متغیر **b** حاوی **۳** باشد ، دستور زیر:

```
num [a+b] + = 3;
```

سه واحد به عنصر **num [5]** اضافه خواهد کرد و این عنصر حاوی عدد **۳** می گردد . برای چاپ مجموع سه عنصر اول آرایه می توانید از دستور زیر استفاده کنید :

```
cout << num[0] + num[1] + num[2] << endl;
```

برای تقسیم عنصر چهارم آرایه بر **۲** و قرار دادن حاصل در متغیر **x** از دستور زیر می توانید استفاده کنید :

```
x = num[3]/2;
```

نکته : توجه داشته باشید که عنصر چهارم آرایه با عنصر شماره چهار (با اندیس چهار) متفاوت می باشد . همانطور که در دستور فوق دیدید عنصر شماره چهار دارای اندیس سه می باشد ، دلیل این امر اینست که اندیس گذاری از صفر شروع می شود . در آرایه فوق عنصر چهارم آرایه **num[3]=-50** می باشد ولی عنصر شماره چهار (با اندیس چهار) **num[4]=32500** می باشد .

همانند متغیرها چند آرایه را نیز می توان توسط یک دستور تعریف کرد :

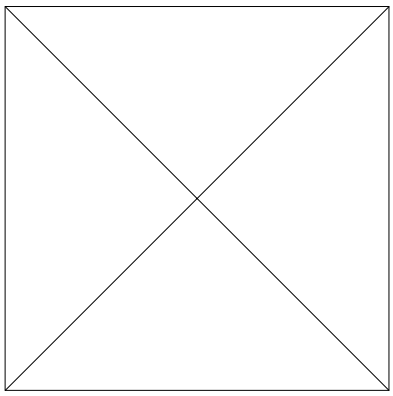
```
int b[100] , x[27] ;
```

دستور فوق **۱۰۰** خانه از نوع عدد صحیح را برای آرایه با نام **b** و **۲۷** خانه از نوع عدد صحیح را برای آرایه با نام **x** در نظر می گیرد .

برای مقدار دهی اولیه به هر یک از عناصر آرایه می توانید از شیوه زیر استفاده کنید :

```
int n[5] = {32 , 27 , 64 , 18 , 95 }
```

دستور فوق عناصر آرایه **n** را به صورت زیر مقدار دهی می کند .



اگر طول آرایه هنگام تعریف آرایه تعیین نشده باشد و لیست مقدار عناصر نوشته شود ، همانند دستور زیر :

```
int n[] = { 1 , 2 , 3 , 4 , 5 }
```

در این صورت کامپایلر به تعداد عناصر لیست ، خانه حافظه برای آرایه در نظر می گیرد ، مثلاً در دستور فوق **۵** خانه حافظه برای آرایه **n** در نظر گرفته می شود .

راه دیگری که برای مقدار دهی اولیه به عناصر آرایه وجود دارد استفاده از روش زیر است :

```
int num[10] = {0}
```

دستور فوق **۱۰** خانه حافظه برای آرایه **num** در نظر می گیرد و مقادیر همه آنها را صفر می کند . توجه داشته باشید که اگر از دستور زیر استفاده کنیم :

```
int num[10] = {1}
```

تمامی عناصر مقدار **۱** را نمی گیرند بلکه عنصر اول آرایه یک می شود و بقیه عناصر مقدار صفر را می گیرند .

در تعریف آرایه دیدید که طول آرایه را با عدد صحیح ثابتی تعیین می کنیم . هر جا که از عدد ثابتی استفاده می شود ، متغیر ثابت نیز می توان به کار برد . متغیرهای ثابت به صورت زیر تعریف می شوند :

```
const    نام متغیر ثابت = مقدار متغیر    نوع داده متغیر  ;
```

به عنوان مثال :

```
const  int  arraySize = 10;
```

دستور فوق عدد **۱۰** را به متغیر **arraySize** ثابت انتساب می دهد . توجه داشته باشید که مقدار یک متغیر ثابت را در طول برنامه نمی توان تغییر داد و نیز متغیر ثابت در هنگام تعریف شدن ، مقدار اولیه اش نیز باید تعیین گردد . به متغیرهای ثابت ، متغیرهای فقط خواندنی نیز گفته می شود . کلمه "متغیر ثابت" یک کلمه مرکب ضد و نقیض می باشد چون کلمه متغیر متضاد ثابت می باشد و این اصطلاحی است که برای اینگونه متغیرهای در اکثر زبانهای برنامه نویسی به کار می رود . برنامه زیر نحوه تعریف یک متغیر ثابت را نشان می دهد :

```
#include <iostream.h>

void main()
{
   const int x = 7;

   cout << "The value of constant variable x is: "
        << x << endl;

}
```

برنامه فوق عدد **۷** را در متغیر ثابت **x** قرار می دهد و توسط دستور **cout** آنرا چاپ می کند . همانطور که گفتیم مقدار متغیر ثابت در هنگام تعریف باید تعیین گردد و نیز ثابت قابل تغییر نمی باشد ، به برنامه زیر توجه کنید .

```
#include <iostream.h>

void main()
{
   const int x;

   x=7;
}
```

برنامه فوق هنگام کامپایل شدن دو پیغام خطا خواهد داد ، چون متغیر ثابت هنگام تعریف مقدار دهی نشده و نیز در برنامه دستوری برای تغییر مقدار آن آورده نشده است .

```
 Compiling C:\TCP\BIN\CONST1.CPP:
Error : Constant variable 'x' must be initialized
Error : Cannot modify a const object
```

مثال : در برنامه زیر طول آرایه توسط متغیر ثابتی تعیین می گردد و عناصر آرایه توسط حلقه **for** مقدار دهی شده و سپس توسط حلقه **for** دیگری ، مقدار عناصر آرایه چاپ می گردد .

```
#include <iostream.h>

void main()
{
   const int arraySize = 10;

   int s[ arraySize ];

   for ( int i = 0; i < arraySize; i++ )
   s[ i ] = 2 + 2 * i;

   cout << "Element Value" << endl;

   for ( int j = 0; j < arraySize; j++ )
       cout << j << "\t " << s[ j ] << endl;

}
```

خروجی برنامه فوق به صورت زیر می باشد :

```
Element  Value
0        2
1        4
2        6
3        8
4        10
5        12
6        14
7        16
8        18
9        20
```

چند مثال از آرایه ها و کاربرد آنها

مثال : برنامه ای بنویسید که ۱۰ عدد صحیح را ورودی دریافت کرده و در آرایه ای قرار داده سپس مجموع عناصر آرایه را محاسبه کرده و درخروجی چاپ نماید .

```
#include <iostream.h>

void main()
{
  const int arraySize = 10;
  int total = 0,i;
  int a[ arraySize ];

  for (i = 0; i < arraySize; i++)
    {
     cout << "Enter number " << i << " : ";
     cin >> a[ i ];
    }

  for (i = 0; i < arraySize; i++ )
     total += a[ i ];

  cout << "Total of array element values is "
       << total << endl;

}
```

خروجی برنامه به صورت زیر می باشد :

```
Enter number 0 : 12
Enter number 1 : 3
Enter number 2 : 4
Enter number 3 : 654
Enter number 4 : 34
Enter number 5 : 2
Enter number 6 : 123
Enter number 7 : 794
Enter number 8 : 365
Enter number 9 : 23
Total of array element values is 2014
```

مثال : برنامه ای بنویسید که توسط آرایه ، نمودار میله ای افقی برای اعداد {۱ و۱۷ و۵ و۱۳ و۹ و۱۱ و۷ و۱۵ و۳ و۱۹ } رسم کند .

```
#include <iostream.h>

int main()
{
  const int arraySize = 10;
  int n[ arraySize ] = { 19, 3, 15, 7, 11,
                         9, 13, 5, 17, 1 };

  cout << "Element" << " Value" << endl;

  for ( int i = 0; i < arraySize; i++ ) {
     cout << i << "\t " << n[ i ] << "\t";

     for ( int j = 0; j < n[ i ]; j++ )
        cout << '*';
     cout << endl;
  }

return 0;

}
```

خروجی برنامه به صورت زیر می باشد :

```
Element  Value
0        19     *******************
1        3      ***
2        15     ***************
3        7      *******
4        11     ***********
5        9      *********
6        13     *************
7        5      *****
8        17     *****************
9        1      *
```

مثال : برنامه ای بنویسید که یک تاس را **۶۰۰۰** بار پرتاب کرده و توسط آرایه ای تعداد دفعات آمدن هر وجه را حساب کند .(تعداد دفعات آمدن هر وجه را یک عنصر آرایه ای در نظر بگیرید)

```
#include <iostream.h>
#include <stdlib.h>
#include <time.h>

void main()
{
  const int arraySize = 7;
  int frequency[ arraySize ] = { 0 };

  srand( time( 0 ) );

  for ( int roll = 1; roll <= 6000; roll++ )
     ++frequency[ 1 + rand() % 6 ];

  cout << "Face Frequency" << endl;

  for ( int face = 1; face < arraySize; face++ )
     cout << face << "\t" << frequency[face] << endl;

}
```

خروجی برنامه به صورت زیر می باشد :

```
Face  Frequency
1       1023
2       990
3       1008
4       971
5       1025
6       983
```

دستور **++frequency [rand()%6 + 1];** ، مقدار عنصر مربوط به هر وجه را یک واحد اضافه می کند ، زیرا **rand()%6 + 1** عددی بین ۱ تا ۶ تولید می کند ، پس هر بار به طور تصادفی تنها مقدار عنصر مربوط به یکی از وجوه افزایش می یابد .

یکی از کار برد های آرایه ها ، استفاده از آنها برای ذخیره رشته ای از حروف می باشد . تا به حال متغیرهایی که از ورودی دریافت می کردیم و یا آرایه هایی که تا به اینجا دیدید تنها شامل اعداد می شدند در اینجا به عنوان مثال نحوه دریافت یک نام از ورودی و چاپ آن در خروجی را بررسی می کنیم .(در فصل بعد یعنی اشاره گرها و رشته ها ، به طور مفصل تر راجع به رشته ها صحبت خواهیم کرد)

یک عبارت رشته ای مانند: **"hello"** در واقع آرایه ای از حروف می باشد.

```
char string1[]="hello";
```

دستور فوق آرایه ای به نام **string1** را با کلمه **hello** مقدار دهی می کند. طول آرایه فوق برابر است با طول کلمه **hello** یعنی ۵ بعلاوه یک واحد که مربوط است به کاراکتر پوچ که انتهای رشته را مشخص می کند. لذا آرایه **string1** دارای طول ۶ می باشد. کاراکتر پوچ در زبان C++ توسط **'\0'** مشخص می گردد. انتهای کلیه عبارات رشته ای با این کاراکتر مشخص می شود.

```
char string1[]={'h','e','l','l','o','\0'}
```

دستور فوق عناصر آرایه string1 را جداگانه مقدار دهی می کند. توجه داشته باشید که عناصر آرایه در دستور فوق داخل (') قرار گرفتند و نیز انتهای رشته با **'\0'** تعیین شد. نتیجه همانند دستور **char string1[]="hello";** می باشد.

چون عبارت رشته ای، آرایه ای از حروف می باشند، لذا به هر یک از حروف رشته، توسط اندیس عنصری که شامل آن حرف می باشد، می توان دسترسی پیدا کرد. به عنوان مثال **string1[0]** شامل **'h'** و **string1[3]** شامل **'l'** و **string1[5]** شامل **'\0'** می باشد.

توسط دستور cin نیز می توان به طور مستقیم کلمه وارد شده از صفحه کلید را در آرایه ای رشته ای قار داد.

```
char string2[20];
```

دستور فوق یک آرایه رشته ای که قابلیت دریافت کلمه ای با طول ۱۹ به همراه کاراکتر پوچ را دارا می باشد.

```
cin >> string2;
```

دستور فوق رشته ای از حروف را از صفحه کلید خوانده و در **string2** قرار می دهدو کاراکتر پوچ را به انتهای رشته وارد شده توسط کاربر اضافه می کند. به طور پیش فرض دستور cin کاراکتر ها را از صفحه کلید تا رسیدن به اولین فضای خالی در یافت می کند. به عنوان مثال اگر هنگام اجرای دستور **cin >> string2;** کاربر عبارت "**hello there**" را وارد کند، تنها کلمه **hello** در **string2** قرار می گیرد. چون عبارت وارد شده شامل کاراکتر فاصله است. برنامه زیر نحوه به کار گیری آرایه های رشته ای را نشان می دهد.

```
#include <iostream.h>

void main()
{
  char name[ 20 ];

  cout << "Please Enter your name : ";
```

```
  cin >> name;

  cout << "Welcome, " << name
       << " to this program. \n" ;

  cout << "Your separated name is\n";

  for ( int i = 0; name[ i ] != '\0'; i++ )
       cout << name[ i ] << ' ';

}
```

خروجی برنامه به صورت زیر می باشد :

```
Please Enter your name : Mohammad
Welcome, Mohammad to this program.
Your separated name is
M o h a m m a d
```

در برنامه فوق حلقه for ، حروف نام وارد شده توسط کاربر را جدا جدا در خروجی چاپ می کند. ضمنا شرط حلقه **name[i] != '\0'** می باشد و تا وقتی این شرط برقرار است که حلقه به انتهای رشته نرسیده باشد.

در مبحث قوانین حوزه دیدید که اگر بخواهیم یک متغیر محلی تابع، مقدار خود را حفظ کرده و برای دفعات بعدی فراخوانی تابع نیز نگه دارد، از کلمه **static** استفاده کردیم. نوع **static** را برای آرایه ها نیز می توان به کار برد و از همان قوانین گفته شده در مبحث مذکور پیروی می کند. به برنامه زیر و خروجی آن توجه کنید.

```
#include <iostream.h>

void staticArrayInit( void );
void automaticArrayInit( void );

int main()
{
  cout << "First call to each function:\n";
  staticArrayInit();
  automaticArrayInit();

  cout << "\n\nSecond call to each function:\n";

  staticArrayInit();
  automaticArrayInit();
  cout << endl;
```

```
return 0;

}

// function to demonstrate a static local array
void staticArrayInit( void )
{
  // initializes elements to 0
  // first time function is called
  static int array1[ 3 ]={0};

  cout << "\nValues on entering staticArrayInit:\n";

  // output contents of array1
  for ( int i = 0; i < 3; i++ )
     cout << "array1[" << i << "] = "
          << array1[ i ] << " ";

  cout << "\nValues on exiting staticArrayInit:\n";

  // modify and output contents of array1
  for ( int j = 0; j < 3; j++ )
     cout << "array1[" << j << "] = "
          << ( array1[ j ] += 5 ) << " ";

} // end function staticArrayInit

// function to demonstrate an automatic local array
void automaticArrayInit( void )
{
  // initializes elements each time function is called
  int array2[ 3 ] = { 1, 2, 3 };

  cout << endl << endl;
  cout << "Values on entering automaticArrayInit:\n";

  // output contents of array2
  for ( int i = 0; i < 3; i++ )
     cout << "array2[" << i << "] = "
          << array2[ i ] << " ";

  cout << "\nValues on exiting automaticArrayInit:\n";

  // modify and output contents of array2
  for ( int j = 0; j < 3; j++ )
    cout << "array2[" << j << "] = "
```

```
        << ( array2[ j ] += 5 ) << " ";

}
```

خروجی برنامه به صورت زیر می باشد :

```
First call to each function:

Values on entering staticArrayInit:
array1[0] = 0  array1[1] = 0  array1[2] = 0
Values on exiting staticArrayInit:
array1[0] = 5  array1[1] = 5  array1[2] = 5

Values on entering automaticArrayInit:
array2[0] = 1  array2[1] = 2  array2[2] = 3
Values on exiting automaticArrayInit:
array2[0] = 6  array2[1] = 7  array2[2] = 8

Second call to each function:

Values on entering staticArrayInit:
array1[0] = 5  array1[1] = 5  array1[2] = 5
Values on exiting staticArrayInit:
array1[0] = 10  array1[1] = 10  array1[2] = 10

Values on entering automaticArrayInit:
array2[0] = 1  array2[1] = 2  array2[2] = 3
Values on exiting automaticArrayInit:
array2[0] = 6  array2[1] = 7  array2[2] = 8
```

در برنامه فوق عناصر آرایه **array1** در اولین بار فراخوانی تابع **staticArrayInit** مقدار صفر را می گیرند ولی در دفعات بعدی فراخوانی این تابع، آخرین مقدار قبلی خود را حفظ می کنند . اما آرایه **array2** در هر بار فراخوانی تابع **automaticArrayInit** مقدار دهی اولیه می شود و با خروج از تابع مقدار خود را از دست می دهد.

ارسال آرایه ها به توابع

برای ارسال یک آرایه به عنوان آرگومان به یک تابع ، کافیست نام آرایه را بدون علامت براکت ([]) به کار ببرید . به عنوان مثال اگر آرایه ای با نام **x** به صورت زیر تعریف شده باشد :

```
int x[24];
```

برای ارسال آن به تابع **modifyArray** کافیست تابع را به صورت زیر:

```
modifyArray(x,24);
```

فراخوانی کنید . دستور فوق آرایه و طول آن را به تابع **modifyArray** ارسال می کند . معمولاً هنگامی که آرایه ای را به تابعی ارسال می کنند ، طول آرایه را نیز همراه آرایه به عنوان یک آرگومان جداگانه به تابع می فرستند.

C++ آرایه ها را با شیوه شبیه سازی شده ارسال آرگومان ها با ارجاع به تابع ارسال می نماید ، لذا تابع فراخوانی شده مقدار عناصر آرایه ارسالی را می تواند تغییر دهد . هنگامی که نام تابع را به عنوان آرگومان تابع به کار می بریم ، آدرس خانه حافظه اولین عنصر آرایه به تابع ارسال می شود لذا تابع می داند که عناصر آرایه در کجای حافظه قرار گرفته اند . بنابراین هنگامی که تابع فراخوانی شده عناصر آرایه را تغییر می دهد ، این تغییرات روی عناصر آرایه اصلی که به تابع ارسال شده است ، انجام می پذیرد .

نکته : توجه داشته باشید که عناصر آرایه را به صورت جداگانه همانند ارسال متغیرها و یا مقادیر عددی به تابع ارسال کرد . در این صورت تغییر بر روی آرگومان ارسالی ، تأثیری بر روی عنصر آرایه نخواهد داشت . در حقیقت این شیوه ، همان ارسال با مقدار می باشد .

