

C++

به زبان ساده

فهرست مطالب

8	C++ چیست
9	ویژوال استودیو
10	دانلود و نصب ویژوال استودیو
19	قانونی کردن ویژوال استودیو
22	به ویژوال استودیو خوش آمدید
25	ساخت یک برنامه ساده
33	توضیحات
34	کاراکترهای کنترلی
37	متغیر
38	انواع ساده
39	استفاده از متغیرها
43	ثابت
44	عبارات و عملگرها
45	عملگرهای ریاضی
49	عملگرهای تخصیصی
50	عملگرهای مقایسه‌ای
52	عملگرهای منطقی
54	عملگرهای بیتی
59	تقدم عملگرها
61	گرفتن ورودی از کاربر
62	ساختارهای تصمیم
63	دستور if
67	دستور if...else
68	عملگر شرطی
69	دستور if چندگانه
71	دستور if تو در تو
73	استفاده از عملگرهای منطقی

75.....	دستور Switch
79.....	تکرار
80.....	حلقه While
81.....	حلقه do while
83.....	حلقه for
84.....	حلقه‌های تو در تو (Nested Loops)
86.....	خارج شدن از حلقه با استفاده از break و continue
87.....	آرایه‌ها
90.....	آرایه‌های چند بعدی
95.....	متد
97.....	مقدار برگشتی از یک متد
100	پارامترها و آرگومان‌ها
103	ارسال آرگومان‌ها به روش ارجاع
104	ارسال آرایه به عنوان آرگومان
106	محدوده متغیر
107	پارامترهای اختیاری
108	سربارگذاری متدها
109	بازگشت (Recursion)
111	شمارش (Enumeration)
114	اشاره گر (Pointer)
120	مراجع (References)
121	تبدیل ضمنی
122	تبدیل صریح
125	برنامه نویسی شیء گرا (Object Oriented Programming)
126	کلاس
128	سازنده‌ها (Constructors)
132	مخرب‌ها (Destructors)
133	سطح دسترسی
134	کپسوله کردن (Encapsulation)

135	خواص (Property).....
141	فضای نام (Namespace).....
144	وراثت
148	سطح دسترسی Protect
149	اعضای استاتیک
151	کلاس استاتیک.....
152	ترکیب (Composition).....
154	متدهای مجازی.....
156	کلاس تو در تو (Nested Class).....
157	تابع دوست (Friend Function).....
158	Downcasting و Upcasting
162	چند ریختی (polymorphism).....
165	رابط (interface).....
170	ساختار (Struct).....
173	ایجاد آرایه‌ای از کلاسها
174	Template
175	متدهای عمومی.....
178	سربارگذاری متدهای عمومی.....
178	کلاس‌های عمومی.....
180	سربارگذاری عملگرها (Operator Overloading).....
195	مدیریت استثناءها و خطایابی
197	دستورات try و catch
200	راه‌اندازی مجدد استثناء

برای دریافت فایل‌ها و آپدیت‌های جدید این کتاب به سایت www.w3-farsi.com مراجعه فرمایید.

راه‌های ارتباط با نویسنده

وب سایت: www.w3-farsi.com

لینک تلگرام: https://telegram.me/ebrahimi_younes

ID تلگرام: @ebrahimi_younes

پست الکترونیکی: younes.ebrahimi.1391@gmail.com

تقديم به:

همسر و پسر عزيزم



مبانی زبان سی پلاس پلاس

C++ چیست

C++ یک زبان برنامه نویسی شیءگراست که در سال ۱۹۸۵ توسط Bjarne Stroustrup دانشمند دانمارکی به وجود آمد. C++ نسخه توسعه یافته زبان C می باشد و بیشتر کدهای زبان C به راحتی می تواند در C++ کامپایل شود. در C++ از ویژگی های مهمی که به C اضافه شده است می توان به برنامه نویسی شیءگرا، سربارگذاری عملگرها، وراثت چندگانه و مدیریت خطاها اشاره نمود. توسعه C++ در سال ۱۹۷۹ آغاز شد و ۷ سال پس از زبان C به نمایش گذاشته شد. با وجود قدیمی بودن زبان های C و C++، هنوز هم به صورت گسترده ای در نرم افزارهای صنعتی مورد استفاده قرار می گیرد. این زبان ها برای ساخت هر چیزی از سیستم عامل گرفته تا نرم افزارهای توکار، برنامه های دسکتاپ و بازی ها مورد استفاده قرار می گیرد.

در مقایسه با زبان های جدیدتر، برنامه های نوشته شده با C++ اغلب پیچیده تر می باشند و زمان بیشتری برای توسعه نیاز دارد. در عوض، C++ زبانی است که به شما اجازه می دهد که هم به صورت High-level (نزدیک به زبان انسان) و هم به صورت low-level (نزدیک به زبان ماشین) سخت افزار را تحت کنترل خود قرار دهید. همچنین با پشتیبانی از سبک های مختلف برنامه نویسی از جمله رویه ای، شیءگرا یا عمومی، دست برنامه نویس را در انتخاب سبک مورد نظرش آزاد می گذارد. اکنون ۵ نسخه از استاندارد این زبان منتشر شده است؛ و استاندارد C++17 نیز برای انتشار در سال ۲۰۱۷ برنامه ریزی شده است.

نام غیر رسمی	استاندارد C++	سال
C++98	ISO/IEC 14882:1998	1998
C++03	ISO/IEC 14882:2003	2003
C++07/TR1	ISO/IEC TR 19768:2007	2007
C++11	ISO/IEC 14882:2011	2011
C++14	ISO/IEC 14882:2014	2014
C++17	هنوز تعیین نشده.	2017

برای اجرای کدهای C++ نیاز به یک کامپایلر داریم. کامپایلرها و محیط های برنامه نویسی (IDE) گوناگونی برای زبان C++ وجود دارند از بین معروف ترین آن ها می توان موارد زیر اشاره نمود:

- Turbo C
- Turbo C++
- Borland C++
- Microsoft visual Studio

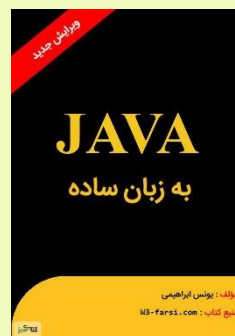
زبان C++ وابسته به یک سیستم عامل نیست یعنی شما بعد از نوشتن برنامه خود به زبان C++، اگر کد استاندارد نوشته باشید می‌توانید با توجه به سیستم عامل، کدتان را کامپایل کنید. می‌توان کد C++ را در هر محیطی، مثلاً NotePad در ویندوز و یا gEdit در گنو/لینوکس نوشته و بعد آن را بوسیله یک کامپایلر کامپایل کنیم، ولی برای راحتی کار ما می‌توانیم از یک IDE مناسب، نیز بهره ببریم. البته در این سری آموزشی ما از بهترین IDE برای کامپایل کدها استفاده می‌کنیم.



برای خرید کتاب بالا به یکی از دو لینک زیر مراجعه فرمایید

<https://goo.gl/MrdTL8>

<http://www.w3-farsi.com/product>



برای خرید کتاب بالا به یکی از دو لینک زیر مراجعه فرمایید

<https://goo.gl/MrdTL8>

<http://www.w3-farsi.com/product>

ویژوال استودیو

ویژوال استودیو محیط توسعه یکپارچه‌ای است، که دارای ابزارهایی برای کمک به شما برای توسعه برنامه‌های C++ می‌باشد. شما می‌توانید یک برنامه C++ را با استفاده از برنامه notepad یا هر برنامه ویرایشگر متن دیگر بنویسید و با استفاده از کامپایلر C++ از آن استفاده کنید، اما این کار بسیار سخت است چون اگر برنامه شما دارای خطا باشد خطایابی آن سخت می‌شود. توصیه می‌کنیم که از محیط ویژوال استودیو برای ساخت برنامه استفاده کنید چون این محیط دارای ویژگی‌های زیادی برای کمک به شما جهت توسعه برنامه‌های C++ می‌باشد. تعداد زیادی از پردازش‌ها که وقت شما را هدر می‌دهند به صورت خودکار توسط ویژوال استودیو انجام می‌شوند.

یکی از این ویژگی‌ها اینتلی سنس (Intellisense) است که شما را در تایپ سریع کدهایتان کمک می‌کند. ویژوال استودیو برنامه شما را خطایابی می‌کند و حتی خطاهای کوچک (مانند بزرگ یا کوچک نوشتن حروف) را برطرف می‌کند، همچنین دارای ابزارهای طراحی برای ساخت یک رابط گرافیکی است که بدون ویژوال استودیو برای ساخت همچنین رابط گرافیکی باید کدهای زیادی نوشت. با این برنامه‌های قدرتمند بازدهی شما افزایش می‌یابد و در وقت شما با وجود این ویژگیهای شگفت انگیز صرفه‌جویی می‌شود.

در حال حاضر آخرین نسخه ویژوال استودیو Visual Studio 2017 است. این نسخه به دو نسخه Visual Studio Professional (ارزان قیمت) و Visual Studio Enterprise (گرانقیمت) تقسیم می‌شود و دارای ویژگی‌های متفاوتی هستند. خبر خوب برای توسعه‌دهندگان نرم‌افزار این است که مایکروسافت تصمیم دارد که ویژوال استودیو را به صورت متن باز ارائه دهد. یکی از نسخه‌های ویژوال استودیو، Visual Studio Community می‌باشد که آزاد است و می‌توان آن را دانلود و از آن استفاده کرد. این برنامه ویژگی‌های کافی را برای شروع برنامه‌نویسی C++ در اختیار شما قرار می‌دهد. این نسخه (Community) کامل نیست و خلاصه‌شده نسخه اصلی است. به هر حال استفاده از Visual Studio Community که جایگزین Visual Studio Express شده و به نوعی همان نسخه Visual Studio Professional است، برای انجام تمرینات این سایت کافی است.

Visual Studio Enterprise 2017 دارای محیطی کامل‌تر و ابزارهای بیشتری جهت عیب‌یابی و رسم نمودارهای مختلف است که در Visual Studio Community وجود ندارند. ویژوال استودیو فقط به C++ خلاصه نمی‌شود و دارای زبان‌های برنامه‌نویسی دیگری از جمله ویژوال بیسیک نیز می‌باشد.

دانلود و نصب ویژوال استودیو

در این درس می‌خواهیم نحوه دانلود و نصب نرم افزار Visual Studio Community 2017 را آموزش دهیم. در جدول زیر لیست نرم افزارها و سخت افزارهای لازم جهت نصب ویژوال استودیو 2017 آمده است:

سیستم عامل	سخت افزار
Windows 10	1.6 GHz or faster processor
Windows 8.1	1 GB of RAM (1.5 GB if running on a virtual machine)
Windows 8	4 GB of available hard disk space
Windows 7 Service Pack 1	5400 RPM hard disk drive

DirectX 9-capable video card that runs at 1024 x 768 or higher display resolution	Windows Server 2012 R2
	Windows Server 2012
	Windows Server 2008 R2 SP1

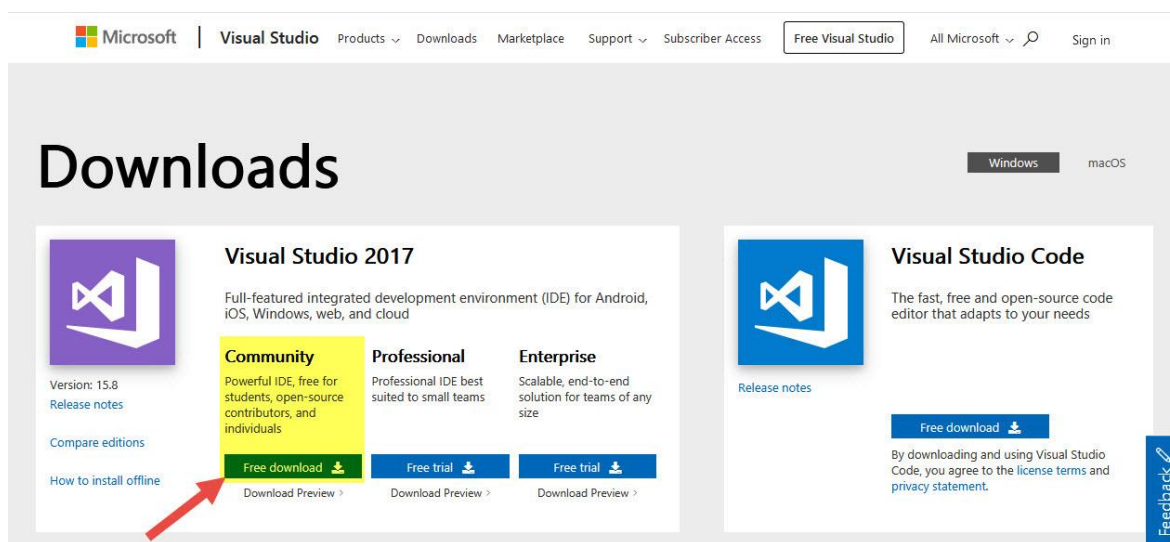
دانلود Visual Studio Community 2017

Visual Studio Community 2017 به صورت آزاد در دسترس است و می‌توانید آن را از لینک زیر دانلود کنید:

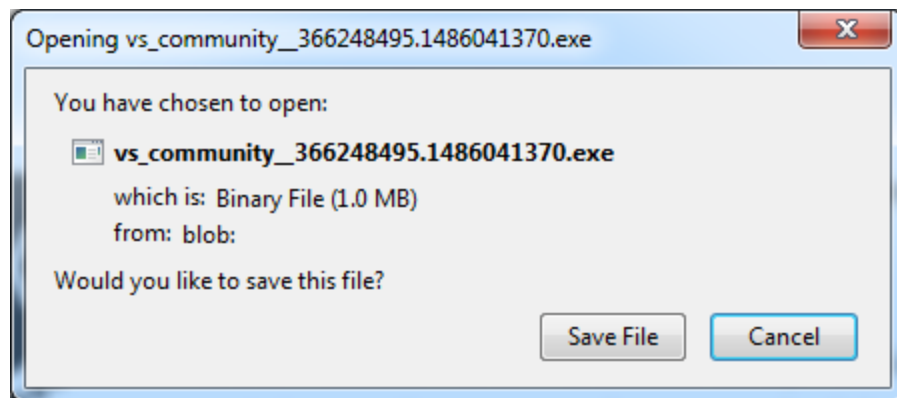
<https://www.visualstudio.com/en-us/downloads/download-visual-studio-vs.aspx>

با کلیک بر روی لینک بالا صفحه ای به صورت زیر ظاهر می‌شود که در داخل این صفحه می‌توان با کلیک بر روی Visual

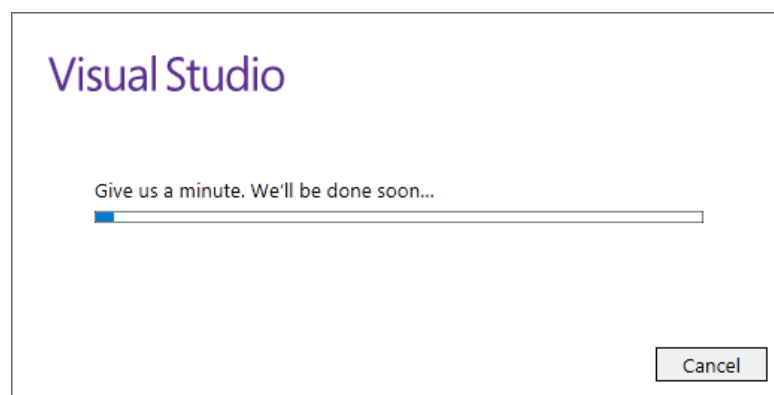
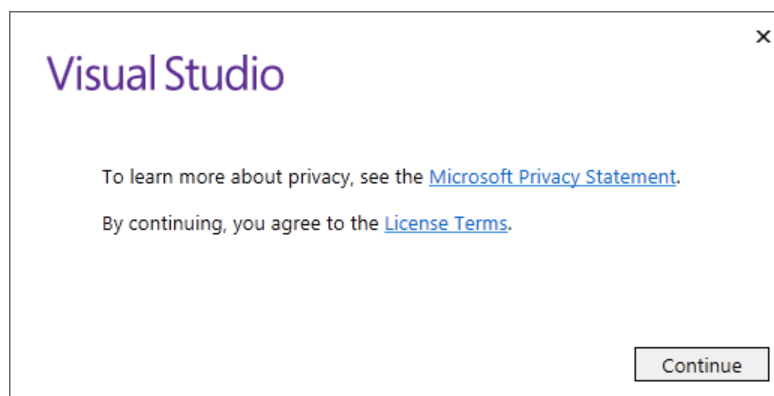
Studio Community 2017 آن را دانلود کرد:



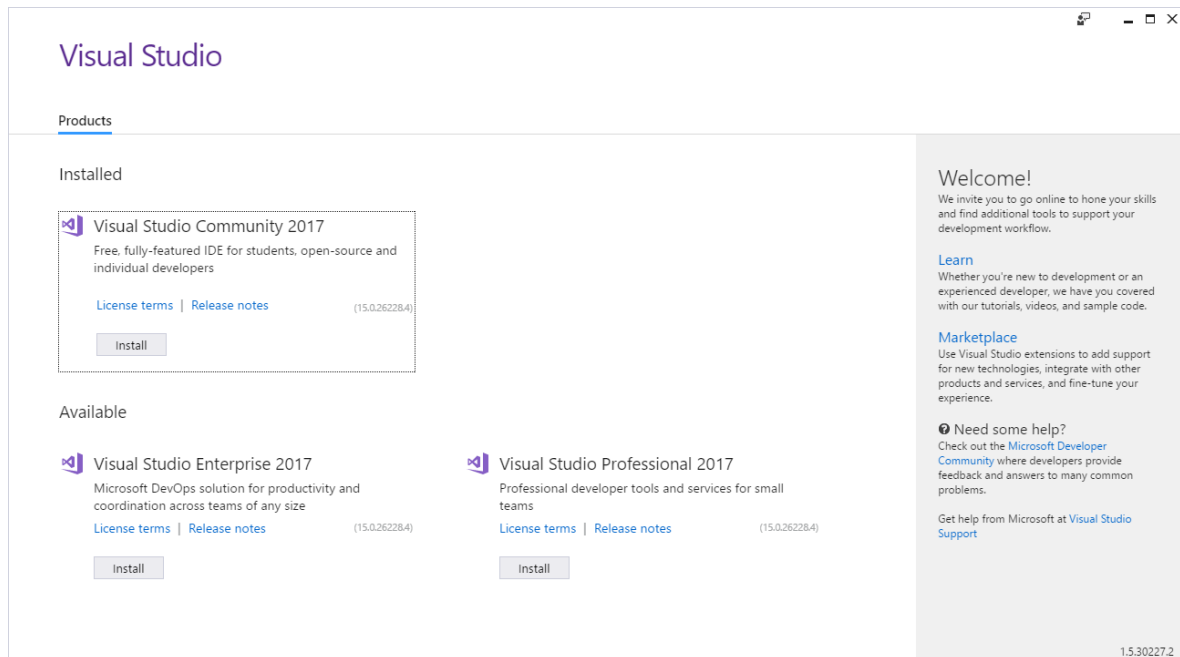
بعد از کلیک بر روی گزینه Download یک صفحه به صورت زیر باز می‌شود و از شما می‌خواهد که فایلی با نام vs_community.exe را ذخیره کنید:



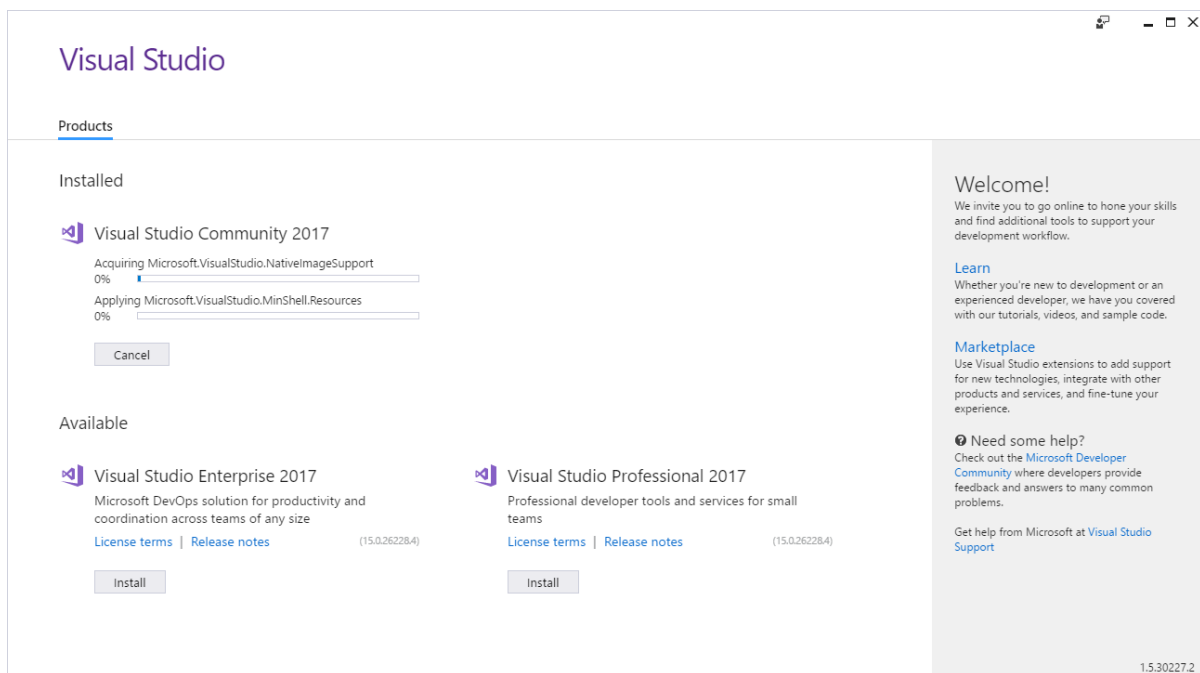
با ذخیره و اجرای این فایل مراحل نصب Visual Studio Community 2017 آغاز می‌شود (Visual Studio Community 2017 حدود 5 گیگابایت حجم دارد و برای دانلود آن به یک اینترنت پر سرعت دارید):



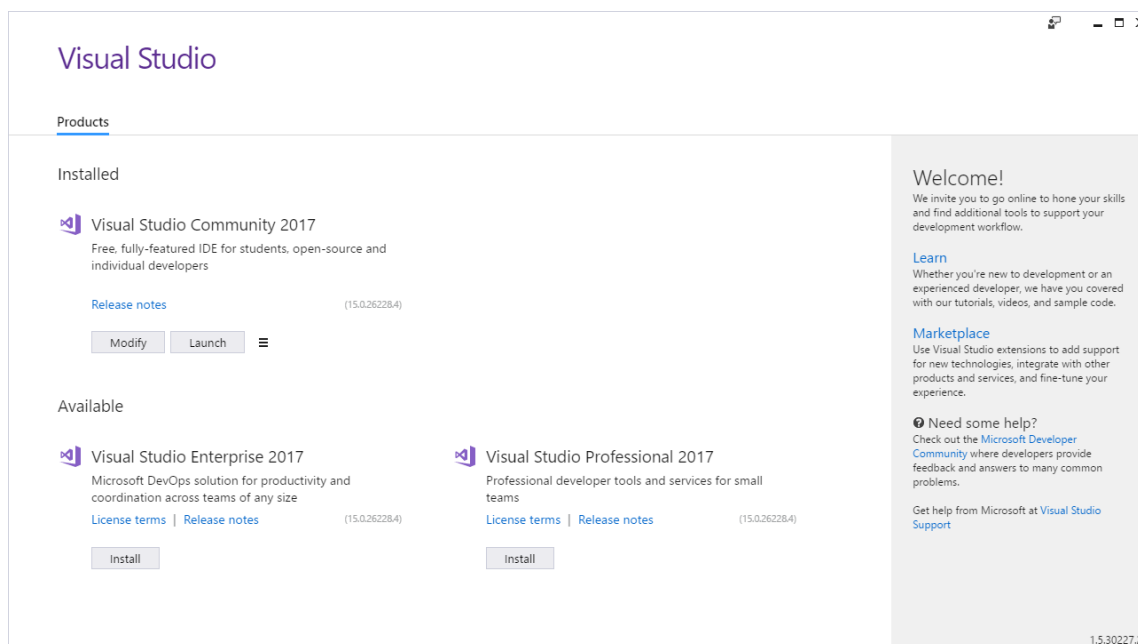
بعد از گذراندن دو صفحه بالا صفحه ای به صورت زیر باز می‌شود که در آن نسخه‌های مختلف ویژوال استودیو به شما نمایش داده می‌شود. بر روی گزینه Install روبروی Visual Studio Community کلیک کنید:



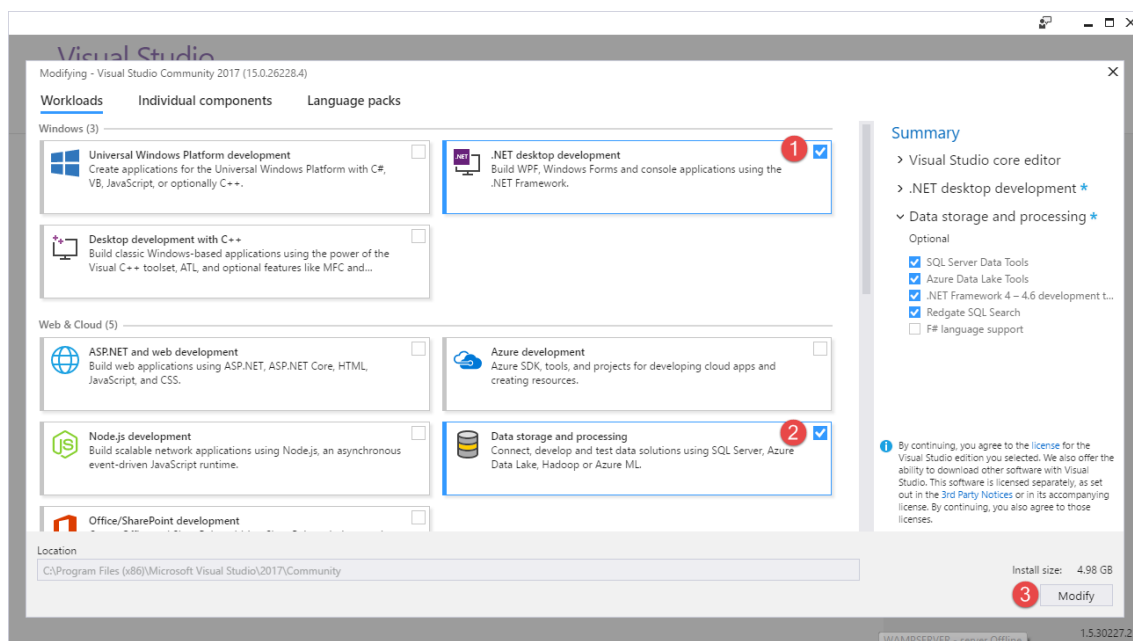
بعد از کلیک بر روی دکمه Install مرحله نصب شروع می‌شود:



بعد از اتمام مرحله بالا صفحه ای به صورت زیر باز می‌شود:



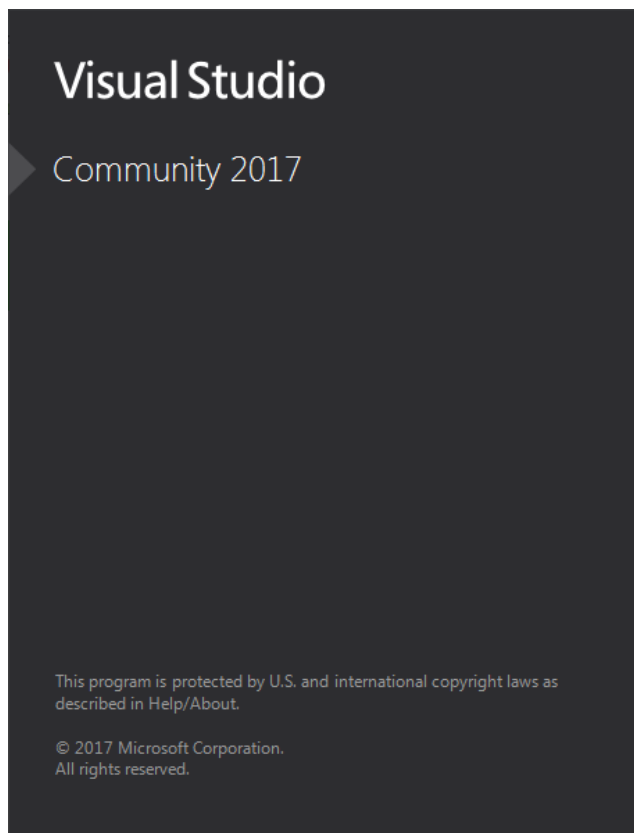
در صفحه بالا بر روی گزینه Modify کلیک کنید و گزینه‌های زیر را تیک بزنید و سپس بر روی دکمه Modify کلیک کنید:



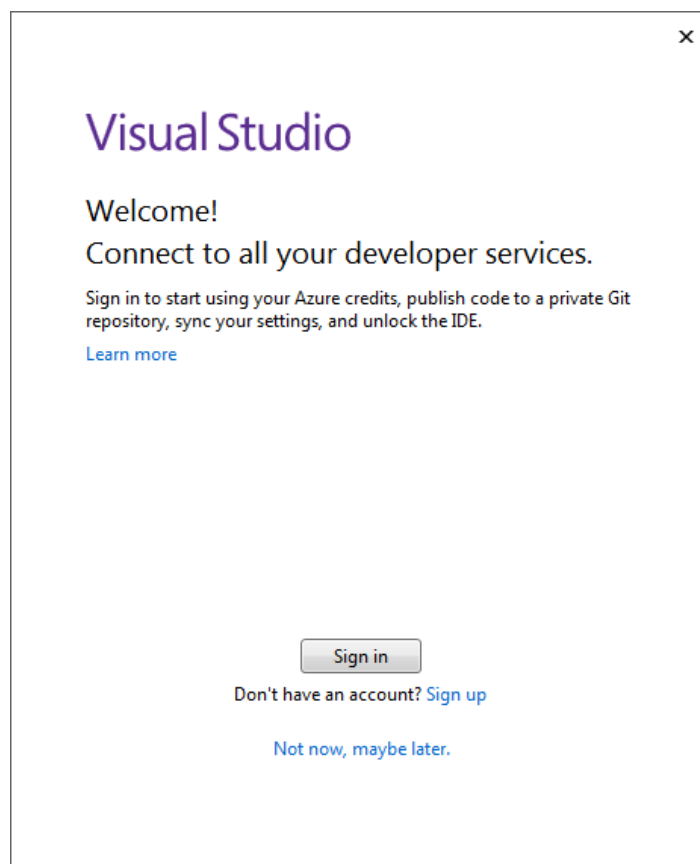
بعد از این مرحله ویژوال استودیو به صورت کامل نصب شده و شما می‌توانید از آن استفاده کنید.

شروع کار با Visual Studio Community

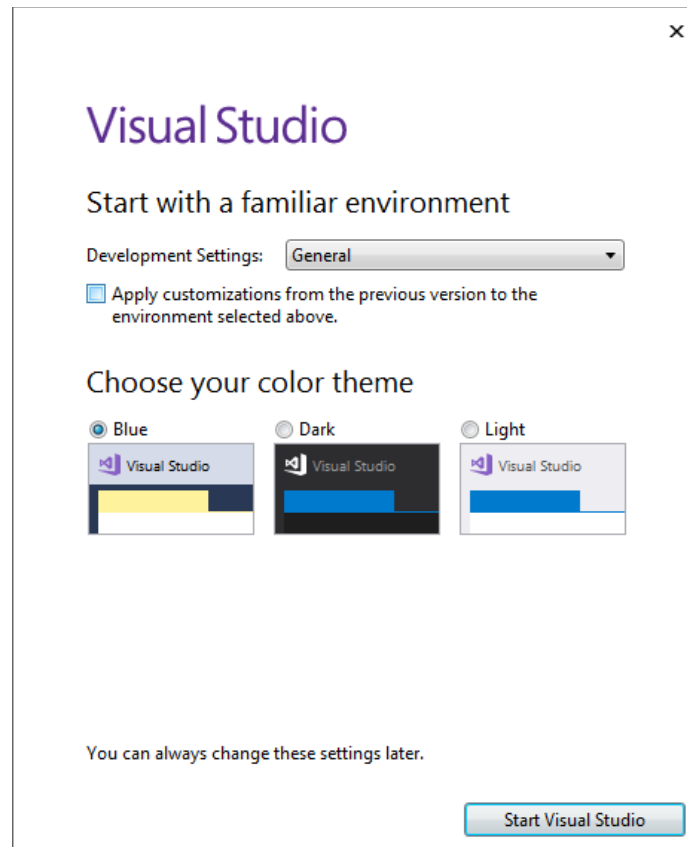
برنامه ویژوال استودیو را اجرا کرده و منتظر بمانید تا صفحه آن بارگذاری شود:



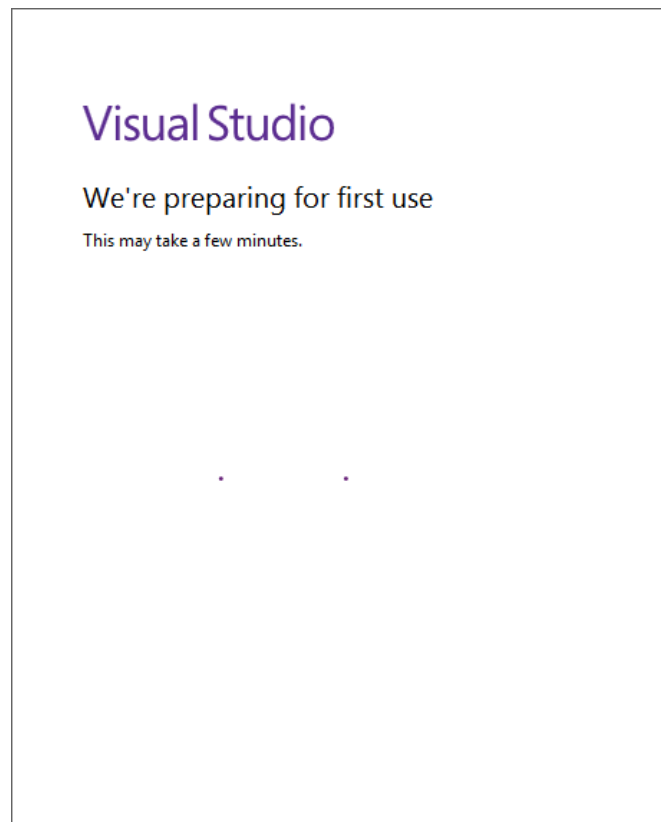
اگر دارای یک اکانت مایکروسافت باشید می‌توانید تغییراتی که در ویژوال استودیو می‌دهید را در فضای ابری ذخیره کرده و اگر آن را در کامپیوتر دیگر نصب کنید، می‌توانید با وارد شده به اکانت خود، تغییرات را به صورت خودکار بر روی ویژوال استودیویی که تازه نصب شده اعمال کنید. البته می‌توانید این مرحله را با زدن دکمه Not now, maybe later رد کنید:



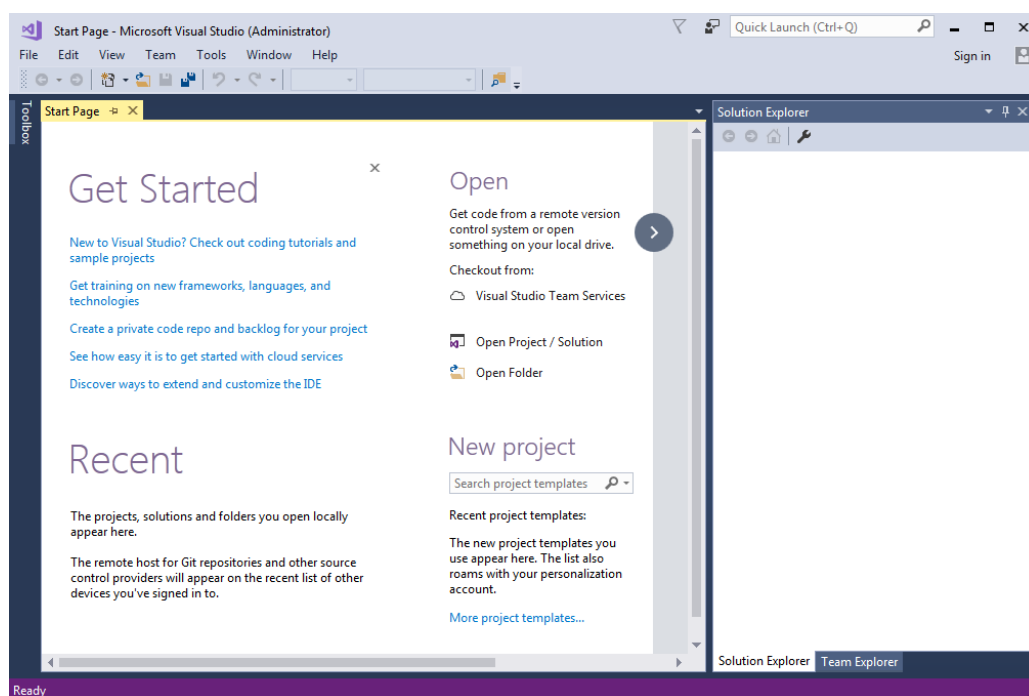
شما می‌توانید از بین سه ظاهر از پیش تعریف شده در ویژوال استودیو یکی را انتخاب کنید. من به صورت پیشفرض ظاهر Blue را انتخاب می‌کنم ولی شما می‌توانید بسته به سلیقه خود، ظاهر دیگر را انتخاب کنید:



بعد از زدن دکمه Start Visual Studio صفحه ای به صورت زیر ظاهر می شود:



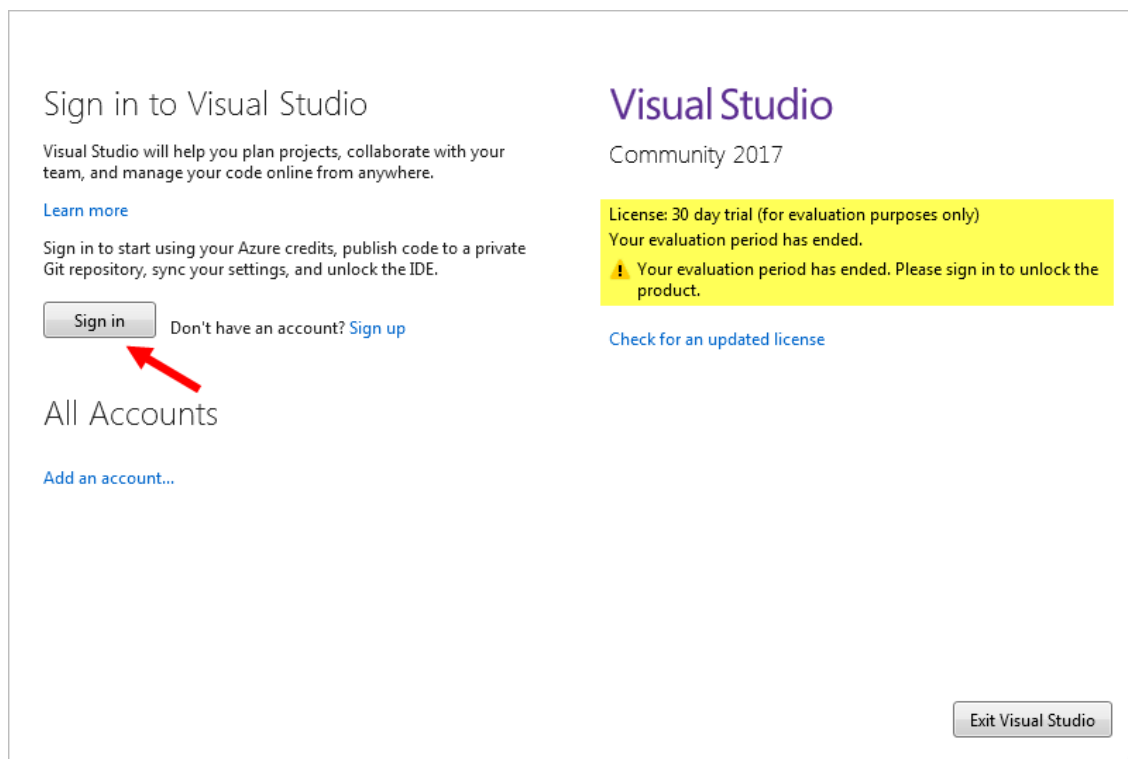
بعد از بارگذاری کامل Visual Studio Community صفحه اصلی برنامه به صورت زیر نمایش داده می‌شود که نشان از نصب کامل آن دارد:



قانونی کردن ویژوال استودیو

Visual Studio Community 2017 رایگان است. ولی گاهی اوقات ممکن است با پیغامی به صورت زیر مبنی بر منقضی شدن

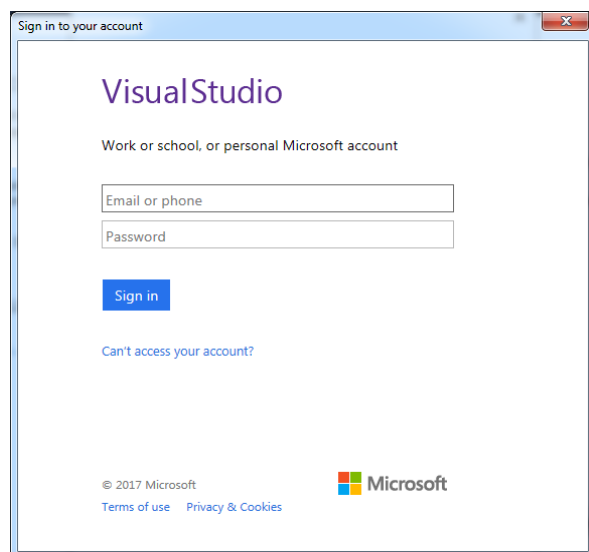
آن مواجه شوید:



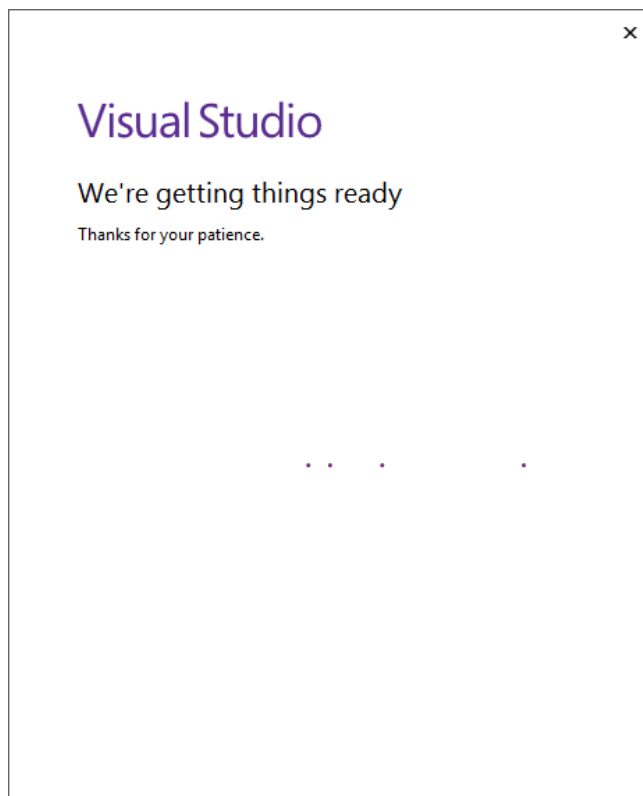
همانطور که در شکل بالا مشاهده می‌کنید، بر روی دکمه Signin کلیک می‌کنید تا وارد اکانت مایکروسافت خود شوید. اگر اکانت ندارید، می‌توانید از لینک زیر یک اکانت ایجاد کنید:

<http://www.w3-farsi.com/?p=22201>

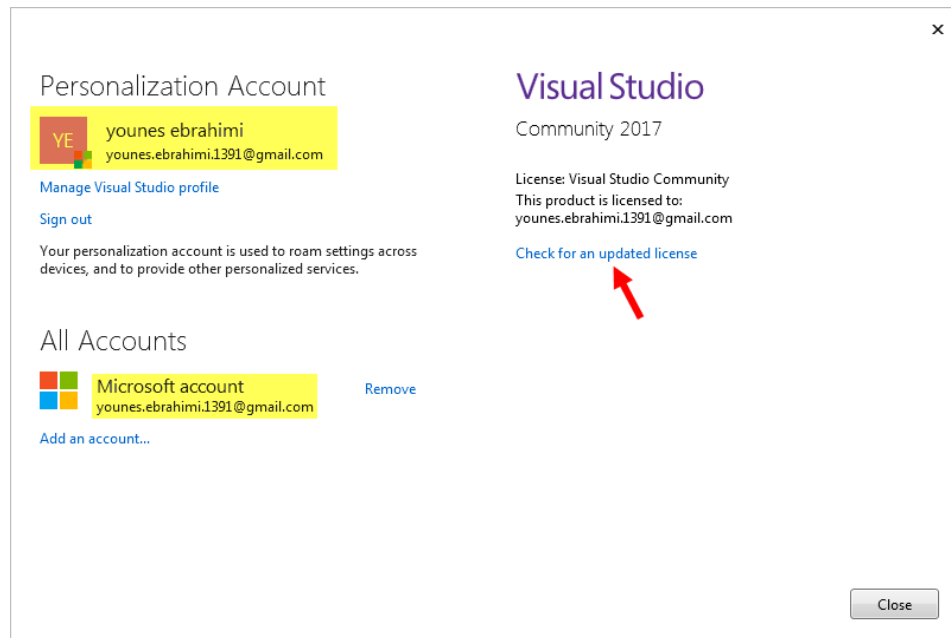
بعد از ایجاد اکانت همانطور که در شکل بالا مشاهده می‌کنید، بر روی گزینه Singin کلیک می‌کنیم. با کلیک بر روی این گزینه صفحه ای به صورت زیر ظاهر می‌شود که از شما مشخصات اکانتتان را می‌خواهد، آن‌ها را وارد کرده و بر روی گزینه Singin کلیک کنید:



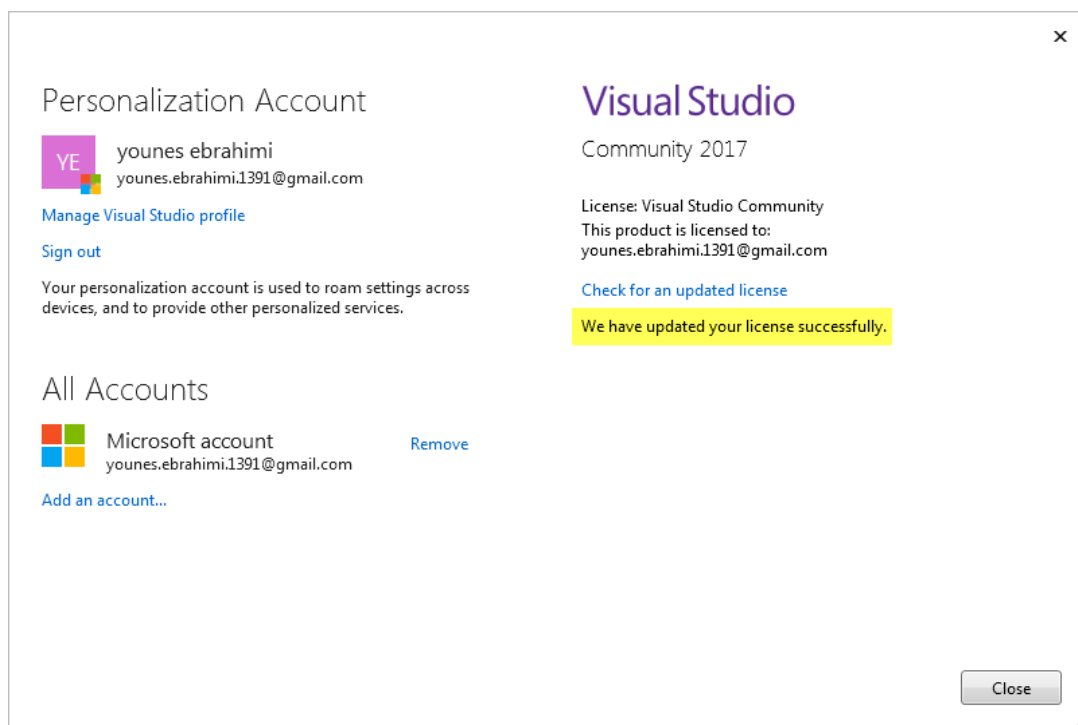
با کلیک بر روی گزینه Signin پنجره ای به صورت زیر نمایش داده می شود، منتظر می مانید تا پنجره بسته شود:



با بسته شدن پنجره بالا، پنجره ای به صورت زیر ظاهر می شود که مشخصات اکانت شما در آن نمایش داده می شود، که نشان از ورود موفقیت آمیز شما دارد. در این صفحه بر روی گزینه Check an updated license کلیک کنید:

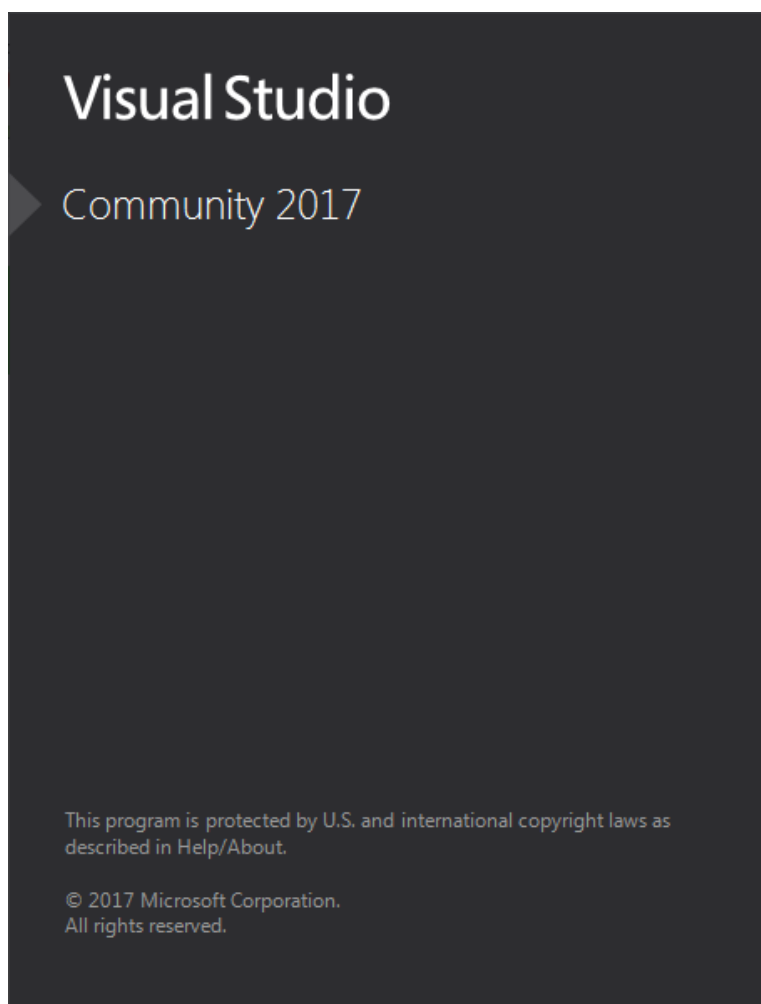


با کلیک بر روی این گزینه بعد از چند ثانیه پیغام we have updated your license successfully نمایش داده می‌شود
و به این صورت ویژوال استودیو قانونی می‌شود:

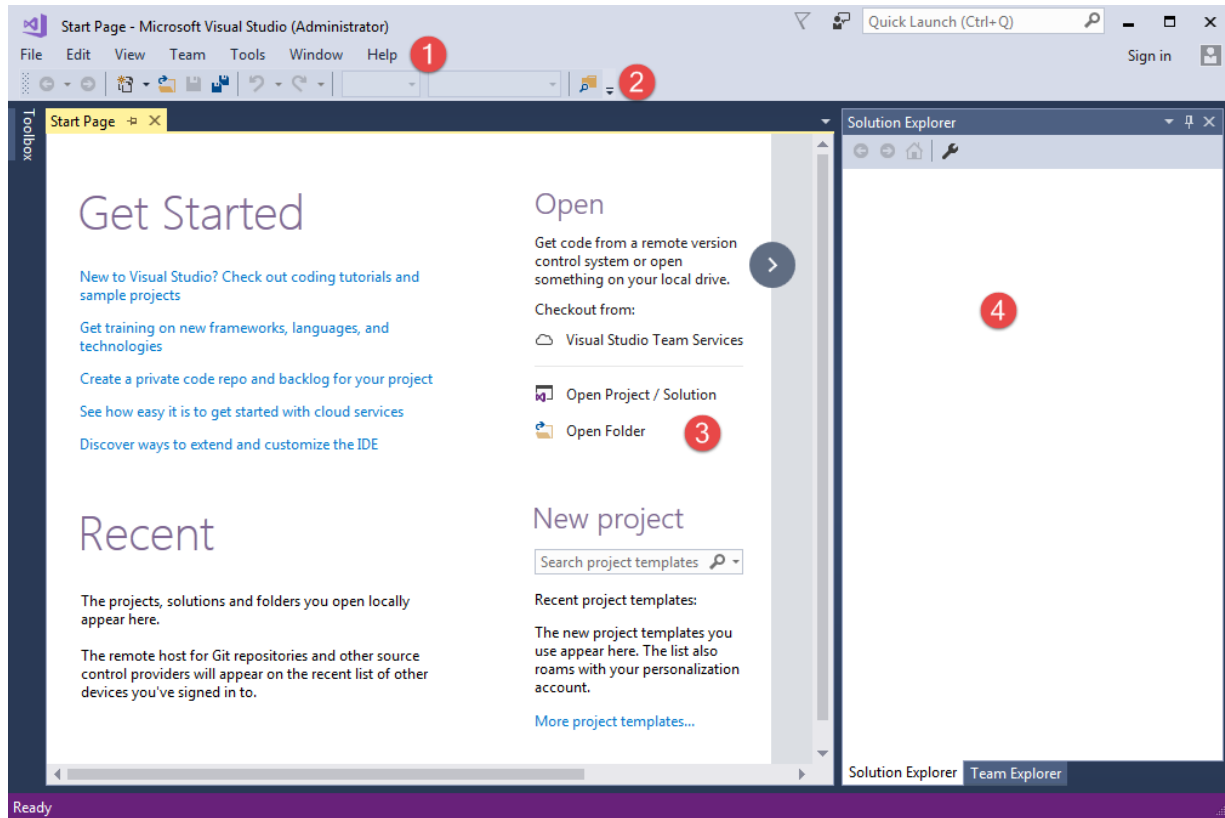


به ویژوال استودیو خوش آمدید

در این بخش می‌خواهیم درباره قسمت‌های مختلف محیط ویژوال استودیو به شما مطالبی آموزش دهیم. لازم است که با انواع ابزارها و ویژگیهای این محیط آشنا شوید. برنامه ویژوال استودیو را اجرا کنید:



بعد از اینکه صفحه بالا بسته شد وارد صفحه آغازین ویژوال استودیو می‌شویم:



این صفحه بر طبق عناوین خاصی طبقه بندی شده که در مورد آن‌ها توضیح خواهیم داد.

