

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ

تقدیم به:

مادر، پدر و خواهرم

آموزش C++

ویرایش سوم

مؤلف:

سید ایمان علوی فاضل

عنوان و پدیدآور	: علوی فاضل، سید ایمان - ۱۳۷۷	سرشناسه
وضعیت ویراست	: آموزش C++ / تالیف سید ایمان علوی فاضل	
مشخصات نشر	: ویراست ۳	
مشخصات ظاهری	: اهواز: تراوا، ۱۳۹۷	
شابک	: ۹۷۸-۳۴۷-۶۰۰-۹۷۸	
وضعیت فهرست نویسی	: فیبا	
موضوع	: سی++ (زبان برنامه‌نویسی کامپیوتر)	C (computer program language)++
موضوع	: زبان‌های برنامه‌نویسی کامپیوتر	Programming languages (Electronic computers)
رده بندی کنگره	: QA76/۷۳ / ۱۳۹۷ ۷۵۹۳، س	
رده بندی دیوبی	: ۱۳۳/۰۰۵	
شماره کتابشناسی ملی	: ۵۲۹۰۲۹۵	

نام کتاب: آموزش C++

مؤلف: سید ایمان علوی فاضل

ویراستار ادبی: محمد مستغان

طرح روی جلد: مشتاق مودت

ناشر: تراوا

شماره نشر: ۵۱۴

نوبت چاپ: سوم (ویرایش سوم)/ ۱۳۹۷

تیراز: ۱۰۰۰ نسخه

شابک: ۹۷۸-۶۰۰-۳۴۷-۳۳۷-۹

نشر تراوا: اهواز- کیانپارس- خیابان نهم غربی- پلاک ۱۲۸

نمبر: ۰۹۱۶۱۱۳۶۷۸۵- ۰۶۱۳۳۹۰۳۷۱۴

Taravapublication@yahoo.com

www.tarava.com

فروشگاه اینترنتی



@taravapub

حق چاپ و نشر مخصوص ناشر است.

فهرست

صفحه	عنوان
۷	پیشگفتار مؤلف
۹	مقدمه
۱۳	اولین برنامه
۱۷	متغیر و انواع داده (۱)
۲۶	اپراتورها (۱)
۴۴	ساختارهای شرطی
۵۲	ساختارهای تکرار
۶۱	تابع (۱)
۷۳	متغیر و انواع داده (۲)
۸۰	آرایه
۸۷	متغیر و انواع داده (۳)
۹۹	اپراتورها (۲)
۱۰۱	کار با حافظه (۱)
۱۲۲	رشته
۱۲۸	کلاس (۱) (کلاس‌ها و اشیا)
۱۳۸	کلاس (۲) (کپسول کردن و لیست انتساب دهندهی عضو)
۱۴۸	کلاس (۳) (ادامه‌ی مباحث پیهای کلاسها)
۱۵۹	کلاس (۴) (وراثت، اعضای friend و سربارگذاری اپراتور)
۱۷۴	کلاس (۵) (چندريختگی)
۱۸۹	کلاس (۶) (توابع عضو خاص)
۲۰۳	واژه‌ی static
۲۰۹	چند فایل کردن کد منبع
۲۲۵	namespace
۲۳۲	پیشپردازنده

۲۳۹	قالب تابع
۲۵۲	قالب کلاس و متغیر
۲۶۶	<code>std::map</code> و <code>std::vector</code> , <code>std::array</code>
۲۸۳	کار با فایل
۲۸۸	تابع (۲)
۳۱۱	استثناء
۳۲۷	کار با حافظه (۲) (اشارهگرهای هوشمند)
۳۴۳	تولید اعداد تصادفی
۳۵۳	ضمیمه (۱) کمپایلر برای C++
۳۵۸	ضمیمه (۲) کمپایل با C++14, C++11 و C++17
۳۵۹	ضمیمه (۳) گرفتن آرگومان از cli
۳۶۱	ضمیمه (۴) نوع بازگشتی کوواریانت (covariant)
۳۶۲	ضمیمه (۵) Most vexing parse در C++
۳۶۵	ضمیمه (6) دسته بندی مقادیر

پیشگفتار مؤلف

در این کتاب اصطلاحاتی که بین برنامه نویس های فارسی زبان آشنا هستند، مانند اشارهگر، شی و ... به فارسی ترجمه شده‌اند، و واژگانی همچون namespace به همان صورت انگلیسی در متن کتاب قرار گرفتند. با تغییراتی که در استانداردهای جدید C++ افزوده شده، توضیح خیلی از مطالب به یکدیگر وابسته شده‌اند؛ در این کتاب سعی شده این مفاهیم به بهترین ترتیب ممکن در کتاب بیان شوند.

با وجود اینکه کتابهای خوبی به فارسی و انگلیسی منتشر شده است، در این کتاب سعی شده تفاوت‌هایی با برخی از آن‌ها داشته باشد؛ از جمله عدم قرار دادن مثالی دشوار بعد از توضیح یک بخش. در کتاب حاضر به دو دلیل از مثالهای ساده و کوتاه استفاده شده است؛ تا اولاً جنبه‌ی آموزشی آن مبحث از نظر نهوه‌ی استفاده از ساختار مربوطه شیوآتر بیان شود و ثانیاً خواندن و تایپ کدهای کوتاه از روی کتاب آسانتر باشد. برنامه‌های کاربردی و طولانی، در اکانت [github](#) قرار خواهند گرفت و بروزرسانی خواهند شد.

همچنین در این کتاب سعی شده است که ایده‌های کلی از C++ به خواننده منتقل شود. در صورت نیاز به جزئیات کامل، کتاب [The C++ Programming Language](#) تالیف بیارنه استراستروپ - مختصر C++ - توصیه می‌شود. در پایان از پدر و مادرم که همیشه راهنمای مناند کمال امتحان را دارم،

برای کد و برنامه‌های نوشته شده به [github](#) زیر مراجعه کنید:
github.com/alavifazel

مقدمه

ابتدا با چند مفهوم کلی در برنامه نویسی آشنا می‌شویم:

(Machine Language)

زبان ماشین

به دستورالعمل هایی که مستقیم بر روی پردازنده (CPU) اجرا می‌شوند، زبان ماشین گفته می‌شود. این دستورات متشکل از اعداد (معمولاً اعداد باینری)‌اند و مجموعه‌ای از آنها ایجاد یک دستورالعمل می‌کند که برای کامپیوتر معنی خاصی میدهد. دستورات زبان ماشین علاوه بر دشوار بودن، مختص سخت افزار خاصی اند. به علت دشواری و وابستگی این دستورات به سخت افزار خاص، از این زبان به طور مستقیم برای توسعهٔ نرمافزار استفاده نمی‌شود.

(Assembly language)

زبان اسembly

در زبان اسembly (یا `asm`) برای دستورالعملهای زبان ماشین، جایگزین اسمی وجود دارد؛ به طوری که در این زبان، به جای نوشتن مستقیم دستورالعملها از واژه‌ها برای کدنویسی استفاده می‌شود. لازم به ذکر است که رابطه‌ی بین دستورات اسembly و زبان ماشین لزوماً یک رابطه‌ی یک به یک نیست. این زبان، کدنویسی را مقداری ساده‌تر کرد. نمونه‌ای از یک دستور اسembly:

`MOV AL`

از آنجایی که پردازنده فقط قادر به اجرای زبان ماشین است، دستورات اسembly در نهایت باید توسط برنامه‌ای به نام/اسembler¹ به زبان ماشین تبدیل شوند. زبان اسembly یک زبان سطح پایین است به این معنا که برنامه‌های نوشته شده در این زبان، به سخت افزار نزدیکاند. نرمافزارهای سطح پایین با سرعت زیادی اجرا می‌شوند و دسترسی مستقیم به

¹. Assemblers

منابع سیستم را دارند. کدنویسی در این زبان علاوه بر جنبه‌های آموزشی، در برنامه‌هایی که نیاز به دسترسی مستقیم به سختافزار است هنوز کاربرد دارد.

(High level languages)

زبان‌های سطح بالا

زبان‌های سطح بالا مانند C++ با قرار دادن سطح زیادی از انتزاع (Abstraction)، کدنویسی را آسانتر کرده است. با استفاده از زبانهای سطح بالا، مقدار زیادی از کارکرد سختافزار از برنامه نویس مخفی می‌ماند. اکثر نرم افزارهای مدرن نیز با این زبان‌ها نوشته می‌شوند. به دلیل آنکه CPU فقط قادر به اجرای زبان ماشین است، کدهای نوشته شده در این زبان‌ها نیز نهایتاً باید به طریقی به زبان ماشین تبدیل شوند. این زبان‌ها معمولاً با یکی از دو نوع نرم افزار زیر (یا ترکیبی از این دو) پیاده می‌شوند:

۱. کمپایلر^۱

۲. مفسر^۲

۱. کمپایلر:

کدهای نوشته شده در زبان‌هایی مانند C و C++ معمولاً توسط برنامه‌ای به اسم کمپایلر به زبان ماشین تبدیل می‌شوند. به تبدیل کدهای این زبان‌ها به زبان ماشین توسط کمپایلر، کمپایل می‌گویند. کمپایلر تمام کدهای منبع را یکجا کمپایل کرده و یک خروجی نهایی تولید می‌کند. این خروجی سیپس، برای هر بار اجرا نیاز به کمپایل مجدد ندارد. همچنین کمپایلر، برنامه را برای یک نوع پردازنده و سیستم عامل خاص کمپایل می‌کند. مثلاً برنامه‌های که برای لینوکس ۶۴ بیت کمپایل شده، قابل اجرا بر روی لینوکس ۳۲ بیت نیست. برای اجرا، برنامه باید برای لینوکس ۳۲ بیت جداگانه کمپایل شود.

۲. مفسر:

زبان‌های سطح بالایی مانند Ruby از مفسر به جای کمپایلر استفاده می‌کنند. مفسر

¹. Compiler

². Interpreter

کدها را خط به خط و در حین اجرا به زبان ماشین تبدیل می‌کند. به همین دلیل برنامه‌هایی که از مفسر استفاده می‌کنند معمولاً کندر از برنامه‌های کمپایلی هستند. مفسر توانایی بهینه سازی، مدیریت خودکار منابع برنامه و ... را ساده تر از کمپایلر میتواند پیاده کند. همچنین مفسر قابلیت حمل (portability) برنامه را افزایش می‌دهد؛ زیرا کدها، بدون (یا با مقدار کمی) تغییر بر روی هر سیستمی که مفسر آن زبان نصب باشد قابل اجرا هستند.

نیاز به ذکر هست که کمپایلر و مفسر، ابزارهایی برای پیادهسازی زبانهای برنامه نویسی هستند و خود زبان برنامه نویسی، مفسری یا کمپایلی نیست؛ با این وجود به دلیل اینکه پیاده سازیهای بعضی از زبانها مثل C++ معمولاً با کمپایلر صورت میگیرد، به C++ زبان کمپایلی میگویند.

زبان C++

C++ (سی پلاس پلاس) یک زبان همه منظوره‌ی برنامه نویسی است که در سال ۱۹۷۹ توسط بیارنه استروسترود^۱ برای بهبود زبان C طراحی شد. این زبان در ابتدا C همراه کلاس نام داشت که در سال ۱۹۸۳ به C++ تغییر کرد. این زبان از قابلیت‌های سطح بالایی همچون شی گرایی^۲، برنامه نویسی جنریک^۳، تابعی^۴ و علاوه بر آن از قابلیت‌های سطح پایین مانند کار با حافظه^۵ نیز برخوردار است. این ویژگی‌ها، C++ را به یک زبان معروف و تجاری تبدیل کرده است. برنامه‌های نوشته شده در این زبان با استانداردهای اخیر از امنیت بالایی نیز برخوردارند. با اصطلاحات نام برده شده در طول کتاب آشنا می‌شویم.

^۱. Bjarne Stroustrup

^۲. Object-oriented programming

^۳. Generic programming

^۴. Functional programming

^۵. Memory management

اولین برنامه

برای کدنویسی به زبان C++ نخست، نیاز به یک کمپایلر است. با داشتن کمپایلر از هر ویرایشگر متنی برای نوشتن و ویرایش دستورات، مثل emacs، vim یا Notepad، می‌توان استفاده کرد. با این وجود نرم افزارهایی تحت عنوان محیط یکپارچه برنامه نویسی یا IDE¹ - که اکثراً رایگان هستند - وجود دارند که نوشتن، غلط‌گیری و کمپایل کد منبع را آسانتر می‌کنند. IDE‌های Visual Studio، codeblocks و qtcreator جز معروف ترین آن‌هاست. هدف این نرم افزارها یکسان و انتخاب آن‌ها سلیقه‌ای می‌باشد. برای راهنمایی نصب کمپایلر و IDE به ضمیمه‌ی ۱ مراجعه کنید.

هدف در این قسمت، معرفی برنامه ایست که یک عدد صحیح را در خروجی نمایش می‌دهد. دستورات زیر را تایپ کنید. سپس به بررسی جدالگانه‌ی هر خط می‌پردازیم.

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

int main()
{
    std::cout << 4;
    system("pause");
    return 0;
}
```

با کمپایل و اجرای برنامه، خروجی زیر تولید می‌شود:

4

خروجی این برنامه در کنسول است؛ این محیط در لینوکس و macOS تحت نرم افزاری‌هایی مانند terminal و در ویندوز cmd قابل مشاهده است.

اکنون به توضیح هر خط می‌پردازیم:

Visual Studio #در صورت عدم استفاده از **#include "stdafx.h"** این

¹1. Intergrated Development Environment

دستور را حذف کنید (در پایان این بخش بیشتر توضیح داده شده).
#include <iostream> دستور **#include** محتوای فایلی که بعد از آن قرار میگیرد را به برنامه میافزاید. فایل‌هایی که با این دستور به برنامه افزوده میشوند، هدر^۱ نام دارند. هدر **iostream** که در این خط به برنامه افزوده میشود، یک سری قابلیتها را برای ورودی و خروجی کنسول در دسترس قرار میدهد. از این قابلیتها جهت چاپ یک عدد صحیح در ادامه برنامه استفاده میشود.

خط 3: یک خط خالی (blank) است و توسط کمپایلر نادیده گرفته میشود. این خط برای خوانایی بهتر منظور گردیده است.

int main(): این خط تابعی به نام **main** تعریف میکند. تابع **main** قسمتی از برنامه است که با اجرا شدن آن، سیستم به دنبال و اجرای دستورات این تابع میپردازد. خطوط 5 و 9: آکولاد باز و بسته اند که دستورات تابع **main** در میان آنها قرار میگیرند. به دستوراتی که بین {} و بعد از نام تابع میگیرند بدن، و در اینجا بدنی تابع **main** گفته میشود.

std::cout << 4;: در این خط از **std::cout** برای خروجی در کنسول استفاده شده. عبارت بعد از اپراتور << به خروجی ارسال میشود که در اینجا عدد 4 است.

system("pause");: در صورت استفاده از **Visual Studio**، اگر با گزینه‌ی **Local Windows Debugger** برنامه را اجرا میکنید، بعد از چاپ خروجی بلافاصله **cmd** بسته میشود. برای جلوگیری از بسته شدن **cmd**، یا برنامه را با **Ctrl + F5** اجرا کنید و یا این خط را به برنامه بیافزایید.

return 0;: این دستور اتمام برنامه را بدون خطا به سیستم اعلام میکند. در صورتی که عدد بعد از **return** غیر از صفر باشد در اکثر کمپایلرها به معنی اتمام برنامه

¹. Header

با خطاست. در صورت نوشته نشدن این دستور، کمپایلر بطور پیش فرض ۰ را در نظر میگیرد.

توجه شود که در این برنامه یک سمیکلن (;) در پایان هر دستور (غیر از دستورات #include و بدنه []) باید قرار گیرد.

در cout با اپراتور << میتوان چند عبارت را با تفکیک در خروجی چاپ کرد. مثلاً دستور زیر:

```
std::cout << 4 << 5;
```

۴۵ را در خروجی چاپ میکند.

دلیل قرار دادن std:: قبل از cout در خط ۵ این است که cout یک جز از کتابخانه استاندارد است. به جای قرار دادن std:: میتوانستیم یک بار از دستور زیر:

```
using namespace std;
```

در بدنه main یا قبل از main - برای تاثیر دادنش در تمام برنامه - استفاده کنیم. این دستور باعث میشود که امکان استفاده از اجزای کتابخانه استاندارد بدون نیاز به std:: قبل از نام آنها فراهم شود.

نکته‌ی قابل ذکر این است که علاوه بر خطهای خالی، تمام فاصله‌های دیگر مثل tab و space بین دستورات (غیر از دستور #include) نیز توسط کمپایلر نادیده گرفته میشوند و فقط جهت خوانایی قرار می‌گیرند. برای مثال، میتوانستیم کل برنامه را به صورت زیر بنویسیم:

```
#include "stdafx.h"
#include <iostream>
int main(){std::cout << 4;system("pause");return 0;}
```

توجه: فایل stdafx.h فایلی است که به طور خودکار توسط Visual Studio تولید میشود. با از پیش کمپایل آن، که خود شامل هدایت‌های استاندارد سیستم و

هدرهای پروژه است، مدت زمان کمپایل کاهش می‌یابد؛ افزودن هدر "stdafx.h" و خط `(system("pause"))` در مثال‌های بعدی قرار نخواهد گرفت. لذا در صورت استفاده از **Visual Studio**، این دستورات را برای هر مثال بنویسید.
در این فصل هدف، آشنایی اولیه با C++ بود. در فصل آینده با ذخیره‌ی مقادیر در حافظه آشنا می‌شویم که امکان نوشتن برنامه‌های کاربردی‌تر را فراهم می‌کند.

متغیر و انواع داده (۱)

در C++ برای ذخیره‌ی یک مقدار در حافظه، نوع داده^۱ی آن باید مشخص باشد. در این فصل با اکثر انواع داده‌های اساسی برای ذخیره‌ی اعداد صحیح، اعشاری، بولین^۲ و ... آشنا می‌شویم.

پنج نوع داده‌ای را که در ایجا بررسی می‌کنیم عبارتند از:

- برای ذخیره سازی کاراکترها مانند 'a'.

• int برای اعداد صحیح.

• float برای اعداد اعشاری.

• double برای اعداد اعشاری بزرگ‌تر از float و با دقت اعشار بیشتر.

• bool برای مقادیر صحیح و غلط (false و true).

حجمی که یک کمپایلر برای انواع داده‌های عددی اختصاص میدهد یکسان نیست. در اکثر سیستم‌ها برای ذخیره‌سازی int چهار بایت و float شانزده بایت اشغال می‌شود. بازه‌های که با انواع عددی می‌توان ذخیره کرد براساس حجمی که اشغال می‌کنند متفاوت است. در جدول زیر مقادیر مجاز پنج نوع معرفی شده نوشته شده است (بازه‌های عددی برابر با حداقل مقدار گارانتی شده در استاندارد C++ هستند):

مقدار	نوع
۱۲۷+ تا ۱۲۷-	char
۳۶۷۶۷+ تا ۳۲۷۶۷-	int
حدود ۱۰ ^{-۳۸} تا ۱۰ ^{۳۸} (با ۷ رقم دقت)	float
حدود ۱۰ ^{-۳۰۸} تا ۱۰ ^{۳۰۸} (با ۱۵ رقم دقت)	double

^۱. Data type

^۲ مقادیر بولین می‌توانند فقط مقدار true و false را اختیار کنند.

false و true	bool
---------------------	-------------

برای اطلاع از بازه دقیق انواع عددی بر روی سیستم خود میتوانید از `<limits>` از هدر `std::numeric_limit` استفاده کنید.
(برای مثال)

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

int main(){
    cout << numeric_limits<double>::lowest() << ' '
        << numeric_limits<double>::max();
}
```

کمترین و بیشترین مقدار `double` را نمایش میدهد.¹

برای ذخیره‌ی یک مقدار و استفاده از آن در طول برنامه، دانستن انواع داده‌های ذکر شده کافی نیست بلکه باید متغیر² از یک نوع داده اعلان شود. متغیر برای فضایی از حافظه با نوع داده‌ی مشخص، یک نام اختصاص می‌دهد. با نام متغیر سپس مقداردهی، گرفتن مقدار و تغییر مقدار روی حافظه متغیر میتواند صورت گیرد. فرمت اعلان یک متغیر به صورت زیر است:

```
datatype id;  
نام متغیری است که اعلان میشود.
```

برای مثال با:

```
int a;  
یک متغیر به نام a از نوع int اعلان میشود.
```

¹ استفاده از تمام قابلیتهای `numeric_limits` C++11 نیازمند به بعد است. برای آشنایی با نحوه کمپایل با استاندارد های جدید، ضمیمه‌ی ۲ را مطالعه کنید.

². Variable

مثال ۱) برنامه‌ای که شامل یک متغیر از نوع `int` برای ذخیره‌ی یک عدد صحیح است.

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

int main() {
    int a = 4;
    cout << "a's value = " << a << '\n';
}
```

خروجی برنامه به صورت زیر است:

```
a's value = 4
```

توضیح) در دستور `cout`, کاراکتر '`\n`' یک کاراکتر خاص است که با قرارگیری آن، یک خط جدید چاپ می‌شود.

برای مقداردهی اولیه‌ی متغیر، از ۴ روش زیر می‌توان استفاده کرد:

```
int a = 3;
int a(3);
int a{3};
int a = {3};
```

دستورات فوق تاثیر یکسانی دارند و هرکدام یک متغیر از نوع `int` اعلان و آن را با ۳ مقداردهی اولیه می‌کند. به اسمی که برای عناصر تعریفی می‌گذاریم شناسه (identifier) گفته می‌شود. در دستورات بالا، `a` یک شناسه است.

متغیر پس از اعلان، جداگانه می‌تواند با اپراتور `=` (اپراتور انتساب) مقداردهی شود. برای مثال:

```
int a;
a = 4;
```

نوع داده‌ی یک متغیر مشخص می‌کند چه فرآیندهایی روی آن می‌توان انجام داد. در استانداردهای جدید اگر متغیر با اعلان مقداردهی اولیه شود، به جای نوشتن نوع داده

آن، از واژه‌ی **auto** میتوان استفاده کرد. برای مثال با دستور زیر:

```
auto i = 5;
```

کمپایلر نوع **i** را براساس نوع **5** (int) اعلام و آن را با **5** مقداردهی اولیه میکند. برای تعریف چند متغیر از یک نوع داده می‌توان چند شناسه را با کاما (,) تفکیک داد و نوشت. مثلاً با:

```
int a = 4, b = 2;
```

دو متغیر به نامهای **a** و **b** از نوع **int** ایجاد و مقداردهی اولیه می‌شوند. برخلاف **cout** که تاکنون استفاده کردیم، با استفاده از **cin** می‌توان یک سری ورودی را از طریق کنسول از کاربر دریافت کرد. با **cin** مقادیر مورد نیاز جهت مقداردهی متغیرها را نیز در حین اجرا میتوان از کاربر گرفت. مثال ۲) برنامه‌ای که شامل یک متغیر است که مقدارش در حین اجرا و با **cin** دریافت می‌شود.

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

int main() {
    int x;
    cout << "Enter a value:" ;
    cin >> x;
    cout << "You entered:" << x << '\n';
}
```

دستورات را کمپایل و اجرا کنید تا به ازای یک ورودی مثل **7**، خروجی زیر تولید شود:

```
Enter a value:7
You have entered:7
```

توضیح) مقدار وارد شده از طریق **cin** با اپراتور **<>** (برخلاف **<<** در **cout**) باید در متغیر ذخیره شود.