برای اینکه تابعی قادر به دریافت یک آرایه به عنوان ورودی باشد ، هنگام تعریف تابع در لیست آرگومانهای آن ، این مطلب باید مشخص گردد . به عنوان مثال تعریف تابع **modifyArray** را به صورت زیر می توان نوشت :

```
void modifyArray (int b[] ,int array size)
```

دستور فوق مشخص می کند که تابع **modifyArray** قادر به دریافت آدرس آرایه ای از اعداد صحیح توسط آرگومان **b** و تعداد عناصر آرایه توسط آرگومان **arraySize** می باشد . ضمناً تعداد عناصر آرایه را لازم نیست بین براکت ها ([]) بنویسید ، اگر این کار نیز صورت پذیرد ، کامپایلر آن را نادیده می گیرد . توجه داشته باشید که پیش تعریف تابع فوق را به صورت زیر بنویسید :

```
void modifyArray (int [] , int);
```

برنامه زیر نحوه ارسال یک آرایه را به تابع و تفاوت ارسال یک عنصر آرایه به تابع و ارسال کل آرایه به تابع را نشان می دهد .

```
#include <iostream.h>

void modifyArray( int [], int );
```

```
void modifyElement( int );

void main()
{
  const int arraySize = 5;
  int a[ arraySize ] = { 0, 1, 2, 3, 4 };

  cout<<"Effects of passing entire array by reference:"
      <<"\n\nThe values of the original array are:\n";

  // output original array
  for ( int i = 0; i < arraySize; i++ )
    cout << "\t"<< a[ i ];

  cout << endl;

  // pass array a to modifyArray by reference
  modifyArray( a, arraySize );

  cout << "The values of the modified array are:\n";

  // output modified array
  for ( int j = 0; j < arraySize; j++ )
    cout << "\t" << a[ j ];

  // output value of a[ 3 ]
  cout<<"\n\n\n"
      <<"Effects of passing array element by value:"
      <<"\n\nThe value of a[3] is " << a[ 3 ] << '\n';

  // pass array element a[ 3 ] by value
  modifyElement( a[ 3 ] );

  // output value of a[ 3 ]
  cout << "The value of a[3] is " << a[ 3 ] << endl;

}

// in function modifyArray, "b" points to
// the original array "a" in memory
void modifyArray( int b[], int sizeOfArray )
{
  // multiply each array element by 2
  for ( int k = 0; k < sizeOfArray; k++ )
  b[ k ] *= 2;
}
```

```
// in function modifyElement, "e" is a local copy of
// array element a[ 3 ] passed from main
void modifyElement( int e )
{
  // multiply parameter by 2
  cout << "Value in modifyElement is "
       << ( e *= 2 ) << endl;
}
```

خروجی برنامه فوق به صورت زیر می باشد :

```
Effects of passing entire array by reference:

The values of the original array are:
        0       1       2       3       4
The values of the modified array are:
        0       2       4       6       8


Effects of passing array element by value:

The value of a[3] is 6
Value in modifyElement is 12
The value of a[3] is 6
```

در برنامه فوق ، تابع **modifyArray** مقدار عناصر آرایه **a** را که به آن فرستاده شده است دو برابر می کند . تابع **modifyElement** مقدار آرگومان دریافتی را دو برابر کرده و در خروجی چاپ می کند ولی تأثیری در نسخه اصلی عنصر آرایه نداشته و تغییری در مقدار آن ایجاد نمی کند .

بعضی مواقع ممکن است بخواهید که تابعی ، اجازه تغییر عناصر آرایه ای که به آن فرستاده شده است را نداشته باشد . برای این کار هنگام تعریف تابع کافی است از کلمه **const** قبل از آرگومان مربوط به آن آرایه استفاده کنید ، در چنین حالتی اگر داخل تابع قصد تغییر مقدار عناصر آرایه را داشته باشید با یک پیغام خطای کامپایلر مواجه می شوید و کامپایلر اجازه این کار را به شما نمی دهد . به برنامه زیر توجه کنید :

```
#include <iostream.h>

void tryToModifyArray( const int [] );

void main()
{
  int a[] = { 10, 20, 30 };
```

```
  tryToModifyArray( a );

  cout << a[0] <<' '<< a[1] <<' '<< a[2] <<'\n';

}

// In function tryToModifyArray, "b" cannot be used
// to modify the original array "a" in main.
void tryToModifyArray( const int b[] )
{
  b[ 0 ] /= 2; // error
  b[ 1 ] /= 2; // error
  b[ 2 ] /= 2; // error
}
```

هنگام کامپایل کردن برنامه فوق با پیغام های خطای زیر مواجه خواهید شد ، چون در تابع قصد تغییر عناصر آرایه ای را که به صورت ثابت به تابع ارسال شده بود ، داشتیم .

```
Error in line 19: Cannot modify a const object
Error in line 20: Cannot modify a const object
Error in line 21: Cannot modify a const object
```

مرتب کردن آرایه ها

مرتب کردن اطلاعات چه به صورت صعودی یا نزولی ، یکی از مهمترین وظایف کامپیوتر می باشد . به عنوان مثال تعیین رتبه دانش آموزان یک مدرسه بر اساس معدل ، تعیین رتبه شرکت کنندگان در کنکور ، مرتب کردن شماره تلفن ها بر اساس نام صاحب تلفن را می توان نام برد . برای آشنایی با شیوه مرتب کردن ، لیست اعداد زیر را در نظر بگیرید :

2,5,4,3,6,1

برای مرتب کردن لیست اعداد فوق از کوچک به بزرگ آنها را در آرایه ای قرار می دهیم :

```
int a[] = { 2 , 5 , 4 , 3 , 6 , 1};
```

حال کافی است آرایه **a** را به صورت صعودی مرتب کنیم . برای انجام این کار از روشی به نام مرتب کردن حبابی استفاده می کنیم . این تکنیک به دلیل اینکه مقادیر کوچکتر همانند حبابی در آب به سمت بالا حرکت می کنند ، مرتب کردن حبابی

گفته می شود . برای مرتب کردن آرایه چندین بار باید روی آرایه حرکت کنیم و در هر بار حرکت عناصر دو به دو با هم مقایسه می شوند ، و در صورتی که به صورت نزولی قرار داشته باشند مقادیرشان جابه جا می گردد و در غیر این صورت به همان ترتیب باقی می مانند .

برنامه زیر لیست اعداد ذکر شده را به شیوه مرتب کردن حبابی ، از کوچک به بزرگ مرتب می کند .

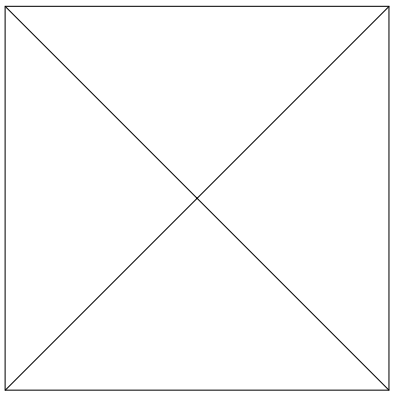
```
#include <iostream.h>

void showArray(const int [] , int);

void main()
{
  const int arraySize = 6;
  int a[ arraySize ] = { 2, 5, 4, 3, 6 ,1};
  int hold;

  cout << "Data items in original order\n";

  showArray(a,arraySize);

  for ( int i = 0; i < arraySize - 1 ; i++ )
     for ( int j = 0; j < arraySize - 1; j++ )
        if ( a[ j ] > a[ j + 1 ] ) {
           hold = a[ j ];
           a[ j ] = a[ j + 1 ];
           a[ j + 1 ] = hold;
         }

  cout << "\nData items in ascending order\n";

  showArray(a,arraySize);

}

void showArray( const int array[] ,int arraySize)
{
  for (int c=0; c<arraySize ;c++)
     cout << array[c] << " ";
     cout << endl;
}
```

خروجی برنامه فوق به صورت زیر می باشد :

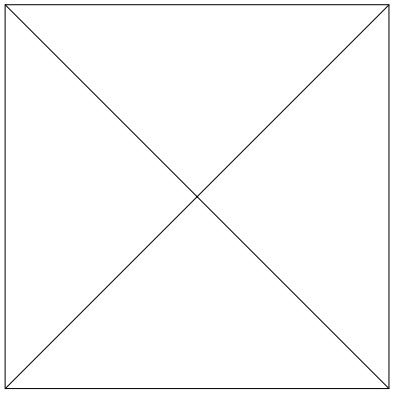
```
Data items in original order
```

```
2 5 4 3 6 1

Data items in ascending order
1 2 3 4 5 6
```

در برنامه فوق تابع **showArray** وظیفه نمایش عناصر آرایه را به عهده دارد . در اولین اجرای دستورات حلقه ها ، **i = 0** می باشد . در اولین دور اجرای حلقه داخلی ، با شمارنده **j** عناصر آرایه به صورت زیر با هم مقایسه می شوند .



پس از اولین دور حرکت روی عناصر آرایه، ترتیب اعداد به صورت فوق خواهد شد. سپس **i = 1** می گردد و دفعات بعدی مقایسه انجام گرفته و در انتهای هر بار مقایسه ترتیب عناصر به صورت زیر می شود .



که سرانجام با به انتها رسیدن حرکت روی آرایه عناصر به صورت صعودی مرتب می شوند .

جستجو در آرایه ها

عمل جستجو یکی از مهمترین وظایف برنامه های کامپیوتری می باشد . به عنوان مثال دفتر تلفنی را در نظر بگیرید که به دنبال نام فردی در آن می گردیم و یا جستجوی نام یک دانشجو در لیست دانشجویان کلاس . در این مبحث دو روش جستجو را مورد بررسی قرار می دهیم . یک روش جستجوی خطی است که معمولاً در آرایه های نا مرتب مورد استفاده قرار می گیرد و روش دیگر جستجوی دو دویی می باشد که در آرایه های مرتب از این شیوه می توانیم استفاده کنیم .

در روش جستجوی خطی ، عنصر مورد جستجو با هر یک از عناصر آرایه مقایسه می شود ، چنانچه دو عنصر برابر بودند ، عمل جستجو به پایان می رسد و اندیس عنصر برگردانده می شود و گرنه مقایسه با عنصر بعدی آرایه انجام می پذیرد . از آنجا که عناصر آرایه نا مرتب می باشند عنصر مورد جستجو در هر کجای آرایه می تواند باشد لذا عمل مقایسه تا یافتن عنصر مورد نظر و یا رسیدن به انتهای آرایه یعنی جستجو در همه عناصر آرایه ادامه می یابد . برنامه زیر نمونه ای از جستجوی خطی در آرایه می باشد :

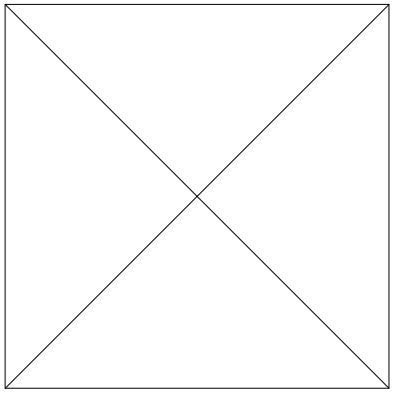
```
#include <iostream.h>

int linearSearch(const int [], int, int );

void main()
{
   const int arraySize = 7;
   int a[ arraySize ]={2,6,4,3,12,10,5};
   int searchKey;

   cout << "Enter integer search key: ";
   cin >> searchKey;

   int element=linearSearch(a, searchKey, arraySize);

   if ( element != -1 )
      cout << "Found value in element "
           << element << endl;
   else
      cout << "Value not found" << endl;

}

int linearSearch( const int array[],
                  int key, int sizeOfArray )
{
   for ( int j = 0; j < sizeOfArray; j++ )

   if ( array[ j ] == key )
      return j;

   return -1;
}
```

خروجی برنامه فوق به صورت زیر می باشد :

```
Enter integer search key: 12
```

```
Found value in element 4
```

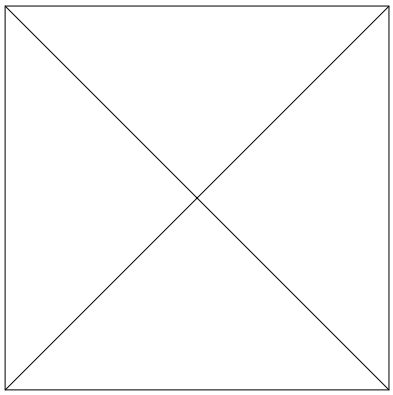
نحوه جستجوی عدد ۱۲ در آرایه به صورت زیر می باشد :



بار دیگر برنامه را برای جستجوی عدد ۲۰ در آرایه اجرا می کنیم . خروجی برنامه به صورت زیر می باشد .

```
Enter integer search key: 20
Value not found
```

نحوه جستجوی عدد ۲۰ در آرایه به صورت زیر می باشد :



روش جستجوی دو دویی در آرایه های مرتب شده قابل استفاده می باشد و از سرعت بالایی برخوردار می باشد . در این الگوریتم ، در هر بار مقایسه ، نیمی از عناصر آرایه حذف می شوند . الگوریتم عنصر میانی آرایه را می یابد و آن را با عنصر مورد جستجو، مقایسه می کند . اگر برابر بودند ، جستجو به پایان رسیده و اندیس عنصر برگردانده می شود ، در غیر این صورت عمل جستجو روی نیمی از عناصر انجام می گیرد . اگر عنصر مورد جستجو کوچکتر از عنصر میانی باشد ، جستجو روی نیمه اول آرایه صورت می پذیرد ، در غیر این صورت نیمه دوم آرایه جستجو می شود . این جستجوی جدید روی زیر آرایه طبق الگوریتم جستجو روی آرایه اصلی انجام می شود یعنی عنصر میانی زیر آرایه یافته می شود و با عنصر مورد

جستجو مقایسه می گردد ، اگر برابر نباشند زیر آرایه مجدداً نصف می شود و در هر بار جستجو زیر آرایه ها کوچکتر می گردند . عمل جستجو تا یافتن عنصر مورد نظر(یعنی برابر بودن عنصر مورد جستجو با عنصر میانی یکی از زیر آرایه ها) و یا نیافتن عنصر مورد نظر (برابر نبودن عنصر مورد جستجو با عنصر زیر آرایه ای شامل تنها یک عنصر) ادامه می یابد . برنامه زیر نمونه ای از جستجوی دو دویی در آرایه مرتب می باشد .

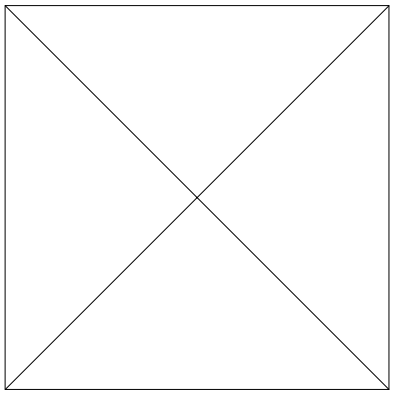
```
#include <iostream.h>

int binarySearch( const int [], int, int);

void main()
{
  const int arraySize = 15;
  int a[ arraySize ]={0,2,4,6,8,10,12,14,
                      16,18,20,22,24,26,28};
  int key;

  cout << "Enter a number between 0 and 28: ";
  cin >> key;

  int result =
      binarySearch( a, arraySize, key);

  if ( result != -1 )
    cout << '\n' << key << " found in array element "
         << result << endl;
  else
    cout << '\n' << key << " not found" << endl;
}

int binarySearch( const int b[],
                  int arraySize ,
                  int searchKey )
{
  int middle,low=0,high=arraySize - 1;

  while ( low <= high )
  {
    middle = ( low + high ) / 2;
    if ( searchKey < b[ middle ] )
      high = middle - 1;
    else
      if ( searchKey > b[ middle ] )
        low = middle + 1;
      else return middle;
  }
```

```
  return -1;
}
```

خروجی برنامه فوق به صورت زیر می باشد :

```
Enter a number between 0 and 28: 8

8 found in array element 4
```

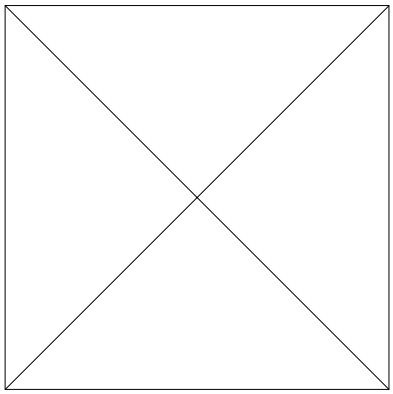
نحوه جستجوی عدد ۸ در آرایه به صورت زیر می باشد :



بار دیگر برنامه را برای جستجوی عدد ۲۵ اجرا می کنیم . خروجی برنامه به صورت زیر می باشد :

```
Enter a number between 0 and 28: 25

25 not found
```

نحوه جستجوی عدد ۲۵ در آرایه به صورت زیر می باشد :

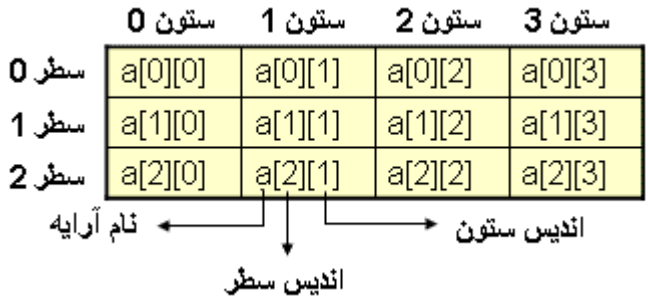


آرایه های چند بعدی

آرایه ها در **C++** می توانند بیش از یک اندیس داشته باشند . بدین صورت یک آرایه چند اندیسه یا چند بعدی خواهیم داشت . کاربردی ترین آرایه چند بعدی ، آرایه دو بعدی می باشد که توسط آن می توان جدولی حاوی مقادیر مختلف را شبیه سازی کرد . به دستور زیر توجه کنید :

```
int a[3][4];
```

دستور فوق یک آرایه دو بعدی **۳** در **۴** را به صورت زیر ایجاد می کند :



هر عنصر آرایه به صورت **a[i][j]**، که در آن **i** شماره سطر و **j** شماره ستون می باشد ، قابل دسترسی است .