منو بار (Menu Bar)

(1) Menu Bar که شامل منوهای مختلفی برای ساخت، توسعه، نگهداری، خطایابی و اجرای برنامه‌ها است. با کلیک بر روی هر منو دیگر منوهای وابسته به آن ظاهر می‌شوند. به این نکته توجه کنید که منو بار دارای آیتم‌های مختلفی است که فقط در شرایط خاصی ظاهر می‌شوند. به عنوان مثال آیتم‌های منوی Project در صورتی نشان داده خواهند شد که پروژه فعال باشد. در زیر برخی از ویژگیهای منوها آمده است:

منو	توضیح
File	شامل دستوراتی برای ساخت پروژه یا فایل، باز کردن و ذخیره پروژه‌ها و خروج از آن‌ها می‌باشد
Edit	شامل دستوراتی جهت ویرایش از قبیل کپی کردن، جایگزینی و پیدا کردن یک مورد خاص می‌باشد

View	به شما اجازه می‌دهد تا پنجره‌های بیشتری باز کرده و یا به آیتم‌های toolbar آیتمی اضافه کنید.
Project	شامل دستوراتی در مورد پروژه ای است که شما بر روی آن کار می‌کنید.
Debug	به شما اجازه کامپایل، اشکال زدایی و اجرای برنامه را می‌دهد
Data	شامل دستوراتی برای اتصال به دیتابیس‌ها می‌باشد.
Format	شامل دستوراتی جهت مرتب کردن اجزای گرافیکی در محیط گرافیکی برنامه می‌باشد.
Tools	شامل ابزارهای مختلف، تنظیمات و ... برای ویژوال استودیو می‌باشد.
Window	به شما اجازه تنظیمات ظاهری پنجره‌ها را می‌دهد.
Help	شامل اطلاعاتی در مورد برنامه ویژوال استودیو می‌باشد

The Toolbars

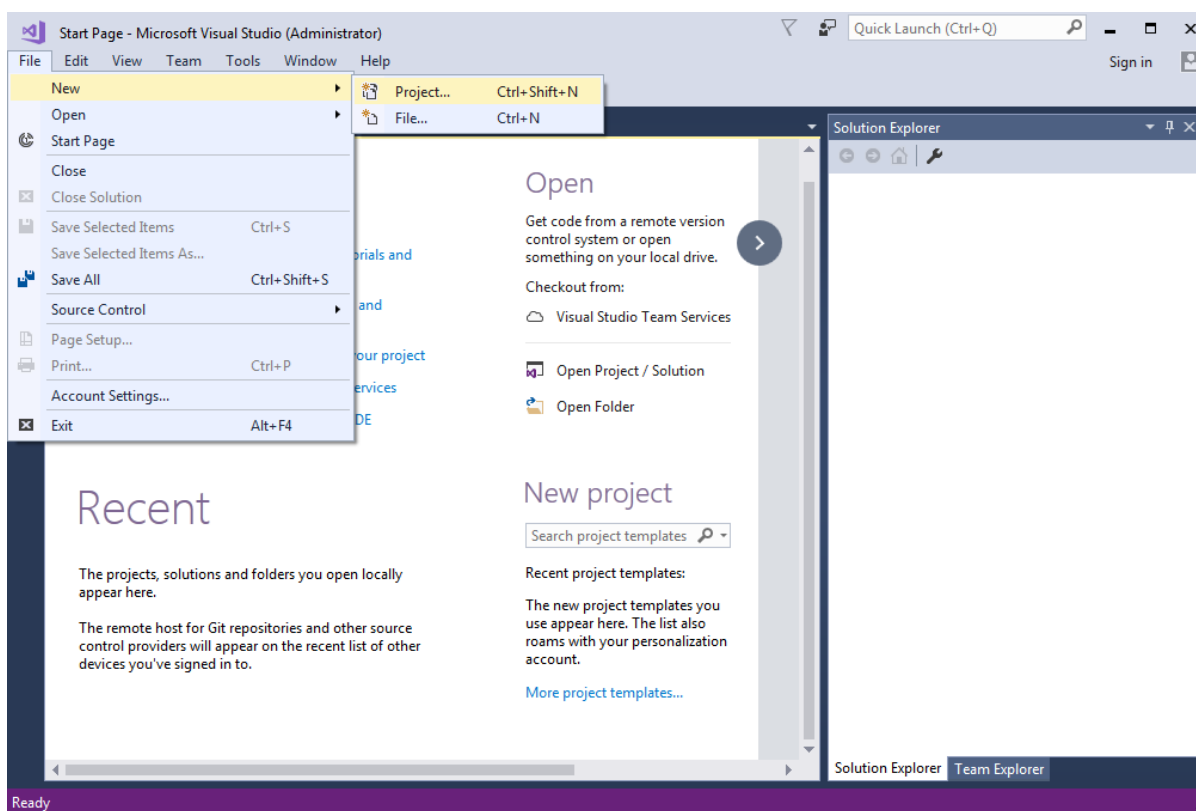
(2) Toolbar به طور معمول شامل همان دستوراتی است که در داخل منوها قرار دارند. Toolbar همانند یک میانبر عمل می‌کند. هر دکمه در Toolbar دارای آیکونی است که کاربرد آنرا نشان می‌دهد. اگر در مورد عملکرد هر کدام از این دکمه‌ها شک داشتید می‌توانید با نشانگر ماوس بر روی آن مکث کوتاهی بکنید تا کاربرد آن به صورت یک پیام (tool tip) نشان داده شود. برخی از دستورات مخفی هستند و تحت شرایط خاص ظاهر می‌شوند. همچنین می‌توانید با کلیک راست بر روی منطقه خالی از Toolbar و یا از مسیر View > Toolbars دستورات بیشتری به آن اضافه کنید. برخی از دکمه‌ها دارای فلش‌های کوچکی هستند که با کلیک بر روی آن‌ها دیگر دستورات وابسته به آن‌ها ظاهر می‌شوند. سمت چپ هر Toolbar به شما اجازه جا به جایی آن را می‌دهد.

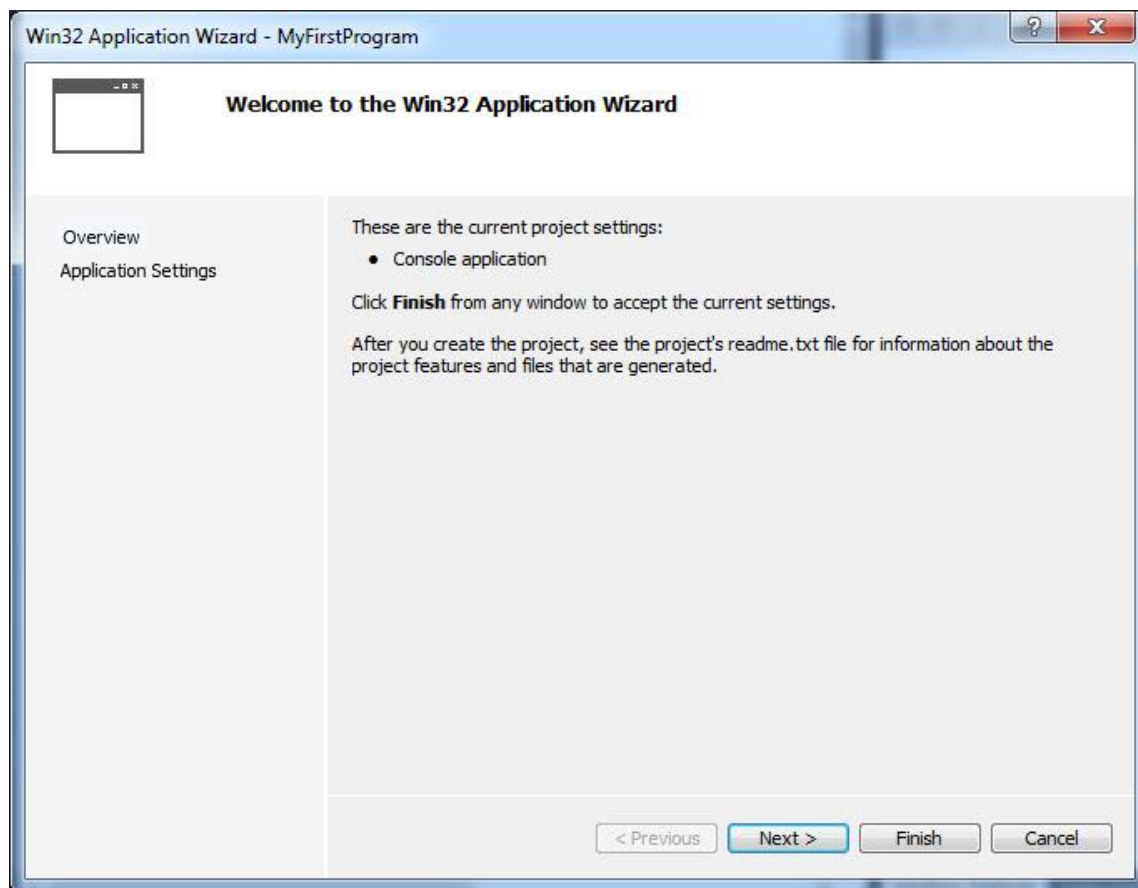
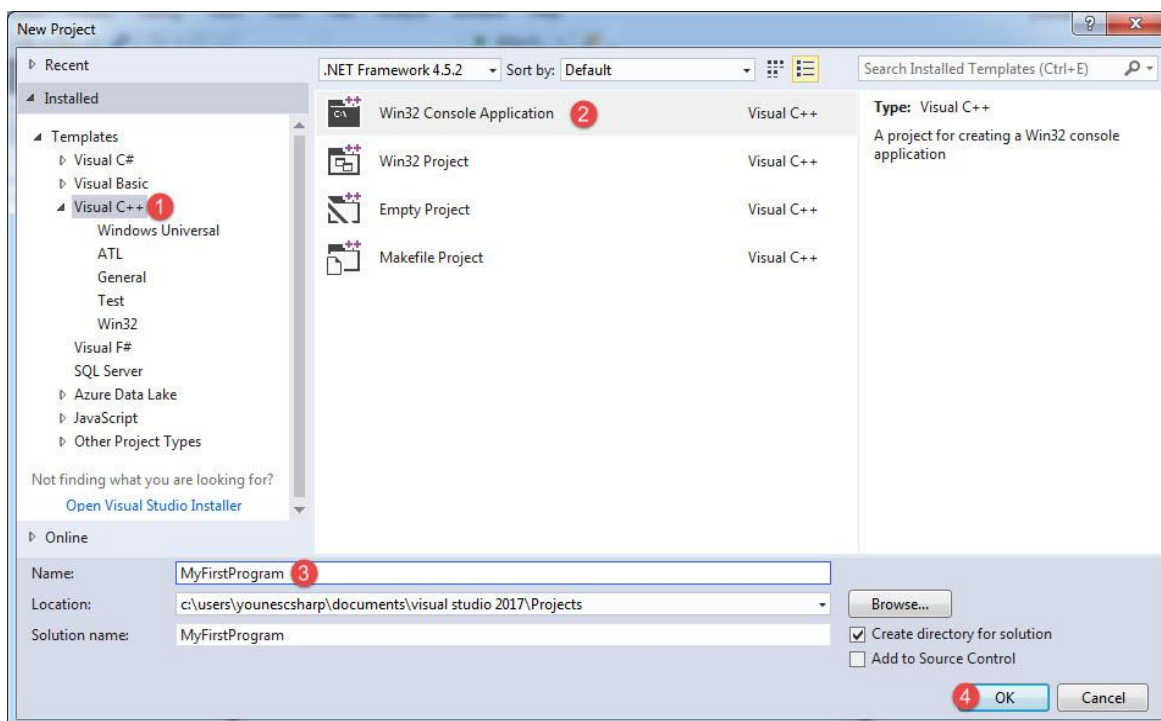
صفحه آغازین (Start Page)

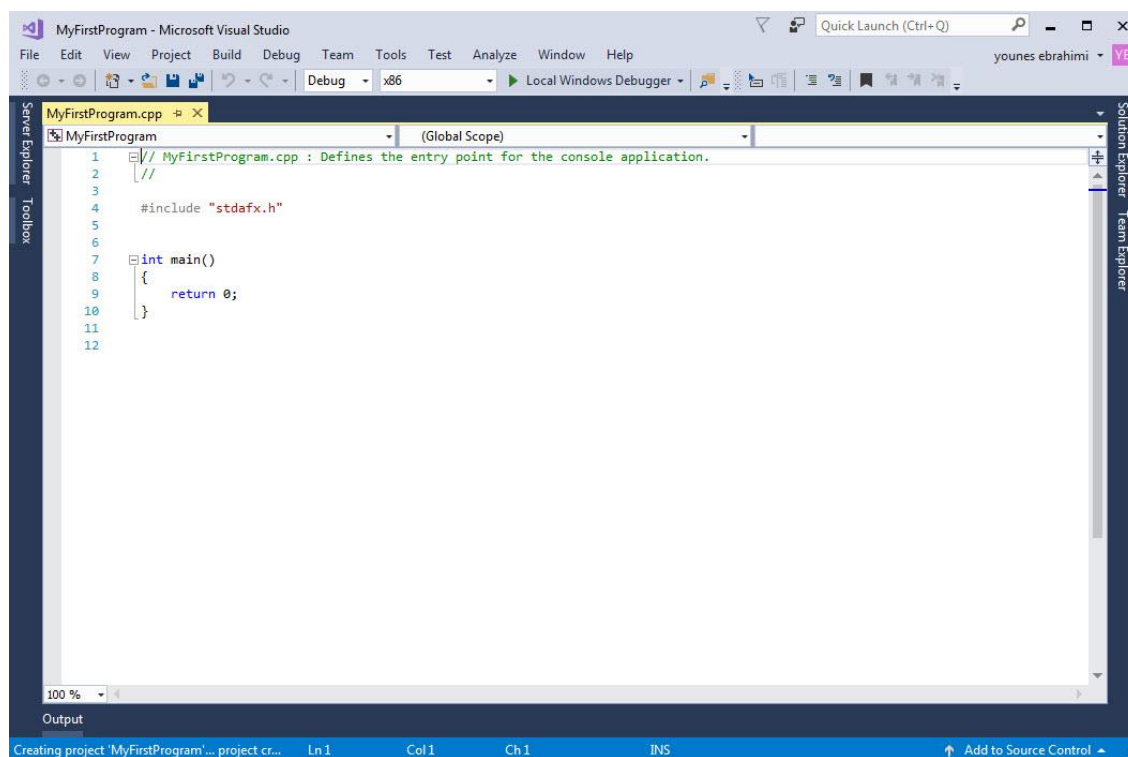
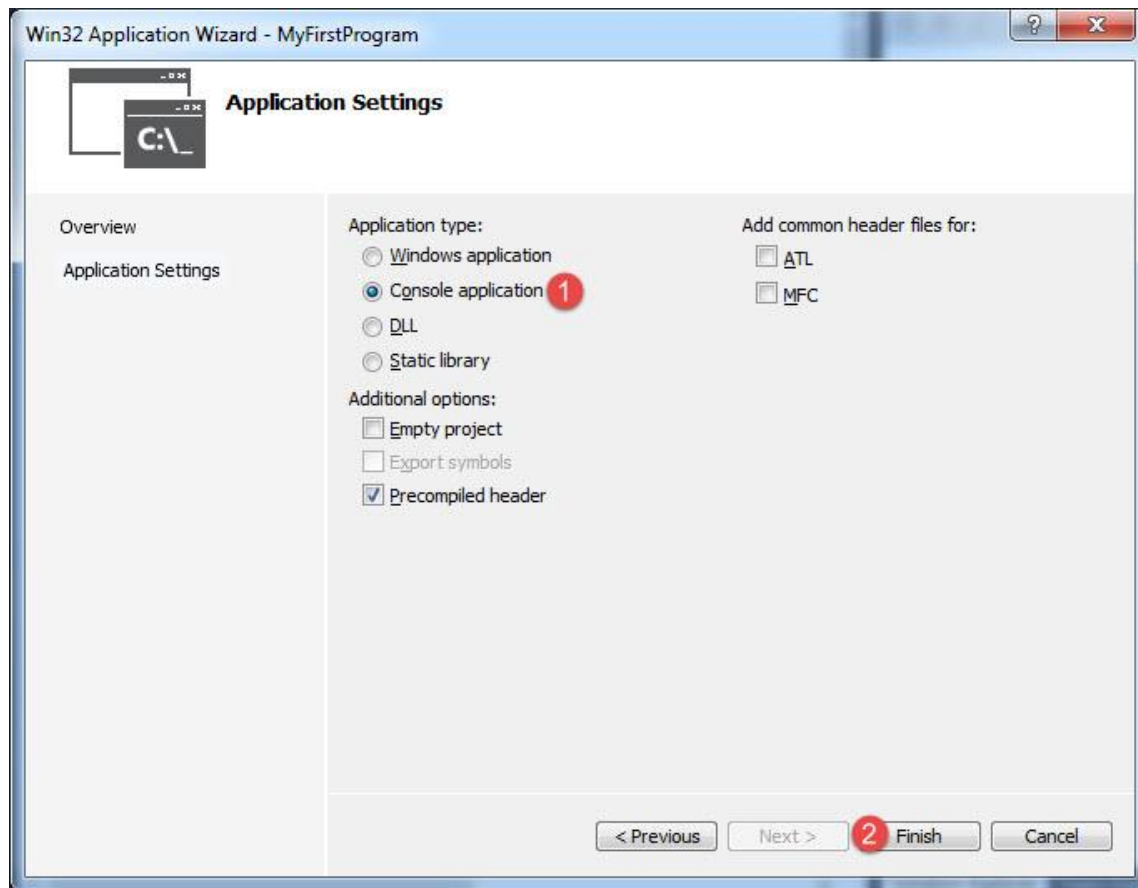
(3) Start برای ایجاد یک پروژه و باز کردن آن از این قسمت استفاده می‌شود. همچنین اگر از قبل پروژه ای ایجاد کرده‌اید می‌توانید آن را در Recent Projects مشاهده و اجرا کنید.

ساخت یک برنامه ساده

اجازه بدهید یک برنامه بسیار ساده به زبان سی پلاس پلاس (C++) بنویسیم. این برنامه یک پیغام را در محیط کنسول نمایش می‌دهد. در این درس، می‌خواهم ساختار و دستور زبان یک برنامه ساده C++ را توضیح دهم. هر چند که محیط‌های کدنویسی زیادی برای C++ وجود دارند، ولی ما از ساده‌ترین روش برای کدنویسی استفاده می‌کنیم. برنامه ویژوال استودیو را باز کرده و به صورت زیر یک پروژه ایجاد کنید :





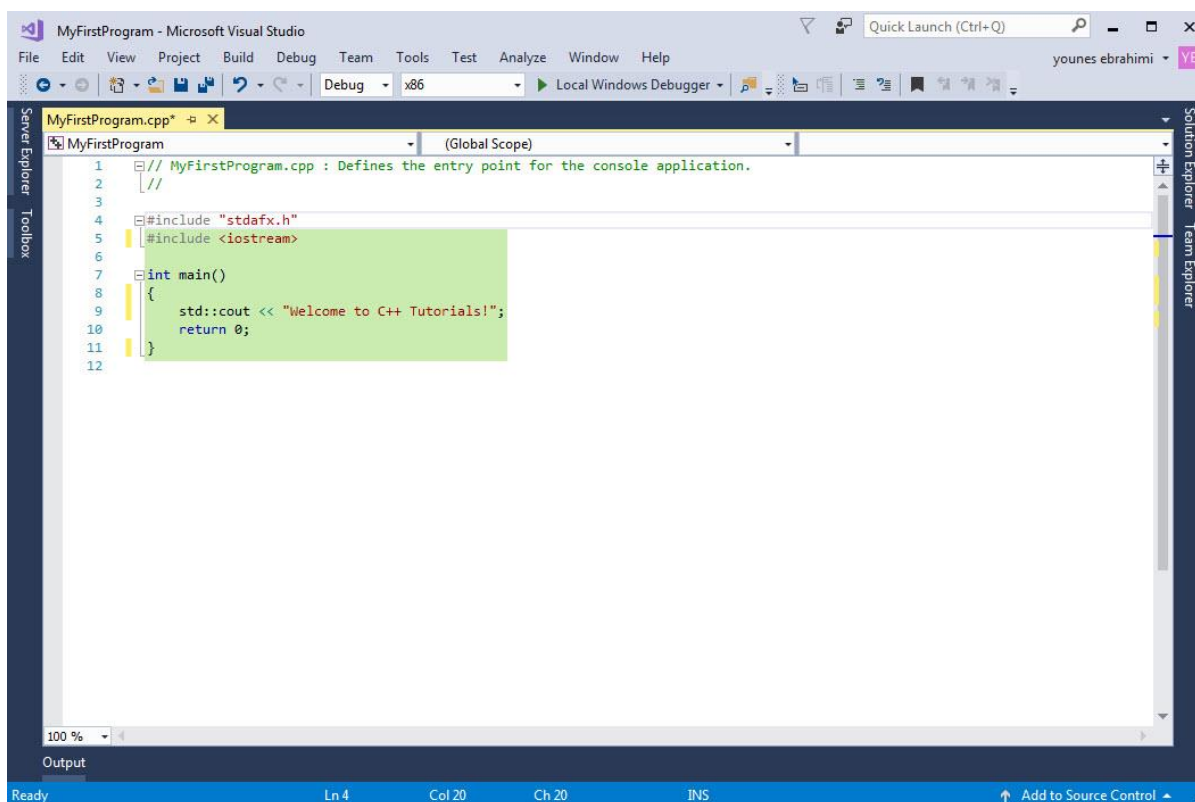


حال کدهای زیر را در این محیط نوشته :

```
#include <iostream>

int main()
{
    std::cout << "Welcome to C++ Tutorials!";
}
```

تا شکل نهایی برنامه به صورت زیر در آید:



ساختار یک برنامه در C++

مثال بالا، ساده‌ترین برنامه‌ای است که شما می‌توانید در C++ بنویسید. هدف در مثال بالا نمایش یک پیغام در صفحه نمایش است. هر زبان برنامه نویسی دارای قواعدی برای کدنویسی است. اجازه بدهید هر خط کد را در مثال بالا توضیح بدهیم. در خطوط 4 و 5، فایل هدر یا سرآیند آمده است. فایل‌های سرآیند کتابخانه استاندارد C++ می‌باشند و در این برنامه ما به فایل سرآیند `iostream` نیاز داریم (در درس‌های آینده در مورد این فایل‌ها به طور مفصل توضیح می‌دهیم). خط 7 متد `main()` یا متد اصلی

نامیده می‌شود. هر متد شامل یک سری کد است که وقتی اجرا می‌شوند که متد را صدا بزنیم. درباره متد و نحوه صدا زدن آن در فصول بعدی توضیح خواهیم داد. متد `main()` نقطه آغاز اجرای برنامه است. این بدان معناست که ابتدا تمام کدهای داخل متد `main()` و سپس بقیه کدها اجرا می‌شود. درباره متد `main()` در فصول بعدی توضیح خواهیم داد. متد `main()` و سایر متدها دارای آکولاد و کدهایی در داخل آن‌ها می‌باشند و وقتی کدها اجرا می‌شوند که متدها را صدا بزنیم. هر خط کد در C++ به یک سمیکال (:) ختم می‌شود. اگر سمیکال در آخر خط فراموش شود برنامه با خطا مواجه می‌شود. مثالی از یک خط کد در C++ به صورت زیر است :

```
std::cout << "Welcome to C++ Tutorials!";
```

این خط کد پیغام Welcome to Visual C++ Tutorials! را در صفحه نمایش نشان می‌دهد. از شیء `cout` برای چاپ یک رشته استفاده می‌شود. یک رشته گروهی از کاراکترها است، که به وسیله دابل کوتیشن (") محصور شده است. مانند "Welcome to Visual C++ Tutorials!"

یک کاراکتر می‌تواند یک حرف، عدد، علامت یا ... باشد. در کل مثال بالا نحوه استفاده از شیء `cout` است که در داخل فضای نام `std` قرار دارد را نشان می‌دهد. توضیحات بیشتر در درسهای آینده آمده است. C++ فضای خالی و خطوط جدید را نادیده می‌گیرد. بنابراین شما می‌توانید همه برنامه را در یک خط بنویسید. اما اینکار خواندن و اشکال زدایی برنامه را مشکل می‌کند. یکی از خطاهای معمول در برنامه نویسی فراموش کردن سمیکال در پایان هر خط کد است. به مثال زیر توجه کنید :

```
std::cout <<
"Welcome to C++ Tutorials!";
```

سی پلاس پلاس فضای خالی بالا را نادیده می‌گیرد و از کد بالا اشکال نمی‌گیرد. اما از کد زیر ایراد می‌گیرد :

```
std::cout << ;
"Welcome to C++ Tutorials!";
```

به سمیکال آخر خط اول توجه کنید. برنامه با خطای نحوی مواجه می‌شود چون دو خط کد مربوط به یک برنامه هستند و شما فقط باید یک سمیکال در آخر آن قرار دهید. همیشه به یاد داشته باشید که C++ به بزرگی و کوچکی حروف حساس است. یعنی به طور مثال `man` و `MAN` در سی پلاس پلاس با هم فرق دارند. رشته‌ها و توضیحات از این قاعده مستثنی هستند که در درسهای آینده توضیح خواهیم داد. مثلاً کدهای زیر با خطا مواجه می‌شوند و اجرا نمی‌شوند :

```
std::cout << "Welcome to C++ Tutorials!";  
STD::cout << "Welcome to C++ Tutorials!";  
Std::Cout << "Welcome to C++ Tutorials!";
```

تغییر در بزرگی و کوچکی حروف از اجرای کدها جلوگیری می کند. اما کد زیر کاملاً بدون خطا است :

```
std::cout << "Welcome to C++ Tutorials!";
```


همیشه کدهای خود را در داخل آکولاد بنویسید .


```
{  
    statement1;  
}
```


این کار باعث می شود که کدنویسی شما بهتر به چشم بیاید و تشخیص خطاها راحت تر باشد .

ذخیره پروژه و برنامه

برای ذخیره پروژه و برنامه می توانید به مسیر **File > Save All** بروید یا از کلیدهای میانبر **Ctrl+Shift+S** استفاده کنید.

همچنین می توانید از قسمت **Toolbar** بر روی شکل  کلیک کنید. برای ذخیره یک فایل ساده می توانید به مسیر **File >**

Save (FileName) بروید یا از کلیدهای میانبر **Ctrl+S** استفاده کنید. همچنین می توانید از قسمت **Toolbar** بر روی شکل  کلیک کنید.

برای باز کردن یک پروژه یا برنامه از منوی **File** گزینه **Open** را انتخاب می کنید یا بر روی آیکون  در **toolbar** کلیک کنید.

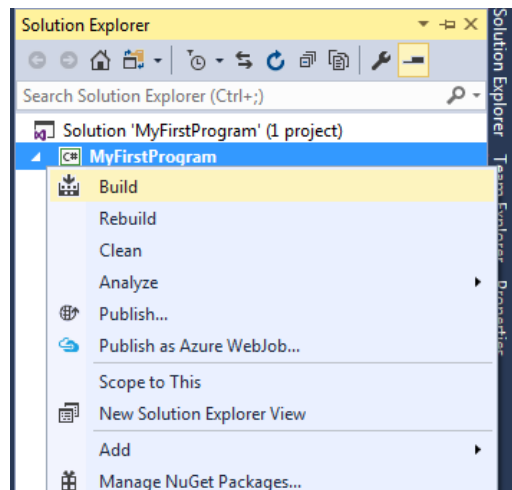
سپس به محلی که پروژه در آنجا ذخیره شده می روید و فایلی با پسوند **sln** یا پروژه با پسوند **cspj** را باز می کنید.

کامپایل برنامه

برای کامپایل برنامه از منوی **Debug** گزینه **Build Solution** را انتخاب می کنید یا دکمه **F6** را بر روی صفحه کلید فشار می دهیم.

این کار همه پروژه های داخل **solution** را کامپایل می کند. برای کامپایل یک قسمت از **solution** به **Solution Explorer**

می رویم و بر روی آن قسمت راست کلیک کرده و از منوی باز شوند گزینه **build** را انتخاب می کنید. مانند شکل زیر:



اجرای برنامه

دو راه برای اجرای برنامه وجود دارد:

- اجرا همراه با اشکال زدایی (Debug)
- اجرا بدون اشکال زدایی (Non-Debug)

اجرای بدون اشکال زدایی برنامه، خطاهای برنامه را نادیده می‌گیرد. با اجرای برنامه در حالت Non-Debug سریعاً برنامه اجرا می‌شود و شما با زدن یک دکمه از برنامه خارج می‌شوید. در حالت پیش فرض حالت Non-Debug مخفی است و برای استفاده از آن می‌توان از منوی Debug گزینه Start Without Debugging را انتخاب کرد یا از دکمه‌های ترکیبی `Ctrl + F5` استفاده نمود:


```
Welcome to C++ Tutorials!Press any key to continue...
```

به این نکته توجه کنید که پیغام `Press any key to continue...` جز خروجی به حساب نمی‌آید و فقط نشان دهنده آن است که برنامه در حالت Non-Debug اجرا شده است و شما می‌توانید با زدن یک کلید از برنامه خارج شوید. برای اینکه تفکیکی بین عبارت مورد نظر ما و عبارت به وجود بیاید کافیهست که خط 9 کد ابتدای درس را به صورت زیر تغییر دهید :

```
std::cout << "Welcome to C++ Tutorials!" << endl;
```

حال اگر برنامه را دوباره اجرا کنید، خروجی به صورت زیر نمایش داده می‌شود :

```
Welcome to C++ Tutorials!
Press any key to continue...
```

دسترسی به حالت Debug Mode آسان تر است و به صورت پیشفرض برنامه‌ها در این حالت اجرا می‌شوند. از این حالت برای رفع خطاها و اشکال زدایی برنامه‌ها استفاده می‌شود که در درس‌های آینده توضیح خواهیم داد. شما همچنین می‌توانید از `break points` و قسمت Help برنامه در مواقعی که با خطا مواجه می‌شوید استفاده کنید. برای اجرای برنامه با حالت Debug Mode می‌توانید از منوی Debug گزینه Start Debugging را انتخاب کرده و یا دکمه F5 را فشار دهید. همچنین می‌توانید بر روی شکل  در toolbar کلیک کنید. اگر از حالت Debug Mode استفاده کنید برنامه نمایش داده شده و فوراً ناپدید می‌شود. برای جلوگیری از این اتفاق شما می‌توانید از کلاس و متد `std::cin.get()` قبل از عبارت `return 0`، برای توقف برنامه و گرفتن ورودی از کاربر جهت خروج از برنامه استفاده کنید (درباره متدها در درس‌های آینده توضیح خواهیم داد):

```
1 #include "stdafx.h"
2 #include <iostream>
3
4 int main()
5 {
6     std::cout << "Welcome to C++ Tutorials!" << endl;
7     std::cin.get();
8     return 0;
9 }
```

به این نکته توجه کنید که در درس‌های بعدی خط 1 کد بالا را حذف نکرده و از این خط به بعد کدهای خود را بنویسید.

وارد کردن فضای نام در برنامه

در برنامه فوق ما یک فضای نام در برنامه‌مان با نام `std` داریم، اما سی پلاس پلاس دارای تعداد زیادی فضای نام می‌باشد. یکی از این فضاهای نامی، فضای نام `std` است که شیء `cout` که ما از آن در برنامه بالا استفاده کردیم در این فضای نام قرار دارد.

```
std::cout << "Welcome to C++ Tutorials!" << endl;
```

اینکه قبل از استفاده از هر کلاس ابتدا فضای نام آن را مانند کد بالا بنویسیم کمی خسته کننده است. خوشبختانه C++ به ما اجازه می‌دهد که برای جلوگیری از تکرار مکررات، فضاهای نامی را که قرار است در برنامه استفاده کنیم با استفاده از دستور `using` و کلمه `namespace` در ابتدای برنامه وارد نماییم :


```
using namespace NameofNameSpace;
```

دستور بالا نحوه وارد کردن یک فضای نام در برنامه را نشان می‌دهد. در نتیجه به جای آنکه به صورت زیر ابتدا نام فضای نام و سپس نام کلاس را بنویسیم :

```
std::cout << "Welcome to C++ Tutorials!" << endl;
```

می‌توانیم فضای نام را با دستوری که ذکر شد وارد برنامه کرده و کد بالا را به صورت خلاصه شده زیر بنویسیم :

```
cout << "Welcome to C++ Tutorials!" << endl;
```

دستورات using که باعث وارد شدن فضاهای نامی به برنامه می‌شوند عموماً در ابتدای برنامه و قبل از همه کدها نوشته می‌شوند، پس برنامه‌ی این درس را می‌توان به صورت زیر نوشت :

```
#include "stdafx.h"
#include <iostream>
using namespace std;

int main()
{
    cout << "Welcome to C++ Tutorials!" << endl;
}
```

حال که با خصوصیات و ساختار اولیه C++ آشنا شدید در درسهای آینده مطالب بیشتری از این زبان برنامه نویسی قدرتمند خواهید آموخت.

توضیحات

وقتی که کدی تایپ می‌کنید شاید بخواهید که متنی جهت یادآوری وظیفه آن کد به آن اضافه کنید. در C++ (و بیشتر زبانهای برنامه نویسی) می‌توان این کار را با استفاده از توضیحات انجام داد. توضیحات متونی هستند که توسط کامپایلر نادیده گرفته می‌شوند و به عنوان بخشی از کد محسوب نمی‌شوند.

هدف اصلی از ایجاد توضیحات، بالا بردن خوانایی و تشخیص نقش کدهای نوشته شده توسط شما، برای دیگران است. فرض کنید که می‌خواهید در مورد یک کد خاص، توضیح بدهید، می‌توانید توضیحات را در بالای کد یا کنار آن بنویسید. از توضیحات برای مستند سازی برنامه هم استفاده می‌شود. در برنامه زیر نقش توضیحات نشان داده شده است :

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     // This line will print the message hello world
7     cout << "Hello World!";
8 }
```

در کد بالا، خط 6 یک توضیح درباره خط 7 است که به کاربر اعلام می‌کند که وظیفه خط 7 چیست؟ با اجرای کد بالا فقط جمله Hello World چاپ شده و خط 7 در خروجی نمایش داده نمی‌شود چون کامپایلر توضیحات را نادیده می‌گیرد. توضیحات بر دو نوع‌اند :

توضیحات تک خطی

```
// single line comment
```

توضیحات چند خطی

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

توضیحات تک خطی همانگونه که از نامش پیداست، برای توضیحاتی در حد یک خط به کار می‌روند. این توضیحات با علامت // شروع می‌شوند و هر نوشته‌ای که در سمت راست آن قرار بگیرد جز توضیحات به حساب می‌آید. این نوع توضیحات معمولاً در بالا یا کنار کد قرار می‌گیرند. اگر توضیح درباره یک کد به بیش از یک خط نیاز باشد از توضیحات چند خطی استفاده می‌شود. توضیحات چند خطی با /* شروع و با */ پایان می‌یابند. هر نوشته‌ای که بین این دو علامت قرار بگیرد جز توضیحات محسوب می‌شود.

کاراکترهای کنترلی

کاراکترهای کنترلی کاراکترهای ترکیبی هستند که با یک بک اسلش (\) شروع می‌شوند و به دنبال آن‌ها یک حرف یا عدد می‌آید و یک رشته را با فرمت خاص نمایش می‌دهند. برای مثال برای ایجاد یک خط جدید و قرار دادن رشته در آن می‌توان از کاراکتر کنترلی \n استفاده کرد :

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

int main()
{
    cout << "Hello\nWorld";
}
```

```
Hello
World
```

مشاهده کردید که کامپایلر بعد از مواجهه با کاراکتر کنترلی \n نشانگر ماوس را به خط بعد برده و بقیه رشته را در خط بعد نمایش می‌دهد. جدول زیر لیست کاراکترهای کنترلی و کارکرد آن‌ها را نشان می‌دهد :

عملکرد	کاراکتر کنترلی	عملکرد	کاراکتر کنترلی
Form Feed	\f	چاپ کوتیشن	\'
خط جدید	\n	چاپ دابل کوتیشن	\"
سر سطر رفتن	\r	چاپ بک اسلش	\\
حرکت به صورت افقی	\t	چاپ فضای خالی	\0
حرکت به صورت عمودی	\v	صدای بیپ	\a
چاپ کاراکتر یونیکد	\u	حرکت به عقب	\b

ما برای استفاده از کاراکترهای کنترلی از بک اسلش (\) استفاده می‌کنیم. از آنجاییکه \ معنای خاصی به رشته‌ها می‌دهد برای چاپ بک اسلش (\) باید از (\\) استفاده کنیم :

```
cout << "We can print a \\ by using the \\\\ escape sequence.";
```

```
We can print a \ by using the \\ escape sequence.
```

یکی از موارد استفاده از \\، نشان دادن مسیر یک فایل در ویندوز است :

```
cout << "C:\\Program Files\\Some Directory\\SomeFile.txt";
```

```
C:\Program Files\Some Directory\SomeFile.txt
```

از آنجاییکه از دابل کوتیشن (") برای نشان دادن رشته‌ها استفاده می‌کنیم برای چاپ آن از \" استفاده می‌کنیم :

```
cout << "I said, \"Motivate yourself!\".";
```

```
I said, "Motivate yourself!".
```

همچنین برای چاپ کوتیشن (') از \' استفاده می‌کنیم :

```
cout << "The programmer\'s heaven.";
```

```
The programmer's heaven.
```

برای ایجاد فاصله بین حروف یا کلمات از \t استفاده می‌شود :

```
cout << "Left\tRight";
```

```
Left Right
```

هر تعداد کاراکتر که بعد از کاراکتر کنترلی \r بیایند به اول سطر منتقل و جایگزین کاراکترهای موجود می‌شوند :

```
cout << "Mitten\rK";
```

```
Kitten
```

مثلاً در مثال بالا کاراکتر K بعد از کاراکتر کنترلی \r آمده است. کاراکتر کنترلی حرف K را به ابتدای سطر برده و جایگزین حرف M می‌کند. برای چاپ کاراکترهای یونیکد می‌توان از \u استفاده کرد. برای استفاده از \u ، مقدار در مبنای 16 کاراکتر را درست بعد از علامت \u قرار می‌دهیم. برای مثال اگر بخواهیم علامت (T-) را چاپ کنیم باید بعد از علامت \u مقدار 00A9 را قرار دهیم مانند :

```
cout << "\u00A9";
```

```
T-
```

برای مشاهده لیست مقادیر مبنای 16 برای کاراکترهای یونیکد به لینک زیر مراجعه نمایید :

<http://www.ascii.cl/htmlcodes.htm>

اگر کامپایلر به یک کاراکتر کنترلی غیر مجاز برخورد کند، برنامه پیغام خطا می‌دهد. بیشترین خطا زمانی اتفاق می‌افتد که برنامه نویس برای چاپ اسلش (\) از \\ استفاده می‌کند. برای دریافت اطلاعات بیشتر در مورد کاراکترهای کنترلی به لینک زیر مراجعه کنید :

<https://msdn.microsoft.com/en-us/library/h21280bw.aspx>

متغیر

متغیر مکانی از حافظه است که شما می‌توانید مقادیری را در آن ذخیره کنید. می‌توان آن را به عنوان یک ظرف تصور کرد که داده‌های خود را در آن قرار داده‌اید. محتویات این ظرف می‌تواند پاک شود یا تغییر کند. هر متغیر دارای یک نام نیز هست. که از طریق آن می‌توان متغیر را از دیگر متغیرها تشخیص داد و به مقدار آن دسترسی پیدا کرد. همچنین دارای یک مقدار می‌باشد که می‌تواند توسط کاربر انتخاب شده باشد یا نتیجه یک محاسبه باشد. مقدار متغیر می‌تواند تهی نیز باشد. متغیر دارای نوع نیز هست بدین معنی که نوع آن با نوع داده‌ای که در آن ذخیره می‌شود یکی است. متغیر دارای عمر نیز هست که از روی آن می‌توان تشخیص داد که متغیر باید چقدر در طول برنامه مورد استفاده قرار گیرد. و در نهایت متغیر دارای محدوده استفاده نیز هست که به شما می‌گوید که متغیر در چه جای برنامه برای شما قابل دسترسی است. ما از متغیرها به عنوان یک انبار موقتی برای ذخیره داده استفاده می‌کنیم. هنگامی که یک برنامه ایجاد می‌کنیم احتیاج به یک مکان برای ذخیره داده، مقادیر یا داده‌هایی که توسط کاربر وارد می‌شوند داریم. ایم مکان همان متغیر است. برای این از کلمه متغیر استفاده می‌شود چون ما می‌توانیم بسته به نوع شرایط هر جا که لازم باشد مقدار آن را تغییر دهیم. متغیرها موقتی هستند و فقط موقعی مورد استفاده قرار می‌گیرند که برنامه در حال اجراست و وقتی شما برنامه را می‌بندید محتویات متغیرها نیز پاک می‌شود. قبلاً ذکر شد که به وسیله نام متغیر می‌توان به آن دسترسی پیدا کرد. برای نامگذاری متغیرها باید قوانین زیر را رعایت کرد :

- نام متغیر باید با یک از حروف الفبا (a-z or A-Z) شروع شود.
- نمی‌تواند شامل کاراکترهای غیرمجاز مانند #، ؟، \$ و . باشد.
- نمی‌توان از کلمات رزرو شده در C++ برای نام متغیر استفاده کرد.
- نام متغیر نباید دارای فضای خالی (spaces) باشد.
- اسامی متغیرها نسبت به بزرگی و کوچکی حروف حساس هستند. در C++ دو حرف مانند a و A دو کاراکتر مختلف به حساب می‌آیند.

دو متغیر با نامهای myNumber و MyNumber دو متغیر مختلف محسوب می‌شوند چون یکی از آن‌ها با حرف کوچک m و دیگری با حرف بزرگ M شروع می‌شود. شما نمی‌توانید دو متغیر را که دقیق شبیه هم هستند را در یک scope (محدوده) تعریف کنید. Scope به معنای یک بلوک کد است که متغیر در آن قابل دسترسی و استفاده است. در مورد Scope در فصل‌های آینده بیشتر توضیح

خواهیم داد. متغیر دارای نوع هست که نوع داده‌ای را که در خود ذخیره می‌کند را نشان می‌دهد. معمول‌ترین انواع داده `int`، `double`، `string`، `char` و `float` می‌باشند. برای مثال شما برای قرار دادن یک عدد صحیح در متغیر باید از نوع `int` استفاده کنید.