با اپراتور **<>** چند ورودی را با تفکیک می‌توان از کاربر دریافت کرد. کاربر یا قرار دادن

فاصله مثلاً با space و enter می‌تواند ورودی‌ها را تفکیک داده شده به برنامه ارسال کند.

مثال ۳) برنامه‌ای که ۳ عدد صحیح int را از ورودی می‌گیرد و آن‌ها را در خروجی نمایش می‌دهد:

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

int main() {
    int a, b, c;
    cout << "Enter 3 values and press enter: ";
    cin >> a >> b >> c;
    cout << "You have entered: " << a << " " << b << "
" << c << '\n';
}
```

خروجی برنامه به صورت زیر است:

```
Enter 3 values and press enter: 2 5 4
You have entered: 2 5 4
```

به نوع داده‌ی int، یک سری پیشوند برای علامتدار یا بی علامت بودن و تعیین سایز آن می‌تواند افزوده شود؛ که عبارت‌اند از:

.۱ unsigned و signed برای تعیین علامت،

.۲ long و short برای تعیین سایز.

انواع ایجاد شده با پیشوندهای فوق حداقل بازه‌های زیر را ایجاد می‌کنند (با اعلان int نوع آن به طور پیش فرض از نوع signed قرار می‌گیرد):

حداقل بازه	نوع
۳۶۷۶۷- تا ۳۲۷۶۷+	short int
۰ تا ۶۵۵۳۵	unsigned short int

۰ تا ۶۵۵۳۵	<code>unsigned int</code>
$\pm 2.14 \cdot 10^9$	<code>long int</code>
۰ تا ۱۰۹۴۲۹	<code>unsigned long int</code>
$\pm 9.22 \cdot 10^{18}$	<code>long long int</code>
۰ تا $10^{19} \cdot 1.84$	<code>unsigned long long int</code>

همانگونه که در جدول دیده میشود از هر نوع پیشوند (علامتدار بودن و سایز) فقط یکی از آنها را میتوان نوشت. اگر پیشوندها نیز به تنها یکی نوشته شوند، به معنای پیشوند به علاوه‌ی `int` است؛ برای مثال:

`long i = 10;`

نماینده از نوع `long int` اعلان میکند.

به عنوان یک قاعده‌ی کلی برای ذخیره اعداد صحیح از `int` استفاده کنید مگر اینکه نیاز به ذخیره‌ی اعداد بزرگ شود که در آن صورت `long long` و `long` گزینه‌های مناسب‌بیاند. همچنین مقادیر `unsigned` و `signed` هم باید با هم ترکیب شوند زیرا نتایج غیرقابل پیش‌بینی ایجاد میکنند.

بازه‌های عددی ذکر شده در جدول، حداقل گارانتی شده در استاندارد C++ را نشان میدهند و کمپایلرهای مختلف میتوانند بازه‌های بیشتری را تخصیص دهند. در صورت نیاز به سایز و بازه‌های ثابت و جلوگیری از محدودیت برنامه به کمپایلر خاص، از نوعها موجود در هدر `cstdint` میتوانید استفاده کنید. انواع موجود در این هدر شامل انواعی با سایز ثابت و غیرثابت است. برای مثال:

```
#include <cstdint>
```

```
...
int16_t i = 10;
```

متغیر `i` را با مقدار ثابت ۴ بایت اعلان میکند که عدد صحیح بین ± 32767 را در آن میتوان ذخیره کرد.

غیر از انواع ذکر شده، انواع اساسی دیگری نیز وجود دارد که در ادامه با آنها آشنا

میشوند.

عبارت و دستور در C++

دستور (statement) قسمتی از کدهای برنامه است که عملی را انجام می‌دهد و معمولاً با ; نیز به پایان میرسد. برای مثال خط زیر یک دستور است:

```
int a = 1;
```

عبارت (expression) در حالت ساده بیانگر یک مقدار در برنامه است. مثل:

5

اگر یک عبارت (expression) با یک علامت ; به پایان برسد نیز یک دستور (statement) حساب میشود.

در پایان این بخش سه نکته بررسی می‌شود:

۱. در برنامه از // جهت توضیح استفاده میشود و کمپایلر، کلاراکترهای مقابلش را تا انتهای خط نادیده می‌گیرد. برنامه نویسها برای خود و دیگران این توضیحات را قرار می‌دهند. برای مثال:

```
cout << 1 << '\n'; // prints 1
```

به جای استفاده از //، می‌توان یک سری توضیح را بین /* و */ قرار داد. مثال:

```
/*
This is a comment.
And will be ignored by compiler...
*/
```

۲. همچنین چهار نکته را در مورد شناسه‌ها باید در نظر گرفت:

۱. C++ به بزرگی و کوچکی حروف حساس است و این شامل شناسه‌ها نیز

می‌شود.

۲. نام شناسه می‌تواند متشكل از حرف، عدد و underscore (_) باشد.

۳. نام شناسه نمی‌تواند با یک عدد شروع شود.
۴. نام شناسه نمی‌تواند همنام با واژگان از پیش تعریف شده برای کمپایلر باشد.
- این واژه‌ها در آخرین استانداردهای C++ شامل واژه‌های زیر است:

<code>reinterpret_cast</code>	<code>dynamic_cast</code>	
<code>return</code>	<code>else</code>	<code>alignas</code>
<code>short</code>	<code>enum</code>	<code>alignof</code>
<code>signed</code>	<code>explicit</code>	<code>and</code>
<code>sizeof</code>	<code>export</code>	<code>and_eq</code>
<code>static</code>	<code>extern</code>	<code>asm</code>
<code>static_assert</code>	<code>false</code>	<code>auto</code>
<code>static_cast</code>	<code>float</code>	<code>bitand</code>
<code>struct</code>	<code>for</code>	<code>bitor</code>
<code>switch</code>	<code>friend</code>	<code>bool</code>
<code>template</code>	<code>goto</code>	<code>break</code>
<code>this</code>	<code>if</code>	<code>case</code>
<code>thread_local</code>	<code>inline</code>	<code>catch</code>
<code>throw</code>	<code>int</code>	<code>char</code>
<code>true</code>	<code>long</code>	<code>char16_t</code>
<code>try</code>	<code>mutable</code>	<code>char32_t</code>
<code>typedef</code>	<code>namespace</code>	<code>class</code>
<code>typeid</code>	<code>new</code>	<code>compl</code>
<code>typename</code>	<code>noexcept</code>	<code>concept</code>
<code>union</code>	<code>not</code>	<code>const</code>
<code>unsigned</code>	<code>not_eq</code>	<code>constexpr</code>
<code>using</code>	<code>nullptr</code>	<code>const_cast</code>
<code>virtual</code>	<code>operator</code>	<code>continue</code>
<code>void</code>	<code>or</code>	<code>decltype</code>
<code>volatile</code>	<code>or_eq</code>	<code>default</code>
<code>wchar_t</code>	<code>private</code>	<code>delete</code>
<code>while</code>	<code>protected</code>	<code>do</code>
<code>xor</code>	<code>public</code>	<code>double</code>
<code>xor_eq</code>	<code>register</code>	

دو واژه‌ی زیر نیز فقط در مکانهای خاصی از پیش تعریف شده‌اند:

`override`
`final`

به این ترتیب اعلان متغیرهای زیر غلط است:

```
int double; //error
int 4var; //error
```

اپراتورها (۱)

با استفاده از یک اپراتور (عملگر) می‌توان عملی را بر روی مقادیر انجام داد تا نتایج جدیدی حاصل شود. به عنوان مثال با اپراتور $+$ می‌توان عبارت $4+3$ را در برنامه نوشت تا جمع آن‌ها، یعنی 7 ، برگردد. همچنین در مثال مذکور 4 و 3 هر کدام یک اپراند^۱ نام دارند. به مجموعهای از اپراتورها و اپراندهایشان، یک عبارت (expression) گفته می‌شود. در اینجا به 5 دسته از اپراتورها می‌پردازیم:

- .۱ اپراتورهای محاسباتی
- .۲ اپراتورهای افزایش/کاهش
- .۳ اپراتورهای انتساب
- .۴ اپراتورهای ربطی
- .۵ اپراتورهای منطقی

اپراتورهای محاسباتی^۲

این نوع اپراتورها یک سری محاسبات ریاضی روی اپراندها اعمال می‌کنند. اگر a و b به ترتیب اعداد 4 و 2 باشند:

اپراتور	توضیح	نتیجه
$+$	اپراندها را جمع می‌کند.	$a+b$ می‌دهد 6
$-$	اپراند بعد را از قبلی کم می‌کند.	$a-b$ می‌دهد 2
$*$	اپراندها را در هم ضرب می‌کند.	$a*b$ می‌دهد 8
$/$	اپراند قبل را بر بعد تقسیم می‌کند.	a/b می‌دهد 2
$\%$	باقي‌ماندهی تقسیم اپراند قبل بر بعد از اپراتور را برمی‌گرداند.	$a \% b$ می‌دهد 0

¹. Operand

². Arithmetic operators

مثال ۱) برنامه‌ای که ۲ عدد صحیح را از ورودی می‌گیرد و میانگین آن‌ها را حساب می‌کند و در خروجی نمایش می‌دهد:

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

int main() {
    int a, b;
    double average;
    cout << "Enter 2 values and press enter: ";
    cin >> a >> b;
    average = (a + b) / 2;
    cout << "Average = " << average << '\n';
}
```

خروجی برنامه به صورت زیر است:

```
Enter 2 values and press enter: 9 7
Average = 8
```

نکته: در خط هشتم عبارت:

`average = (a + b) / 2;`

کمپایلر مقدار راست اپراتور = (اپراتور انتساب) را حساب و با نتیجه‌ی آن، متغیر سمت چپ = را مقداردهی می‌کند. پس دستور زیر نیز صحیح است:

`a = a + 5;`

با وجود این که از نظر جبری درست نیست.

اپراتورهای افزایش/کاهش

این اپراتورها یک واحد به اپراند خود میافزایند یا از آن میکاهند. اگر مقدار متغیر ۴، باشد:

اپراتور	توضیح	نتیجه
<code>++</code>	بر روی یک اپراند اعمال می‌شود و به مقدار عددی آن یکی می‌دهد	<code>++a</code>

	می افزاید.	
--	بر روی یک اپراند اعمال می شود و از مقدار عددی آن یکی می کاهد.	-- می دهد ۳

مثال ۲) اپراتورهای افزایش و کاهش می توانند هم قبل و هم بعد از متغیر نوشته شود. در مثال زیر فرق آن ها را بررسی می کنیم:

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

int main() {
    int a = 4;
    cout << a++ << '\n';
    cout << a << '\n';

    cout << ++a << '\n';
    cout << a << '\n';
}
```

خروجی برنامه به صورت زیر است:

```
4
5
6
6
```

توضیح) همانطور که از خروجی مشاهده می شود اگر اپراتور افزایش بعد از متغیر نوشته شود مثل `a++`, اپراتور به مقدار آن یکی می افزاید ولی مقدار قبل از افزایش را برمیگرداند. اگر اپراتور افزایش، قبل از متغیر نوشته شود مثل `++a`, یکی به مقدار متغیر افزوده می شود و مقدار جدید متغیر در دسترس قرار می گیرد.

اپراتورهای انتساب^۱:

این اپراتورها برای تعیین یا تغییر مقدار پیشین یک متغیر استفاده می شوند. لیست این

^۱. Assignment operators

اپراتورها با توضیح آن‌ها در جدول زیر آمده است:

اپراتور	توضیح
=	اپراتوری که متغیر را با مقدار راست خود مقداردهی می‌کند.
+=	$a = a + b$ به معنی $a += b$ است.
-=	$a = a - b$ به معنی $a -= b$ است.
*=	$a = a * b$ به معنی $a *= b$ است.
/=	$a = a / b$ به معنی $a /= b$ است.
%=	$a = a \% b$ به معنی $a \%= b$ است.

(۱) مثال

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

int main() {
    int a = 4;
    a += 10;
    cout << a << '\n';
}
```

خروجی برنامه به صورت زیر است:

14

هنگام استفاده از این اپراتور، اپراند سمت چپ آن باید `lvalue` و قابل مقداردهی باشد. `lvalue` مقداری است که به یک شی (فضای پیوسته‌ای از حافظه) تلقی می‌شود (مثل یک متغیر); در مقابل `lvalue`, `rvalue` هر مقداری است که `lvalue` نباشد. برای

مثال:

```
int a = 0, b = 3;
a + b = 13; // error
```

عبارت $a + b$ یک مقدار موقت و از نوع `rvalue` است در نتیجه مقداردهی آن بیمعنی است.

اپراتورهای ربطی^۱

این اپراتورها روابط ریاضی مساوی، بزرگتری، کوچکتری و ... دو اپراند را چک میکنند و بر میگردانند. اگر a و b به ترتیب اعداد 4 و 2 باشند، در جدول زیر نتایج آنها بررسی میشود:

اپراتور	توضیح	نتیجه
<code>==</code>	برابری مقادیر اپراندها را چک میکند در صورت برابر <code>false</code> بودن، <code>true</code> و در غیر این صورت	$a == b$ میدهد <code>false</code>
<code>!=</code>	برابری مقادیر اپراندها را چک میکند در صورت برابر <code>true</code> نبودن، <code>false</code> و در غیر این صورت	$a \neq b$ میدهد <code>true</code>
<code>></code>	در صورت بزرگتر بودن اپراند چپ از راست، <code>true</code> و در غیر این صورت <code>false</code>	$a > b$ میدهد <code>true</code>
<code><</code>	در صورت کوچکتر و مساوی بودن اپراند چپ از راست <code>false</code> و در غیر این صورت <code>true</code>	$a < b$ میدهد <code>false</code>
<code>>=</code>	در صورت بزرگتر و یا مساوی بودن اپراند چپ از راست <code>false</code> و در غیر این صورت <code>true</code>	$a \geq b$ میدهد <code>true</code>
<code><=</code>	در صورت کوچکتر و یا مساوی بودن اپراند چپ از راست <code>true</code> و در غیر این صورت <code>false</code>	$a \leq b$ میدهد <code>false</code>

با کاربرد این اپراتور در آینده آشنا خواهیم شد. با این حال در مثال زیر نتایج این

^۱. Relational operators

اپراتورها نشان داده شده:

(۱) مثال

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

int main() {
    int a = 4, b = 10, c = 4;
    cout << (a < b) << '\n';
    cout << (a == c) << '\n';
    cout << (b < c) << '\n';
}
```

خروجی برنامه به صورت زیر است:

```
1
1
0
```

از نظر عددی `false` برابر صفر و `true`، یک است و `cout` نیز مقدار آنها را با `0` و `1` نمایش میدهد. برای چاپ خود عبارت `false` یا `true`، باید فرمت `std::boolalpha` برای `std::cout` تعیین شود.

برای مثال:

```
cout << boolalpha;
cout << false << '\n'; // false
cout << true << '\n'; // true
برای لغو کردن این فرمت، باید فرمت std::noboolalpha را تعیین کرد. مثل:
cout << noboolalpha;
```

^۱ اپراتورهای منطقی

این اپراتور روی ۲ اپراند بولین (و یا هر اپراندی که به بولین تبدیل شود) اعمال میشود و `true` یا `false` را بر میگرداند. اگر `b` و `a` به ترتیب `true` و `false` باشند:

^۱. Logical operators

اپراتور	توضیح	نتیجه
&&	به معنی <u>و</u> که اگر هر دو اپراند true بود true را برミگرداند.	(a && b) false میدهد
	به معنی <u>یا</u> که اگر حداقل یکی از دو اپراند true بود true را برミگرداند.	(a b) true میدهد
!	عبارت بولین را برعکس می‌کند. مثلاً اگر عبارت بود آن را به false تغییر می‌دهد.	a! false میدهد

مثال ۱) برنامه‌ای که شامل اپراتورهای منطقی است:

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

int main() {
    int a = 4, b = 10, c = 20;
    cout << ((a<b) && (b<c)) << '\n';
    cout << ((a<b) || (b>c)) << '\n';
}
```

خروجی برنامه به صورت زیر است:

```
1
1
```

توضیح) در دستور چاپ اول به دلیل اینکه هر دو عبارت درون پرانتز true و برنامه، ۱ را در خروجی چاپ کرد. در چاپ بعدی چون یکی از آنها (اولی) صحیح است، کل عبارت صحیح می‌شود و دوباره ۱ در خروجی چاپ می‌شود. استفاده از پرانتز () تقدم عبارت درون خود را بالا می‌برد. برای مثال:

$(2 + 4) * 5;$

مانند تعیین اولویت در ریاضی، اول درون پرانتز حساب سپس نتیجه‌اش در ۵ ضرب

می شود.

به اپراتورهایی که روی یک اپراند اعمال می شوند مانند `++`, اپراتور یگانی (unary) و به اپراتورهایی که روی دو اپراند اعمال می شوند، مثل `+`, اپراتور باینری (binary) گفته می شود.

اولویت بعضی از اپراتورها از بالا به پایین به صورت زیر است:

واستگی	توضیح	اپراتور
چپ به راست	پسوند افزایش/کاهش	<code>--a</code> و <code>++a</code>
راست به چپ	پیشوند افزایش/کاهش	<code>a--</code> و <code>a++</code>
راست به چپ	اپراتور یگانی جمع و تفریق	<code>a-</code> و <code>a+</code>
راست به چپ	اپراتور منطقی	<code>!</code>
چپ به راست	ضرب، تقسیم و باقیمانده	<code>a%b</code> و <code>a/b</code> و <code>a*b</code>
چپ به راست	جمع و تفریق	<code>a-b</code> و <code>a+b</code>
چپ به راست	اپراتورهای منطقی	<code>=></code> و <code>></code> <code>=<</code> و <code><</code>
چپ به راست	اپراتورهای منطقی	<code>!=</code> و <code>==</code>
چپ به راست	اپراتورهای منطقی	<code>&&</code>
چپ به راست	اپراتورهای منطقی	<code> </code>
راست به چپ	اپراتور انتساب	<code>=</code>
راست به چپ	اپراتورهای انتساب	<code>-=</code> و <code>=+</code>
راست به چپ	اپراتورهای انتساب	<code>=/.</code> و <code>=/*</code>

ثوابت^۱

ثوابت در برنامه عباراتی هستند که مقدارشان ثابت است و نمی‌تواند تغییر کند. لیترال‌ها ثوابتی هستند که برای کمپایلر از پیش تعریف شده‌اند. برای مثال وقتی می‌نویسیم:

```
int a = 5;
```

عدد 5 یک لیترال است. از نظر برنامه نویسی چون 5 یک ثابت است دستور زیر صحیح نیست:

```
5 = 3;
```

لیترال‌ها به چند دسته تقسیم می‌شوند.

۱. اعداد صحیح^۲

در مثال قسمت قبل، 5 یک لیترال عدد صحیح است. در C++ علاوه بر اعداد صحیح دهدھی، اعداد اکتال، هگزادسیمال و باینری نیز از نوع لیترال عدد صحیح‌اند. به طوری که عدد اکتال با یک 0، هگزادسیمال با 0X (یا 0x) و باینری با 0b (یا 0B) قبل از ارقامش مشخص می‌شود. برای مثال، ۴ عبارت:

```
25  
031  
0x19  
0b00011001
```

با هم برابر و از نوع لیترال عدد صحیح‌اند.

با قرار دادن یک سری پسوند به لیترال عدد صحیح، می‌توان نوع لیترال را از یک نوع داده‌های خاص تعیین کرد. پسوندها به صورت زیراند:

پسوند	تغییر دهنده‌ی نوع
U یا u	unsigned

¹. Constants

². Integer numerals

L یا l	long
LL یا ll	long long

پسوند `unsigned` با دو پسوند دیگر می‌تواند ترکیب شود. برای مثال `ll` به معنی `unsigned long long` است. مثال:

```
55u // unsigned
55l // long
55ul // unsigned long
```

در غیر این صورت، نوع لیترال‌های عدد صحیح از نوع اولین نوع داده‌ای است که می‌تواند بر اساس لیست زیر آن را ذخیره کند (این لیست بر اساس پایه و پسوند لیترال است):

های عدد صحیح قابل تعیین استهایی که برای لیترال نوع		
پسوند	دهی‌های ده‌پایه	های باینری، اوکتال یا هگزادسیمال‌پایه
بدون پسوند	int long int long long int	int unsigned int long int unsigned long int long long int unsigned long long int
u یا U	unsigned int unsigned long int unsigned long long int	unsigned int unsigned long int unsigned long long int
l یا L	long int unsigned long int long long int	long int unsigned long int long long int unsigned long long int
l/L و u/U	unsigned long int unsigned long long int	unsigned long int unsigned long long int
ll یا LL	long long int	long long int unsigned long long int
ll/LL و u/U	unsigned long long int	unsigned long long int

به عنوان مثال لیترال عدد صحیح در خط زیر از نوع `unsigned long long` تعیین می‌شود:

```
cout << 1234567891234567891ULL << '\n'
```

در مقداردهیها، بین انواع مختلف عددی یک سری تبدیل به صورت خودکار صورت میگیرد. برای مثال در دستور زیر:

```
double d = 10;
```

d از نوع double با مقدار ۱۰ (لیترال از نوع int) مقداردهی میشود. در نتیجه در هر مقداردهی نیازی به تعیین نوع لیترال متناسب با نوع متغیر نیست.

۲. اعداد اعشاری^۱

این لیترال‌ها بیانگر اعداد اعشاری در برنامه هستند که با یک نقطه (.) اعشارشان تعیین میشود. علاوه بر اعشار، با استفاده از e (به معنی عدد ضرب در ۱۰ به توان X و X عددی است که بعد از e قرار می‌گیرد) نیز میتوان مقدار آن را تعیین کرد. برای مثال:

```
1.42353 // 1.42353
```

:۹

```
2.4e11 // 2.4 * 10^11
```

دو لیترال قابل قبولاند.