برای مقدار دهی اولیه به عناصر آرایه می توانید مانند دستور زیر عمل کنید :

```
int b[2][2] = {{1,2},{3,4}};
```

دستور فوق آرایه **b** را به صورت زیر مقدار دهی می کند :

2	1
4	3

در برنامه زیر چند نمونه از مقدار دهی اولیه به آرایه دو بعدی ۲ در ۳ آورده شده است :

```
#include <iostream.h>

void printArray( int [][ 3 ] );
```

```
void main()
{
  int array1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } };
  int array2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 };
  int array3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };

  cout << "Values in array1 by row are:" << endl;
  printArray( array1 );

  cout << "Values in array2 by row are:" << endl;
  printArray( array2 );

  cout << "Values in array3 by row are:" << endl;
  printArray( array3 );

}

void printArray( int a[][ 3 ] )
{
  for ( int i = 0; i < 2; i++ )
  {
    for ( int j = 0; j < 3; j++ )
      cout << a[ i ][ j ] << ' ';
    cout << endl;
  }

}
```

خروجی برنامه به صورت زیر می باشد :

```
Values in array1 by row are:
1 2 3
4 5 6
Values in array2 by row are:
1 2 3
4 5 0
Values in array3 by row are:
1 2 0
4 0 0
```

در برنامه فوق تابع **PrintArray** وظیفه چاپ عناصر آرایه را بر روی صفحه نمایش دارا می باشد . توجه داشته باشید که ارسال آرایه به تابع به صورت**int a[][3]** انجام گرفت . اگر بیاد داشته باشید در آرایه های یک بعدی نیازی به ذکر طول آرایه نبود اما آرایه های بیش از یک بعد تعداد عناصر بعدهای دیگر باید ذکر شود ، اما نیازی به ذکر طول بعد اول نمی باشد .

مثال : در برنامه زیر آرایه **۲** بعدی **۱۰** در **۱۰** را با مقادیر جدول ضرب ، مقدار دهی می کنیم و سپس آن را بر روی صفحه نمایش چاپ می کنیم .

```
#include <iostream.h>
void main( )
{

  int a[10][10],i,j;

  for (i=0;i<10;i++)
    for (j=0;j<10;j++)
      a[i][j]=(i+1)*(j+1);


  for (i=0;i<10;i++){
    for (j=0;j<10;j++)
      cout <<a[i][j]<<"\t";
    cout<<endl;
  }
}
```

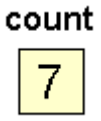
خروجی برنامه فوق به صورت زیر می باشد :

```
1   2   3   4   5   6   7   8   9   10

2   4   6   8   10  12  14  16  18  20

3   6   9   12  15  18  21  24  27  30

4   8   12  16  20  24  28  32  36  40

5   10  15  20  25  30  35  40  45  50

6   12  18  24  30  36  42  48  54  60

7   14  21  28  35  42  49  56  63  70

8   16  24  32  40  48  56  64  72  80
```

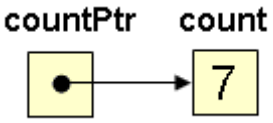
```
9   18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90  100
```

تعریف اشاره گر

متغیرهای اشاره گر، آدرس خانه های حافظه را در خود نگهداری می کنند . متغیرها معمولاً مقدار مشخصی را در خود دارند ولی اشاره گرها آدرس یک متغیر را در خود دارند . نام یک متغیر به طور مستقیم به یک مقدار ، مراجعه می کند اما یک اشاره گر به طور غیر مستقیم به یک مقدار مراجعه می کند . به شکل زیر توجه کنید :



count به طور مستقیم به مقدار **۷** مراجعه می کند .



countPtr به طور غیر مستقیم به متغیری که حاوی **۷** می باشد مراجعه می کند .

اشاره گرها نیز مانند هر متغیر دیگری ، قبل از استفاده باید تعریف شوند . به عنوان مثال دستور زیر متغیر **count** را از نوع **int** و متغیر **countPtr** را اشاره گری به متغیری از نوع **int** تعریف می کند .

```
int count , *countPtr ;
```

برای تعریف هر متغیری از نوع اشاره گر ، از علامت ستاره * قبل از نام آن استفاده می کنیم .

به دستور زیر توجه کنید :

```
double *xPtr , *yPtr ;
```

در دستور فوق ، **xPtr** و **yPtr** اشاره گرهایی به متغیرهایی از نوع **double** تعریف می شوند .

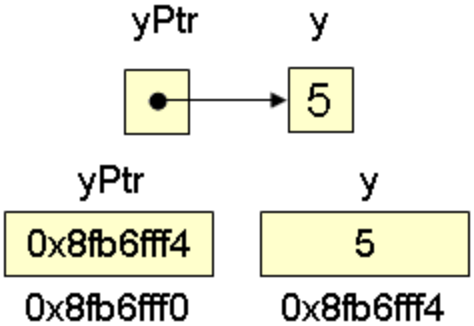
نکته: استفاده از **Ptr** در انتهای نام متغیرهای اشاره گر الزامی نمی باشد ولی برای اینکه برنامه قابل فهم تر باشد توصیه می شود از **Ptr** در انتهای نام اشاره گر استفاده کنید .

عملگرهای اشاره گر

عملگر آدرس (**&**) عملگری است که آدرس خانه حافظه عملوند خود را بر می گرداند .

```
int y=5;
int *yPtr;
yPtr = &y;
```

دستورات فوق متغیر **y** را از نوع **int** با عدد ۵ مقدار دهی کرده و سپس **yPtr** ، اشاره گری به متغیری از نوع **int** تعریف می شود و سرانجام آدرس خانه حافظه **y** در **yPtr** قرار می گیرد .



همانطور که در شکل فوق می بینید ، **yPtr** حاوی آدرس خانه حافظه **y** می باشد .

برای آشنایی با نحوه استفاده از اشاره گرها به برنامه زیر توجه کنید :

```
#include <iostream.h>

void main ()
{
  int x = 5, y = 15;
  int *xPtr, *yPtr;

  xPtr = &x;
  yPtr = &y;
```

```
  cout << "The value of x is " << x
       << "\nThe address of x is " << &x
       << "\nThe value of xPtr is " << xPtr;

  cout << "\n\nThe value of y is " << y
       << "\nThe address of y is " << &y
       << "\nThe value of yPtr is " << yPtr;

  *xPtr = 10;
  cout << "\n\nx=" << x << " and y=" << y;

  *yPtr = *xPtr;
  cout << "\nx=" << x << " and y=" << y;

  xPtr = yPtr;
  cout << "\nx=" << x << " and y=" << y;

  *xPtr = 20;
  cout << "\nx=" << x << " and y=" << y;
}
```

خروجی برنامه فوق به صورت زیر می باشد :

```
The value of x is 5
The address of x is 0x8fb4fff4
The value of xPtr is 0x8fb4fff4

The value of y is 15
The address of y is 0x8fb4fff2
The value of yPtr is 0x8fb4fff2

x=10 and y=15
x=10 and y=10
x=10 and y=10
x=10 and y=20
```

در برنامه فوق دو متغیر **x** و**y** از نوع عدد صحیح تعریف شده و **x** حاوی ۵ و **y** حاوی ۱۵ می گردد سپس **xPtr** و **yPtr** اشاره گری به عدد صحیح تعریف می شوند .

```
xPtr = &x;
yPtr = &y;
```

دو دستور فوق همانطور که در خروجی برنامه نیز می بیند ، آدرس خانه حافظه **x** را در **xPtr** و آدرس خانه حافظه **y** را در **yPtr** قرار می دهد .

دستور ***xPtr = 10;** در خانه ای از حافظه که **xPtr** اشاره می کند (یعنی متغیر **x**) عدد ۱۰ را قرار می دهد سپس***yPtr = *xPtr;** مقدار خانه حافظه ای که **xPtr** به آن اشاره می کند را در خانه ای از حافظه که **yPtr** به آن اشاره می کند قرار می دهد یعنی مقدار متغیر **x** در متغیر **y** قرار می گیرد .

دستور **xPtr = yPtr;** مقدار **yPtr** را که همان آدرس خانه حافظه **y** می باشد در **xPtr** قرار می دهد پس با اجرای این دستور **xPtr** دیگر به **x** اشاره نمی کند بلکه به **y** اشاره خواهد کرد ، لذا با اجرای دستور ***xPtr = 20;** همانطور که مشاهده می کنید **x** حاوی ۲۰ نمی شود بلکه این مقدار **y** است که به ۲۰ تغییر می یابد .

ارسال آرگومان به تابع توسط اشاره گر

تا به حال با دو روش ارسال آرگومانها به توابع آشنا شده اید . ارسال با مقدار و ارسال با ارجاع . در این مبحث روش دیگری را که ارسال توسط اشاره گر می باشد مورد بررسی قرار می دهیم . اشاره گرها مانند آرگومانهای ارجاع می توانند برای تغییر یک یا چند متغیر ارسال شده از داخل تابع و یا برای ارسال داده های بزرگ به توابع مورد استفاده قرار گیرند . در برنامه زیر ، شیوه ارسال آرگومان توسط اشاره گر به تابع مورد استفاده قرار گرفته است .

```
#include <iostream.h>

void callByPointer( int * );

int main()
{
  int number = 5;

  cout << "The original value of number is " << number;

  // pass address of number to callByPointer
  callByPointer( &number );

  cout << "\nThe new value of number is "
       << number << endl;

  return 0;
}

void callByPointer( int *nPtr )
{
  *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
}
```

خروجی برنامه فوق به صورت زیر می باشد :

```
The original value of number is 5
The new value of number is 125
```

همانطور که در برنامه فوق مشاهده می کنید ، برای ارسال آرگومان به تابع توسط اشاره گر ، در پیش تعریف تابع پس از مشخص کردن نوع آرگومان از علامت * استفاده می کنیم و در تعریف تابع نیز علامت * را قبل از نام آرگومان اشاره گر قرار می دهیم . ضمناً از آنجا که اشاره گرها آدرس متغیرها را در خود قرار می دهند برای ارسال آرگومان توسط اشاره گر به یک تابع ، هنگام فراخوانی تابع باید نام متغیر ارسالی را همراه علامت **&** به کار ببریم چون تنها در این صورت آدرس متغیر ارسال می گردد .

Constو اشاره گرها

همانطور که می دانید **const** به برنامه نویس این امکان را می دهد که مقدار یک متغیر را که از نوع **const** تعریف شده است ، در طول برنامه نتوان تغییر داد . داده هایی که توسط اشاره گرها به توابع ارسال می شوند ، داخل تابع قابل تغییر می باشند و ممکن است نا خواسته تغییر یابند . برای جلوگیری از تغییرات نا خواسته ، آنها را به صورت ثابت به تابع ارسال می کنیم . به برنامه زیر توجه کنید :

```
void f( const int * ); // prototype

int main()
{
  int y;

  f( &y ); // f attempts illegal modification

  return 0;
}

// xPtr cannot modify the value of the variable
// to which it points
void f( const int *xPtr )
{
  *xPtr = 100; // error: cannot modify a const object
}
```

برنامه فوق هنگام کامپایل شدن پیغام خطایی مبنی بر اینکه داده ثابت قابل تغییر نمی باشد را خواهد داد .

```
Compiling CONSTP1.CPP:
Error CONSTP1.CPP 16: Cannot modify a const object
```

همانطور که مشاهده می کنید برای ارسال آرگومان توسط اشاره گر به صورت ثابت به تابع از دستور **const int *xptr** استفاده کردیم . و در تابع هنگام تغییر مقدار اشاره گر ثابت با پیغام خطا مواجه شدیم .

یکی دیگر از کاربردهای **const** همراه اشاره گرها ، ایجاد اشاره گر ثابتی به داده ای غیر ثابت می باشد . چنین اشاره گری همواره به یک خانه حافظه اشاره می کند و نمی توان آن را به خانه حافظه دیگری اشاره داد ولی می توان داده داخل همان خانه حافظه را تغییر داد . به برنامه زیر توجه کنید :

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

  // ptr is a constant pointer to an integer that can
  // be modified through ptr, but ptr always points
  // to the same memory location.
  int * const ptr = &x;

  *ptr = 7; // allowed: *ptr is not const
  ptr = &y; // error: ptr is const;
            //        cannot assign new address

  return 0;
}
```

برنامه فوق هنگام کامپایل شدن پیغام خطایی می دهد ، مبنی بر اینکه به اشاره گر ثابت آدرس جدیدی را نمی توان نسبت داد .

```
Compiling CONSTP2.CPP:
Error CONSTP2.CPP 11: Cannot modify a const object
```

در برنامه فوق توسط دستور **int * const ptr=&x;** اشاره گر **ptr** به آدرس متغیر **x** اشاره خواهد کرد و چون از نوع ثابت تعریف شده است نمی تواند به آدرس دیگری اشاره کند اما مقداری را که ptr به آن اشاره می کند قابل تغییر است لذا با اجرای دستور **ptr=7*** خطایی رخ نمی دهد و مقدار متغیر **x** برابر ۷ می شود ولی با اجرای دستور **y&=ptr** می خواهیم اشاره گر به متغیر **y** اشاره کند و از آنجایی که **ptr** تنها باید به **x** اشاره کند با پیغام خطا مواجه می شویم .

کاربرد دیگری از **const** همراه اشاره گرها ، ایجاد اشاره گر ثابتی به داده ای ثابت می باشد . چنین اشاره گری همواره به یک خانه حافظه اشاره می کند و از طریق این اشاره گر نیز نمی توان داده داخل آن خانه حافظه را تغییر داد . به برنامه زیر توجه کنید :

```
#include <iostream.h>

int main()
{
  int x = 5, y;

  // ptr is a constant pointer to a constant integer.
  // ptr always points to the same location;
  // the integer at that location cannot be modified.
  const int *const ptr = &x;

  cout << *ptr << endl;

  *ptr = 7; // error: *ptr is const;
            //        cannot assign new value
  ptr = &y; // error: ptr is const;
            //        cannot assign new address

  return 0;
}
```

هنگام کاپایل کردن برنامه فوق با پیغام خطایی مبنی بر اینکه مقدار اشاره گر و آدرس اشاره گر را نمی توان تغییر داد ، مواجه می شویم .

```
Compiling CONSTP3.CPP:
Error CONSTP3.CPP 14: Cannot modify a const object
Error CONSTP3.CPP 16: Cannot modify a const object
```

در برنامه فوق توسط دستور **const int * const ptr = &x;** آدرس و مقدار اشاره گر **ptr** از نوع ثابت تعریف شده لذا هنگام تغییر مقدار خانه ای از حافظه که **ptr** به آن اشاره می کند توسط دستور ***ptr = 7;** با پیغام خطا مواجه می شویم و نیز هنگام تغییر آدرسی که **ptr** با آن اشاره می کند توسط دستور **ptr = &y;** پیغام خطای دیگری دریافت می کنیم .

اعمال محاسباتی با اشاره گرها

اشاره گرها عملوندهایی مجاز در عبارات محاسباتی ، و رابطه ای می باشند ، البته تمامی عملگرهایی که در اینگونه عبارات به کار می روند برای اشاره گرها مجاز نمی باشند . در این مبحث به بررسی عملگرهایی که عملوندی از نوع اشاره گر می توانند داشته باشند و نحوه کاربرد آنها می پردازیم .

یک اشاره گر می تواند افزایش (++) یا کاهش (--) یابد . یک عدد صحیح می تواند به آن اضافه (+= یا +) و یا از آن کم (-= یا -) شود .

فرض کنید که آرایه**int v[5]** تعریف شده است و اولین خانه آن در آدرس **۳۰۰۰** از حافظه قرار دارد و فرض کنید که **vptr** به خانه**v[0]** از آرایه اشاره می کند . توجه داشته باشید که برای اینکه **vptr** به آرایه **v** اشاره کند کافی است از یکی از دستورات زیر استفاده کنیم :

```
vptr = v;
vptr = &v[0];
```

در ریاضیات معمول **۲+۳۰۰۰** برابر **۳۰۰۲** می شود اما در محاسبات اشاره گرها معمولاً بدین صورت نم باشد . هنگامی که عدد صحیحی به یک اشاره گر اضافه و یا از آن کم می شود ، مقدار اشاره گر معمولاً به اندازه آن عدد زیاد یا کم نمی شود ، بلکه به اندازه طول نوع داده ای که اشاره گر به آن اشاره می کند ، زیاد یا کم می شود . به عنوان مثال دستور **vptr += 2;** حاصلش برابر **3008 (3000 + 2*4)** خواهد شد البته با فرض اینکه متغیری از نوع **int** در چهار بایت از حافظه قرار می گیرد . در آرایه **v** ، اشاره گر **vptr** حالا به خانه**v[2]** اشاره می کند .

اگر متغیری از نوع **int** در دو بایت از حافظه قرار بگیرد حاصل دستور **vptr += 2;** خانه **(3000 + 2*2)3004** می بود .

اگر **vptr** به خانه **v[4]** که در آدرس **۳۰۱۶** است اشاره کند اجرای دستور زیر :

```
vptr -= 4;
```

باعث می شود که **vptr** به خانه**3000(3016 - 4*4)** یعنی**v[0]** اشاره کند .

هر یک از دستورات زیر باعث می شود که اشاره گر به خانه بعدی آرایه **v** ، اشاره کند .

```
++vptr;
vptr++;
```

و هر یک از دستورات زیر باعث می شود که اشاره گر به خانه قبلی آرایه **v** ، اشاره کند .