انواع ساده

انواع ساده انواعی از داده‌ها هستند که شامل اعداد، کاراکترها و رشته‌ها و مقادیر بولی می‌باشند. به انواع ساده انواع اصلی نیز گفته می‌شود چون از آن‌ها برای ساخت انواع پیچیده‌تری مانند کلاس‌ها و ساختارها استفاده می‌شود. انواع ساده دارای مجموعه مشخصی از مقادیر هستند و محدوده خاصی از اعداد را در خود ذخیره می‌کنند. در C++ هفت نوع داده وجود دارد که در جدول زیر ذکر شده‌اند :

کلمه کلیدی	نوع
<code>bool</code>	<code>Boolean</code>
<code>char</code>	<code>Character</code>
<code>int</code>	<code>Integer</code>
<code>float</code>	<code>Floating point</code>
<code>double</code>	<code>Double floating point</code>
<code>void</code>	<code>Valueless</code>
<code>wchar_t</code>	<code>Wide character</code>

انواع بالا (به جز `void`) می‌توانند با عباراتی مثل `signed`، `long`، `unsigned` و `short` ترکیب شده و نوع‌های دیگری را به وجود آورند :

نوع	مقدار فضایی که از حافظه اشغال می‌کند	محدوده
<code>char</code>	<code>1byte</code>	<code>-128 to 127 or 0 to 255</code>
<code>unsigned char</code>	<code>1byte</code>	<code>0 to 255</code>
<code>signed char</code>	<code>1byte</code>	<code>-128 to 127</code>
<code>int</code>	<code>4bytes</code>	<code>-2147483648 to 2147483647</code>
<code>unsigned int</code>	<code>4bytes</code>	<code>0 to 4294967295</code>

signed int	4bytes	-2147483648 to 2147483647
short int	2bytes	-32768 to 32767
unsigned short int	2bytes	0 to 65,535
signed short int	2bytes	-32768 to 32767
long int	8bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
signed long int	8bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long int	8bytes	0 to 18,446,744,073,709,551,615
float	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	8bytes	+/- 1.7e +/- 308 (~15 digits)
wchar_t	2 or 4 bytes	1 wide character

نوع char برای ذخیره کاراکترهای یونیکد استفاده می‌شود. کاراکترها باید داخل یک کوتیشن ساده قرار بگیرند مانند ('a').

نوع bool فقط می‌تواند مقادیر درست (true) یا نادرست (false) را در خود ذخیره کند و بیشتر در برنامه‌هایی که دارای ساختار تصمیم‌گیری هستند مورد استفاده قرار می‌گیرد. نوع string برای ذخیره گروهی از کاراکترها مانند یک پیغام استفاده می‌شود.

مقادیر ذخیره شده در یک رشته باید داخل دابل کوتیشن قرار گیرند تا توسط کامپایلر به عنوان یک رشته در نظر گرفته شوند مانند ("message").

استفاده از متغیرها

در مثال زیر نحوه تعریف و مقدار دهی متغیرها نمایش داده شده است :

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

int main()
{
```

```

//Declare variables
int    num1;
int    num2;
double num3;
double num4;
bool   boolVal;
char   myChar;
string message;

//Assign values to variables
num1   = 1;
num2   = 2;
num3   = 3.54;
num4   = 4.12;
boolVal = true;
myChar  = 'R';
message = "Hello World!";

//Show the values of the variables
cout << "num1   = " << num1   << endl;
cout << "num2   = " << num2   << endl;
cout << "num3   = " << num3   << endl;
cout << "num4   = " << num4   << endl;
cout << "boolVal = " << boolVal << endl;
cout << "myChar  = " << myChar  << endl;
cout << "message = " << message << endl;

}

```

```

num1   = 1
num2   = 2
num3   = 3.54
num4   = 4.12
boolVal = 1
myChar  = R
message = Hello World!

```

تعریف متغیر

در کد بالا متغیرهایی با نوع و نام متفاوت تعریف شده‌اند. ابتدا باید نوع داده‌هایی را که این متغیرها قرار است در خود ذخیره کنند را مشخص کنیم و سپس یک نام برای آن‌ها در نظر بگیریم و در آخر سیمیکولن بگذاریم. همیشه به یاد داشته باشید که قبل از مقدار دهی و استفاده از متغیر باید آن را تعریف کرد. شاید برایتان این سؤال پیش آمده باشد که کاربرد endl چیست؟ endl برای ایجاد خط جدید مورد استفاده قرار گرفته است. یعنی نشانگر ماوس را همانند کاراکتر کنترلی \n به خط بعد می‌برد، در نتیجه خروجی کد بالا در خطوط جداگانه چاپ می‌شود.

```

//Declare variables
int    num1;

```



```
int    num2;
double num3;
double num4;
bool   boolVal;
char   myChar;
string message;
```

نحوه تعریف متغیر به صورت زیر است:

```
data_type identifier;
```

date_type همان نوع داده است مانند int، double و identifier. ... نیز نام متغیر است که به ما امکان استفاده و دسترسی به مقدار متغیر را می‌دهد. برای تعریف چند متغیر از یک نوع می‌توان به صورت زیر عمل کرد :

```
data_type identifier1, identifier2, ... identifierN;
```

مثال

```
int num1, num2, num3, num4, num5;
string message1, message2, message3;
```

در مثال بالا 5 متغیر از نوع صحیح و 3 متغیر از نوع رشته تعریف شده است. توجه داشته باشید که بین متغیرها باید علامت کاما (,) باشد.

نامگذاری متغیرها

- نام متغیر باید با یک حرف یا زیرخط و به دنبال آن حرف یا عدد شروع شود.
- نمی‌توان از کاراکترهای خاص مانند #، %، & یا عدد برای شروع نام متغیر استفاده کرد مانند 2numbers.
- نام متغیر نباید دارای فاصله باشد. برای نام‌های چند حرفی می‌توان به جای فاصله از علامت زیرخط یا _ استفاده کرد.

نامهای مجاز :

num1	myNumber	studentCount	total	first_name	_minimum
num2	myChar	average	amountDue	last_name	_maximum
name	counter	sum	isLeapYear	color_of_car	_age

نامهای غیر مجاز :

123	#numbers#	#ofstudents	1abc2
123abc	\$money	first name	ty.np
my number	this&that	last name	1:00

اگر به نامهای مجاز در مثال بالا توجه کنید متوجه قراردادهای به کار رفته در نامگذاری آنها خواهید شد. یکی از روشهای نامگذاری، نامگذاری کوهان شتری است. در این روش که برای متغیرهای دو کلمه‌ای به کار می‌رود، اولین حرف اولین کلمه با حرف کوچک و اولین حرف دومین کلمه با حرف بزرگ نمایش داده می‌شود مانند myNumber. توجه کنید که اولین حرف کلمه Number با حرف بزرگ شروع شده است. مثال دیگر کلمه numberOfStudents است. اگر توجه کنید بعد از اولین کلمه حرف اول سایر کلمات با حروف بزرگ نمایش داده شده است.

محدوده متغیر

متغیرهای ابتدای درس در داخل متد main() تعریف شده‌اند. در نتیجه این متغیرها فقط در داخل متد main() قابل دسترسی هستند. محدوده یک متغیر مشخص می‌کند که متغیر در کجای کد قابل دسترسی است. هنگامیکه برنامه به پایان متد main() می‌رسد متغیرها از محدوده خارج و بدون استفاده می‌شوند تا زمانی که برنامه در حال اجراست. محدوده متغیرها انواعی دارد که در درس‌های بعدی با آنها آشنا می‌شوید. تشخیص محدوده متغیر بسیار مهم است چون به وسیله آن می‌فهمید که در کجای کد می‌توان از متغیر استفاده کرد. باید یاد آور شد که دو متغیر در یک محدوده نمی‌توانند دارای نام یکسان باشند. مثلاً کد زیر در برنامه ایجاد خطا می‌کند :

```
int num1;
int num1;
```

از آنجاییکه C++ به بزرگی و کوچکی حروف حساس است می‌توان از این خاصیت برای تعریف چند متغیر هم نام ولی با حروف متفاوت (از لحاظ بزرگی و کوچکی) برای تعریف چند متغیر از یک نوع استفاده کرد مانند :

```
int num1;
int Num1;
int NUM1;
```

مقداردهی متغیرها

می‌توان فوراً بعد از تعریف متغیرها مقادیری را به آنها اختصاص داد. این عمل را مقداردهی می‌نامند. در زیر نحوه مقدار دهی متغیرها نشان داده شده است :

```
data_type identifier = value;
```

به عنوان مثال :

```
int myNumber = 7;
```

همچنین می‌توان چندین متغیر را فقط با گذاشتن کاما بین آن‌ها به سادگی مقدار دهی کرد :

```
data_type variable1 = value1, variable2 = value2, ... variableN, valueN;
int num1 = 1, num2 = 2, num3 = 3;
```

تعریف متغیر با مقدار دهی متغیرها متفاوت است. تعریف متغیر یعنی انتخاب نوع و نام برای متغیر ولی مقدار دهی یعنی اختصاص یک مقدار به متغیر.

اختصاص مقدار به متغیر

در زیر نحوه اختصاص مقادیر به متغیرها نشان داده شده است:

```
num1    = 1;
num2    = 2;
num3    = 3.54;
num4    = 4.12;
boolVal = true;
myChar  = 'R';
message = "Hello World!";
```

به این نکته توجه کنید که شما به مغیری که هنوز تعریف نشده نمی‌توانید مقدار بدهید. شما فقط می‌توانید از متغیرهایی استفاده کنید که هم تعریف و هم مقدار دهی شده باشند. مثلاً متغیرهای بالا همه قابل استفاده هستند. در این مثال num1 و num2 هر دو تعریف شده‌اند و مقادیری از نوع صحیح به آن‌ها اختصاص داده شده است. اگر نوع داده با نوع متغیر یکی نباشد برنامه پیغام خطا می‌دهد.

ثابت

ثابت‌ها انواعی هستند که مقدار آن‌ها در طول برنامه تغییر نمی‌کند. ثابت‌ها حتماً باید مقدار دهی اولیه شوند و اگر مقدار دهی آن‌ها فراموش شود در برنامه خطا به وجود می‌آید. بعد از این که به ثابت‌ها مقدار اولیه اختصاص داده شد هرگز در زمان اجرای برنامه نمی‌توان آن را تغییر داد. برای تعریف ثابت‌ها باید از کلمه کلیدی const و #define استفاده کرد. معمولاً نام ثابت‌ها را طبق قرارداد با حروف بزرگ می‌نویسند تا تشخیص آن‌ها در برنامه راحت باشد. نحوه تعریف ثابت در زیر آمده است :

```
const data_type identifier = initial_value;
```

در کد بالا ابتدا کلمه کلیدی const و سپس نوع ثابت و بعد نام ثابت را با حروف بزرگ می‌نویسیم. و در نهایت یک مقدار را به آن اختصاص می‌دهیم و علامت سمیکالن می‌گذاریم.

```
#define data_type identifier initial_value
```

در روش بالا فقط #define را نوشته و سپس نام ثابت و بعد مقداری که قرار است دریافت کند. به این نکته توجه کنید که در روش بالا نه علامت سمیکالن وجود دارد و نه علامت مساوی. مثال :

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

int main()
{
    const int NUMBER = 1;

    NUMBER = 20; //ERROR, Cant modify a constant

    cout << NUMBER;
}
```

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

int main()
{
    #define NUMBER 1

    NUMBER = 20; //ERROR, Cant modify a constant

    cout << NUMBER;
}
```

در این مثال می بینید که مقدار دادن به یک ثابت، که قبلاً مقدار دهی شده برنامه را با خطا مواجه می کند. نکته دیگری که نباید فراموش شود این است که نباید مقدار ثابت را با مقدار دیگر متغیرهای تعریف شده در برنامه برابر قرار داد. مثال :

```
int someVariable;
const int MY_CONST = someVariable;
```

ممکن است این سؤال برایتان پیش آمده باشد که دلیل استفاده از ثابت ها چیست؟ اگر مطمئن هستید که مقادیری در برنامه وجود دارند که هرگز در طول برنامه تغییر نمی کنند بهتر است که آنها را به صورت ثابت تعریف کنید. این کار هر چند کوچک کیفیت برنامه شما را بالا می برد.

عبارات و عملگرها

ابتدا با دو کلمه آشنا شوید :

- عملگر: نمادهایی هستند که اعمال خاص انجام می‌دهند.
- عملوند: مقادیری که عملگرها بر روی آنها عملی انجام می‌دهند.

مثلاً $X+Y$: یک عبارت است که در آن X و Y عملوند و علامت $+$ عملگر به حساب می‌آیند.

زبانهای برنامه نویسی جدید دارای عملگرهایی هستند که از اجزاء معمول زبان به حساب می‌آیند. C++ دارای عملگرهای مختلفی از جمله عملگرهای ریاضی، تخصیصی، مقایسه‌ای، منطقی و بیتی می‌باشد. از عملگرهای ساده ریاضی می‌توان به عملگر جمع و تفریق اشاره کرد. سه نوع عملگر در C++ وجود دارد :

- یگانی (Unary) - به یک عملوند نیاز دارد.
- دودویی (Binary) - به دو عملوند نیاز دارد.
- سه تایی (Ternary) - به سه عملوند نیاز دارد.

انواع مختلف عملگر که در این بخش مورد بحث قرار می‌گیرند عبارت‌اند از :

- عملگرهای ریاضی
- عملگرهای تخصیصی
- عملگرهای مقایسه‌ای
- عملگرهای منطقی
- عملگرهای بیتی
- عملگرهای ریاضی

عملگرهای ریاضی

C++ از عملگرهای ریاضی برای انجام محاسبات استفاده می‌کند. جدول زیر عملگرهای ریاضی سی پلاس پلاس را نشان می‌دهد :

عملگر	دسته	مثال	نتیجه
+	Binary	<code>var1 = var2 + var3;</code>	Var1 برابر است با حاصل جمع var2 و var3
-	Binary	<code>var1 = var2 - var3;</code>	Var1 برابر است با حاصل تفریق var2 و var3

var1 برابر است با حاصلضرب var2 در var3	var1 = var2 * var3;	Binary	*
var1 برابر است با حاصل تقسیم var2 بر var3	var1 = var2 / var3;	Binary	/
var1 برابر است با باقیمانده تقسیم var2 و var3	var1 = var2 % var3;	Binary	%
var1 برابر است با مقدار var2	var1 = +var2;	Unary	+
var1 برابر است با مقدار var2 ضربدر -1	var1 = -var2	Unary	-

مثال بالا در از نوع عددی استفاده شده است. اما استفاده از عملگرهای ریاضی برای نوع رشته‌ای نتیجه متفاوتی دارد. همچنین در جمع دو کاراکتر کامپایلر معادل عددی آن‌ها را نشان می‌دهد. دیگر عملگرهای C++ عملگرهای کاهش و افزایش هستند. این عملگرها مقدار 1 را از متغیرها کم یا به آن‌ها اضافه می‌کنند. از این متغیرها اغلب در حلقه‌ها استفاده می‌شود :

عملگر	دسته	مثال	نتیجه
++	Unary	var1 = ++var2;	مقدار var1 برابر است با var2 بعلاوه 1
--	Unary	var1 = --var2;	مقدار var1 برابر است با var2 منهای 1
++	Unary	var1 = var2++;	مقدار var1 برابر است با var2. به متغیر var2 یک واحد اضافه می‌شود.
-	Unary	var1 = var2--;	مقدار var1 برابر است با var2. از متغیر var2 یک واحد کم می‌شود.

به این نکته توجه داشته باشید که محل قرار گیری عملگر در نتیجه محاسبات تأثیر دارد. اگر عملگر قبل از متغیر var2 بیاید افزایش یا کاهش var1 اتفاق می‌افتد. چنانچه عملگرها بعد از متغیر var2 قرار بگیرند ابتدا var1 برابر var2 می‌شود و سپس متغیر var2 افزایش یا کاهش می‌یابد. به مثال‌های زیر توجه کنید :

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

int main()
{
    int x = 0;
    int y = 1;

    x = ++y;
```

```
cout << "x=" << x << endl;
cout << "y=" << y << endl;
}
```

```
x=2
y=2
```

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

int main()
{
    int x = 0;
    int y = 1;

    x = --y;

    cout << "x=" << x << endl;
    cout << "y=" << y << endl;
}
```

```
x=0
y=0
```

همانطور که در دو مثال بالا مشاهده می‌کنید، درج عملگرهای ++ و -- قبل از عملوند y باعث می‌شود که ابتدا یک واحد از y کم و یا یک واحد به y اضافه شود و سپس نتیجه در عملوند x قرار بگیرد. حال به دو مثال زیر توجه کنید :

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

int main()
{
    int x = 0;
    int y = 1;

    x = y--;

    cout << "x=" << x << endl;
    cout << "y=" << y << endl;
}
```

```
x=1
y=0
```

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

int main()
```

```
{
    int x = 0;
    int y = 1;

    x = y++;

    cout << "x=" << x << endl;
    cout << "y=" << y << endl;
}
```

```
x=1
y=2
```

همانطور که در دو مثال بالا مشاهده می‌کنید، درج عملگرهای ++ و -- بعد از عملوند y باعث می‌شود که ابتدا مقدار y در داخل متغیر x قرار بگیرد و سپس یک واحد از y کم و یا یک واحد به آن اضافه شود. حال می‌توانیم با ایجاد یک برنامه نحوه عملکرد عملگرهای ریاضی در ++C را یاد بگیریم :

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

int main()
{
    //Variable declarations
    int num1, num2;
    string msg1, msg2;

    //Assign test values
    num1 = 6;
    num2 = 3;

    //Demonstrate use of mathematical operators
    cout << "The sum of num1 and num2 is " << (num1 + num2) << endl;
    cout << "The difference of num1 and num2 is " << (num1 - num2) << endl;
    cout << "The product of num1 and num2 is " << (num1 * num2) << endl;
    cout << "The quotient of num1 and num2 is " << (num1 / num2) << endl;
    cout << "The remainder of num1 and num2 is " << (num1 % num2) << endl;

    msg1 = "Hello ";
    msg2 = "World!";
    cout << msg1 + msg2;
}
```

```
The sum of 6 and 3 is 9.
The difference of 6 and 3 is 3.
The product of 6 and 3 is 18.
The quotient of 6 and 3 is 2.
The remainder of 6 divided by 3 is 0
Hello World!
```

برنامه بالا نتیجه هر عبارت را نشان می‌دهد. در این برنامه از endl برای نشان دادن نتایج در سطریهای متفاوت استفاده شده است. ++C خط جدید و فاصله و فضای خالی را نادیده می‌گیرد. در خط 22 مشاهده می‌کنید که دو رشته به وسیله عملگر + به هم متصل

شده‌اند. نتیجه استفاده از عملگر + برای چسباندن دو کلمه "Hello" و "World!" رشته "Hello World!" خواهد بود. به فاصله‌های خالی بعد از اولین کلمه توجه کنید اگر آن‌ها را حذف کنید از خروجی برنامه نیز حذف می‌شوند.

عملگرهای تخصیصی

نوع دیگر از عملگرهای C++ عملگرهای جایگزینی نام دارند. این عملگرها مقدار متغیر سمت راست خود را در متغیر سمت چپ قرار می‌دهند. جدول زیر انواع عملگرهای تخصیصی در C++ را نشان می‌دهد:

عملگر	مثال	نتیجه
=	var1 = var2;	مقدار var1 برابر است با مقدار var2
+=	var1 += var2;	مقدار var1 برابر است با حاصل جمع var1 و var2
-=	var1 -= var2;	مقدار var1 برابر است با حاصل تفریق var1 و var2
*=	var1 *= var2;	مقدار var1 برابر است با حاصل ضرب var1 در var2
/=	var1 /= var2;	مقدار var1 برابر است با حاصل تقسیم var1 بر var2
%=	var1 %= var2;	مقدار var1 برابر است با باقیمانده تقسیم var1 بر var2

استفاده از این نوع عملگرها در واقع یک نوع خلاصه نویسی در کد است. مثلاً شکل اصلی کد `var1 += var2` به صورت `var1 = var1 + var2` می‌باشد. این حالت کدنویسی زمانی کارایی خود را نشان می‌دهد که نام متغیرها طولانی باشد. برنامه زیر چگونگی استفاده از عملگرهای تخصیصی و تأثیر آن‌ها را بر متغیرها نشان می‌دهد.

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

int main()
{
    int number;

    cout << "Assigning 10 to number..." << endl;
    number = 10;
    cout << "Number = " << number << endl;

    cout << "Adding 10 to number..." << endl;
    number += 10;
    cout << "Number = " << number << endl;
```

```
cout << "Subtracting 10 from number..." << endl;
number -= 10;
cout << "Number = " << number << endl;
}
```

```
Assigning 10 to number...
Number = 10
Adding 10 to number...
Number = 20
Subtracting 10 from number...

Number = 10
```

در برنامه از 3 عملگر تخصیصی استفاده شده است. ابتدا یک متغیر و مقدار 10 با استفاده از عملگر = به آن اختصاص داده شده است. سپس به آن با استفاده از عملگر += مقدار 10 اضافه و در آخر به وسیله عملگر -= عدد 10 از آن کم شده است.

عملگرهای مقایسه‌ای

از عملگرهای مقایسه‌ای برای مقایسه مقادیر استفاده می‌شود. نتیجه این مقادیر یک مقدار بولی (منطقی) است. این عملگرها اگر نتیجه مقایسه دو مقدار درست باشد مقدار 1 و اگر نتیجه مقایسه اشتباه باشد مقدار 0 را نشان می‌دهند. این عملگرها به طور معمول در دستورات شرطی به کار می‌روند به این ترتیب که باعث ادامه یا توقف دستور شرطی می‌شوند. جدول زیر عملگرهای مقایسه‌ای در C++ را نشان می‌دهد:

عملگر	دسته	مثال	نتیجه
==	Binary	var1 = var2 == var3	var1 در صورتی 1 است که مقدار var2 با مقدار var3 برابر باشد در غیر اینصورت 0 است
!=	Binary	var1 = var2 != var3	var1 در صورتی 1 است که مقدار var2 با مقدار var3 برابر نباشد در غیر اینصورت 0 است
<	Binary	var1 = var2 < var3	var1 در صورتی 1 است که مقدار var2 کوچک‌تر از var3 مقدار باشد در غیر اینصورت 0 است
>	Binary	var1 = var2 > var3	var1 در صورتی 1 است که مقدار var2 بزرگ‌تر از مقدار var3 باشد در غیر اینصورت 0 است

var1 در صورتی 1 است که مقدار var2 کوچکتر یا مساوی مقدار var3 باشد در غیر اینصورت 0 است	var1 = var2 <= var3	Binary	<=
var1 در صورتی 1 است که مقدار var2 بزرگتر یا مساوی var3 مقدار باشد در غیر اینصورت 0 است	var1 = var2 >= var3	Binary	>=

برنامه زیر نحوه عملکرد این عملگرها را نشان می‌دهد :

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

int main()
{
    int num1 = 10;
    int num2 = 5;

    cout << num1 << " == " << num2 << " : " << (num1 == num2) << endl;
    cout << num1 << " != " << num2 << " : " << (num1 != num2) << endl;
    cout << num1 << " < " << num2 << " : " << (num1 < num2) << endl;
    cout << num1 << " > " << num2 << " : " << (num1 > num2) << endl;
    cout << num1 << " <= " << num2 << " : " << (num1 <= num2) << endl;
    cout << num1 << " >= " << num2 << " : " << (num1 >= num2) << endl;
}
```

```
10 == 5 : 0
10 != 5 : 1
10 < 5 : 0
10 > 5 : 1
10 <= 5 : 0
10 >= 5 : 1
```

در مثال بالا ابتدا دو متغیر را که می‌خواهیم با هم مقایسه کنیم را ایجاد کرده و به آن‌ها مقادیری اختصاص می‌دهیم. سپس با استفاده از یک عملگر مقایسه‌ای آن‌ها را با هم مقایسه کرده و نتیجه را چاپ می‌کنیم. به این نکته توجه کنید که هنگام مقایسه دو متغیر از عملگر == به جای عملگر = باید استفاده شود. عملگر = عملگر تخصیصی است و در عبارتی مانند $x = y$ مقدار y را در به x اختصاص می‌دهد. عملگر == عملگر مقایسه‌ای است که دو مقدار را با هم مقایسه می‌کند مانند $x == y$ و اینطور خوانده می‌شود x برابر است با y .

عملگرهای منطقی

عملگرهای منطقی بر روی عبارات منطقی عمل می کنند و نتیجه آن ها نیز یک مقدار بولی است. از این عملگرها اغلب برای شرطهای پیچیده استفاده می شود. همانطور که قبلاً یاد گرفتید مقادیر بولی می توانند false یا true باشند. فرض کنید که var2 و var3 دو مقدار بولی هستند.

عملگر	نام	دسته	مثال
&&	منطقی AND	Binary	var1 = var2 && var3;
	منطقی OR	Binary	var1 = var2 var3;
!	منطقی NOT	Unary	var1 = !var1;

عملگر منطقی AND(&&)

اگر مقادیر دو طرف عملگر AND ، true باشند عملگر AND مقدار true را بر می گرداند. در غیر اینصورت اگر یکی از مقادیر یا هر دوی آن ها false باشند مقدار false را بر می گرداند. در زیر جدول درستی عملگر AND نشان داده شده است :

X	Y	X && Y
true	true	true
true	false	false
false	true	false
false	false	false

برای درک بهتر تأثیر عملگر AND یاد آوری می کنم که این عملگر فقط در صورتی مقدار true را نشان می دهد که هر دو عملوند مقدارشان true باشد. در غیر اینصورت نتیجه تمام ترکیب های بعدی false خواهد شد. استفاده از عملگر AND مانند استفاده از عملگرهای مقایسه ای است. به عنوان مثال نتیجه عبارت زیر درست (true) است اگر سن (age) بزرگ تر از 18 و salary کوچک تر از 1000 باشد.

```
result = (age > 18) && (salary < 1000);
```

عملگر AND زمانی کارآمد است که ما با محدود خاصی از اعداد سرو کار داریم. مثلاً عبارت $100 \leq x \leq 10$ بدین معنی است که x می‌تواند مقداری شامل اعداد 10 تا 100 را بگیرد. حال برای انتخاب اعداد خارج از این محدوده می‌توان از عملگر منطقی AND به صورت زیر استفاده کرد.

```
inRange = (number <= 10) && (number >= 100);
```

عملگر منطقی (||) OR

اگر یکی یا هر دو مقدار دو طرف عملگر OR، درست (true) باشد، عملگر OR مقدار true را بر می‌گرداند. جدول درستی عملگر OR در زیر نشان داده شده است:

X	Y	X Y
true	true	true
true	false	true
false	true	true
false	false	false

در جدول بالا مشاهده می‌کنید که عملگر OR در صورتی مقدار false را بر می‌گرداند که مقادیر دو طرف آن false باشند. کد زیر را در نظر بگیرید. نتیجه این کد در صورتی درست (true) است که رتبه نهایی دانش آموز (finalGrade) بزرگ‌تر از 75 یا یا نمره نهایی امتحان آن 100 باشد.

```
isPassed = (finalGrade >= 75) || (finalExam == 100);
```

عملگر منطقی (!) NOT

برخلاف دو اپراتور OR و AND عملگر منطقی NOT یک عملگر یگانی است و فقط به یک عملوند نیاز دارد. این عملگر یک مقدار یا اصطلاح بولی را نفی می‌کند. مثلاً اگر عبارت یا مقدار true باشد آنرا false و اگر false باشد آنرا true می‌کند. جدول زیر عملکرد اپراتور NOT را نشان می‌دهد:

X	!X
true	false
false	true

```
isMinor = !(age >= 18);
```

شما سیستم باینری و نحوه تبدیل اعداد دهدهی به باینری را از لینک زیر یاد بگیرید:

<http://www.w3-farsi.com/?p=5698>

در سیستم باینری (دودویی) که کامپیوتر از آن استفاده می‌کند وضعیت هر چیز یا خاموش است یا روشن. برای نشان دادن حالت روشن از عدد 1 و برای نشان دادن حالت خاموش از عدد 0 استفاده می‌شود. بنابراین اعداد باینری فقط می‌توانند صفر یا یک باشند. اعداد باینری را اعداد در مبنای 2 و اعداد اعشاری را اعداد در مبنای 10 می‌گویند. یک بیت نشان دهنده یک رقم باینری است و هر بایت نشان دهنده 8 بیت است. به عنوان مثال برای یک داده از نوع int به 32 بیت یا 4 بایت فضا برای ذخیره آن نیاز داریم، این بدین معناست که اعداد از 32 رقم 0 و 1 برای ذخیره استفاده می‌کنند. برای مثال عدد 100 وقتی به عنوان یک متغیر از نوع int ذخیره می‌شود در کامپیوتر به صورت زیر خوانده می‌شود :

```
000000000000000000000000000000000000000000000000000
```

عدد 100 در مبنای ده معادل عدد 1100100 در مبنای 2 است. در اینجا 7 رقم سمت راست نشان دهنده عدد 100 در مبنای 2 است و مابقی صفرهای سمت راست برای پر کردن بیت‌هایی است که عدد از نوع int نیاز دارد. به این نکته توجه کنید که اعداد باینری از سمت راست به چپ خوانده می‌شوند. عملگرهای بیتی C++ در جدول زیر نشان داده شده‌اند :

عملگر	نام	دسته	مثال
&	بیتی AND	Binary	x = y & z;
	بیتی OR	Binary	x = y z;
^	بیتی XOR	Binary	x = y ^ z;
~	بیتی NOT	Unary	x = ~y;
&=	بیتی - تخصیصی AND	Binary	x &= y;

$x \mid= y;$	Binary	OR - تخبیصی	$\mid=$
$x \wedge= y;$	Binary	XOR - تخبیصی	$\wedge=$

X	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

ابتدا دو عدد 5 و 3 به معادل باینری‌شان تبدیل می‌شوند. از آنجاییکه هر عدد صحیح (int) 32 بیت است از صفر برای پر کردن بیت‌های خالی استفاده می‌کنیم. با استفاده از جدول درستی عملگر بیتی AND می‌توان فهمید که چرا نتیجه عدد یک می‌شود.

X	Y	X XOR Y
1	1	0
1	0	1
0	1	1
0	0	0

در صورتیکه عملوندهای دو طرف این عملگر هر دو صفر یا هر دو یک باشند نتیجه صفر در غیر اینصورت نتیجه یک می‌شود. در مثال زیر تأثیر عملگر بیتی XOR را بر روی دو مقدار مشاهده می‌کنید :

```
int result = 5 ^ 7;
```

```
cout << result;
```

2

در زیر معادل باینری اعداد بالا (5 و 7) نشان داده شده است.

```
5: 00000000000000000000000000000000000000000000000000000
```

```
7: 000000000000000000000000000000000000111
```

```
-----  
2: 0000000000000000000000000000000000000000000000000
```

با نگاه کردن به جدول درستی عملگر بیتی XOR، می‌توان فهمید که چرا نتیجه عدد 2 می‌شود.

عملگر پیتی (~) NOT

این عملگر یک عملگر یگانی است و فقط به یک عملوند نیاز دارد. در زیر جدول درستی این عملگر آمده است:

	X	NOT X
1	1	0
0	0	1

عملگر بیتی NOT مقادیر بیت‌ها را معکوس می‌کند. در زیر چگونگی استفاده از این عملگر آمده است :

```
int result = ~7;
```

```
cout << result;
```

-8

```
-8: 111111111111111111111111111000
```

مشاهده می‌کنید که همه بیت‌ها به اندازه دو واحد به سمت چپ منتقل شده‌اند. در این انتقال دو صفر از صفرهای سمت چپ کم می‌شود و در عوض دو صفر به سمت راست اضافه می‌شود.

اگر ما حق تقدم عملگرها را رعایت نکنیم و عبارت بالا را از سمت چپ به راست انجام دهیم نتیجه 9 خواهد شد ($1+2=3$) سپس $3 \times 3=9$ و در آخر $9/1=9$). اما کامپایلر با توجه به تقدم عملگرها محاسبات را انجام می‌دهد. برای مثال عمل ضرب و تقسیم نسبت به جمع و تفریق تقدم دارند. بنابراین در مثال فوق ابتدا عدد 2 ضربدر 3 و سپس نتیجه آن‌ها تقسیم بر 1 می‌شود که نتیجه 6 به دست می‌آید. در آخر عدد 6 با 1 جمع می‌شود و عدد 7 حاصل می‌شود. در جدول زیر تقدم عملگرهای ++C از بالا به پایین آمده است :

Level	Precedence group	Operator	Grouping
1	Scope	::	Left-to-right
2	Postfix (unary)	++ -	Left-to-right
		()	
		[]	
		. ->	
3	Prefix (unary)	++ -	Right-to-left
		~ !	
		+ -	
		& *	
		new delete	
		sizeof	
		(type)	
4	Pointer-to-member	.* ->*	Left-to-right
5	Aritdmetic: scaling	* / %	Left-to-right
6	Aritdmetic: addition	+ -	Left-to-right
7	Bitwise shift	<< >>	Left-to-right
8	Relational	< > <= >=	Left-to-right
9	Equality	== !=	Left-to-right
10	And	&	Left-to-right
11	Exclusive or	^	Left-to-right
12	Inclusive or		Left-to-right
13	Conjunction	&&	Left-to-right
14	Disjunction		Left-to-right
15	Assignment-level expressions	= *= /= %= += -= >>= <<= &= ^= =	Right-to-left
		?:	
16	Sequencing	,	Left-to-right

ابتدا عملگرهای با بالاترین و سپس عملگرهای با پایین‌ترین حق تقدم در محاسبات تأثیر می‌گذارند. به این نکته توجه کنید که تقدم عملگرها ++ و - به مکان قرارگیری آن‌ها بستگی دارد (در سمت چپ یا راست عملوند باشند). به عنوان مثال :

```
int number = 3;

number1 = 3 + ++number; //results to 7
number2 = 3 + number++; //results to 6
```

در عبارت اول ابتدا به مقدار number یک واحد اضافه شده و 4 می‌شود و سپس مقدار جدید با عدد 3 جمع می‌شود و در نهایت عدد 7 به دست می‌آید. در عبارت دوم مقدار عددی 3 به مقدار number اضافه می‌شود و عدد 6 به دست می‌آید. سپس این مقدار در متغیر number2 قرار می‌گیرد. و در نهایت مقدار number به 4 افزایش می‌یابد. برای ایجاد خوانایی در تقدم عملگرها و انجام محاسباتی که در آن‌ها از عملگرهای زیادی استفاده می‌شود از پرانتز استفاده می‌کنیم :

```
number = ( 1 + 2 ) * ( 3 / 4 ) % ( 5 - ( 6 * 7 ) );
```

در مثال بالا ابتدا هر کدام از عباراتی که داخل پرانتز هستند مورد محاسبه قرار می‌گیرند. به نکته‌ای در مورد عبارتی که در داخل پرانتز سوم قرار دارد توجه کنید. در این عبارت ابتدا مقدار داخلی‌ترین پرانتز مورد محاسبه قرار می‌گیرد یعنی مقدار 6 ضربدر 7 شده و سپس از 5 کم می‌شود. اگر دو یا چند عملگر با حق تقدم یکسان موجود باشد ابتدا باید هر کدام از عملگرها را که در ابتدای عبارت می‌آیند مورد ارزیابی قرار دهید. به عنوان مثال :

```
number = 3 * 2 + 8 / 4;
```

هر دو عملگر * و / دارای حق تقدم یکسانی هستند. بنابر این شما باید از چپ به راست آن‌ها را در محاسبات تأثیر دهید. یعنی ابتدا 3 را ضربدر 2 می‌کنید و سپس عدد 8 را بر 4 تقسیم می‌کنید. در نهایت نتیجه دو عبارت را جمع کرده و در متغیر number قرار می‌دهید.

گرفتن ورودی از کاربر

سی پلاس پلاس دارای تعدادی شیء و متد برای گرفتن ورودی از کاربر می‌باشد. حال می‌خواهیم درباره cin یکی دیگر از اشیاء کلاس Istraem بحث کنیم که یک مقدار را از کاربر دریافت می‌کند. کار cin این است که تمام کاراکترهایی را که شما در محیط کنسول تایپ می‌کنید تا زمانی که دکمه Enter را می‌زنید می‌خواند. به برنامه زیر توجه کنید :

```
#include <iostream>
#include <string>
```

```
using namespace std;

int main()
{
    string name;
    int age;
    double height;

    cout << "Enter your name: ";
    cin >> name;
    cout << "Enter your age: ";
    cin >> age;
    cout << "Enter your height: ";
    cin >> height;

    //Print a blank line
    cout << endl;

    //Show the details you typed
    cout << "Name is " << name << endl;
    cout << "Age is " << age << endl;
    cout << "Height is " << height << endl;
}
```

```
Enter your name: John
Enter your age: 18
Enter your height: 160.5
```

```
Name is John.
Age is 18.
Height is 160.5.
```

ابتدا 3 متغیر را برای ذخیره داده در برنامه تعریف می‌کنیم (خطوط 8 و 9 و 10). برنامه از کاربر می‌خواهد که نام خود را وارد کند (خط 12). در خط 13 شما به عنوان کاربر نام خود را وارد می‌کنید. مقدار متغیر نام، برابر مقداری است که توسط cin خوانده می‌شود. از آنجاییکه نام از نوع رشته است باید کتابخانه مربوط به رشته‌ها را در ابتدای برنامه وارد کنیم :

```
#include <string>
```

سپس برنامه از ما سن را سؤال می‌کند (خط 14). آن را در خط 15 وارد کرده و در نهایت در خط 16 و 17 هم قد را وارد می‌کنیم.

ساختارهای تصمیم

تقریباً همه زبانهای برنامه نویسی به شما اجازه اجرای کد را در شرایط مطمئن می‌دهند. حال تصور کنید که یک برنامه دارای ساختار تصمیم‌گیری نباشد و همه کدها را اجرا کند. این حالت شاید فقط برای چاپ یک پیغام در صفحه مناسب باشد ولی فرض کنید که

شما بخواهید اگر مقدار یک متغیر با یک عدد برابر باشد سپس یک پیغام چاپ شود آن وقت با مشکل مواجه خواهید شد. C++ راه‌های مختلفی برای رفع این نوع مشکلات ارائه می‌دهد. در این بخش با مطالب زیر آشنا خواهید شد :

- دستور if
- دستور if...else
- عملگر سه تایی
- دستور if چندگانه
- دستور if تو در تو
- عملگرهای منطقی
- دستور switch

دستور if

می‌توان با استفاده از دستور if و یک شرط خاص که باعث ایجاد یک کد می‌شود یک منطق به برنامه خود اضافه کنید. دستور if ساده‌ترین دستور شرطی است که برنامه می‌گوید اگر شرطی برقرار است کد معینی را انجام بده. ساختار دستور if به صورت زیر است :

```
if (condition)
    code to execute;
```

قبل از اجرای دستور if ابتدا شرط بررسی می‌شود. اگر شرط برقرار باشد یعنی درست باشد سپس کد اجرا می‌شود. شرط یک عبارت مقایسه‌ای است. می‌توان از عملگرهای مقایسه‌ای برای تست درست یا اشتباه بودن شرط استفاده کرد. اجازه بدهید که نگاهی به نحوه استفاده از دستور if در داخل برنامه بیندازیم. برنامه زیر پیغام Hello World را اگر مقدار number کمتر از 10 و Goodbye World را اگر مقدار number از 10 بزرگ‌تر باشد در صفحه نمایش می‌دهد.

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      //Declare a variable and set it a value less than 10
7      int number = 5;
8
9      //If the value of number is less than 10
10     if (number < 10)
11         cout << "Hello World." << endl;
```

```

12
13 //Change the value of a number to a value which is greater than 10
14 number = 15;
15
16 //If the value of number is greater than 10
17 if (number > 10)
18     cout << "Goodbye World.";
19 }

```

```

Hello World.
Goodbye World.

```

در خط 7 یک متغیر با نام number تعریف و مقدار 5 به آن اختصاص داده شده است. وقتی به اولین دستور if در خط 10 می‌رسیم برنامه تشخیص می‌دهد که مقدار number از 10 کمتر است یعنی 5 کوچک‌تر از 10 است.

منطقی است که نتیجه مقایسه درست می‌باشد بنابراین دستور if دستور را اجرا می‌کند (خط 11) و پیغام Hello World چاپ می‌شود. حال مقدار number را به 15 تغییر می‌دهیم (خط 14). وقتی به دومین دستور if در خط 17 می‌رسیم برنامه مقدار number را با 10 مقایسه می‌کند و چون مقدار number یعنی 15 از 10 بزرگ‌تر است برنامه پیغام Goodbye World را چاپ می‌کند (خط 18). به این نکته توجه کنید که دستور if را می‌توان در یک خط نوشت :

```
if (number > 10) cout << "Goodbye World.";
```

شما می‌توانید چندین دستور را در داخل دستور if بنویسید. کافیست که از یک آکولاد برای نشان دادن ابتدا و انتهای دستورات استفاده کنید. همه دستورات داخل بین آکولاد جز بدنه دستور if هستند. نحوه تعریف چند دستور در داخل بدنه if به صورت زیر است :

```

if (condition)
{
    statement1;
    statement2;
    .
    .
    .
    statementN;
}

```

این هم یک مثال ساده :

```

if (x > 10)
{
    cout << "x is greater than 10." << endl;
    cout << "This is still part of the if statement.";
}

```


در مثال بالا اگر مقدار x از 10 بزرگتر باشد دو پیغام چاپ می‌شود. حال اگر به عنوان مثال آکولاد را حذف کنیم و مقدار x از 10 بزرگتر نباشد مانند کد زیر :

```
if (x > 10)
cout << "x is greater than 10." << endl;
cout << "This is still part of the if statement. (Really?);"
```

کد بالا در صورتی بهتر خوانده می‌شود که بین دستورات فاصله بگذاریم.

```
if (x > 10)
cout << "x is greater than 10." << endl;

cout << "This is still part of the if statement. (Really?);"
```

می‌بیند که دستور دوم (خط 3) در مثال بالا جز دستور if نیست. اینجاست که چون ما فرض را بر این گذاشته‌ایم که مقدار x از 10 کوچک‌تر است پس خط (Really?) This is still part of the if statement چاپ می‌شود. در نتیجه اهمیت وجود آکولاد مشخص می‌شود. به عنوان تمرین همیشه حتی اگر فقط یک دستور در بدنه if داشتید برای آن یک آکولاد بگذارید. فراموش نکنید که از قلم انداختن یک آکولاد باعث به وجود آمدن خطا شده و یافتن آن را سخت می‌کند. مثالی دیگر در مورد دستور if :

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

int main()
{
    int firstNumber;
    int secondNumber;

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

    cout << "Enter another number: ";
    cin >> secondNumber;

    if (firstNumber == secondNumber)
    {
        cout << firstNumber << " == " << secondNumber << endl;
    }
    if (firstNumber != secondNumber)
    {
        cout << firstNumber << " != " << secondNumber << endl;
    }
    if (firstNumber < secondNumber)
    {
        cout << firstNumber << " < " << secondNumber << endl;
    }
    if (firstNumber > secondNumber)
    {
```

```
    cout << firstNumber << " > " << secondNumber << endl;
}
if (firstNumber <= secondNumber)
{
    cout << firstNumber << "<= " << secondNumber << endl;
}
if (firstNumber >= secondNumber)
{
    cout << firstNumber << ">= " << secondNumber << endl;
}
}
```

```
Enter a number: 2
Enter another number: 5
2 != 5
2 < 5
2 <= 5
Enter a number: 10
Enter another number: 3
10 != 3
10 > 3
10 >= 3
Enter a number: 5
Enter another number: 5
5 == 5
5 <= 5
5 >= 5
```

ما از عملگرهای مقایسه‌ای در دستور `if` استفاده کرده‌ایم. ابتدا دو عدد که قرار است با هم مقایسه شوند را به عنوان ورودی از کاربر می‌گیریم. اعداد با هم مقایسه می‌شوند و اگر شرط درست بود پیغامی چاپ می‌شود. به این نکته توجه داشته باشید که شرط‌ها مقادیر بولی هستند، بنابراین شما می‌توانید نتیجه یک عبارت را در داخل یکمتغیر بولی ذخیره کنید و سپس از متغیر به عنوان شرط در دستور `if` استفاده کنید. اگر مقدار `year` برابر 2000 باشد سپس حاصل عبارت در متغیر `isNewMillenium` ذخیره می‌شود. می‌توان از متغیر برای تشخیص کد اجرایی بدنه دستور `if` استفاده کرد خواه مقدار متغیر درست باشد یا نادرست.

```
bool isNewMillenium = year == 2000;

if (isNewMillenium)
{
    cout << "Happy New Millenium!";
}
```

دستور if...else

دستور if فقط برای اجرای یک حالت خاص به کار می‌رود یعنی اگر حالتی برقرار بود کار خاصی انجام شود. اما زمانی که شما بخواهید اگر شرط خاصی برقرار شد یک دستور و اگر برقرار نبود دستور دیگر اجرا شود باید از دستور if else استفاده کنید. ساختار دستور if else در زیر آمده است :

```
if (condition)
{
    code to execute if condition is true;
}
else
{
    code to execute if condition is false;
}
```

از کلمه کلیدی else نمی‌توان به تنهایی استفاده کرد بلکه حتماً باید با if به کار برده شود. اگر فقط یک کد اجرایی در داخل بدنه if و بدنه else دارید استفاده از آکولاد اختیاری است. کد داخل بلوک else فقط در صورتی اجرا می‌شود که شرط داخل دستور if نادرست باشد. در زیر نحوه استفاده از دستور if...else آمده است.

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int number = 5;
7
8      //Test the condition
9      if (number < 10)
10     {
11         cout << "The number is less than 10." << endl;
12     }
13     else
14     {
15         cout << "The number is either greater than or equal to 10." << endl;
16     }
17
18     //Modify value of number
19     number = 15;
20
21     //Repeat the test to yield a different result
22     if (number < 10)
23     {
24         cout << "The number is less than 10." << endl;
25     }
26     else
27     {
28         cout << "The number is either greater than or equal to 10." << endl;
29     }
30 }
```

در خط 6 یک متغیر به نام number تعریف کرده ایم و در خط 9 تست می کنیم که آیا مقدار متغیر number از 10 کمتر است یا نه و چون کمتر است در نتیجه کد داخل بلوک if اجرا می شود (خط 11) و اگر مقدار number را تغییر دهیم و به مقداری بزرگتر از 10 تغییر دهیم (خط 19)، شرط نادرست می شود (خط 22) و کد داخل بلوک else اجرا می شود (خط 28).

عملگر شرطی

عملگر شرطی (?:) در C++ مانند دستور شرطی if...else عمل می کند. در زیر نحوه استفاده از این عملگر آمده است:

```
<condition> ? <result if true> : <result if false>
```

عملگر شرطی تنها عملگر سه تایی C++ است که نیاز به سه عملوند دارد، شرط، یک مقدار زمانی که شرط درست باشد و یک مقدار زمانی که شرط نادرست باشد. اجازه بدهید که نحوه استفاده از این عملگر را در داخل برنامه مورد بررسی قرار دهیم.

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

int main()
{
    string pet1 = "puppy";
    string pet2 = "kitten";
    string type1;
    string type2;

    type1 = (pet1 == "puppy") ? "dog" : "cat";
    type2 = (pet2 == "kitten") ? "cat" : "dog";

    cout << type1 << endl;
    cout << type2;
}
```

```
dog
cat
```

برنامه بالا نحوه استفاده از این عملگر شرطی را نشان می دهد. خط یک به صورت زیر ترجمه می شود: اگر مقدار pet1 برابر با puppy سپس مقدار dog را در type1 قرار بده در غیر این صورت مقدار cat را type1 قرار بده. خط دو به صورت زیر ترجمه می شود: اگر مقدار pet2 برابر با kitten سپس مقدار cat را در type2 قرار بده در غیر این صورت مقدار dog. حال برنامه بالا را با استفاده از دستور if else می نویسیم:

```
if (pet1 == "puppy")
    type1 = "dog";
else
    type1 = "cat";
```

هنگامی که چندین دستور در داخل یک بلوک if یا else دارید از عملگر شرطی استفاده نکنید چون خوانایی برنامه را پایین می‌آورد.

دستور if چندگانه

اگر بخواهید چند شرط را بررسی کنید چکار می‌کنید؟ می‌توانید از چندین دستور if استفاده کنید و بهتر است که این دستورات if را به صورت زیر بنویسید :

```
if (condition)
{
    code to execute;
}
else
{
    if (condition)
    {
        code to execute;
    }
    else
    {
        if (condition)
        {
            code to execute;
        }
        else
        {
            code to execute;
        }
    }
}
```

خواندن کد بالا سخت است. بهتر است دستورات را به صورت تو رفتگی در داخل بلوک else بنویسید. می‌توانید کد بالا را ساده‌تر کنید :

```
if (condition)
{
    code to execute;
}
else if (condition)
{
    code to execute;
}
else if (condition)
{
    code to execute;
}
else
{
    code to execute;
}
```

```
}
```

حال که نحوه استفاده از دستور `else if` را یاد گرفتید باید بدانید که مانند `else if`، نیز به دستور `if` وابسته است. دستور `else if` وقتی اجرا می شود که اولین دستور `if` اشتباه باشد. حال اگر `else if` اشتباه باشد دستور `else if` بعدی اجرا می شود. و اگر آن نیز اجرا نشود در نهایت دستور `else` اجرا می شود. برنامه زیر نحوه استفاده از دستور `if else` را نشان می دهد :

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

int main()
{
    int choice;

    cout << "What's your favorite color?" << endl;
    cout << "[1] Black" << endl;
    cout << "[2] White" << endl;
    cout << "[3] Blue" << endl;
    cout << "[4] Red" << endl;
    cout << "[5] Yellow" << endl;

    cout << "Enter your choice: ";
    cin >> choice;

    if (choice == 1)
    {
        cout << "You might like my black t-shirt." << endl;
    }
    else if (choice == 2)
    {
        cout << "You might be a clean and tidy person." << endl;
    }
    else if (choice == 3)
    {
        cout << "You might be sad today." << endl;
    }
    else if (choice == 4)
    {
        cout << "You might be inlove right now." << endl;
    }
    else if (choice == 5)
    {
        cout << "Lemon might be your favorite fruit." << endl;
    }
    else
    {
        cout << "Sorry, your favorite color is not in the choices above." << endl;
    }
}
```

```
What's your favorite color?
[1] Black
[2] White
```

```
[3] Blue
[4] Red
[5] Yellow

Enter your choice: 1
You might like my black t-shirt.
What's your favorite color?
[1] Black
[2] White
[3] Blue
[4] Red
[5] Yellow

Enter your choice: 999
Sorry, your favorite color is not in the choices above.
```

خروجی برنامه بالا به متغیر choice وابسته است. بسته به اینکه شما چه چیزی انتخاب می‌کنید پیغام‌های مختلفی چاپ می‌شود. اگر عددی که شما تایپ می‌کنید در داخل حالت‌های انتخاب نباشد کد مربوط به بلوک else اجرا می‌شود.

دستور if تو در تو

می‌توان از دستور if تو در تو در C++ استفاده کرد. یک دستور ساده if در داخل دستور if دیگر.