مقادیر این نوع لیترال از نوع double است ولی مانند لیترال‌های عدد صحیح، در صورت نیاز می‌توانند توسط کمپایلر به انواع داده‌ی دیگر تبدیل شوند. به عنوان مثال:

```
float a = 4.2;
```

۴.۲ از float به double تبدیل میشود.

با استفاده از پسوندهای زیر می‌توان نوع داده را به طور خاص به نوع داده‌های float یا long double تغییر داد.

نوع داده	پسوند
----------	-------

^۱. Floating point numerals

float	F یا f
long double	L یا l

با قرار دادن پیشوند X یا 0X نیز میتوان لیترالهای عدد اعشاری را از نوع هگزادسیمال تعیین کرد.

در لیترالهای عددی به طور کلی، با استفاده از کتیشن تکی ' میتوان بین ارقام فاصله گذاشت تا باعث خوانایی بهتر شود. مثل:

```
int a = 1'000'000'000;
float b = 1'000.5;
```

۳. کاراکتر و رشته^۱

کاراکتر یک حرف، عدد یا سمبل قابل قبول در انکوding^۲ است. یک کاراکتر در دو تک کتیشن '' قرار می‌گیرد و نوع داده‌ی آن char است. مانند:

'x'

مثال ۱) برنامه‌ای که متغیر آن با کاراکتر مقداردهی اولیه میشود و سپس مقدار آن در خروجی نمایش مییابد:

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

int main() {
    char a = 'w';
    cout << a << '\n';
}
```

خروجی برنامه به صورت زیر است:

w

¹. Character and string

². Encoding

به مجموعه‌ای از کاراکترها رشته گفته می‌شود. یک رشته در کوتیشن دو تایی "" قرار می‌گیرد. مثل:

"this is a sentence."

یک سری کاراکترهای خاص با \ (بک اسلش) شروع می‌شوند و معنی خاصی برای کمپایلر دارند. لیست و کاربرد این کاراکترها در جدول زیر آمده:

کاراکتر	توضیح
\n	افروzen خط جدید
\r	رفتن به اول خط
\t	تب
\v	تب عمودی
\b	بک سپیس
\f	فرم فید
\a	به صدا درآوردن صدای سیستم
\'	کتیشن تکی
\"	کتیشن دوتایی
\?	علامت سوال
\\	بک اسلش

این کاراکترها escape sequence نام دارند. برای مثال:
cout << "This \v can \t be \n anything\n";

خروجی زیر را تولید می‌کند:

```
This
    can    be
anything
```

دون رشته اگر یکی از کاراکترهای escape sequence به یک کاراکتر دیگر بچسبد کمپایلر اولین کاراکتر بعد از \ را به حساب می‌آورد. برای مثال:

```
cout << "This is a single\ttab" << '\n';
```

خروجی زیر را تولید می‌کند:

This is a single	tab
------------------	-----

اگر بین چند رشته یک کاراکتر خالی مثل space، tab یا خط جدید قرار گیرد، کمپایلر رشته‌ها را به هم می‌چسباند و یک رشته‌ی بزرگ‌تر ایجاد می‌کند. مثلاً:

```
cout << "a string" " another string" << '\n';
```

خروجی زیر را دارد:

a string	another string
----------	----------------

برای ادامه دادن یک رشته و نوشتن ادامه‌ی آن در خط بعد، از یک \ در انتهای خط استفاده می‌شود. مثلاً:

```
"This is a \
single string"
```

برابر است با:

```
"This is a single string"
```

لیترالهای رشته میتوانند پیشوند U، u، L یا u8 بگیرند که به ترتیب رشته را با انکوڈینگ wide utf-8، utf-16، utf-32 یا از نوع رشته wide تعیین میکنند. برای مثال:

```
cout << u8"فارسی" << '\n' ;
```

یک رشته‌ی فارسی را چاپ می‌کند.

کاراکترهای رشته‌ی wide از نوع wchat_t است که بسته به پیاده سازیهای مختلف معنی خاص میدهد.

به طور مشابه لیترال‌های کاراکتر نیز پیشوندهای L، u، u8، wide یا میگیرند که به معنای انکودينگ 4، utf-8، ucs-2، ucs-4 است. علاوه بر لیترال‌های بولین^۱ نیز متشکل از true و false اند که به مقدار صحیح یا غلط تلقی می‌شود.

فرق std::endl با '\n'

در بعضی از کتابها و کدهای C++ برای افزودن خط جدید در کنسول، به جای استفاده از کاراکتر '\n'، از std::endl استفاده می‌شود. برای مثال به جای دستور زیر:

```
cout << 4 << '\n';
```

از دستور زیر استفاده می‌کنند:

```
cout << 4 << endl;
```

تفاوت endl با کاراکتر '\n' در این است که در حالت عادی خروجی cout لزوماً در هر بار نوشته شدن به خروجی ارسال نمی‌شود. این عملکرد cout برای افزایش سرعت ورودی و خروجی صورت می‌گیرد؛ به طوری که پس از رسیدن حجم بافر به یک حد معین، سپس مرحله‌ی ارسال حافظه به خروجی انجام می‌شود. endl علاوه بر قراردادن کاراکتر '\n'، بافر cout را نیز بلافصله به خروجی ارسال می‌کند (اصطلاحاً flush می‌کند). در مواقعي که نیاز به flush کردن بلافصله در cout نیست، استفاده از '\n' مقداری سریعتر از endl خواهد بود. نیاز به flush در هر دستور cout، برای مثال زمانی ایجاد می‌شود که یک برنامه به خروجی لحظه‌ای برنامه‌ی ما وابسته است.

استفاده از endl در دستور زیر:

```
cout << 4 << endl;
```

مشابه دستور زیر اجرا می‌شود:

```
cout << 4 << '\n' << flush;
```

¹. Boolean (bool)

اعلان متغیر ثابت

اگر در تعریف یک متغیر قبل یا بعد از نوع داده‌ی آن واژه‌ی `constexpr` قرار گیرد، متغیر از نوع متغیر ثابت تعریف می‌شود. برای مثال:

```
constexpr int a = 4;
```

سپس با سعی به تغییر مقدار آن، مانند دستور زیر:

```
a = 2;
```

خطا صادر خواهد شد.

متغیرهای ثابت را میتوان به قسمتهای مختلف برنامه پاس داد و اطمینان داشت که مقدار آنها عوض نمی‌شود. برای مثال برای ذخیره کردن یک عدد ثابت مثل `pi` (3.14) در یک متغیر، به دلیل اینکه تغییر یافتن این مقدار بیمعنی است، آن را در یک متغیر `constexpr` ذخیره می‌کنند.

متغیر تعریف شده از نوع `constexpr` میتواند به عنوان عبارت ثابت (`constant expression`) مثل لیترال‌ها استفاده شود. عبارت ثابت عبارتی است که در حین کمپایل ارزیابی می‌شود و میتواند در مکانهایی که نیاز به این نوع عبارتها است (مثل پارامتر قالبها¹) استفاده شود.

برای اعلان یک متغیر از نوع ثابت علاوه بر واژه‌ی `const`، از واژه‌ی `constexpr` نیز میتوان استفاده کرد. مقداردهی اولیه‌ی متغیرهای `const`، میتواند لزوماً با مقادیر مشخص در حین کمپایل صورت نگیرد. مانند:

```
int i;  
cin >> i;  
constexpr int s = i;
```

ولی اگر با مقادیر مشخص در حین کمپایل مقداردهی اولیه شود، مانند یک متغیر اعلان خواهد شد (در حین کمپایل ارزیابی می‌شود). مثل اعلان متغیر

¹ بررسی در فصل قالبها

زیر:

const int i = 4;

ساختارهای شرطی

^۱if ساختار شرطی

ساختار شرطی **if** به برنامه این امکان را می‌دهد تا در صورت درست (**true**) بودن یک عبارت بولین، یک سری دستور را اجرا و اختیاراً در صورت غلط (**false**) بودن یک سری دستور دیگر را اجرا کند.

فرمت ایجاد ساختار شرطی **if** به صورت زیر است:

```
format 1
if(expr) {
    // statement(s)
}
```

عبارتی است که **expr** **true** یا **false** بودنش مورد بررسی قرار می‌گیرد. **statement(s)** نیز دستور یا دستوراتی است که در صورت درست بودن **expr** اجرا می‌شوند.

(۱) مثال

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

int main() {
    int x;
    cout << "Enter an integer: ";
    cin >> x;
    if(x<0) {
        cout << "x is negative.\n";
    }
}
```

خروجی آن به ازای ورود عدد - 8 چنین می‌باشد:

```
Enter an integer: -8
```

^۱. Decision making structure

```
x is negative.
```

توضیح) این برنامه یک عدد صحیح از کاربر می‌گیرد و با ساختار شرطی بررسی می‌کند آیا منفی است یا خیر. در صورت منفی بودن، یک متنی را در خروجی نمایش میدهد. اگر (if) statement(s) فقط یک دستور باشد، می‌توان آکولاد باز و بسته {} را برای قرار نداد. مثل:

```
if (x<0)
    cout<<"x is negative.\n";
```

در ساختار شرطی برای اجرای یک سری دستور در صورت غلط (false) بودن شرط if، از دستور else استفاده می‌شود. با فرمت:

```
format 2
if(expr){
    // statement(s)
} else {
    // statement(s)
}
```

مثال ۲) تکمیل مثال ۱ به صورت زیر است:

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

int main() {
    int x;
    cout << "Enter an integer: ";
    cin >> x;
    if(x<0)
        cout << "x is negative.\n";
    else
        cout << "x is positive.\n";
}
```

خروجی برنامه به صورت زیر است:

```
Enter an integer: 5
x is positive.
```

با استفاده از ساختار `if else if` در صورت برقرار نبودن شرط `if`, یکسری دستورات دیگر را با برقراری شرط دیگری میتوان اجرا کرد.

```
format 3
if (expr1) {
    // statement(s)1
} else if (expr2) {
    // statement(s)2
}
```

اجرای این دستورات به این صورت است که: ابتدا `expr1` بررسی میشود؛ در صورت صحیح بودن، `statement(s)1` اجرا میگردد. در صورت عدم برقراری `expr1`، شرط `expr2` بررسی خواهد شد و در صورت صحت، `statement(s)2` به اجرا در میآید.

در صورت برقرار نبودن هیچ یک از شرطها، هیچ دستوری اجرا نخواهد شد.

همچنین به تعداد نامحدودی میتوان `if else if` قرار داد ولی شرط `else` فقط یکبار میتواند نوشته شود. مثال کامل شدهٔ مثال ۲ به صورت زیر است:

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

int main() {
    int x;
    cout << "Enter an integer: ";
    cin >> x;
    if (x<0)
        cout << "x is negative.\n";
    else if (x>0)
        cout << "x is positive.\n";
    else
        cout << "x is zero.\n";
}
```

نکته (۱): همانطور که در بخش اپراتورها دیدیم، برای بررسی مساوی بودن دو عبارت از اپراتور `==` باید استفاده شود (اپراتور `=` اپراتور انتساب است).
مثال:

```
int x = 3;
if(x==3)
    cout << "x is 3\n";
else
    cout << "x is not 3\n";
```

نکته (۲): با استفاده از اپراتورهای منطقی (`logical`) می‌توان چند عبارت را در پرانتز `if` بررسی کرد. با استفاده از این اپراتورها نیز در نهایت یک `true` یا `false` در شرط `if` ایجاد و بررسی می‌شود. مثال:

```
int x = 3;
if (x > 5 && x < 10)
    cout << "x's value is between 5 and 10.\n";
```

نکته (۳): در `if`، اگر از متغیر `float` در عبارت شرط استفاده کنیم، مثل:

```
float x = 1.3;
if(x == 1.3){
    // statement(s)
}
```

قسمت `statement(s)` در این دستورات اجرا نخواهد شد. این عملکرد به دلیل نحوی مقداردهی اولیه‌ی `x` است؛ به این صورت که در مقداردهی اولیه‌ی `x` (دستور `float x = 1.3`) از نوع `float` به `double` تبدیل می‌شود. در نتیجه مقدار متغیر `x` (۱.۳) از نوع `float` با ۱.۳ از نوع `double` مقایسه می‌شود و مقدار `false` بر می‌گردد (زیرا در حافظه با مقادیر متفاوتی ذخیره می‌شوند). برای حل این مشکل، متغیر `x` در ساختار `if` باید با لیترال ۱.۳ از نوع `float` (با قرار دادن پسوند `f`) مقایسه شود.

برای مثال:

```
float x = 1.3;
if(x == 1.3f){
    // statement(s)
}
```

در این صورت (s) statement اجرا خواهد شد.

ساختار شرطی موجود دیگر علاوه بر if، switch است. این ساختار برای بررسی یک عبارت به ازای تعداد مشخصی از مقادیر ثابت استفاده می‌شود. با فرمت:

```
format 4
switch(expr){
    case constant_expr:
        // statement(s)
        break;           //Optional
    case constant_expr:
        // statement(s)
        break;           //Optional

    // you can have any number of case statements.
    default :           //Optional
        // statement(s)
```

یک عبارت ثابت است. از break برای خروج از switch و پایان دادن به بررسی expr استفاده می‌شود. مثال:

```
switch(1) {
    case 1:
        cout << '1';
        break;
    case 2:
        cout << '2';
        break;
}
```

مقدار 1 در خروجی نمایش می‌یابد.

در حالت ساده، عبارت مورد ارزیابی `switch` باید عدد صحیح و یا قابل تبدیل به عدد صحیح باشد. همچنین در بدنه `switch` میتوان به تعداد مورد نیاز `case` - تا وقتی که مقدار آنها مجزا باشد - قرار داد. با بررسی تمام `case`ها، دستور `default` (در صورت وجود) زمانی اجرا میشود که عبارت با هیچ `case`ای منطبق نباشد.

مثال ۳) این مثال پس از اجرا، دو عدد صحیح را به عنوان ورودی و به دنبال آن یکی از اپراتورهای جمع، تفریق یا ضرب را دریافت میکند. در آخر نتیجه‌ی اعمال اپراتور دریافت شده بر روی اپراندها نمایش داده میشود. توجه شود `char` در C و C++ قابل تبدیل به مقدار عدد صحیح است؛ به همین دلیل این نوع داده در ساختار `switch` قابل استفاده است:

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

int main() {
    char o;
    int num1, num2;
    cout << "Enter two integers: ";
    cin >> num1 >> num2;
    cout << "Specify an operator + - /*: ";
    cin >> o;
    switch(o) {
        case '+':
            cout << num1 << "+" << num2 << "=" << num1+num2;
            break;
        case '-':
            cout << num1 << "-" << num2 << "=" << num1-num2;
            break;
        case '*':
            cout << num1 << "*" << num2 << "=" << num1*num2;
            break;
        default:
            cout << "Error! operator is not supported\n";
    }
}
```

```

        break;
    }
}

```

خروجی برنامه به صورت زیر است:

```

Enter two integers: 4      6
Specify an operator +/-/*: +
4+6=10
Enter two integers: 4      6
Specify an operator +/-/*: *
4*6=24

```

با نوشته نشدن `break`, برنامه پس از تطبیق عبارت `switch` با اولین `case` تمام `case` های دیگر را نیز اجرا میکند:

```

int i = 2;
switch(i) {
    case 1: cout << "1";
    case 2: cout << "2";    // starts here
    case 3: cout << "3";
}

```

خروجی ۲۳ را خواهد داشت.

لازم به ذکر است که ساختار `if-else` با `switch` قابل پیادهسازی است. مثلا زیر با هم برابراند: `if-else` و `switch`

```

switch(x) {
    case 1 : //...
    case 2 : //...
}

if(x==1) //...
else if(x==2) //...

```

ولی علاوه بر اینکه `switch` ساختار واضحتری دارد، کمپایلرها معمولاً کدهای بهینهتری برای آن تولید میکند.

در C++17 ساختارهای `if` و `switch` میتوانند شامل اعلان متغیر و یا یک عبارت قبل از بررسی شرط نیز باشند. مثل:

```
if(int x = 10; x < 3) { /* ... */ }
```

در ادامه به بررسی این ساختار و کاربردهای آن میپردازیم.

ساختارهای تکرار^۱

با ساختار تکرار می‌توان یک یا چند دستور را بیش از یک بار اجرا کرد.

۳ نوع ساختار تکرار در C++ وجود دارد:

۱. تکرار **while**

۲. تکرار **do...while**

۳. تکرار **for**

و در آخر با **goto** نیز آشنا خواهیم شد.

تکرار **while**

با استفاده از این دستور می‌توان یک یا مجموعه‌ای از دستورات را تا زمانی که یک عبارت بولی مقدار صحیح (true) دارد اجرا کرد.

فرمت آن به صورت زیر است:

```
format{1}
while(expr) {
    // statement(s)
}
```

مثال ۱) برنامه‌ای که شامل یک ساختار شرطی while است:

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

int main() {
    int a = 0;
    while(a < 10) {
        cout << "A ";
        ++a;
    }
}
```

خروجی برنامه به صورت زیر است:

^۱. Loop structures

```
A A A A A A A A A A
```

توضیح) متغیری به نام `a` اعلان و با `0` مقداردهی میشود. در این برنامه تا وقتی که `a` کمتر از `20` است رشته‌ای در خروجی نمایش میباید.

نکته: به متغیرهای مانند متغیر `a` در مثال بالا، که برای شمارش استفاده میشوند شمارنده^۱ می‌گویند.

در شرط `while` مانند ساختار شرطی از اپراتورهای منطقی نیز میتوان استفاده کرد.
مثل:

```
while(a < 40 || b < 45) {
    // ...
}
```

تکرار `do...while`

فرمت آن به صورت زیر است:

```
format{2}
do {
    // statement(s);
} while(expr);
```

این حلقه برخلاف حلقه‌ی `while`, `expr` را در انتهای حلقه چک می‌کند. تا زمانی که شرط برقرار باشد اجرای دستورات تداوم میباید و در غیر این صورت برنامه از حلقه خارج می‌شود.

مثال (۲)

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

int main () {
    int a = 0;
    do {
        cout << a << '\n';
    }
```

¹. Counter

```

    a++;
} while( a < 5 );
}

```

خروجی برنامه به صورت زیر است:

```

0
1
2
3
4

```

در صورت غلط بودن شرط while نیز دستورات do حداقل یک بار اجرا میشوند.

در ساختار های حلقه علاوه بر false شدن یک عبارت مورد ارزیابی، اگر برنامه به دستور break برسد نیز از حلقه خارج خواهد شد.

مثال (۳)

```

#include <iostream>
using namespace std;

int main () {
    int a = 0;
    while( a < 100 ){
        cout << a << ' ';
        ++a;
        if (a>5){
            cout<<"\nThe break has been executed.\n";
            break;
        }
    }
}

```

خروجی برنامه به صورت زیر است:

```

0 1 2 3 4 5
The break has been executed.

```

دستور `continue` تا حدودی همانند دستور `break` عمل می‌کند با این تفاوت که باعث خروجی برنامه از حلقه نمی‌شود ولی باعث نادیده گرفته شدن دستورات بعد از خود در حلقه می‌شود. پس از نادیده گرفته شدن بقیه‌ی دستورات، در صورت برقراری شرط، بلوک `while` دوباره اجرا می‌شود؛ در غیر این صورت حلقه به پایان خواهد رسید.

مثال ۴

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

int main () {
    int x = 0;
    while(x < 5){
        if(x==3){
            ++x;
            continue;
        }
        cout << x << ' ';
        ++x;
    }
}
```

خروجی برنامه به صورت زیر است:

0 1 2 4

به طور کلی از در تمام ساختارهای حلقه از `continue` و `break` می‌توان استفاده کرد.

مثال ۵) مثالی که شامل `while` و `if` است:

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

int main() {
    int a, b;
    bool running = true;
    while(running == true) {
```

```

cout << "enter two numbers: ";
cin >> a >> b;
if(a == 0 && b == 0) {
    cout << "exiting while loop\n";
    running = false;
} else {
    cout << "sum = " << a + b << '\n';
}
}
}

```

خروجی برنامه به صورت زیر است:

```

enter two numbers: 2 40
sum = 42
enter two numbers: 4 7
sum = 11
enter two numbers: 0 0
exiting while loop

```

حلقه‌ی **for**

فرمت تعریف **for** به صورت زیر است:

```

format{3}
for(init-statement; condition; increment/decrement)
{
    // statement(s)
}

```

قسمت مقداردهی اولیه‌ی حلقه است، **init-statement** تا زمانی که **true** است حلقه ادامه پیدا می‌کند، **condition** محل افزایش یا کاهش مقدار شمارنده است.

(۶) مثال

```

#include <iostream>
using namespace std;

```

```
int main() {
    for(int c = 0; c < 10; ++c) {
        cout << c << ' ';
    }
    cout << '\n';
}
```

توضیح) خروجی برنامه به صورت زیر است:

```
0 1 2 3 4 5 6 7 8 9
```

در قسمتهای دستور **for** با قرار دادن کاما (,), در بین عبارات می‌توان چند متغیر را در حلقه اعلان، مقدار افزود/کم کرد و یا بررسی نمود. مانند:

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

```
int main() {
    for(int c = 0, i = 10; c != i; ++c, i--) {
        cout << c << ' ' << i << '\n';
    }
}
```

خروجی زیر را تولید می‌کند:

```
0 10
1 9
2 8
3 7
4 6
```

استفاده از اپراتور **++** و **--** قبل یا بعد از متغیر در **for** تاثیری در عملکرد حلقه ندارد؛ ولی در بعضی از متغیرهای استفاده از این اپراتورها قبل از متغیر (مثل **i++**) میتواند منجر به افزایش سرعت اجرا شود. زیرا در این نوع ساختار، ابتدا یک واحد به مقدار اپراند افزوده و سپس عبارت ارزیابی می‌شود. هنگام قرارگیری این اپراتورها بعد از متغیر ابتدا یک کپی از اپراند ایجاد و پس از افزایش مقدار متغیر، کپی ایجاد شده (با مقدار قبل) برمی‌گردد. در

در صورت استفاده از یک کمپیلرهای مدرن تفاوتی بین قراردادن اپراتور قبل یا بعد از متغیر وجود ندارد و کد بهینه توسط کمپایلر ایجاد خواهد شد.

قسمت مقداردهی اولیهی **for**، به جای اعلان متغیر میتواند شامل یک عبارت (**expression**) که با ; به پایان رسیده نیز باشد. برای مثال:

```
int n = 1;
for(n += 3; n < 10; ++n) {
    cout << n << ' ';
} // 4 5 6 7 8 9
```

ساختار **for** میتواند فاقد عبارتی در اجزаш هایش باشد. برای مثال ساختار **for** زیر یک دستور صحیح:

```
for(;;){  
}
```

همچنین دستور **for** همیشه با یک دستور **while** قابل بازنویسی است. مثلاً ساختار **for** زیر را:

```
for(int i = 0; i < 10; ++i){  
    // ...  
}
```

را با **while** میتوان به صورت زیر نوشت:

```
{  
    int i = 0;  
    while(i < 10){  
        // ...  
        ++i;  
    }  
}
```

ولی ساختار **for** برای زمانی که نیاز به متغیر مثل شمارنده در حلقه داریم واضحتر است.

دستور goto

این دستور جز دستورات شرطی نیست ولی معمولاً در این ساختارها قرار می‌گیرد. این دستور باعث ارجاع بیشرط برنامه به مکان نشانه گذاری شده می‌شود. استفاده از این دستور معمولاً توصیه نمی‌شود چرا که ویرایش برنامه را برای برنامه‌نویس سخت می‌کند. در اکثر مواقع برنامه را میتوان با استفاده از دستورات دیگر به نحو بهتری کدنویسی کرد. فرمت آن به صورت زیر است:

```
format {4}
goto label;
.
.
.
label:
```

(مثال ۷)

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

int main () {
    int x = 5;
    while( x < 14 ){
        if(x == 10) {
            x = x + 1;
            goto stop;
        }
        cout << x << ' ';
        x += 1;
    }
stop:
    cout << "\njumped to stop";
}
```

خروجی برنامه به صورت زیر است:

```
5 6 7 8 9  
jumped to stop
```

تابع (۱) به عنوان واحد مستقل

تابع^۱ مجموعه‌ای از دستورات است که به عنوان واحد مستقل تعریف و سپس در برنامه اجرا (اصطلاحاً فرآخوانی) می‌شود. یک تابع می‌تواند هیچ، یک و یا چند ورودی بگیرد و مجموعه‌ای از دستورات را اجرا کند و اختیاراً یک خروجی را نیز برگرداند. پس تابع لزوماً ورودی ندارد و همچنین تابع ممکن است همانند توابع ریاضی مقداری را بازگرداند و یا اساساً فاقد مقدار بازگشتی باشد.