```
--vptr;
vptr--;
```

اشاره گرهایی که به خانه های یک آرایه اشاره می کنند می توانند از یکدیگر کم شوند . به عنوان مثال اگر **vptr** حاوی **۳۰۰۰** (یعنی به خانه**v[0]** اشاره کند) و **v2ptr** حاوی **۳۰۰۸** (یعنی به خانه**v[2]** اشاره کند) دستور زیر :

```
x=v2ptr - vptr;
```

تعداد خانه های بین **vptr** تا **v2ptr** را در **x** قرار می دهد که در اینجا **x** حاوی **۲** می گردد . محاسبات اشاره گرها تنها هنگامی که اشاره گرها به خانه های یک آرایه اشاره می کنند ، معنا دار است . همچنین مقایسه اشاره گرها نیز هنگامی مفهوم دارد که به خانه های یک آرایه اشاره کنند به عنوان مثال مقایسه دو اشاره گر می تواند نشان دهد که یکی از اشاره گرها به خانه ای با اندیس بزرگتر نسبت به اشاره گر دیگر ، اشاره می کند .

ارتباط اشاره گرها و آرایه ها

آرایه ها و اشاره گرها در زبان **C++** ارتباط نزدیکی با یکدیگر داشته و تقریباً می توان آنها را به جای یکدیگر به کار برد نام آرایه را می توان به عنوان یک اشاره گر ثابت در نظر گرفت و تمام اعمالی که توسط اندیس آرایه می توان انجام داد توسط اشاره گر نیز قابل انجام است و از طریق اشاره گر نیز می توان به تک تک عناصر آرایه دست یافت . دستور زیر را در نظر بگیرید :

```
int b[5];
int bptr;
```

از آنجا که نام آرایه (بدون اندیس) اشاره گری به عنصر اول آرایه می باشد ، می توانیم اشاره گر **bptr** را توسط دستور زیر، به اولین عنصر آرایه **b** اشاره دهیم :

```
bptr = b;
```

و یا می توانیم از دستور زیر برای اشاره دادن **bptr** به عنصر اول آرایه **b** استفاده کنیم :

```
bptr = &b[0]
```

برای دستیابی به عنصر **b[3]** توسط اشاره گر **bptr** که توسط یکی از دستورهای فوق به آرایه **b** مرتبط شد، می توان از دستور زیر استفاده کرد :

```
*(bptr + 3)
```

در مبحث قبل با مفهوم **bptr+3** و عباراتی از این قبیل آشنا شده اید. از آنجایی که **bptr** به عنصر اول آرایه اشاره می کند پس حاوی آدرس عنصر اول آرایه یعنی **b[0]** می باشد، لذا **bptr+3** آدرس خانه **b[3]** خواهد بود پس

***(bptr + 3)** به خانه **b[3]** اشاره خواهد کرد . توجه داشته باشید که استفاده از پرانتز در اینجا اجباری می باشد ، چون عملگر * اولویت بالاتری نسبت به عملگر + دارد . اگر دستور فوق را بدون پرانتز به کار ببریم یعنی از ***bptr+3** استفاده کنیم عدد ۳ به خانه **b[0]** اضافه می گردد .

توجه داشته باشید که ***(b + 3)** (در اینجا **b** نام آرایه ای می باشد که در دستورات فوق تعریف گردید) نیز به خانه **b[3]** از آرایه اشاره می کند ، چون همانطور که در ابتدای این بحث گفته شد نام آرایه همانند یک اشاره گر ثابت می باشد .

اشاره گرها را نیز می توان مانند آرایه ها اندیس دار کرد ، به عنوان مثال **bptr[1]** به عنصر **b[1]** رجوع خواهد کرد چون **bptr** اشاره گری به آرایه**b** می باشد .

نکته : همانطور که می دانید نام آرایه ، اشاره گر ثابتی می باشد لذا دستوری مانند **b+=3;** برای آرایه ای با نام **b** قابل استفاده نمی باشد ، چون اشاره گر ثابت همواره به یک خانه از حافظه اشاره می کند .

برای درک نحوه ارتباط و شباهت آرایه ها و اشاره گرها ، به برنامه زیر که در آن عناصر آرایه ای توسط چهار روش متفاوت در خروجی چاپ می شوند ، توجه کنید :

```
#include <iostream.h>

void main()
{
  int b[] = { 10, 20, 30, 40 };
  int *bPtr = b;
  int i;

  cout << "b[i]:\n";
  for ( i = 0; i < 4; i++ )
    cout << "b[" << i << "] = " << b[ i ] << '\n';

  cout << "\n*(b + i):\n";
  for ( i = 0; i < 4; i++ )
    cout << "*(b + " << i << ") = "
         << *( b + i ) << '\n';


  cout << "\nbPtr[i]:\n";
  for ( i = 0; i < 4; i++ )
    cout << "bPtr[" << i << "] = " << bPtr[ i ] <<'\n';

  cout << "\n*(bPtr + i):\n";
  for ( i = 0; i < 4; i++ )
    cout << "*(bPtr + " << i << ") = "
         << *( bPtr + i ) << '\n';
```

```
}
```

خروجی برنامه فوق به صورت زیر می باشد :

```
b[i]:
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

*(b + i):
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40

bPtr[i]:
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

*(bPtr + i):
*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40
```

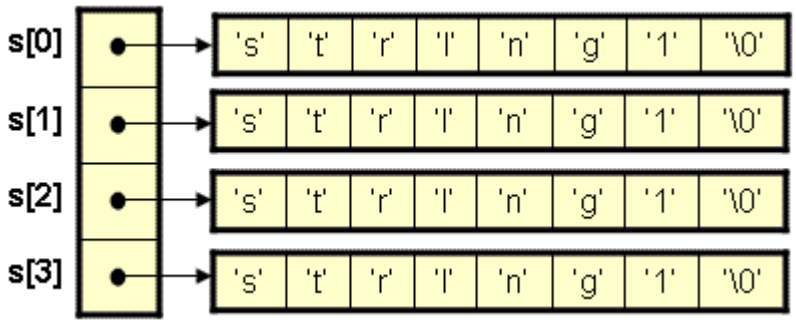
آرایه ای از اشاره گرها

عناصر یک آرایه می توانند اشاره گرها باشند . یکی از کاربردهای چنین آرایه ای ساختن آرایه ای شامل رشته هایی از حروف می باشد . هر عنصر چنین آرایه ای یک رشته از حروف می باشد که این رشته از حروف توسط اشاره گری به اولین حرف رشته مشخص می شود . به دستور زیر توجه کنید :

```
const char *s[4] =
 { "string1","string2","string3","string4" }
```

دستور فوق آرایه ای چهار عنصری شامل رشته هایی از حروف ایجاد می کند. **char*** در دستور فوق مشخص می کند که هر عنصر آرایه **s** ، اشاره گری به داده ای از نوع **char** می باشد . چهار مقداری که در آرایه قرار می گیرند **"string1"** و **"string2"** و **"string3"** و **"string4"** می باشند . انتهای هر کدام از این رشته ها که در حافظه قرار می

گیرند با کاراکتر پوچ مشخص می شود ، لذا طول این رشته ها یک واحد بیشتر از تعداد حروفی می باشد که بین " " قرار دارد . در این دستور طول هر یک از رشته ها هشت حرف می باشد . اگر چه به نظر می رسد که این رشته از حروف درآرایه قرار می گیرند ولی در واقع اشاره گرهایی به اولین حرف هر یک از این رشته ها در آرایه قرار دارد . به شکل زیر توجه کنید :



اگر چه آرایه طول ثابتی دارد ، اما این شیوه ما را قادر می سازد که به رشته هایی با طول نا مشخص دسترسی پیدا کنیم . این انعطاف پذیری مثالی از توان زبان **C++** در ایجاد ساختمان های داده ای می باشد .

نکته : توجه داشته باشید که آرایه ای از حروف را می توان توسط یک آرایه دو بعدی از نوع **char** نیز ایجاد کرد که هر سطر حاوی یک رشته و هر ستون حاوی یک حرف از یک رشته می باشد . در چنین حالتی تعداد ستون ها در هر سطر باید عددی ثابت باشد و این عدد باید برابر با طول بزرگترین رشته باشد ، لذا هنگامی که رشته های زیادی از حروف داریم و طول اکثر رشته ها از طول بزرگترین رشته کمتر می باشند ، مقدار زیادی از خانه های حافظه به هدر می رود . به شکل زیر توجه کنید :

'R'	'e'	'd'	'\0'				
'Y'	'e'	'l'	'l'	'o'	'w'	'\0'	
'H'	'o'	't'	'p'	'i'	'n'	'k'	'\0'
'G'	'r'	'a'	'y'	'\0'			

اشاره گر به تابع

یک اشاره گر به تابع حاوی آدرس آن تابع در حافظه می باشد . همانطور که می دانید نام یک آرایه در واقع آدرس اولین عنصر آرایه در حافظه می باشد. مشابهاً ، نام یک تابع ، آدرس ابتدای کدهای یک تابع ، در حافظه می باشد . اشاره گر به یک تابع می تواند به عنوان آرگومان به توابع ارسال شود ، به عنوان خروجی تابعی برگردانده شود، در آرایه قرار گیرد و یا به تابعی دیگر اشاره داده شود.

برای آشنایی با نحوه کاربرد اشاره گرهای توابع، برنامه مرتب کردن حبابی عناصر آرایه را بازنویسی می کنیم. این برنامه دارای توابع **ascending** ، **swap**، **bubble** و **descending** می باشد . تابع **bubble** یک اشاره گر تابع را به عنوان آرگومان ، همراه با آرایه و عدد ثابتی به عنوان طول آرایه ، دریافت می کند .

این اشاره گر به تابع ، اشاره گری به یکی از توابع **ascending** یا **descending** می باشد که این دو تابع نحوه مرتب شدن آرایه را از نظر صعودی یا نزولی بودن تشخیص می دهند. برنامه در ابتدا از کاربر نحوه مرتب کردن عناصر را که به صورت صعودی مرتب شوند یا نزولی ، می پرسد . اگر کاربر عدد **۱** را وارد کند ، یک اشاره گر به تابع **ascending** به عنوان آرگومان، به تابع **bubble** ارسال می شود که باعث می شود عناصر آرایه به صورت صعودی مرتب شوند. و اگر کاربر عدد **۲** را وارد کند ، یک اشاره گر به تابع **descending** به عنوان آرگومان، به تابع **bubble** ارسال می گردد که باعث می شود عناصر آرایه به صورت نزولی مرتب شوند. به کدهای این برنامه توجه کنید :

```
#include <iostream.h>

void bubble( int [], const int, int (*)( int, int ) );
void swap( int * const, int * const );
int ascending( int, int );
int descending( int, int );

void main()
{
  const int arraySize = 10;
  int order;
  int counter;
  int a[ arraySize ] = { 2, 6, 4, 8, 10,
                         12, 89, 68, 45, 37 };

  cout << "Enter 1 to sort in ascending order,\n"
       << "Enter 2 to sort in descending order: ";
  cin >> order;
  cout << "\nData items in original order\n";

  for ( counter = 0; counter < arraySize; counter++ )
    cout << " " << a[ counter ];

  if ( order == 1 ) {
    bubble( a, arraySize, ascending );
    cout << "\nData items in ascending order\n";
  }
  else {
    bubble( a, arraySize, descending );
    cout << "\nData items in descending order\n";
  }

  for ( counter = 0; counter < arraySize; counter++ )
```

```
    cout << " " << a[ counter ];

  cout << endl;
}

void bubble( int work[], const int size,
             int (*compare)( int, int ) )
{
  for ( int pass = 1; pass < size; pass++ )
    for ( int count = 0; count < size-1; count++)
      if ( (*compare)( work[count], work[count + 1]))
        swap( &work[ count ], &work[count + 1]);
}

void swap( int * const element1Ptr,
           int * const element2Ptr )
{
  int hold = *element1Ptr;
  *element1Ptr = *element2Ptr;
  *element2Ptr = hold;
}

int ascending( int a, int b )
{
  return b < a;
}

int descending( int a, int b )
{
  return b > a;
}
```

خروجی برنامه فوق به صورت زیر می باشد :

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1

Data items in original order
  2  6  4  8  10  12  89  68  45  37
Data items in ascending order
  2  4  6  8  10  12  37  45  68  89
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2

Data items in original order
```

```
  2  6  4  8  10  12  89  68  45  37
Data items in descending order
  89  68  45  37  12  10  8  6  4  2
```

همانطور که در برنامه دیدید آرگومان زیر در تابع **bubble** مورد استفاده قرار گرفت :

```
int (*compare) (int , int)
```

این دستور به تابع **bubble** می گوید که آرگومانی که دریافت می کند، یک اشاره گر به تابعی می باشد که دو آرگومان از نوع عدد صحیح دریافت می کند و خروجی آن از نوع **int** می باشد.

پرانتزهای به کار رفته در کنار **compare*** برای اینکه مشخص کنیم **compare** اشاره گری به یک تابع می باشد ، الزامیند. اگر پرانتزها را در کنار **compare*** به کار نبریم دستور زیر را خواهیم داشت :

```
int *compare (int,int)
```

که این دستور ، تابعی را تعریف می کند که دو عدد صحیح را دریافت کرده و اشاره گری به مقداری از نوع **int** را به عنوان خروجی بر می گرداند .

این آرگومان در پیش تعریف تابع **bubble** به صورت زیر می باشد :

```
int (*)(int,int)
```

توجه داشته باشید که تنها نوع داده ها مشخص شده است و نیازی به ذکر نام داده ها و آرگومانها نمی باشد .

تابع ارسال شده به **bubble** توسط دستور زیر فراخوانی می شود :

```
(*compare) (work[count],work[count+1])
```

یکی دیگر از کاربردهای اشاره گرهای تابع در انتخاب یکی از موارد یک فهرست می باشد . برنامه از کاربر می خواهد که گزینه ای را از فهرستی انتخاب کند . هر گزینه به تابعی مرتبط است که با انتخاب آن گزینه ، تابع مرتبط به گزینه انتخاب شده ، اجرا می شود . اشاره گرهای به هر تابع ، در یک آرایه قرار می گیرند . در چنین حالتی همه این توابع باید ورودی و خروجی یکسانی از نظر نوع داده داشته باشند . انتخاب کاربر به عنوان اندیسی از آرایه اشاره گرهای به توابع ، مورد استفاده قرار می گیرد و اشاره گر موجود در آرایه برای فراخوانی تابع مربوط استفاده می شود. در برنامه زیر نحوه استفاده از آرایه ای از اشاره گرهای به توابع نشان داده شده است :

```
#include <iostream.h>

void function1( int );
void function2( int );
void function3( int );

int main()
{
  void(*f[ 3 ])(int)={function1, function2, function3};

  int choice;

  cout << "Enter a number between 0 and 2, 3 to end:";
  cin >> choice;

  while ( choice >= 0 && choice < 3 ) {

   (*f[ choice ])( choice );

    cout <<"Enter a number between 0 and 2, 3 to end:";
    cin >> choice;
  }

  cout << "Program execution completed." << endl;

  return 0;
}

void function1( int a )
{
  cout << "You entered " << a
       << " so function1 was called\n\n";
}

void function2( int b )
{
  cout << "You entered " << b
       << " so function2 was called\n\n";
}

void function3( int c )
{
  cout << "You entered " << c
       << " so function3 was called\n\n";
}
```

خروجی برنامه فوق به صورت زیر می باشد :

```
Enter a number between 0 and 2, 3 to end: 2
You entered 2 so function3 was called

Enter a number between 0 and 2, 3 to end: 0
You entered 0 so function1 was called

Enter a number between 0 and 2, 3 to end: 1
You entered 1 so function2 was called

Enter a number between 0 and 2, 3 to end: 3
Program execution completed.
```

در برنامه فوق سه تابع با نامهای **function1** و **function2** و **function3** که هر یک عدد صحیحی را به عنوان ورودی دریافت می کنند و خروجی ندارند ، تعریف شده است. دستور زیر اشاره گرهای به این سه تابع را در آرایه ای با نام **f** قرار می دهد .

```
void (*f[3]) (int) = {function1,function2,function3};
```

همانطور که می بینید نوع آرایه کاملاً مثل نوع توابع، تعریف شده است ، چون این آرایه حاوی اشاره گرهایی به توابع می باشد . هنگامی که کاربر عددی را بین ۰ تا **۲** وارد می کند ، عدد وارد شده به عنوان اندیس آرایه **f** استفاده می شود . لذا دستور زیر :

```
(*f[choice]) (choice);
```

باعث اجرای یکی از توابع آرایه **f** می شود و **choice** به عنوان آرگومان به تابع ارسال می شود .

پردازش رشته ها

در این مبحث تعدادی از توابع پردازش رشته را که در فایل کتابخانه ای **string.h** قرار دارند مورد بررسی قرار می دهیم .

```
char *strcpy (char *s1, const char *s2);
```

تابع فوق رشته **s2** را در رشته **s1** کپی می کند و مقدار **s1** به عنوان خروجی تابع برگردانده می شود .

```
char *strncpy (char *s1, const char *s2, size_t n);
```

تابع فوق تعداد **n** حرف را از رشته **s2** در رشته **s1** کپی می کند و **s1** را به عنوان خروجی برمی گرداند . به برنامه زیر توجه کنید :

```
#include <iostream.h>
#include <string.h>

void main()
{
  char x[] = "Happy Birthday to You";
  char y[ 25 ];
  char z[ 15 ];

  strcpy( y, x ); // copy contents of x into y

  cout << "The string in array x is: " << x
       << "\nThe string in array y is: " << y << '\n';

  // copy first 14 characters of x into z
  strncpy( z, x, 14 ); // does not copy null character
  z[ 14 ] = '\0'; // append '\0' to z's contents

  cout << "The string in array z is: " << z << endl;
}
```

خروجی برنامه فوق به صورت زیر می باشد :

```
The string in array x is: Happy Birthday to You
The string in array y is: Happy Birthday to You
The string in array z is: Happy Birthday
```

```
char *strcat (char *s1, const char *s2);
```

تابع فوق رشته **s2** را به انتهای رشته **s1** اضافه می کند و **s1** را به عنوان خروجی بر می گرداند .

```
char *strncat (char *s1, const char *s2, size_t n);
```

تابع فوق **n** حرف از رشته **s2** را به رشته **s1** اضافه می کند و **s1** را به عنوان خروجی تابع بر می گرداند. به برنامه زیر توجه کنید :

```
#include <iostream.h>
```

```
#include <string.h>

void main()
{
  char s1[ 20 ] = "Happy ";
  char s2[] = "New Year ";
  char s3[ 40 ] = "";

  cout << "s1 = " << s1 << "\ns2 = " << s2;

  strcat( s1, s2 ); // concatenate s2 to s1

  cout << "\n\nAfter strcat(s1, s2):\ns1 = " << s1
       << "\ns2 = " << s2;

  // concatenate first 6 characters of s1 to s3
  // and places '\0' after last character
  strncat( s3, s1, 6 );


  cout << "\n\nAfter strncat(s3, s1, 6):\ns1 = " << s1
       << "\ns3 = " << s3;

  strcat( s3, s1 ); // concatenate s1 to s3
  cout << "\n\nAfter strcat(s3, s1):\ns1 = " << s1
       << "\ns3 = " << s3 << endl;
}
```