```
if (condition)
{
    code to execute;

    if (condition)
    {
        code to execute;
    }
    else if (condition)
    {
        if (condition)
        {
            code to execute;
        }
    }
}
else
{
    if (condition)
    {
        code to execute;
    }
}
```

اجازه بدهید که نحوه استفاده از دستور if تو در تو را نشان دهیم :

```
1 #include <iostream>
```

```
2  #include <string>
3  using namespace std;
4
5  int main()
6  {
7      int age;
8      string gender;
9
10     cout << "Enter your age: ";
11     cin >> age;
12
13     cout << "Enter your gender (male/female): ";
14     cin >> gender;
15
16     if (age > 12)
17     {
18         if (age < 20)
19         {
20             if (gender == "male")
21             {
22                 cout << "You are a teenage boy." << endl;
23             }
24             else
25             {
26                 cout << "You are a teenage girl." << endl;
27             }
28         }
29         else
30         {
31             cout << "You are already an adult." << endl;
32         }
33     }
34     else
35     {
36         cout << "You are still too young." << endl;
37     }
38 }
```

```
Enter your age: 18
Enter your gender: male
You are a teenage boy.
Enter your age: 12
Enter your gender: female
You are still too young.
```

اجازه بدهید که برنامه را کالبد شکافی کنیم. ابتدا برنامه از شما درباره ستان سؤال می‌کند (خط 10). در خط 13 درباره جنسیت از شما سؤال می‌کند. سپس به اولین دستور if می‌رسد (خط 16). در این قسمت اگر سن شما بیشتر از 12 سال باشد برنامه وارد بدنه دستور if می‌شود در غیر اینصورت وارد بلوک else (خط 34) مربوط به همین دستور if می‌شود.

حال فرض کنیم که سن شما بیشتر از 12 سال است و شما وارد بدنه اولین if شده‌اید. در بدنه اولین if دو دستور if دیگر را مشاهده می‌کنید. اگر سن کمتر 20 باشد شما وارد بدنه if دوم می‌شوید و اگر نباشد به قسمت else متناظر با آن می‌روید (خط

(29). دوباره فرض می‌کنیم که سن شما کمتر از 20 باشد، در اینصورت وارد بدنه if دوم شده و با یک if دیگر مواجه می‌شوید (خط 20). در اینجا جنسیت شما مورد بررسی قرار می‌گیرد که اگر برابر "male" باشد، کدهای داخل بدنه سومین if اجرا می‌شود در غیر اینصورت قسمت else مربوط به این if اجرا می‌شود (خط 24). پیشنهاد می‌شود که از if تو در تو در برنامه کمتر استفاده کنید چون خوانایی برنامه را پایین می‌آورد.

استفاده از عملگرهای منطقی

عملگرهای منطقی به شما اجازه می‌دهند که چندین شرط را با هم ترکیب کنید. این عملگرها حداقل دو شرط را در گیر می‌کنند و در آخر یک مقدار بولی را بر می‌گردانند. در جدول زیر برخی از عملگرهای منطقی آمده است:

عملگر	تلفظ	مثال	تأثیر
&&	And	$z = (x > 2) \&\& (y < 10)$	مقدار Z در صورتی true است که هر دو شرط دو طرف عملگر مقدارشان true باشد. اگر فقط مقدار یکی از شروط false باشد مقدار Z، false خواهد شد.
	Or	$z = (x > 2) (y < 10)$	مقدار Z در صورتی true است که یکی از دو شرط دو طرف عملگر مقدارشان true باشد. اگر هر دو شرط مقدارشان false باشد مقدار Z، false خواهد شد.
!	Not	$z = !(x > 2)$	مقدار Z در صورتی true است که مقدار شرط false باشد و در صورتی false است که مقدار شرط true باشد.

به عنوان مثال جمله $z = (x > 2) \&\& (y < 10)$ را به این صورت بخوانید: "در صورتی مقدار Z برابر true است که مقدار x بزرگ‌تر از 2 و مقدار y کوچک‌تر از 10 باشد در غیر اینصورت false است". این جمله بدین معناست که برای اینکه مقدار کل دستور true باشد باید مقدار همه شروط true باشد. عملگر منطقی (||) تأثیر متفاوتی نسبت به عملگر منطقی (&&) دارد. نتیجه عملگر منطقی OR برابر true است اگر فقط مقدار یکی از شروط true باشد. و اگر مقدار هیچ یک از شروط true نباشد نتیجه false خواهد شد. می‌توان عملگرهای منطقی AND و OR را با هم ترکیب کرده و در یک عبارت به کار برد مانند :

```
if ((x == 1) && ((y > 3) || z < 10)) {  
    //do something here  
}
```

در اینجا استفاده از پرانتز مهم است چون از آن در گروه بندی شرطها استفاده می‌کنیم. در اینجا ابتدا عبارت $(z < 10)$ بررسی قرار می‌گیرد. (به علت تقدم عملگرها) سپس نتیجه آن بوسیله عملگر AND با نتیجه $(x == 1)$ مقایسه می‌شود. حال بیا ببینیم نحوه استفاده از عملگرهای منطقی در برنامه را مورد بررسی قرار دهیم :

```
#include <iostream>  
#include <String>  
using namespace std;  
  
int main()  
{  
    int age;  
    string gender;  
  
    cout << "Enter your age: ";  
    cin >> age;  
  
    cout << "Enter your gender (male/female): ";  
    cin >> gender;  
  
    if (age > 12 && age < 20)  
    {  
        if (gender == "male")  
        {  
            cout << "You are a teenage boy." << endl;  
        }  
        else  
        {  
            cout << "You are a teenage girl." << endl;  
        }  
    }  
    else  
    {  
        cout << "You are still too young." << endl;  
    }  
}
```

```
Enter your age: 18  
Enter your gender (male/female): female  
You are a teenage girl.  
Enter you age: 10  
Enter your gender (male/female): male  
You are not a teenager.
```

برنامه بالا نحوه استفاده از عملگر منطقی AND را نشان می‌دهد (خط 16). وقتی به دستور if می‌رسید (خط 16) برنامه سن شما را چک می‌کند. اگر سن شما بزرگ‌تر از 12 و کوچک‌تر از 20 باشد (سنتان بین 12 و 20 باشد) یعنی مقدار هر دو true باشد سپس

کدهای داخل بلوک if اجرا می‌شوند. اگر نتیجه یکی از شروط false باشد کدهای داخل بلوک else اجرا می‌شود. عملگر AND عملوند سمت چپ را مورد بررسی قرار می‌دهد. اگر مقدار آن false باشد دیگر عملوند سمت راست را بررسی نمی‌کند و مقدار false را بر می‌گرداند. بر عکس عملگر || عملوند سمت چپ را مورد بررسی قرار می‌دهد و اگر مقدار آن true باشد سپس عملوند سمت راست را نادیده می‌گیرد و مقدار true را بر می‌گرداند.

```
if (x == 2 & y == 3)
{
    //Some code here
}

if (x == 2 | y == 3)
{
    //Some code here
}
```

نکته مهم اینجاست که شما می‌توانید از عملگرهای & و | به عنوان عملگر بیتی استفاده کنید. تفاوت جزئی این عملگرها وقتی که به عنوان عملگر بیتی به کار می‌روند این است که دو عملوند را بدون در نظر گرفتن مقدار عملوند سمت چپ مورد بررسی قرار می‌دهند. به عنوان مثال حتی اگر مقدار عملوند سمت چپ false باشد عملوند سمت چپ به وسیله عملگر بیتی (&) ارزیابی می‌شود. اگر شرطها را در برنامه ترکیب کنید استفاده از عملگرهای منطقی AND (&&) و OR (||) به جای عملگرهای بیتی (&) و (|) بهتر خواهد بود. یکی دیگر از عملگرهای منطقی عملگر NOT (!) است که نتیجه یک عبارت را خنثی یا منفی می‌کند. به مثال زیر توجه کنید:

```
if (!(x == 2))
{
    cout << "x is not equal to 2.";
}
```

اگر نتیجه عبارت x == 2 برابر false باشد عملگر! آن را True می‌کند.

دستور Switch

در C++ ساختاری به نام switch وجود دارد که به شما اجازه می‌دهد که با توجه به مقدار ثابت یک متغیر چندین انتخاب داشته باشید. دستور switch معادل دستور if تو در تو است با این تفاوت که در دستور switch متغیر فقط مقادیر ثابتی از اعداد، رشته‌ها و یا کاراکترها را قبول می‌کند. مقادیر ثابت مقادیری هستند که قابل تغییر نیستند. در زیر نحوه استفاده از دستور switch آمده است :

```

switch (testVar)
{
case compareVa11:
    code to execute if testVar == compareVa11;
    break;
case compareVa12:
    code to execute if testVar == compareVa12;
    break;
.
.
.
case compareVa1N:
    code to execute if testVer == compareVa1N;
    break;
default:
    code to execute if none of the values above match the testVar;
    break;
}

```

ابتدا یک مقدار در متغیر switch که در مثال بالا testVar است قرار می‌دهید. این مقدار با هر یک از عبارتهای case داخل بلوک switch مقایسه می‌شود. اگر مقدار متغیر با هر یک از مقادیر موجود در دستورات case برابر بود کد مربوط به آن case اجرا خواهد شد. به این نکته توجه کنید که حتی اگر تعداد خط کدهای داخل دستور case از یکی بیشتر باشد نباید از آکولاد استفاده کنیم. آخر هر دستور case با کلمه کلیدی break تشخیص داده می‌شود که باعث می‌شود برنامه از دستور switch خارج شده و دستورات بعد از آن اجرا شوند. اگر این کلمه کلیدی از قلم بیوفتد، برنامه با خطا مواجه می‌شود. دستور switch یک بخش default دارد. این دستور در صورتی اجرا می‌شود که مقدار متغیر با هیچ یک از مقادیر دستورات case برابر نباشد. دستور default اختیاری است و اگر از بدنه switch حذف شود هیچ اتفاقی نمی‌افتد. مکان این دستور هم مهم نیست، اما بر طبق تعریف آن را در پایان دستورات می‌نویسند. به مثالی در مورد دستور switch توجه کنید :

```

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

int main()
{
    int choice;

    cout << "What's your favorite pet?" << endl;
    cout << "[1] Dog" << endl;
    cout << "[2] Cat" << endl;
    cout << "[3] Rabbit" << endl;
    cout << "[4] Turtle" << endl;
    cout << "[5] Fish" << endl;
    cout << "[6] Not in the choices" << endl;
    cout << "Enter your choice: " << endl;

    cin >> choice;
}

```

```
switch (choice)
{
case 1:
    cout << "Your favorite pet is Dog." << endl;
    break;
case 2:
    cout << "Your favorite pet is Cat." << endl;
    break;
case 3:
    cout << "Your favorite pet is Rabbit." << endl;
    break;
case 4:
    cout << "Your favorite pet is Turtle." << endl;
    break;
case 5:
    cout << "Your favorite pet is Fish." << endl;
    break;
case 6:
    cout << "Your favorite pet is not in the choices." << endl;
    break;
default:
    cout << "You don't have a favorite pet." << endl;
    break;
}
```

What's your favorite pet?

```
[1] Dog
[2] Cat
[3] Rabbit
[4] Turtle
[5] Fish
[6] Not in the choices
```

Enter your choice: 2

Your favorite pet is Cat.

What's your favorite pet?

```
[1] Dog
[2] Cat
[3] Rabbit
[4] Turtle
[5] Fish
[6] Not in the choices
```

Enter your choice: 99

You don't have a favorite pet.

برنامه بالا به شما اجازه انتخاب حیوان مورد علاقه‌تان را می‌دهد. به اسم هر حیوان یک عدد نسبت داده شده است. شما عدد را وارد می‌کنید و این عدد در دستور switch با مقادیر case مقایسه می‌شود و با هر کدام از آن مقادیر که برابر بود پیغام مناسب نمایش داده خواهد شد. اگر هم با هیچ کدام از مقادیر case ها برابر نبود دستور default اجرا می‌شود. یکی دیگر از ویژگی‌های

دستور switch این است که شما می‌توانید از دو یا چند case برای نشان داده یک مجموعه کد استفاده کنید. در مثال زیر اگر مقدار number عدد 1، 2 یا 3 باشد یک کد اجرا می‌شود. توجه کنید که case ها باید پشت سر هم نوشته شوند.

```
switch (number)
{
case 1:
case 2:
case 3:
    cout << "This code is shared by three values." << endl;
    break;
}
```

همانطور که قبلاً ذکر شد دستور switch معادل دستور if تو در تو است. برنامه بالا را به صورت زیر نیز می‌توان نوشت :

```
if (choice == 1)
    cout << "Your favorite pet is Dog." << endl;
else if (choice == 2)
    cout << "Your favorite pet is Cat." << endl;
else if (choice == 3)
    cout << "Your favorite pet is Rabbit." << endl;
else if (choice == 4)
    cout << "Your favorite pet is Turtle." << endl;
else if (choice == 5)
    cout << "Your favorite pet is Fish." << endl;
else if (choice == 6)
    cout << "Your favorite pet is not in the choices." << endl;
else
    cout << "You don't have a favorite pet." << endl;
```

کد بالا دقیقاً نتیجه‌ای مانند دستور switch دارد. دستور default معادل دستور else می‌باشد. حال از بین این دو دستور (if else و switch) کدامیک را انتخاب کنیم. از دستور switch موقعی استفاده می‌کنیم که مقداری که می‌خواهیم با دیگر مقادیر مقایسه شود ثابت باشد. مثلاً در مثال زیر هیچگاه از switch استفاده نکنید.

```
int myNumber = 5;
int x = 5;

switch (myNumber)
{
case x:
    cout << "Error, you can't use variables as a value to be compared in a case statement.";
    break;
}
```

مشاهده می‌کنید که با اینکه مقدار x عدد 5 است و به طور واضح با متغیر myNumber مقایسه شده است برنامه خطا می‌دهد چون x یک ثابت نیست بلکه یک متغیر است یا به زبان ساده‌تر، قابلیت تغییر را دارد. اگر بخواهید از x استفاده کنید و برنامه خطا ندهد باید از کلمه کلیدی const به صورت زیر استفاده کنید.

```
int myNumber = 5;
const int x = 5;

switch (myNumber)
{
case x:
    cout << "Error has been fixed!" << endl;
    break;
}
```

از کلمه کلیدی `const` برای ایجاد ثابت‌ها استفاده می‌شود. توجه کنید که بعد از تعریف یک ثابت نمی‌توان مقدار آن را در طول برنامه تغییر داد. به یاد داشته باشید که باید ثابت‌ها را حتماً مقداردهی کنید. دستور `switch` یک مقدار را با مقادیر `case` ها مقایسه می‌کند و شما لازم نیست که به شکل زیر مقادیر را با هم مقایسه کنید :

```
switch (myNumber)
{
case x > myNumber:
    cout << "switch staments can't test if a value is less than or greater than the other value.";
    break;
}
```

تکرار

ساختارهای تکرار به شما اجازه می‌دهند که یک یا چند دستور کد را تا زمانی که یک شرط برقرار است تکرار کنید. بدون ساختارهای تکرار شما مجبورید همان تعداد کدها را بنویسید که بسیار خسته کننده است. مثلاً شما مجبورید 10 بار جمله "Hello World." را تایپ کنید مانند مثال زیر :

```
cout << "Hello World." << endl;
cout << "Hello World." << endl;
cout << "Hello World." << endl;
cout << "Hello World." << endl;
cout << "Hello World." << endl;
cout << "Hello World." << endl;
cout << "Hello World." << endl;
cout << "Hello World." << endl;
cout << "Hello World." << endl;
cout << "Hello World." << endl;
```

البته شما می‌توانید با کپی کردن این تعداد کد را راحت بنویسید ولی این کار در کل کیفیت کدنویسی را پایین می‌آورد. راه بهتر برای نوشتن کدهای بالا استفاده از حلقه‌ها است. حلقه‌ها در C++ عبارت‌اند از :

- while •
- do while •
- for •

ابتدای‌ترین ساختار تکرار در C++ حلقه While است. ابتدا یک شرط را مورد بررسی قرار می‌دهد و تا زمانی‌که شرط برقرار باشد کدهای درون بلوک اجرا می‌شوند. ساختار حلقه While به صورت زیر است :

```
while(condition)
{
    code to loop;
}
```

می‌بینید که ساختار While مانند ساختار if بسیار ساده است. ابتدا یک شرط را که نتیجه آن یک مقدار بولی است می‌نویسیم اگر نتیجه درست یا true باشد سپس کدهای داخل بلوک While اجرا می‌شوند. اگر شرط غلط یا false باشد وقتی که برنامه به حلقه While برسد هیچکدام از کدها را اجرا نمی‌کند. برای متوقف شدن حلقه باید مقادیر داخل حلقه While اصلاح شوند. به یک متغیر شمارنده در داخل بدنه حلقه نیاز داریم. این شمارنده برای آزمایش شرط مورد استفاده قرار می‌گیرد و ادامه یا توقف حلقه به نوعی به آن وابسته است. این شمارنده را در داخل بدنه باید کاهش یا افزایش دهیم. در برنامه زیر نحوه استفاده از حلقه While آمده است :

```
#include <iostream>

using namespace std;

int main()
{
    int counter = 1;

    while (counter <= 10)
    {
        cout << "Hello World!" << endl;
        counter++;
    }
}
```

[illegible]


```
Hello World!
```

برنامه بالا 10 بار پیغام Hello World! را چاپ می‌کند. اگر از حلقه در مثال بالا استفاده نمی‌کردیم مجبور بودیم تمام 10 خط را تایپ کنیم. اجازه دهید که نگاهی به کدهای برنامه فوق ببندیم. ابتدا در خط 7 یک متغیر تعریف و از آن به عنوان شمارنده حلقه استفاده شده است. سپس به آن مقدار 1 را اختصاص می‌دهیم چون اگر مقدار نداشته باشد نمی‌توان در شرط از آن استفاده کرد.

در خط 9 حلقه While را وارد می‌کنیم. در حلقه While ابتدا مقدار اولیه شمارنده با 10 مقایسه می‌شود که آیا از 10 کمتر است یا با آن برابر است. نتیجه هر بار مقایسه ورود به بدنه حلقه While و چاپ پیغام است. همانطور که مشاهده می‌کنید بعد از هر بار مقایسه مقدار شمارنده یک واحد اضافه می‌شود (خط 12). حلقه تا زمانی تکرار می‌شود که مقدار شمارنده از 10 کمتر باشد.

اگر مقدار شمارنده یک بماند و آن را افزایش ندهیم و یا مقدار شرط هرگز false نشود یک حلقه بین‌هایت به وجود می‌آید. به این نکته توجه کنید که در شرط بالا به جای علامت < از <= استفاده شده است. اگر از علامت < استفاده می‌کردیم کد ما 9 بار تکرار می‌شد چون مقدار اولیه 1 است و هنگامی که شرط به 10 برسد false می‌شود چون $10 < 10$ نیست. اگر می‌خواهید یک حلقه بی‌نهایت ایجاد کنید که هیچگاه متوقف نشود باید یک شرط ایجاد کنید که همواره درست (true) باشد.

```
while(true)
{
    //code to loop
}
```

این تکنیک در برخی موارد کارایی دارد و آن زمانی است که شما بخواهید با استفاده از دستورات break و return که در آینده توضیح خواهیم داد از حلقه خارج شوید.

حلقه do while

حلقه do while یکی دیگر از ساختارهای تکرار است. این حلقه بسیار شبیه حلقه while است با این تفاوت که در این حلقه ابتدا کد اجرا می‌شود و سپس شرط مورد بررسی قرار می‌گیرد. ساختار حلقه do while به صورت زیر است :

```
do
{
    code to repeat;
} while (condition);
```

همانطور که مشاهده می‌کنید شرط در آخر ساختار قرار دارد. این بدین معنی است که کدهای داخل بدنه حداقل یکبار اجرا می‌شوند. برخلاف حلقه while که اگر شرط نادرست باشد دستورات داخل بدنه اجرا نمی‌شوند. برای اثبات این موضوع به کدهای زیر توجه کنید :

```
int number = 1;
do
{
    cout << "Hello World!" << endl;
} while (number > 10);
```

Hello World!

با اجرای کد بالا، اول دستورات بلوک do اجرا می‌شوند و بعد مقدار number با عدد 10 مقایسه می‌شود. در نتیجه حتی اگر شرط نادرست باشد باز هم قسمت do حداقل یک بار اجرا می‌شوند.

```
int number = 1;
while (number > 10)
{
    cout << "Hello World!" << endl;
}
```

اما در کد بالا چون اول مقدار number ابتدا مورد مقایسه قرار می‌گیرد، اگر شرط درست نباشد دیگر کدی اجرا نمی‌شود. یکی از موارد برتری استفاده از حلقه while نسبت به حلقه while زمانی است که شما بخواهید اطلاعاتی از کاربر دریافت کنید. در دو کد زیر، یک عملیات یکسان توسط دو حلقه while و do while پیاده سازی شده است :

```
//while version

cout << "Enter a number greater than 10: " << endl;
cin >> number;

while (number < 10)
{
    cout << "Enter a number greater than 10: " << endl;
    cin >> number;
}
```

```
//do while version

do
{
    cout << "Enter a number greater than 10: " << endl;
    cin >> number;
} while (number < 10);
```

مشاهده می‌کنید که از کدهای کمتری در بدنه while نسبت به while استفاده شده است.

حلقه for

یکی دیگر از ساختارهای تکرار حلقه for است. این حلقه عملی شبیه به حلقه while انجام می‌دهد و فقط دارای چند خصوصیت اضافی است. ساختار حلقه for به صورت زیر است :

```
for(initialization; condition; operation)
{
    code to repeat;
}
```

مقدار دهی اولیه (initialization) اولین مقداری است که به شمارنده حلقه می‌دهیم. شمارنده فقط در داخل حلقه for قابل دسترسی است.

شرط (condition) در اینجا مقدار شمارنده را با یک مقدار دیگر مقایسه می‌کند و تعیین می‌کند که حلقه ادامه یابد یا نه.

عملگر (operation) که مقدار اولیه متغیر را کاهش یا افزایش می‌دهد.

در زیر یک مثال از حلقه for آمده است :

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

int main()
{
    for (int i = 1; i <= 10; i++)
    {
        cout << "Number " << i << endl;
    }
}
```

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

برنامه بالا اعداد 1 تا 10 را با استفاده از حلقه for می‌شمارد. ابتدا یک متغیر به عنوان شمارنده تعریف می‌کنیم و آن را با مقدار 1 مقدار دهی اولیه می‌کنیم. سپس با استفاده از شرط آن را با مقدار 10 مقایسه می‌کنیم که آیا کمتر است یا مساوی؟ توجه کنید که

قسمت سوم حلقه (i++) فوراً اجرا نمی‌شود. کد اجرا می‌شود و ابتدا رشته Number و سپس مقدار جاری i یعنی 1 را چاپ می‌کند. آنگاه یک واحد به مقدار i اضافه شده و مقدار i برابر 2 می‌شود و بار دیگر i با عدد 10 مقایسه می‌شود و این حلقه تا زمانی که مقدار شرط true شود ادامه می‌یابد. حال اگر بخواهید معکوس برنامه بالا را پیاده سازی کنید یعنی اعداد از بزرگ به کوچک چاپ شوند باید به صورت زیر عمل کنید :

```
for (int i = 10; i > 0; i--)
{
    //code omitted
}
```

کد بالا اعداد را از 10 به 1 چاپ می‌کند (از بزرگ به کوچک). مقدار اولیه شمارنده را 10 می‌دهیم و با استفاده از عملگر کاهش (--). برنامه‌ای که شمارش معکوس را انجام می‌دهد ایجاد می‌کنیم. می‌توان قسمت شرط و عملگر را به صورت‌های دیگر نیز تغییر داد. به عنوان مثال می‌توان از عملگرهای منطقی در قسمت شرط و از عملگرهای تخصیصی در قسمت عملگر افزایش یا کاهش استفاده کرد. همچنین می‌توانید از چندین متغیر در ساختار حلقه for استفاده کنید.

```
for (int i = 1, y = 2; i < 10 && y > 20; i++, y -= 2)
{
    //some code here
}
```

به این نکته توجه کنید که اگر از چندین متغیر شمارنده یا عملگر در حلقه for استفاده می‌کنید باید آن‌ها را با استفاده از کاما از هم جدا کنید.

حلقه‌های تو در تو (Nested Loops)

C++ به شما اجازه می‌دهد که از حلقه‌ها به صورت تو در تو استفاده کنید. اگر یک حلقه در داخل حلقه دیگر قرار بگیرد، به آن حلقه تو در تو گفته می‌شود. در این نوع حلقه‌ها، به ازای اجرای یک بار حلقه بیرونی، حلقه داخلی به طور کامل اجرا می‌شود. در زیر نحوه ایجاد حلقه تو در تو آمده است :

```
for (init; condition; increment)
{
    for (init; condition; increment)
    {
        //statement(s);
    }
    //statement(s);
}
```

```
while (condition)
{
    while (condition)
    {
        //statement(s);
    }
    //statement(s);
}
```

```
do
{
    //statement(s);
    do
    {
        //statement(s);
    } while (condition);
} while (condition);
```

نکته‌ای که در مورد حلقه‌های تو در تو وجود دارد این است که می‌توان از یک نوع حلقه در داخل نوع دیگر استفاده کرد. مثلاً می‌توان از حلقه for در داخل حلقه while استفاده نمود. در مثال زیر نحوه استفاده از این حلقه‌ها ذکر شده است. فرض کنید که می‌خواهید یک مستطیل با 3 سطر و 5 ستون ایجاد کنید :

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      for (int i = 1; i <= 4; i++)
7      {
8          for (int j = 1; j <= 5; j++)
9          {
10             cout << " * ";
11          }
12          cout << endl;
13      }
14
15 }
```

```
* * * * *
* * * * *
* * * * *
* * * * *
```

در کد بالا به ازای یک بار اجرای حلقه for اول (خط 6)، حلقه for دوم (11-8) به طور کامل اجرا می‌شود. یعنی وقتی مقدار i برابر عدد 1 می‌شود، علامت * توسط حلقه دوم 5 بار چاپ می‌شود، وقتی i برابر 2 می‌شود، دوباره علامت * پنج بار چاپ می‌شود و

در کل منظور از دو حلقه for این است که در 4 سطر علامت * در 5 ستون چاپ شود یا 4 سطر ایجاد شود و در هر سطر 5 بار علامت * چاپ شود. خط 12 هم برای ایجاد خط جدید است. یعنی وقتی حلقه داخلی به طور کامل اجرا شد، یک خط جدید ایجاد می شود و علامت های * در خطوط جدید چاپ می شوند.

خارج شدن از حلقه با استفاده از break و continue

گاهی اوقات با وجود درست بودن شرط می خواهیم حلقه متوقف شود. سؤال اینجاست که چطور این کار را انجام دهید؟ با استفاده از کلمه کلیدی break حلقه را متوقف کرده و با استفاده از کلمه کلیدی continue می توان بخشی از حلقه را رد کرد و به مرحله بعد رفت. برنامه زیر نحوه استفاده از break و continue را نشان می دهد :

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Demonstrating the use of break" << endl;

    for (int x = 1; x < 10; x++)
    {
        if (x == 5)
            break;

        cout << "Number " << x << endl;
    }

    cout << endl;

    cout << "Demonstrating the use of continue." << endl;

    for (int x = 1; x < 10; x++)
    {
        if (x == 5)
            continue;

        cout << "Number " << x << endl;
    }
}
```

Demonstrating the use of break.

Number 1
Number 2
Number 3
Number 4

Demonstrating the use of continue.

```
Number 1
Number 2
Number 3
Number 4
Number 6
Number 7
Number 8
Number 9
```

در این برنامه از حلقه for برای نشان دادن کاربرد دو کلمه کلیدی فوق استفاده شده است اگر به جای for از حلقه‌های while و do...while استفاده می‌شد نتیجه یکسانی به دست می‌آمد. همانطور که در شرط برنامه (خط 11) آمده است وقتی که مقدار x به عدد 5 رسید سپس دستور break اجرا شود (خط 12)

حلقه بلافاصله متوقف می‌شود حتی اگر شرط $x < 10$ برقرار باشد. از طرف دیگر در خط 24 حلقه for فقط برای یک تکرار خاص متوقف شده و سپس ادامه می‌یابد. (وقتی مقدار x برابر 5 شود حلقه از 5 رد شده و مقدار 5 را چاپ نمی‌کند و بقیه مقادیر چاپ می‌شوند).

آرایه‌ها

آرایه نوعی متغیر است که لیستی از آدرس‌های مجموعه‌ای از داده‌های هم نوع را در خود ذخیره می‌کند. تعریف چندین متغیر از یک نوع برای هدفی یکسان بسیار خسته کننده است. مثلاً اگر بخواهید صد متغیر از نوع اعداد صحیح تعریف کرده و از آن‌ها استفاده کنید. مطمئناً تعریف این همه متغیر بسیار کسالت آور و خسته کننده است. اما با استفاده از آرایه می‌توان همه آن‌ها را در یک خط تعریف کرد. در زیر راهی ساده برای تعریف یک آرایه نشان داده شده است :

```
datatype arrayName[length];
```

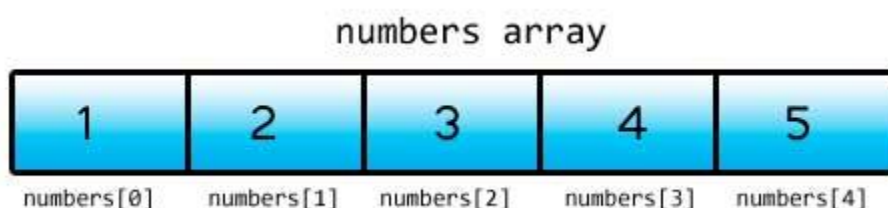
Datatype نوع داده‌هایی را نشان می‌دهد که آرایه در خود ذخیره می‌کند. گروهی که بعد از نوع داده قرار می‌گیرد و نشان دهنده استفاده از آرایه است. lenght یا طول آرایه که به کامپایلر می‌گوید شما قصد دارید چه تعداد داده یا مقدار را در آرایه ذخیره کنید. arrayName که نام آرایه را نشان می‌دهد. هنگام نامگذاری آرایه بهتر است که نام آرایه نشان دهنده نوع آرایه باشد. به عنوان مثال برای نامگذاری آرایه‌ای که اعداد را در خود ذخیره می‌کند از کلمه numbers استفاده کنید. برای تعریف یک آرایه که 5 مقدار از نوع اعداد صحیح در خود ذخیره می‌کند باید به صورت زیر عمل کنیم :

```
int numbers[5];
```

در این مثال 5 آدرس از فضای حافظه کامپیوتر شما برای ذخیره 5 مقدار رزرو می‌شود. حال چطور مقادیرمان را در هر یک از این آدرس‌ها ذخیره کنیم؟ برای دسترسی و اصلاح مقادیر آرایه از اندیس یا مکان آن‌ها استفاده می‌شود.

```
numbers[0] = 1;
numbers[1] = 2;
numbers[2] = 3;
numbers[3] = 4;
numbers[4] = 5;
```

اندیس یک آرایه از صفر شروع شده و به یک واحد کمتر از طول آرایه ختم می‌شود. به عنوان مثال شما یک آرایه 5 عضوی دارید، اندیس آرایه از 0 تا 4 می‌باشد چون طول آرایه 5 است پس 1-5 برابر است با 4. این بدان معناست که اندیس 0 نشان دهنده اولین عضو آرایه است و اندیس 1 نشان دهنده دومین عضو و الی آخر. برای درک بهتر مثال بالا به شکل زیر توجه کنید :



به هر یک از اجزاء آرایه و اندیس‌های داخل گروه توجه کنید. کسانی که تازه شروع به برنامه نویسی کرده‌اند معمولاً در گذاشتن اندیس دچار اشتباه می‌شوند و مثلاً ممکن است در مثال بالا اندیس‌ها را از 1 شروع کنند. یکی دیگر از راه‌های تعریف سریع و مقدار دهی یک آرایه به صورت زیر است :

```
datatype arrayName[length] = { val1, val2, ... valN };
```

در این روش شما می‌توانید فوراً بعد از تعریف اندازه آرایه مقادیر را در داخل آکولاد قرار دهید. به یاد داشته باشید که هر کدام از مقادیر را با استفاده از کاما از هم جدا کنید. همچنین تعداد مقادیر داخل آکولاد باید با اندازه آرایه تعریف شده برابر باشد. به مثال زیر توجه کنید :

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

این مثال با مثال قبل هیچ تفاوتی ندارد و تعداد خط‌های کدنویسی را کاهش می‌دهد. شما می‌توانید با استفاده از اندیس به مقدار هر یک از اجزاء آرایه دسترسی یابید و آن‌ها را به دلخواه تغییر دهید. تعداد اجزاء آرایه در مثال بالا 5 است و ما 5 مقدار را در آن قرار می‌دهیم. اگر تعداد مقادیری که در آرایه قرار می‌دهیم کمتر یا بیشتر از طول آرایه باشد با خطا مواجه می‌شویم. یک راه بسیار ساده‌تر برای تعریف آرایه به صورت زیر است :


```
int numbers[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

به سادگی و بدون احتیاج ذکر طول می‌توان مقادیر را در داخل آکولاد قرار داد. کامپایلر به صورت اتوماتیک با شمارش مقادیر طول آرایه را تشخیص می‌دهد.

دستیابی به مقادیر آرایه با استفاده از حلقه for

در زیر مثالی در مورد استفاده از آرایه‌ها آمده است. در این برنامه 5 مقدار از کاربر گرفته شده و میانگین آن‌ها حساب می‌شود:

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

int main()
{
    int numbers[5];

    int total = 0;
    double average;

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

    for (int i = 0; i < size(numbers); i++)
    {
        total += numbers[i];
    }

    average = total / (double)size(numbers);

    cout << "Average = " << average << endl;
}
```

```
Enter a number: 90
Enter a number: 85
Enter a number: 80
Enter a number: 87
Enter a number: 92
Average = 86
```

در خط 6 یک آرایه تعریف شده است که می‌تواند 5 عدد صحیح را در خود ذخیره کند. خطوط 8 و 9 متغیرهایی تعریف شده‌اند که از آن‌ها برای محاسبه میانگین استفاده می‌شود. توجه کنید که مقدار اولیه total صفر است تا از بروز خطا هنگام اضافه شدن مقدار به آن جلوگیری شود. در خطوط 11 تا 15 حلقه for برای تکرار و گرفتن ورودی از کاربر تعریف شده است. از متد size() برای تشخیص تعداد اجزای آرایه استفاده می‌شود. اگر چه می‌توانستیم به سادگی در حلقه for مقدار 5 را برای شرط قرار دهیم

ولی استفاده از خاصیت طول آرایه کار راحت‌تری است و می‌توانیم طول آرایه را تغییر دهیم و شرط حلقه for با تغییر جدید هماهنگ می‌شود. در خط 14 ورودی دریافت شده از کاربر در آرایه ذخیره می‌شود. اندیس استفاده شده در number (خط 14) مقدار i جاری در حلقه است. برای مثال در ابتدای حلقه مقدار i صفر است بنابراین وقتی در خط 14 اولین داده از کاربر گرفته می‌شود اندیس آن برابر صفر می‌شود. در تکرار بعدی i یک واحد اضافه می‌شود و در نتیجه در خط 14 و بعد از ورود دومین داده توسط کاربر اندیس آن برابر یک می‌شود. این حالت تا زمانی که شرط در حلقه for برقرار است ادامه می‌یابد. در خطوط 17-20 از حلقه for دیگر برای دسترسی به مقدار هر یک از داده‌های آرایه استفاده شده است. در این حلقه نیز مانند حلقه قبل از مقدار متغیر شمارنده به عنوان اندیس استفاده می‌کنیم.

هر یک از اجزای عددی آرایه به متغیر total اضافه می‌شوند. بعد از پایان حلقه می‌توانیم میانگین اعداد را حساب کنیم (خط 22). مقدار total را بر تعداد اجزای آرایه (تعداد عددها) تقسیم می‌کنیم. برای دسترسی به تعداد اجزای آرایه می‌توان از متد size() استفاده کرد. توجه کنید که در اینجا ما خروجی متد size() را به نوع double تبدیل کرده‌ایم بنابراین نتیجه عبارت یک مقدار از نوع double خواهد شد و دارای بخش کسری می‌باشد. حال اگر عملوندهای تقسیم را به نوع double تبدیل نکنیم نتیجه تقسیم یک عدد از نوع صحیح خواهد شد و دارای بخش کسری نیست. خط 24 مقدار میانگین را در صفحه نمایش چاپ می‌کند. طول آرایه بعد از مقدار دهی نمی‌تواند تغییر کند. به عنوان مثال اگر یک آرایه را که شامل 5 جز است مقدار دهی کنید دیگر نمی‌توانید آن را مثلاً به 10 جز تغییر اندازه دهید. البته تعداد خاصی از کلاس‌ها مانند آرایه‌ها عمل می‌کنند و توانایی تغییر تعداد اجزای تشکیل دهنده خود را دارند. آرایه‌ها در برخی شرایط بسیار پر کاربرد هستند و تسلط شما بر این مفهوم و اینکه چطور از آن‌ها استفاده کنید بسیار مهم است.

آرایه‌های چند بعدی

آرایه‌های چند بعدی آرایه‌هایی هستند که برای دسترسی به هر یک از عناصر آن‌ها باید از چندین اندیس استفاده کنیم. یک آرایه چند بعدی را می‌توان مانند یک جدول با تعدادی ستون و ردیف تصور کنید. با افزایش اندیس‌ها اندازه ابعاد آرایه نیز افزایش می‌یابد و آرایه‌های چند بعدی با بیش از دو اندیس به وجود می‌آیند. نحوه ایجاد یک آرایه با دو بعد به صورت زیر است :

```
datatype arrayName[lengthX][lengthY];
```

و یک آرایه سه بعدی به صورت زیر ایجاد می‌شود :

```
datatype arrayName[lengthX][lengthY][lengthZ];
```

می‌توان یک آرایه با تعداد زیادی بعد ایجاد کرد به شرطی که هر بعد دارای طول مشخصی باشد. به دلیل اینکه آرایه‌های سه بعدی یا آرایه‌های با بیشتر از دو بعد بسیار کمتر مورد استفاده قرار می‌گیرند اجازه بدهید که در این درس بر روی آرایه‌های دو بعدی تمرکز کنیم. در تعریف این نوع آرایه ابتدا نوع آرایه یعنی اینکه آرایه چه نوعی از انواع داده را در خود ذخیره می‌کند را مشخص می‌کنیم. سپس نام آرایه و در نهایت دو جفت کروشه قرار می‌دهیم. در یک آرایه دو بعدی برای دسترسی به هر یک از عناصر به دو مقدار نیاز داریم یکی مقدار X و دیگری مقدار Y که مقدار x نشان دهنده ردیف و مقدار Y نشان دهنده ستون آرایه است البته اگر ما آرایه دو بعدی را به صورت جدول در نظر بگیریم. یک آرایه سه بعدی را می‌توان به صورت یک مکعب تصور کرد که دارای سه بعد است و x طول، Y عرض و z ارتفاع آن است. یک مثال از آرایه دو بعدی در زیر آمده است :

```
int numbers[3][5];
```

کد بالا به کامپایلر می‌گوید که فضای کافی به عناصر آرایه اختصاص بده (در این مثال 15 خانه). در شکل زیر مکان هر عنصر در یک آرایه دو بعدی نشان داده شده است.

number [3][5]

number [0][0]	number [0][1]	number [0][2]	number [0][3]	number [0][4]
number [1][0]	number [1][1]	number [1][2]	number [1][3]	number [1][4]
number [2][0]	number [2][1]	number [2][2]	number [2][3]	number [2][4]

مقدار 3 را به x اختصاص می‌دهیم چون 3 سطر و مقدار 5 را به Y چون 5 ستون داریم اختصاص می‌دهیم. چطور یک آرایه چند بعدی را مقدار دهی کنیم؟ چند راه برای مقدار دهی به آرایه‌ها وجود دارد.

```
datatype arrayName[x][y] = {
    { r0c0, r0c1, ... r0cX },
    { r1c0, r1c1, ... r1cX },
    .
    .
    .
    { rYc0, rYc1, ... rYcX }
};
```

البته می‌توان تعداد سطرها را هم نوشت ولی تعداد ستون‌ها حتماً باید ذکر شوند :

```
datatype arrayName[][y] = {
    { r0c0, r0c1, ... r0cX },
    { r1c0, r1c1, ... r1cX },
    :
    :
    { rYc0, rYc1, ... rYcX }
};
```

به عنوان مثال :

```
int numbers[][5] = {
    { 1, 2, 3, 4, 5 },
    { 6, 7, 8, 9, 10 },
    { 11, 12, 13, 14, 15 }
};
```

و یا می‌توان مقدار دهی به عناصر را به صورت دستی انجام داد مانند :

```
array[0][0] = value;
array[0][1] = value;
array[0][2] = value;
array[1][0] = value;
array[1][1] = value;
array[1][2] = value;
array[2][0] = value;
array[2][1] = value;
array[2][2] = value;
```

همانطور که مشاهده می‌کنید برای دسترسی به هر یک از عناصر در یک آرایه دو بعدی به سادگی می‌توان از اندیس‌های X و Y و یک جفت کروشه مانند مثال استفاده کرد.

گردش در میان عناصر آرایه‌های چند بعدی

گردش در میان عناصر آرایه‌های چند بعدی نیاز به کمی دقت دارد. برنامه زیر نشان می‌دهد که چطور از حلقه for برای خواندن همه مقادیر آرایه و تعیین انتهای ردیف‌ها استفاده کنید.

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     int numbers[3][5] = {
7         { 1 , 2 , 3 , 4 , 5 },
8         { 6 , 7 , 8 , 9 , 10 },
```

```

9           { 11, 12, 13, 14, 15 }
10        };
11
12    for (int row = 0; row < size(numbers); row++)
13    {
14        for (int col = 0; col < size(numbers[0]); col++)
15        {
16            cout << numbers[row][col] << " ";
17        }
18
19        //Go to the next line
20        cout << endl;
21    }
22 }

```

```

1 2 3 4 5
6 7 8 9 10
11 12 13 14 15

```

همانطور که در مثال بالا نشان داده شده است با استفاده از یک حلقه for نمیتوان به مقادیر دسترسی یافت بلکه به یک حلقه for تو در تو نیاز داریم، زیرا آرایه دو بعدی به صورت یک جدول شامل سطر و ستون است، پس لازم است که از یک حلقه for برای گردش در میان سطرها و از حلقه for دیگر برای گردش در میان ستونهای این جدول (آرایه) استفاده کنیم. اولین حلقه for (خط 12) برای گردش در میان ردیفهای آرایه به کار می‌رود. این حلقه تا زمانی ادامه می‌یابد که مقدار ردیف کمتر از طول اولین بعد باشد (زیرا اندیس ابعاد آرایه از صفر شروع می‌شود. در مثال بالا مقدار اولین بعد برابر 3 است). در این مثال از متد `size()` استفاده کرده‌ایم. این متد طول آرایه را در یک بعد خاص نشان می‌دهد. به عنوان مثال برای به دست آوردن طول اولین یا همان تعداد سطرها کافیست که نام آرایه را به این متد ارسال می‌کنیم.

در داخل اولین حلقه for حلقه for دیگری تعریف شده است (خط 14). در این حلقه یک شمارنده برای شمارش تعداد ستونهای `numbers[0]` (col) هر ردیف تعریف شده است و در شرط داخل آن بار دیگر از متد `size()` استفاده شده است، ولی این بار مقدار `numbers[0]` را به آن ارسال می‌کنیم تا طول بعد دوم آرایه را به دست آوریم. پس به عنوان مثال وقتی که مقدار ردیف (row) صفر باشد، حلقه دوم از `[0][0]` تا `[4][0]` اجرا می‌شود. سپس مقدار هر عنصر از آرایه را با استفاده از حلقه نشان می‌دهیم، اگر مقدار ردیف (row) برابر 0 و مقدار ستون (col) برابر 0 باشد مقدار عنصری که در ستون 1 و ردیف 1 (`numbers[0][0]`) قرار دارد نشان داده خواهد شد که در مثال بالا عدد 1 است.

بعد از اینکه دومین حلقه تکرار به پایان رسید، فوراً دستورات بعد از آن اجرا خواهند شد، که در اینجا دستور `cout << endl` به برنامه اطلاع می‌دهد که به خط بعد برود. سپس حلقه با اضافه کردن یک واحد به مقدار row این فرایند را دوباره تکرار می‌کند. سپس دومین حلقه for اجرا شده و مقادیر دومین ردیف نمایش داده می‌شود. این فرایند تا زمانی اجرا می‌شود که مقدار row

کمتر از طول اولین بعد باشد. حال بیایید آنچه را از قبل یاد گرفته ایم در یک برنامه به کار ببریم. این برنامه نمره چهار درس مربوط به سه دانش آموز را از ما می گیرد و معدل سه دانش آموز را حساب می کند.

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      double studentGrades[3][4];
8      double total;
9
10     for (int student = 0; student < size(studentGrades); student++)
11     {
12         total = 0;
13
14         cout << "Enter grades for Student " << (student + 1) << endl;
15
16         for (int grade = 0; grade < size(studentGrades[0]); grade++)
17         {
18             cout << "Enter Grade #" << (grade + 1) << " : " ;
19             cin >> studentGrades[student][grade];
20             total += studentGrades[student][grade];
21         }
22
23         cout << "Average is " << (total / size(studentGrades[0])) << endl;
24         cout << endl;
25     }
26 }
```

```
Enter grades for Student 1
Enter Grade #1: 92
Enter Grade #2: 87
Enter Grade #3: 89
Enter Grade #4: 95
Average is 90.75
```

```
Enter grades for Student 2
Enter Grade #1: 85
Enter Grade #2: 85
Enter Grade #3: 86
Enter Grade #4: 87
Average is 85.75
```

```
Enter grades for Student 3
Enter Grade #1: 90
Enter Grade #2: 90
Enter Grade #3: 90
Enter Grade #4: 90
Average is 90.00
```

در برنامه بالا یک آرایه چند بعدی از نوع double تعریف شده است (خط 7). همچنین یک متغیر به نام total تعریف می‌کنیم که جمع نمرات وارد شده برای دانش آموز در آن قرار می‌گیرد. حال وارد حلقه for تو در تو می‌شویم (خط 10) در اولین حلقه for یک متغیر به نام student تعریف کرده‌ایم که مقادیر اولین بعد آرایه (که همان تعداد دانش آموزان است) در آن قرار می‌گیرد. از متد size() هم برای تشخیص تعداد دانش آموزان استفاده شده است. وارد بدنه حلقه for می‌شویم. در خط 12 مقدار متغیر total را برابر صفر قرار می‌دهیم. سپس برنامه یک پیغام را نشان می‌دهد و از شما می‌خواهد که نمرات دانش آموز را وارد کنید (student + 1). عدد 1 را به student اضافه کرده‌ایم تا به جای نمایش 0 Student، با 1 Student شروع شود، تا طبیعی‌تر به نظر برسد.

سپس به دومین حلقه for در خط 16 می‌رسیم. وظیفه این حلقه گردش در میان دومین بعد که همان نمرات دانش آموز است می‌باشد. برنامه چهار نمره مربوط به دانش آموز را می‌گیرد. هر وقت که برنامه یک نمره را از کاربر دریافت می‌کند، نمره به متغیر total اضافه می‌شود. وقتی همه نمره‌ها وارد شدند، متغیر total هم جمع همه نمرات را نشان می‌دهد. در خطوط 23-24 معدل دانش آموز نشان داده می‌شود. معدل از تقسیم کردن total (جمع) بر تعداد نمرات به دست می‌آید. از size(studentGrades[0]) هم برای به دست آوردن تعداد نمرات استفاده می‌شود.

متد

متدها به شما اجازه می‌دهند که یک رفتار یا وظیفه را تعریف کنید و مجموعه‌ای از کدها هستند که در هر جای برنامه می‌توان از آن‌ها استفاده کرد. متدها دارای آرگومان‌هایی هستند که وظیفه متد را مشخص می‌کنند. نمی‌توان یک متد را در داخل متد دیگر تعریف کرد. وقتی که شما در برنامه یک متد را صدا می‌زنید برنامه به قسمت تعریف متد رفته و کدهای آن را اجرا می‌کند. در C++ متدی وجود دارد که نقطه آغاز هر برنامه است و بدون آن برنامه‌ها نمی‌دانند با ید از کجا شروع شوند، این متد main() نام دارد. پارامترها همان چیزهایی هستند که متد منتظر دریافت آن‌ها است.