تعریف تابع

فرمت تعریف یک تابع به صورت زیر است.

```
returnType f(optional_parameters) {  
    // statement)s  
}
```

که در آن:

نوع مقدار بازگشتی تابع، مثل `int`, `double` و ... است. در صورتی که هیچ مقدار بازگشتی نداشته باشد تابع از نوع `void` تعریف می‌شود. `f` نام تابع و `optional_parameters` ورودی‌های تابع است. به این ورودی‌ها پارامتر^۲ گفته می‌شود و به کل پارامترها لیست پارامتر^۳ می‌گویند. لیست پارامتر، اختیاری است و می‌تواند تابع هیچ پارامتری (هیچ ورودی) نداشته باشد.

همچنین اگر نوع بازگشتی تابع `void` نباشد، تابع، مقداری را باید با دستور `return` برگرداند (برنامه هنگام اجرای تابع، زمانی که به دستور `return` بررسد مقداری مقابل آن را می‌گرداند و از تابع خارج می‌شود).

statement(s) دستوراتی است که در {} قرار می‌گیرد. به کل این دستورات بدنده‌ی

¹. Function

². Parameter

³. Parameter list

تابع^۱ گفته می‌شود.

به تعریف تابع بدون بدنه، اعلان تابع می‌گویند. مثلاً:

`returnType f(optional_parameters);`

اعلان تابع f است.

فراخوانی تابع^۲

به اجرای تابع در برنامه نویسی اصطلاحاً فراخوانی می‌گویند. فرمت فراخوانی تابع به صورت زیر است:

`f(optional_arguments);`

که در آن: f نام تابع تعریف شده و optional_arguments مقادیری است که به عنوان ورودی به تابع پاس داده می‌شود. این مقادیر آرگومان نام دارند. هنگام فراخوانی تابع، آرگومانها طبیعتاً باید به ترتیب، تعداد و نوع پارامترهای تابع نوشته شوند. پارامترهای تابع نیز با این آرگومانها مقداردهی می‌شوند.

مثال ۱) برنامه‌ای که شامل تابعی است که دو آرگومان از نوع int می‌گیرد و بزرگترین آن‌ها را بر می‌گرداند.

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

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

int main() {
```

^۱. Function's body

^۲. Function call

```

cout << max(2,3) << '\n';
cout << max(5,4) << '\n';
}

```

خروجی برنامه به صورت زیر است:

```

3
5

```

توضیح) تابع `max` در این مثال، دو آرگومان پاس داده شده را با ساختارهای شرطی بررسی میکند و بزرگترین آن‌ها را برمیگرداند.

تاکنون تمام برنامه‌هایی که نوشته‌یم شامل تابع خاصی به نام `main` بوده است. با اجرای برنامه، سیستم به دنبال این تابع و اجرا آن میپردازد. نوع مقدار بازگشتی این تابع باید از نوع `int` تعیین شود و مقدار بازگشتی (`return`) آن اگر صفر باشد به معنای به انتهای رسیدن برنامه بدون خطأ و اگر عدد غیر از صفر باشد - تقریباً در تمام سیستم - به معنی اتمام برنامه با خطأ است. همچنین در تابع `main` دیدیم که با نوشته نشدن `return 0`؛ زمانی که برنامه به انتهای این تابع برسد نیز این دستور قرار خواهد گرفت.

مثال ۲) برنامه‌ای که شامل یک تابع `void` است.

```

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

void print_int(int n) {
    cout << "passed int = " << n << '\n';
}

int main() {
    print_int(3);
    print_int(6);
    print_int(14);
}

```

خروجی برنامه به صورت زیر است:

```
passed int = 3
passed int = 6
passed int = 14
```

توضیح) تابع `print` به دلیل اینکه فقط ورودی خود را نمایش میدهد، نیازی به بازگشتن مقداری ندارد؛ به همین دلیل آنرا از نوع `void` تعریف کردیم.
توابع `void` نیز میتوانند شامل دستور `return`; بدون مقدار باشند. دستور `return` در توابع `void`، اجرای تابع را قبل از رسیدن به انتهایش متوقف میکند.
مثال (۳) تابعی از نوع `bool` که در صورت فرد بودن پارامتر، `true`، در غیر این صورت `false` برمیگرداند.

```
bool isodd(int n){
    return n % 2;
}
```

```
//...
```

```
isodd(4); // false
isodd(3); // true
```

یکی از کاربردهای توابع جلوگیری از تکرار و یا اصطلاحاً **کپی** و **پیست** کد است. با تابع یک سری دستورات را فقط یکبار مینویسیم و هر چند بار که نیاز شود آنرا در برنامه اجرا میکنیم.

مثال) در این برنامه تابعی تعریف میکنیم که دو عدد طبیعی را گرفته و اولی را به توان دومی میرساند. برای این کار میتوان از حلقه‌ی `for` استفاده کرد به طوری که شمارنده‌ی حلقه از ۰ شروع و تا به `n` (آرگومان دوم) نرسیده، مقدار آرگومان اول را در خود ضرب کند. تابع این برنامه کابردی نیست زیرا این تابع فقط برای به توان رساندن اعداد طبیعی میتواند استفاده شود و صرفاً آن را برای آشنایی بیشتر با توابع تعریف میکنیم.

مثال (۳)

```

#include <iostream>
using namespace std;

int power(int x, int n) {
    if(n < 0) {
        cout << "negative powers are not supported.\n";
        return 0;
    }
    int tmp = 1;
    for(int i = 0; i < n; ++i) {
        tmp *= x;
    }
    return tmp;
}

int main() {
    cout << power(3, 4) << '\n';
    cout << power(2, -2) << '\n';
}

```

خروجی برنامه به صورت زیر است:

```

81
negatives aren't supported.
0

```

برای محاسبه‌ی توان‌های دیگر کافی است دو مقدار دیگر به تابع پاس داده شود (بدون بازنویسی دوباره‌ی حلقه). به طور کلی برای محاسبه‌ی توان در برنامه، از تابع `pow` از هدر `<cmath>` میتوان استفاده کرد. این تابع، مانند تابعی که نوشتمیم دو آرگومان را می‌گیرد و اولی را به توان دومی میرساند؛ با این تفاوت که از آن برای به توان رساندن اعداد غیر از اعداد طبیعی و مثبت نیز میتواند استفاده شود. برای مثال:

```

#include <cmath>
using namespace std;
...
pow(3, 3);      // 27

```

```
pow(4, -2); // 0.0625
pow(3, 2.3); // 12.5135
```

پروتوتایپ تابع^۱

در C++ اجرای دستورات از خط اول شروع می‌شود و با رسیدن به `return` در تابع `main` به اتمام می‌رسد. تمام مثال‌هایی که تاکنون در مورد توابع بررسی کردیم، توابع `main` از `main` تعریف و سپس در `main` استفاده می‌شوند. اگر تابع را بعد از تابع `main` تعریف کنیم، هنگام فراخوانی در `main` پیغام خطا صادر خواهد شد؛ زیرا با وجود این که این تابع در برنامه موجودند، هنگام اجرای `main` برنامه از وجود آن‌ها مطلع نمی‌شود. برای جلوگیری از این مشکل می‌توان تابع را قبل از `main` اعلام کرد و سپس تعریف آن را بعد از `main` قرار داد. به این کار اعلان پروتوتایپ گفته می‌شود.

(مثال ۴)

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

void f();

int main() {
    f();
}

void f() {
```

```
    cout << "f()\n";
}
```

به این ترتیب، اعلان با تعریف تابع فرق دارد. اعلان فقط وجود تابع را به کمپایلر اطلاع می‌دهد و بدنی آن در جای دیگری از برنامه (حتی در یک فایل دیگر) می‌تواند قرار گیرد.

هنگام اعلان تابع می‌توان فقط نوع داده‌ی پارامترها را نوشت. مثلاً در مثال ۲ اعلان تابع

¹. Function prototype

میتواند به صورت زیر نوشته شود:

```
void printMsg(int);
```

سربارگذاری توابع^۱

در برنامه میتوان چند تابع همنام و متفاوت در تعداد یا نوع داده‌ی پارامترها تعریف کرد. در این صورت با فراخوانی، تابع مورد نظر با توجه به پارامتر(های) مناسب، اجرا می‌شود. به این عمل سربارگذاری تابع گفته می‌شود.

مثال (۵) برنامه‌ای که سربارگذاری توابع را نشان میدهد:

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

void f(int a) {
    cout << "a as int = " << a << '\n';
}

void f(double a) {
    cout << "a as double = " << a << '\n';
}

int main() {
    f(3);
    f(4.5);
}
```

خروجی برنامه به صورت زیر است:

```
a as int = 3
a as double = 4.5
```

پارامترهای پیش فرض^۲

در اعلان توابع می‌توان برای یک سری از پارامترها مقادیری پیش فرض قرار داد تا در

¹. Function overloading

². Default parameters

صورت عدم تعیین آنها هنگام فراخوانی، پارامترها با این مقادیر مقداردهی شوند.
مثال (۶)

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

void f(int a, int b = 4) {
    cout << "a = " << a << " b = " << b << '\n';
}

int main() {
    f(2,3);
    f(4);
}
```

خروجی برنامه به صورت زیر است:

```
a = 2 b = 3
a = 4 b = 4
```

توضیح) در اولین فراخوانی تابع، پارامتر دوم تابع `f` یعنی `b` با مقدار پیش فرض 4 مقداردهی میشود. همچنین زمانی که برای یک آرگومان مقدار پیش فرض تعیین میشود، برای کل آرگومانهای بعد از آن نیز باید مقادیر پیش فرض تعیین کرد؛ زیرا فراخوانی یک تابع به صورت زیر جایز نیست:

```
f(,2); // error
```

سربارگذاری توابع با وجود پارامترهای پیش فرض امکان‌پذیر است. البته نباید آرگومان‌های پیش فرض باعث از بین رفتن تفاوت در فراخوانی توابع شود. اعلان توابع و دستور فراخوانی زیر خطای کمپایلر را به همراه دارد:

```
void f(int a, int b = 3);
void f(int a);
f(1); // error
```

زیرا با پاس دادن عدد `int`، هر دو تابع قابل اجرا نند و کمپایلر از فراخوانی تابع مورد نظر اطلاع نمییابد.

در استانداردهای جدید نوع بازگشتی تابع میتواند از نوع `auto` تعیین شود؛ در این صورت نوع بازگشتی تابع بر اساس نوع مقدار `return` تعیین خواهد شد. برای مثال:

```
auto add(int a, int b){
    return a + b;
}
add(3,5); // 8
```

نوع بازگشتی تابع از نوع عبارت `a+b`، یعنی `int`، تعیین میشود.

مثال (۷) برنامهای که برای پیدا کردن و نمایش ریشه‌ی یک تابع از روش دوبخشی^۱ استفاده میکند. در این برنامه یک مثال کاربردی از توابع را بررسی میکنیم:

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

double f(double x) {
    return pow(x, 2) + x - 1;
}

void bisection(double a, double b, int m) {
    double fa = f(a), fb = f(b);
    if (fa * fb > 0) {
        cout << "[error] f(a) and f(b) have same sign.\n";
    } else {
        double interval = b - a;
        for(int n = 0; n < m; ++n) {
            interval /= 2;
            double c = a + interval;
            double fc = f(c);
            if(fa * fc < 0) {
                b = c;
                fb = fc;
            } else {
```

^۱. Bisection method

```

        a = c;
        fa = fc;
    }
    cout << "c: " << c << '\n';
}
}

int main() {
    int m = 20;
    bisection(0, 1, m);
}

```

خروجی برنامه به صورت زیر است:

```

...
c: 0.618031
c: 0.618032
c: 0.618033

```

بازگشتی^۱ در توابع

بازگشتی تکنیکی است که در آن یک تابع خود را فراخوانی می‌کند. توابعی که تاکنون نوشتیم اگر خود را فراخوانی کنند، یک حلقه‌ی بیپایان ایجاد می‌شود. برای جلوگیری از این حلقه، در تابع یک دستور باید قرار گیرد تا خروج از حلقه ممکن شود؛ این دستور حالت پایه^۲ نام دارد.

مثال ۸) برنامه‌ای که با استفاده از بازگشتی مقدار فاکتوریل عدد پاس داده شده را برمی‌گرداند:

```

#include <iostream>
using namespace std;

int factorial(int x) {

```

¹. Recursion

². Base case

```

if(x==1)
    return 1;
else
    return x*factorial(x-1);
}

int main() {
    cout << factorial(4) << '\n';
}

```

خروجی برنامه به صورت زیر است:

24

توضیح) در مثال بالا **return 1**; حالت پایه‌ی بازگشتی است.
مثال ۹) برنامه‌ای که شامل تابعی است که جمله‌های دنباله‌ی n ام فیبوناچی را حساب میکند و آن را برمی‌گرداند:

```

#include <iostream>
using namespace std;

int fibonacci(int n) {
    if(n==0)
        return 0;
    if(n==1)
        return 1;
    return fibonacci(n-1) + fibonacci(n-2);
}

int main() {
    for(int i = 0; i < 10; ++i) {
        cout << fibonacci(i) << ' ';
    }
}

```

خروجی برنامه به صورت زیر است:

0 1 1 2 3 5 8 13 21 34

متغیر و انواع داده (۲)

تبدیل explicit و implicit مقادیر

در صورتی که در برنامه عبارت `2/3` را بنویسیم، مقدار `0` برمیگردد. برای مثال خط زیر:

```
cout << 2/3;
```

را در خروجی نمایش خواهد داد. دلیل این نوع عملکرد این است که کمپایلر نوع داده‌ی `2` و `3` به طور پیش فرض از نوع `int` در نظر میگیرد و حاصل اپراتور تقسیم نیز از نوع `int` بر خواهد گشت. در این مثال خاص برای جلوگیری از این کار در این دستور میتوان یک و یا هر دو عدد را به صورت `2.0` یا `3.0` یا با پسوند `f` مانند `2f` و `3f` نوشت تا از نوع عدد اعشاری تعیین شوند؛ اما در حالت کلی مقادیر را در صورت امکان میتوان به یکدیگر تبدیل کرد. به این عمل، تبدیل مقدار به صورت `explicit` و یا گست (cast) گفته میشود. گست به یکی از دو فرمت زیر میتواند صورت گیرد:

```
(typeName) expression;  
typeName (expression);
```

روش اول از C++ به وارد شده و روش دوم مختص C++ است. در مثال `2/3` برای دریافت مقدار اعشاری، با گست کردن میتوان آنها را به صورت زیر نوشت:

```
(float)2/(float)3
```

گست میتوان بین مقادیر عددی، اشارهگرهای رفرنسها، `const` و ... صورت گیرد. استفاده از دو روش گست کردن که بررسی کردیم، آزادی زیادی به برنامه نویس میدهدن. در نتیجه یکسری تبدیلهای (خصوص بین اشارهگرهای رفرنسها) در برنامه صحیح و قابل کمپایل میشود که در اجرا ایجاد خطای نیکنند. به همین منظور از روش‌های دیگر گست کردن استفاده میشود که در اینجا `static_cast` را بررسی می‌کنیم. فرمت استفاده از `static_cast` به صورت زیر است:

```
static_cast<targetType>(expr);
```

برای مثال، تبدیل `2` از `int` به `float` به صورت زیر انجام میشود:

```
static_cast<float>(2);
```

امتیاز استفاده از `static_cast` در چک شدن تبدیلها در حین کمپایل است. به این ترتیب از خطاهای احتمالی جلوگیری خواهد شد. استفاده از این اپراتور علاوه بر تبدیل مقادیر اشاره‌گرها، برای نوع داده‌های اساسی نیز توصیه می‌شود: چون با جستجوی واژه `static_cast` می‌توان محل گستتها را در کدمنبع پیدا کرد.

همچنین بین مقادیر با انواع داده‌های مشابه، از نوع بزرگتر به کوچکتر و بلعکس یکسری تبدیلها به صورت خودکار و توسط کمپایلر صورت می‌گیرد؛ این نوع تبدیلها، تبدیل‌های `char`, `short`, (`int`) `implicit` نام دارند. برای مثال مقادیر صحیح کوچکتر از `int` (و یا `unsigned short` و ...) و یا `bool` توسط کمپایلر به `int` (و یا `unsigned short` صورت نیاز) تبدیل می‌شوند. مانند:

```
short a = 3, b = 2;
short result = a + b;
```

در خط دوم ابتدا مقادیر `a` و `b` به `int` تبدیل، سپس نتیجه‌ی حاصل از جمع آنها به `short` تبدیل خواهد شد. این نوع تبدیل جهت افزایش سرعت اجرا صورت می‌گیرد. زیرا در بعضی از سیستم‌ها و کمپایلرها کارکردن با مقادیر کوچکتر از `int` مقداری کندر از `int` است. به نوع تبدیل‌ها که از نوع داده‌ی کوچکتر به بزرگتر صورت می‌گیرد، `promotion` (ارتقا) گفته می‌شود. بین مقادیر اعشار از `float` به `double` نیز صورت می‌گیرد و چون از نوع داده‌ی کوچکتر به بزرگتر تبدیل می‌شوند، حفظ ماندن مقدار و یا اعشار آن تضمین شده است.

مثال (۱) `promotion` در مثال زیر قابل مشاهده است:

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

void printInteger(int x){
    cout << "int passed.\n";
}
```

```
void printInteger(short x){
    cout << "short passed.\n";
}

int main() {
    short a = 1, b = 2;
    printInteger(a + b);
}
```

خروجی برنامه به صورت زیر است:

```
int passed.
```

همچنین بین مقادیر عددی مختلف نیز میتواند تبدیل^۱ هایی صورت گیرد که تاکنون نیز با آن برخورد کردہایم. برای مثال وقتی مینویسیم:

```
float a = 4;
```

مقدار ۴ از **int** به **float** میشود و یا در دستور زیر:

```
int a = false;
```

مقدار **float** از **bool** به **false** تبدیل میشود و متغیر **a** با ۰ مقداردهی خواهد شد. قوانین و حالت های مختلف بین تبدیل های عددی و کلا بین تبدیل های **implicit** زیاد است و در صورت نیاز به رفرنس C++ مراجعه کنید ولی نکته ذکر این است که در تبدیل های عددی (**numeric conversion**) بر خلاف **promotion** مقدار داده با تبدیل شدن میتواند تغییر کند. برای مثال در دستور زیر:

```
int a = 1.5;
```

مقدار اعشار ۱.۵ از بین می رود و متغیر **a** با ۱ مقداردهی خواهد شد. به این نوع تبدیلهای عددی که از نوع بزرگتر به کوچکتر صورت میگیرد تبدیلهای **narrowing** گفته میشود؛ و مانند دستور بالا، احتمال تغییر مقدار آنها وجود دارد. بعضی از کمپایلرها برای تبدیلهای **narrowing** پیغام خطای صادر میکنند ولی در تمامی کمپایلرها این دستورات کمپایل میشوند (برای الزام کمپایل شدن کدهای C یا C++ قدیمی عموماً). برای جلوگیری از تغییر مقادیر در تبدیلهای عددی در هنگام مقداردهی اولیه میتوان از {} یا {}= استفاده کرد.

^۱. Numeric conversion

برای مثال:

```
int a{1.5}; // error
```

این ساختارهای مقداردهی اولیه در استانداردهای جدید افزوده شده‌اند و نسبت به `=` و `()` نیز توصیه می‌شوند.^۱.

ناحیه (scope)

هر اسمی که در برنامه اعلان می‌شود (مثل اعلان متغیر) فقط در قسمت خاصی از برنامه به نام ناحیه صحیح و قابل استفاده است. در C++ ناحیه‌ها عبارت اند از:

۱. ناحیه‌ی بلوک (block): متغیری که در یک بلوک {} مثل بدنی تابع، ساختار شرطی و ... اعلان شود، با خروج برنامه از آن بلوک غیر قابل دسترس می‌شود. توجه شود که یک بلوک میتواند صرفاً متعلق به یک تابع، ساختار شرطی و ... نباشد. برای مثال:

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

int main() {
{
    int a = 1;
    cout << a << '\n';
}
cout << a << '\n'; // error!
}
```

۲. ناحیه‌ی namespace: اسمی که در یک namespace اعلان می‌شود قابل دسترس در کل اجزای namespace و ناحیه‌ای است که این عضو با using در آن معرفی شده. با وجود اینکه namespaces را در فصلی جداگانه بررسی خواهیم کرد، در اینجا با مثالی که شامل اعلان متغیر در یک namespace است، آشنا می‌شویم.

مثال (۸)

```
#include <iostream>
```

^۱ با این وجود در این کتاب از `=` جهت مقداردهی اولیه متغیرهای `int` و ... استفاده شده؛ زیرا تمام خوانندگان عزیز لزوماً از کمپایلر جدید استفاده نمی‌کنند.