خروجی برنامه به صورت زیر می باشد :

```
s1 = Happy
s2 = New Year

After strcat(s1, s2):
s1 = Happy New Year
s2 = New Year

After strncat(s3, s1, 6):
s1 = Happy New Year
s3 = Happy

After strcat(s3, s1):
s1 = Happy New Year
s3 = Happy Happy New Year
```

```
int strcmp (const char *s1, const char *s2);
```

تابع فوق رشته **s1** را با رشته **s2** مقایسه می کند . اگر دو رشته برابر بودند مقدار صفر برگردانده می شود ، اگر رشته **s1** کوچکتر از **s2** باشد عددی منفی و اگر رشته **s1** بزرگتر از **s2** باشد عددی مثبت خروجی تابع خواهند بود .

```
int strncmp (const char *s1, const char *s2, size_t n);
```

تابع فوق **n** حرف اول رشته **s1** را با رشته **s2** مقایسه می کند و خروجی تابع همانند خروجی **strcmp** خواهد بود . به برنامه زیر توجه کنید :

```
#include <iostream.h>
#include <string.h>

void main()
{
  char *s1 = "Happy New Year";
  char *s2 = "Happy New Year";
  char *s3 = "Happy Holidays";

  cout << "s1 = " << s1 << "\ns2 = " << s2
       << "\ns3 = " << s3 << "\n\nstrcmp(s1, s2) = "
       << " " << strcmp( s1, s2 )
       << "\nstrcmp(s1, s3) = " << " "
       << strcmp( s1, s3 ) << "\nstrcmp(s3, s1) = "
       << " " << strcmp( s3, s1 );

  cout << "\n\nstrncmp(s1, s3, 6) = " << " "
       << strncmp(s1,s3,6) << "\nstrncmp(s1,s3,7) = "
       << " " << strncmp( s1, s3, 7 )
       << "\nstrncmp(s3, s1, 7) = "
       << " " << strncmp( s3, s1, 7 ) << endl;
}
```

خروجی برنامه فوق به صورت زیر می باشد :

```
s1 = Happy New Year
s2 = Happy New Year
s3 = Happy Holidays

strcmp(s1, s2) = 0
strcmp(s1, s3) = 6
```

```
strcmp(s3, s1) = -6

strncmp(s1, s3, 6) = 0
strncmp(s1, s3, 7) = 6
strncmp(s3, s1, 7) = -6
```

```
char *strtok (char *s1, const char *s2);
```

تابع فوق رشته **s1** را توسط **s2** جدا جدا می کند . به برنامه زیر توجه کنید :

```
#include <iostream.h>
#include <string.h>

void main()
{
  char sentence[] ="This is a sentence with 7 tokens";
  char *tokenPtr;

  cout << "The string to be tokenized is:\n"<<sentence
       << "\n\nThe tokens are:\n\n";

  tokenPtr = strtok( sentence, " " );

  while ( tokenPtr != NULL ) {
    cout << tokenPtr << '\n';
    tokenPtr = strtok( NULL, " " ); // get next token
  }
  cout <<"\nAfter strtok, sentence ="<<sentence<<endl;

}
```

خروجی برنامه فوق به صورت زیر می باشد :

```
The string to be tokenized is:
This is a sentence with 7 tokens

The tokens are:

This
is
a
sentence
```

```
with
7
tokens

After strtok, sentence = This
```

```
size_t strlen (const char *s);
```

تابع فوق طول رشته **s** را به عنوان خروجی بر می گرداند . به برنامه زیر توجه کنید :

```
#include <iostream.h>
#include <string.h>

void main()
{
char *string1 = "abcdefghijklmnopqrstuvwxyz";
char *string2 = "four";
char *string3 = "Boston";

cout << "The length of \"" << string1
     << "\" is " << strlen( string1 )
     << "\nThe length of \"" << string2
     << "\" is " << strlen( string2 )
     << "\nThe length of \"" << string3
     << "\" is " << strlen( string3 ) << endl;
}
```

خروجی برنامه فوق به صورت زیر می باشد :

```
The length of "abcdefghijklmnopqrstuvwxyz" is 26
The length of "four" is 4
The length of "Boston" is 6
```

تعریف ساختار

ساختارها نوعی داده ، متشکل از تعدادی عنصر(عضو) می باشند که این عناصر هر یک ممکن است از نوع داده ای متفاوتی با بقیه باشند و یا از نوع ساختار دیگری باشند . به تعریف ساختار زیر توجه کنید :

```
struct Time {
             int hour;
             int minute;
             int second;
            };
```

در دستور فوق کلمه **struct** یک ساختار را با نام **Time** ایجاد می کند ساختار ایجاد شده با نام **Time** دارای سه عنصر به نامهای **hour** (برای نگهداری ساعت) ، **minute** (برای نگهداری دقیقه) ، **second** (برای نگهداری ثانیه) می باشد که این عناصر از نوع عدد صحیح می باشند . عناصر یک ساختار نمی توانند دارای نامهای یکسانی باشند ولی دو ساخاختار متفاوت می توانند دارای عناصری با نامهای یکسان باشند و این کار تداخلی بین آنها ایجاد نمی کند.

پس از اینکه ساختار **Time** ایجاد شد می توانیم از این نوع داده جدید استفاده کنیم و متغیرهایی را از این نوع داده تعریف کنیم . به دستورات زیر توجه کنید :

```
Time timeObject;
Time timeArray[10];
Time *timePtr=&timeObject;
```

توسط دستورات فوق **timeObject** متغیری از نوع **Time** تعریف می شود . **timeArray** آرایه ای که شامل **۱۰** عنصر از نوع **Time** می باشد ، تعریف می شود . **timePtr** اشاره گری به خانه ای از حافظه با نام **timeObject** از نوع **Time** تعریف می شود.

دسترسی به عناصر ساختار

برای دستیابی به عناصر یک ساختار از عملگر نقطه (.) و یا عملگر پیکان (<-) استفاده می کنیم . عملگر نقطه برای دستیابی به عنصر یک ساختار از طریق نام یک متغیر ، مورد استفاده قرار می گیرد ، به عنوان مثال به دستور زیر توجه کنید :

```
timeObject.hour = 13;
timeObject.minute = 33;
timeObject.second = 20;
```

دستور فوق عناصر **hour** و **minute** و **second** متغیری از نوع **Time** را به ترتیب با مقادیر **۱۳** و **۳۳** و **۲۰** مقدار دهی می کند و دستور زیر :

```
cout << timeObject.hour;
```

برای چاپ کردن عنصر**hour** از متغیر**time Object** به کار می رود . عملگر پیکان که از یک علامت منفی (-) و یک علامت بزرگتر (>) بدون فاصله بین این دو علامت تشکیل شده است ، برای دستیابی به عنصر یک ساختار از طریق اشاره گر مورد استفاده قرار می گیرد .

```
cout << timePtr -> hour;
```

دستور فوق برای چاپ کردن عنصر **hour** از متغیر **timeObject** توسط اشاره گر **timePtr** به کار رفته است . به جای استفاده از **timePtr -> hour** برای دستیابی به عنصر **hour** می توان از **(*timePtr).hour** استفاده کرد . توجه داشته باشید که استفاده از پرانتز در کنار **timePtr*** الزامی می باشد . به عنوان مثال از دستور زیر نیز برای چاپ عنصر **hour** از متغیر **timeObject** توسط اشاره گر **timePtr** می توان استفاده کرد .

```
cout << (*timePtr).hour;
```

در برنامه زیر ساختار **Time** مورد استفاده قرار گرفته است و توسط روش ارسال آرگومان با ارجاع به توابع ، شیء **dinnerTime** که از نوع ساختار **Time** می باشد به توابع **printUniversal** و **printStandard** ارسال شده است.

```
#include <iostream.h>

struct Time {
             int hour; // 0-23 (24-hour clock format)
             int minute; // 0-59
             int second; // 0-59
            };

void printUniversal( const Time & );
void printStandard( const Time & );

int main()
{
  Time dinnerTime;

  dinnerTime.hour = 18;
  dinnerTime.minute = 30;
  dinnerTime.second = 0;

  cout << "Dinner will be held at ";
  printUniversal( dinnerTime );
  cout << " universal time,\nwhich is ";
  printStandard( dinnerTime );
  cout << " standard time.\n";
```

```
  dinnerTime.hour= 29; //set hour to invalid value
  dinnerTime.minute= 73; //set minute to invalid value

  cout << "\nTime with invalid values: ";
  printUniversal( dinnerTime );
  cout << endl;

  return 0;
}

void printUniversal( const Time &t )
{
  cout << (t.hour<10 ? "0":"") << t.hour << ":"
       << (t.minute<10 ? "0":"") << t.minute << ":"
       << (t.second<10 ? "0":"") << t.second;
}


void printStandard( const Time &t )
{
  cout << ( ( t.hour == 0 || t.hour == 12 ) ?
                    12 : t.hour % 12 ) << ":"
       << (t.minute<10 ? "0":"") << t.minute << ":"
       << (t.second<10 ? "0":"") << t.second
       << ( t.hour < 12 ? " AM" : " PM" );
}
```

خروجی برنامه فوق به صورت زیر می باشد:

```
Dinner will be held at 18:30:00 universal time,
which is 6:30:00 PM standard time.

Time with invalid values: 29:73:00
```

همانطور که در برنامه فوق مشاهده کردید ، امکان مقدار دهی نادرست به شیئی از نوع ساختار **Time** وجود دارد. اما در مبحث بعدی در کلاسها خواهید دید که می توان مقدار دهی به اعضا را کنترل کرد.

تعریف کلاس

توسط کلاس ها می توان اشیایی را با خصوصیات (که توسط عناصر نمایش داده می شوند) و اعمال (که توسط توابع مشخص می شوند) مدلسازی کرد . کلاسها همانند ساختارها تعریف می شوند با این تفاوت که در کلاسها توابع نیز می توانند به عنوان عنصر کلاس باشند . یک کلاس توسط کلمه **class** تعریف می شود . به تعریف کلاس زیر توجه کنید :

```
class Time {
  public:
    Time();
    void setTime( int, int, int );
    void printUniversal();
    void printStandard();

  private:
    int hour;
    int minute;
    int second;
};
```

تعریف کلاس **Time** با کلمه **class** شروع می شود و بدنه کلاس با {} مشخص می شود و در انتها تعریف کلاس با یک نقطه ویرگول (;) به پایان می رسد. همانند ساختار **Time** که در مبحث قبل ایجاد کردیم کلاس **Time** دارای سه عنصر از نوع عدد صحیح با نامهای**hour** و**minute** و**second** می باشد سایر اجزای کلاس جدید می باشد که به بررسی آنها می پردازیم . برچسبهای :**public** و :**private** معرفهای دستیابی به عنصر (عضو) نامیده می شوند . هر عضو داده ای یا تابع عضوی که بعد از **public** و قبل از **private** تعریف می شود ، در هر جایی که برنامه به شیئی از نوع **Time** دسترسی دارد قابل دستیابی می باشد. هر عضو داده ای یا تابعی که بعد از **private** تعریف می شود تنها برای توابع عضو کلاس قابل دستیابی می باشد و به طور مستقیم از داخل برنامه قابل دسترسی نمی باشند . تأکید می شود که بعد از معرفهای دستیابی به اعضاء یک علامت دو نقطه (:) قرار می گیرد . ضمناً این معرفها را در تعریف کلاسها ، چندین بار و به هر ترتیبی می توان به کار برد ولی توصیه می شود که برای خوانایی برنامه هر یک از این معرفها را تنها یکبار به کار ببرید و ابتدا اعضای عمومی (**public**) را قرار دهید . تعریف کلاس **Time** شامل پیش تعریف چهار تابع **printUniversal** , **Time**, **setTime** و **printStandard** می شود که این توابع ، اعضای عمومی کلاس **Time** می باشند ، چون بعد از **publice** تعریف شده اند .

توجه داشته باشید که تابع عضوی همنام با کلاس وجود دارد که این تابع ، تابع سازنده نامیده می شود . تابع سازنده ، تابع عضو بخصوصی است که مقادیر اولیه اعضای داده ای را تعیین می کند . هنگامی که برنامه شیئی از کلاسی را ایجاد می کند ، تابع سازنده به طور خودکار فراخوانی می گردد. توجه داشته باشید که تابع سازنده نباید مقداری را به عنوان خروجی خود برگرداند .

همانند ساختارها ، هنگامی که کلاس **Time** تعریف گردید می توان اشیایی را از نوع کلاس **Time** ایجاد کرد . به دستورات زیر توجه کنید :

```
Time sunset;
Time arrayofTime [5];
Time *timeptr = & sunset;
```

توسط دستورات فوق **sunset** متغیری از نوع کلاس **Time** تعریف می شود. **arrayofTime** آرایه ای شامل **۱۰** عنصر از نوع کلاس **Time** می باشد و **timePtr** اشاره گری به خانه ای از حافظه با نام **sunset** از نوع کلاس **Time** تعریف می گردد.

در برنامه زیر کلاس **Time** مورد استفاده قرار گرفته است و توابع عضو آن تعریف گردیده اند. در این برنامه شیئی از نوع کلاس **Time** بانام **t** ایجاد شده است. هنگامی که این شیء ایجاد می شود تابع سازنده **Time** به طور خودکار فراخوانی می شود ومقدار هر یک از اعضای داده (**secound , hour , minute**) را عدد صفر قرار می دهد. پس از ایجاد شدن شیء **t** توابع عضو **printUniversal** و **printStandard** فراخوانی می شوند تا با چاپ زمان توسط این دو تابع از مقدار دهی اولیه به شیء **t** اطمینان حاصل کنیم. سپس با فراخوانی تابع **setTime** ، شیء **t** مقدار دهی می شود و سپس مجددا مقدار این شیء به دو روش چاپ می گردد. هنگامی که با فراخوانی مجدد تابع عضو **setTime** سعی در مقدار دهی نادرست به شیء **t** می کنیم ، تابع عضو **setTime** از این کار جلوگیری می کند.

```
#include <iostream.h>

class Time {
  public:
    Time();
    void setTime( int, int, int );
    void printUniversal();
    void printStandard();

  private:
    int hour; // 0 - 23 (24-hour clock format)
    int minute; // 0 - 59
    int second; // 0 - 59
}

Time::Time()
{
  hour = minute = second = 0;
}

void Time::setTime( int h, int m, int s )
{
  hour = ( h >= 0 && h < 24 ) ? h : 0;
  minute = ( m >= 0 && m < 60 ) ? m : 0;
  second = ( s >= 0 && s < 60 ) ? s : 0;
}

void Time::printUniversal()
{
  cout << (hour<10 ? "0":"") << hour << ":"
       << (minute<10 ? "0":"") << minute << ":"
```

```
       << (second<10 ? "0":"") << second;
}


void Time::printStandard()
{
  cout << ( ( hour == 0 || hour == 12 ) ?
             12 : hour % 12 ) << ":"
       << (minute<10 ? "0":"") << minute << ":"
       << (second<10 ? "0":"") << second
       << (hour < 12 ? " AM" : " PM" );
}

int main()
{
  Time t;
  cout << "The initial universal time is ";
  t.printUniversal(); // 00:00:00

  cout << "\nThe initial standard time is ";
  t.printStandard(); // 12:00:00 AM

  t.setTime( 13, 27, 6 ); // change time

  cout << "\n\nUniversal time after setTime is ";
  t.printUniversal(); // 13:27:06

  cout << "\nStandard time after setTime is ";
  t.printStandard(); // 1:27:06 PM

  t.setTime( 99, 99, 99 ); // attempt invalid settings

  cout << "\n\nAfter attempting invalid settings:"
       << "\nUniversal time: ";
  t.printUniversal(); // 00:00:00

  cout << "\nStandard time: ";
  t.printStandard(); // 12:00:00 AM
  cout << endl;

  return 0;
}
```

خروجی برنامه فوق به صورت زیر می باشد:

```
The initial universal time is 00:00:00
The initial standard time is 12:00:00 AM

Universal time after setTime is 13:27:06
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM
```

همانطور که در برنامه فوق مشاهده می کنید هنگام تعریف کلاس تنها پیش تعریف توابع عضو آورده شده است و خود توابع در خارج از کلاس تعریف شده اند. برای این کار قبل از نام تابع ، نام کلاس را همراه با عملگر تفکیک دامنه(::) قرار می دهیم :

```
void Time::printUniversal()
{
  cout << (hour<10 ? "0":"") << hour << ":"
       << (minute<10 ? "0":"") << minute << ":"
       << (second<10 ? "0":"") << second;
}
```

مبحث بعدی

حوزه کلاس و دسترسی به اعضای کلاس

اعضای داده یک کلاس (متغیرهایی که در کلاس تعریف شده اند) و توابع عضو کلاس (توابعی که در کلاس تعریف گردیده ا ند) به حوزه این کلاس متعلق می باشند و توابعی که عضو کلاس نمی باشند، دارای حوزه فایل می باشند.

در حوزه یک کلاس ، همه اعضای کلاس توسط توابع عضو کلاس قابل دسترسی می باشند و می توان توسط نامشان مستقیما به آنها مراجعه کرد. در بیرون از حوزه کلاس برای دسترسی به اعضای کلاس از عملگر نقطه (.) و یا عملگر پیکان (<-) استفاده می کنیم. عملگر هایی که برای دسترسی به اعضای کلاس مورد استفاده قرار می گیرند همانند عملگر های دسترسی به عناصر ساختارها می باشند. عملگر نقطه (.) برای دستیابی به عضو یک کلاس از طریق نام یک متغیر و عملگر پیکان (<-) ، برای دستیابی به عضو یک کلاس از طریق اشاره گر مورد استفاده قرار می گیرد .

توابع عضو یک کلاس تنها توسط سایر توابع عضو همان کلاس می توانند گرانبار شوند. برای گرانبار کردن یک تابع عضو کافی است هنگام تعریف کلاس برای هر یک از نسخه های تابع عضوی که می خواهیم آنرا گرانبارکنیم، یک پیش تعریف قرار دهیم و برای هر یک از نسخه های این تابع تعریف جداگانه ای در نظر بگیریم.

متغیرهایی که در یک تابع عضو کلاس تعریف شده اند، دارای حوزه تابع می باشند، یعنی فقط برای همان تابع شناخته شده اند. اگر در تابع عضوی، متغیری همنام با نام متغیری که دارای حوزه کلاس است تعریف شده باشد، در این صورت متغیری

که دارای حوزه کلاس می باشد در داخل تابع به طور مستقیم نمی توان رجوع کرد و برای دسترسی به آن باید عملگر تفکیک حوزه (::) را قبل از نام متغیر قرار داد.