آرگومان‌ها مقادیری هستند که به پارامترها ارسال می‌شوند.

گاهی اوقات دو کلمه پارامتر و آرگومان به یک منظور به کار می‌روند. ساده‌ترین ساختار یک متد به صورت زیر است :

```
returnType MethodName(Parameter List)
{
    code to execute;
}
```

به برنامه ساده زیر توجه کنید. در این برنامه از یک متد برای چاپ یک پیغام در صفحه نمایش استفاده شده است :

```
1 #include <iostream>
2 using namespace std;
3
4 void PrintMessage()
5 {
6     cout << "Hello World!";
7 }
8
9 int main()
10 {
11     PrintMessage();
12 }
```

در خطوط 4-7 یک متد تعریف کرده ایم. مکان تعریف آن در داخل کلاس مهم نیست. به عنوان مثال می توانید آن را زیر متد `main()` تعریف کنید. می توان این متد را در داخل متد دیگر صدا زد (فراخوانی کرد). متد دیگر ما در اینجا متد `main()` است که می توانیم در داخل آن نام متدی که برای چاپ یک پیغام تعریف کرده ایم (یعنی متد `PrintMessage()`) را صدا بزنیم. در تعریف متد بالا کلمه کلیدی `void` آمده است که نشان دهنده آن است که متد مقدار برگشتی ندارد. در درس آینده در مورد مقدار برگشتی از یک متد و استفاده از آن برای اهداف مختلف توضیح داده خواهد شد. نام متد ما `PrintMessage()` است. به این نکته توجه کنید که در نامگذاری متد از روش پاسکال (حرف اول هر کلمه بزرگ نوشته می شود) استفاده کرده ایم. این روش نامگذاری قراردادی است و می توان از این روش استفاده نکرد، اما پیشنهاد می شود که از این روش برای تشخیص متدها استفاده کنید. بهتر است در نامگذاری متدها از کلماتی استفاده شود که کار متد را مشخص می کند مثلاً نام هایی مانند `GoToBed` یا `OpenDoor`. همچنین به عنوان مثال اگر مقدار برگشتی متد یک مقدار بولی باشد می توانید اسم متد خود را به صورت یک کلمه سوالی انتخاب کنید مانند `IsLeapyear` یا `IsTeenager`. ولی از گذاشتن علامت سؤال در آخر اسم متد خودداری کنید. دو پرانتزی که بعد از نام می آید نشان دهنده آن است که نام متد متعلق به یک متد است. در این مثال در داخل پرانتزها هیچ چیزی نوشته نشده چون پارامتری ندارد. در درس های آینده در مورد متدها بیشتر توضیح می دهیم.

بعد از پرانتزها دو آکولاد قرار می دهیم که بدنه متد را تشکیل می دهد و کدهایی را که می خواهیم اجرا شوند را در داخل این آکولادها می نویسیم. در داخل متد `main()` متدی را که در خط 11 ایجاد کرده ایم را صدا می زنیم. برای صدا زدن یک متد کافیست نام آن را نوشته و بعد از نام پرانتزها را قرار دهیم.

اگر متد دارای پارامتر باشد باید شما آراگومانها را به ترتیب در داخل پرانتزها قرار دهید. در این مورد نیز در درس های آینده توضیح بیشتری می دهیم. با صدا زدن یک متد کدهای داخل بدنه آن اجرا می شوند. برای اجرای متد `PrintMessage()` برنامه از متد `main()` به محل تعریف متد `PrintMessage()` می رود. مثلاً وقتی ما متد `PrintMessage()` را در خط 11 صدا می زنیم برنامه

از خط 11 به خط 4، یعنی جایی که متد تعریف شده می‌رود. اکنون ما یک متد در برنامه class داریم و همه متدهای این برنامه می‌توانند آن را صدا بزنند.

مقدار برگشتی از یک متد

متدها می‌توانند مقدار برگشتی از هر نوع داده‌ای داشته باشند. این مقادیر می‌توانند در محاسبات یا به دست آوردن یک داده مورد استفاده قرار بگیرند. در زندگی روزمره فرض کنید که کارمند شما یک متد است و شما او را صدا می‌زنید و از او می‌خواهید که کار یک سند را به پایان برساند. سپس از او می‌خواهید که بعد از اتمام کارش سند را به شما تحویل دهد. سند همان مقدار برگشتی متد است. نکته مهم در مورد یک متد، مقدار برگشتی و نحوه استفاده شما از آن است. برگشت یک مقدار از یک متد آسان است. کفایت در تعریف متد به روش زیر عمل کنید :

```
returnType MethodName()
{
    return value;
}
```

returnType در اینجا نوع داده‌ای مقدار برگشتی را مشخص می‌کند (bool، int). در داخل بدنه متد کلمه کلیدی return و بعد از آن یک مقدار یا عبارتی که نتیجه آن یک مقدار است را می‌نویسیم. نوع این مقدار برگشتی باید از انواع ساده بوده و در هنگام نامگذاری متد و قبل از نام متد ذکر شود. اگر متد ما مقدار برگشتی نداشته باشد باید از کلمه void قبل از نام متد استفاده کنیم. مثال زیر یک متد که دارای مقدار برگشتی است را نشان می‌دهد.

```
1  #include <iostream>
2  using namespace std;
3
4  int CalculateSum()
5  {
6      int firstNumber = 10;
7      int secondNumber = 5;
8
9      int sum = firstNumber + secondNumber;
10
11     return sum;
12 }
13
14 int main()
15 {
16     int result = CalculateSum();
17
18     cout << "Sum is " << result;
19 }
```

همانطور که در خط 4 مثال فوق مشاهده می‌کنید هنگام تعریف متد از کلمه `int` به جای `void` استفاده کرده‌ایم که نشان دهنده آن است که متد ما دارای مقدار برگشتی از نوع اعداد صحیح است. در خطوط 6 و 7 دو متغیر تعریف و مقدار دهی شده‌اند.

توجه کنید که این متغیرها، متغیرهای محلی هستند. و این بدان معنی است که این متغیرها در سایر متدها مانند متد `main()` قابل دسترسی نیستند و فقط در متدی که در آن تعریف شده‌اند قابل استفاده هستند. در خط 10 جمع دو متغیر در متغیر `sum` قرار می‌گیرد. در خط 11 مقدار برگشتی `sum` توسط دستور `return` فراخوانی می‌شود. در داخل متد `main()` یک متغیر به نام `result` در خط 16 تعریف می‌کنیم و متد `CalculateSum()` را فراخوانی می‌کنیم.

متد `CalculateSum()` مقدار 15 را بر می‌گرداند که در داخل متغیر `result` ذخیره می‌شود. در خط 18 مقدار ذخیره شده در متغیر `result` چاپ می‌شود. متدی که در این مثال ذکر شد متد کاربردی و مفیدی نیست. با وجودیکه کدهای زیادی در متد بالا نوشته شده ولی همیشه مقدار برگشتی 15 است، در حالیکه می‌توانستیم به راحتی یک متغیر تعریف کرده و مقدار 15 را به آن اختصاص دهیم. این متد در صورتی کارآمد است که پارامترهایی به آن اضافه شود که در درس‌های آینده توضیح خواهیم داد. هنگامی که می‌خواهیم در داخل یک متد از دستور `if` یا `switch` استفاده کنیم باید تمام کدها دارای مقدار برگشتی باشند. برای درک بهتر این مطلب به مثال زیر توجه کنید :

```

1  #include <iostream>
2  using namespace std;
3
4  int GetNumber()
5  {
6      int number;
7
8      cout << "Enter a number greater than 10: ";
9      cin >> number;
10
11     if (number > 10)
12     {
13         return number;
14     }
15     else
16     {
17         return 0;
18     }
19 }
20
21 int main()
22 {
23     int result = GetNumber();
24
25     cout << "Result is " << result;
26 }
```

```

Enter a number greater than 10: 11
Result = 11
```

```
Enter a number greater than 10: 9
Result = 0
```

در خطوط 4-19 یک متد با نام `GetNumber()` تعریف شده است که از کاربر یک عدد بزرگتر از 10 را می‌خواهد. اگر عدد وارد شده توسط کاربر درست نباشد متد مقدار صفر را بر می‌گرداند. و اگر قسمت `else` دستور `if` و یا دستور `return` را از آن حذف کنیم در هنگام اجرای برنامه با پیغام خطا مواجه می‌شویم.

چون اگر شرط دستور `if` نادرست باشد (کاربر مقداری کمتر از 10 را وارد کند) برنامه به قسمت `else` می‌رود تا مقدار صفر را بر گرداند و چون قسمت `else` حذف شده است برنامه با خطا مواجه می‌شود و همچنین اگر دستور `return` حذف شود چون برنامه نیاز به مقدار برگشتی دارد پیغام خطا می‌دهد. و آخرین مطلبی که در این درس می‌خواهیم به شما آموزش دهیم این است که شما می‌توانید از یک متد که مقدار برگشتی ندارد خارج شوید. حتی اگر از نوع داده‌ای `void` در یک متد استفاده می‌کنید باز هم می‌توانید کلمه کلیدی `return` را در آن به کار ببرید. استفاده از `return` باعث خروج از بدنه متد و اجرای کدهای بعد از آن می‌شود.

```
1  #include <iostream>
2  using namespace std;
3
4  void TestReturnExit()
5  {
6      cout << "Line 1 inside the method TestReturnExit()" << endl;
7      cout << "Line 2 inside the method TestReturnExit()" << endl;
8
9      return;
10
11     //The following lines will not execute
12     cout << "Line 3 inside the method TestReturnExit()" << endl;
13     cout << "Line 4 inside the method TestReturnExit()" << endl;
14 }
15
16 int main()
17 {
18     TestReturnExit();
19
20     cout << "Hello World!";
21 }
```

```
Line 1 inside the method TestReturnExit()
Line 2 inside the method TestReturnExit()
Hello World!
```

در برنامه بالا نحوه خروج از متد با استفاده از کلمه کلیدی `return` و نادیده گرفتن همه کدهای بعد از این کلمه کلیدی نشان داده شده است. در پایان برنامه متد تعریف شده (`TestReturnExit()`) در داخل متد `main()` فراخوانی و اجرا می‌شود.

پارامترها و آرگومان‌ها

پارامترها داده‌های خامی هستند که متد آن‌ها را پردازش می‌کند و سپس اطلاعاتی را که به دنبال آن هستید در اختیار شما قرار می‌دهد. فرض کنید پارامترها مانند اطلاعاتی هستند که شما به یک کارمند می‌دهید که بر طبق آن‌ها کارش را به پایان برساند. یک متد می‌تواند هر تعداد پارامتر داشته باشد. هر پارامتر می‌تواند از انواع مختلف داده باشد. در زیر یک متد با N پارامتر نشان داده شده است :

```
returnType MethodName(datatype param1, datatype param2, ... datatype paramN)
{
    code to execute;
}
```

پارامترها بعد از نام متد و بین پرانتزها قرار می‌گیرند. بر اساس کاری که متد انجام می‌دهد می‌توان تعداد پارامترهای زیادی به متد اضافه کرد. بعد از فراخوانی یک متد باید آرگومان‌های آن را نیز تأمین کنید. آرگومان‌ها مقادیری هستند که به پارامترها اختصاص داده می‌شوند. ترتیب ارسال آرگومان‌ها به پارامترها مهم است. عدم رعایت ترتیب در ارسال آرگومان‌ها باعث به وجود آمدن خطای منطقی و خطای زمان اجرا می‌شود. اجازه بدهید که یک مثال بزنیم :

```
1  #include <iostream>
2
3  using namespace std;
4
5  int CalculateSum(int number1, int number2)
6  {
7      return number1 + number2;
8  }
9
10 int main()
11 {
12     int num1, num2;
13
14     cout << "Enter the first number: ";
15     cin >> num1;
16     cout << "Enter the second number: ";
17     cin >> num2;
18
19     cout << "Sum = " << CalculateSum(num1, num2);
20 }
```

```
Enter the first number: 10
Enter the second number: 5
Sum = 15
```

در برنامه بالا یک متد به نام `CalculateSum()` (خطوط 5-8) تعریف شده است که وظیفه آن جمع مقدار دو عدد است. چون این متد مقدار دو عدد صحیح را با هم جمع می‌کند پس نوع برگشتی ما نیز باید `int` باشد. متد دارای دو پارامتر است که اعداد را به آن‌ها ارسال می‌کنیم. به نوع داده‌ای پارامترها توجه کنید. هر دو پارامتر یعنی `number1` و `number2` مقادیری از نوع اعداد صحیح (`int`) دریافت می‌کنند. در بدنه متد دستور `return` نتیجه جمع دو عدد را بر می‌گرداند. در داخل متد `main()` برنامه از کاربر دو مقدار را درخواست می‌کند و آن‌ها را داخل متغیرها قرار می‌دهد. حال متد را که آرگومان‌های آن را آماده کرده‌ایم فراخوانی می‌کنیم. مقدار `num1` به پارامتر اول و مقدار `num2` به پارامتر دوم ارسال می‌شود. حال اگر مکان دو مقدار را هنگام ارسال به متد تغییر دهیم (یعنی مقدار `num2` به پارامتر اول و مقدار `num1` به پارامتر دوم ارسال شود) هیچ تغییری در نتیجه متد ندارد چون جمع خاصیت جابه جایی دارد.

فقط به یاد داشته باشید که باید ترتیب ارسال آرگومان‌ها هنگام فراخوانی متد دقیقاً با ترتیب قرار گیری پارامترها تعریف شده در متد مطابقت داشته باشد. بعد از ارسال مقادیر 10 و 5 به پارامترها، پارامترها آن‌ها را دریافت می‌کنند. به این نکته نیز توجه کنید که نام پارامترها طبق قرارداد به شیوه کوهان شتری یا `camelCasing` (حرف اول دومین کلمه بزرگ نوشته می‌شود) نوشته می‌شود. در داخل بدنه متد (خط 7) دو مقدار با هم جمع می‌شوند و نتیجه به متد فراخوان (متدی که متد `CalculateSum()` را فراخوانی می‌کند) ارسال می‌شود. در درس آینده از یک متغیر برای ذخیره نتیجه محاسبات استفاده می‌کنیم ولی در اینجا مشاهده می‌کنید که می‌توان به سادگی نتیجه جمع را نشان داد (خط 7). در داخل متد `main()` از ما دو عدد که قرار است با هم جمع شوند درخواست می‌شود.

در خط 19 متد `CalculateSum()` را فراخوانی می‌کنیم و دو مقدار صحیح به آن ارسال می‌کنیم. دو عدد صحیح در داخل متد با هم جمع شده و نتیجه آن‌ها برگردانده می‌شود. مقدار برگشت داده شده از متد به وسیله متد `cout` نمایش داده می‌شود (خط 19). در برنامه زیر یک متد تعریف شده است که دارای دو پارامتر از دو نوع داده‌ای مختلف است:

```

1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  void ShowMessageAndNumber(string message, int number)
6  {
7      cout << message << endl;
8      cout << "Number = " << number;
9  }
10
11 int main()
12 {
13     ShowMessageAndNumber("Hello World!", 100);
14 }
```

```
Hello World!  
Number = 100
```

در مثال بالا یک متدی تعریف شده است که اولین پارامتر آن مقداری از نوع رشته و دومین پارامتر آن مقداری از نوع `int` دریافت می‌کند. متد به سادگی دو مقداری که به آن ارسال شده است را نشان می‌دهد. در خط 13 متد را اول با یک رشته و سپس یک عدد خاص فراخوانی می‌کنیم. حال اگر متد به صورت زیر فراخوانی می‌شد :

```
ShowMessageAndNumber(100, "Welcome to Gimme C++!");
```

در برنامه خطا به وجود می‌آید چون عدد 100 به پارامتری از نوع رشته و رشته `Hello World!` به پارامتری از نوع اعداد صحیح ارسال می‌شد. این نشان می‌دهد که ترتیب ارسال آرگومان‌ها به پارامترها هنگام فراخوانی متد مهم است. به مثال 1 توجه کنید در آن مثال دو عدد از نوع `int` به پارامترها ارسال کردیم که ترتیب ارسال آن‌ها چون هردو پارامتر از یک نوع بودند مهم نبود. ولی اگر پارامترهای متد دارای اهداف خاصی باشند ترتیب ارسال آرگومان‌ها مهم است.

```
void ShowPersonStats(int age, int height)  
{  
    cout << "Age = " << age;  
    cout << "Height = " << height;  
}  
  
//Using the proper order of arguments  
ShowPersonStats(20, 160);  
  
//Acceptable, but produces odd results  
ShowPersonStats(160, 20);
```

در مثال بالا نشان داده شده است که حتی اگر متد دو آرگومان با یک نوع داده‌ای قبول کند باز هم بهتر است ترتیب بر اساس تعریف پارامترها رعایت شود. به عنوان مثال در اولین فراخوانی متد بالا اشکالی به چشم نمی‌آید چون سن شخص 20 و قد او 160 سانتی متر است. اگر آرگومان‌ها را به ترتیب ارسال نکنیم سن شخص 160 و قد او 20 سانتی متر می‌شود که به واقعیت نزدیک نیست. دانستن مبانی مقادیر برگشتی و ارسال آرگومان‌ها باعث می‌شود که شما متدهای کارآمد تری تعریف کنید. تکه کد زیر نشان می‌دهد که شما حتی می‌توانید مقدار برگشتی از یک متد را به عنوان آرگومان به متد دیگر ارسال کنید.

```
int MyMethod()  
{  
    return 5;  
}  
  
void AnotherMethod(int number)  
{  
    cout << number;  
}  
  
// Codes skipped for demonstration
```

```
AnotherMethod(MyMethod());
```

چون مقدار برگشتی متد `MyMethod()` عدد 5 است و به عنوان آرگومان به متد `AnotherMethod()` ارسال می‌شود خروجی کد بالا هم عدد 5 است.

ارسال آرگومان‌ها به روش ارجاع

آرگومان‌ها را می‌توان به کمک ارجاع ارسال کرد. این بدان معناست که شما آدرس متغیری را ارسال می‌کنید نه مقدار آن را. ارسال با ارجاع زمانی مفید است که شما بخواهید یک آرگومان که دارای مقدار بزرگی است (مانند یک آبجکت) را ارسال کنید. در این حالت وقتی که آرگومان ارسال شده را در داخل متد اصلاح می‌کنیم مقدار اصلی آرگومان در خارج از متد هم تغییر می‌کند. در زیر دستورالعمل پایه‌ای تعریف پارامترها که در آن‌ها به جای مقدار از آدرس استفاده شده است نشان داده شده است. برای ارسال یک متغیر با ارجاع کافیست که قبل از نام پارامتر یک علامت `&` قرار دهید :

```
returnType MethodName(datatype &param1)
{
    code to execute;
}
```

اجازه دهید که تفاوت بین ارسال با ارجاع و ارسال با مقدار آرگومان را با یک مثال توضیح دهیم :

```
1  #include <iostream>
2  using namespace std;
3
4  void ModifyNumberVal(int number)
5  {
6      number += 10;
7      cout << "Value of number inside method is " << number << endl;
8  }
9
10 void ModifyNumberRef(int &number)
11 {
12     number += 10;
13     cout << "Value of number inside method is " << number << endl;
14 }
15
16 int main()
17 {
18     int num = 5;
19
20     cout << "num = " << num << endl;
21
22     cout << "Passing num by value to method ModifyNumberVal() ..." << endl;
23     ModifyNumberVal(num);
```

```

24     cout << "Value of num after exiting the method is " << num << endl;
25
26     cout << "Passing num by ref to method ModifyNumberRef() ..." << endl;
27     ModifyNumberRef(num);
28     cout << "Value of num after exiting the method is " << num << endl;
29 }

```

```

num = 5
Passing num by value to method ModifyNumberVal() ...
Value of number inside method is 15
Value of num after exiting the method is 5
Passing num by ref to method ModifyNumberRef() ...
Value of number inside method is 15
Value of num after exiting the method is 15

```

در برنامه بالا دو متد که دارای یک هدف یکسان هستند تعریف شده‌اند و آن اضافه کردن عدد 10 به مقداری است که به آن‌ها ارسال می‌شود. اولین متد (خطوط 5-9) دارای یک پارامتر است که نیاز به یک مقدار آرگومان (از نوع int) دارد. وقتی که متد را صدا می‌زنیم و آرگومانی به آن اختصاص می‌دهیم (خط 24)، کپی آرگومان به پارامتر متد ارسال می‌شود. بنابراین مقدار اصلی متغیر خارج از متد هیچ ارتباطی به پارامتر متد ندارد. سپس مقدار 10 را به متغیر پارامتر (number) اضافه کرده و نتیجه را چاپ می‌کنیم. برای اثبات اینکه متغیر num هیچ تغییری نکرده است مقدار آن را یکبار دیگر چاپ کرده و مشاهده می‌کنیم که تغییری نکرده است. دومین متد (خطوط 11-15) نیاز به یک مقدار با ارجاع دارد. در این حالت به جای اینکه یک کپی از مقدار به عنوان آرگومان به آن ارسال شود آدرس متغیر به آن ارسال می‌شود. حال پارامتر به مقدار اصلی متغیر که زمان فراخوانی متد به آن ارسال می‌شود دسترسی دارد. وقتی که ما مقدار متغیر پارامتری که شامل آدرس متغیر اصلی است را تغییر می‌دهیم (خط 13) در واقع مقدار متغیر اصلی در خارج از متد را تغییر داده‌ایم. در نهایت مقدار اصلی متغیر را وقتی که از متد خارج شدیم را نمایش می‌دهیم و مشاهده می‌شود که مقدار آن واقعاً تغییر کرده است.

ارسال آرایه به عنوان آرگومان

می‌توان آرایه‌ها را به عنوان آرگومان به متد ارسال کرد. ابتدا شما باید پارامترهای متد را طوری تعریف کنید که آرایه دریافت کنند. به مثال زیر توجه کنید:

```

1  #include <iostream>
2  using namespace std;
3
4  void TestArray(int numbers[])
5  {
6      for (int i = 0; i <= sizeof(numbers) ; i++)
7      {
8          cout << numbers[i] << endl;

```



```

9     }
10  }
11
12  int main()
13  {
14      int array[] = { 1, 2, 3, 4, 5 };
15
16      TestArray(array);
17  }

```

```

1
2
3
4
5

```

مشاهده کردید که به سادگی می‌توان با گذاشتن کروه بعد از نام پارامتر یک متد ایجاد کرد که پارامتر آن، آرایه دریافت می‌کند. وقتی متد در خط 16 فراخوانی می‌شود، آرایه را فقط با استفاده از نام آن و بدون استفاده از اندیس ارسال می‌کنیم. پس آرایه‌ها هم به روش ارجاع به متدها ارسال می‌شوند. در خطوط 6-9 از حلقه for برای دسترسی به اجزای اصلی آرایه که به عنوان آرگومان به متد ارسال کرده‌ایم استفاده می‌کنیم. در زیر نحوه ارسال یک آرایه به روش ارجاع نشان داده شده است.

```

1  #include <iostream>
2  using namespace std;
3
4  void IncrementElements(int numbers[])
5  {
6      for (int i = 0; i <= sizeof(numbers); i++)
7      {
8          numbers[i]++;
9      }
10 }
11
12 int main()
13 {
14     int array[] = { 1, 2, 3, 4, 5 };
15
16     IncrementElements(array);
17
18     for (int i = 0; i < size(array); i++)
19     {
20         cout << array[i] << endl;
21     }
22 }

```

```

2
3
4
5
6

```

برنامه بالا یک متد را نشان می‌دهد که یک آرایه را دریافت می‌کند و به هر یک از عناصر آن یک واحد اضافه می‌کند. در داخل متد ما مقادیر هر یک از اجزای آرایه را افزایش داده‌ایم. سپس از متد خارج شده و نتیجه را نشان می‌دهیم. مشاهده می‌کنید که هر یک از مقادیر اصلی متد هم اصلاح شده‌اند. می‌توان دو یا چند آرایه را به عنوان آرگومان به متد ارسال کرد :

```
void MyMethod(int param1[], int param2[])
{
    //code here
}
```

محدوده متغیر

متغیرها در C++ دارای محدوده هستند. محدوده یک متغیر به شما می‌گوید که در کجای برنامه می‌توان از متغیر استفاده کرد و یا متغیر قابل دسترسی است. به عنوان مثال متغیری که در داخل یک متد تعریف می‌شود فقط در داخل بدنه متد قابل دسترسی است. می‌توان دو متغیر با نام یکسان در دو متد مختلف تعریف کرد. برنامه زیر این ادعا را اثبات می‌کند :

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

void DemonstrateScope()
{
    int number = 5;

    cout << "number inside method DemonstrateScope() = " << number << endl;
}

int main()
{
    int number = 10;

    DemonstrateScope();

    cout << "number inside the Main method = " << number << endl;
}
```

```
number inside method DemonstrateScope() = 5
number inside the Main method = 10
```

مشاهده می‌کنید که حتی اگر ما دو متغیر با نام یکسان تعریف کنیم که دارای محدوده‌های متفاوتی هستند، می‌توان به هر کدام از آن‌ها مقادیر مختلفی اختصاص داد. متغیر تعریف شده در داخل متد main() هیچ ارتباطی به متغیر داخل متد DemonstrateScope() ندارد. وقتی به مبحث کلاس‌ها رسیدیم در این باره بیشتر توضیح خواهیم داد.

پارامترهای اختیاری

پارامترهای اختیاری همانگونه که از اسمشان پیداست اختیاری هستند و می‌توان به آن‌ها آرگومان ارسال کرد یا نه. این پارامترها دارای مقادیر پیشفرضی هستند. اگر به اینگونه پارامترها آرگومانی ارسال نشود از مقادیر پیشفرض استفاده می‌کنند. به مثال زیر توجه کنید :

```

1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 void PrintMessage(string message = "Welcome to Visual C# Tutorials!")
6 {
7     cout << message << endl;
8 }
9
10 int main()
11 {
12     PrintMessage();
13
14     PrintMessage("Learn C# Today!");
15 }
```

```

Welcome to Visual C# Tutorials!
Learn C# Today!
```

متد () PrintMessage (خطوط 5-8) یک پارامتر اختیاری دارد. برای تعریف یک پارامتر اختیاری می‌توان به آسانی و با استفاده از علامت = یک مقدار را به یک پارامتر اختصاص داد (مثال بالا خط 5). دو بار متد را فراخوانی می‌کنیم. در اولین فراخوانی (خط 12) ما آرگومانی به متد ارسال نمی‌کنیم بنابراین متد از مقدار پیشفرض (Welcome to Visual C# Tutorials!) استفاده می‌کند. در دومین فراخوانی (خط 14) یک پیغام (آرگومان) به متد ارسال می‌کنیم که جایگزین مقدار پیشفرض پارامتر می‌شود. اگر از چندین پارامتر در متد استفاده می‌کنید همه پارامترهای اختیاری باید در آخر بقیه پارامترها ذکر شوند. به مثال‌های زیر توجه کنید.

```

void SomeMethod(int opt1 = 10, int opt2 = 20, int req1, int req2) //ERROR
void SomeMethod(int req1, int opt1 = 10, int req2, int opt2 = 20) //ERROR
void SomeMethod(int req1, int req2, int opt1 = 10, int opt2 = 20) //Correct
```

وقتی متدهای با چندین پارامتر اختیاری فراخوانی می‌شوند باید به پارامترهایی که از لحاظ مکانی در آخر بقیه پارامترها نیستند مقدار اختصاص داد. به یاد داشته باشید که نمی‌توان برای نادیده گرفتن یک پارامتر به صورت زیر عمل کرد :

```

void SomeMethod(int required1, int optional1 = 10, int optional2 = 20)
```

```
{  
    //Some Code  
}  
  
// ... Code omitted for demonstration  
  
SomeMethod(10, , 100); //Error
```

سربارگذاری متدها

سربارگذاری متدها (Method Overloading) به شما اجازه می‌دهد که دو متد با نام یکسان تعریف کنید که دارای امضا و تعداد پارامترهای مختلف هستند. برنامه از روی آرگومان‌هایی که شما به متد ارسال می‌کنید به صورت خودکار تشخیص می‌دهد که کدام متد را فراخوانی کرده‌اید یا کدام متد مد نظر شماست. امضای یک متد نشان دهنده ترتیب و نوع پارامترهای آن است. به مثال زیر توجه کنید :

```
void MyMethod(int x, double y, string z)
```

که امضای متد بالا

```
MyMethod(int, double, string)
```

به این نکته توجه کنید که نوع برگشتی و نام پارامترها شامل امضای متد نمی‌شوند. در مثال زیر نمونه‌ای از سربارگذاری متدها آمده است.

```
1  #include <iostream>  
2  using namespace std;  
3  
4  void ShowMessage(double number)  
5  {  
6      cout << "Double version of the method was called." << endl;  
7  }  
8  
9  void ShowMessage(int number)  
10 {  
11     cout << "Integer version of the method was called." << endl;  
12 }  
13  
14 int main()  
15 {  
16     ShowMessage(9.99);  
17     ShowMessage(9);  
18 }
```

```
Double version of the method was called.  
Integer version of the method was called.
```

در برنامه بالا دو متد با نام مشابه تعریف شده‌اند. اگر سربارگذاری متد توسط ++C پشتیبانی نمی‌شد برنامه زمان زیادی برای انتخاب یک متد از بین متدهایی که فراخوانی می‌شوند لازم داشت. رازی در نوع پارامترهای متد نهفته است. کامپایلر بین دو یا چند متد همنام در صورتی فرق می‌گذارد که پارامترهای متفاوتی داشته باشند. وقتی یک متد را فراخوانی می‌کنیم، متد نوع آرگومان‌ها را تشخیص می‌دهد.

در فراخوانی اول (خط 16) ما یک مقدار double را به متد ShowMessage() ارسال کرده‌ایم در نتیجه متد ShowMessage() (خطوط 7-4) که دارای پارامتری از نوع double اجرا می‌شود. در بار دوم که متد فراخوانی می‌شود (خط 17) ما یک مقدار int را به متد ShowMessage() ارسال می‌کنیم متد ShowMessage() (خطوط 12-9) که دارای پارامتری از نوع int است اجرا می‌شود. معنای اصلی سربارگذاری متد همین است که توضیح داده شد. هدف اصلی از سربارگذاری متدها این است که بتوان چندین متد که وظیفه یکسانی انجام می‌دهند را تعریف کرد.

بازگشت (Recursion)

بازگشت (Recursion)، فرایندی است که در آن متد مدام خود را فراخوانی می‌کند تا زمانی که به یک مقدار مورد نظر برسد. بازگشت یک مبحث پیچیده در برنامه نویسی است و تسلط به آن کار را حتی نیست. به این نکته هم توجه کنید که بازگشت باید در یک نقطه متوقف شود وگرنه برای بی نهایت بار، متد، خود را فراخوانی می‌کند. در این درس یک مثال ساده از بازگشت را برای شما توضیح می‌دهیم. فاکتوریل یک عدد صحیح مثبت ($n!$) شامل حاصل ضرب همه اعداد مثبت صحیح کوچک‌تر یا مساوی آن می‌باشد. به فاکتوریل عدد 5 توجه کنید.

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

بنابراین برای ساخت یک متد بازگشتی باید به فکر توقف آن هم باشیم. بر اساس توضیح بازگشت، فاکتوریل فقط برای اعداد مثبت صحیح است. کوچک‌ترین عدد صحیح مثبت 1 است. در نتیجه از این مقدار برای متوقف کردن بازگشت استفاده می‌کنیم.

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

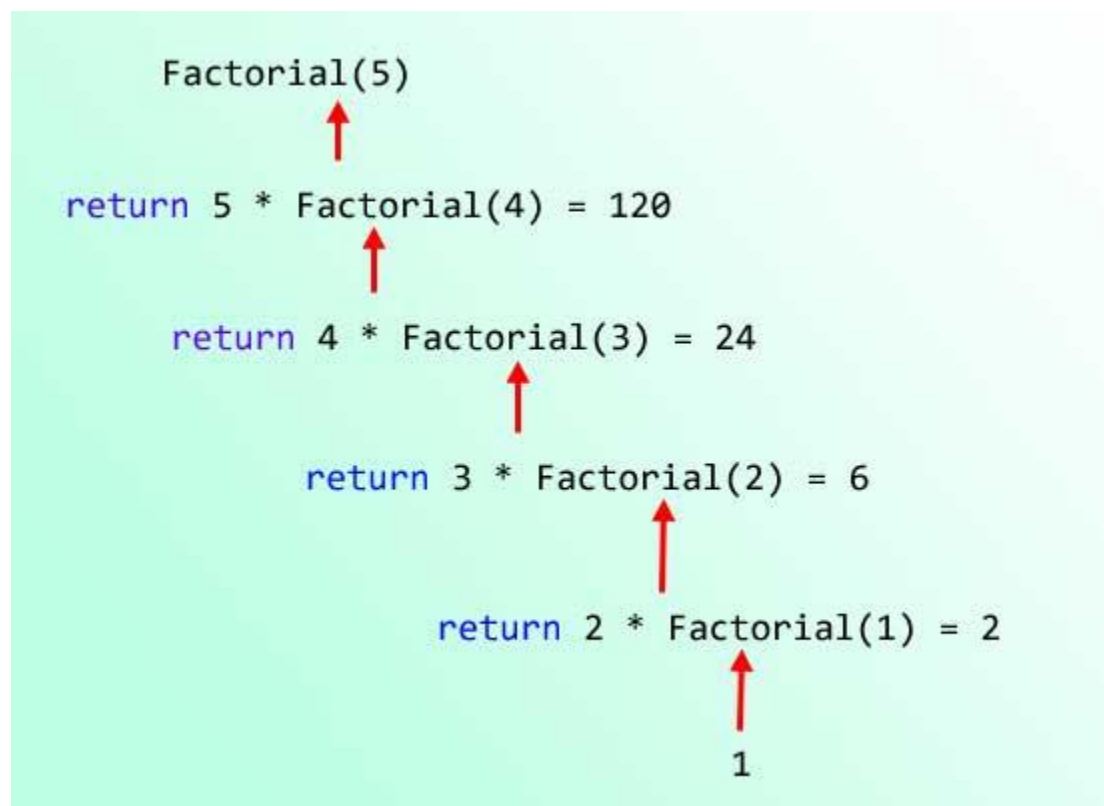
long Factorial(int number)
{
    if (number == 1)
        return 1;

    return number * Factorial(number - 1);
}
```

```
int main()
{
    cout << Factorial(5);
}
```

120

متد مقدار بزرگی را بر می گرداند چون محاسبه فاکتوریل می تواند خیلی بزرگ باشد. متد یک آرگومان که یک عدد است و می تواند در محاسبه مورد استفاده قرار گیرد را می پذیرد. در داخل متد یک دستور if می نویسیم و در خط 7 می گوئیم که اگر آرگومان ارسال شده برابر 1 باشد سپس مقدار 1 را برگردان در غیر اینصورت به خط بعد برو. این شرط باعث توقف تکرارها نیز می شود. در خط 9 مقدار جاری متغیر number در عددی یک واحد کمتر از خودش (number - 1) ضرب می شود. در این خط متد Factorial خود را فراخوانی می کند و آرگومان آن در این خط همان number - 1 است. مثلاً اگر مقدار جاری 10 number باشد یعنی اگر ما بخواهیم فاکتوریل عدد 10 را به دست بیاوریم آرگومان متد Factorial در اولین ضرب 9 خواهد بود. فرایند ضرب تا زمانی ادامه می یابد که آرگومان ارسال شده با عدد 1 برابر نشود. شکل زیر فاکتوریل عدد 5 را نشان می دهد.



کد بالا را به وسیله یک حلقه for نیز می توان نوشت.

```
factorial = 1;

for(int counter = number; counter >= 1; counter--)
    factorial *= counter;
```

این کد از کد معادل بازگشتی آن آسان تر است. از بازگشت در زمینه های خاصی در علوم کامپیوتر استفاده می شود. استفاده از بازگشت حافظه زیادی اشغال می کند پس اگر سرعت برای شما مهم است از آن استفاده نکنید.

شمارش (Enumeration)

Enumeration یا شمارش راهی برای تعریف داده هایی است که می توانند مقادیر محدودی که شما از قبل تعریف کرده اید را بپذیرند. به عنوان مثال شما می خواهید یک متغیر تعریف کنید که فقط مقادیر جهت (جغرافیایی) مانند north, west, east و south را در خود ذخیره کند. ابتدا یک enumeration تعریف می کنید و برای آن یک اسم انتخاب کرده و بعد از آن تمام مقادیر ممکن که می توانند در داخل بدنه آن قرار بگیرند تعریف می کنید. به نحوه تعریف یک enumeration توجه کنید:

```
enum enumName
{
    value1,
    value2,
    value3,
    .
    .
    .
    valueN
};
```

ابتدا کلمه کلیدی enum و سپس نام آن را به کار می بریم. در C++ برای نامگذاری enumeration از روش پاسکال استفاده کنید. در بدنه enum مقادیری وجود دارند که برای هر کدام یک نام در نظر گرفته شده است. به یک مثال توجه کنید :

```
enum Direction
{
    North,
    East,
    South,
    West
};
```

در حالت پیش فرض مقادیری که یک enumeration می تواند ذخیره کند از نوع int هستند. به عنوان مثال مقدار پیش فرض north صفر و مقدار بقیه مقادیر یک واحد بیشتر از مقدار قبلی خودشان است. بنابراین مقدار east برابر 1، مقدار south برابر 2 و مقدار west برابر 3 است. می توانید این مقادیر پیش فرض را به دلخواه تغییر دهید، مانند:

```
enum Direction
```

```
{
    North = 3,
    East  = 5,
    South = 7,
    West  = 9
};
```

اگر به عنوان مثال هیچ مقداری به یک عنصر اختصاص ندهید آن عنصر به صورت خودکار مقدار می گیرد:

```
enum Direction
{
    North = 3,
    East  = 5,
    South,
    West
};
```

در مثال بالا مشاهده می کنید که ما هیچ مقداری به south در نظر نگرفته ایم بنابر این به صورت خودکار یک واحد بیشتر از east یعنی 6 و به west یک واحد بیشتر از south یعنی 7 اختصاص داده می شود. همچنین می توان مقادیر یکسانی برای عناصر enumeration در نظر گرفت. مثال :

```
enum Direction
{
    North = 3,
    East,
    South = North,
    West
};
```

می توانید مقادیر بالا را حدس بزنید؟ مقادیر north, east, south و west به ترتیب 3، 4، 3، 4 است. وقتی مقدار 3 را به north می دهیم مقدار east برابر 4 می شود. سپس وقتی مقدار south را برابر 3 قرار دهیم به صورت اتوماتیک مقدار west برابر 4 می شود. به نحوه استفاده از enumeration در یک برنامه C++ توجه کنید :

```
1 #include <iostream>
2 using namespace std;
3
4 enum Direction
5 {
6     North = 1,
7     East,
8     South,
9     West
10 };
11
```



```

12 int main()
13 {
14     Direction myDirection;
15     myDirection = Direction::North;
16
17     cout << "Direction: " << myDirection;
18
19 }

```

```
Direction: 1
```

ابتدا enumeration را در خطوط 4-10 تعریف می‌کنیم. توجه کنید که enumeration را خارج از کلاس قرار داده‌ایم. این کار باعث می‌شود که enumeration در سراسر برنامه در دسترس باشد. می‌توان enumeration را در داخل کلاس هم تعریف کرد ولی در این صورت فقط در داخل کلاس قابل دسترس است.

```

enum Direction
{
    //Code omitted
};

int main()
{
    //Code omitted
}

```

برنامه را ادامه می‌دهیم. در داخل بدنه enumeration نام چهار جهت جغرافیایی وجود دارد که هر یک از آن‌ها با 1 تا 4 مقدار دهی شده‌اند. در خط 15 یک متغیر تعریف شده است که مقدار یک جهت را در خود ذخیره می‌کند. نحوه تعریف آن به صورت زیر است :

```
enumType variableName;
```

در اینجا enumType نوع داده شمارشی (مثلاً Direction یا مسیر) می‌باشد و variableName نیز نامی است که برای آن انتخاب کرده‌ایم که در مثال قبل myDirection است. سپس یک مقدار به متغیر myDirection اختصاص می‌دهیم (خط 15). برای اختصاص یک مقدار به صورت زیر عمل می‌کنیم :

```
variable = enumType::value;
```

ابتدا نوع Enumeration سپس علامت دو نقطه (::) و بعد مقدار آن (مثلاً North) را می‌نویسیم. می‌توان یک متغیر را فوراً، به روش زیر مقدار دهی کرد :

```
Direction myDirection = Direction::North;
```

حال در خط 17 با استفاده از cout مقدار myDirection را چاپ می‌کنیم. تصور کنید که اگر enumeration نبود شما مجبور بودید که به جای کلمات اعداد را حفظ کنید چون مقادیر enumeration در واقع اعدادی هستند که با نام مستعار توسط شما یا هر کس دیگر تعریف می‌شوند.

اشاره گر (Pointer)

همانطور که می‌دانید، هر متغیر در مکانی از حافظه ذخیره می‌شود. زمانی که یک متغیر را تعریف می‌کنید، بخشی از حافظه برای ذخیره مقدار آن تخصیص داده می‌شود. مکان این بخش از حافظه توسط یک آدرس مشخص می‌شود. این آدرس در حافظه، یک مقدار عددی است و شما می‌توانید با استفاده از نام به آن دسترسی داشته باشید.

ایجاد یک اشاره گر

تعریف یک اشاره گر دقیقاً مانند تعریف یک متغیر از انواع داده دیگر می‌باشد، با این تفاوت که بین نوع داده و نام متغیر یک ستاره (*) قرار می‌گیرد. نوع داده‌ای که برای اشاره گر در نظر می‌گیریم، نوع حافظه‌ای که می‌خواهیم به آن اشاره کنیم را مشخص می‌کند.

```
//pointer to int
int* p;

//pointer to double
double* d;

//pointer to float
float* f;
```

ما می‌توانیم آدرس یک متغیر در حافظه را با استفاده از قرار دادن علامت ampersand یا همان & پشت آن متغیر به دست آوریم (خط 7 کد زیر):

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

int main()
{
    int a = 10;
    cout << "Address of \"a\" variable is : " << &a << endl;

    int* p;
    p = &a;
```

```
cout << "Address of \"p\" variable is : " << p << endl;

cout << "Value of \"p\" variable is : " << *p << endl;

}
```

```
Address of "a" variable is : 0042F714
Address of "p" variable is : 0042F714
Value of "p" variable is : 10
```

به دست آوردن مقدار موجود در یک آدرس

اشاره گر بالا (p) شامل آدرس مکانی در حافظه است که مقدار یک عدد صحیح را نگه داری می‌کند (خط 10). برای به دست آوردن مقدار واقعی ذخیره شده در یک آدرس باید کاراکتر ستاره (*) را قبل از نام اشاره گر قرار دهیم (خط 13). به کاراکتر ستاره در اینجا عملگر dereference گفته می‌شود. زمانی که داخل یک اشاره گر چیزی می‌ریزیم، اگر کاراکتر * را قبل از اشاره گر قرار ندهیم، مقداری که داخل این متغیر می‌ریزیم یک آدرس حافظه در نظر گرفته می‌شود. ولی اگر قبل از آن کاراکتر * را قرار دهیم، مقداری که داخل این متغیر ریخته می‌شود، همان مقدار واقعی ذخیره شده در آن خانه حافظه می‌باشد. اگر دو اشاره گر p و p2 را داشته باشیم، و مقدار p را داخل p2 بریزیم، یک کپی از آدرس مکانی که p به آن اشاره می‌کند را در داخل اشاره گر p2 قرار می‌گیرد:

```
int* p2 = p;
```

اشاره به یک اشاره گر (اشاره گرهای چندگانه)

در برنامه نویسی، اشاره کردن به یک اشاره گر دیگر گاهی اوقات مورد استفاده قرار می‌گیرد. برای انجام این کار، زمانی که می‌خواهیم متغیر مربوطه را تعریف کنیم از دو کاراکتر * استفاده می‌کنیم.

```
int** r = &p; // pointer to p (assigns address of p)
```

برای اینکه این موضوع را به خوبی درک کنید در قالب یک مثال برای شما بیان می‌کنیم:

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     int a = 10;
7
8     int* p;
```

```

9      p = &a;
10
11      *p = 20;
12
13      int** r = &p;
14
15      cout << "Address of \"p\" variable is : " << r << endl;
16      cout << "Address of \"a\" variable is : " << *r << endl;
17      cout << "Value of \"a\" variable is : " << **r << endl;
18  }
```

```

Address of "p" variable is : 002CFABC
Address of "a" variable is : 002CFAC8
Value of "a" variable is : 20
```

قبل از هر توضیحی یک نکته را یادآور می‌شویم و آن این است که، برای تغییر مقدار یک متغیر با استفاده از اشاره گر کافیست قبل از نام یک اشاره گر یک علامت * قرارداده و سپس یک مقدار به اشاره گر اختصاص دهیم کاری که در خط 11 انجام داده‌ایم. در این کد ما ابتدا در عدد صحیح a مقدار 10 را قرار داده‌ایم (خط 6). سپس به کمک dereference، مقدار a را به 20 تغییر دادیم (خط 11). بنابراین اکنون در متغیر a عدد صحیح 20 و در اشاره گر p، آدرس مکانی از حافظه (برای مثال 002CFABC) که مقدار a در آن قرار دارد را داریم.

زمانی که می‌نویسیم &p، با توجه به مطالبی که قبلاً گفته شد، می‌خواهیم آدرس مکانی از حافظه که مقدار p در آن قرار دارد را داشته باشیم. آدرس فعلی p برابر با 002CFABC می‌باشد. بنابراین ما در اینجا می‌خواهیم آدرس مکانی از حافظه که مقدار 002CFABC را دارد را داشته باشیم که برای مثال CFAC8002 می‌باشد. پس مقدار موجود در متغیر r نیز 002CFAC8 می‌باشد. در خطوط 15-17 مقادیر مختلف r را چاپ کرده‌ایم. در خط 15، آدرس مکانی از حافظه که دارای مقدار p است یعنی 002CFABC، در خط 16 همانطور که گفتیم، زمانی که * را پشت یک متغیر قرار می‌دهیم، می‌خواهیم مقدار واقعی ذخیره شده در آدرس r را داشته باشیم که همان p می‌باشد. زمانی که دو کاراکتر * قرار می‌دهیم به این معنی است که می‌خواهیم مقدار واقعی ذخیره شده در آدرس r* را داشته باشیم. با توجه به حالت قبل که مقدار r* برابر با p بود. پس انگار نوشته‌ایم p* و می‌خواهیم مقدار واقعی ذخیره شده در آدرس p را داشته باشیم که همان 20 است.