```

namespace a{
    int x;
}

using namespace a;
using namespace std;

int main() {
    cout << x << '\n';
}

```

همچنین اگر متغیر خارج از هر بلوک، تابع و ... در بالاترین سطح کد اعلان شود، متعلق به محدوده‌ی یک **namespace** به نام **global namespace** است. این **namespace** قابل دسترس در واحد ترجمه‌ی (translation unit) است که در آن تعریف می‌شود. در مثال هایی که تاکنون بررسی کردیم، واحد ترجمه‌ی کل برنامه است. به متغیرهایی که در **global namespace** اعلان شده‌اند متغیر سراسری^۱ نیز می‌گویند.

مثال ۲) برنامه‌ای که دسترسی به یک متغیر سراسری را نشان میدهد:

```

#include <iostream>
using namespace std;

int a = 10;
void print1(){ cout << a << '\n';}
void print2(){ cout << a << '\n';}

int main() {
    print1();
    print2();
}

```

خروجی برنامه به صورت زیر است:

```
10
10
```

۳. ناحیه‌ی پروتوتایپ تابع: اگر اسمی در لیست پارامترهای اعلان تابع نوشته شود و تعریف این تابع نوشته نشود، اسم اعلان شده در انتهای اعلان تابع از دسترس خارج می‌شود.

¹. Global variables

به متغیری که در یک بلوک مانند تابع و یا در پارامتر تابع اعلان شده، متغیر محلی^۱ میگویند.

۴. ناحیه‌ی تابع: با اعلان یک **label** در بدن‌هی تابع، دسترسی به آن میتواند در کل تابع (حتی قبل از اعلان آن) صورت گیرد. برای یادآوری یک **label** در یکی از سه مکان زیر قرار می‌گیرد:

- **goto** در **target**
- **switch** در ساختار **case**
- **switch** در ساختار **default**

برای مثال:

```
void f() {
{
    goto label; // label decled after this line
    label:;
}
goto label; // label ignores block scope
}
```

۵. ناحیه‌ی **enumeration** (فصل متغیر و انواع داده (۳))

۶. ناحیه‌ی کلاس (فصل کلاس (۱))

۷. ناحیه‌ی پارامتر قالب (فصل قالبها)

اکنون که با مفهوم ناحیه در C++ آشنا شدیم به ساختار شرطی **if** و **switch** با قسمت مقداردهی اولیه میپردازیم. در بعضی از مواقع (خصوص در انواع داده‌های پیشرفته‌تر) یک متغیر را گاهی اوقات فقط برای استفاده در **if** یا **switch** اعلان و مقداردهی میکنیم. برای مثال:

```
bool status() {
    bool c = f();
    if( c != true ){
        return c;
    }
    //...
}
```

¹. Local variable

با اعلان `c`، اسم این متغیر در کل ناحیه‌ی تابع `status()` قرار می‌گیرد (اصطلاحاً نشت (`leak`) می‌کند). یکی از راههای محدود کردن ناحیه‌ی این متغیر، قرار دادن اعلان و مقداردهی آن با `if` در یک ناحیه‌ی مجزا است (با `{}`). برای مثال:

```
bool status() {
{
    bool c = f();
    if( c != true ){
        //...
        return c; // c also used here
    }
    //...
}
}
```

در C++17 این کار را به صورت زیر و کوتاهتر می‌توان انجام داد:

```
if(bool c = f(); c != true ){
    return c; // c also used here
}
```

متغیری که در این ساختار اعلان و یا مقداردهی شده، در بلوک `else` و یا `if` که در ادامه‌ی ساختار `if` قرار گرفته نیز تعریف شده است.

به طور کلی قسمت مقداردهی اولیه‌ی این ساختار، می‌تواند شامل اعلان متغیر و یا یک عبارت که با `;` به پایان رسیده (مثل `n = n + 10;`) باشد.

آرایه¹

آرایه مجموعه‌های از عناصر از یک نوع داده است. از آرایه همانند متغیر، جهت ذخیره داده استفاده می‌شود؛ با این تفاوت که در یک آرایه، چند مقدار را می‌توان هم زمان ذخیره کرد. فرمت اعلان یک آرایه به صورت زیر است:

```
data_type a[size];
```

که در آن:

نوع داده‌ی عناصر، data_type

نام آرایه‌ای است که اعلان می‌شود و a

تعداد عناصر آرایه را مشخص می‌کند. مثلاً:

```
int a[10];
```

یک آرایه از نوع int به نام a با 10 عنصر اعلان می‌کند به طوری که هر یک از عناصر به طور جداگانه می‌تواند مقداردهی شوند. برای دستیابی به عناصر یک آرایه، نام آرایه همراه با شماره‌ی عنصر مورد نظر در [] نوشته می‌شود.

شمارش عناصر یک آرایه همچنین از ۰ شروع می‌شود. مثلاً در آرایه‌ی a

```
a[0]
```

به عنصر اول و:

```
a[4]
```

به عنصر پنجم تلقی می‌شود. به این ترتیب اگر سایز آرایه n باشد، ایندکس آخرین عنصر n-1 است.

مثال ۱) برنامه‌ای که شامل یک آرایه است:

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

```
int main() {
```

¹. Array

```

int a[4];
a[0] = 2;
a[1] = 11;
a[2] = 500;
a[3] = 55;
for(int i = 0; i < 4; ++i) {
    cout << "a[" << i << "]" << " = " << a[i] <<
'\n';
}
}

```

خروجی برنامه به صورت زیر است:

```

a[0] = 2
a[1] = 11
a[2] = 500
a[3] = 55

```

توضیح) ابتدا آرایه‌ای به نام **a** از نوع **int** و با 4 عنصر اعلان می‌شوند. در ادامه عناصر ۰ تا ۳ این آرایه با یک سری اعداد، مقداردهی و در ادامه با حلقه‌ی **for** نمایش می‌یابند. برای مقداردهی عناصر یک آرایه در حین اعلان می‌توان از {} استفاده کرد. این نوع مقداردهی، مقداردهی aggregate^۱ نام دارد.

مثال (۲)

```

#include <iostream>
using namespace std;

int main() {
    int a[4] = {2, 11, 500 ,55};
    for(int i = 0; i < 4; ++i) {
        cout << "a[" << i << "]" << " = " << a[i] << '\
n';
    }
}

```

^۱. Aggregate initialization

خروجی برنامه به صورت زیر است:

```
a[0] = 2
a[1] = 11
a[2] = 500
a[3] = 55
```

توضیح) این برنامه مانند مثال ۱ است، با این تفاوت که آرایه با لیست {} مقداردهی میشود. توجه شود در ساختار حلقه، مقدار ایندکس در آرایههای بزرگ میتواند بیشتر از مقدار قابل ذخیره در `int` (از نوع `int`) باشد. برای جلوگیری از این مشکل، این متغیر، باید از نوع `std::size_t` که در `<cstdlib>` موجود است اعلام شود. متغیرهای این نوع داده میتوانند بزرگترین سایز یک شی یا آرایه از هر نوع را (بر حسب بایت) ذخیره کنند. با این نوع همچنین میتوان بزرگترین ایندکس هر آرایهای رانیز ذخیره کرد. این مقدار از نوع عدد صحیح مثبت است و بسته به پیاده‌سازیهای مختلف میتواند از انواع متفاوتی باشد (مثلًا `unsigned int`). حلقه‌ی `for` در نتیجه باید به صورت زیر بازنویسی شود:

```
for(std::size_t i = 0; i < 4; ++i) { ... }
```

نکته: اگر تعداد عناصر آرایه از تعداد مقادیر در {} بیشتر باشد، کمپایلر به تعداد مقادیر نوشته نشده ۰ میگذارد. اگر در مثال ۲ هنگام مقداردهی دستور زیر را مینوشتیم:
`int a[4] = {2, 11}`

آرایه به صورت زیر مقداردهی می‌شد:

```
a[0] = 2
a[1] = 11
a[2] = 0
a[3] = 0
```

نیاز به ذکر است که اگر از لیست {} برای مقداردهی اولیه استفاده شود، نیازی به تعیین سایز آرایه در اعلان آن نیست. در این حالت کمپایلر به تعداد مقادیر {} برای آرایه سایز تعیین میکند. در مثال ۲ آرایه را به صورت زیر میتوانستیم مقداردهی کنیم:

```
int a[] = {2, 11, 500, 55};
```

مقداردهی اولیه با {} زمانی مناسب است که تعداد مقادیر کم و در حین اعلان نیز مشخص باشند. در صورت زیاد بودن عناصر آرایه، با ساختارهای حلقه معمولاً یک سری مقادیر حساب و با آن‌ها عناصر آرایه مقداردهی می‌شوند.

پاس دادن آرایه به یک تابع

برای این کار چندین روش وجود دارد که یکی از آن‌ها در اینجا بررسی می‌شود. هنگام فراخوانی، باید نام آرایه بدون [] به تابع پاس داده شود و تابع، آرایه را همراه با [] باید در پارامتر خود دریافت کند.

مثال (۳)

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

void printArray(int a[], size_t n) {
    for(size_t i = 0; i < n; ++i) {
        cout << a[i] << ' ';
    }
}

int main() {
    int a[] = {2, 11, 4, 7};
    printArray(a, 4);
}
```

خروجی برنامه به صورت زیر است:

2 11 4 7

آرایه‌ی چند بعدی^۱

^۱. Multidimensional array

برای اعلان آرایه‌ی دو بعدی می‌توان با فرمت زیر عمل کرد:

```
data_type a[row][column];
```

این دستور یک آرایه از آرایه اعلان می‌کند؛ به طوری که `row` را تعداد عناصر سطر و `column` را تعداد عناصر ستون آرایه می‌توان فرض کرد. برای مثال عناصر آرایه‌ی دو بعدی زیر:

```
int a[2][4];
```

میتوانند به صورت زیر فرض شوند:

a[0] [0]	a[0] [1]	a[0] [2]	a[0] [3]
a[1] [0]	a[1] [1]	a[1] [2]	a[1] [3]

(۴) مثال

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

int main() {
    int a[2][3] = {{4,3,3}, {3,2,10}};
    for(size_t r = 0; r < 2; ++r) {
        for(size_t c = 0; c < 3; ++c) {
            cout << a[r][c] << ' ';
        }
        cout << '\n';
    }
}
```

خروجی برنامه به صورت زیر است:

4	3	3
3	2	10

برای اعلان آرایه با بیش از دو بعد، باید به تعداد بعد [] نوشته شود. برای مثال با دستور:

```
int a[4][2][5];
```

آرایه‌ی ۳ بعدی اعلان خواهد شد.

پاس دادن آرایه‌ی چند بعدی به تابع

برای پاس دادن یک آرایه‌ی چند بعدی، مانند آرایه‌ی یک بعدی میتوان عملکرد با این تفاوت که غیر از سایز بعد اول، سایز بقیه‌ی بعدهای آرایه در پارامتر باید نوشته شوند.

(۵) مثال

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

void printArray(int arr[][3], size_t row) {
    for(size_t r = 0; r < row; ++r) {
        for(size_t c = 0; c < 3; ++c){
            cout << arr[r][c] << ' ';
        }
        cout << '\n';
    }
}

int main() {
    int a[2][3] = {{4,3,3}, {3,2,1}};
    printArray(a, 2);
}
```

خروجی برنامه به صورت زیر است:

4	3	3
3	2	1

استفاده از **for** مبتنی بر بازه در آرایه‌ها

در استانداردهای جدید برای ایجاد حلقه در عناصر آرایه علاوه بر حلقه‌ی `for` عادی، از ساختار `for` مبتنی بر بازه^۱ نیز میتوان استفاده کرد. برای مثال:

```
int a[] = {1,2,3,4};
for(int i : a){
    //...
}
```

در هر حلقه‌ی این ساختار، متغیر `i` مستقیم با خود مقدار عناصر `a` مقداردهی میشود.

مثال ۶

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

int main() {
    int a[] = {1,2,3,4};
    for (int i : a) {
        cout << i << ' ';
    }
    cout << '\n';
    return 0;
}
```

خروجی برنامه به صورت زیر است:

```
1 2 3 4
```

در حلقه‌ی `for` مبتنی بر بازه، با اعلان متغیر از نوع `auto` میتوان بدون نیاز به دانستن نوع داده‌ی یک آرایه، در آن ایجاد حلقه کرد. مثلاً:

```
for(auto i : a)  {
    //...
}
```

¹. Range-based for loop

متغیر و انواع داده (۳) **struct**

با واژه‌ی **struct** می‌توان یک نوع داده‌ی جدید، بر اساس انواع داده‌ی موجود در برنامه ایجاد کرد. تعریف یک **struct** به صورت زیر است:

```
struct s {
    // members
};
```

s نام **struct** و **members** اعضای **struct** اند. در C++ یک سری نوع داده برای کار با مقادیر شناخته شده مثل **int** برای اعداد صحیح، **float** برای اعداد اعشاری و ... به طور پیش فرض وجود دارد؛ ولی در برنامه گاهی اوقات نیاز به ایجاد انواع داده‌های پیشرفته‌تر مثل نوع داده‌ای برای ذخیره‌ی اطلاعات نقطه در فضای دو بعدی می‌شود. به این منظور می‌توان **struct** زیر را تعریف کرد:

```
struct Point {
    double x;
    double y;
};
```

به دلیل اینکه x و y های یک نقطه اعداد اعشاری نیز می‌توانند باشند، یک **struct** تعریف می‌کنیم که در قالب متغیرهای **double** این اطلاعات در آن قابل ذخیره باشد. سپس هر متغیر اعلان شده از آن **struct**، خود شامل متغیرهای عضو x و y خواهد بود.

اعلان متغیر از یک **struct**، مانند هر متغیر دیگر به صورت زیر انجام می‌شود:

```
MyStruct m;
```

بعد از اعلان متغیر از **struct**، دسترسی به اعضای آن با اپراتور دسترسی عضو^۱ صورت می‌گیرد. مانند:

¹. Member access operator

m.member

(مثال ۱)

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

struct Point {
    double x;
    double y;
};

int main(){
    Point p1, p2;
    p1.x = 1.3;
    p1.y = 3.5;
    p2.x = 4.3;
    p2.y = 3.5;
    cout << "p1: (" << p1.x << ", " << p1.y << ")\n";
    cout << "p2: (" << p2.x << ", " << p2.y << ")\n";
}
```

خروجی برنامه به صورت زیر است:

```
p1: (1.3, 3.5)
p2: (4.3, 3.5)
```

تعريف متغير از **struct**، می‌تواند در انتهای تعريف **struct** و قبل از **#include** قرار گيرد (تعريف چند متغير را با تفکیک، نوشته میشود). مانند:

```
struct Point {
    double x;
    double y;
} p1, p2, p3;
```

برای فرمهای سادهی **Point** مثل **struct** در مثال ۱، اگر مقادیر مورد نیاز برای مقداردهی متغير در هنگام اعلان مشخص باشند، مانند آرایه از {} برای مقداردهی

اعضای متغیر میتوان استفاده کرد (مقداردهی اولیه‌ی aggregate). مانند:

```
Point p = {1.1, 2.4};
```

و یا:

```
Point p{1.1, 2.4};
```

با این روش، اعضای متغیر به صورت نظریر به نظریر، از عناصر درون {} مقداردهی میشود. نیاز به ذکر است که اعضای یک struct لزوماً از یک نوع نیستند. برای مثال اگر نیاز به ذخیره‌ی اطلاعات محصولات یک فروشگاه شود - با فرض اینکه نیاز به ذخیره‌ی وزن، قیمت و موجود بودن محصول در برنامه هست - struct زیر را میتوان تعریف و استفاده کرد:

```
struct Product {
    double weight;
    int price;
    bool inStock;
};
```

```
Product orange{0.2, 3000, true};
Product apple{0.1, 2000, false};
```

همانگونه که مشاهده میشود، متغیرهای عضو این struct از یک نوع نیستند. زمانی از struct استفاده می‌کنیم که می‌خواهیم یک سری داده‌های مرتبط به هم در یک واحد ذخیره شوند. در مثال ۱ می‌توانستیم هر عضو struct را در یک آرایه‌ی جدا ذخیره کنیم (یک آرایه برای ذخیره Xها، یک آرایه‌ای دیگر برای Uها). در نتیجه جهت پاس دادن این مقادیر به یکتابع، تعریف تابع باید به صورت زیر نوشته شود:

```
void f(double x[], double y[])
// ...
}
```

ولی با قرار دادن آنها در یک struct، متغیرهای ساخته شده از struct را میتوان در

یک آرایه ذخیره کرد. در نتیجه تعریف تابع به صورت زیر میشود:

```
void f(Point b[]){  
    // ...  
}
```

همانطور که دیده میشود دادههای مرتبط به هم با استفاده از **struct**, به نحو بهتری مدیریت و استفاده خواهد شد.

توجه شود که در یک **struct** میتوان توابع، توابع **virtual** و اجزای پیشرفته‌تری نیز اعلان کرد (در فصل کلاسها بررسی میشوند)؛ ولی از **struct** در C++, به طور معمول فقط برای ذخیره‌ی مجموعه‌ای از دادهها استفاده میشود.

همچنین به متغیرهای ساخته شده از **struct**, شی نیز میگویند. البته شی در C++ تعریف کلیتری دارد و به هر قسمتی از حافظه که سایز، طول عمر، مقدار و ... داشته باشد نیز شی گفته میشود.

union

یک **union** است که در هر لحظه فقط یک عضو آن میتواند مقدار داشته باشد. حافظه‌ی مورد نیاز برای ذخیره‌ی شی **union**, به اندازه‌ی بزرگترین متغیر عضو آن است؛ زیرا همانند **struct** نیازی به اشغال حافظه، برای تمام اجزای **union** نیست. فرمت تعریف یک **union** به صورت زیر است:

```
union id {  
    // members  
} optional_variables;
```

union نام **id**

اعضای **union** **members** است،

در قسمت **optional_variables**, با کاما میتوان متغیر از **union** اعلان کرد.

دسترسی به اعضای **union** نیز توسط اپراتور دسترسی عضو . صورت می‌گیرد.

مثال ۲) برنامه‌ای که شامل **union** است:

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

union Data {
    int a;
    int b;
    int c[2];
};

int main() {
    Data d;
    d.a = 3;
    cout << d.a << '\n';
    d.b = 44;
    cout << d.b << '\n';
    d.c[0] = 3;
    d.c[1] = 4;
    cout << d.c[0] << ' ' << d.c[1] << '\n';
}
```

خروجی برنامه به صورت زیر است:

```
3
44
3 4
```

توضیح) توجه شود قبل از استفاده از `union` باید بدانیم کدام عضو آن هم اکنون مقدار دارد؛ استفاده از اعضای دیگر غیر از این عضو، باعث بروز خطا یا تولید نتایج غلط در برنامه میشود.

هنگام استفاده از `{ } در مقداردهی union`، مانند:

```
U u = {10};
```

اولین عضو `union` مقداردهی میشود.

ایجاد `union` بدون اسم نیز امکان پذیر است. این `union`ها، `union`های ناشناس¹ نام

¹. Anonymous unions

دارند. دسترسی به اعضای union‌های ناشناس بدون متغیر صورت می‌گیرد و اعضای آنها مستقیم در ناحیه‌ای که اعلان می‌شود در دسترس قرار می‌گیرد.

مثال (۳)

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

int main() {
    union {
        int a;
        int b;
    };
    a = 4;
    cout << a << '\n';
}
```

خروجی برنامه به صورت زیر است:

4

در union نیز (نه union ناشناس) - مانند struct - میتوان تابع تعریف کرد. ولی از این قابلیت به ندرت در C++ استفاده می‌شود.
هدف اصلی استفاده از union‌ها صرفهجویی در حافظه است. زمانی که در طول عمر چند متغیر، در هر لحظه نیازی به کار و مقداردهی بیش از یکی از آنها نیست، با میتوانند به صورت بهینه‌تر پیاده شوند.

enum

با واژه‌ی enum میتوان انواع داده‌هایی را که متشکل از یکسری اسم‌های نامگذاری شده است اعلان کرد. به این انواع داده‌ها، انواع داده‌ی شمارشی^۱ می‌گویند. فرمت تعریف یک enum به صورت زیر است:

¹. Enumerated types

```
enum id {
    value1,
    value2,
    ...
} optional_variables;
```

نوع داده‌ی `id`، `enum` می‌توانند اختیار `value1`, `value2` و ... مقادیری اند که متغیرهای ایجاد شده از `id` می‌توانند انتخاب کنند (این مقادیر شمارنده¹ نام دارند). در قسمت `optional_variables`, با کاما می‌توان متغیر از `id` اعلان کرد.