در برنامه زیر یک کلاس ساده به نام **Count** با یک عضو داده عمومی از نوع **int** به نام **x** و یک تابع عضو عمومی به نام **print** ، تعریف شده است. در این برنامه دو شیء از نوع کلاس **Count** به نامهای **counter** و **countPtr** (اشاره گری به یک شیء **Count**) تعریف شده اند که متغیر**countPtr** به **counter** اشاره داده شده است. لازم به ذکر است که عضو داده **x** فقط برای این به صورت عمومی تعریف شده که نحوه دسترسی به اعضای عمومی را نشان دهیم. همان طور که قبلا ذکر کردیم، اعضای داده ای معمولاً به صورت خصوصی اعلان می شوند . به برنامه زیر توجه کنید :

```
#include <iostream.h>

class Count {

  public:
     int x;

     void print()
     {
      cout << x << endl;
     }
};

void main()
{
  Count counter;
  Count *counterPtr = &counter;

  cout << "Assign 1 to x and print "
       << "using the object's name: ";
  counter.x = 1;
  counter.print();

  cout << "Assign 3 to x and print using a pointer: ";
  counterPtr->x = 3;
  counterPtr->print();
}
```

خروجی برنامه فوق به صورت زیر می باشد:

```
Assign 1 to x and print using the object's name: 1
Assign 3 to x and print using a pointer: 3
```

کنترل دسترسی به اعضا

معرفهای دسترسی به اعضا یعنی **public** و **private** ، نحوه دسترسی به اعضای داده و توابع عضو کلاس را کنترل می کنند. حالت پیش فرض برای دسترسی به اعضای داده و توابع عضو در کلاس **private** می باشد، لذا همه اعضایی که بعد از تعیین نام کلاس و قبل از اولین برچسب معرف آمده اند، خصوصی در نظر گرفته می شوند. پس از هر معرف، حالتی که معرف مربوطه تعیین می کند، به همه اعضا تا برچسب بعدی یا آکلاد بسته مربوط به پایان تعریف کلاس اعمال می شود. برچسبهای **public** و **private** را می توان چندین بار در تعریف کلاس به کار برد، اما این کار معمول نمی باشد و باعث پیچیدگی کد برنامه می شود. اعضای خصوصی یک کلاس فقط از طریق توابع عضو آن کلاس قابل دسترسی می باشند در مقابل اعضای عمومی کلاس توسط هر تابعی در برنامه قابل دسترسی می باشند.

برنامه زیر نشان می دهد که اعضای خصوصی کلاس، خارج از کلاس قابل دسترسی نمی باشند :

```
#include <iostream.h>

class Time {
  public:
    Time();
    void setTime( int, int, int );
    void printUniversal();
    void printStandard();

  private:
    int hour; // 0 - 23 (24-hour clock format)
    int minute; // 0 - 59
    int second; // 0 - 59
}

Time::Time()
{
  hour = minute = second = 0;
}

void Time::setTime( int h, int m, int s )
{
  hour = ( h >= 0 && h < 24 ) ? h : 0;
  minute = ( m >= 0 && m < 60 ) ? m : 0;
  second = ( s >= 0 && s < 60 ) ? s : 0;
}

void Time::printUniversal()
{
  cout << (hour<10 ? "0":"") << hour << ":"
```

```
       << (minute<10 ? "0":"") << minute << ":"
       << (second<10 ? "0":"") << second;
}


void Time::printStandard()
{
  cout << ( ( hour == 0 || hour == 12 ) ?
                  12 : hour % 12 ) << ":"
       << (minute<10 ? "0":"") << minute << ":"
       << (second<10 ? "0":"") << second
       << (hour < 12 ? " AM" : " PM" );
}

int main()
{
  Time t; // create Time object

  t.hour = 7; // error: 'Time::hour' is not accessible

  // error: 'Time::minute' is not accessible
  cout << "minute = " << t.minute;

  return 0;
}
```

هنگام کامپایل کردن برنامه فوق با پیغام خطایی مبنی بر اینکه عضو خصوصی قابل دسترسی نمی باشد، مواجه می شویم .

```
Error in line 50: 'Time::hour' is not accessible
Error in line 53: 'Time::minute' is not accessible
```

سازنده ها

هنگامی که شیئی از یک کلاس ساخته می شود، اعضای داده آن را می توان با تابع سازنده آن کلاس مقدار دهی اولیه کرد. تابع سازنده ، تابع عضو بخصوصی است که مقادیر اولیه اعضای داده ای را تعیین می کند. هنگامی که برنامه شیئی از کلاسی را ایجاد می کند ، تابع سازنده به طور خودکار فراخوانی می گردد. توجه داشته باشید که تابع سازنده نباید مقداری را به عنوان خروجی خود برگرداند.

سازنده به کار رفته در برنامه مبحث تعریف کلاس مقادیر اولیه **minute** ، **hour** و **second** را برابر با صفر قرار داد. اما توابع سازنده می توانند دارای آرگومان باشند، که معمولا این آرگومان ها برای مقدار دهی اولیه به شیئی از نوع کلاس مربوطه به کار می روند. از آنجا که هنگام ایجاد شیء، تابع سازنده به طور خودکار فراخوانی می شود، اعضای داده ای کلاس را توسط آرگومان های در یافتی خود، می تواند مقدار دهی اولیه کند. مقادیر اولیه یک کلاس را می توان هنگام تعریف شیئی از آن کلاس، درون پرانتزهایی که در سمت راست نام شیء و پیش از نقطه ویرگول می آیند قرار داد. این مقادیر اولیه به عنوان آرگومان به تابع سازنده کلاس ارسال می گردند. به برنامه زیر توجه کنید:

```
#include <iostream.h>

class CRectangle {
    int width, height;
  public:
    CRectangle (int,int);
    int area (void) {return (width*height);}
};

CRectangle::CRectangle (int a, int b) {
  width = a;
  height = b;
}

int main () {
  CRectangle rect (3,4);
  CRectangle rectb (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
}
```

خروجی برنامه فوق به صورت زیر می باشد:

```
rect area: 12
rectb area: 30
```

سازنده ها را برای اینکه مقدار دهی اولیه به اعضای داده ای کلاس به صورتهای مختلف امکان پذیر باشد، می توان گرانبار کرد. به نحوه گرانبار کردن تابع سازنده، در برنامه زیر توجه نمایید:

```
#include <iostream.h>

class CRectangle {
    int width, height;
  public:
    CRectangle ();
```

```
    CRectangle (int,int);
    int area (void) {return (width*height);}
};

CRectangle::CRectangle () {
  width = 5;
  height = 5;
}

CRectangle::CRectangle (int a, int b) {
  width = a;
  height = b;
}

int main () {
  CRectangle rect (3,4);
  CRectangle rectb;
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
}
```

خروجی برنامه فوق به صورت زیر می باشد:

```
rect area: 12
rectb area: 25
```

سازنده ها می توانند دارای آرگومان های پیش فرض نیز باشند. در برنامه زیر در پیش تعریف تابع سازنده **time** و برای هر یک از آرگومانها مقدار پیش فرض صفر در نظر گرفته شده است. با مشخص کردن مقدار پیش فرض برای آرگومانهای تابع سازنده، حتی اگر هنگام فراخوانی تابع سازنده مقداری نیز برای آن تعیین نگردد، مقادیر پیش فرض به کار گرفته می شود. این کار تضمین می نماید که اعضاء کلاس همواره مقادیر درستی را در خود نگهداری می کنند.

```
#include <iostream.h>

class Time {
  public:
    Time( int = 0, int = 0, int = 0);
    void setTime( int, int, int );
    void printUniversal();
    void printStandard();

  private:
    int hour; // 0 - 23 (24-hour clock format)
    int minute; // 0 - 59
```

```
    int second; // 0 - 59
}

Time::Time( int hr, int min, int sec )
{
  setTime( hr, min, sec );
}

void Time::setTime( int h, int m, int s )
{
  hour = ( h >= 0 && h < 24 ) ? h : 0;
  minute = ( m >= 0 && m < 60 ) ? m : 0;
  second = ( s >= 0 && s < 60 ) ? s : 0;
}

void Time::printUniversal()
{
  cout << (hour<10 ? "0":"") << hour << ":"
       << (minute<10 ? "0":"") << minute << ":"
       << (second<10 ? "0":"") << second;
}


void Time::printStandard()
{
  cout << ( ( hour == 0 || hour == 12 ) ?
             12 : hour % 12 ) << ":"
       << (minute<10 ? "0":"") << minute << ":"
       << (second<10 ? "0":"") << second
       << (hour < 12 ? " AM" : " PM" );
}


void main()
{
  Time t1;                // all arguments defaulted
  Time t2( 2 );           // minute and second defaulted
  Time t3( 21, 34 );      // second defaulted
  Time t4( 12, 25, 42 ); // all values specified
  Time t5( 27, 74, 99 ); // all bad values specified

  cout << "Constructed with:\n\n"
       << "all default arguments:\n ";
  t1.printUniversal(); // 00:00:00
  cout << "\n ";
  t1.printStandard(); // 12:00:00 AM
```

```
  cout << "\n\nhour specified;";
       << " default minute and second:\n ";
  t2.printUniversal(); // 02:00:00
  cout << "\n ";
  t2.printStandard(); // 2:00:00 AM

  cout << "\n\nhour and minute specified;";
       << " default second:\n ";
  t3.printUniversal(); // 21:34:00
  cout << "\n ";
  t3.printStandard(); // 9:34:00 PM

  cout << "\n\nhour, minute, and second specified:\n ";
  t4.printUniversal(); // 12:25:42
  cout << "\n ";
  t4.printStandard(); // 12:25:42 PM

  cout << "\n\nall invalid values specified:\n ";
  t5.printUniversal(); // 00:00:00
  cout << "\n ";
  t5.printStandard(); // 12:00:00 AM
  cout << endl;

}
```

خروجی برنامه فوق به صورت زیر می باشد:

```
Constructed with:

all default arguments:
  00:00:00
  12:00:00 AM

hour specified; default minute and second:
  02:00:00
  2:00:00 AM

hour and minute specified; default second:
  21:34:00
  9:34:00 PM

hour, minute, and second specified:
  12:25:42
  12:25:42 PM
```

```
all invalid values specified:
  00:00:00
  12:00:00 AM
```

در برنامه فوق پنج شیء از کلاس **Time** ایجاد شده اند که **t1** هنگام فراخوانی سازنده از هر آرگومان پیش فرض استفاده کرده است، **t2** یک آرگومان، **t3** دو آرگومان، **t4** سه آرگومان را برای سازنده خود مشخص کرده اند و **t5** آرگومانهایی با مقادیر غیر مجاز را مشخص کرده است. ضمنا برای بررسی مجاز بودن آرگومان های دریافت شده توسط تابع سازنده، این تابع ، **setTime** را با آرگومانهایی که دریافت کرده است فراخوانی می کند تا اطمینان حاصل شود که **hour** و **minute** و **second** با مقادیر درستی مقدار دهی می شوند.

نابودکننده ها

نابودکننده همانند سازنده تابع عضو ویژه ای از کلاس و همنام با کلاس می باشد و با کاراکتر (~) شروع می گردد.

تابع نابودکننده یک کلاس هنگامی فراخوانده می شود که شیئی از آن کلاس نابود شود (مثلا وقتی که برنامه در حال اجرا، حوزه ای را که در آن، شیئی از آن کلاس نمونه سازی شده ترک می کند). خود نابودکننده شی را نابود نمی کند، بلکه امور خانه داری مربوط به پایان دهی را پیش از آن که سیستم، حافظه شی را باز پس گیرد و این حافظه برای اشیای بعدی آزاد شود، انجام می دهد.

نابودکننده هیچ پارامتری نمی گیرد و هیچ مقداری باز نمی گرداند. یک کلاس فقط یک نابودکننده می تواند داشته باشد. به عبارت دیگر گرانبار کردن نابودکننده مجاز نیست. در قطعه برنامه زیر نحوه تعریف یک نابود کننده را می بینید:

```
class CreateAndDestroy {

public:
  CreateAndDestroy();
  ~CreateAndDestroy();

private:
  int objectID;
};

CreateAndDestroy::CreateAndDestroy(int objectNumber)
{
  objectID = objectNumber;
  cout << "Object "<<objectID<<" constructor runs\n";
}


CreateAndDestroy::~CreateAndDestroy()
```

```
{
  cout << "Object "<<objectID<<" destructor runs\n";
}
```

فراخوانی سازنده ها و نابودکننده ها

سازنده ها و نابودکننده ها به طور خودکار فراخوانی می شوند. ترتیب فراخوانی این توابع بستگی به ترتیب ورود و ترک حوزه ای که اشیا در آن تعریف شده اند، دارد. به برنامه زیر توجه کنید:

```
#include <iostream.h>

class CreateAndDestroy {
  public:
    CreateAndDestroy( int, char * );
    ~CreateAndDestroy();

  private:
    int objectID;
    char *message;
};

CreateAndDestroy::CreateAndDestroy(
            int objectNumber, char *messagePtr )
{
  objectID = objectNumber;
  message = messagePtr;

  cout << "Object " << objectID << " constructor runs "
       << message << endl;
}

CreateAndDestroy::~CreateAndDestroy()
{
  cout << ( objectID == 1 || objectID == 6 ? "\n" : "" );

  cout << "Object " << objectID << " destructor runs "
       << message << endl;
}

void create( void );

CreateAndDestroy first(1,"(global before main)");
```

```
void main()
{
  cout << "\nMAIN FUNCTION: EXECUTION BEGINS" << endl;

  CreateAndDestroy second(2,"(local automatic in main)");

  create(); // call function to create objects

  cout << "\nMAIN FUNCTION: EXECUTION RESUMES" << endl;

  CreateAndDestroy third(3,"(local automatic in main)");

  cout << "\nMAIN FUNCTION: EXECUTION ENDS" << endl;
}

void create( void )
{
  cout << "\nCREATE FUNCTION: EXECUTION BEGINS" << endl;

  CreateAndDestroy fourth(4,"(local automatic in create)");

  cout << "\nCREATE FUNCTION: EXECUTION ENDS" << endl;
}
```

خروجی برنامه فوق به صورت زیر می باشد:

```
Object 1   constructor runs (global before main)

MAIN FUNCTION: EXECUTION BEGINS
Object 2   constructor runs (local automatic in main)

CREATE FUNCTION: EXECUTION BEGINS
Object 4   constructor runs (local automatic in create)

CREATE FUNCTION: EXECUTION ENDS
Object 4   destructor runs  (local automatic in create)

MAIN FUNCTION: EXECUTION RESUMES
Object 3   constructor runs (local automatic in main)

MAIN FUNCTION: EXECUTION ENDS
Object 3   destructor runs  (local automatic in main)
Object 2   destructor runs  (local automatic in main)

Object 1   destructor runs  (global before main)
```

اشیاء ثابت و توابع عضو ثابت

همانطور که می دانید کلمه **const** از تغییر بر روی متغیری که بعد از این کلمه قرار می گیرد جلوگیری می کند. برای ایجاد یک شیء ثابت و غیر قابل تغییرنیز می توان از **const** استفاده کرد.

```
const Time noon(12, 0, 0);
```

دستور فوق شیء ثابتی از نوع **Time** با نام **noon** ایجاد می کند و با **۱۲** ظهر آنرا مقدار دهی می کند که این مقدار قابل تغییر نمی باشد.

کامپایلر **C++** اجازه فراخوانی توابع عضو را برای اشیاء ثابت نمی دهد مگر اینکه توابع عضو نیز خودشان به صورت **const** تعریف شده باشند. برای اینکه تابع عضوی به صورت **const** تعریف شود، باید هم در پیش تعریف تابع و هم در تعریف تابع از کلمه **const** بعد از لیست آرگومان های تابع استفاده کنیم.

```
int Time::getHour() const
{
   return hour;
}
```

دستور فوق تابع عضو **getHour** از کلاس **Time** را به صورت ثابت تعریف می کند.

نکته:

۱. تابع عضوی که یک عضو داده از شیئی را تغییر می دهد، اگر به صورت ثابت تعریف شود، باعث وقوع خطا می گردد.
۲. تابع عضوی که تابع عضو غیر ثابت دیگری را فراخوانی می کند، اگر به صورت ثابت تعریف شود، باعث وقوع خطا می گردد.
۳. فراخوانی یک تابع عضو غیر ثابت از شیئی که به صورت ثابت تعریف شده است، باعث ایجاد خطا می گردد.
۴. یک تابع عضو غیر ثابت را می توان با نسخه غیر ثابتی از آن گرانبار نمود. اینکه کدام تابع عضو گرانبار شده فراخوانی شود، توسط کامپایلر و بر اساس ثابت بودن یا نبودن شیء تعیین می گردد.
۵. توجه داشته باشید از آنجا که سازنده ها و نابود کننده ها نیاز به تغییر اشیاء دارند نباید آنها را به صورت ثابت تعریف نمود و اگر کلمه **const** برای آنها به کار رود و به صورت ثابت تعریفشان کنیم یک پیغام خطا دریافت خواهیم کرد.
۶. سازنده ها در اشیاء ثابت می توانند تابع عضو غیر ثابتی را فراخوانی کنند.