عملیات ریاضی بر روی اشاره گرها

همانطور که گفته شد، اشاره گرها مقادیر عدد هستند. شما می‌توانید با استفاده از عملگرهای ++، +، - و - عملیات ریاضی را بر روی اشاره گرها انجام دهید. عملگر افزایش (++)، مقدار اشاره گر را بر حسب نوع عملگر، افزایش می‌دهد. به عبارت دیگر، اگر نوع داده اشاره گر ma int باشد، از آنجایی که int در C++ چهار بیتی است، بنابراین زمانی که ما یک اشاره گر از نوع int را یک واحد افزایش می‌دهیم، در حافظه چهار بایت به سمت جلو حرکت می‌کند. برای مثال ما یک اشاره گر از نوع آرایه اعداد صحیح داریم:

```

1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int* a;
7      a = new int[3]{ 5, 3, 9 };
8
9      cout << a      << endl;
10     cout << ++a    << endl;
11     cout << --a    << endl;
12     cout << a + 2  << endl;
13 }

```

```

00570380
00570384
00570380
00570388

```

در ابتدا این اشاره گر، مقدار آدرس شروع در حافظه را ذخیره می‌کند (یعنی آدرس عدد 5). اگر ما `a` را افزایش دهیم:

```
++a;
```

مقدار اشاره گر به آدرس عدد صحیح بعدی در آرایه تغییر پیدا می‌کند. به طور مشابه، عملگر کاهش (`-`) نیز مقدار اشاره گر را بر حسب نوع عملگر، کاهش می‌دهد و آدرس عنصر قبلی در نوع اشاره گر را ذخیره می‌کند. اگر اشاره گر `a` را کاهش دهیم:

```
--a;
```

آدرس عنصر قبلی در آرایه را ذخیره می‌کند. شما همچنین می‌توانید یک عدد صحیح را از اشاره گر کم (تفریق) و یا به آن اضافه (جمع) کنید. اگر به یک اشاره گر مقدار `n` را اضافه کنید، اشاره گر، `n` عنصر متناسب با نوع مشخص شده به جلو حرکت می‌کند. برای مثال ما می‌توانیم مقدار اشاره گر `a` را 2 واحد افزایش دهیم:

```
a = a + 2;
```

با این کار، اشاره گر `a`، آدرس عنصر سوم در آرایه را ذخیره می‌کند (خط 12). حال خطوط 9-12 کد بالا را به صورت زیر تغییر داده و برنامه را اجرا و نتیجه را مشاهده کنید:

```

cout << *(a      ) << endl;
cout << *(++a    ) << endl;
cout << *(--a    ) << endl;
cout << *(a + 2) << endl;

```

اشاره گر Null

زمانی که در یک اشاره گر آدرس معتبری وجود نداشته باشد، مقدار صفر در آن قرار می‌گیرد که به آن null pointer نیز گفته می‌شود. این ویژگی به شما کمک می‌کند تا بتوانید عمل dereference را با اطمینان انجام دهید. زیرا یک اشاره گر معتبر هرگز مقدار صفر را نخواهد داشت. اگرچه ما حافظه تخصیص داده شده به d را با استفاده از delete آزاد کردیم ولی d همچنان به مکانی از حافظه که غیر قابل دسترس است اشاره می‌کند. اگر تلاش کنیم تا عمل dereference را بر روی d انجام دهیم، باعث بروز خطا در زمان اجرا می‌شود. برای جلوگیری از این مشکل، باید مقدار حافظه حذف شده را صفر قرار دهیم. به این نکته توجه داشته باشید که اگر یک حافظه را با استفاده از delete حذف کنیم و سپس مقدار صفر را در آن قرار دهیم (null pointer) و دوباره آن را حذف کنیم مشکلی به وجود نمی‌آید. ولی اگر مقدار اشاره گر را برابر با صفر قرار نداده باشیم و تلاش کنیم تا آن دوباره را حذف کنیم، ممکن است باعث متوقف شدن برنامه شود.

```
d = 0; // mark as null pointer

delete d; // safe
```

از آنجایی که همیشه نمی‌دانیم که یک اشاره گر معتبر است یا خیر، بنابراین بهتر است قبل از آنکه یک اشاره گر را dereference کنیم، مطمئن شویم که مقدار آن صفر نباشد:

```
if (d != 0)
{
    *d = 10;
}
```

ثابت NULL برای مشخص کردن یک null pointer مورد استفاده قرار می‌گیرد. NULL در C++ معمولاً صفر در نظر گرفته می‌شود. این ثابت در فایل کتابخانه استاندارد stdio.h قرار دارد که می‌تواند از طریق iostream در دسترس باشد.

```
#include <iostream>

if (d != NULL) { *d = 10; } // check for null pointer
```

استفاده از اشاره گرها به عنوان پارامتر یک تابع

زمانی که شما یک متغیر را به یک متد ارسال می‌کنید، شما فقط مقدار متغیر را ارسال کنید و نمی‌توانید خود متغیر را در متد تغییر دهید. اما گاهی اوقات تغییر مقدار متغیرها در داخل یک متد می‌تواند کاربردی باشد. برای انجام این کار کافی است که شما آدرس

متغیر را به متد ارسال کنید. این به این معنی است که پارامتر متد یک اشاره گر است و متد می‌تواند به مکان متغیری که به آن پاس داده شده در حافظه دسترسی داشته باشد. به مثال زیر توجه کنید:

```

1  #include <iostream>
2  using namespace std;
3
4  void changeA(int* a)
5  {
6      (*a) = 10;
7  }
8
9  int main()
10 {
11     int *k = new int;
12     (*k) = 2;
13
14     cout << "k before calling function " << *k << endl;
15     changeA(k);
16     cout << "k after calling function " << *k << endl;
17 }
```

```

k before calling function 2
k after calling function 10
```

در کد بالا پارامتر متد `int* a` می‌باشد که یک اشاره گر است و در داخل متد می‌توانیم به راحتی مقدار متغیر پاس داده شده را تغییر دهیم. برای تست این که آیا واقعاً مقدار متغیر عوض شده یا خیر، ابتدا یک اشاره گر از نوع `int` به نام `k` ایجاد کردیم و مقدار 2 را در آن قرار دادیم (خط 12). در خط 14 مقدار فعلی `k` را که همان 2 است را چاپ می‌کنیم و در خط 15 متد `changeA` را با ارسال `k` به عنوان پارامتر فراخوانی کردیم. و در خط 16 دوباره مقدار `k` را چاپ کردیم.

ارسال آرایه به یک تابع

شما می‌توانید یک آرایه را به عنوان یک اشاره گر به یک متد پاس دهید. دلیل اینکه این کار را می‌توانید انجام دهید این است که نام یک آرایه، یک اشاره گر به ابتدای آرایه است.

```

#include <iostream>
using namespace std;

int sum(int* arr, int size)
{
    int sum = 0;
    for (int i = 0; i != size; ++i)
        sum += arr[i];
    return sum;
}
```

```
int main()
{
    int myArr[5];

    cout << "Sum of myArr's elements is " << sum(myArr, 5);
}
```

```
Sum of myArr's elements is -4
```

زمانی که شما یک آرایه را به صورت بالا (خط 14) تعریف می‌کنید، myArr یک اشاره گر به ابتدای آرایه است. در خطوط 4-10 یک متد تعریف کرده‌ایم که مجموع عناصر یک آرایه را حساب کند. این متد دو پارامتر دریافت می‌کند. یکی اشاره گر و دیگری سایز آرایه می‌باشد (خط 4). سپس این آرایه را با مقادیر دلخواه پر (خط 7) و در خط 16 با فراخوانی متد مجموع عناصر آن را حساب کنیم. در اینجا (sum(myArr, 5) آرایه myArr را با استفاده از اشاره گری که ابتدای آرایه را نشان می‌دهد به متد sum پاس داده‌ایم.

مراجع (References)

مراجع به یک برنامه نویس اجازه می‌دهند تا یک نام جدید را برای یک متغیر ایجاد کند. مراجع نسبت به اشاره گرها ساده‌تر، قابل اطمینان تر ولی با قدرت کمتر هستند.

ایجاد یک مرجع

تعریف یک مرجع مانند تعریف یک متغیر معمولی است با این تفاوت که یک ampersand یا & بین نوع داده و نام متغیر قرار می‌گیرد. علاوه بر این، در همان زمانی که یک مرجع تعریف می‌شود باید با یک متغیر از نوع داده مشخص شده مقدار دهی اولیه شود.

```
int x = 5;
int& r = x; // r is an alias to x
int &s = x; // alternative syntax
```

اولین باری که یک مرجع مقداردهی شد، هرگز نمی‌تواند به جای متغیر دیگری بنشیند. مرجع به یک نام مستعار برای متغیر تبدیل می‌شود و می‌تواند دقیقاً مانند همان متغیر اصلی مورد استفاده قرار گیرد.

```
r = 10; // assigns value to r/x
```


مراجع و اشاره گرها

یک مرجع شبیه به یک اشاره گر است که همیشه به یک چیز اشاره می‌کند. در حالی که یک اشاره گر، متغیری است که می‌تواند به دیگر متغیرها اشاره کند، یک مرجع فقط یک نام مستعار است و آدرس خودش را ندارد.

```
int* ptr = &x; // ptr assigned address to x
```

استفاده از مرجع و اشاره گر

هر زمان که شما به یک اشاره گر نیاز دارید ولی هرگز نمی‌خواهید آن را تغییر دهید، بهتر است از مرجع به جای اشاره گر استفاده کنید زیرا از آنجایی که می‌خواهید همیشه به یک متغیر اشاره کنید، یک مرجع قابل اطمینان تر از یک اشاره گر است. این به این معنی است که در یک مرجع نیاز ندارید تا بررسی کنید که آیا این مرجع به null اشاره می‌کند یا خیر، در حالی که در اشاره گرها بهتر است تا این بررسی انجام شود. همچنین احتمال آن وجود دارد که یک مرجع نامعتبر شود. برای مثال اگر یک مرجع به یک null pointer اشاره کند مقدارش نامعتبر می‌شود. اما جلوگیری از این نوع خطاها، زمانی که با یک مرجع کار می‌کنیم آسان‌تر از اشاره گرهاست.

```
int* ptr = 0; // null pointer
int& ref = *ptr;
ref = 10; // segmentation fault (invalid memory access)
```

تبدیل ضمنی

یک تبدیل ضمنی به صورت خودکار توسط کامپایلر انجام می‌شود. برای مثال تمام تبدیل‌ها بین انواع داده اولیه (مانند int، long، double و ...) می‌تواند به صورت ضمنی توسط کامپایلر انجام شود.

```
long number1 = 5;           // int implicitly converted to long
double number2 = number1;   // long implicitly converted to double
```

تبدیل‌های ضمنی به دو دسته تقسیم می‌شوند:

- تبدیل ضمنی به نوع بالاتر (promotion)

- تبدیل ضمنی به نوع پایین تر (demotion)

تبدیل Promotion زمانی اتفاق می افتد که نوع داده ما به صورت ضمنی به یک نوع داده بزرگتر تبدیل شود و تبدیل demotion زمانی اتفاق می افتد که نوع داده ما به صورت ضمنی به یک نوع داده کوچکتر تبدیل شود. تبدیل به یک نوع پایین تر می تواند باعث از بین رفتن بخشی از داده ما شود و این نوع تبدیل ها در بیشتر کامپایلرها به عنوان یک هشدار نمایش داده می شود. اگر هدف واقعی شما از بین رفتن بخشی از داده است، برای برطرف شدن هشدارها می توانید این تبدیل به نوع پایین تر را به صورت صریح انجام دهید.

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     // Promotion
7     long longVar = 5;
8     double doubleVar = longVar;
9
10    // Demotion
11    int intVar = 10.5;
12    bool boolVar = intVar;
13
14    cout << longVar << endl;
15    cout << doubleVar << endl;
16
17    cout << intVar << endl;
18    cout << boolVar << endl;
19 }
```

```
5
5
10
1
```

همانطور که در خطوط 7-8 کد بالا مشاهده می کنید، در این خطوط تبدیل ضمنی از نوع Promotion اتفاق افتاده در نتیجه اصلاحات از بین نمی روند. ولی چون در خطوط 11-12 ما یک مقدار از یک نوع بزرگتر را در یک نوع کوچکتر قرار داده ایم، بخشی از اطلاعات از بین می روند.

تبدیل صریح

تبدیل صریح از زبان C به ارث رسیده است که به آن C-style cast گفته می‌شود. برای مثال به منظور تبدیل صریح یک عدد اعشاری به یک عدد صریح که منجر به از بین رفتن بخش اعشاری آن می‌شود، نوع داده int را داخل یک پرانتز و سمت چپ عدد اعشاری مورد نظر قرار می‌دهیم:

```
int intVar = (int)10.5;    // double demoted to int
char charVar = (char)intVar; // int demoted to char
```

C-style cast برای انجام تبدیل‌های صریح بین انواع داده اولیه مناسب است که از زبان C به ارث رسیده است. از C-style cast می‌توان برای تبدیل صریح بین کلاس‌ها و اشاره گر‌ها نیز استفاده کرد ولی به منظور کنترل بیشتر بر روی انواع تبدیل‌ها، در C++ چهار نوع تبدیل صریح جدید که به آن new-style cast می‌گویند ارائه شده است. این تبدیل‌ها عبارتند از:

```
static_cast    <new_type> (expression)
reinterpret_cast<new_type> (expression)
const_cast    <new_type> (expression)
dynamic_cast   <new_type> (expression)
```

همانطور که در بالا نیز مشاهده می‌کنید، در این ساختار جدید عبارتی که می‌خواهیم تبدیل کنیم (expression) را داخل پرانتز قرار می‌دهیم و به جای new_type، نوعی که می‌خواهیم به آن تبدیل شود را می‌نویسیم. این cast ها کنترل دقیق‌تری را در مورد نحوه انجام تبدیل‌ها در اختیار ما قرار می‌دهند. تفاوت new-style cast با C-style cast این است که اگر یک تبدیل به هر دلیلی انجام نشود، اگر از new-style cast استفاده کرده باشیم، در زمان کامپایل و اگر از C-style cast استفاده کرده باشیم، در زمان اجرا با خطا رو به رو می‌شویم.

Static cast

از static cast برای تبدیل بین انواع متناسب به یک دیگر مورد استفاده قرار می‌گیرد. این نوع تبدیل شبیه به C-style cast است با این تفاوت که محدودیت بیشتری دارد. برای مثال، C-style cast به شما اجازه می‌دهد تا با استفاده از یک اشاره گر از نوع عدد صحیح را به یک کاراکتر اشاره کنید:

```
char charVar = 10;    // 1 byte
int *p = (int*)&charVar; // 4 bytes
```

از آنجایی که این تبدیل باعث می‌شود تا با یک اشاره گر 4 بایتی به 1 بایت از حافظه اشاره کنیم، این تبدیل در زمان اجرا یا با خطا رو به رو می‌شود و یا بر روی حافظه‌های همجوار می‌نویسد.

```
*p = 5; // run-time error: stack corruption
```

بر خلاف C-style cast، در static cast کامپایلر با بررسی اینکه آیا نوع داده اشاره گرها با یکدیگر متناسب است یا خیر، به برنامه نویس اجازه می‌دهد تا این تبدیل نادرست را در زمان کامپایل متوجه شود.

```
int *q = static_cast(&charVar); // compile-time error
```

Reinterpret cast

برای آنکه مشابه کاری که به صورت پشت صحنه در C-style cast اتفاق می‌افتد، کامپایلر را مجبور کنیم تا یک تبدیل را انجام دهد، می‌توانیم از reinterpret cast استفاده کنیم.

```
int *r = reinterpret_cast<int*>(&charVar); // forced conversion
```

با استفاده از این تبدیل می‌توانیم انواعی که با یک دیگر تناسبی ندارند (مانند تبدیل یک اشاره گر از نوع عدد صحیح به یک اشاره گر از نوع کاراکتر) را به یک دیگر تبدیل کنیم. این کار را به سادگی و فقط با یک کپی باینری از داده‌ها، بدون هیچ گونه تغییری در الگوی بیت‌ها انجام می‌شود. توجه داشته باشید که نتیجه این کار در سیستم عامل‌های مختلف می‌تواند متفاوت باشد و در همه جا یکسان نیست. بنابراین از این نوع تبدیل باید با احتیاط استفاده شود.

Const cast

سومین تبدیل صریح const cast می‌باشد. یکی از کاربردهای این تبدیل، افزودن یا حذف مقدار یک متغیر ثابت (constant) می‌باشد.

```
const int myConst = 5;  
int *nonConst = const_cast<int*>(&myConst); // removes const
```

اگرچه const cast به شما اجازه می‌دهد تا مقدار یک ثابت را تغییر دهید، ولی اگر این مقدار ثابت در بخش فقط خواندنی (read-only) حافظه قرار داشته باشد می‌تواند باعث بروز خطا در زمان اجرا شود.

```
*nonConst = 10; // potential run-time error
```

اگر تابعی مانند زیر داشته باشیم که به عنوان آرگومان یک اشاره گر غیر ثابت را بپذیرد:

```
void print(int *p) { std::cout << *p; }
```

حتی اگر متد هیچ تغییری بر روی آن انجام ندهد، نمی‌توانیم یک متغیر از نوع ثابت را به آن پاس دهیم و در صورت انجام این کار با خطا رو به رو می‌شویم. برای جلوگیری از این خطا، مقدار ثابت خودمان را با استفاده از `const cast` به متد ارسال می‌کنیم.

```
print(&myConst); // error: cannot convert
                // const int* to int*

print(nonConst); // allowed
```

new-style cast و C-style cast

به خاطر داشته باشید که شما با استفاده از `C-style cast` نیز می‌توانید یک مقدار `const` تغییر دهید ولی بهتر است از `new-style cast` برای این کار استفاده کنید. دلیل دیگر برای استفاده از `new-style cast` به جای `C-style cast` این است که پیدا کردن آن‌ها در سورس کد نسبت به `C-style cast` آسان‌تر است. به صورت کلی پیدا کردن خطاهای تبدیل صریح، می‌تواند سخت‌تر باشد. دلیل سوم این است که نوشتن `new-style cast` کمی ناخوشایند و سخت‌تر از `C-style cast` است و همین ناخوشایند بودن به برنامه نویس کمک می‌کند تا به دنبال راه‌های دیگری به غیر از `cast` بگردد و به صورت بی‌رویه از این نوع تبدیل‌ها در برنامه نویسی استفاده نکند.

برنامه نویسی شیء گرا (Object Oriented Programming)

برنامه نویسی شیء گرا (OOP) شامل تعریف کلاسها و ساخت اشیاء مانند ساخت اشیاء در دنیای واقعی است. برای مثال یک ماشین را در نظر بگیرید. این ماشین دارای خواصی مانند رنگ، سرعت، مدل، سازنده و برخی خواص دیگر است. همچنین دارای رفتارها و حرکتی مانند شتاب و پیچش به چپ و راست و ترمز است. اشیاء در ++C تقلیدی از یک شیء مانند ماشین در دنیای واقعی هستند. برنامه نویسی شیء گرا با استفاده از کدهای دسته بندی شده کلاسها و اشیاء را بیشتر قابل کنترل می‌کند. در ابتدا ما نیاز به تعریف یک کلاس برای ایجاد اشیاءمان داریم. شیء در برنامه نویسی شیء گرا از روی کلاسی که شما تعریف کرده‌اید ایجاد می‌شود. برای مثال نقشه ساختمان شما یک کلاس است که ساختمان از روی آن ساخته شده است. کلاس شامل خواص یک ساختمان مانند مساحت، بلندی و مواد مورد استفاده در ساخت خانه می‌باشد. در دنیای واقعی ساختمان‌ها نیز بر اساس یک نقشه (کلاس) پایه گذاری (تعریف) شده‌اند.

برنامه نویسی شیء گرا یک روش جدید در برنامه نویسی است که بوسیله برنامه نویسان مورد استفاده قرار می‌گیرد و به آنها کمک می‌کند که برنامه‌هایی با قابلیت استفاده مجدد، خوانا و راحت طراحی کنند. ++C نیز یک برنامه شیء گراست. در درس زیر به شما

نحوه تعریف کلاس و استفاده از اشیاء آموزش داده خواهد شد. همچنین شما با دو مفهوم وراثت و چند ریختی که از مباحث مهم در برنامه نویسی شیء گرا هستند در آینده آشنا می شوید.

کلاس

کلاس به شما اجازه می دهد یک نوع داده ای که توسط کاربر تعریف می شود و شامل فیلدها و متدها است را ایجاد کنید. کلاس در حکم یک نقشه برای یک شیء می باشد. شیء یک چیز واقعی است که از ساختار، خواص و یا رفتارهای کلاس پیروی می کند. وقتی یک شیء می سازید یعنی اینکه یک نمونه از کلاس ساخته اید (در درس ممکن است از کلمات شیء و نمونه به جای هم استفاده شود). برای تعریف کلاس از کلمه کلیدی class استفاده می شود. نحوه تعریف کلاس در زیر آمده است :

```
class ClassName
{
    field1;
    field2;
    ...
    fieldN;

    method1;
    method2;
    ...
    methodN;
};
```

این کلمه کلیدی را قبل از نامی که برای کلاس نام انتخاب می کنیم می نویسیم. در نامگذاری کلاس ها هم از روش نامگذاری Pascal استفاده می کنیم. در بدنه کلاس فیلدها و متدهای آن قرار داده می شوند. بعد از آکولاد بسته هم علامت سمیکالن (;) قرار می گیرد. در زیر نحوه تعریف و استفاده از یک کلاس ساده به نام person نشان داده شده:

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  class Person
6  {
7      public:
8          string  name;
9          int     age;
10         double  height;
11
12         public: void TellInformation()
13         {
14             cout << "Name: "    << name    << endl;
15             cout << "Age: "     << age     << " years old" << endl;
```

```

16         cout << "Height: " << height << "cm" << endl;
17     }
18 };
19
20 int main()
21 {
22     Person firstPerson;
23     Person secondPerson;
24
25     firstPerson.name = "Jack";
26     firstPerson.age = 21;
27     firstPerson.height = 160;
28     firstPerson.TellInformation();
29
30     cout << endl; //Separator
31
32     secondPerson.name = "Mike";
33     secondPerson.age = 23;
34     secondPerson.height = 158;
35     secondPerson.TellInformation();
36 }

```

```

Name: Jack
Age: 21 years old
Height: 160cm

Name: Mike
Age: 23 years old
Height: 158cm

```

برنامه بالا شامل دو کلاس Person و Program می‌باشد. می‌دانیم که کلاس Program شامل متد main() است که برنامه برای اجرا به آن احتیاج دارد ولی اجازه دهید که بر روی کلاس Person تمرکز کنیم. در خطوط 5-17 کلاس Person تعریف شده است. در خط 5 یک نام به کلاس اختصاص داده‌ایم تا به وسیله آن قابل دسترسی باشد. در داخل بدنه کلاس فیلدهای آن تعریف شده‌اند (خطوط 9-7) و سطح دسترسی آنها را public قرار داده‌ایم تا در خارج از کلاس قابل دسترسی باشند. درباره سطوح دسترسی در یک درس جداگانه بحث خواهیم کرد. این سه فیلد تعریف شده خصوصیات واقعی یک فرد در دنیای واقعی را در خود ذخیره می‌کنند. یک فرد در دنیای واقعی دارای نام، سن و قد می‌باشد. در خطوط 11-16 یک متد هم در داخل کلاس به نام TellInformation() تعریف شده است که رفتار کلاس‌مان است و مثلاً اگر از فرد سوالی بپرسیم در مورد خودش چیزهایی می‌گوید. در داخل متد کدهایی برای نشان دادن مقادیر موجود در فیلدها نوشته شده است. نکته‌ای درباره فیلدها وجود دارد و این است که چون فیلدها در داخل کلاس تعریف و به عنوان اعضای کلاس در نظر گرفته شده‌اند محدوده آنها یک کلاس است.

این بدین معناست که فیلدها فقط می‌توانند در داخل کلاس یعنی جایی که به آن تعلق دارند و یا به وسیله نمونه ایجاد شده از کلاس مورد استفاده قرار بگیرند. در داخل متد main() و در خطوط 21-22 دو نمونه یا دو شیء از کلاس Person ایجاد می‌کنیم.

برای ایجاد یک نمونه از یک کلاس ابتدا نام کلاس و سپس نامی که برای شیء قرار است انتخاب کنیم و در نهایت علامت سمیکالن می‌گذاریم :

```
ClassName ObjectName ;
```

وقتی نمونه کلاس ایجاد شد، سازنده را صدا می‌زنیم. یک سازنده متد خاصی است که برای مقداردهی اولیه به فیلدهای یک شیء به کار می‌رود. وقتی هیچ آرگومانی در داخل پرانتزها قرار ندهید، کلاس یک سازنده پیشفرض بدون پارامتر را فراخوانی می‌کند. درباره سازنده‌ها در درس‌های آینده توضیح خواهیم داد. در خطوط 24-26 مقداری به فیلدهای اولین شیء ایجاد شده از کلاس (first Person) اختصاص داده شده است. برای دسترسی به فیلدها یا متدهای یک شیء از علامت نقطه (دات) استفاده می‌شود. به عنوان مثال کد firstPerson.name نشان دهنده فیلد name از شیء firstPerson می‌باشد. برای چاپ مقادیر فیلدها باید متد TellInformation() شیء firstPerson را فراخوانی می‌کنیم. در خطوط 31-33 نیز مقداری به شیء دومی که قبلاً از کلاس ایجاد شده تخصیص می‌دهیم و سپس متد TellInformation() را فراخوانی می‌کنیم. به این نکته توجه کنید که firstPerson و secondPerson نسخه‌های متفاوتی از هر فیلد دارند بنابراین تعیین یک نام برای secondPerson هیچ تاثیری بر نام firstPerson ندارد. در مورد اعضای کلاس در درسهای آینده توضیح خواهیم داد.

سازنده‌ها (Constructors)

سازنده‌ها متدهای خاصی هستند که وجود آنها برای ساخت اشیا لازم است. آنها به شما اجازه می‌دهند که مقداری را به هر یک از فیلدها اختصاص دهید و کدهایی که را که می‌خواهید هنگام ایجاد یک شیء اجرا شوند را به برنامه اضافه کنید. اگر از هیچ سازنده‌ای در کلاستان استفاده نکنید، کامپایلر از سازنده پیشفرض که یک سازنده بدون پارامتر است استفاده می‌کند. می‌توانید در برنامه‌تان از تعداد زیادی سازنده استفاده کنید که دارای پارامترهای متفاوتی باشند. در مثال زیر یک کلاس که شامل سازنده است را مشاهده می‌کنید:

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  class Person
6  {
7  public:
8      string  name;
9      int     age;
10     double  height;
11
12     //Explicitly declare a default constructor
13     Person()
```



```

14     {
15     }
16
17     //Constructor that has 3 parameters
18     Person(string n, int a, double h)
19     {
20         name = n;
21         age = a;
22         height = h;
23     }
24
25     void ShowInformation()
26     {
27         cout << "Name: " << name << endl;
28         cout << "Age: " << age << " years old" << endl;
29         cout << "Height: " << height << "cm" << endl;
30     }
31 };
32
33 int main()
34 {
35     Person firstPerson;
36     Person secondPerson("Mike", 23, 158);
37
38     firstPerson.name = "Jack";
39     firstPerson.age = 21;
40     firstPerson.height = 160;
41     firstPerson.ShowInformation();
42
43     cout << endl; //Seperator
44
45     secondPerson.ShowInformation();
46 }

```

```

Name: Jack
Age: 21 years old
Height: 160cm

Name: Mike
Age: 23 years old
Height: 158cm

```

همانطور که مشاهده می‌کنید در مثال بالا دو سازنده را به کلاس Person اضافه کرده‌ایم. یکی از آنها سازنده پیشفرض (خطوط 10-12) و دیگری سازنده‌ای است که سه آرگومان قبول می‌کند (خطوط 15-20). به این نکته توجه کنید که سازنده درست شبیه به یک متد است با این تفاوت که

- نه مقدار برگشتی دارد و نه از نوع void است.
- نام سازنده باید دقیقاً شبیه نام کلاس باشد.

سازنده پیشفرض در داخل بدنه اش هیچ چیزی ندارد و وقتی فراخوانی می شود که ما از هیچ سازنده ای در کلاسمان استفاده نکنیم. در آینده متوجه می شوید که چطور می توان مقادیر پیشفرضی به اعضای داده ای اختصاص داد، وقتی که از یک سازنده پیشفرض استفاده می کنید. به دومین سازنده توجه کنید. اولاً که نام آن شبیه نام سازنده اول است. سازنده ها نیز مانند متدها می توانند سربرگذاری شوند. حال اجازه دهید که چطور می توانیم یک سازنده خاص را هنگام تعریف یک نمونه از کلاس فراخوانی کنیم.

```
Person firstPerson;

Person secondPerson("Mike", 23, 158);
```

در اولین نمونه ایجاد شده از کلاس Person از سازنده پیشفرض استفاده کرده ایم چون پارامتری برای دریافت آرگومان ندارد. در دومین نمونه ایجاد شده، از سازنده ای استفاده می کنیم که دارای سه پارامتر است. کد زیر تأثیر استفاده از دو سازنده مختلف را نشان می دهد :

```
firstPerson.name    = "Jack";
firstPerson.age     = 21;
firstPerson.height  = 160;
firstPerson.ShowInformation();

cout << endl; //Seperator

secondPerson.ShowInformation();
```

همانطور که مشاهده می کنید لازم است که به فیلدهای شیء ای که از سازنده پیشفرض استفاده می کند مقادیری اختصاص داده شود تا این شیء نیز با فراخوانی متد ShowInformation() آن ها را نمایش دهد. حال به شیء دوم که از سازنده دارای پارامتر استفاده می کند توجه کنید، مشاهده می کنید که با فراخوانی متد ShowInformation() همه چیز همانطور که انتظار می رود اجرا می شود. این بدین دلیل است که شما هنگام تعریف نمونه و از قبل مقادیری به هر یک از فیلدها اختصاص داده اید بنابراین آنها نیاز به مقدار دهی مجدد ندارند مگر اینکه شما بخواهید این مقادیر را اصلاح کنید.

اختصاص مقادیر پیشفرض به سازنده پیشفرض

در مثالهای قبلی یک سازنده پیشفرض با بدنه خالی نشان داده شد. شما می توانید به بدنه این سازنده پیشفرض کدهایی اضافه کنید. همچنین می توانید مقادیر پیشفرضی به فیلدهای آن اختصاص دهید.

```
Person()
{
    name = "No Name";
```

```

    age    = 0;
    height = 0;
}

```

همانطور که در مثال بالا می‌بینید سازنده پیشفرض ما چیزی برای اجرا دارد. اگر نمونه‌ای ایجاد کنیم که از این سازنده پیشفرض استفاده کند، نمونه ایجاد شده مقادیر پیشفرض سازنده پیشفرض را نشان می‌دهد.

```

Person person1;

person1.ShowInformation();

```

```

Name: No Name
Age: 0 years old
Height: 0cm

```

استفاده از کلمه کلیدی this

راهی دیگر برای ایجاد مقادیر پیشفرض استفاده از کلمه کلیدی this است. فرض کنید نام پارامترهای متد کلاس شما یا سازنده، شبیه نام یکی از فیلدها باشد.

```

Person(string name, int age, double height)
{
    name    = name;
    age     = age;
    height  = height;
}

```

این نوع کدنویسی ابهام بر انگیز است و کامپایلر نمی‌تواند متغیر را تشخیص داده و مقداری به آن اختصاص دهد. اینجاست که از کلمه کلیدی this استفاده می‌کنیم.

```

Person(string name, int age, double height)
{
    this->name    = name;
    this->age     = age;
    this->height  = height;
}

```

قبل از هر فیلدی کلمه کلیدی this را می‌نویسیم و نشان می‌دهیم که این همان چیزی است که می‌خواهیم به آن مقداری اختصاص دهیم. کلمه کلیدی this ارجاع یک شیء به خودش را نشان می‌دهد.

مخرب‌ها (Destructors)

مخرب‌ها نقطه مقابل سازنده‌ها هستند. مخرب‌ها متدهای خاصی هستند که هنگام تخریب یک شیء فراخوانی می‌شوند. اشیا از حافظه کامپیوتر استفاده می‌کنند و اگر پاک نشوند ممکن است با کمبود حافظه مواجه شوید. می‌توان از مخرب‌ها برای پاک کردن منابعی که در برنامه مورد استفاده قرار نمی‌گیرند استفاده کرد. با استفاده از مخرب‌ها، کدی را تعریف می‌کنید که وقتی اجرا می‌شود که یک شیء تخریب شده باشد. معمولاً شیء وقتی تخریب می‌شود که از محدوده خارج شود. دستور نوشتن مخرب کمی با سازنده متفاوت است :

```
~ClassName()  
{  
    code to execute;  
}
```

مانند سازنده‌ها، مخرب‌ها باید همانم کلاسی باشند که در آن تعریف شده‌اند. به این نکته توجه کنید که قبل از نام مخرب علامت (~) را درج کنید. برنامه زیر نحوه فراخوانی سازنده و مخرب را نشان می‌دهد :

```
1  #include <iostream>  
2  #include <string>  
3  using namespace std;  
4  
5  class Person  
6  {  
7      public:  
8          Person()  
9          {  
10             cout << "Constructor was called." << endl;  
11         }  
12  
13         ~Person()  
14         {  
15             cout << "Destructor was called." << endl;  
16         }  
17     };  
18  
19     int main()  
20     {  
21         Person p1;  
22  
23     }
```

در کلاس Person یک سازنده و یک مخرب تعریف شده است. سپس در داخل متد main() یک نمونه از کلاس ایجاد کرده‌ایم. وقتی یک نمونه از کلاس ایجاد می‌کنیم سازنده فراخوانی شده و پیغام مناسب نمایش داده می‌شود. وقتی از متد main() خارج می‌شویم نمونه ایجاد شده نابود و مخرب فراخوانی می‌شود.

سطح دسترسی

سطح دسترسی مشخص می‌کند که متدها یا فیلدهای یک کلاس در چه جای برنامه قابل دسترسی هستند. در این درس می‌خواهیم به سطح دسترسی `private` و `public` نگاهی بیندازیم. سطح دسترسی `public` زمانی مورد استفاده قرار می‌گیرد که شما بخواهید به یک متد یا فیلد در خارج از کلاس دسترسی یابید. به عنوان مثال به کد زیر توجه کنید :

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 class Test
6 {
7     public:
8         int number1;
9 };
10
11 int main()
12 {
13     Test T1;
14
15     T1.number = 10;
16
17 }
```

در خط 7 کد بالا سطح دسترسی فیلدها را برابر `public` قرار داده‌ایم. در نتیجه می‌توانیم در خارج از کلاس بعد از ایجاد نمونه از کلاس به فیلد دسترسی داشته باشیم و آن را مقداردهی کنیم (خط 17). سطح دسترسی پیشفرض اعضا و متدهای یک کلاس در C++ به صورت `private` می‌باشد. در نتیجه اگر سطح دسترسی برای اعضای کلاس تعریف نکنیم نمی‌توانیم در خارج از کلاس به آنها دسترسی داشته باشیم :

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 class Test
6 {
7     int number1;
8 };
9
10 int main()
11 {
12     Test T1;
13
14     T1.number = 10;    //Error
15
16 }
```

```
}
```

اعضای داده‌ای private فقط در داخل کلاس Test قابل دسترسی هستند. سطح دسترسی دیگری هم در C++ وجود دارد که بعد از مبحث وراثت در درسهای آینده در مورد آنها توضیح خواهیم داد.

کپسوله کردن (Encapsulation)

کپسوله کردن (تلفیق داده‌ها با یکدیگر) یا مخفی کردن اطلاعات فرایندی است که طی آن اطلاعات حساس یک موضوع از دید کاربر مخفی می‌شود و فقط اطلاعاتی که لازم باشد برای او نشان داده می‌شود. وقتی که یک کلاس تعریف می‌کنیم معمولاً تعدادی اعضای داده‌ای (فیلد) برای ذخیره مقادیر مربوط به شیء نیز تعریف می‌کنیم. برخی از این اعضای داده‌ای توسط خود کلاس برای عملکرد متدها و برخی دیگر از آنها به عنوان یک متغیر موقت به کار می‌روند. به این اعضای داده‌ای، اعضای مفید نیز می‌گویند چون فقط در عملکرد متدها تأثیر دارند و مانند یک داده قابل رویت کلاس نیستند. لازم نیست که کاربر به تمام اعضای داده‌ای یا متدهای کلاس دسترسی داشته باشد. اینکه فیلدها را طوری تعریف کنیم که در خارج از کلاس قابل دسترسی باشند بسیار خطرناک است چون ممکن است کاربر رفتار و نتیجه یک متد را تغییر دهد. به برنامه ساده زیر توجه کنید :

```
1  #include <iostream>
2  using namespace std;
3
4  class Test
5  {
6      public:
7
8          int five = 5;
9
10         int AddFive(int number)
11         {
12             number += five;
13             return number;
14         }
15     };
16
17     int main()
18     {
19         Test x ;
20
21         x.five = 10;
22         cout << x.AddFive(100);
23     }
24 }
```

متد داخل کلاس Test به نام AddFive() دارای هدف ساده‌ای است و آن اضافه کردن مقدار 5 به هر عدد می‌باشد. در داخل متد main() یک نمونه از کلاس Test ایجاد کرده‌ایم و مقدار فیلد آن را از 5 به 10 تغییر می‌دهیم (در اصل نباید تغییر کند چون ما از برنامه خواسته‌ایم هر عدد را با 5 جمع کند ولی کاربر به راحتی آن را به 10 تغییر می‌دهد (خط 21)). همچنین متد AddFive() را فراخوانی و مقدار 100 را به آن ارسال می‌کنیم. مشاهده می‌کنید که قابلیت متد AddFive() به خوبی تغییر می‌کند و شما نتیجه متفاوتی مشاهده می‌کنید. اینجاست که اهمیت کپسوله سازی مشخص می‌شود. اینکه ما در درسهای قبلی فیلدها را به صورت public تعریف کردیم و به کاربر اجازه دادیم که در خارج از کلاس به آنها دسترسی داشته باشد کار اشتباهی بود. فیلدها باید همیشه به صورت private تعریف شوند. پس برنامه بالا را به صورت زیر اصلاح می‌کنیم :

```

1  #include <iostream>
2  using namespace std;
3
4  class Test
5  {
6      private:
7          int five = 5;
8
9      public:
10         int AddFive(int number)
11         {
12             number += five;
13             return number;
14         }
15     };
16
17     int main()
18     {
19         Test x ;
20
21         cout << x.AddFive(100);
22     }
23 
```

خواص (Property)

Property (خصوصیت) استاندارد در C++، برای دسترسی به اعضای داده‌ای (فیلدها) با سطح دسترسی private در داخل یک کلاس می‌باشد. همانطور که در درس قبل اشاره شد، تعریف فیلدها در داخل کلاس به صورت public اشتباه است، چون کاربران می‌توانند با ایجاد یک شیء از کلاس به آنها دسترسی داشته باشند و هر مقداری که دوست دارند به آنها اختصاص دهند. برای رفع این مشکل مفهوم property ارائه شد. هر property دارای دو بخش می‌باشد، یک بخش جهت مقدار دهی (بلوک

set) و یک بخش برای دسترسی به مقدار (بلوک get) یک داده private می باشد. Property ها باید به صورت public تعریف شوند تا در کلاسهای دیگر نیز قابل دسترسی می باشند. در زیر نحوه تعریف و استفاده از property آمده است :

```
1  #include<string>
2  #include<iostream>
3  using namespace std;
4
5  class Person
6  {
7      private:
8          string Name;
9          int Age;
10         double Height;
11
12     public:
13         string getName()
14         {
15             return Name;
16         }
17         void setName(string value)
18         {
19             Name = value;
20         }
21
22         int getAge()
23         {
24             return Age;
25         }
26         void setAge(int value)
27         {
28             Age = value;
29         }
30
31         double getHeight()
32         {
33             return Height;
34         }
35         void setHeight(double value)
36         {
37             Height = value;
38         }
39
40     public:
41         Person(string name, int age, double height)
42         {
43             this->Name = name;
44             this->Age = age;
45             this->Height = height;
46         }
47 };
48
49
50 int main()
51 {
52     Person person1("Jack", 21, 160);
```



```

53     Person person2("Mike", 23, 158);
54
55     cout << "Name: "    << person1.getName()    << endl;
56     cout << "Age: "     << person1.getAge()      << " years old" << endl;
57     cout << "Height: "  << person1.getHeight()   << "cm" << endl;
58
59     cout << endl; //Seperator
60
61     cout << "Name: "    << person2.getName()    << endl;
62     cout << "Age: "     << person2.getAge()      << " years old" << endl;
63     cout << "Height: "  << person2.getHeight()   << "cm" << endl;
64
65     person1.setName("Frank");
66     person1.setAge(19);
67     person1.setHeight(162);
68
69     person2.setName("Ronald");
70     person2.setAge(25);
71     person2.setHeight(174);
72
73     cout << endl; //Seperator
74
75     cout << "Name: "    << person1.getName()    << endl;
76     cout << "Age: "     << person1.getAge()      << " years old" << endl;
77     cout << "Height: "  << person1.getHeight()   << "cm" << endl;
78
79     cout << endl;
80
81     cout << "Name: "    << person2.getName()    << endl;
82     cout << "Age: "     << person2.getAge()      << " years old" << endl;
83     cout << "Height: "  << person2.getHeight()   << "cm" << endl;
84
85 }

```

```

Name: Jack
Age : 21 years old
Height : 160cm

```

```

Name : Mike
Age : 23 years old
Height : 158cm

```

```

Name : Frank
Age : 19 years old
Height : 162cm

```

```

Name : Ronald
Age : 25 years old
Height : 174cm

```

در برنامه بالا نحوه استفاده از property آمده است. همانطور که مشاهده می‌کنید در این برنامه ما سه فیلد (خطوط 10-8) تعریف

کرده‌ایم (سه فیلد با سطح دسترسی private):

```
string Name;
```

```
int Age;  
double Height;
```

دسترسی به مقادیر این فیلدها فقط از طریق property های ارائه شده (خطوط 14-45) امکان پذیر است.

```
string getName()  
{  
    return Name;  
}  
void setName(string value)  
{  
    Name = value;  
}  
  
int getAge()  
{  
    return Age;  
}  
void setAge(int value)  
{  
    Age = value;  
}  
  
double getHeight()  
{  
    return Height;  
}  
void setHeight(double value)  
{  
    Height = value;  
}
```

وقتی یک خاصیت ایجاد می‌کنیم، باید سطح دسترسی آن را public تعریف کرده و نوع داده‌ای را که بر می‌گرداند یا قبول می‌کند را، مشخص کنیم. به این نکته توجه کنید که نام property ها همانند نام فیلدهای مربوطه می‌باشد با این تفاوت که حرف اول آنها بزرگ نوشته می‌شود (خطوط 14، 25 و 36).

البته یادآور می‌شویم که شباهت نام property ها و فیلدها اجبار نیست و یک قرارداد در C++ می‌باشد. برای تعریف Property ها دو متد تعریف می‌کنیم. یک متد، که با کلمه set شروع می‌شود و برای مقدار دهی به فیلدها (اعضای داده‌ای) به کار می‌رود. و متد دیگر که با کلمه get شروع می‌شود و به شما اجازه می‌دهد که یک مقدار را از فیلدها (اعضای داده‌ای) استخراج کنید.

به کلمه value در داخل متد set توجه کنید. Value همان مقداری است که از طریق property به فیلد اختصاص می‌دهیم. برای اختصاص یک مقدار به یک فیلد از طریق property کافیست که از متدی که با کلمه set شروع شده است استفاده کنیم

(کاری که در خطوط 72-78 انجام داده‌ایم). مثلاً برای اختصاص مقدار به سه فیلد age, name و height از طریق property باید به صورت زیر عمل کنید :

```
person1.setName("Frank");
person1.setAge(19);
person1.setHeight(162);
```

دستورات بالا متد set مربوط به هر property را فراخوانی کرده و مقادیری به هر یک از فیلدها اختصاص می‌دهد. برای فراخوانی متدی که با حرف get شروع می‌شود کافیهست که نام شیء و سپس علامت نقطه و در آخر نام متد دلخواه را بنویسیم. با این کار به برنامه می‌فهمانیم که ما نیاز به مقدار فیلد داریم.

```
cout << "Name: " << person1.getName() << endl;
cout << "Age: " << person1.getAge() << " years old" << endl;

cout << "Height: " << person1.getHeight() << "cm" << endl;
```

به این نکته توجه کنید که در داخل متدهایی که با کلمه get شروع می‌شوند هم می‌توان تغییراتی بر روی فیلدها اعمال کرد. مثلاً فرض کنید که یک فیلد دارید که مقادیر پولی را در خود ذخیره می‌کند. شما می‌توانید در متد get نحوه نمایش مقدار موجود در این فیلد را مشخص کنید. مثلاً خروجی به صورت سه رقم سه رقم نمایش داده شود. استفاده از property ها کد نویسی را انعطاف پذیر می‌کند مخصوصاً اگر بخواهید یک اعتبارسنجی برای اختصاص یک مقدار به فیلدها یا استخراج یک مقدار از آنها ایجاد کنید. پس می‌توان گفت که :

کاربرد اصلی property ها، اعتبار سنجی مقادیری است که کاربر می‌خواهد به فیلدها اختصاص دهد.

مثلاً شما می‌توانید یک محدودیت ایجاد کنید که فقط اعداد مثبت به فیلد age (سن) اختصاص داده شود. همانطور که در کد ابتدای درس مشاهده می‌کنید ما نوع فیلد age را int قرار داده‌ایم. یعنی کاربر می‌تواند هر رقمی بین اعداد -2147483648 تا 2147483647 را به این فیلد اختصاص دهد. ولی چون غیر معقولانه است و سن (age) باید یک عدد مثبت و از لحاظ عقلی عددی از 1 تا 100 باشد می‌توانیم کاربر را با استفاده از بخش set مجبور کنیم که رقمی بین این دو عدد را به age اختصاص دهد. می‌توانید با تغییر بخش set خاصیت Age این کار را انجام دهید :

```
int getAge()
{
    return age;
}
void setAge(int value)
{
    if (value > 0 && value <= 100)
```

```
    age = value;
else
    age = 0;
}
```

حال اگر کاربر بخواهد یک مقدار منفی به فیلد age اختصاص دهد مقدار age صفر خواهد شد. همچنین می توان یک property فقط خواندنی (read-only) ایجاد کرد. این property فاقد بخش set است. به عنوان مثال می توان یک خاصیت Name فقط خواندنی مانند زیر ایجاد کرد :

```
string getName()
{
    return name;
}
```

در این مورد اگر بخواهید یک مقدار جدید به فیلد name اختصاص دهید با خطا مواجه می شوید. نکته دیگری که باید به آن توجه کنید این است که شما می توانید برای بخش set یا get سطح دسترسی ایجاد کنید. به تکه کد زیر توجه کنید :

```
public:
    string getName()
    {
        return Name;
    }

private:
    void setName(string value)
    {
        Name = value;
    }
```

خاصیت Name فقط در خارج از کلاس قابل خواندن است اما متدها فقط داخل کلاس Person می توانند مقادیر جدید بگیرند. یک property می تواند دارای دو فیلد باشد. به کد زیر توجه کنید :

```
string getFullName()
{
    return firstName + " " + lastName;
}
```

همانطور که در مثال بالا مشاهده می کنید یک property فقط خواندنی تعریف کرده ایم که مقدار برگشتی آن ترکیبی از دو فیلد firstName و lastName است که به وسیله فاصله از هم جدا شده اند.

فضای نام (Namespace)

از فضاهای نام به منظور جلوگیری از تداخل نام کلاس‌ها و توابع و گروه بندی منطقی آن‌ها مورد استفاده قرار می‌گیرد. در مثال زیر دو کلاس هم نام را در یک scope یا حوزه دید قرار دادیم که باعث بروز خطا می‌شود :

```
class Sample {};  
class Sample {}; // Error
```

اولین راه برای حل این مشکل، تغییر دادن نام یکی از آن‌هاست و راه حل دیگر، قرار دادن یکی یا هر دو آن‌ها در فضای نام‌های متفاوت است تا از تداخل نام آن‌ها جلوگیری شود.

```
namespace MyNamespace1  
{  
    class Sample {};  
}  
  
namespace MyNamespace2  
{  
    class Sample {};  
}
```

دسترسی به اعضای یک فضای نام

برای دسترسی به اعضای یک فضای نام در خارج از آن، باید نام کامل عضو مورد را بنویسید. منظور از نام کامل این است که فضای نام را قبل از نام کلاس مورد نظر بنویسید. برای این منظور، ابتدا اسم فضای نام را نوشته سپس (::) را قرار داده و در نهایت نام کلاس مورد نظر را بنویسید. به این ترتیب مشخص می‌شود هر کلاس متعلق به کدام فضای نام است.

```
int main()  
{  
    MyNamespace1::Sample s;  
    MyNamespace2::Sample s;  
}
```

فضای نام‌های تو در تو

شما می‌توانید به هر تعداد که بخواهید فضای نام‌های تو در تو (nested) داشته باشید.

```
namespace MyNamespace1
{
    namespace MyNamespace2 { class Sample {}; }
}
```

برای دسترسی به یک عضو از فضای نام داخلی، باید اسم فضای نام های بیرونی را قبل از آن بنویسید :

```
MyNamespace1::MyNamespace2::Sample s;
```

وارد کردن فضای نام

شما هر زمان که بخواهید از اعضای یک فضای نام استفاده کنید، مجبورید اسم فضای نام را قبل از آن ها بنویسید. برای اینکه مجبور نباشید هر بار این کار را انجام دهید، می توانید فضای نام مورد نظر خود را using کنید. برای این منظور، ابتدا کلمات کلیدی using namespace را نوشته و بعد از آن فضای نام مورد نظر خود را بنویسید. از کلمات کلیدی namespace using ، هم می توانید به صورت سراسری (global) و هم به صورت محلی (local) استفاده کنید. به این ترتیب، برای دسترسی به اعضای یک فضای نام، نوشتن نام کامل عضو لازم نیست.

```
using namespace MyNamespace1; // global namespace import

int main()
{
    using namespace MyNamespace1; // local namespace import
}
```

از فضای نامی که به صورت سراسری using شده باشد در کل فایل سورس و از فضای نامی که به صورت محلی using شده باشد فقط در بلوکی که داخل آن تعریف شده می توان استفاده کرد.

وارد کردن عضو یک فضای نام

زمانی که یک فضای نام را using می کنید، می توانید به تمام اعضای آن دسترسی داشته باشید. اگر نمی خواهید از تمام اعضای یک فضای نام استفاده کنید و فقط به یک عضو آن نیاز دارید، با استفاده از کلمه کلیدی using ، فقط نام کامل آن عضو را بنویسید. مثال :

```
using MyNamespace1::Sample; // import a single namespace member
```

منظور نام کامل، نوشتن فضای نام قبل از نام کلاس مورد نظر است.

نام مستعار یک فضای نام

روش دیگر برای خلاصه نویسی نام کامل عضو یک فضای نام، استفاده از نام مستعار است. برای این کار ابتدا کلمه کلیدی namespace و سپس نام مستعاری که می‌خواهید را بنویسید. حال فضای نام‌های مورد نظر خود را به این نام مستعار انتساب دهید :

```
namespace myAlias = MyNamespace1::Sample; // namespace alias
```

اکنون می‌توانید از این نام مستعار هر کجا که بخواهید به جای نام کامل استفاده کنید :

```
myAlias::Sample s;
```

نام مستعار برای یک Type

شما همچنین می‌توانید از نام مستعار برای تعریف Type ها نیز استفاده کنید. برای این منظور باید از کلمه کلیدی typedef به صورت زیر استفاده کنید :

```
typedef my::name::MyClass MyType;
```

در اینجا ما به جای my::name::MyClass از نام مستعار MyType استفاده کردیم. حال هر کجا که بخواهیم می‌توانیم از MyType یک نمونه تعریف کنیم :

```
MyType t;
```

از Typedef نه تنها برای Type های از قبل تعریف شده، بلکه برای تعریف یک Type جدید مانند struct ، class ، union یا enum نیز می‌توان استفاده کرد.