مثال (۴)

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

enum Color {
    GREEN, BLACK, WHITE, BLUE, PURPLE, YELLOW
};

int main() {
    Color c;
    c = BLACK;
    if(c == BLACK)
        cout << "c represents black\n";
    else
        cout << "c represents any color except black\n";
}

```

خروجی برنامه به صورت زیر است:

```
c represents black
```

در `enum`ها از ساختار شرطی `switch` نیز می‌توان استفاده کرد. برای مثال:

¹. Enumerator

```

Color color{GREEN};
switch(color){
    case BLACK:
        cout << "black\n";
        break;
    case GREEN:
        cout << "green\n";
        break;
    // ...
}

```

زیرا قسمت شرط `switch`, علاوه بر مقادیر عدد صحیح میتواند شامل انواع `enum` نیز باشد.

مقادیر شمارندها از نظر زیرساختی با مقادیر عدد صحیح در برنامه مشخص میشوند.^۱ این مقادیر به طور پیش فرض از ۰ شروع میشوند و سپس مقادیر یک واحد بیشتر از شمارندهی قبل از خود را اختیار میکنند. کمپایلرها برای بهینهسازی، معمولاً از کوچکرین نوع عددی ممکن برای تعیین نوع `enum` ها استفاده میکنند. مثلا در مثال^۲، کمپایلرها معمولاً نوع `char` را برای `Color` در نظر میگیرند. ثابت نبودن نوع مقادیر `enum` ها محدودیتی نیز ایجاد میکند: امکان اعلان قبل از تعریف آنها ممکن نخواهد بود. برای برطرف کردن این محدودیت میتوان نوع `enum` را به صورت زیر و از نوع ثابت تعريف کرد:

```
enum id: type { }
```

برای مثال:

```
enum Color: int {BLACK, WHITE};
```

همچنین با اپراتور `=` میتوان مقدار شمارندهها را به اعداد دلخواه تغییر داد. در این

^۱ به نوع این مقادیر underlying type گفته میشود.

صورت مقدار هر شمارنده که تعیین نشود، با یک عدد بیشتر از مقدار شمارنده‌ی قبل از خود مقداردهی می‌شود. برای مثال:

```
enum Book{
    FICTIONAL = 2,
    DOCUMENTARY = 5,
    SCIENTIFIC
};
```

مقدار SCIENTIFIC برابر با 6 خواهد شد.

با تعریف یک enum، شمارنده‌ها در برنامه مستقیماً و به صورت implicit می‌توانند به انواع داده‌های دیگر تبدیل شوند. قابل تبدیل بودن مقادیر شمارنده‌ها در خیلی از مکانها، امکان نوشتن کدهای بی معنی ولی قابل کمپایل را فراهم می‌کنند. برای مثال دستورات زیر صحیح‌اند:

```
enum Color{black, white};
Color c = black;
if( c < 1.5) {... // implicit conversion!}
```

در استانداردهای جدید یک نوع enum به اسم کلاس¹ افزوده شده که این مشکل را ندارند. مثل:

```
enum class Color { black, white, green };
```

در کلاس‌های enum، مقادیر شمارنده‌ها به صورت خودکار به مقادیر عدد صحیح تبدیل نمی‌شوند؛ ولی در صورت نیاز می‌توان آنها را به مقادیر مختلف به صورت تبدیل کرد. برای مثال:

```
enum class Color{black, white};
Color c = Color::black;
if( static_cast<double>(c) < 1.5) {... // valid
```

¹. enum class

تفاوت دیگر یک کلاس enum با یک کلاس enum در این است که شمارندهای enum عادی، در کل ناحیهای که enum تعریف شده قرار میگیرند؛ در نتیجه احتمال وجود اسمهای مشابه در برنامه های بزرگ افزایش مییابد. برای مثال:

```
enum color {black, white, green};  
int black; // error!
```

دسترسی به شمارندهای یک کلاس enum، با : صورت میگیرد. در نتیجه باعث قرار دادن مستقیم آنها در خارج از بلوک کلاس enum نخواهد شد.

برای مثال:

```
enum class Color { black, white, green };  
Color c = Color::black;
```

تفاوت آخر enum با کلاس enum در نوع دادهی شمارندها است. در کلاس enum، به طور پیش فرض کمپایلر، نوع دادهی int را برای شمارندها قرار میدهد؛ در نتیجه بدون نیاز به تعیین نوع داده آن مانند زیر:

```
enum class Color: int{black, white};
```

میتوانند قبل از تعریف، اعلان شوند.

alias و `typedef`

با `typedef` میتوان نوع دادهی جدیدی را بر اساس نوع دادهی موجود در برنامه تعریف کرد. فرمت آن به صورت زیر است:

```
typedef existing_type new_type;
```

نوع دادهی new_type موجود و existing_type نوع دادهی جدیدی است که تعریف میشود.

مثال ۵) برنامهای شامل `:typedef`

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

typedef unsigned long ulongint;

int main() {
    ulongint i = 23;
    cout << i << '\n';
}
```

خروجی برنامه به صورت زیر است:

34

توضیح) کمپایلر اسم جدید ulongint را برای نوع داده unsigned long قرار میدهد.

در استاندارد های جدید با واژه‌ی using، مانند `typedef`، میتوان اسم جدیدی برای نوع داده‌ی موجود اعلان کرد. با فرمت:

```
using newType = existingType;
```

به نوع داده‌ی اعلان شده، alias گفته میشود (newType در بالا). برای مثال دو دستور زیر با هم برابرند:

```
typedef unsigned long ulongint;
using ulongint = unsigned long;
```

با فرق `typedef` و اعلان alias، پس از بررسی قالبها آشنا خواهیم شد. با استفاده از `typedef` و اعلان alias، می‌توان انواع داده‌هایی که نام طولانی دارند را با نام کوتاهتر جایگزین کرد. البته انواع داده‌هایی که تاکنون استفاده کردیم همگی نام کوتاه داشتند. در آینده با انواع داده‌هایی آشنا خواهیم شد که بدون قرار دادن نام جایگزین استفاده از آنها بسیار مشکل خواهد بود. به عنوان مثال:

```
using vector3d =
```

```
std::vector<std::vector<std::vector<std::vector<double>>>;
```

به این ترتیب با استفاده از اعلان `alias`، نوع داده‌ی طولانی فقط یک بار نوشته و بهجای آن از `vector3d` در برنامه استفاده خواهد شد.

اپراتورها (۲)

اپراتور sizeof

در C و C++ با اپراتور `sizeof` می‌توان مقداری که یک شی یا نوع داده در حافظه اشغال می‌کند را از نوع بایت بدست آورد. یک `char` نیز همیشه برابر یک بایت است.

اپراتور `sizeof` می‌تواند هم نوع داده و هم متغیر باشد.

مثال ۱) برنامه‌ای که مقدار اشغال شده برای یک متغیر `int` را در خروجی چاپ می‌کند:

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

```
int main() {
    int a;
    cout << sizeof(a) << '\n';
}
```

خروجی برنامه به صورت زیر است:

4

مقدار فضایی که یک متغیر در سیستم‌های مختلف اشغال می‌کند، همیشه یکسان نیست؛ در نتیجه خروجی مثال ۱ ممکن است برای سیستم شما ۴ نباشد. یکی کاربردهای اپراتور `sizeof` نیز دستیابی به مقادیر اشغال شده، مختص سیستمی است که برنامه روی آن اجرا می‌شود.

دستورات زیر تعداد عناصر یک آرایه را از طریق تقسیم مقدار اشغال شده برای آرایه بر مقدار مورد نیاز برای اشغال یک عنصر نمایش میدهد:

```
int a[6];
cout << sizeof(a)/sizeof(int) << '\n'; // size
亨گام استفاده از sizeof برای یک مقدار (برخلاف نوع)، میتوان پرانتز آن را قرار نداد.  
مثلا:
```

```
cout << sizeof a / sizeof(int) << '\n'; // size
```

مقدار بازگشته اپراتور `std::size_t` از نوع `sizeof` است. جهت یادآوری، با این نوع داده میتوان بیشترین سایز ممکن برای اشغال یک شی (یا آرایه) از هر نوع را ذخیره کرد. با این نوع همچنین میتوان بیشترین مقدار ایندکس ممکن آرایه را نیز ذخیره کرد. اپاندهای اپراتور `sizeof` ارزیابی نمیشوند. برای مثال با دستور زیر:

```
sizeof(cout << 4);
```

فقط سایز عبارت برمیگردد و خروجی `cout` نمایش نمییابد.

اپراتور آدرس^۱

آدرس فضای یک متغیر در حافظه با قرار دادن اپراتور آدرس (`&`) قبل از اسم آن بدست میآید.

مثال (۲)

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

int main() {
    int a;
    cout << "address of a : " << &a << '\n';
}
```

دستورات را کمپایل و اجرا کنید تا خروجی مشابه زیر تولید شود:

```
address of a : 0x7ffd67ff742c
```

کار با حافظه (۱)^۲

اشارهگر^۳

¹. Address-of operator

². Memory management

³. Pointer

اشاره‌گر یک نوع متغیر است که به جای ذخیره کردن مقداری از انواع داده‌ای که تاکنون بررسی کردیم (`int`, `char` و ...)، آدرسی از حافظه را در خود ذخیره می‌کند. به این ذخیره سازی اصطلاحاً اشاره کردن گفته می‌شود. اشاره‌گر هم میتواند به یک متغیر و یا به طور کلی به یک شی اشاره کند. در این بخش اشاره به متغیر را بررسی خواهیم کرد.

تعريف اشاره‌گر

فرمت اعلان یک اشاره‌گر به صورت زیر است:

```
data_type *p;
```

که `data_type` نوع داده‌ی شی و `p` نیز نام اشاره‌گر است.

مانند:

```
int *a;
```

اشاره به یک متغیر

برای این کار باید نام اشاره‌گر را بدون ستاره نوشت و آن را با آدرس متغیری که می‌خواهیم به آن اشاره شود (با استفاده از اپراتور آدرس `&`) مقداردهی کرد. مانند:

```
int b;
int *a;
a = &b;
```

به `a` اشاره خواهد کرد.

پس از اشاره‌ی یک اشاره‌گر، `a` بیانگر آدرسی از حافظه و `a*` به مقدار متغیر مورد اشاره تلقی می‌شود. دسترسی اشاره‌گر به مقدار متغیر با `*`، دیرفرنس¹ نام دارد. مثال ۱) برنامه‌ای که شامل یک اشاره‌گر است.

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

¹. Dereference

```

int main() {
    int b = 3;
    int *a;
    a = &b;
    cout << "dereferencing a: " << *a << '\n';
}

```

خروجی برنامه به صورت زیر است:

Dereferencing a : 3

با دیرفرنس، مستقیماً به مقدار ذخیره شده در متغیر مورد اشاره دسترسی ایجاد میشود؛ با تغییر دادن این مقدار با اشارهگر، مقدار متغیر نیز عوض میشود.

مثال (۲)

```

#include <iostream>
using namespace std;

int main() {
    int b = 3;
    int *a = &b;
    *a = 10;
    cout << "b is now " << b << '\n';
}

```

خروجی برنامه به صورت زیر است:

b is now 10

توجه شود که در دستور:

int *a = &b;

مقدار **a*** برابر با **b&** نیست (**a** برابر با **b&** است) ولی استفاده از اپراتور *****، در اعلان **a** نیاز است و به کمپایلر اطلاع می‌دهد که **a** از نوع اشارهگر است.

تاکنون با پاس دادن یک عبارت به تابع، از کپی آرگومان جهت مقداردهی پارامتر استفاده میشد. در نتیجه با تغییر مقدار پارامتر در تابع، مقدار اصلی آرگومان تغییر نمی‌یافتد.

مثال (۳)

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

void f(int a) {
    a = 5000;
}

int main(){
    int x = 4;
    f(x);
    cout << "x is now : " << x << '\n';
}
```

خروجی برنامه به صورت زیر است:

```
x is now : 4
```

با پاس دادن آدرس و دریافت آن توسط یک اشارهگر،تابع مستقیماً با مقدار آرگومان کار خواهد کرد، در نتیجه با تغییر مقدار آن از طریق دیرفرنس اشارهگر، مقدار آرگومان نیز عوض می شود.

مثال (۴)

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

void f(int *a) {
    *a = 5000;
}

int main() {
    int x = 4;
    f(&x);
    cout << "x is now : " << x << '\n';
}
```

خروجی برنامه به صورت زیر است:

```
x is now : 5000
```

توضیح) همانند مثال ۳ است با این تفاوت که پارامترتابع از نوع اشارهگر است و آدرس `X` در فراخوانی پاس داده می‌شود.

یکی کاربردهای تعیین پارامتر از نوع اشارهگر، جلوگیری از کپی اشیای بزرگ است که نیاز به صرف منابع زیاد دارد.

اشاره به `nullptr`

اگر در برنامه یک اشارهگر به `nullptr` اشاره کنند، به این معنی است که اشارهگر به هیچ شیای اشاره نمیکند. برای مثال:

```
int *p;
p = nullptr;
```

همچنین از نظر `false` و `true` بودن، اگر اشارهگر به `nullptr` اشاره کند `false` و در غیر این صورت `true` برمیگردد. به این ترتیب با ساختار شرطی میتوان اشاره یا عدم اشاره‌ی اشارهگر به یک شی را بررسی کرد.

(۵) مثال

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

int main() {
    int a = 3;
    int *b = &a;
    b = nullptr;
    if(!b)
        cout << "b is pointing to nullptr\n";
    else
        cout << "b is not pointing to nullptr\n";
}
```

خروجی برنامه به صورت زیر است:

```
b is pointing to nullptr
```

`nullptr` در استانداردهای پیشین `NULL` نام داشت و برابر 0 عددی بود. به دلیل عددی بودن مقدار `NULL`، اگر دو تابع سربارگذاری شده داشته باشیم به طوری که پارامتر یکی از آنها اشارهگر و دیگری از نوع عدد صحیح (مثل `int`) باشد، با پاس دادن `NULL`، تابع با پارامتر عدد صحیح فراخوانی میشود:

```
void f (int *a);
void f (int a);
f(NULL); // f(int)
```

این مشکل با `std::nullptr_t` برطرف شده است. `nullptr` از نوع `nullptr_t` است و فقط میتواند به اشارهگر و یا مقدار `bool` تبدیل شود. برای مثال:

```
int a = NULL; // old code. works.
int a = nullptr; // error!
int *a = nullptr; // ok.
bool a = nullptr; // ok.
```

تخصیص حافظهٔ دینامیک^۱

طول عمر یک متغیر معمولی محدود به ناحیه (scope)‌ای است که در آن اعلان میشود. در بعضی از موقع نیاز به ایجاد اشیایی است که وابسته به ناحیه (scope) اعلانشان نیستند؛ برای مثال در بعضی از توابع، مناسب است که فضای متغیرهایی که در آنها اعلان میشوند، با اتمام اجرای آنها از بین نرونند. تخصیص این نوع اشیا، از نوع تخصیص حافظهٔ دینامیک باید صورت گیرد. به مکانی از حافظه که این اشیا در آن تخصیص میباند `free store` یا `heap` (در مقابل `stack`) گفته میشود. این اشیا همچنین برخلاف اشیایی که تاکنون ایجاد کردیم، در حین اجرای برنامه تخصیص میباند و مقدار آنها نیز در حین اجرا مشخص میشود.

^۱. Dynamic memory allocation

در تخصیص دینامیک، فضای اشیای ایجاد شده توسط اشارهگر نگهداری و استفاده میشود. در این نوع کاربرد اشارهگرها در واقع اشارهگر به جای اشاره به یک متغیر، به طور کلیتر به یک شی در حافظه اشاره می‌کند.

[] new و new

برای تخصیص دینامیک یک شی، از اپراتور **new** استفاده میشود. برای مثال دستور زیر:

new int;

یک شی از نوع **int** ایجاد میکند. این اپراتور پس از ایجاد شی، یک اشارهگر به ابتدای فضای شی ایجاد شده برمیگرداند. برای استفاده از این شی باید آدرس برگشته را در یک اشارهگر ذخیره کرد. مثل:

```
int *p = new int;
```

مثال ۶) برنامه‌ای که در آن یک متغیر به صورت دینامیک تخصیص مییابد (توجه شود این مثال نیاز به تکمیل دارد. در پایان مثال به تکمیل آن میپردازیم):

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

```
int main() {
    int *a = new int;
    *a = 5;
    cout << "a = " << *a << '\n';
}
```

خروجی برنامه به صورت زیر است:

```
a = 5
```

اشیای تخصیص یافته با **new**، به طور خودکار آزاد نمیشوند. این کار باید با اپراتور **delete** و توسط برنامه نویس صورت گیرد. برای مثال:

```
int *i = new int;
```

```
delete i;
```

مثال کامل شدهی مثال ۶ به صورت زیر نوشته میشود:

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

int main() {
    int *a = new int;
    *a = 5;
    cout << "a = " << *a << '\n';
    delete a;
}
```

برای مقداردهی اولیه یک شی که به صورت دینامیک تخصیص یافته است، از () یا {} میتوان استفاده کرد:

```
int *p = new int{6};
int *p2 = new int(6);
```

در غیر این صورت به صورت پیش فرض مقداردهی میشود.
تخصیص دینامیک یک آرایه، با اپراتور new [] صورت میگیرد. مانند:

```
new int[size];
```

و آزاد کردن آن با

```
delete [] a;
```

انجام میشود. برای مثال:

```
int *a = new int[10];
delete [] a;
```

مقداردهی اولیهی یک آرایهی دینامیک میتواند توسط {} و یا با () خالی انجام شود. در صورت استفاده از (), تمام عنصرها با ۰ مقداردهی میشوند و با {} آرایه مقداردهی اولیهی aggregate خواهد شد.

برای مثال:

```
int *a = new int[size](); // 0
int *a = new int[4]{1,2,3,4}; // {1,2,3,4}
```

مثال ۷) برنامه‌ای که شامل یک آرایه است که سایز آن با تخصیص دینامیک آرایه، در حین اجرای برنامه و توسط کاربر مشخص می‌شود:

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

int main() {
    size_t n;
    int *a;
    cout << "Specify array length: ";
    cin >> n;
    a = new int[n];

    for(size_t i = 0; i < n; ++i)
        a[i] = i;

    for(size_t i = 0; i < n; ++i)
        cout << "[" << i << "] = " << a[i] << '\n';
}
```

خروجی برنامه به صورت زیر است:

```
Specify array length: 6
[0] = 0
[1] = 1
[2] = 2
[3] = 3
[4] = 4
[5] = 5
```

نشت حافظه (memory leak)

نشت حافظه یا memory leak در برنامه زمانی رخ میدهد که فضای تخصیص یافته‌ای که دیگر نیازی به آن نیست آزاد نشود. برای مثال در دستورات زیر:

```
void f(){
```

```

int *p = new int;
}
int main(){
    f(); // memory leak
}

```

اشارهگر p، بدون `delete` کردن فضای خود - با پایان اجرای تابع f() - از بین میرود؛ در چنین حالتی شی آزاد نشده علاوه بر از بین نرفتنش، غیر قابل دسترس نیز میشود. نشت حافظه معمولاً در کدنویسی غلط توابع ایجاد میشود. برای مثال اگر برنامه قبل از آزاد کردن منابع دینامیکی که در یک تابع تخصیص یافته‌اند، به دستور `return` برسد و یا استثنای ایجاد شود (بررسی در فصل استثنایها)، منجر به نشت حافظه خواهد شد. نشت حافظه در برنامه‌هایی که به طور مداوم در اجرا باقی میمانند (مثل برنامه‌های سرور)، به مرور زمان حافظه را پر میکند و درنهایت منجر به توقف برنامه میشود.

مثال:

```

int status(){
    int *p = new int;
    if(condition)
        return 4;
    delete p; // may not be called.
}

```

در هنگام تخصیص حافظه دینامیک یکی از مشکلات دیگری که در رابطه با حافظه ایجاد میشود، اشارهگر معلق (dangling pointer) است (البته اشارهگر معلق محدود به حافظه دینامیک نیست). اشارهگر معلق، اشارهگری است که به شیای اشاره میکند که صحیح نیست یا از بین رفته. برای مثال:

```

int *i = new int;
int *i2 = i;
delete i;
i = nullptr;

```

در این صورت `i2` یک اشارهگر معلق میشود. یک نمونه از اشارهگر معلق خارج از

تخصیص دینامیک، به صورت زیر است:

```
{
    int *i = nullptr;
{
    int c;
    dp = &c;
}
}
```

اپراتور انتخاب عضو - <^۱

زمانی که یک متغیر (مثلاً متغیر struct) از نوع اشارهگر داریم، برای دسترسی به اعضای آن به جای نوشتن دستور:

`(*v).x`

می‌توان از اپراتور `->` استفاده کرد. مانند:

`v->x`

(۸) مثال

```
#include <iostream>
using namespace std;
struct MyStruct{
    int a;
};

int main() {
    MyStruct m;
    m.a = 10;
    MyStruct *m2 = &m;
    cout << m2->a << '\n';
}
```

خروجی برنامه به صورت زیر است:

^۱. Arrow member selection operator

10

اشاره به اشارهگر^۱

یک اشارهگر می‌تواند به اشارهگر دیگر اشاره کند. در این صورت برای دیفرنس کردن، باید به تعداد سطح اشاره، * قرار گیرد.