در برنامه زیر دو شیء نوع **Time** ایجاد شده اند که یکی ثابت و دیگری غیر ثابت می باشد. در این برنامه تابع عضو **setHour** و تابع عضو **printStandard** که به صورت توابع عضو غیر ثابت می باشند، هنگام فراخوانی همانطور که در **نکته ۲** ذکر شد باعث وقوع خطا می گردند. برنامه زیر نکات گفته شده در این مبحث را روشن می سازد:

```
#include <iostream.h>

class Time {
  public:
    Time( int = 0, int = 0, int = 0);
    void setTime( int, int, int );

    void setHour( int );
    void setMinute( int );
    void setSecond( int );

    int getHour() const;
    int getMinute() const;
    int getSecond() const;

    void printUniversal() const;
    void printStandard();

  private:
    int hour; // 0 - 23 (24-hour clock format)
    int minute; // 0 - 59
    int second; // 0 - 59
}

Time::Time( int hour, int minute, int second )
{
  setTime( hour, minute, second );
}

void Time::setTime( int hour, int minute, int second )
{
  setHour( hour );
  setMinute( minute );
  setSecond( second );
}

void Time::setHour( int h )
{
  hour = ( h >= 0 && h < 24 ) ? h : 0;
}

void Time::setMinute( int m )
```

```
{
  minute = ( m >= 0 && m < 60 ) ? m : 0;
}

void Time::setSecond( int s )
{
  second = ( s >= 0 && s < 60 ) ? s : 0;
}

int Time::getHour() const
{
  return hour;
}

int Time::getMinute() const
{
  return minute;
}

int Time::getSecond() const
{
  return second;
}

void Time::printUniversal() const
{
cout << (hour<10 ? "0":"") << hour << ":"
     << (minute<10 ? "0":"") << minute << ":"
     << (second<10 ? "0":"") << second;
}

void Time::printStandard()
{
  cout << ( ( hour == 0 || hour == 12 ) ?
             12 : hour % 12 ) << ":"
       << (minute<10 ? "0":"") << minute << ":"
       << (second<10 ? "0":"") << second
       << (hour < 12 ? " AM" : " PM" );
}

int main()
{
  Time wakeUp( 6, 45, 0 );      // non-constant object
  const Time noon( 12, 0, 0 );  // constant object

                            // OBJECT     MEMBER FUNCTION
  wakeUp.setHour( 18 );     // non-const  non-const
```

```
  noon.setHour( 12 );      // const      non-const

  wakeUp.getHour();        // non-const  const

  noon.getMinute();        // const      const
  noon.printUniversal();   // const      const

  noon.printStandard();    // const      non-const

  return 0;
}
```

پیغام های خطای برنامه فوق به صورت زیر می باشد:

```
Warning W8037 100: Non-const function
Time::setHour(int)
called for const object in function main()
Warning W8037 107: Non-const function
Time::printStandard()
called for const object in function main()
*** 2 errors in Compile ***
```

ترکیب : اشیاء به عنوان اعضای کلاس ها

شیء **AlarmClock** (زنگ ساعت) باید بداند که چه وقت به صدا در آید، پس بهتر است شیئی از نوع **Time** را به عنوان یکی از اعضای خود داشته باشد. چنین قابلیتی در کلاس ها ترکیب نامیده می شود. یعنی یک کلاس می تواند کلاس یا کلاسهای دیگری را به عنوان یکی از اعضای خود داشته باشد.

با روش ارسال آرگومانها به تابع سازنده هنگام ایجاد شیء آشنا می باشیم. در این مبحث روش ارسال آرگومان به تابع سازنده شیء عضو کلاس و مقدار دهی اولیه به آن را بررسی می کنیم. برای آشنایی با این روش به برنامه زیر توجه کنید:

```
#include <iostream.h>
#include <string.h>

class Date {

  public:
    Date( int = 1, int = 1, int = 1900 );
```

```
    void print() const;
    ~Date();

  private:
    int month; // 1-12 (January-December)
    int day; // 1-31 based on month
    int year; // any year

    int checkDay( int ) const;
};

Date::Date( int mn, int dy, int yr )
{
  if ( mn > 0 && mn <= 12 )
    month = mn;
  else {
    month = 1;
    cout << "Month " << mn
         << " invalid. Set to month 1.\n";
  }

  year = yr;
  day = checkDay( dy );

  cout << "Date object constructor for date ";
  print();
  cout << endl;
}


void Date::print() const
{
  cout << month << '/' << day << '/' << year;
}

Date::~Date()
{
  cout << "Date object destructor for date ";
  print();
  cout << endl;
}

int Date::checkDay( int testDay ) const
{
  static const int daysPerMonth[ 13 ] =
    { 0,31,28,31,30,31,30,31,31,30,31,30,31 };
```

```
if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
  return testDay;

if ( month == 2 && testDay == 29 &&
     ( year % 400 == 0 ||
       ( year % 4 == 0 && year % 100 != 0 ) ) )
  return testDay;

  cout << "Day " << testDay
       << " invalid. Set to day 1.\n";

  return 1;
}

class Employee {

  public:
    Employee(const char *, const char *,
              const Date &, const Date & );

    void print() const;
    ~Employee();

  private:
    char firstName[ 25 ];
    char lastName[ 25 ];
    const Date birthDate; //composition: member object
    const Date hireDate;  //composition: member object
};


Employee::Employee(const char *first, const char *last,
      const Date &dateOfBirth, const Date &dateOfHire )
     : birthDate( dateOfBirth ),
       hireDate( dateOfHire )
{
  int length = strlen( first );
  length = ( length < 25 ? length : 24 );
  strncpy( firstName, first, length );
  firstName[ length ] = '\0';

  length = strlen( last );
  length = ( length < 25 ? length : 24 );
  strncpy( lastName, last, length );
  lastName[ length ] = '\0';
```

```
  cout << "Employee object constructor: "
       << firstName << ' ' << lastName << endl;
}

void Employee::print() const
{
  cout << lastName << ", " << firstName << "\nHired: ";
  hireDate.print();
  cout << " Birth date: ";
  birthDate.print();
  cout << endl;
}

Employee::~Employee()
{
  cout << "Employee object destructor: "
       << lastName << ", " << firstName << endl;
}


int main()
{
  Date birth( 7, 24, 1949 );
  Date hire( 3, 12, 1988 );
  Employee manager( "Bob", "Jones", birth, hire );

  cout << '\n';
  manager.print();

  cout << "\nTest Date constructor "
       <<"with invalid values:\n";
  Date lastDayOff(14,35,1994); //invalid month and day
  cout << endl;

  return 0;
}
```

خروجی برنامه فوق به صورت زیر می باشد:

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Employee object constructor: Bob Jones

Jones, Bob
Hired: 3/12/1988 Birth date: 7/24/1949
```

```
Test Date constructor with invalid values:
Month 14 invalid. Set to month 1.
Day 35 invalid. Set to day 1.
Date object constructor for date 1/1/1994

Date object destructor for date 1/1/1994
Employee object destructor: Jones, Bob
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
```

در برنامه فوق دو کلاس **Date** و **Employee** ایجاد شده است. کلاس **Employee** دارای چهار عضو داده خصوصی به نامهای **firstName** و **lastName** و **birthDate** و **hireDate** می باشد. اعضای **birthDate** و **hireDate** خود اشیاء ثابتی از نوع کلاس **Date** می باشند که دارای اعضای داده خصوصی **month** و **day** و **year** هستند. تعریف تابع سازنده کلاس **Employee** مشخص می کند که سازنده چهار آرگومان **fname** و **lname** و **dateOfBirth** و **dateOfHire** را دریافت می کند. دو آرگومان اول اعضای **firstName** و **lastName** را مقدار دهی می کنند. دو آرگومان بعدی یعنی **dateOfBirth** و **dateOfHire** توسط **birthDate** و **hireDate** که اشیایی از نوع کلاس **Date** می باشند به سازنده های این اشیاء ارسال می گردند و این دو شیء را مقدار دهی اولیه می کنند.

```
Employee::Employee(const char *first, const char *last,
      const Date &dateOfBirth, const Date &dateOfHire )
     : birthDate( dateOfBirth ),
       hireDate( dateOfHire )
```

همانطور که در دستور فوق می بینید بعد از لیست آرگومانها علامت (:) قرار گرفته است و بعد از آن نام دو شیء عضو کلاس **Employee** که از نوع کلاس **Date** می باشند، قرار دارد. این روش برای مقدار دهی اولیه به اشیاء عضو یک کلاس از طریق آرگومانهای کلاس اصلی، مورد استفاده قرار می گیرد.

نکته: تابع عضو **print** از کلاس **Date** ، در تابع سازنده این کلاس بدون هیچ آرگومانی فراخوانی شده است. این کار در **C++** بسیار مرسوم است. در اینجا در تابع **print** مشخص شده است که کدام یک از اعضای کلاس باید چاپ شود، و نیازی به دریافت آرگومان ندارد.

توابع دوست و کلاس های دوست

تابع دوست یک کلاس در خارج از حوزه آن کلاس تعریف می شود اما همچنان به اعضای غیر عمومی آن کلاس دسترسی دارد. توابع تنها (توابعی که عضو کلاسی نمی باشند) و یا توابع عضو یک کلاس می توانند به عنوان دوست کلاس دیگری تعریف گردند. برای تعریف یک تابع به عنوان دوست یک کلاس، پیش تعریف تابع را در آن کلاس به همراه کلمه **friend** می آوریم.

```
class Count{
    friend void setX(Count & , int);
    int x;
}
```

در دستور فوق تابع **setX** به عنوان دوست کلاس **Count** در نظر گرفته می شود. برای اینکه تمام توابع عضو کلاسی با نام **classTwo** به عنوان توابع دوست کلاس دیگری با نام **classOne** در نظر گرفته شوند. دستور زیر را در تعریف کلاس **classOne** می آوریم:

```
friend class classTwo;
```

در برنامه زیر تابع **setX** به عنوان دوست کلاس **Count** در نظر گرفته شده است لذا اجازه دستیابی به عضو داده **x** از کلاس **Count** را دارا می باشد. ضمنا بهتر است که توابع دوست در ابتدای تعریف اعضای کلاس ، تعریف گردند.

```
#include <iostream.h>

class Count {
  friend void setX( Count &, int );

  public:
    Count(): x( 0 ) // initialize x to 0
    {
      // empty body
    }

    void print() const
    {
      cout << x << endl;
    }

  private:
    int x;
};

void setX( Count &c, int val )
{
  c.x = val; // legal: setX is a friend of Count
```

```
}

void main()
{
  Count counter;

  cout << "counter.x after instantiation: ";
  counter.print();

  setX( counter, 8 ); // set x with a friend

  cout << "counter.x after call"
       <<"to setX friend function: ";
  counter.print();
}
```

خروجی برنامه فوق به صورت زیر می باشد:

```
counter.x after instantiation: 0
counter.x after call to setX friend function: 8
```

در برنامه فوق توابع عضو کلاس **Count** در داخل کلاس تعریف شده اند. همچنین عضو داده خصوصی **x** ، توسط دستور زیر با عدد ۰ مقدار دهی اولیه شده است.

```
Count(): x( 0 )
```

اشاره گر this

در **C++** کلمه کلیدی this همراه با کلاس ها وجود دارد. هر شیء از طریق اشاره گری به نام **this** به آدرس خود دسترسی دارد. از این اشاره گر می توان برای بررسی اینکه آیا آرگومان ارسال شده به تابع عضو یک شیء، خود شیء می باشد یا خیر ، استفاده کرد. به برنامه زیر توجه نمایید:

```
#include <iostream.h>

class CDummy {
  public:
    int isitme (CDummy& param);
};

int CDummy::isitme (CDummy& param)
{
```

```
  if (&param == this) return 1;
  else return 0;
}

int main () {
  CDummy a;
  CDummy* b = &a;
  if ( b->isitme(a) )
    cout << "yes, &a is b";
  return 0;
}
```

خروجی برنامه فوق به صورت زیر می باشد:

```
yes, &a is b
```

مدیریت حافظه پویا

C++ برنامه نویسان را قادر می سازد تا تخصیص حافظه و یا آزادسازی حافظه را برای هر نوع داده ای در برنامه مدیریت و کنترل کنند. این قابلیت، مدیریت حافظه پویا نامیده می شود و توسط عملگرهای **new** و **delete** صورت می پذیرد. دو عملگر مذکور در فایل **new.h** قرار دارند. لذا برای به کارگیری این دو عملگر باید از دستور **#include <new.h>** در برنامه استفاده کنیم. دستورات زیر را در نظر بگیرید:

```
Time *timePtr;
timePtr = new Time;
```

عملگر **new** در دستورات فوق شیئی را با اندازه داده ای از نوع **Time** می سازد و سازنده این شیء را فراخوانی کرده و یک اشاره گر از نوع داده قرار گرفته در سمت راست عملگر **new** برمی گرداند. توجه داشته باشید که عملگر **new** برای هر نوع داده ای (مانند **double** و **int** و ...) و یا کلاسها می تواند به کار رود. اگر **new** نتواند فضای خالی برای شیء در حافظه بیابد، یک اشاره گر• را باز می گرداند. برای نابودسازی شیئی که به صورت پویا برای آن حافظه تخصیص داده شده، از عملگر **delete** به صورت زیر استفاده می کنیم:

```
delete timePtr;
```

دستور فوق ابتدا نابودکننده شیء را که **timePtr** به آن اشاره می کند، فراخوانی کرده؛ سپس حافظه اختصاص داده شده به شیء را آزاد می سازد و حافظه برای تخصیص به شیء دیگری آماده می شود.

C++ اجازه مقداردهی اولیه را همزمان با تخصیص حافظه پویا به یک شیء می دهد. به دستور زیر توجه کنید:

```
double *ptr=new double(3.14159);
```

دستور فوق حافظه تخصیص داده شده به داده ای از نوع **double** را با ۳,۱۴۱۵۹ مقداردهی می کندو اشاره گر به آن را در **Ptr** قرار می دهد. روش مشابه دستور فوق می تواند برای آرگومانهای تابع سازنده یک شیء به کار رود. به دستور زیر توجه کنید:

```
Time *timePtr= new Time(12,0,0);
```

دستور فوق حافظه تخصیص داده شده به شیئی از نوع کلاس **Time** را با ۱۲ ظهر مقداردهی کرده و اشاره گر با آن را در **timePtr** قرار می دهد.

عملگر **new** امکان تخصیص حافظه پویا برای آرایه ها را نیز فراهم می سازد. به عنوان مثال برای یک آرایه ۱۰ عنصری از نوع عدد صحیح توسط دستور زیر حافظه پویا تخصیص می یابد:

```
int *gradesArray=new int[10];
```

در دستور فوق اشاره گر به اولین عنصر در حافظه پویای تخصیص یافته به آرایه ۱۰ عنصری از نوع عدد صحیح در اشاره گر **gradesArray** قرار می گیرد.

برای آزادسازی حافظه تخصیص داده شده به آرایه ها از دستور زیر می توان استفاده کرد:

```
delete [] gradesArray;
```

نکته: توجه داشته باشید که برای آزادسازی خانه های حافظه تخصیص یافته به یک آرایه از [] **delete** استفاده نمایید و **delete** را به تنهایی به کار نبرید.

اعضای ایستای کلاس

همان طور که می دانید متغیری که در یک تابع از نوع **static** تعریف می شد، هنگام خروج از تابع از بین نمی رفت و مقدار خود را حفظ می کرد و با فراخوانی مجدد تابع قابل دسترسی بود. در کلاسها نیز می توان اعضا را به صورت ایستا (**static**) تعریف کرد. چنین عضوی که به صورت ایستا تعریف می شود، برای همه اشیایی که از نوع آن کلاس تعریف می شوند، به صورت مشترک قابل دسترسی است و هنگامی که مقدار این عضو در یکی از این اشیا تغییر می کند مقدار جدید در سایر اشیا از نوع آن کلاس نیز قابل استفاده می باشد. چنین عضوی ممکن است مانند متغیر عمومی به نظر آید، اما این عضو دارای حوزه کلاس می باشد و مانند متغیر عمومی دارای حوزه فایل نیست. اعضای ایستای یک کلاس می توانند از نوع **public** ویا **private** باشند. همچنین اعضای ایستا از طریق توابع عضو کلاس و یا توابع دوست یک کلاس قابل

دسترسی می باشند. ضمنا به اعضای ایستایی که به صورت عمومی تعریف شده اند، می توان مستقیما از طریق عملگر (::) دسترسی یافت.

برای آشنایی با نحوه به کار گیری اعضای ایستای کلاس و نیز مدیریت حافظه پویا به برنامه زیر توجه کنید:

```
#include <iostream.h>
#include <new.h>
#include <string.h>

class Employee {
  public:
    Employee( const char *, const char * );
    ~Employee();
    const char *getFirstName() const;
    const char *getLastName() const;

    // static member function
    static int getCount();

  private:
    char *firstName;
    char *lastName;

    // static data member
    static int count;
};

int Employee::count = 0;

int Employee::getCount()
{
  return count;
}

// constructor dynamically allocates space for
// first and last name and uses strcpy to copy
// first and last names into the object
Employee::Employee(const char *first,const char *last)
{
  firstName = new char[ strlen( first ) + 1 ];
  strcpy( firstName, first );

  lastName = new char[ strlen( last ) + 1 ];
  strcpy( lastName, last );

  ++count; // increment static count of employees
```

```
  cout << "Employee constructor for " << firstName
       << ' ' << lastName << " called." << endl;
}

// destructor deallocates dynamically allocated memory
Employee::~Employee()
{
  cout << "~Employee() called for " << firstName
       << ' ' << lastName << endl;

  delete [] firstName; // recapture memory
  delete [] lastName; // recapture memory

  --count; // decrement static count of employees
}

const char *Employee::getFirstName() const
{
  return firstName;
}

const char *Employee::getLastName() const
{
  return lastName;
}

int main()
{
  cout <<"Number of employees before instantiation is "
       <<Employee::getCount() << endl;

  Employee *e1Ptr = new Employee( "Susan", "Baker" );
  Employee *e2Ptr = new Employee( "Robert", "Jones" );

  cout << "Number of employees after instantiation is "
       << e1Ptr->getCount();

  cout << "\n\nEmployee 1: "
       << e1Ptr->getFirstName()
       << " " << e1Ptr->getLastName()
       << "\nEmployee 2: "
       << e2Ptr->getFirstName()
       << " " << e2Ptr->getLastName() << "\n\n";

  delete e1Ptr; //recapture memory
  e1Ptr = 0; //disconnect pointer from free-store space
```

```
  delete e2Ptr; //recapture memory
  e2Ptr = 0; //disconnect pointer from free-store space

  cout << "Number of employees after deletion is "
       << Employee::getCount() << endl;

  return 0;
}
```

خروجی برنامه فوق به صورت زیر می باشد:

```
Number of employees before instantiation is 0
Employee constructor for Susan Baker called.
Employee constructor for Robert Jones called.
Number of employees after instantiation is 2

Employee 1: Susan Baker
Employee 2: Robert Jones

~Employee() called for Susan Baker
~Employee() called for Robert Jones
Number of employees after deletion is 0
```

در برنامه بالا عضو داده خصوصی **count** و تابع عضو عمومی **getCount** به صورت ایستا (**static**) در کلاس **Employee** تعریف شدند، و توسط دستور زیر:

```
int Employee::count = 0;
```

عضو داده خصوصی **count** از کلاس **Employee** مقداردهی اولیه شد. **count** تعداد اشیای ایجاد شده از کلاس **Employee** را می شمارد و در خود نگهداری می کند. هر بار که یک شی جدید از نوع کلاس **Employee** ایجاد می شود توسط ستور **count**++ در سازنده کلاس **count** ، **Employee** مقدارش یک واحد افزایش می یابد و هر بار که یک شی از نوع کلاس **Employee** نابود می شود، توسط دستور **count**-- در نابود کننده کلاس **count** ، **Employee** مقدارش یک واحد کاهش می یابد.

دستورات زیر دو شی از نوع **Employee** را توسط عملگر **new** ایجاد می کنند و اشاره گر به حافظه تخصیص یافتهبه این دو شی به ترتیب در **e1Ptr** و **e2Ptr** قرار می گیرد:

```
  Employee *e1Ptr = new Employee( "Susan", "Baker" );
  Employee *e2Ptr = new Employee( "Robert", "Jones" );
```

همچین دستورات زیر حافظه تخصیص یافته به دو شی از نوع کلاس **Employee** را آزاد می سازند:

```
  delete e1Ptr; //recapture memory
  e1Ptr = 0; //disconnect pointer from free-store space
  delete e2Ptr; //recapture memory
  e2Ptr = 0; //disconnect pointer from free-store space
```

توجه داشته باشید، هنگامی که هنوز شیئی از نوع کلاس **Employee** ساخته نشده است ولی اعضای **getCount** و **count** از این کلاس موجود می باشد و می توان به مقدار **count** از طریق تابع عضو **getCount** دسترسی پیدا کرد در برنامه فوق دستور زیر این کار را انجام می دهد:

```
Employee::getCount()
```

مبانی گرانبار کردن عملگرها

در مباحث قبلی دیدید که برای کار با اشیای یک کلاس، شیوه فراخوانی توابع عضو را به کار بردیم. به عنوان مثال توسط یک تابع عضو، اعضای یک کلاس را مقداردهی کردیم و یا توسط تابع عضو دیگری مقدار عضو یک کلاس را نمایش دادیم. این کار برای برخی از کلاسها، خصوصا کلاسهای ریاضی دشوار می باشد. برای این کار می توان از مجموعه عملگرهای زبان **C++** استفاده کرد، البته برای به کار بردن عملگرهای زبان **C++** برای کلاسهایی که ما می سازیم، این عملگرها باید آماده سازی شوند. به این کار گرانبار کردن عملگرها گفته می شود. به عنوان مثال عملگرهای + یا - در زبان **C++** گرانبار شده اند. این عملگرها با توجه به نوع عملوندهای خود که اعداد صحیح یا اعشاری می باشند، به شیوه متفاوتی عمل می کنند.

گرچه **C++** اجازه ایجاد عملگر جدیدی را به ما نمی دهد، ولی برنامه نویسان را قادر می سازد تا اکثر عملگرهای این زبان را برای جایی که می خواهند آنها را به کار ببرند، گرانبار کنند. در حالتی که عملگری گرانبار شده باشد، با توجه به جایی که این عملگر به کار رفته، کامپایلر کد مرتبط به آن را به صورت خودکار تولید می کند. ضمنا کاری که توسط عملگرها انجام می شود، توابع نیز می توانند انجام دهند. ولی معمولا عملگرها وضوح برنامه را بیشتر می کنند.

برای گرانبار کردن یک عملگر کافی است یک تابع برای آن تعریف کنید ، البته نام تابع باید متشکل از کلمه **operator** و نماد عملگر مورد نظر باشد. به عنوان مثال نام تابعی که عملگر + را گرانبار می کند باید **operator+** باشد. در برنامه زیر عملگر + را برای محاسبه مجموع دو نقطه مختصات گرانبار کرده ایم:

```
#include <iostream.h>

class CVector
{
  public:
    CVector (int =0 ,int =0 );
    CVector operator+ (CVector);
    void showCVector(void);
```

```
  private:
    int x,y;
};

CVector::CVector (int a, int b)
{
  x = a;
  y = b;
}

CVector CVector::operator+ (CVector vector)
{
  CVector temp;
  temp.x = x + vector.x;
  temp.y = y + vector.y;
  return temp;
}

void CVector::showCVector(void)
{
  cout << "("<< x << "," << y << ")";
}

int main ()
{
  CVector a (3,1);
  CVector b (1,2);
  CVector c;
  c = a + b;
  a.showCVector();
  cout << "+";
  b.showCVector();
  cout << "=";
  c.showCVector();
  return 0;
}
```

خروجی برنامه به صورت زیر می باشد :

```
(3,1)+(1,2)=(4,3)
```

محدودیت های گرانبار کردن عملگر ها

اکثر عملگر های C++ را می توان گرانبار کرد. جدول این عملگرها به صورت زیر می باشد:

->		( )	[ ]
*	&	--	++
!	~	-	+
<<	%	/	->*
<=	>	<	>>
^	!=	==	>=
	||	&&	|
%=	/=	*=	=
>>=	<<=	-=	+=
,	|=	^=	&=
delete[]	new[]	delete	new

عملگرهایی که قابلیت گرانبار شدن را ندارند، در جدول زیر ذکر شده اند:

sizeof	?:	::	.*	.

توابع عملگر به عنوان اعضای کلاس یا توابع دوست

توابع عملگر می توانند توابع عضو یا توابع غیر عضو باشند، که البته توابع غیر عضو معمولا به صورت توابع دوست می باشند. هنگام فراخوانی یک تابع عملگر غیر عضو ، عملوند ها باید به عنوان آرگومان به تابع فرستاده شوند.

هنگام گرانبار کردن عملگر های () ، [] و <- یا هر عملگر انتساب دیگری، تابع گرانبار کننده عملگر باید به عنوان یک تابع عضو تعریف شود ولی برای سایر عملگرها، تابع گرانبار کننده عملگر می تواند به صورت تابع غیر عضو تعریف شود.

هنگامی که یک تابع عملگر به عنوان یک تابع عضو تعریف می شود، عملوند سمت چپ باید شیئی از کلاس عملگر باشدو اگر لازم است که عملوند سمت چپ شیئی از کلاس دیگری باشد، این تابع عملگر باید از نوع تابع غیر عضو تعریف شود. تابع عملگر غیر عضو در صورتی که باید به یک عضو خصوصی (**private**) کلاس دسترسی داشته باشد، باید از نوع توابع دوست (**friend**) تعریف گردد.

تابع عملگر عضو کلاس تنها هنگامی فراخوانی می شود که عملوند سمت چپ عملگر دودویی (دو عملوندی) شیئی از نوع کلاس تابع عملگر باشد و یا تنها عملوند عملگر یگانی (تک عملوندی) شیئی از نوع کلاس عملگر باشد.

دلیل دیگری که ممکن است یک تابع غیر عضو برای گرانبار کردن یک عملگر به کار رود اینست که بخواهیم عملگر خاصیت جابجایی داشته باشد. به عنوان مثال فرض کنید متغیری به نام **number** از نوع **long int** داریم و شیئی با نام

bigInteger1 از نوع کلاس **HugeInteger** (کلاسی که در آن اعداد صحیح می توانند مقادیر خیلی بزرگتری نسبت به مقادیرقابل ایجاد توسط انواعی مانند **int long** و **double** داشته باشند). عملگر جمع (+) هنگام جمع یک عدد **HugeInteger** با یک عدد **long int** (مثلا در عبارت **bigInteger1+number**) و یا جمع یک عدد **long int** با یک عدد **HugeInteger** (مثلا در عبارت **number+bigInteger1**) ، شیئی موقت از نوع کلاس **HugeInteger** ایجاد می کند، لذا این عملگر باید خاصیت جابجایی داشته باشد (که معمولا عملگر جمع در عبارات محاسباتی دارای خاصیت جابجایی می باشد). در اینجا اگر عملگر جمع به صورت تابع عضو تعریف گردد، در سمت چپ آن تنها باید شیئی از کلاس **HugeInteger** قرار گیرد، لذا امکان ایجاد خاصیت جابجایی در این حالت میسر نمی باشد، به همین دلیل این عملگر را به صورت تابع غیر عضو گرانبار می کنیم تا بتوانیم خاصطت جا بجایی را به آن بدهیم.

در زیر برنامه ای آورده شده است که عمل جمع و تفریق را برای نقاط صفحه مختصات انجام می دهد. در این برنامه عملگر تفریق به صورت تابع غیر عضو دوست تعریف شده است و عملگر جمع به صورت تابع عضو تعریف شده است:

```
#include <iostream.h>

class CVector
{
  friend CVector operator- (CVector & , CVector);

  public:
    CVector (int =0 ,int =0 );
    CVector operator+ (CVector);
    void showCVector(void);
  private:
    int x,y;
};

CVector::CVector (int a, int b)
{
  x = a;
  y = b;
}

void CVector::showCVector(void)
{
  cout << "("<< x << "," << y << ")";
}

CVector CVector::operator+ (CVector vector)
{
  CVector temp;
  temp.x = x + vector.x;
  temp.y = y + vector.y;
  return temp;
```

```
}

CVector operator- (CVector &v, CVector vector)
{
  CVector temp;
  temp.x = v.x - vector.x;
  temp.y = v.y - vector.y;
  return temp;
}
int main ()
{
  CVector a (3,1);
  CVector b (1,2);
  CVector c,d;
  c = a + b;
  a.showCVector();
  cout << "+";
  b.showCVector();
  cout << "=";
  c.showCVector();

  cout << "\n";

  d = a - b;
  a.showCVector();
  cout << "-";
  b.showCVector();
  cout << "=";
  d.showCVector();
  return 0;
}
```

خروجی برنامه به صورت زیر می باشد :

```
(3,1)+(1,2)=(4,3)
(3,1)-(1,2)=(2,-1)
```

گرانبار کردن عملگر های >> و <<

C++ قادر به دریافت داده های تعریف شده در این زبان و ارسال آنها به خروجی می باشد و برای اینکار از عملگر های >> و << استفاده می کند. این عملگر ها برای انواع داده ای موجود در این زبان گرانبار شده اند. ما نیز می توانیم این عملگر ها را برای انواع داده و کلاسهایی که خودمان ایجاد کرده ایم، گرانبار کنیم. در برنامه زیر کلاسی با نام **phoneNumber** که

شماره تلفن را در خود نگهداری می کند ایجاد شده است. در این برنامه توسط عملگر << یک شماره تلفن از ورودی دریافت می شود و توسط عملگر >> این شماره در صفحه نمایش چاپ می گردد. به این برنامه توجه کنید:

```
#include <iostream.h>

class PhoneNumber {
  friend ostream &operator<<
        ( ostream&, const PhoneNumber & );
  friend istream &operator>>
        ( istream&, PhoneNumber & );

  private:
    char areaCode[ 4 ];  // 3-digit area code and null
    char exchange[ 4 ];  // 3-digit exchange and null
    char line[ 5 ];      // 4-digit line and null
};

ostream &operator<<
    ( ostream &output, const PhoneNumber &num )
{
  output << num.areaCode << " "
         << num.exchange << " " << num.line;

  return output;   // enables cout << a << b << c;
}

istream &operator>>
    ( istream &input, PhoneNumber &num )
{
  input >> num.areaCode;  // input area code
  input >> num.exchange;  // input exchange
  input >> num.line;      // input line

  return input; // enables cin >> a >> b >> c;
}

int main()
{
  PhoneNumber phone;

  cout << "Enter phone number in the form "
       << "123 456 7890:\n";

  cin >> phone;

  cout << "The phone number entered was: ";
```

```
  cout << phone << endl;

  return 0;
}
```

خروجی برنامه به صورت زیر می باشد :

```
Enter phone number in the form 123 456 7890:
021 224 5348
The phone number entered was: 021 224 5348
```

همانطور که در برنامه فوق می بینید توابع عملگرهای >> و << از نوع توابع دوست تعریف شده اند. چون عملگر >> دارای یک عملوند سمت چپ از نوع **ostream &** می باشد مانند **cout** در دستور **classObject>>cout** و نیز عملگر << دارای یک عملوند سمت چپ از نوع**istream &** می باشد مانند **cin** در دستور **classObject<<cin**. همچنین این توابع گرانبار شده باید به اعضای داده شیئی که باید از ورودی دریافت و در خروجی چاپ شود، دسترسی داشته باشند. به دلایل ذکر شده این توابع از نوع توابع عملگر دوست تعریف شده اند.

تابع عملگر **operator>>** یک آرگومان از نوع **istream** با نام **input** و آرگومان دیگری از نوع کلاس **phoneNumber** با نام **num** در یافت می کند و خروجی تابع از نوع **istream** می باشد. این تابع شماره تلفن هایی به صورت

```
800 555 1212
```

را از ورودی دریافت کرده و آنها را در شیئی از نوع کلاس **phoneNumber** قرار می دهد. هنگامی که کامپایلر دستور زیر را می بیند:

```
cin>>phone;
```

تابع **operator>>** به صورت زیر تولید می شود:

```
operator>>(cin,phone);
```

هنگامی که تابع فوق تولید شد، آرگومان ارجاعی **input** نام مستعار **cin** و آرگومان ارجاعی **num** نام مستعاری برای **phone** در نظر گرفته می شوند. بدین ترتیب سه رشته دریافت شده از ورودی در اعضای **areacode** ، **exchange** و **line** قرار می گیرند.

تابع عملگر **operator<<** یک آرگومان از نوع **ostream** با نام **output** و آرگومان دیگری از نوع کلاس **phoneNumber** با نام **num** در یافت می کند و خروجی تابع از نوع **ostream** می باشد. این تابع شماره تلفن هایی به صورت

```
800 555 1212
```

را که شیئی از نوع کلاس **phoneNumber** می باشد، نمایش می دهد. هنگامی که کامپایلر دستور زیر را می بیند:

```
cout<<phone;
```

تابع **operator<<** به صورت زیر تولید می شود:

```
operator<<(cout,phone);
```

هنگامی که تابع فوق تولید شد، آرگومان ارجاعی **output** نام مستعار **cout** و آرگومان ارجاعی **num** نام مستعاری برای **phone** در نظر گرفته می شوند. بدین ترتیب سه رشته موجود در اعضای **areacode** ، **exchange** و **line** به شیوه مورد نظر نمایش می یابند.

گرانبار کردن عملگر های یگانی

به طور کلی برای گرانبار کردن یک عملگر یگانی به صورت تابع عضو ، به شیوه زیر آن را تعریف می کنیم:

```
class نام کلاس{
  public:
    نوع خروجی عملگر  oprerator علامت عملگر() const;
  ...
}
```

و برای گرانبار کردن یک عملگر یگانی به صورت تابع دوست ، به شیوه زیر آن را تعریف می کنیم:

```
class نام کلاس{
    friend نوع خروجی عملگر  oprerator علامت عملگر
                              (const نام کلاس &);
  ...
}
```

گرانبار کردن عملگر های دودویی

به طور کلی برای گرانبار کردن یک عملگر دودویی به صورت تابع عضو ، به شیوه زیر آن را تعریف می کنیم:

```
class نام کلاس{
  public:
    const نام کلاس &oprerator علامت عملگر
                    (const نام کلاس &);
  ...
}
```

و برای گرانبار کردن یک عملگر دودویی به صورت تابع دوست ، به شیوه زیر آن را تعریف می کنیم:

```
class نام کلاس{
  friend const نام کلاس &oprerator علامت عملگر
                        ( نام کلاس &, const نام کلاس &);
  ...
}
```

تبدیل انواع به یکدیگر

هنگام برنامه نویسی مواردی پیش می آید که نیاز به تبدیل انواع به یکدیگر داریم. مثلا هنگامی که اشیایی از کلاسهای متفاوت تعریف شده توسط کاربر را با یکدیگر ترکیب شوند کامپایلر نمی داند که چگونه تبدیلات لازم را انجام دهد و برنامه نویس باید شیوه تبدیل را مشخص نماید. تبدیل انواع به یکدیگر را می توان توسط سازنده تبدیل انجام داد. این تابع که تک آرگومانی می باشد، می تواند اشیایی از یک کلاس را به اشیایی از کلاس دیگر تبدیل کند. در واقع هر سازنده تک آرگومانی می تواند به عنوان سازنده تبدیل در نظر گرفته شود.

عملگر تبدیل نیز می تواند شیئی از کلاسی را به شیئی از کلاسی دیگر یا انواع اولیه تبدیل نماید. این عملگر باید به صورت تابع عضو تعریف شود. به دستور زیر توجه نمایید:

```
  test::operator char* () const;
```

دستور فوق یک یک تابع تبدیل را گرانبار می نماید. این تابع یک شیء موقت از نوع **char*** را به شیئی از نوع ایجاد شده توسط کاربر با نام **test** تبدیل می نماید. همانطور که مشاهده می کنید برای این تابع نوع خروجی در نظر گرفته نشده است. در حقیقت عملگر تبدیل نباید نوع خروجی داشته باشد و نوع خروجی ، نوع شیئی می باشد که در حال تبدیل است. مثلا اگر **s** شیئی از کلاس **test** در دستور بالا باشد هنگامی که کامپایلر به عبارت **(char *) s** برخورد می کند، فراخوانی **s.operator char *()** ایجاد می گردد.

```
myClass::operator int() const;
myClass::operator otherClass() const;
```

دو دستور فوق به ترتیب عملگر تبدیلی برای تبدیل شیئی از کلاس **myClass** به یک عدد صحیح و دومی عملگر تبدیلی برای تبدیل شیئی از کلاس **myClass** به شیئی از کلاس دیگری با نام **otherClass** تعریف می کنند.

گرانبار کردن عملگر های ++ و --

در **C++** می توان عمگر های ++ و -- را گرانبار کرد. به طور کلی برای گرانبار کردن این عملگر ها به عنوان تابع عضو به صورت زیر عمل می کنیم:

```
نوع داده operator++()                //++x
{
   ...
}

نوع داده operator++(int x)           //x++
{
   ...
}

نوع داده operator--()               //--x
{
   ...
}

نوع داده operator--(int x)          //x--
{
   ...
}
```

و برای گرانبار کردن عملگرهای فوق توسط توابع دوست ، به شیوه زیر آنها را تعریف می کنیم:

```
friend نوع داده operator++(نوع داده &op)           //++x
{
   ...
}

friend نوع داده operator++(نوع داده &op, int x)    //x++
{
```

```
  ...
}

friend نوع داده operator--(نوع داده &op)          //--x
{
  ...
}

friend نوع داده operator--(نوع داده &op, int x)   //x--
{
  ...
}
```