```
typedef struct { int len; } Length;
Length a, b, c;
```

ساخت نام مستعار با استفاده از Typedef خیلی توصیه نمی شود و باعث ایجاد ابهام در کدها می شود. گرچه اگر از نام های مستعار به درستی استفاده شود، می تواند از طولانی شدن و تداخل نام ها در کد جلوگیری کند.

تفاوت Include و using

Include کردن یک فایل header به این معنی است که می خواهیم قبل از عملیات کامپایل، محتویات این فایل در مکانی که آن را include کردیم کپی شود. در حالی که using فقط نام ها را از یک scope به scope فعلی می آورد و تاثیری بر کامپایلر ندارد. به همین دلیل است که وقتی ما برای مثال iostream را include می کنیم، برای استفاده از فضای نام std که در iostream قرار دارد باید آن را using کنیم :

```
// Include input/output prototypes
#include <iostream>

// Import standard library namespace to global scope
using namespace std;
```

نکته: برای include کردن فایل های header از قبل تعریف شده، از <#include> و برای فایل های تعریف شده توسط برنامه نویس، از "#include" استفاده می شود.

وراثت

وراثت به یک کلاس اجازه می دهد که خصوصیات یا متدهایی را از کلاس دیگر به ارث برد. وراثت مانند رابطه پدر و پسر می ماند به طوریکه فرزند خصوصیات از قبیل قیافه و رفتار را از پدر خود به ارث برده باشد:

- کلاس پایه یا کلاس والد کلاسی است که بقیه کلاسها از آن ارث می برند.
- کلاس مشتق یا کلاس فرزند کلاسی است که از کلاس پایه ارث بری می کند.

همه متد و خصوصیات کلاس پایه می توانند در کلاس مشتق مورد استفاده قرار بگیرند به استثنای اعضا و متدهای با سطح دسترسی private. مفهوم اصلی وراثت در مثال زیر نشان داده شده است :

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 class Parent
6 {
```



```

7     private:
8         string message;
9
10    public:
11        void ShowMessage()
12        {
13            cout << message << endl;
14        }
15
16        Parent()
17        {
18        }
19
20        Parent(string message)
21        {
22            this->message = message;
23        }
24    };
25
26    class Child : public Parent
27    {
28    public:
29        Child(string message) : Parent(message)
30        {
31        }
32    };
33
34    int main()
35    {
36        Parent myParent("Message from parent.");
37        Child myChild("Message from child.");
38
39        myParent.ShowMessage();
40
41        myChild.ShowMessage();
42
43        //myChild.message; ERROR: can't access private members of base class
44
45    }

```

Message from parent.

Message from child.

در این مثال دو کلاس با نامهای Parent و Child تعریف شده است. در این مثال یک فیلد را با سطح دسترسی private (خط 8) و سپس یک متد با سطح دسترسی public (خط 11-14) برای نمایش پیام در کلاس Parent تعریف کرده‌ایم. یک سازنده در کلاس Parent تعریف شده است که یک آرگومان از نوع رشته قبول می‌کند و یک پیغام نمایش می‌دهد (خطوط 20-23). حال به کلاس Child توجه کنید (خطوط 26-33). این کلاس تمام متدها و خاصیت‌های کلاس Parent را به ارث برده است. در خط 44

همانطور که نشان داده شده است اگر بخواهید از طریق کلاس فرزند به عضوی با سطح دسترسی private دست یابید با پیغام خطا مواجه می شوید. نحوه ارث بری یک کلاس به صورت زیر است :

```
class DerivedClass : public BaseClass
{
    //some code
};
```

براحتی می توان با قرار دادن یک کالن (:.) بعد از نام کلاس و سپس کلمه public و نوشتن نام کلاسی که از آن ارث بری می شود (کلاس پایه) این کار را انجام داد (خط 26). در داخل کلاس Child هم یک سازنده ساده وجود دارد که یک آرگومان رشته ای قبول می کند. وقتی از وراثت در کلاسها استفاده می کنیم، هم سازنده کلاس مشتق و هم سازنده پیشفرض کلاس پایه هر دو اجرا می شوند. سازنده پیشفرض یک سازنده بدون پارامتر است. اگر برای یک کلاس سازنده ای تعریف نکنیم کامپایلر به صورت خودکار یک سازنده برای آن ایجاد می کند. اگر هنگام صدا زدن سازنده کلاس مشتق بخواهیم سازنده کلاس پایه را صدا بزنیم باید بعد از پرانتز سازنده کلاس مشتق نام سازنده کلاس پایه را را بنویسیم و علامت پرانتز هم بگذاریم :

```
ChildConstructor(type parameter) : ParentConstructor(parameter)
{
}
```

در مثال بالا به وسیله تأمین مقدار پارامتر message سازنده کلاس مشتق و ارسال آن به داخل پرانتز سازنده کلاس Parent یا والد (خط 29)، سازنده معادل آن در کلاس پایه فراخوانی شده (خط 20) و مقدار message را به آن ارسال می کند. سازنده کلاس Parent هم این مقدار (مقدار message) را در یک عضو داده ای (private) قرار می دهد (خط 8). می توانید کدهایی را به داخل بدنه سازنده Child اضافه کنید تا بعد از سازنده Parent اجرا شوند. اگر از هیچ آرگومانی استفاده نشود سازنده پیشفرض کلاس پایه فراخوانی می شود :

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  class Parent
6  {
7      private:
8          string message;
9
10     public:
11         void ShowMessage()
12         {
13             cout << message << endl;
14         }
15
16         Parent()
```

```

17     {
18         cout << "This is default parent constructor!";
19     }
20
21     Parent(string message)
22     {
23         this->message = message;
24     }
25 };
26
27 class Child : public Parent
28 {
29     public:
30     Child()
31     {
32
33     }
34     Child(string message) : Parent(message)
35     {
36
37     }
38 };
39
40 int main()
41 {
42     Child FirstChild;
43     Child SecondChild("This is message from child!");
44
45     FirstChild.ShowMessage();
46     SecondChild.ShowMessage();
47 }

```

```
This is default parent constructor!
```

```
This is message from child!
```

که در کد بالا مشاهده می‌کنید که دو شیء در خطوط 42 و 43 از کلاس Child ایجاد کرده‌ایم. همانطور که در خروجی مشاهده می‌کنید چون در خط 42 هیچ پارامتری به شیء اول نداده‌ایم در نتیجه سازنده پیشفرض کلاس Parent (خطوط 19-16) و در خط 43 چون به شیء ایجاد شده پارامتر داده‌ایم در نتیجه سازنده خطوط 21-24 فراخوانی می‌شود. C++ از وراثت چندگانه هم پشتیبانی می‌کند. کافیست که بعد از علامت: نام کلاس‌هایی را که می‌خواهید یک کلاس از آنها ارث بری کند را ذکر کرده و با کاما از هم جدا کنید :

```

class A
{
};

class B
{
};

```

```
class C : public A, public B
{
};
```

سطح دسترسی Protect

سطح دسترسی protect اجازه می‌دهد که اعضای کلاس، فقط در کلاسهای مشتق شده از کلاس پایه قابل دسترسی باشند. بدیهی است که خود کلاس پایه هم می‌تواند به این اعضا دسترسی داشته باشد. کلاس‌هایی که از کلاس پایه ارث بری نکرده‌اند نمی‌توانند به اعضای با سطح دسترسی protect یابند. در مورد سطوح دسترسی public و private قبلاً توضیح دادیم. در جدول زیر نحوه دسترسی به سه سطح ذکر شده نشان داده شده است :

قابل دسترسی در	public	private	protected
داخل کلاس	true	true	true
خارج از کلاس	true	false	false
کلاس مشتق	true	false	true

مشاهده می‌کنید که public بیشترین سطح دسترسی را داراست. صرف نظر از مکان، اعضای public در هر جا فراخوانی می‌شوند و قابل دسترسی هستند. اعضای private فقط در داخل کلاسی که به آن تعلق دارند قابل دسترسی هستند. کد زیر رفتار اعضای دارای این سه سطح دسترسی را نشان می‌دهد :

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Parent
6  {
7      protected: int protectedMember = 10;
8      private : int privateMember = 10;
9      public : int publicMember = 10;
10 };
11
12 class Child : public Parent
13 {
14     Child()
15     {
16         protectedMember = 100;
17         privateMember = 100;
```

```

18     publicMember = 100;
19 }
20 };
21
22 int main()
23 {
24     Parent myParent;
25
26     myParent.protectedMember = 100;
27     myParent.privateMember = 100;
28     myParent.publicMember = 100;
29
30 }

```

کدهایی که با خط قرمز نشان داده شده‌اند نشان دهنده وجود خطا هستند، چون فیلدهای `private` و `protected` در خارج از کلاسی که به آن تعلق دارند یعنی کلاس `Parent`، غیر قابل دسترسی هستند. همانطور که در خط 17 مشاهده می‌کنید کلاس `Child` سعی می‌کند که به عضو `private` کلاس `Parent` دسترسی یابد. از آنجاییکه اعضای `private` در خارج از کلاس قابل دسترسی نیستند، حتی کلاس مشتق در خط 17 نیز ایجاد خطا می‌کند. اگر شما به خط 14 توجه کنید کلاس `Child` می‌تواند به عضو `protected` کلاس `Parent` دسترسی یابد چون کلاس `Child` از کلاس `Parent` مشتق شده است. حال به خط 26 و 27 نگاهی بیندازید. می‌بینید که برنامه پیغام خطا می‌دهد، چون اعضای `private` و `protected` در خارج از کلاسی که به آن تعلق دارند غیر قابل دسترسی هستند.

اعضای استاتیک

اگر بخواهیم عضو داده‌ای (فیلد) یا خاصیتی ایجاد کنیم که در همه نمونه‌های کلاس قابل دسترسی باشد از کلمه کلیدی `static` استفاده می‌کنیم. کلمه کلیدی `static` برای اعضای داده‌ای و خاصیت‌هایی به کار می‌رود که می‌خواهند در همه نمونه‌های کلاس تقسیم شوند. وقتی که یک متد یا خاصیت به صورت `static` تعریف شود، می‌توانید آنها را بدون ساختن نمونه‌ای از شی، فراخوانی کنید. به مثالی در مورد متدها و خاصیت‌های `static` توجه کنید :

```

1 #include<string>
2 #include<iostream>
3 using namespace std;
4
5 class SampleClass
6 {
7     public:
8         static string StaticMessage;
9
10        string NormalMessage;

```

```
11
12     static void ShowStaticMessage()
13     {
14         cout << StaticMessage;
15     }
16
17     void ShowNormalMessage()
18     {
19         cout << NormalMessage;
20     }
21
22     void ShowStaticFromInstance()
23     {
24         cout << StaticMessage;
25     }
26 };
27
28 string SampleClass::StaticMessage = "This is the static message!\n";
29
30 int main()
31 {
32     SampleClass sample1;
33     SampleClass sample2;
34
35     SampleClass::ShowStaticMessage();
36
37     cout << endl;
38
39     sample1.NormalMessage = "Message from sample1!\n";
40     sample1.ShowNormalMessage();
41     sample1.ShowStaticFromInstance();
42
43     cout << endl;
44
45     sample2.NormalMessage = "Message from sample2!\n";
46     sample2.ShowNormalMessage();
47     sample2.ShowStaticFromInstance();
48 }
```

```
This is the static message!
```

```
Message from sample1!
```

```
This is the static message!
```

```
Message from sample2!
```

```
This is the static message!
```

در مثال بالا یک خاصیت استاتیک به نام StaticMessage (خط 8) و یک متد استاتیک به نام ShowStaticMessage() (خطوط 12-15) تعریف کرده ایم. وقتی یک عضو داده ای static را در داخل کلاس تعریف می کنید (خط 8)، حافظه به آن اختصاص نمی یابد. در عوض باید در خارج از کلاس، آن را دوباره تعریف و مقداردهی کنید (خط 28). همانطور که در خط 28 مشاهده می کنید، باید قبل از نام متغیر استاتیک، نام کلاس را به همراه علامت دو نقطه بنویسید. مقدار خاصیت StaticMessage در همه نمونه های کلاس SampleClass قابل دسترسی است. متد استاتیک را نمی توان به وسیله نمونه ایجاد شده از کلاس

SampleClass فراخوانی کرد. برای فراخوانی یک متد یا خاصیت استاتیک، به سادگی می‌توان نام کلاس و بعد از آن علامت دو نقطه (::) و در آخر نام متد یا خاصیت را نوشت. این موضوع را می‌توان در خطوط 28 و 35 مشاهده کرد. مشاهده می‌کنید که لازم نیست هیچ نمونه‌ای از کلاس ایجاد شود.

همانطور که مشاهده می‌کنید یک پیغام را به StaticMessage اختصاص داده‌ایم و با فراخوانی یک متد استاتیک (ShowStaticMessage()) مقدار آن را نمایش می‌دهیم. در مرحله بعد خاصیت NormalMessage را به وسیله دو نمونه ایجاد شده فراخوانی می‌کنیم و یک متد را هم برای نشان دادن مقدار آن فراخوانی می‌کنیم. همانطور که مشاهده می‌کنید نمونه‌های ایجاد شده دارای مقدار NormalMessage مخصوص خودشان هستند چون خاصیت NormalMessage غیر استاتیک است.

سپس ShowStaticFromInstance() فراخوانی می‌کنیم که متدی برای نشان دادن مقدار StaticMessage می‌باشد. متدهای غیر استاتیک می‌توانند از فیلدها و خاصیت‌های استاتیک استفاده کنند ولی عکس این قضیه امکان پذیر نیست. به عنوان مثال اگر شما یک متد static داشته باشید نمی‌توانید از هر خاصیت، متد، یا فیلدی که static نیست، استفاده کنید.

```
private:
    int number = 10;

public:
    static void ShowNumber()
    {
        cout << number; //Error: cannot use non-static membe
    }
```

اصرار بر این کار باعث بروز خطا می‌شود.

کلاس استاتیک

یک کلاس static کلاسی است که همه اعضای آن static باشند. یکی از روش‌های معمول استفاده از کلاس static، ایجاد یک کتابخانه ریاضی که شامل تعدادی از توابع و مقادیر است، می‌باشد. کلاس Math یکی از این کلاس‌ها است که در درس‌های آینده در مورد آن توضیح می‌دهیم. موارد مهمی که در مورد این کلاس‌ها باید بدانید عبارتند از :

- در تعریف کلاس static باید از کلمه static استفاده شود.
- همه اعضای این کلاس‌ها باید static باشند.
- از کلاس static نمی‌توان نمونه (Object) ایجاد کرد.
- کلاس‌های استاتیک به صورت ضمنی sealed هستند، در نتیجه نمی‌توان از آنها ارث بری کرد.

- این کلاس ها نمی توانند سازنده یا constructor داشته باشند، ولی با این حال می توان از static constructor ها برای مقدار دهی به عناصر static کلاس، استفاده کرد. توجه کنید که static constructor ها پارامتر و سطح دسترسی ندارند.
- برای دسترسی به اعضای static ای که در کلاس هستند، کافی است ابتدا نام کلاس را نوشته، سپس عملگر (::) و در آخر نام عضو استاتیک را بنویسید.

به مثال زیر توجه کنید :

```

1  #include<string>
2  #include<iostream>
3  using namespace std;
4
5  static class Person
6  {
7      public:
8          static string ShowMessage(string message)
9          {
10             return message;
11          }
12 };
13
14 int main()
15 {
16     cout << Person::ShowMessage("Hello World!");
17 }
```

Hello World!

در کد بالا یک کلاس استاتیک به نام Person در خطوط 5-12 تعریف شده است که دارای یک متد استاتیک با نام ShowMessage() می باشد (خطوط 8-10). همانطور که در خط 16 مشاهده می کنید برای دسترسی به این متد استاتیک، بدون اینکه از کلاس شیء ایجاد کنیم، ابتدا نام کلاس سپس علامت :: و در آخر نام متد را می نویسیم.

ترکیب (Composition)

در دنیای واقعی یک شیء از بخش های مختلفی تشکیل شده است. برای مثال یک ماشین از بخش های مختلفی نظیر موتور، چرخ، در و ... تشکیل شده است. به این نوع از رابطه بین بخش های مختلف یک شیء composition گفته می شود. همچنین آن را رابطه has-a نیز می نامند. Composition به ما اجازه می دهد تا برای هر وظیفه یک کلاس ایجاد کنیم. مزایای composition عبارتند از:

- هر کلاس می‌تواند ساده و صریح باشد.
- یک کلاس می‌تواند بر روی وظایف مشخصی تمرکز کند.
- نوشتن، خطایابی، درک و استفاده از کلاس‌ها ساده‌تر می‌شود.
- باعث کاهش پیچیدگی شیء می‌شود.
- یک کلاس پیچیده می‌تواند در صورت لزوم کنترل برخی اعمال را به کلاس‌های کوچک‌تر منتقل کند.

برنامه زیر composition را نشان می‌دهد:

```

1  #include<iostream>
2  #include<string>
3  using namespace std;
4
5  class Engine
6  {
7      public:
8          int power;
9  };
10
11 class Car
12 {
13     public:
14         Engine engine;
15         string company;
16         string color;
17
18         void showDetails()
19         {
20             cout << "Compnay is: " << company << endl;
21             cout << "Color is: " << color << endl;
22             cout << "Engine horse power is: " << engine.power << endl;
23         }
24 };
25
26 int main()
27 {
28     Car car;
29     car.engine.power = 500;
30     car.company = "hyundai";
31     car.color = "black";
32     car.showDetails();
33 }
```

```

Compnay is: hyundai
Color is: black
Engine horse power is: 500
```

همانطور که در کد بالا مشاهده می‌کنید ما یک کلاس به نام Engine تعریف کرده‌ایم (خطوط 5-9) که دارای یک فیلد به نام Power (خط 8) می‌باشد. حال همین کلاس را به عنوان یک فیلد از کلاس Car در خط 14 معرفی نموده‌ایم. در نتیجه برای استفاده از

خاصیت Power باید به صورت تودرتو به آن دست یابیم (خط 28). نکته‌ای که در خط 28 وجود دارد این است که قبلاً ذکر کردیم که برای دسترسی به فیلدهای یک کلاس باید از علامت نقطه (.) استفاده کنیم. در نتیجه در این خط چون Power یک فیلد از کلاس Engine و Engine هم یک فیلد از کلاس Car است، به این صورت عمل کرده‌ایم.

متدهای مجازی

متدهای مجازی متدهایی از کلاس پایه هستند که می‌توان به وسیله یک متد از کلاس مشتق آنها را override کرده و به صورت دلخواه پیاده سازی نمود. به عنوان مثال شما متد A را در کلاس A دارید و کلاس B از کلاس A ارث بری می‌کند، در این صورت متد A در کلاس B در دسترس خواهد بود. اما متد A دقیق همان متدی است که از کلاس A به ارث برده شده است. حال اگر بخواهید که این متد رفتار متفاوتی از خود نشان دهد چکار می‌کنید؟ متد مجازی این مشکل را برطرف می‌کند. به تکه کد زیر توجه کنید :

```
1  #include <iostream>
2  using namespace std;
3
4  class Parent
5  {
6      public:
7          virtual void ShowMessage()
8          {
9              cout << "Message from Parent." << endl;
10         }
11     };
12
13     class Child : Parent
14     {
15         public:
16             void ShowMessage()
17             {
18                 cout << "Message from Child.";
19             }
20     };
21
22     int main()
23     {
24         Parent myParent;
25         Child myChild;
26
27         myParent.ShowMessage();
28         myChild.ShowMessage();
29
30     }
```

Message from Parent.

Message from Child.

متد مجازی با قرار دادن کلمه کلیدی virtual هنگام تعریف متد، تعریف می‌شود. (خط 7) این کلمه کلیدی نشان می‌دهد که متد می‌تواند override شود یا به عبارت دیگر می‌تواند به صورت دیگر پیاده سازی شود. کلاسی که از کلاس Parent ارث می‌برد شامل متدی است که متد مجازی کلاس پایه را override یا به صورت دیگری پیاده سازی می‌کند. با استفاده از نام کلاس والد و علامت :: (خط 18) می‌توانید متد مجازی را در داخل متد override شده فراخوانی کنید :

```
1  #include <iostream>
2  using namespace std;
3
4  class Parent
5  {
6      public:
7          virtual void ShowMessage()
8          {
9              cout << "Message from Parent." << endl;
10         }
11     };
12
13     class Child : Parent
14     {
15         public:
16             void ShowMessage()
17             {
18                 Parent::ShowMessage();
19                 cout << "Message from Child.";
20             }
21     };
22
23     int main()
24     {
25         Parent myParent;
26         Child myChild;
27
28         myParent.ShowMessage();
29         myChild.ShowMessage();
30
31     }
```

```
Message from Parent.
Message from Parent.

Message from Child.
```

کلاس تو در تو (Nested Class)

C++ به برنامه نویسان اجازه می دهد تا یک کلاس را داخل کلاسی دیگر تعریف کنند. به این کلاس ها، nested کلاس نیز گفته

می شود. زمانی که کلاس B داخل کلاس A تعریف می شود، کلاس B نمی تواند به اعضای کلاس A دسترسی داشته باشد ولی کلاس

A می تواند به اعضای کلاس B از طریق یک شیء از آن دسترسی داشته باشد. ویژگی های یک کلاس تو در تو عبارتند از:

- یک کلاس تو در تو داخل یک کلاس دیگر تعریف می شود
- محدوده یا میدان دیدن کلاس داخلی در scope کلاس بیرونی است.
- برای تعریف یک شیء از کلاس داخلی، باید نام کلاس بیرونی به همراه عملگر (::) پشت آن قرار بگیرد.

برنامه زیر نحوه استفاده از کلاس های تو در تو را نشان می دهد:

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 class Student
6 {
7     private:
8         int Age;
9
10    public:
11        class Name
12        {
13            private:
14                string name;
15
16            public:
17                string getName()
18                {
19                    return name;
20                }
21                void setName(string n)
22                {
23                    name = n;
24                }
25        };
26 };
27
28 int main()
29 {
30     Student::Name n;
31
32     n.setName("John");
33     cout << "Name is : " << n.getName() << endl;
34 }
```

Name is : John

تابع دوست (Friend Function)

یک متد دوست تابعی است که از اعضای یک کلاس نیست ولی می‌تواند به اعضای یک کلاس که دارای سطح دسترسی `private` و `protected` هستند دسترسی داشته باشد. برای اینکه یک متد خارجی را به عنوان دوست یک کلاس تعریف کنیم از کلمه کلیدی `friend` استفاده می‌کنیم. ساختار کلی تعریف یک متد دوست به صورت زیر می‌باشد:

```
class ClassName
{
    friend return-type function-name(params-list);
};
```

همانطور که مشاهده می‌کنید ما فقط `prototype` متد را داخل کلاس تعریف کردیم و پیاده سازی آن را در خارج از کلاس انجام می‌دهیم. منظور از `prototype` این است که فقط متد را اعلان کنیم یعنی نام تابع، نوع برگشتی و پارامترهای آن را مشخص کنیم و پیاده سازی آن را در جای دیگری انجام دهیم. ویژگی‌های یک متد دوست عبارتند از:

- تابع دوست یک متد عادی است که در خارج از یک کلاس تعریف می‌کنیم فقط با این تفاوت که می‌توانیم به اعضای یک کلاس که دارای سطح دسترسی `private` و `protected` هستند دسترسی داشته باشیم.
- نمی‌توانیم به توابع دوست با استفاده از عملگر `(.)` یا `->` دسترسی داشته باشیم.
- یک متد دوست نمی‌تواند به عنوان عضو یک کلاس در نظر گرفته شود
- یک متد می‌تواند در هر تعداد کلاسی که می‌خواهد به عنوان دوست تعریف شود
- از آنجایی که یک متد دوست یک متد عضو محسوب نمی‌شود، بنابراین ما نمی‌توانیم با استفاده از اشاره گر `this` به آن‌ها اشاره کنیم.
- کلمه کلیدی `friend` فقط برای اعلان متد مورد استفاده قرار می‌گیرد و در زمان تعریف متد از آن استفاده نمی‌کنیم.
- یک متد دوست می‌تواند به اعضای یک کلاس با استفاده از یک شیء از آن کلاس دسترسی داشته باشد.

برنامه زیر نحوه استفاده از یک متد دوست را نشان می‌دهد:

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

class Student
{
private:
    string Name;
```

```
public:
    void setName(string);
    friend void ShowDetails(Student);
};

void Student::setName(string name)
{
    Name = name;
}

void ShowDetails(Student s)
{
    cout << "Name of the student is: " << s.Name << endl;
}

int main()
{
    Student student1;
    student1.setName("Jack");
    ShowDetails(student1);
}
```

Jack

در برنامه بالا متد ShowDetails می‌تواند به متغیر name که یک متغیر private در کلاس Student است دسترسی داشته باشد.

Downcasting و Upcasting

به فرآیندی که می‌توانیم یک شیء (اشاره گر یا مرجع) از کلاس فرزند را به عنوان یک شیء از کلاس پدر در نظر بگیریم، Upcasting گفته می‌شود؛ یا به عبارت دیگر اگر اشاره گر کلاس پایه حاوی آدرس شیء ایجاد شده از کلاس مشتق باشد، به این فرایند Upcasting گفته می‌شود. برای انجام Upcasting نیاز نیست کار خاصی انجام دهید. اگر یک شیء از کلاس فرزند را داخل یک شیء از کلاس پدر بریزید این کار انجام می‌شود:

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  class Parent
6  {
7      public:
8          void sleep()
9          {
10             cout << "Father is sleeping" << endl;
11         }
```

```

12 };
13
14 class Child : public Parent
15 {
16     public:
17     void gotoSchool()
18     {
19         cout << "The child goes to school" << endl;
20     }
21 };
22
23 int main()
24 {
25     Parent *parent;
26     Child child;
27
28     parent = &child;
29
30     parent->sleep();
31 }

```

Father is sleeping.

کلاس Child فرزند کلاس Parent است، بنابراین همه اعضای داده‌ای و توابع عضو کلاس Parent را به ارث می‌برد. در نتیجه هر کاری که ما با یک شیء از کلاس Parent می‌توانیم انجام دهیم با یک شیء از کلاس Child نیز می‌توانیم انجام دهیم. پس تابعی که برای کار با یک اشاره گر (مرجع) از Parent ایجاد شده می‌تواند همان اعمال را بر روی یک شیء از Child نیز بدون هیچ مشکلی انجام دهد. خطوط 25-28 کد بالا را به دو صورت زیر هم می‌توان نوشت:

```
Parent *parent = new Child();
```

یا

```
Child child;
Parent *parent = &child;
```

به فرآیندی که ما یک اشاره گر (مرجع) از کلاس پدر را به یک اشاره گر (مرجع) از کلاس فرزند تبدیل می‌کنیم، Downcasting گفته می‌شود. برای انجام این کار باید حتماً از تبدیل صریح (cating) استفاده کرد. دلیل این محدودیت این است که در بیشتر موارد رابطه is-a متقارن نیست. برای مثال خرگوش یک حیوان است ولی هر حیوانی خرگوش نیست و این به معنی عدم تقارن در رابطه is-a است. اعضای داده‌ای و توابع عضو جدیدی که در کلاس فرزند وجود دارد، نمی‌تواند به کلاس پدر اضافه شود:

```
Parent *parent;
```

```
Child *child = (Child*)&parent;

child->gotoSchool();
```

ما نمی‌توانیم از طریق یک شیء از کلاس Parent به متد gotoSchool() دسترسی داشته باشیم.

```
Child *child = &parent; // actually this won't compile
                        // error: cannot convert from 'Parent *' to 'Child *'
```

کد بالا کامپایل نمی‌شود زیرا نمی‌تواند به صورت ضمنی یک اشاره گر از Parent را به یک اشاره گر از Child تبدیل کند. می‌توانیم از اشاره گر برای فراخوانی متد gotoSchool() به صورت زیر استفاده کنیم:

```
child -> gotoSchool();
```

از آنجایی که Parent یک Child نیست (یک Parent به متد gotoSchool() نیازی ندارد)، بنابراین عمل downcasting در خط بالا می‌تواند یک عملیات نا امن (unsafe) باشد. در C++ نوع خاصی از تبدیل صریح به نام dynamic_cast یا تبدیل صریح پویا پشتیبانی می‌شود که این تبدیل را انجام می‌دهد. بنا بر قوانین شیء گرایی، اشیاء یک کلاس فرزند را همیشه می‌توانیم در داخل متغیرهایی از کلاس پدر بریزیم. اما Downcasting بر خلاف این قوانین است. زمانی که یک اشاره گر به یک کلاس پدر ایجاد می‌کنیم دو امکان وجود دارد، یا این اشاره گر به یک شیء از همان کلاس پدر اشاره می‌کند یا اینکه با عمل upcasting به یک شیء از کلاس فرزند اشاره می‌کند.

dynamic cast عملگری است که می‌تواند یک نوع را به صورت امن، به یک نوع دیگر تبدیل کند. اگر عمل تبدیل به صورت امن ممکن بود، آدرس شیء تبدیل شده را بر می‌گرداند. در غیر این صورت null pointer یا همان مقدار 0 را بر می‌گرداند. به مثال زیر توجه کنید:

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  class Parent
6  {
7      public:
8          void sleep()
9          {
10             cout << "Father is sleeping" << endl;
11         }
12 };
13
14 class Child : public Parent
15 {
16     public:
```



```

17     void gotoSchool()
18     {
19         cout << "The child goes to school" << endl;
20     }
21 };
22
23 int main()
24 {
25     Parent *parent = new Parent;
26     Parent *child = new Child;
27
28     Child *p1 = (Child *)parent; // #1
29     Parent *p2 = (Child *)child; // #2
30 }

```

در دو خط 28 و 29 مثال بالا ما عمل تبدیل صریح را انجام دادیم. سوالی که در اینجا مطرح می‌شود این است که کدام تبدیل امن است؟ از بین دو حالتی که در کد بالا وجود دارد، فقط یک حالت است که تضمین می‌کند این تبدیل به صورت امن تبدیل شود. تبدیل صریح #1 امن نیست، زیرا آدرس یک شیء از کلاس پدر (Parent) را در داخل یک اشاره گر از کلاس فرزند (Child) قرار دادیم. با توجه به این حالت انتظار داریم که با استفاده از یک شیء از کلاس پدر بتوانیم متد gotoSchool() را فراخوانی کنیم، در صورتی که نمی‌توانیم از طریق یک شیء از کلاس پدر به اعضای کلاس فرزند دسترسی داشته باشیم. اما تبدیل صریح #2 امن است، زیرا می‌توانیم آدرس یک شیء از کلاس فرزند را در داخل یک اشاره گر از نوع کلاس پدر قرار دهیم. نحوه استفاده از عملگر dynamic_cast را به صورت کلی در زیر مشاهده می‌کنید:

```
Child *child = dynamic_cast<Child *>(parent)
```

عملگر dynamic_cast بررسی می‌کند که آیا می‌تواند اشاره گر parent را به صورت صریح و امن به نوع * Child تبدیل کند یا خیر؟ اگر بتواند، آدرس شیء و در غیر این صورت 0 را بر می‌گرداند. ما چگونه می‌توانیم از dynamic_cast استفاده کنیم؟

```

void f(Parent* parent)
{
    Child *child = dynamic_cast<Child*>(parent);

    if (child)
    {
        // we can safely use child
    }
}

```

در کد بالا اگر (child) از نوع Child باشد یا به صورت مستقیم و یا غیر مستقیم فرزند کلاس Child باشد، dynamic_cast اشاره گر parent را به یک اشاره گر از نوع Child تبدیل می‌کند. در غیر این صورت اگر نتواند این تبدیل را انجام دهد مقدار 0 که همان null pointer است را بر می‌گرداند. به عبارت دیگر می‌توانیم قبل از اینکه عملیاتی را بر روی اشاره گر parent انجام دهیم، با استفاده از dynamic_cast بررسی کنیم که آیا این تبدیل را می‌توان انجام داد یا خیر. به صورت کلی ما زمانی به

dynamic_cast نیاز داریم که بخواهیم عملیاتی را بر روی شیئی از کلاس فرزند انجام دهیم ولی فقط یک اشاره گر یا مرجع به کلاس پدر را در اختیار داریم. Upcasting و Downcasting در مبحث چند ریختی کاربرد دارند که در درس آینده توضیح می‌دهیم.

چند ریختی (polymorphism)

چند ریختی به کلاسهایی که در یک سلسله مراتب وراثتی مشابه هستند اجازه تغییر شکل و سازگاری مناسب می‌دهد و همچنین به برنامه نویس این امکان را می‌دهد که به جای ایجاد برنامه‌های خاص، برنامه‌های کلی و عمومی‌تری ایجاد کند. به عنوان مثال در دنیای واقعی همه حیوانات غذا می‌خورند، اما روش‌های غذا خوردن آنها متفاوت است. در یک برنامه برای مثال، یک کلاس به نام Animal ایجاد می‌کنید. بعد از ایجاد این کلاس می‌توانید آن را چند ریخت (تبدیل) به کلاس Bird کنید و متد Fly() را فراخوانی کنید. به مثالی درباره چند ریختی توجه کنید :

```
1  #include <iostream>
2  using namespace std;
3
4  class Animal
5  {
6      public:
7          virtual void Eat()
8          {
9              cout << "The animal ate!" << endl;
10         }
11     };
12
13     class Dog : public Animal
14     {
15         public:
16             void Eat()
17             {
18                 cout << "The dog ate!" << endl;
19             }
20     };
21
22     class Bird : public Animal
23     {
24         public:
25             void Eat()
26             {
27                 cout << "The bird ate!" << endl;
28             }
29     };
30
31     class Fish : public Animal
32     {
33         public:
34             void Eat()
35             {
```

```

36         cout << "The fish ate!" << endl;
37     }
38 };
39
40 int main()
41 {
42     Dog    myDog ;
43     Bird    myBird;
44     Fish    myFish;
45     Animal myAnimal;
46
47     myAnimal.Eat();
48
49     Animal *animal1 = &myDog;
50     animal1->Eat();
51     Animal *animal2 = &myBird;
52     animal2->Eat();
53     Animal *animal3 = &myFish;
54     animal3->Eat();
55
56 }

```

```

The animal ate!
The dog ate!
The bird ate!

The fish ate!

```

همانطور که مشاهده می‌کنید 4 کلاس مختلف تعریف کرده‌ایم. Animal کلاس پایه است و سه کلاس دیگر از آن مشتق می‌شوند. هر کلاس متد Eat() مربوط به خود را دارد. نمونه‌ای از هر کلاس ایجاد کرده‌ایم (42-45). سپس متد Eat() را به وسیله نمونه ایجاد شده از کلاس Animal فراخوانی می‌کنیم (خط 47). در مرحله بعد چندریختی روی می‌دهد. همانطور که در مثال بالا مشاهده می‌کنید شیء Dog را با استفاده از عمل UpCast برابر نمونه ایجاد شده از کلاس Animal قرار می‌دهیم (خط 49) و متد Eat() را بار دیگر فراخوانی می‌کنیم (خط 50). حال با وجود اینکه ما از نمونه کلاس Animal استفاده کرده‌ایم ولی متد Eat() کلاس Dog فراخوانی می‌شود. این به دلیل تأثیر چند ریختی است. سپس دو شیء دیگر (Bird و Fish) را برابر نمونه ایجاد شده از کلاس Animal قرار می‌دهیم (خطوط 51 و 53) و متد Eat() مربوط به هر یک را فراخوانی می‌کنیم (خطوط 52 و 54). اجازه دهید که برنامه بالا را اصلاح کنیم تا مفهوم چند ریختی را بهتر متوجه شوید :

```

1  #include <iostream>
2  using namespace std;
3
4  class Animal
5  {
6      public:
7          virtual void Eat()
8          {
9              cout << "The animal ate!" << endl;

```

```
10     }
11 };
12
13 class Dog : public Animal
14 {
15     public:
16     void Eat()
17     {
18         cout << "The dog ate!" << endl;
19     }
20     void Run()
21     {
22         cout << "The dog ran!" << endl;
23     }
24 };
25
26 class Bird : public Animal
27 {
28     public:
29     void Eat()
30     {
31         cout << "The bird ate!" << endl;
32     }
33     void Fly()
34     {
35         cout << "The bird flew!" << endl;
36     }
37 };
38
39 class Fish : public Animal
40 {
41     public:
42     void Eat()
43     {
44         cout << "The fish ate!" << endl;
45     }
46     void Swim()
47     {
48         cout << "The fish swam!" << endl;
49     }
50 };
51
52 int main()
53 {
54     Animal *animal1 = new Dog;
55     Animal *animal2 = new Bird;
56     Animal *animal3 = new Fish;
57
58
59     Dog *myDog = (Dog*)animal1;
60     myDog->Run();
61     Bird *myBird = (Bird*)animal2;
62     myBird->Fly();
63     Fish *myFish = (Fish*)animal3;
64     myFish->Swim();
65
66 }
```

```
}
```

```
The dog ran!
The bird flew!
The fish swam!
```

در بالا سه شیء از کلاس Animal ایجاد و آنها را بوسیله سه سازنده از کلاسهای مشتق مقدار دهی اولیه کرده ایم (خطوط 54-56). سپس با استفاده از عمل DownCast اشیاء ایجاد شده از کلاس Animal را در نمونه‌هایی از کلاسهای مشتق ذخیره می‌کنیم (خطوط 59، 61 و 63). وقتی این کار را انجام دادیم می‌توانیم متدهای مخصوص به هر یک از کلاسهای مشتق را فراخوانی کنیم (خطوط 60، 62 و 64).

رابط (interface)

رابط‌ها یا اینترفیس‌ها شبیه به کلاسها هستند اما فقط شامل تعاریفی برای متدها و خواص (Property) می‌باشند. رابط‌ها را می‌توان به عنوان پلاگین‌های کلاس‌ها در نظر گرفت. کلاسی که یک رابط خاص را پیاده‌سازی می‌کند لازم است که کدهایی برای اجرا توسط اعضا و متدهای آن فراهم کند چون اعضا و متدهای رابط هیچ کد اجرایی در بدنه خود ندارند. اجازه دهید که نحوه تعریف و استفاده از یک رابط در کلاس را توضیح دهیم :

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  __interface ISample
6  {
7      void ShowMessage(string message);
8  };
9
10 class Sample : public ISample
11 {
12     public:
13         void ShowMessage(string message)
14         {
15             cout << message << endl;
16         }
17 };
18
19 int main()
20 {
21     Sample sample;
22
23     sample.ShowMessage("Implemented the ISample Interface!");
24
25 }
```

```
}
```

Implemented the ISample Interface!

در خطوط 5-8 یک رابط به نام ISample تعریف کرده ایم. در تعریف رابط ها از __interface استفاده می شود (خط 5). بر طبق قراردادهای نامگذاری، رابط ها به شیوه پاسکال نامگذاری می شوند و همه آنها باید با حرف I شروع شوند. یک متد در داخل بدنه رابط تعریف می کنیم (خط 7). به این نکته توجه کنید که متد تعریف شده فاقد بدنه است و در آخران باید از سیمیکولن استفاده شود.

وقتی که متد را در داخل رابط تعریف می کنید فقط لازم است که عنوان متد (نوع، نام و پارامترهای آن) را بنویسید. به این نکته نیز توجه کنید که متدها و خواص تعریف شده در داخل رابط سطح دسترسی ندارند چون باید همیشه هنگام اجرای کلاسها در دسترس باشند. وقتی یک کلاس نیاز به اجرای یک رابط داشته باشد، از همان روشی که در وراثت استفاده می کردیم، استفاده می کنیم، یعنی بعد از علامت دو نقطه باید کلمه کلیدی public را قبل از نام رابط بنویسیم (خط 10). کلاسی که رابط را اجرا می کند کدهای واقعی را برای اعضای آن فراهم می کند. همانطور که در مثال بالا می بینید کلاس Sample، متد ShowMessage() رابط ISample را اجرا و تغذیه می کند. برای روشن شدن کاربرد رابط ها به مثال زیر توجه کنید:

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  class CA
6  {
7      public:
8          string FullName;
9          int Age;
10
11      CA(string fullname, int age)
12      {
13          this->FullName = fullname;
14          this->Age      = age;
15      }
16  };
17
18  class CB
19  {
20      public:
21          string FirstName;
22          string LastName;
23          int PersonsAge;
24
25      CB(string firstname, string lastname, int personage)
26      {
27          this->FirstName = firstname;
28          this->LastName  = lastname;
```

```

29         this->PersonsAge = personage;
30     }
31 };
32
33 static void PrintInfo(CA item)
34 {
35     cout << "Name: " << item.FullName << " , " << "Age: " << item.Age;
36 }
37
38 int main()
39 {
40     CA a("John Doe", 35);
41
42     PrintInfo(a);
43
44 }

```

```
Name: John Doe , Age: 35
```

در کد بالا دو کلاس CA و CB تعریف شده‌اند، در کلاس CA دو فیلد به نام FullName و Age و در کلاس CB سه فیلد به نام‌های FirstName، LastName و PersonsAge تعریف کرده‌ایم. در کلاس Program یک متد به نام PrintInfo() داریم که یک پارامتر از نوع کلاس CA دارد. به شکل ساده در این متد مقدار فیلدهای شیء ای که به این متد ارسال شده است چاپ می‌شود. در متد main() یک شیء از کلاس CA ساخته‌ایم و فیلدهای آن را مقدار دهی کرده‌ایم. سپس این شیء را به متد PrintInfo() ارسال می‌کنیم. کلاس‌های CA و CB از نظر مفهومی شبیه یکدیگر هستند مثلاً کلاس CA فیلد FullName را برای نمایش نام و نام خانوادگی دارد ولی کلاس CB برای نمایش نام و نام خانوادگی دو فیلد جدا از هم به نام‌های FirstName و LastName را دارد. و همچنین یک فیلد برای نگهداری مقدار سن داریم که در کلاس CA نام آن Age و در کلاس CB نام آن PersonAge می‌باشد. مشکل اینجا است که اگر ما یک شیء از کلاس CA را به متد (PrintInfo) ارسال کنیم از آنجایی که در داخل بدنه این متد فقط مقدار دو فیلد چاپ می‌شود اگر بخواهیم یک شیء از کلاس CB را به آن ارسال کنیم که دارای سه فیلد است با خطا مواجه می‌شویم (زیرا متد PrintInfo با ساختار کلاس CA سازگار است و فیلدهای CB را نمی‌شناسد). برای رفع این مشکل باید ساختار دو کلاس CA و CB را شبیه هم کنیم و این کار را با استفاده از Interface انجام می‌دهیم:

```

1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  __interface IInfo
6  {
7      string GetName();
8      string GetAge();
9  };
10
11 class CA : public IInfo
12 {
13     public:

```

```
14     string FullName;
15     int Age;
16
17     CA(string fullname, int age)
18     {
19         this->FullName = fullname;
20         this->Age = age;
21     }
22
23     string GetName() { return FullName; }
24     string GetAge() { return to_string(Age); }
25 };
26
27 class CB : public IInfo
28 {
29     public:
30     string FirstName;
31     string LastName;
32     int PersonsAge;
33
34     CB(string firstname, string lastname, int personage)
35     {
36         this->FirstName = firstname;
37         this->LastName = lastname;
38         this->PersonsAge = personage;
39     }
40
41     string GetName() { return FirstName + " " + LastName; }
42     string GetAge() { return to_string(PersonsAge); }
43 };
44
45 void PrintInfo(IInfo& item)
46 {
47     cout << "Name: " << item.GetName() << " , " << "Age: " << item.GetAge() << endl;
48 }
49
50 int main()
51 {
52     CA a("John Doe", 35);
53     CB b("Jane", "Doe", 33);
54
55     PrintInfo(a);
56     PrintInfo(b);
57 }
58 }
```

کد بالا را می‌توان به اینصورت توضیح داد که در خط 9-5 یک رابط به نام IInfo تعریف و آن را در خطوط 11 و 27 توسط دو کلاس CA و CB پیاده سازی کرده‌ایم. چون این دو کلاس وظیف دارند متدهای این رابط را پیاده سازی کنند پس در خطوط 23-24 و 41-42 کدهای بدنه دو متد این رابط را آن طور که می‌خواهیم، می‌نویسیم. در خط 45 متد PrintInfo() را طوری دستکاری می‌کنیم که یک پارامتر از نوع رابط دریافت کند. حال زمانی که دو شیء از دو کلاس CA و CB در دو خط 52 و 53 ایجاد می‌کنیم و آنها را در دو خط 55 و 56 به متد PrintInfo() ارسال می‌کنیم، چونکه این دو کلاس رابط IInfo را پیاده سازی کرده‌اند به طور صریح به رابط تبدیل می‌شود. یعنی کلاسی که یک رابط را پیاده سازی کند به طور صریح می‌تواند به رابط تبدیل شود. حال بسته به اینکه شیء کدام کلاس به متد PrintInfo() ارسال شده است، متد مربوط به آن کلاس فراخوانی شده و مقادیر فیلدها چاپ می‌شود. می‌توان چند رابط را در کلاس اجرا کرد :


```
class Sample : public ISample1, public ISample2, public ISample3
{
    //Implement all interfaces
};
```

درست است که می‌توان از چند رابط در کلاس استفاده کرد ولی باید مطمئن شد که کلاس می‌تواند همه اعضای رابطها را تغذیه کند. اگر یک کلاس از کلاس پایه ارث ببرد و در عین حال از رابطها هم استفاده کند، در این صورت می‌توان نام کلاس پایه را هم ذکر کرد. به شکل زیر :

```
class Sample : public BaseClass, public ISample1, public ISample2
{
};
```

رابطها حتی می‌توانند رابطهای دیگر را پیاده سازی یا اجرا کنند. به مثال زیر توجه کنید :

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

__interface IBase
{
    void BaseMethod();
};

__interface ISample : IBase
{
    void ShowMessage(string message);
};

class Sample : public ISample
{
public:
    void ShowMessage(string message)
    {
        cout << message << endl;
    }

    void BaseMethod()
    {
        cout << "Method from base interface!" << endl;
    }
};

int main()
{
    Sample sample;

    sample.ShowMessage("Implemented the ISample Interface!");
    sample.BaseMethod();
}
```

```
Implemented the ISample Interface!
```

```
Method from base interface!
```

مشاهده می کنید که حتی اگر کلاس Sample فقط رابط ISample را پیاده سازی کند، لازم است که همه اعضای IBase را هم پیاده سازی کند چون ISample از آن ارث بری می کند (خط 8).

ساختار (Struct)

ساختارها یا struct انواع داده ای هستند که توسط کاربر تعریف می شوند (user-define) و می توانند دارای فیلد و متد باشند. با ساختارها می توان نوع داده ای خیلی سفارشی ایجاد کرد. فرض کنید می خواهیم داده ای ایجاد کنیم که نه تنها نام شخص را ذخیره کند بلکه سن و حقوق ماهیانه او را نیز در خود جای دهد. برای تعریف یک ساختار به صورت زیر عمل می کنیم :

```
struct StructName
{
    member1;
    member2;
    member3;
    ...
    member4;
};
```

برای تعریف ساختار از کلمه کلیدی struct استفاده می شود. برای نامگذاری ساختارها از روش نامگذاری struct استفاده می شود. اعضا در مثال بالا (member1-4) می توانند متغیر باشند یا متد. در زیر مثالی از یک ساختار آمده است :

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  struct Employee
6  {
7      public:
8          string name;
9          int age;
10         double salary;
11 };
12
13 int main()
14 {
15     Employee employee1;
16     Employee employee2;
17
18     employee1.name = "Jack";
```

```

19     employee1.age = 21;
20     employee1.salary = 1000;
21
22     employee2.name = "Mark";
23     employee2.age = 23;
24     employee2.salary = 800;
25
26     cout << "Employee 1 Details" << endl;
27     cout << "Name: " << employee1.name << endl;
28     cout << "Age: " << employee1.age << endl;
29     cout << "Salary: " << employee1.salary << endl;
30
31     cout << endl; //Seperator
32
33     cout << "Employee 2 Details" << endl;
34     cout << "Name: " << employee2.name << endl;
35     cout << "Age: " << employee2.age << endl;
36     cout << "Salary: " << employee2.salary << endl;
37
38 }

```

```

Employee 1 Details
Name: Jack
Age: 21
Salary: $1000

Employee 2 Daitails
Name: Mike
Age: 23
Salary: $800

```

برای درک بهتر، کد بالا را شرح می‌دهیم. در خطوط 5-11 یک ساختار تعریف شده است. قبل از نام ساختار از کلمه کلیدی `struct` استفاده می‌کنیم. نام ساختار نیز از روش نامگذاری پاسکال پیروی می‌کند. در داخل بدنه ساختار سه فیلد تعریف کرده‌ایم. این سه فیلد مشخصات `Employee` (کارمند) مان را نشان می‌دهند. مثلاً یک کارمند دارای نام، سن و حقوق ماهانه می‌باشد. همچنین هر سه فیلد به صورت `Public` تعریف شده‌اند بنابراین در خارج از ساختار نیز می‌توان آنها را فراخوانی کرد. در خطوط 15 و 16 دو نمونه از ساختار `Employee` تعریف شده است. تعریف یک نمونه از ساختارها بسیار شبیه به تعریف یک متغیر معمولی است. ابتدا نوع ساختار و سپس نام آن را مشخص می‌کنید. در خطوط 18 تا 24 به فیلدهای مربوط به هر `Employee` مقادیری اختصاص می‌دهید. برای دسترسی به فیلدها در خارج از ساختار باید آنها را به صورت `Public` تعریف کنید. ابتدا نام متغیر را تایپ کرده و سپس علامت دات (.) و در آخر نام فیلد را می‌نویسیم.