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

int main() {
    int a = 10;
    int *p = &a, **t;
    t = &p;
    cout << **t << '\n';
}
```

خروجی برنامه به صورت زیر است:

10

آرایه‌ی دو بعدی نیز به اشارهگر به اشارهگر می‌تواند تبدیل شود؛ همین‌طور آرایه‌ی n بعدی نیز به همین ترتیب به اشارهگر با n سطح اشاره تبدیل می‌شود.

اشارهگر const و اشاره به const^۲

واژه‌ی const در اشارهگرها دو کاربرد دارد. یک کاربرد آن اشارهگر const اعلان می‌کند و دیگری اشارهگری اعلان می‌کند که به یک شی const اشاره می‌کند.

اشارهگر const: اشارهگرها نیز مانند بقیه‌ی متغیرها می‌توانند به صورت const تعریف شوند. اشارهگر const یک نوع اشارهگر است که آدرسی که به آن اشارهگر اشاره می‌کند ثابت است و نمی‌توان تغییر کند. در اعلان چنین اشارهگری باید پس از *، واژه‌ی const قرار گیرد. فرمت اعلان آن به صورت زیر است:

^۱. Pointer to pointer

^۲. constant pointer and pointer to const

```
dataType * const variableName;
```

توجه شود که مقدار اشارهگر **const** مانند بقیه اشارهگرها باید با اعلان مقداردهی شود. مانند:

```
int a = 4;
int c = 3;
int * const b = &a;
b = &c; // error
```

با وجود اینکه آدرس شیایی که اشارهگر **const** اشاره میکند ثابت است، خود مقدار شی مورد اشاره میتواند تغییر کند. مانند:

```
int a = 4;
int * const b = &a;
*b = 3; // OK
```

اشارهگر به **const**: اشارهگر به **const**، اشارهگری است که مقدار شیایی که به آن اشاره میکند را نمیتواند تغییر دهد؛ ولی آدرس شی مورد اشاره اش را در هر زمان میتوان تغییر داد.

فرمت اعلان آن به صورت زیر است:

```
dataType const * p;
```

: یا

```
const dataType * p;
```

مثال:

```
int a = 5;
int const *b;
b = &a;
*b = 2; // error
```

همچنین اشارهگر میتواند از نوع اشارهگر **const** به **const** باشد؛ در این صورت دو ویژگی **const** در اشارهگر را خواهد داشت. برای مثال:

```
int i = 5;
```

```
const int *const p{&i}; // const pointer to const
```

کاربرد اصلی اشارهگر معمولاً در توابع است. برای مثال در تابعی که نیاز به تغییر مقدار آن نیست و همچنین دریافت آن‌ها به صورت عادی از طریق کپی مناسب نباشد (مثلاً حجم شی زیاد باشد)، پارامتر آن را میتوان از نوع اشارهگر به `const` اعلان کرد؛ این تابع دیگر نمیتواند آرگومانهای دریافت کرده را تغییر دهد. این ویژگی بخصوص در برنامه‌های بزرگ اهمیت دارد؛ زیرا بررسی تعریف تمامی توابع در برنامه امکان‌پذیر نیست. ولی اطمینان حاصل میشود که پارامتر آن از نوع اشارهگر به `const` باشد، شی ما را تغییر نخواهد داد.

*void

* نوع خاصی از اشارهگر است که میتواند به اشیای هر نوع داده‌ای اشاره کند. یکسان یا یکسان نبودن دو اشارهگر از نوع `*void` را مانند اشارهگرهای دیگر میتوان بررسی کرد؛ ولی فرآیندهای دیگر اشارهگرها مثل دیرفرنس، افزودن یک مقدار به آنها و ... ممکن نیست و خطای کمپایلر را به همراه خواهد داشت (زیرا نوع شی مورد اشاره آنها مشخص نیست). با این وجود امکان کست اشارهگرهای `void*` به نوعهای دیگر اشارهگر وجود دارد. برای مثال:

```
int i = 3;
float f = 1.6;

void *p = &i, *p2 = &f;
cout << *static_cast<int*>(p) << '\n';    // cast to int
cout << *static_cast<float*>(p2) << '\n'; // cast to float
```

استفادهٔ معمول اشارهگر `void*` در توابع سطح‌پایین سیستم است که امکان پاس دادن هر نوع شیای را فراهم میکند. در برنامه‌های سطح بالا به طور معمول استفاده نمیشود.

رفنس^۱

رفنس یک نام جدید برای متغیر از پیش تعریف شده ایجاد می‌کند. در C++ دو نوع

رفنس موجود است: رفنس **lvalue** و رفنس **rvalue**.

ابتدا به رفنس **lvalue** میپردازیم؛ همچنین در این کتاب منظور از رفنس، رفنس

lvalue است (مگر اینکه به طور خاص رفنس **rvalue** نوشته شود). برای تعریف

رفنس **lvalue** باید اپراتور & قبل از نام رفنس قرار گیرد.

مثال^(۹) برنامه‌ای که شامل یک رفنس است:

```
#include <iostream>
using namespace std;
int main() {
    int a = 4;
    int &b = a;
    cout << b << '\n';
}
```

خروجی برنامه به صورت زیر است:

4

رفنس **lvalue** میتواند به یک **lvalue** ارجاع داده شود. در عمل،

مقداریست که آدرس آن در برنامه قابل دریافت است (مثل یک متغیر). رفنس مانند

اشارهگر، چون مقدار یک متغیر را کپی نمیکند - و به فضای آن ارجاع داده میشود - با

تغییر مقدار رفنس، فضای متغیر مورد ارجاع نیز تغییر میکند. مثلاً:

```
int a = 2;
int &b = a;
b = 13; // a is now 13
```

مهمنترین تفاوت‌های رفنس با اشارهگر در موارد زیر است:

۱. در طول عمرش می‌تواند فقط به یک متغیر ارجاع داده شود؛ این ارجاع نیز با

مقداردهی اولیه‌ی آن تعیین میشود:

¹. Reference

```
int a = 3;
int &b; // error
b = a;
```

۲. معادل `nullptr` برای رفرنس موجود نیست. یکی از نتیجههای این ویژگی در تعیین نوع پارامتر توابع است؛ برای مثال در توابعی که پارامتر آنها از نوع اشارهگر است و پاس دادن `nullptr` به آنها بیمعنی است، به جای دریافت اشارهگر و بررسی عدم بودن آرگومان، مثل:

```
int f(int *p) {
    if(p == nullptr) {
        // stop function execution, etc
    }
}
```

با تعیین پارامتر از نوع رفرنس، امکان پاس دادن `nullptr` و نیاز به چک آن از بین خواهد رفت. استفاده از اشارهگر در پارامتر و بررسی `nullptr` نبودن (در صورت نیاز) زمانی استفاده میشود که به طور خاص نیاز به اشارهگر باشد؛ در غیر این صورت پارامتر را از نوع رفرنس تعیین میکنند.

۳. برخلاف اشارهگر، نمیتوان آرایهای از رفرنس اعلام کرد.

۴. برخلاف اشارهگر، امکان اعلان رفرنس به رفرنس ممکن نیست.

۵. رفرنس برخلاف اشارهگر، آدرس یکسانی با متغیر مورد ارجاعش دارد.

رفرنس معمولاً در دو جا استفاده میشود: در پارامتر و در بازگشت توابع.

مثال ۱۰) برنامه‌ای که کاربرد رفرنس را در پارامتر تابع نشان میدهد:

```
#include <iostream>
using namespace std;
void f(int &a) {
    a = 6;
}
int main() {
    int a = 4;
    f(a);
```

```
    cout << a << '\n';
}
```

یکی از دلایل دیگر استفاده از رفرنس به جای اشاره‌گر، syntax راحت‌تر کار با رفرنس است؛ در رفرنس، برای دستیابی به مقدار ارجاع داده شده، از اپراتور * استفاده نمی‌شود. راحتی کار با رفرنس در واقع زمانی محسوس است که رفرنس به یک متغیر با عضو ارجاع داده می‌شود. در این صورت برای دسترسی به اعضا آن، به جای استفاده از اپراتور /نتخاب عضو ->، از نام رفرنس و . استفاده می‌شود. برای مثال:

```
void f(MyStruct &a){
    a.m = ...;
}
```

هنگام استفاده از رفرنس در نوع بازگشت توابع، توجه کنید رفرنس نباید به یک شیایی که با خروج از تابع از بین می‌رود، ارجاع داده شود. تابع زیر صحیح نیست:

```
int& returnRef(){
    int x = 10;
    return x; // error
}
```

مثال ۱۱) نمونه‌ای از استفاده‌ی صحیح رفرنس در بازگشت توابع:

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

int& getElement(int *a, size_t n) {
    return a[n];
}

int main () {
    int a[] = {1, 2, 3};
    getElement(a, 1) = 4;
    getElement(a, 2) = 5;
    for(size_t i = 0; i < 3; ++i ){
        cout << i << " : ";
        cout << a[i] << '\n';
    }
}
```

```
}
```

```
}
```

خروجی برنامه به صورت زیر است:

```
0 : 1
1 : 4
2 : 5
```

رفنس مانند اشارهگر میتواند از نوع **const** اعلان شود. رفنس **const** مقدار مورد ارجاع خود را نمیتواند تغییر دهد. برای مثال:

```
int i = 10;
int const& r = i;
```

رفنس **const** (به طور دقیقت، رفنس **lvalue** از نوع **const**) علاوه بر **rvalue** نیز میتواند ارجاع شود. مقداریست که **lvalue** نباشد (لیترالها، اشیای موقت و ...). **rvalue** بیانگر عبارتی است که توسط یک شی قابل شناسایی بیان نشده باشد. برای مثال:

```
int const& r = 5;
```

این نوع رفسنها، از آنجایی که از نوع **const** مقدار مورد ارجاع خود را نمیتوانند تغییر دهند. برای ارجاع به یک **rvalue**، علاوه بر رفنس **lvalue** از نوع **const**، از رفنس **rvalue** نیز میتوان استفاده کرد. رفنس **rvalue** با **&&** مشخص میشود (این اپراتور به معنی رفنس به رفنس نیست. رفنس به رفنس در C++ - برخلاف اشارهگر - وجود ندارد). برای مثال:

```
int &&r = 5;
```

تفاوت رفنس **rvalue** با رفنس **lvalue** از نوع **const** در این است که رفنس **rvalue** میتواند مقدار شی مورد ارجاع خود را تغییر دهد:

```
int const& r = 5;
int &&r2 = 5;
r2 = 7; // ok
```

```
r = 7; // error
```

تفاوت دیگر این رفنسها، هنگام قرارگیری آنها در پارامتر دوتابع سربارگذاری شده است. در این صورت که با پاس دادن `rvalue`، برنامه تابع با رفنس `rvalue` را اجرا میکند.

مثال (۱۲)

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

void f(const int &r) {
    cout << "f(const int&)" << '\n';
}

void f(int &&r) {
    cout << "f(int &&r)" << '\n';
}

int main() {
    int x = 4;
    f(x);
    f(7);
}
```

خروجی برنامه به صورت زیر است:

```
f(const int&)
f(int &&r)
```

توضیح) در این مثال بین یک متغیر و مقادیر موقت قابل تغییر (`rvalue`) تفاوت قائل میشود.

رابطه‌ی آرایه با اشاره‌گر

از نظر زیرساختی، آرایه یک نوع اشاره‌گر نیست؛ ولی یک آرایه را بدون کست کردن میتوان به اشاره‌گر تبدیل کرد. برای مثال:

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

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

اشارهگر p به اولین عنصر آرایه اشاره خواهد کرد؛ و سپس با استفاده از اپراتور [] نیز به عنصر مورد نظر دسترسی ایجاد می‌شود. این اپراتور به طور کلی مانند افزودن عدد صحیح n به اشارهگر و دیرفرنس کردنش به صورت *(p+n) عمل می‌کند. برای مثال، عبارت 4 زیر برابرند:

```
p[2]
a[2]
*(p + 2)
*(a + 2)
```

تفاوت اشارهگر p با آرایه a در این است که اشارهگر، اطلاعات مربوط به آرایه (نوع و سایز) a را از دست میدهد. برای مثال اپراتور **sizeof** دو عبارت یکسان نخواهد بود:

```
int a[] = {1,2,3,4};
int *p = a;
sizeof(p) == sizeof(a); // false
// sizeof(a) == 4 * sizeof(int)
// sizeof(p) == sizeof(int *)
```

و از نظر نوع، اشارهگر p دیگر نوع [4] int را نخواهد داشت؛ به همین دلیل هنگام پاس دادن آرایه به تابع، باید سایز آرایه نیز برای تابع مشخص شود (مثلاً با پاس دادن مستقیم به آن).

مثال (۱۳)

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

void printArray(int *a, size_t N) {
    for(int i = 0; i < N; ++i)
        cout << "[" << i << "] = " << a[i] << '\n';
}

int main() {
    int a[5] = {2,3,7,5,3};
```

```

    printArray(a, 5);
}

```

خروجی برنامه به صورت زیر است:

```

[0] = 2
[1] = 3
[2] = 7
[3] = 5
[4] = 3

```

در C++ (نه C) میتوان یک آرایه را با رفرنس ذخیره کرد. در این صورت سایز و نوع آرایه از بین نمیرود. برای مثال:

```

int a[5] = {2,3,7,5,3};
int (&b)[5] = a;
sizeof(a) == sizeof(b); // true

```

در نتیجه تابع مثال ۱۳ میتواند به صورت زیر تعریف شود:

```

void printArray(int (&a)[5]) {
    for(size_t i = 0; i < 5; ++i)
        cout << "[" << i << "] = " << a[i] << '\n';
}
printArray(a);

```

در این تابع یک آرایه فقط با سایز ۵ دریافت میشود؛ به این ترتیب از پاس دادن سایزهای غلط و بروز نتایج غلط جلوگیری میشود.

رشته

در برنامه نویسی یک رشته شامل مجموعه‌ای از کاراکترها است. برای مثال `name` یک رشته است و از کاراکترهای `n`, `a`, `m` و `e` تشکیل شده است. در زبان C برای ذخیره و کار با رشته‌ها از آرایه‌ای از کاراکترها استفاده می‌شود. در زبان C++ یک کلاس به نام `std::string` وجود دارد که کار با رشته (ذخیره و اعمال تغییر روی رشته) را آسانتر کرده است. به دلیل کاربرد زیاد رشته در C ابتدا با آن آشنا می‌شویم.

رشته در C

همانطور که گفته شد در زبان C برای ذخیره‌سازی و کار با رشته، باید از آرایه‌ای از کاراکترها استفاده کرد.

(۱) مثال

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

int main() {
    char myString[20];
    myString[0] = 'N';
    myString[1] = 'a';
    myString[2] = 'm';
    myString[3] = 'e';
    myString[4] = '\0';
    cout << myString << '\n';
}
```

خروجی برنامه به صورت زیر است:

Name

در مثال بالا یک آرایه‌ای از کاراکتر به اسم `myString` اعلام که به ترتیب عنصر صفر تا سوم آن با کاراکترهای کلمه‌ی `Name` و عنصر آخر نیز با کاراکتر `\0` مقداردهی

میشود. این کاراکتر خاص، کاراکتر `null` نام دارد و پایان رشته را در برنامه مشخص میکند. با مشخص شدن انتهای رشته با این کاراکتر، از بروز مشکلات حافظه جلوگیری خواهد شد. به دلیل نیاز به افزودن کاراکتر `null`، تعداد عناصر آرایه یکی از تعداد کاراکترهای رشته باید بیشتر باشد.

مقداردهی اولیه‌ی آرایه‌ی کاراکتر (مانند هر آرایه‌ای دیگر) میتواند با لیست {} صورت گیرد. برای مثال ۱ میتوان به صورت زیر عمل کرد:

```
char myString[20] = {'N', 'a', 'm', 'e', '\0'};
```

همچنین به جای قرار دادن جداگانه‌ی کاراکترها با ، کل رشته را بدون کاراکتر `null` در "" میتوان قرار داد. مانند:

```
char myString[20] = "Name";
```

عبارت "Name" یک لیترال رشته است و نوع آن در C++ از نوع آرایه‌ای از کاراکترهای **const** با تعداد عناصر مورد نیاز برای ذخیره‌ی کارکترهای آن (در دستور بالا ۵) است. بنابراین به صورت زیر (بدون تعیین عناصر) میتوان آرایه را مقداردهی اولیه کرد:

```
char myString[] = "Name";
myString[0] = 'F'; // ok
```

در C و استاندارهای قدیمیتر C++, دستور زیر نیز صحیح:

```
char *myString = "Name";
```

در این حالت به جای کپی شدن رشته در یک آرایه، یک اشاره‌گر به اول فضای لیترال اعلان خواهد شد. این دستور در استانداردهای جدید C++ غلط است^۱؛ زیرا در C، نوع لیترالهای رشته از نوع `char []` قرار دارند و میتوانند مستقیم به `*char` تبدیل شوند. در C++ لیترالها از نوع **const** میباشند. در نتیجه با تغییر مقدار چنین متغیری، برنامه سعی به تغییر یک **const** میکند:

```
char *myString = "Name";
```

^۱ جهت کمپایل شدن کدهای قدیمی، این دستور کمپایل میشود ولی هشدار کمپایلر را به همراه خواهد داشت.

```
myString[0] = 'F'; // error. assignment to const
```

در هدیر `<cstring>` یک سری توابع برای کارکردن با رشته‌ها وجود دارد که با افزودن این هدیر در برنامه در دسترس قرار می‌گیرد.
تعدادی از توابع مهم `<cstring>` با کاربردشان در جدول زیر قرار دارند:

<code>strcpy(s1, s2)</code>	<code>s2</code> را در <code>s1</code> کپی می‌کند.
<code>strcat(s1, s2)</code>	<code>s2</code> را به انتهای <code>s1</code> می‌چسباند.
<code>strlen(s1)</code>	طول رشته‌ی <code>s1</code> را برابر می‌گرداند.
<code>strcmp(s1, s2)</code>	تابعی است که در صورت برابر بودن <code>s1</code> و <code>s2</code> عدد 0 و در صورتی که اولین کarakتر غیر یکسان در <code>s1</code> مقدار کمتری داشته باشد، عددی کمتر از 0 و در غیر این صورت عددی بیشتر از 0 را بر می‌گرداند.

مثال ۲) برنامه‌ی زیر کاربرد تابع `strcat` را نشان می‌دهد:

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

int main() {
    char s1[] = "Computer";
    char s2[] = "Science";
    strcat(s1, s2);
    cout << "s1 is now: " << s1;
}
```

خروجی برنامه به صورت زیر است:

```
s1 is now: ComputerScience
```

در برنامه طول عمر لیترالهای رشته از نوع `static` است و تا پایان برنامه باقی میماند.
به همین دلیل میتوانند بدون مشکل از توابع بازگردند:

```
const char *return_str(){
    return "a string";
}
```

کلاس std::string

در C++ یک نوع داده مختص کار با رشته به نام `std::string` وجود دارد. برای استفاده از نوع `std::string` ابتدا باید `string` به برنامه افزوده شود.
اعلان متغیر از `std::string` نیز مانند بقیه متغیرها صورت میگیرد:

```
#include <string>
std::string stringName;
```

مقداردهی اولیه شی `string` میتواند به روشهای زیر صورت بگیرد:

```
std::string s{"a string"};
std::string s = {"a string"};
std::string s("a string");
std::string s = "a string";
```

بعد از مقداردهی اولیه متغیر `string`, با `=` میتوان آن را با رشتهای دیگر مقداردهی کرد.

مثال (۳)

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

void printString(const string& s) {
    cout << "Passed string: " << s << '\n';
}

int main() {
    string s = "a string";
    printString(s);
}
```

تبدیل عدد به رشته و برعکس

تبدیل عدد به رشته با تابع `to_string()` صورت می‌گیرد:

```
string a = to_string(59);
```

که 59 به رشته تبدیل می‌شود.

برای تبدیل رشته به عدد، تابع `stoi` وجود دارد:

```
int a = stoi("44");
```

به طور مختص برای تبدیل رشته به `double`, `float`, `...`, توابع، `stod(str)` و `stof(str)` وجود دارند.

تابع عضو `()at()` و `()size()`

تابع `()at()` با پاس دادن عدد `n`, کاراکتر `\n` از متغیر `string` را برمی‌گرداند. علاوه بر `()at()` از اپراتور `[]` نیز مانند آرایه برای دسترسی به کاراکترهای متغیر `string` میتوان استفاده کرد؛ با این تفاوت که `[]`, برخلاف `()at()`, موجود بودن عنصر را بررسی نمیکند.

تابع `()size()` نیز تعداد کل کاراکترهای رشته را (از نوع `size_t`) برمی‌گرداند.

(۴) مثال

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

int main() {
    string s = "A sentence";
    for(size_t i = 0; i < s.size(); ++i) {
        cout << s.at(i) << ' ';
    }
    cout << '\n';
}
```

خروجی برنامه به صورت زیر است:

```
A   s   e   n   t   e   n   c   e
```

گرفتن کل خط با `()getline`

اگر از `cin` برای گرفتن رشته از کنسول استفاده شود، برنامه، رشته را تا قبل از اولین فضا خالی `space` و `tab` دریافت می‌کند. برای دریافت کل خط باید از `()getline` استفاده شود.

مثال (۵)

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

int main() {
    string s;
    cout << "Enter a sentence: ";
    getline(cin, s);
    cout << "You have entered : " << s << '\n';
}
```

خروجی برنامه به صورت زیر است:

```
Enter a sentence: a sentence
You have entered : a sentence
```

متغیر `string` شامل توابع عضو دیگری مثل `replace()`, `find()` و ... نیز هست. در صورت نیاز به رفرنس C++ مراجعه کنید.

کلاس (۱) (کلاس‌ها و اشیا^۱)

یک کلاس شامل مجموعه‌های از متغیرها و توابع است. با تعریف یک کلاس مانند struct می‌توان یک نوع داده جدید در برنامه ایجاد کرد.

برای تعریف یک کلاس در C++ از فرمت زیر استفاده می‌شود:

```
class id {  
    // body  
} optional_obj;
```

id نام کلاس و optional_obj، اختیاراً محل اعلان متغیر از کلاس (تفکیک شده با کاما) است. تعریف متغیر جداگانه از کلاس id به شکل زیر صورت می‌گیرد:

```
id m;  
    پس از اعلان متغیر m، دسترسی به اعضای آن به صورت زیر انجام می‌شود:
```

```
m.member
```

نقطه (.) اپراتور انتخاب عضو^۲ نام دارد.

توابع و متغیرهای عضو یک کلاس می‌توانند یکی از سه نوع سطح دسترسی زیر را داشته باشند:

Private	دسترسی به این اعضا فقط توسط اعضای دیگر کلاس می‌تواند صورت گیرد.
Protected	این اعضا علاوه بر درون کلاس، در کلاس‌های مشتق ^۳ شده از آن کلاس نیز قابل دسترساند.
Public	این اعضا علاوه بر درون کلاس و کلاس مشتق، در کل برنامه (تا جایی که شی آن کلاس تعریف شده باشد) قابل دسترس است.

^۱. Classes and objects

^۲. Member access operator

^۳. مربوط به وراثت است.

تعیین کننده‌ی سطح دسترس^۴ نام دارند.

مثال ۱) برنامه‌ای که تعریف و استفاده از یک کلاس ساده را نشان میدهد.

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

class MyClass {
public:
    void f() {
        cout << "f() called.\n";
    }
};

int main(){
    MyClass m;
    m.f();
}
```

خروجی برنامه به صورت زیر است:

```
f() called.
```

توضیح) در این مثال، تعریف یک کلاس ساده به اسم `MyClass` نشان داده شده. در تابع `main`، از کلاس `MyClass` یک متغیر به اسم `m` اعلان می‌شود و به دلیل اینکه تابع `f()` از کلاس `MyClass` با سطح دسترس `public` تعریف شده، در تابع `main` توسط متغیر `m` قابل فراخوانی می‌باشد.

مثال ۲

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

class Person {
private:
    string name;
```

^۴. Access specifiers

```

public:
    void set_name(string s) {
        name = s;
    }
    void print_name() {
        cout << "Name: " << name << '\n';
    }
};

int main() {
    Person p, p2;
    p.set_name("Dennis");
    p2.set_name("Ken");
    p.print_name();
    p2.print_name();
}

```

خروجی برنامه به صورت زیر است:

```

Name: Dennis
Name: Ken

```

توضیح) در این برنامه یک کلاس به نام **Person** برای ذخیره اسامی افراد در برنامه تعریف کردیم. این کلاس شامل یک متغیر **name** از نوع **string** به صورت **public** و دو تابع **set_name** برای مقداردهی متغیر **name** و نمایش آن در خروجی است. در تابع **main** نیز دو متغیر از **Person** اعلان میشود و متغیر **name** آنها نیز جداگانه مقداردهی و نمایش مییابند.

از یک کلاس جهت ایجاد یک نوع داده‌ی جدید در برنامه استفاده میشود. برای مثال اگر در برنامه اگر نیاز به ذخیره و کارکردن با ماتریسها شود، می‌توان یک کلاس برای کار با ماتریسها تعریف کرد به طوری که شامل یک آرایه‌ی دو بعدی برای ذخیره‌ی درایه‌های ماتریس و همچنین یک سری توابع برای ترانهادن، معکوس کردن و ... ماتریس باشد. در این صورت با فراخوانی این توابع، فرآیند مناسب روی آرایه‌ی دو بعدی اعمال می‌شود. در

نتیجه با متغیرهای ساخته شده از این کلاس میتوان با دادههایی که بیانگر ماتریس در برنامه هست کار کرد.

نکته‌ی ۱: به متغیرهای کلاس، متغیرهای عضو^۱ و به توابع کلاس، توابع عضو^۲ گفته می‌شود.

نکته‌ی ۲: اگر تعیین کننده دسترس در یک کلاس نوشته نشود، کمپایلر به طور پیش فرض اعضاء را **private** در نظر میگیرد. به عنوان مثال:

```
class A{
    int x;
}
```

نکته‌ی ۳: به متغیرهای اعلام شده از یک کلاس، شی نیز میگویند.

فرق کلاس با **struct**

در C++ تفاوت **struct** با کلاس در سطح دسترس پیش فرض اعضاء است. به این صورت که اعضاء **struct** به طور پیش فرض **public** و اعضاء کلاس از نوع **private** تعیین می‌شوند. با این وجود هم اعضاء **struct** و هم اعضاء کلاس را می‌توان با هر نوع سطح دسترس (**private**, **public**, **protected**) قرار داد. برای مثال دو تعریف زیر با هم برابرند:

```
class A {
public:
    int x;
};

class B{
    int x;
};
```

^۱. Member variable

^۲. Member function

از C++ در **struct** معمولاً برای تعریف انواع داده‌های استفاده می‌شود که فقط مشکل از یکسری متغیرهای عضواند (مانند تعریف **struct** هایی که در فصل متغیر و انواع داده (۳) بررسی کردیم). در این انواع داده‌ها، علاوه بر اعلان نکردن توابع، از ویژگی‌های پیشرفته‌ی شی‌گرایی – که در ادامه می‌پردازیم – نیز استفاده نمی‌کنند. مانند:

```
struct House {
    int house_number;
    int postal_code;
    std::string street_name;
};
```

سازنده^۱

سازنده یک تابع بدون مقدار بازگشتی و همنام با نام کلاس است که با ساخت هر شی از کلاس اجرا می‌شود.

مثال (۳) برنامه‌ای که اجرای سازنده را با ساخت شی نشان می‌دهد.