وقتی که از عملگر دات استفاده می‌کنیم این عملگر اجازه دسترسی به اعضای مخصوص آن ساختار یا کلاس را به شما می‌دهد. در خطوط 26 تا 36 نشان داده شده که شما چگونه می‌توانید به مقادیر ذخیره شده در هر فیلد ساختار دسترسی یابید. در مورد کلاس در درسهای آینده توضیح خواهیم داد. می‌توان خطوط 15 و 16 کد بالا را به صورت خلاصه زیر هم نوشت :

```
Employee employee1, employee2;
```

برای ایجاد شیء از ساختار می‌توان به صورت زیر هم عمل نمود :

```
struct Employee
{
    public:
        string name;
        int age;
        double salary;
} employee1, employee2;
```

در نتیجه می‌توان خطوط 15 و 16 کد بالا را پاک کرد. به جای خطوط 18-24 کد بالا می‌توان فیلدهای ساختار را به صورت زیر هم مقداردهی کرد :

```
employee1 = { "Jack", 21, 1000 };

employee2 = { "Mark", 23, 800 };
```

می‌توان به ساختار، متد هم اضافه کرد. مثال زیر اصلاح شده مثال قبل است :

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  struct Employee
6  {
7      public:
8          string name;
9          int age;
10         double salary;
11
12         void SayThanks()
13         {
14             cout << name << " thanked you!" << endl;
15         }
16     };
17
18     int main()
19     {
20         Employee employee1;
21         Employee employee2;
22
23         employee1.name = "Jack";
24         employee1.age = 21;
25         employee1.salary = 1000;
26
27         employee2.name = "Mark";
```

```

28     employee2.age    = 23;
29     employee2.salary = 800;
30
31     cout << "Employee 1 Details" << endl;
32     cout << "Name: "      << employee1.name    << endl;
33     cout << "Age: "       << employee1.age     << endl;
34     cout << "Salary: $"   << employee1.salary  << endl;
35     employee1.SayThanks();
36
37     cout << endl; //Seperator
38
39     cout << "Employee 2 Details" << endl;
40     cout << "Name: "      << employee2.name    << endl;
41     cout << "Age: "       << employee2.age     << endl;
42     cout << "Salary: $"   << employee2.salary  << endl;
43     employee2.SayThanks();
44
45 }

```

```

Employee 1 Details
Name: Jack
Age: 21
Salary: $1000
Jack thanked you!

Employee 2 Details
Name: Mark
Age: 23
Salary: $800
Mark thanked you!

```

در خطوط 12 تا 15 یک متد در داخل ساختار تعریف شده است. این متد یک پیام را در صفحه نمایش نشان می‌دهد و مقدار فیلد name را گرفته و یک پیام منحصر به فرد برای هر نمونه نشان می‌دهد. برای فراخوانی متد، به جای اینکه بعد از علامت دات نام فیلد را بنویسیم، نام متد را نوشته و بعد از آن همانطور که در مثال بالا مشاهده می‌کنید (خطوط 35 و 43) پرانتزها را قرار می‌دهیم و در صورتی که متد به آرگومان هم نیاز داشت در داخل پرانتز آنها را می‌نویسیم.

ایجاد آرایه‌ای از کلاسها

در این درس به شما نشان می‌دهیم که، چگونه می‌توان آرایه‌ای از کلاس‌ها ایجاد کرد. ساخت آرایه‌ای از کلاس‌ها تقریباً شبیه به ایجاد آرایه‌ای از انواع داده‌ای مانند int است. به عنوان مثال می‌توان آرایه‌ای از کلاس Person ایجاد کرد:

```

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

class Person

```

```
{
    public:
        string Name;

        Person()
        {
        }

        Person(string name)
        {
            Name = name;
        }
};

int main()
{
    Person people[3];

    people[0] = Person("Johnny");
    people[1] = Person("Mike");
    people[2] = Person("Sonny");

    for(Person person : people)
    {
        cout << person.Name << endl;
    }
}
```

```
Johnny
Mike
Sonny
```

ابتدا یک کلاس که دارای یک فیلد است، تعریف می‌کنیم. سپس یک آرایه از آن (کلاس ایجاد شده) تعریف می‌کنیم و سپس فیلدهای آن را، مانند بالا مقدار دهی می‌کنیم. در نهایت مقدار فیلد هر یک از اشیاء را، با استفاده از یک حلقه foreach نمایش می‌دهیم.

Template

Template ها، کلاس‌ها یا متدهای هستند که، بسته به نوع داده‌ای که به آنها اختصاص داده می‌شود، رفتارشان را سازگار می‌کنند. به عنوان مثال، یا استفاده از Template، می‌توان یک متد تعریف کرد که هر نوع داده‌ای را قبول کند. همچنین می‌توان یک متد ایجاد کرد که بسته به نوع دریافتی، مقادیری از انواع داده‌ای مانند int، double یا string را نشان دهد. اگر از Template ها استفاده نکنید، باید چند متد و یا حتی چندین متد سربارگذاری شده برای نمایش هر نوع ممکن ایجاد کنید:

```
public:
    void Show(int number)
    {
        cout << number << endl;
    }

    void Show(double number)
    {
        cout << number << endl;
    }

    void Show(string message)
    {
        cout << message << endl;
    }
}
```

با استفاده از Template ها، می‌توان متدی ایجاد کرد که هر نوع داده‌ای را قبول کند:

```
template<class E>
void Show(E item)
{
    cout << item << endl;
}
```

درباره Template ها در درسهای بعد مطالب بیشتری توضیح می‌دهیم.

متدهای عمومی

اگر بخواهید چندین متد با عملکرد مشابه ایجاد کنید و فقط تفاوت آنها در نوع داده‌ای باشد که قبول می‌کنند (مثلاً یکی نوع int و دیگری نوع double را قبول کند) می‌توان از متدهای عمومی برای صرفه جویی در کدنویسی استفاده کرد. ساختار عمومی یک متد عمومی به شکل زیر است:

```
template<class Type, ...>
return-type function - name(Type arg1, ...)
{
    //Body of function template
}
```

مشاهده می‌کنید که بعد از نام متد یک نوع در داخل دو علامت بزرگتر و کوچکتر آمده است (<type>) که همه انواع در C++ می‌توانند جایگزین آن شوند. برنامه زیر مثالی از نحوه استفاده از متد عمومی می‌باشد:

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

template<class E>
void Show(E item)
{
    cout << item << endl;
}

int main()
{
    int    intValue    = 5;
    double doubleValue = 10.54;
    string stringValue = "Hello";
    bool   boolValue   = true;

    Show(intValue);
    Show(doubleValue);
    Show(stringValue);
    Show(boolValue);
}
```

```
5
10.54
Hello
1
```

یک متد جنریک ایجاد کرده ایم که هر نوع داده ای را قبول کرده و مقادیر آنها را نمایش می دهد (خطوط 9-5). سپس داده های مختلفی با وظایف یکسان به آن ارسال می کنیم. متد نیز نوع E را بسته به نوع داده ای که به عنوان آرگومان ارسال شده است تغییر می دهد. همچنین هنگام فراخوانی متد جنریک صریحاً می توانید نوعی را که به وسیله آن مورد استفاده قرار می گیرد ذکر کنید (البته لازم نیست). به عنوان مثال فراخوانی های متد بالا را می توان به صورت زیر هم نوشت :

```
Show<int>(intValue);
Show<double>(doubleValue);
Show<string>(stringValue);
Show<bool>(boolValue);
```

به یک نکته در مورد استفاده از متدهای جنریک توجه کنید و آن این است که، شما می توانید در داخل کدهای مربوط به متد محاسبات انجام دهید، مثلاً دو عدد را با هم جمع کنید ولی باید مراقب باشید که نوع اعداد با هم یکی باشد، مثلاً هر دو int و یا double باشند :

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

template<class E>
```



```
void Show(E x, E y)
{
    cout << x + y << endl;
}

int main()
{
    Show(10, 5);

    Show(1.5, 1.0);
}
```

```
15
2.5
```

شما می‌توانید چندین نوع خاص را برای متد جنریک ارسال کنید، برای این کار هر نوع را به وسیله کاما از دیگری جدا کنید.

```
template<class E, class C>
void Show(E x, C y)
{
    cout << x << endl;
    cout << y << endl;
}
```

به مثال زیر که در آن دو مقدار مختلف به متد ارسال شده است توجه کنید :

```
Show(5, true);

// OR

Show<int, bool>(5, true);
```

مشاهده می‌کنید که X با نوع int و Y با نوع bool جایگزین می‌شود. این نکته را نیز یادآور شویم که شما می‌توانید دو آرگومان

هم نوع را هم به متد ارسال کنید :

```
Show(5, 10);

// OR

Show<int, int>(5, true);
```

سربارگذاری متدهای عمومی

همانطور که یک متد معمولی را می‌توانیم سربارگذاری کنیم، متدهای عمومی را نیز می‌توانیم سربارگذاری کنیم. در سربارگذاری های مختلف یک تابع، نام آن تغییر نمی‌کند، ولی نوع یا تعداد پارامترهای سربارگذاری های مختلف با یکدیگر متفاوت است. مثال زیر که یک متد قالب را سربارگذاری می‌کند را در نظر بگیرید :

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

template<class Type>
void ShowMessage(Type T)
{
    cout << "Double version of the method was called." << endl;
}

void ShowMessage(int number)
{
    cout << "Integer version of the method was called." << endl;
}

int main()
{
    ShowMessage(9);
    ShowMessage(9.99);
}
```

```
Integer version of the method was called.
Double version of the method was called.
```

زمانی که کامپایلر با یک متد سربارگذاری شده رو به رو می‌شود، ابتدا بررسی می‌کند که آیا یک متد عادی برای اینکار وجود دارد یا خیر؟ اگر وجود داشت آن را فراخوانی می‌کند، در غیر اینصورت بررسی می‌کند که آیا متد عمومی وجود دارد یا خیر؟ اگر وجود داشت آن را فراخوانی می‌کند و اگر هیچ متدی وجود نداشت، باعث بروز خطا می‌شود. در مثال بالا می‌توانید مشاهده کنید که، اگر یک مقدار از نوع صحیح را به متد ارسال کنیم (خط 17) در نتیجه چون یکی از سربارگذاری های این متد یک عدد از نوع صحیح دریافت می‌کند (خط 10)، در نتیجه کدهای بدنه این متد اجرا می‌شوند (خط 12). در فراخوانی دوم (خط 18) چون یک مقدار از نوع double به متد ارسال کرده‌ایم در نتیجه کدهای بدنه نسخه عمومی متد یعنی خط 7 اجرا می‌شود.

کلاس‌های عمومی

تعریف یک کلاس جنریک بسیار شبیه به تعریف یک متد جنریک است. کلاس جنریک دارای یک علامت بزرگتر و کوچکتر و یک نوع پارامتر خاص می‌باشد. برنامه زیر مثالی از یک کلاس جنریک می‌باشد :

```
#include <iostream>
```

```
using namespace std;

template<class T>
class GenericClass
{
    private:
        T someField;

    public:
        GenericClass(T someVariable)
        {
            someField = someVariable;
        }

        T getSomeProperty()
        {
            return someField;
        }

        void setSomeProperty(T value)
        {
            someField = value;
        }
};

int main()
{
    GenericClass<double> genericDouble(30.50);
    GenericClass<int> genericInt(10);

    cout << "genericDouble.SomeProperty = " << genericDouble.getSomeProperty() << endl;
    cout << "genericInt.SomeProperty = " << genericInt.getSomeProperty() << endl;
}
```

```
genericDouble.SomeProperty = 30.50
genericInt.SomeProperty = 10
```

در مثال بالا یک کلاس جنریک که دارای یک فیلد، یک خاصیت و یک سازنده است را ایجاد می‌کنیم. تمام مکانهایی که ورودی T در آنها قرار دارد بعداً توسط انواعی که مد نظر شما است، جایگزین می‌شوند. وقتی یک نمونه از کلاس جنریک تان ایجاد می‌کنید، یک نوع هم برای آن در نظر بگیرید (<int>). مانند متدهای جنریک می‌توانید چندین نوع پارامتر به کلاسهای جنریک اختصاص دهید.

```
template<class T1, class T2, class T3>
class GenericClass
{
    private:
        T1 someField1;
        T2 someField2;
        T3 someField3;
};
```

```
genericDouble.SomeProperty = 30.50
genericInt.SomeProperty = 10
```

کلاسهای جنریک می‌توانند از کلاسهای جنریک ارث بری کنند اما کلاسهای فرزند اجازه داشتن ویژگی‌های کلاسهای جنریک والد را ندارند:

```
template<class T1> class Parent
{
    //some code
};

template<class T> class Child : public Parent<T>
{
    //some code
};
```

کلاسهای غیر جنریک می‌توانند از کلاسهای جنریک ارث بری کنند، اما باید نوع آرگومان‌های کلاس جنریک باید در زمان ارث بری (در حین کدنویسی) به صورت زیر مشخص شوند:

```
class Child : public GenericClass<int>
{
    //some code
};
```

یک کلاس جنریک هم می‌تواند از یک کلاس غیر جنریک ارث بری کند.

```
template<class T> class Child : public Parent
{
    //some code
};
```

سربارگذاری عملگرها (Operator Overloading)

سربارگذاری عملگرها به شما اجازه می‌دهد که رفتار عملگرهای C++ را بسته به نوع عملوندهای آنها سفارشی کنید. سربارگذاری عملگرها همچنین به عملگر اجازه می‌دهد که یک شیء را به روشی دیگر ترجمه کند. به کد زیر توجه کنید :

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

class MyNumber
{
public:
    int number;
};

int main()
```

```
{
    MyNumber firstNumber;
    MyNumber secondNumber;

    firstNumber.Number = 10;
    secondNumber.Number = 5;

    MyNumber sum = firstNumber + secondNumber;

    cout << "Sum = " << sum.Number;
}
```

خط پررنگ شده در کد بالا (خط 19) کد قابل قبولی نیست چون کامپایلر نمی‌تواند دو شیء را با هم جمع کند. رفتاری که ما از کد بالا انتظار داریم اضافه کردن مقادیر به خاصیت Number دو عملوند و سپس ایجاد یک شیء جدید که حاصل جمع دو مقدار در داخل آن قرار بگیرد. سپس این شیء جدید به متغیر sum تخصیص داده شود.

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

class MyNumber
{
public:
    int number;

    friend MyNumber operator +(MyNumber n1, MyNumber n2)
    {
        MyNumber result;
        result.number = n1.number + n2.number;
        return result;
    }
};

int main()
{
    MyNumber firstNumber;
    MyNumber secondNumber;

    firstNumber.number = 10;
    secondNumber.number = 5;

    MyNumber sum = firstNumber + secondNumber;

    cout << "Sum = " << sum.number;
}
```

برای سربارگذاری عملگرها به صورت زیر عمل کنید :

```
class className
```

```
{
    public:
        returnType operator symbol (arguments)
        {
        }
}
```

همانطور که مشاهده می‌کنید در سربارگذاری عملگرها از یک متد در داخل کلاس استفاده می‌شود. همانطور که در کد بالا مشاهده می‌کنید بعد از کلمه کلیدی operator از یک عملگر مانند + یا - استفاده می‌کنیم.

مزایای سربارگذاری عملگرها

- سربارگذاری عملگرها برنامه نویس را قادر می‌سازد تا خوانایی برنامه نویس را بالاتر ببرد. برای مثال برای جمع کردن ماتریس‌ها نوشتن $M1+M2$ خوانایی بالاتری نسبت به انجام عملیات جمع با استفاده از یک متد مانند `M1.add(M2)` دارد.
- در سربارگذاری عملگرها از ساختاری مشابه عملگرهای از پیش تعریف شده پیروی می‌شود.
- سربارگذاری عملگرها باعث درک آسان‌تر برنامه می‌شود.

محدودیت‌های سربارگذاری عملگرها

- فقط عملگرهای پیشفرض مانند +، -، *، / و ... می‌توانند سربارگذاری شوند.
- تعداد عملوندهای یک عملگر نمی‌توانند تغییر کنند. برای مثال نمی‌تواند برای عملگر + سه عملوند را در نظر گرفت.
- اولویت عملگرها را نمی‌توان تغییر داد.
- عملگر سربارگذاری شده حداقل باید یک عملوند داشته باشد.
- عملگرهای =، []، ()، > - باید به عنوان member function (داخل یک کلاس) تعریف شوند. عملگر باقیمانده تقسیم هم می‌تواند عضو یک متد باشد و هم می‌تواند نباشد.
- برخی از عملگرها مانند =، & و کاما به صورت پیش فرض از قبل سربارگذاری شده‌اند.
- عملگرهایی مانند ::، و ؟: نمی‌توانند سربارگذاری شوند.

برای سربارگذاری عملگرها باید نکات زیر را در نظر بگیرید:

- در سربارگذاری عملگرها می‌توان بدون تغییر ساختار، معنا را تغییر داد. برای مثال اگر بخواهیم عملگر + را سربارگذاری کنیم نمی‌توانیم ساختار آن که دارای دو عملوند است را تغییر دهیم ولی می‌توانیم به جای جمع دو عدد معمولی، جمع دو عدد مختلط را سربارگذاری کنیم.
- سربارگذاری عملگرها نمی‌تواند اولویت عملگرها را تغییر دهد.
- نمی‌توانیم یک عملگر جدید تعریف کنیم.
- زمانی که عملگرهایی نظیر && و || سربارگذاری می‌شوند، ویژگی مدار کوتاه بودن پردازش را از دست می‌دهند.
- عملگرهای سربارگذاری شده نمی‌توانند آرگومان پیش فرض داشته باشند.
- همه عملگرهای سربارگذاری شده به جز عملگر انتساب توسط کلاس فرزند به ارث برده می‌شوند.
- تعداد عملوندها را در سربارگذاری یک عملگر نمی‌توان تغییر داد.

سربارگذاری عملگرها می‌تواند به دو روش پیاده سازی شود که عبارتند از:

- استفاده از یک member function
- با استفاده از یک friend function

تفاوت بین این دو روش را می‌توانید در جدول زیر مشاهده کنید:

friend function	member function
تعداد پارامترهایی که می‌تواند پاس داده شود بیشتر است	تعداد پارامترهایی که می‌تواند پاس داده شود فقط یکی است و شیء فراخوانی شده به صورت ضمنی فقط یک عملوند دارد.
عملگرهای Unary فقط یک پارامتر را به صورت صریح می‌گیرند	عملگرهای Unary هیچ پارامتر صریحی نمی‌گیرند
عملگرهای Binary دو پارامتر را به صورت صریح می‌گیرند	عملگرهای Binary فقط یک پارامتر صریح می‌گیرند
عملوند سمت چپ به شیء یک کلاس نیاز ندارد	عملوند سمت چپ باید به وسیله شیء فراخوانی شود
نوشتن موارد زیر مجاز است $Obj2 = Obj1 + 10$ $Obj2 = 10 + Obj1$	نوشتن $Obj2 = Obj1 + 10$ مجاز است ولی $Obj2 = 10 + Obj1$ مجاز نیست

سربارگذار عملگرهای تک عملوندی (Unary)

عملگرهایی که فقط با یک عملوند کار می‌کنند به عنوان عملگرهای Unary شناخته می‌شوند. برای مثال ++ (عملگر افزایش)، - (عملگر کاهش)، - (عملگر منهای تک عملوندی)، ! (عملگر منطقی نقیض) و ... در این دسته قرار دارند. برنامه زیر نشان می‌دهد که چگونه عملگر منهای تک عملوندی (-) را به وسیله متد عضو سربارگذاری می‌کنیم:

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

class Number
{
private:
    int x;

public:
    Number(int x)
    {
        this->x = x;
    }

    void operator -()
    {
        x = -x;
    }

    void display()
    {
        cout << "x = " << x << endl;
    }
};

int main()
{
    Number n1(10);
    -n1;
    n1.display();
}
```

-10

در برنامه بالا مقدار x تغییر می‌کند و با استفاده از متد display() چاپ می‌شود. همانطور که در برنامه زیر مشاهده می‌کنید ما می‌توانیم یک شیء از کلاس Number را نیز بعد از تغییر x برگردانیم:

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

class Number
{
private:
```



```

    int x;

public:
    Number(int x)
    {
        this->x = x;
    }

    Number operator -()
    {
        x = -x;
        return Number(x);
    }

    void display()
    {
        cout << "x = " << x << endl;
    }
};

int main()
{
    Number n1(10);
    Number n2 = -n1;
    n2.display();
}

```

-10

زمانی که یک friend function برای سربارگذاری یک عملگر Unary مورد استفاده قرار می‌گیرد باید به نکات زیر توجه شود:

- تابع فقط یک عملوند را به عنوان پارامتر قبول می‌کند.
- عملوند یک شیء از یک کلاس است.
- تابع می‌تواند به اعضای خصوصی (private) فقط از طریق شیء دسترسی داشته باشد.
- تابع ممکن است یک مقدار برگرداند و ممکن است هیچ مقداری را برنگرداند.

برنامه زیر نشان می‌دهد که چگونه می‌توان یک عملگر Unary را با استفاده از friend function سربارگذاری کرد:

```

#include <iostream>
using namespace std;

class Number
{
private:
    int x;
public:
    Number(int x)
    {
        this->x = x;
    }
}

```

```

    friend Number operator -(Number &);

    void display()
    {
        cout << "x = " << x << endl;
    }
};

Number operator -(Number &n)
{
    return Number(-n.x);
}

int main()
{
    Number n1(20);
    Number n2 = -n1;
    n2.display();
}

```

-20

سربارگذاری عملگرهای پیشوندی (prefix)

ساختار کلی سربارگذاری عملگرهای پیشوندی افزایش (++) و کاهش (-) به شکل زیر می باشد:

```

return-type operator ++()
{

}

```

برنامه زیر نشان می دهد چگونه می توانیم عملگرهای پیشوندی کاهش و افزایش را سربارگذاری کنیم:

```

#include <iostream>
using namespace std;
class Number
{
    private:
        int x;

    public:
        Number(int x)
        {
            this->x = x;
        }

        Number operator ++()
        {

```

```

        x = x + 1;
        return Number(x);
    }

    Number operator --()
    {
        x = x - 1;
        return Number(x);
    }

    void display()
    {
        cout << "x = " << x << endl;
    }
};

int main()
{
    Number n1(20);
    Number n2 = ++n1;
    n2.display();
    Number n3(20);
    Number n4 = --n3;
    n4.display();
}

```

```

x = 21
x = 19

```

سربارگذاری عملگرهای پسوندی (postfix)

برای سربارگذاری عملگرهای پسوندی افزایش و کاهش ما باید یک `int` اضافه را به عنوان پارامتر مشخص کنیم تا از تداخل با عملگرهای پیشوندی جلوگیری کند. ساختار کلی سربارگذاری عملگر پسوندی افزایش به صورت زیر است:

```

return-type operator ++(int)
{
}

```

پارامتر `int` یک پارامتر اضافی است و نیازی به دریافت آن نداریم. برنامه زیر نشان می‌دهد که چگونه می‌توانیم عملگرهای پسوندی افزایش و کاهش را سربارگذاری کنیم:

```

#include <iostream>
using namespace std;
class Number
{

```

```
private:
    int x;

public:
    Number(int x)
    {
        this->x = x;
    }

    Number operator ++(int)
    {
        return Number(x++);
    }

    Number operator --(int)
    {
        return Number(x--);
    }

    void display()
    {
        cout << "x = " << x << endl;
    }
};

int main()
{
    Number n1(20);
    Number n2 = n1++;
    n1.display();
    n2.display();
    Number n3 = n2--;
    n3.display();
    n2.display();
}
```

```
x = 21
x = 20
x = 20
x = 19
```

سربارگذاری عملگرهای باینری

همانطور که عملگرهای Unary می‌توانند سربارگذاری شوند، همچنین ما می‌توانیم عملگرهای باینری را نیز سربارگذاری کنیم. ساختار

سربارگذاری یک عملگر باینری با استفاده از member function به صورت زیر می‌باشد:

```
return-type operator op(ClassName &)
{
```

```
}
```

ساختار سربارگذاری یک عملگر باینری با استفاده از friend function به صورت زیر می‌باشد:

```
return-type operator op(ClassName &, ClassName &)
{
}
}
```

برنامه زیر نشان می‌دهد که چگونه می‌توان عملگر باینری + را با استفاده از member function سربارگذاری کرد:

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

class Number
{
private:
    int x;

public:
    Number() {}

    Number(int x)
    {
        this->x = x;
    }

    Number operator +(Number &n)
    {
        Number temp;
        temp.x = x + n.x;
        return temp;
    }

    void display()
    {
        cout << "x = " << x << endl;
    }
};

int main()
{
    Number n1(20);
    Number n2(10);
    Number n3 = n1 + n2;
    n3.display();
}
```

```
x = 30
```

برنامه زیر نشان می‌دهد که چگونه می‌توان عملگر باینری + را با استفاده از friend function سربارگذاری کرد:

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

class Number
{
    private:
        int x;

    public:
        Number() {}

        Number(int x)
        {
            this->x = x;
        }

        friend Number operator +(Number &, Number &);

        void display()
        {
            cout << "x = " << x << endl;
        }
};

Number operator +(Number &n1, Number &n2)
{
    Number temp;
    temp.x = n1.x + n2.x;
    return temp;
}

int main()
{
    Number n1(20);
    Number n2(10);
    Number n3 = n1 + n2;
    n3.display();
}
```

x = 30

عملگرهایی از قبیل =، ()، و -> نمی‌توانند با استفاده از یک friend function سربارگذاری شوند.

سربارگذاری عملگرهای خاص

برخی از عملگرهای خاص در C++ عبارتند از:

- new : به منظور تخصیص حافظه مورد استفاده قرار می‌گیرد.
- delete : به منظور آزاد کردن حافظه مورد استفاده قرار می‌گیرد.

- () و [] : عملگرهای زیرمجموعه.
- >- : عملگر دسترسی به اعضاء.

C++ به برنامه نویس اجازه می‌دهد تا عملگرهای new و delete را سربارگذاری کند که وظایف این عملگرهای عبارتند از:

- به منظور افزودن ویژگی‌های بیشتر در هنگام تخصیص و آزاد کردن حافظه
- به کاربران اجازه می‌دهد تا برنامه خود را دیباگ کنند و عملیات تخصیص و آزاد سازی حافظه را ردیابی کنند.

ساختار سربارگذاری عملگر new به صورت زیر می‌باشد:

```
void* operator new(size_t size);
```

پارامتر size، مشخص می‌کند چه میزان حافظه برای نوع داده size_t تخصیص داده شود. ساختار سربارگذاری عملگر delete به صورت زیر می‌باشد:

```
void operator delete(void*);
```

این متد یک پارامتر از نوع void* دریافت می‌کند و هیچ چیز بر نمی‌گرداند. هر دو تابعی که برای سربارگذاری new و delete نوشته‌ایم به صورت پیش فرض از نوع static هستند و با this نمی‌توان به آن‌ها دسترسی داشت. برای حذف یک آرایه از اشیاء عملگر delete[] باید سربارگذاری شود. برنامه زیر سربارگذاری عملگرهای new و delete را نشان می‌دهد:

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

class Number
{
private:
    int x;

public:
    Number(int x)
    {
        this->x = x;
    }

    void* operator new(size_t size)
    {
        void *ptr = ::new int[size];
        cout << "Memory allocated of size: " << size << endl;
        return ptr;
    }

    void operator delete(void *ptr)
    {
        cout << "Memory deallocated" << endl;
    }
}
```

```

void display()
{
    cout << "x = " << x << endl;
}

};

int main()
{
    Number *n = new Number(10);
    n->display();
    delete n;
}

```

```

Memory allocated of size: 4
x = 10
Memory deallocated
Press any key to continue . . .

```

در برنامه بالا `new::` و `delete::` به عملگرهای `new` و `delete` سراسری اشاره می‌کنند. زمانی که `new` فراخوانی می‌شود، کامپایلر متد سربارگذاری شده برای `new` را فراخوانی می‌کند و همچنین به صورت خودکار متد سازنده را نیز فراخوانی می‌کند. سربارگذاری عملگرهای `new` و `delete` دارای مزایای زیر است :

- تابع سربارگذاری شده عملگر `new` می‌تواند یک چند پارامتر را دریافت کند. این کار باعث انعطاف پذیری و شخصی سازی حافظه تخصیص داده شده می‌شود.
- تابع سربارگذاری شده عملگر `delete` عملیات زباله روب (`garbage collection`) را برای اشیاء کلاس‌ها ارائه می‌دهد.
- برنامه نویسان می‌توانند مدیریت خطا را نیز در حین تخصیص حافظه انجام دهند.
- برنامه نویسان می‌توانند از توابع مدیریت حافظه مانند `malloc()`، `realloc()` و `free()` داخل توابع سربارگذاری شده‌ی `new` و `delete` استفاده کنند.

عملگر `[]` برای دسترسی به عناصر یک آرایه مورد استفاده قرار می‌گیرد. متد تعریف شده برای سربارگذاری `[]` یا `()` باید از اعضای یک کلاس و از نوع `non-static` باشد. ساختار کلی سربارگذاری عملگر `[]` به صورت زیر می‌باشد:

```

int& operator [] (int x)
{
}

```

تابع سربارگذاری شده باید یک عدد صحیح را به روش ارجاع برگرداند. مثال زیر سربارگذاری عملگر `[]` را نشان می‌دهد:


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

class Number
{
private:
    int x[5];

public:
    void read(int n)
    {
        cout << "Enter " << n << " numbers: ";
        for (int i = 0; i < n; i++)
        {
            cin >> x[i];
        }
    }

    int& operator [] (int i)
    {
        return x[i];
    }
};

int main()
{
    Number n1;
    n1.read(5);
    cout << "Element is: " << n1[2];
    return 0;
}
```

```
Enter 5 numbers: 1 2 3 4 5
Element is: 3
```

زمانی که چند زیر مجموعه داریم می‌توانیم به جای سربارگذاری `[]`، از سربارگذاری عملگر `()` استفاده می‌کنیم. ساختار کلی سربارگذاری عملگر `()` به صورت زیر می‌باشد:

```
int& operator () (int i, int j,...)
{
}
}
```

مثال زیر سربارگذاری عملگر `()` را نشان می‌دهد:

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

class Matrix
{
private:
    int x[2][2];
```

```

public:
    void read()
    {
        cout << "Enter 2x2 matrix elements: ";

        for (int i = 0; i<2; i++)
        {
            for (int j = 0; j<2; j++)
                cin >> x[i][j];
        }

        int& operator()(int i, int j)
        {
            return x[i][j];
        }
    };

int main()
{
    Matrix m;
    m.read();
    cout << "Element is: " << m(1, 1);
}

```

```
Enter 2x2 matrix elements : 1 2 3 4
```

```
Element is : 4
```

سربارگذاری عملگر دسترسی به اعضای یک کلاس اعضای یک کلاس را می‌توانیم با سربارگذاری عملگر -> کنترل کنیم. این عملگر یک عملگر unary است که فقط یک شیء به را به عنوان عملوند دارد. این متد سربارگذاری شده باید یک متد non-static باشد که ساختار آن را در زیر مشاهده می‌کنید:

```

ClassName * operator ->(void)
{
}

```

برنامه زیر سربارگذاری عملگر دسترسی به اعضای یک کلاس (->) را نشان می‌دهد:

```

#include <iostream>
using namespace std;

class Number
{
public:
    int x;
}

```

```

        Number(int x)
        {
            this->x = x;
        }

        Number * operator ->()
        {
            return this;
        }
    };

    int main()
    {
        Number n1(30);
        cout << "x = " << n1->x;
    }

```

x = 30

لیست عملگرهایی که قابلیت سربارگذاری را دارند در زیر آمده است.

+	-	*	/	%	^
&		~	!	,	=
<	>	<=	>=	++	-
<<	>>	==	!=	&&	
+=	-=	/=	%=	^=	&=
=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

لیست عملگرهایی که قابلیت سربارگذاری را ندارند در زیر آمده است.

::	.*	.	?:
----	----	---	----

مدیریت استثناءها و خطایابی

بهترین برنامه نویسان در هنگام برنامه نویسی با خطاها و باگ‌ها در برنامه‌شان مواجه می‌شوند. درصد زیادی از برنامه‌ها هنگام تست برنامه با خطا مواجه می‌شوند. بهتر است برای از بین بردن یا به حداقل رساندن این خطاها، به کاربر در مورد دلایل به وجود آمدن آنها اخطار داده شود. خوشبختانه C++ برای این مشکل راه حلی ارائه داده است. این زبان دارای مجموعه‌ای از کلاسهای است که برای برطرف کردن خطاهای خاص از آنها استفاده می‌کند.

استثناها در C++ راهی برای نشان دادن دلیل وقوع خطا در هنگام اجرای برنامه است. استثناها توسط برنامه به وجود می آیند و شما لازم است که آنها را اداره کنید. به عنوان مثال در دنیای کامپیوتر یک عدد صحیح هرگز نمی تواند بر صفر تقسیم شود. اگر بخواهید این کار را انجام دهید (یک عدد صحیح را بر صفر تقسیم کنید)، با خطا مواجه می شوید. اگر یک برنامه در C++ با چنین خطایی مواجه شود پیغام خطای "DivideByZeroException" نشان داده می شود که بدین معنا است که عدد را نمی توان بر صفر تقسیم کرد.

باگ (Bug) اصطلاحاً خطا یا کدی است که رفتارهای ناخواسته ای در برنامه ایجاد می کند. خطایابی فرایند برطرف کردن باگها است، بدین معنی که خطاها را از برنامه پاک کنیم. قبل از اینکه برنامه را به پایان برسانید لازم است که برنامه تان را اشکال زدایی کنید. به صورت کلی سه نوع خطا وجود دارد:

- خطاهای کامپایلری رایج ترین نوع خطا هستند. برای مثال اگر شما نوشتن سمیکالن را فراموش کنید باعث بروز خطای کامپایلری می شود.
- خطاهای منطقی زمانی اتفاق می افتند که یک اشتباهی در منطق برنامه به وجود بیاید. برای مثال در برنامه جمع دو عدد، اگر برنامه نویس به جای به علاوه، منها قرار دهد یک خطای منطقی رخ می دهد. تشخیص این نوع خطاها معمولاً کار دشواری است.
- خطاهای زمان اجرا همانطور که نام آنها پیداست در زمان اجرا اتفاق می افتند. به این نوع خطاها استثناء (Exception) نیز گفته می شود. برای مثال شما از کاربر می خواهید یک عدد صحیح وارد کند ولی یک عدد اعشاری یا یک رشته وارد می کند که باعث بروز خطا می شود.

استثناها دو نوع هستند:

همگام (Synchronous)

ناهمگام (Asynchronous)

استثناهای همگام آن دسته از استثناها می باشند که به دلیل مشکل در کد نوشته شده توسط برنامه نویس رخ می دهند و برنامه نویس می تواند آنها را مدیریت کند. اما استثناهای ناهمگام آن دسته از استثناها می باشند که به دلیل مشکلی خارج از کدهای برنامه رخ می دهند. برای مثال اگر با کمبود حافظه در RAM رو به رو شویم خطایی که رخ می دهد از نوع ناهمگام است و خطاهای ناهمگام را نمی توان مدیریت کرد. در درس های آینده در مورد روش های خطایابی برنامه بیشتر توضیح می دهیم.

دستورات try و catch

می‌توان خطاها را با استفاده از دستور try...catch اداره کرد. بدین صورت که کدی را که احتمال می‌دهید ایجاد خطا کند در داخل بلوک try قرار می‌دهید. بلوک catch هم شامل کدهایی است که وقتی اجرا می‌شوند که برنامه با خطا مواجه شود. تعریف ساده‌ی این دو بلوک به این صورت است که بلوک try سعی می‌کند که دستورات را اجرا کند و اگر در بین دستورات خطایی وجود داشته باشد برنامه دستورات مربوط به بخش catch را اجرا می‌کند. در کل ساختار دستور try catch به صورت زیر است :

```
try
{
    //some code
}
```

کدهای داخل بلوک catch زمانی اجرا می‌شوند که یک خطایی در بدنه try به وجود بیاید. ساختار این بلوک به صورت زیر می‌باشد :

```
catch (Exception - type)
{
    //some code
}
```

از عبارت throw برای به وجود آوردن یک استثنا استفاده می‌شود تا به وسیله آن خطاهای برنامه را مدیریت کنیم. ساختار استفاده از عبارت throw به صورت زیر می‌باشد :

```
throw exception;
```

برنامه زیر نحوه استفاده از دستور try...catch را نمایش می‌دهد :

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int x = 5;
7      int y = 0;
8
9      try
10     {
11         if (y == 0)
12             throw y;
13         else
14             x / y;
15     }
16     catch (int y)
17     {
```

```
18     cout << "An attempt to divide by 0 was detected.";
19 }
20 }
```

```
An attempt to divide by 0 was detected.
```

در کد بالا ما قصد داریم خطاهای احتمالی را که در عملیات تقسیم ممکن است به وجود آید را اداره کنیم. در ریاضیات عمل تقسیم عدد بر صفر ممکن نیست. در خطوط 6 و 7 دو متغیر تعریف کرده ایم که مقدار یکی از آنها 5 و دیگری 0 است. چون ممکن است که عمل تقسیم بر صفر اتفاق افتد پس در قسمت try در خط 11 با استفاده از دستور if چک می کنیم که اگر مقدار متغیر y برابر 0 باشد، استثناء ایجاد (خط 12) در غیر این صورت، عملیات تقسیم انجام شود. از آنجاییکه مقدار متغیر y عدد 0 است، یک استثناء رخ می دهد. در نتیجه برنامه از قسمت try به قسمت catch رفته و در این بلوک، یک پیغام دلخواه و قابل فهم برای نمایش به کاربر می نویسیم (خط 18). اگر فکر می کنید که در بلوک try ممکن است با چندین خطا مواجه شوید می توانید از چندین بلوک catch استفاده نمایید :

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

int main()
{
    int x, y;
    cout << "Enter x and y values: ";
    cin >> x >> y;

    try
    {
        if (y == 0)
            throw y;
        else if (y < 0)
            throw "y cannot be negative";
        else
            cout << "Result of x/y = " << (x / y);
    }
    catch (int y)
    {
        cout << "y cannot be zero";
    }
    catch (const char* message)
    {
        cout << message;
    }
}
```

```
Enter x and y values: 10 0
y cannot be zero
```

```
Enter x and y values: 10 -5
y cannot be negative
```

در برنامه بالا ما می‌خواهیم عدد صحیح x را بر y تقسیم کنیم. همانطور که می‌دانید تقسیم بر صفر باعث بروز خطا می‌شود. در این مثال ما x را 10 و y را 0 در نظر گرفتیم. زمانی که در اولین شرط به دلیل اینکه b مقدار صفر را دارد یک خطا از نوع `int` را `throw` می‌کند. از آنجایی که اولین بلوک `catch` از نوع `int` می‌باشد این خطا را همین بلوک مدیریت می‌کند. در یک مثال دیگر در برنامه بالا ما مقدار x را 10 و مقدار y را -5 در نظر گرفتیم. در اینجا بدنه شرط دوم که منفی بودن y را بررسی می‌کند اجرا می‌شود. در داخل بدنه این شرط، ما یک رشته را `throw` کردیم. از آن جایی که اولین بلوک `catch` از نوع `int` می‌باشد و نمی‌تواند یک رشته را بپذیرد بنابراین کنترل را به بلوک بعدی منتقل می‌کند و چون بلوک `catch` دوم یک آرایه از کاراکترها را می‌پذیرد بنابراین می‌تواند این خطا را مدیریت کند.

حال فرض کنید شما می‌خواهید تمام خطاهای احتمالی که ممکن است در داخل بلوک `try` اتفاق می‌افتند را فهمیده و اداره کنید این کار چگونه امکانپذیر است؟ به راحتی می‌توان از یک بلوک `catch` عمومی برای مدیریت هر نوع خطایی استفاده کرد. ساختار بلوک `catch` عمومی به صورت زیر می‌باشد:

```
catch (...)\n{\n    //some code\n}
```

بهتر است زمانی که از بلوک‌های `catch` چند تایی استفاده می‌کنیم، یک بلوک `catch` عمومی نیز قرار دهیم تا اگر خطایی به غیر از خطاهایی که ما تشخیص دادیم به وجود آمد برنامه دچار مشکل نشود. برنامه زیر نحوه استفاده از یک بلوک `catch` عمومی را نشان می‌دهد:

```
#include <iostream>\nusing namespace std;\n\nint main()\n{\n    int x, y;\n    cout << "Enter x and y values: ";\n    cin >> x >> y;\n\n    try\n    {\n        if (y == 0)\n            throw y;\n        else if (y < 0)\n            throw "y cannot be negative";\n        else\n            cout << "Result of x/y = " << (x / y);\n    }\n    catch (int y)\n    {\n        cout << "y cannot be zero";\n    }\n}
```

```
catch (...)\n{\n    cout << "Unkown exception in program";\n}\n}
```

```
Enter a and b values: 10 -5\nUnkown exception in program
```

با استفاده از این روش دیگر لازم نیست نگران اتفاق خطاهای احتمالی باشید چون بلوک catch برای هرگونه خطایی که در داخل بلوک try تشخیص داده شود پیغام مناسبی نشان می‌دهد.

راه‌اندازی مجدد استثناء

در C++ اگر یک متد یا یک بلوک try-catch داخلی (تو در تو) نخواهد خطایی را مدیریت کند، مدیریت آن را به بلوک try-catch بالایی پاس می‌دهد. ساختار Re-throw کردن یا راه‌اندازی مجدد استثناء به صورت زیر می‌باشد:

```
throw;
```

برنامه زیر نحوه re-throw کردن یک استثنا و مدیریت آن در بلوک try بالاتر را نشان می‌دهد:

```
#include <iostream>\nusing namespace std;\n\nint main()\n{\n    int x, y;\n    cout << "Enter x and y values: "; \n    cin >> x >> y;\n    try\n    {\n        try\n        {\n            if (y == 0)\n                throw y;\n            else if (y < 0)\n                throw "y cannot be negative";\n            else\n                cout << "Result of x/y = " << (x / y);\n        }\n        catch (int y)\n        {\n            cout << "y cannot be zero";\n        }\n        catch (...)\n    }\n}
```



```

        {
            throw;
        }
    }
    catch (const char* message)
    {
        cout << message;
    }
}

```

```

Enter x and y values : 10 -2
y cannot be negative

```

در برنامه بالا مشاهده می‌کنید که خطای به وجود آمده در بلوک try داخلی توسط بلوک catch عمومی داخلی گرفته شد و این بلوک آن را با استفاده از re-throw به بلوک try-catch بالایی پاس داد.

Throw کردن استثناها در تعریف توابع

در هنگام اعلان یک متد می‌توانیم استثناهایی که ممکن است اتفاق بیفتد را throw کنیم. ساختار کلی آن به صورت زیر می‌باشد:

```

return-type function-name(params-list) throw(type1, type2, ...)
{
    // some code
    ...
}

```

برنامه زیر نحوه throw کردن یک استثنا در هنگام تعریف یک متد را نشان می‌دهد:

```

#include <iostream>
using namespace std;

void sum() throw(int)
{
    int x, y;
    cout << "Enter x and y values: ";
    cin >> x >> y;
    if (x == 0 || y == 0)
        throw 1;
    else
        cout << "Sum is: " << (x + y);
}

int main()
{
    try
    {
        sum();
    }
}

```

```
    }  
    catch (int)  
    {  
        cout << "x or y cannot be zero";  
    }  
}
```

```
Enter x and b values: 5 0  
x or y cannot be zero
```

در برنامه بالا متد sum می‌تواند یک استثنا از نوع int را throw کند. بنابراین زمانی که در جایی این متد فراخوانی شود، باید یک بلوک catch از نوع int برای مدیریت این استثنا وجود داشته باشد.

Throw کردن استثناهایی از نوع کلاس

در بخش‌های قبل ما از انواع داده اولیه نظیر int، float، char و ... برای throw کردن یک استثناء استفاده کردیم. اما می‌توانیم یک کلاس ایجاد کنیم و یک استثنا از نوع آن کلاس را throw کنیم. استفاده از کلاس‌های خالی به خصوص در مدیریت استثناءها کاربرد زیادی دارند. برنامه زیر نشان می‌دهد که چگونه می‌توانیم یک استثنا از نوع کلاس را throw کنیم:

```
#include <iostream>  
using namespace std;  
  
class ZeroError {};  
  
void sum()  
{  
    int x, y;  
    cout << "Enter x and y values: ";  
    cin >> x >> y;  
    if (x == 0 || y == 0)  
        throw ZeroError();  
    else  
        cout << "Sum is: " << (x + y);  
}  
  
int main()  
{  
    try  
    {  
        sum();  
    }  
    catch (ZeroError e)  
    {  
        cout << "x or y cannot be zero";  
    }  
}
```

```
}
```

```
Enter x and y values: 0 8
x or y cannot be zero
```

در برنامه بالا ZeroError یک کلاس خالی است که برای مدیریت استثنا ساخته شده است.

مدیریت خطا و وراثت

همانطور که در بخش‌های قبلی نیز بررسی کردیم، زمانی که چند بلوک catch داریم، ترتیب اجرا به این صورت است که ابتدا اولین بلوک catch بررسی می‌کند که آیا نوع خطایی که throw شده را می‌تواند بپذیرد یا خیر؟ اگر بتواند، آن را مدیریت می‌کند و در غیر اینصورت کنترل به بلوک بعدی منتقل می‌شود. در مباحث وراثتی زمانی که ما دو بلوک catch داریم که یکی از نوع کلاس base یا پدر و دیگری از نوع کلاس Derived یا فرزند باشد، بهتر است بلوک catch ای که از نوع کلاس پدر است را بعد از بلوک catch ای که از نوع کلاس فرزند است قرار دهیم. به مثال زیر توجه کنید:

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

class Base
{
};

class Derived : public Base
{
};

int main()
{
    try
    {
        throw Derived();
    }
    catch (Base b)
    {
        cout << "Base object caught";
    }
    catch (Derived d)
    {
        cout << "Derived object caught";
    }
}
```

Base object caught

در برنامه بالا ما یک کلاس به نام base داریم که همان کلاس والد یا پدر است و یک کلاس به نام Derived داریم که همان کلاس فرزند است. در متد main ما یک خطا از نوع کلاس Derived را throw کردیم. همانطور که گفته شد ابتدا بلوک catch اولی بررسی می‌کند که آیا می‌تواند خطا را مدیریت کند یا خیر. از آنجایی که با توجه به مباحث وراثتی می‌توانیم یک نمونه از کلاس فرزند را در یک نمونه از کلاس پدر قرار دهیم. بنابراین بلوک catch اول، خطا را می‌گیرد. در صورتی که ما یک بلوک catch از نوع کلاس Derived یا فرزند نیز داریم و هدف این بود که بلوک catch دوم خطا را مدیریت کند. برای جلوگیری از این نوع مشکلات بهتر است بلوک catch کلاس والد را بعد از بلوک catch کلاس فرزند قرار دهیم. به برنامه زیر توجه کنید:

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

class Base
{
};

class Derived : public Base
{
};

int main()
{
    try
    {
        throw Derived();
    }
    catch (Derived d)
    {
        cout << "Derived object caught";
    }
    catch (Base b)
    {
        cout << "Base object caught";
    }
}
```

Derived object caught

استثناها در متدهای سازنده (constructor) و مخرب (destructor)

ممکن است که خطایی در یک متد سازنده یا مخرب اتفاق بیفتد. اگر یک استثنا در متد سازنده به وجود بیاید، می‌تواند باعث نشت حافظه شود. فرض کنید ما دو متغیر در متد سازنده تعریف کردیم، اگر به هر دلیلی در تخصیص حافظه به یکی از آنها

خطایی به وجود بیاید باعث بروز مشکل در حافظه می‌شود و ممکن است باعث متوقف شدن ناگهانی برنامه شود. همچنین این مشکل در متد مخرب نیز وجود دارد. بنابراین بهتر است یک مدیریت خطای مناسب را داخل متدهای سازنده و مخرب انجام دهیم تا از بروز مشکلات بعدی جلوگیری کنیم. برنامه زیر نحوه مدیریت یک استثنا در متد سازنده و مخرب را نشان می‌دهد:

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

class Divide
{
private:
    int *x;
    int *y;

public:
    Divide()
    {
        x = new int();
        y = new int();
        cout << "Enter two numbers: ";
        cin >> *x >> *y;
        try
        {
            if (*y == 0)
            {
                throw *x;
            }
        }
        catch (int)
        {
            delete x;
            delete y;
            cout << "Second number cannot be zero!" << endl;
            throw;
        }
    }

    ~Divide()
    {
        try
        {
            delete x;
            delete y;
        }
        catch (...)
        {
            cout << "Error while deallocating memory" << endl;
        }
    }

    float division()
    {
        return (float)*x / *y;
    }
};
```

```
int main()
{
    try
    {
        Divide d;
        float res = d.division();
        cout << "Result of division is: " << res;
    }
    catch (...)
    {
        cout << "Unkown exception!" << endl;
    }
}
```

```
Enter two numbers: 5 0
Second number cannot be zero!
Unkown exception!
```

آموزش های بیشتر در سایت w3-farsi.com