```
#include <iostream>
using namespace std;
class MyClass {
public:
    MyClass() {
        cout << "Constructor called." << '\n';
    }
};

int main() {
    MyClass m;
}
```

خروجی برنامه به صورت زیر است:

```
Constructor called.
```

نکته: در صورتی که سازنده نوشته نشود کمپایلر به طور پیش فرض سازنده‌ای را با بدنی

ctor یا به اختصار constructor^۱

حالی در نظر می‌گیرد و هبیج دستوری اجرا نمی‌شود (مانند مثال ۱ و ۲). همچنین سازندهای که پارامتری نداشته باشد و یا همه‌ی پارامترهایش مقدار پیش فرض داشته باشد، سازندهی پیش فرض^۱ نام دارد. در مثال ۳ از این نوع سازنده استفاده می‌شود.

هدف از سازنده، مقداردهی و اجرا فرآیندهایی است که با ساخت شی نیاز به انجاماند. این فرآیندها شامل تخصیص حافظه‌ی دینامیک (در صورت نیاز)، باز کردن فایل، برقراری ارتباط با دیتابیس و ... می‌شود.

سازنده با پارامتر

اگر سازندهای نیاز به پارامتر داشته باشد، مانند هر تابع دیگر می‌توان برای آن لیست پارامتر قرار داد. در ساخت شی نیز، آرگومان‌های مناسب باید به سازنده پاس داده شوند. مثال ۴) برنامه‌ای که با سازنده، متغیرهای عضو آن مقداردهی می‌شوند:

```
#include <iostream>
using namespace std;
class MyClass {
    int x, y;
public:
    MyClass(int a, int b) {
        x = a;
        y = b;
    }
    void printValues() {
        cout << x << ' ' << y << '\n';
    }
};

int main() {
    MyClass m(2,3);
    m.printValues();
```

^۱. Default constructor

{

خروجی برنامه به صورت زیر است:

2 3

۱ مخرب

مخرب برخلاف سازنده تابعی است که با از بین رفتن شی اجرا می‌شود. نام این تابع، هم نام با نام کلاس و با یک آکولاد (~) در اول آن است. با نوشته نشدن مخرب، کمپایلر مانند سازنده نیز یک مخرب با بدنه‌ی خالی در نظر می‌گیرد. یکی از کاربردهای مخرب، آزاد کردن منابعی از سیستم است که به طور خودکار آزاد نمی‌شود؛ مثل `delete` کردن فضاهای تخصیص یافته به صورت دینامیک. مخرب همچنین وظیفه‌ی لاغ (log) گرفتن از آزاد شدن فضای شی (در صورت نیاز) و یا به طوری کلی هر فرآیندی که با از بین رفتن شی نیاز به انجام است را شامل می‌شود.

(۵) مثال

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

class MyClass {
public:
    int *arr;
    size_t length;
    MyClass(int *a, size_t s){
        length = s;
        arr = new int[s];
        for(size_t i = 0; i < s; ++i) {
            arr[i] = a[i];
        }
    }
    void printArray() {
        for(size_t i = 0; i < length; ++i) {
```

¹ به اختصار destructor یا dtor

```

        cout << arr[i] << ' ';
    }
}
~MyClass(){
    delete [] arr;
}
};

int main() {
    int a[5] = {2,3,6,7,3};
    MyClass c(a, 5);
    c.printArray();
}

```

خروجی برنامه به صورت زیر است:

2 3 6 7 3

همچنین منظور از آزاد کردن منابع، فقط آزاد کردن حافظه‌های دینامیک نیست؛ فایل و سوکت باز شده، نخ (thread)، رابط دیتابیس و ... همگی از منابع سیستماند. کلاس‌ها در C++ پایه‌ی برنامه نویسی شی‌گرایاند. هدف از شیگرایی، صرفاً دسته بندی یک سری توابع و متغیرها نیست بلکه ایجاد نوع داده و پیاده کردن مفاهیم کپسول کردن، وراثت، چندریختگی و انتزاع است که در ادامه به هدف و کاربرد هر کدام میپردازیم. یک کلاس علاوه بر متغیر و تابع، شامل `typedef` و `enum` نیز میتواند باشد.

کردن یک کلاس `forward declare`

اگر تعریف یک کلاس بعد از تعریف یک تابع قرار بگیرد و در پارامتر تابع از نام کلاس استفاده شود، مانند:

```

void f(A &a){
}

```

```
class A{  
};  
  
int main() {  
}
```

کمپایل این دستورات خطای مشابه زیر را ایجاد خواهد کرد:

```
'A' was not declared in this scope
```

زیرا تعریف A بعد از تعریف تابع نوشته شده و تابع از وجودش مطلع نمیشود. برای جلوگیری از این خطا (علاوه بر تعریف کردن A قبل از تابع)، میتوان کلاس A را بعد از تابع تعریف کرد ولی اعلان آن را قبل از تابع نوشت. به صورت زیر:

```
class A;  
void f(A &a){  
  
}  
class A{  
};  
  
int main() {  
}
```

به این کار forward declare کردن گفته میشود. کلاس A در این حالت اصطلاحاً یک نوع ناقص¹ است. اعلان توابع میتوانند انواع ناقص دریافت کنند و بازگردانند. مانند:

```
void f(A a);  
  
ولی تعریف توابع فقط میتوانند اشارهگر یا رفرنس از انواع ناقص دریافت کنند و  
بازگردانند. مانند:  
  
void f(A &a){  
}
```

¹. Incomplete type

در آخر این فصل به این نکته میپردازیم که اشیای کلاس‌هایی که تاکنون تعریف کردیم به طور خودکار میتوانند با سازنده یا اپراتور = در اشیای دیگر کلاس کپی شوند. برای مثال:

```
class MyClass{
    int x;
public:
    MyClass(int a){
        x = a;
    }
    void print_x() { cout << x << '\n';}
};

MyClass m(4);
MyClass m2(m);
m2.print_x(); // 4
```

کلاس (۲) (کپسول کردن و لیست انتساب دهنده‌ی عضو)

اگر در برنامه، یک کلاس به نام Person برای ذخیره‌ی اطلاعات افراد داشته باشیم، مثل:

```
#include <string>
class Person{
public:
    std::string Name;
    // ...
};
```

به دلیل public بودن متغیر عضو Name، متغیرهای ساخته شده از این کلاس میتوانند Name را با هر رشته‌ای مقداردهی کنند. برای مثال:

```
Person p;
p.Name = "Dennis01@";
```

با فرض اینکه اسم اشخاص در برنامه نباید شامل عدد، سمبلهای @، \$ و ... و به طور کلی کاراکترهای غیر از حروف الفبا باشد، میتوان متغیر عضو Name را از نوع private تعیین کرد و دوتابع عضو public برای مقداردهی و مقدارگیری این متغیر قرار داد. در این حالت تابعی که برای مقداردهی نوشته شده عدم دریافت یک رشته‌ای که شامل کاراکترهای غیر از الفبا است را میتواند بررسی کند (در مثال زیر، کلاس Person از regex برای این بررسی استفاده میکند. هدف این مثال آشنایی با یک مثال عملی است):

```
#include <string>
#include <regex>
class Person{
private:
    std::string Name;
public:
    void setName(const std::string& s) {
        if(std::regex_match(s,std::regex("^[A-Za-z]+$")))
            Name = s;
    else
```

```

        cout << "[error] Invalid string\n";
    }
    std::string getName(){
        return Name;
    }
};

```

در این صورت استفاده از این کلاس به شکل زیر صورت می‌گیرد:

```

Person p;
p.setName("Dennis");
cout << p.getName() << '\n'; // Dennis

```

در صورت سعی به مقداردهی Name با رشته‌ای که شامل کاراکترهای غیر از الفبا است، یک پیغام خطا در خروجی نمایش می‌یابد و به دنبال آن Name مقداردهی نخواهد شد.

مثل:

```

Person p;
p.setName("@das2"); // [error] invalid string

```

اگر این کلاس، شامل یک متغیر عضو دیگر به اسم age از نوع int برای ذخیره کردن سن افراد باشد، در توابعی با نامهای getAge و setAge می‌توان مقدارگیری و مقداردهی آنها را انجام داد؛ به طوری که در setAge با شرط اینکه age فقط می‌تواند با مقادیر بین ۱ و ۱۰۰ مقداردهی شود، از مقداردهی با مقادیر بیشتر و یا کمتر از این بازه جلوگیری شود. برای مثال:

```

class Person {
private:
    int age;
//...
public:
//...
void setAge(int a) {
    if(a < 1 || a > 100) {
        cout << "[error] invalid age";
    } else {

```

```
    age = a;  
}  
}  
  
int getAge(){  
    return age;  
}  
}
```

در کلاس Person با قرار دادن name و age به صورت private و قرار دادن توابعی برای مقداردهی (setName و setAge) و مقدارگیری (getName و getAge) نمونهای از کپسول کردن است. به این ترتیب در این کلاس، از مقداردهی متغیرهای عضو با مقادیر غیر قابل قبول جلوگیری خواهد شد. کپسول کردن در C++ با قرارگیری اعضا به صورت protected، public و private، پیاده میشود؛ در نتیجه رابطها (interface) و پیاده سازیهای (implementation) یک کلاس را میتوان از هم جدا نمود.

در کپسول کردن به توابع عضوی که برای مقداردهی تعریف میشوند (مثل setName و setAge) توابع setter و توابع عضوی که مقدارگیری با آنها صورت میگیرد (مثل getName و getAge) توابع getter میگویند.

به عنوان مثالی دیگر از کپسول کردن کلاسی است که برای ذخیره و کار با اطلاعات بانکی ایجاد شده است. در این کلاس برای برداشت پول از حساب، با استفاده از توابع عضو و استفاده از تکنیک کپسول کردن میتوان شرط اینکه پولی که میخواهد برداشت شود از مقدار پول موجود در حساب بیشتر نباشد را بررسی کرد.

کپسول کردن برای مخفی کردن دادهها و توابع و به طور کلی کارکرد درونی یک کلاس (private) کردن آنها) و قرار دادن یک سری رابطهای ثابت (از نوع public) برای کار با این اعضا استفاده میشود. نتیجه‌ی این الگو این است که با بزرگتر و پیچیده‌تر شدن کلاس و تغییر عملکرد درونی آن تا وقتی که واسطه‌ها ثابت بمانند، برنامه‌هایی که از

کلاس استفاده میکنند نیازی به تغییر ندارند. کپسول کردن همچنین از تغییر غیر عادی داده‌های اشیا - همانطور که گفته شد - جلوگیری میکند.

در صورت عدم استفاده از الگوی یاد شده، برنامه بدون خطای کمپایل می‌شود ولی در پروژه‌های بزرگ مشکلاتی در جهت مدیریت و استفاده از کلاس بوجود می‌آید.

لیست انتساب دهنده اولیه‌ی عضو^۱

در گذشته برای مقداردهی اولیه‌ی متغیرهای عضو از بدنی سازنده استفاده میکردیم. مانند:

```
class MyClass {
private:
    int a, b;
public:
    MyClass(int x1, int x2){
        a = x1;
        b = x2;
    }
};
```

و با پاس دادن مقادیر `x1` و `a` و `b` مقداردهی میشند. اگرچه برنامه قابل کمپایل بود، ولی شیوه‌ی صحیح مقداردهی در C++ نیست. برای مثال، ثوابت و رفرنس‌ها باید هنگام اعلان مقداردهی شوند. مانند:

```
int const b = 3;
```

و در صورت عدم مقداردهی اولیه، کمپایلر پیغام خطای صادر خواهد کرد. مانند:

```
int const b;
b = 3; // error
```

به همین طریق اگر متغیرهای عضو از نوع ثابت باشند، هنگام مقداردهی به روش گذشته ممکن نیست: زیرا بدنی سازنده پس از اعلان متغیر اجرا می‌شود. مانند:

¹. Member initializer list

```
class MyClass{
private:
    int const a;
    int &b;
public:
    MyClass(int x1, int x2){
        a = x1; // error
        b = x2; // error
    }
};
```

برای مقداردهی ثوابت و همچنین تمام داده‌های دیگر، می‌توان از لیست انتساب دهنده‌ی اولیه‌ی عضو استفاده کرد. این لیست بین اعلان و تعریف سازنده قرار می‌کیرد. به شکل زیر:

```
class MyClass {
    int x;
    int y;
    // ...
public:
    MyClass(a, b):x{a}, y{b}, ...
    {}
};
```

با این روش، متغیرهای عضو در زمان اعلان مقداردهی اولیه می‌شوند.
مثال (۱)

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

class MyClass{
private:
    int a;
    const int b;
    int &c;
public:
    MyClass(int x1, int x2, int x3):a{x1}, b{x2}, c{x3},
```

```
c{x3} {
    cout << "Normal var : " << a << '\n';
    cout << "Const var : " << b << '\n';
    cout << "Ref var : " << c << '\n';
}
};

int main() {
    int i = 10;
    MyClass S0(1,7,i);
}
```

خروجی برنامه به صورت زیر است:

```
Normal var : 1
Const var : 7
Ref var : 10
```

چند نکته در مورد لیست انتساب دهنده عضو:

۱. به جای استفاده از {} برای مقداردهی، از () میتوان استفاده کرد.
۲. ترتیب مقداردهی متغیرهای عضو معمولی^۱، به ترتیب اعلان آنها در کلاس است.

برای مثال تعریف کلاس زیر یک تعریف صحیح است:

```
class MyClass{
    int a;
    int b;
public:
    MyClass(int x): b{a}, a{x} {
        cout << "b=" << b << " a=" << a << '\n';
    };
};

MyClass M0(6);
```

و b بعد از a مقداردهی می‌شود.

^۱ منظور از معمولی، اعضای غیر static است. تاکنون تمام مقداردهی که در کلاس‌ها و اشیا کارکردیم از نوع غیر static بودند.

۳. زمانی که مقداردهی اولیه‌ی متغیر عضو توسط لیست انتساب دهنده انجام شود، سازنده مناسب از آن اجرا خواهد شد. در صورت استفاده نکردن از این لیست، سازنده‌ی پیش فرض عضو اجرا می‌شود. برای مثال:

```
class A {
private:
    int x;
public:
    A() { x = 0; }
    A(int v) { x = v; }
};

class B {
private:
    A a;
public:
    B(int v):a{v} { // A(int v) is called
    }
};

B b(10);
```

ساخت متغیر با سازنده‌ی پیش فرض و سپس مقداردهی آن، در تعریف شی از `int` و اکثر انواع دیگر تاثیری در عملکرد و یا بازده (به صورت چشمگیر) ندارد ولی توصیه می‌شود برای مقداردهی اولیه تمام اعضای کلاس از این لیست استفاده کنید. به این ترتیب بدنی سازنده فقط دستورات مورد نیاز غیر از مقداردهی اولیه‌ی عضوها را انجام خواهد داد.

۴. در لیست انتساب‌دهنده‌ی اولیه، اسمهای درون `{ }` یا `()` در ناحیه‌ی سازندهای که این لیست مشخص شده ارزیابی می‌شوند. در نتیجه نام متغیر عضو با پارامتر سازنده، می‌تواند یکسان باشد. برای مثال:

```
class MyClass{
int a;
```

```
public:
    MyClass(int a): a{a} // ok
    {}
};


```

سازنده‌ی delegating

یک سازنده میتواند در لیست انتساب دهنده‌ی خود یک سازنده‌ی دیگر را از کلاس اجرا کند. به چنین سازنده‌های، سازنده‌ی delegating گفته میشود و به سازنده‌های که توسط این نوع سازنده اجرا میشود، سازنده‌ی target میگویند.

برای مثال:

```
class MyClass {
    int i;
public:
    MyClass(int i, char c): i{i} {}
    MyClass(): MyClass(1, 'a') {}
};


```

در سازنده‌ی delegating، لیست انتساب دهنده فقط میتواند شامل سازنده‌ی target باشد؛ و افزودن عناصر دیگر خطای کمپایلر را به همراه دارد. دلیل این نوع عملکرد این است که انتظار میروود مقداردهی اولیه‌ی شی به طور کامل در سازنده‌ی target صورت گیرد. همچنین برنامه پس از اجرای سازنده‌ی target، به اجرای بدنی سازنده‌ی delegating میپردازد تا در صورت نیاز یکسری دستورات نیز در آنجا اجرا شود.

قابلیت سازنده‌ی delegating از C++11 افروده شده؛ به این ترتیب از تکرار کد در سازنده‌های مختلف و یا لزوم ایجاد یک تابع عضو (مثلاً با نام `init`)، فقط برای استفاده از آن در سازنده‌ها جلوگیری میشود. استفاده از یک تابع عضو جداگانه (مثل `init`

علاوه بر پیچیده کردن کلاس، خارج از سازنده‌ها نیز میتواند فراخوانی شود. به این ترتیب به جای نوشتتن کلاس زیر:

```
class MyClass {  
private:  
    void init(double x, double y) {  
        // init  
    }  
public:  
    MyClass() {  
        init(0,0);  
    }  
  
    MyClass(double x) {  
        init(x,0);  
    }  
};
```

به صورت زیر میتواند بازنویسی شود:

```
class MyClass {  
public:  
    MyClass (double x, double y) // : (mem-  
initializer) {  
        // init  
    }  
    MyClass(): MyClass(0, 0) {  
    }  
  
    MyClass (double x): MyClass(x, 0) {  
    }  
};
```

کلاس (۳) (ادامه‌ی مباحث پایه‌ای کلاسها)

پاس دادن اشیا به توابع

به دلیل اینکه اشیای کلاسها معمولاً حجم زیادی در حافظه اشغال می‌کنند، پاس دادن آن‌ها به طور عادی و کپی شدن فضای آنها در اکثر موقع مناسب نیست. اشیای یک کلاس مانند هر شی دیگر میتوانند با اشاره‌گر یا رفرنس در تابع دریافت شوند. برای مثال:

void f (MyClass &x);

یا:

void f (MyClass *x);

همچنین در صورتی که اشیای پاس داده شده در تابع نباید تغییر کنند، پارامتر باید از نوع اشاره‌گر یا رفرنس به ثابت اعلان شود. مانند:

void f(const MyClass &x);

void f(const MyClass *x);

به طور کلی زمانی از اشیا از نوع عادی به تابع پاس داده میشوند که تابع بخواهد از شی کپی ایجاد و روی کپی ایجاد شده تغییر ایجاد کند.

توجه شود در صورتی که شی از طریق اشاره‌گر در تابع دریافت شده، دسترسی به اعضای شی به جای:

(*obj).member

میتواند با اپراتور انتخاب عضو **->** به صورت زیر صورت گیرد:

obj->member

تعريف عضو خارج از کلاس

برای این کار باید اعلان عضو باید در کلاس نوشته شود و سپس تعریف آن با اپراتور تفکیک ناحیه (**::**)، خارج از کلاس قرار گیرد. مثال:

class Class1{

```
public:
    void f ();
};

void Class1::f (){
    // statement(s)
}
```

شارهگر this

شی هر کلاس، struct یا union شامل یک اشارهگر به نام **this** است که به خود شی اشاره می‌کند. در C++ از **this** معمولاً در کلاسها فقط استفاده می‌شود. برای مثال:

```
class MyClass{
    int a;
public:
    MyClass(int a): a{a} {}
    void print_a(){ cout << this->a << '\n'; }
};
```

```
MyClass m(10);
m.print_a();
```

۱۰ را در خروجی نمایش میدهد. البته در این مثال نوشتن **a** بدون **this** نیز عملکرد مشابهی ایجاد می‌کند. با این حال به طور معمول در دو جا قرار دادن **this** کاربرد دارد:

۱. در رفع ابهام زمانی که یک متغیر یا پارامتر تابع عضو با متغیر عضو کلاس هم نام باشد. مثلاً:

```
class MyClass{
    int a;
public:
    MyClass(int a): a{a} {}
    void f() {
        int a = 10;
        cout << a << ' ' << this->a << '\n';
    }
}
```

```
};
```

```
MyClass m(5);
m.f(); // 10 5
```

که `a` به متغیر عضو و `this->a` به متغیر اعلان شده در `f()` تلقی می‌شود.
۲. زمانی که نیاز به بازگردانی شی از خود کلاس است. برای مثال:

```
MyClass& MyClass::f(){
    return *this;
}
```

با فراخوانی این تابع عضو، شی `MyClass` از نوع رفرنس برمی‌گردد.

^۱توابع عضو `const`

در کلاس‌ها یکسری از توابع عضو را می‌توان به صورت `const` تعریف کرد. توابع عضوی که از نوع `const` تعیین شده‌اند نمی‌توانند مقدار متغیرهای عضو کلاس را تغییر دهند. به عنوان نمونه تابع `const` `getter` را معمولاً از نوع `const` تعیین می‌کنند. فرمت تعریف تابع `const` به صورت زیر است:

```
class id{
access_modifier:
    returnType funcName(parm_list) const
{
    // body
}
};
```

توجه شود که واژه‌ی `const` بعد از نام تابع، هم در اعلان و هم در تعریف (در صورت جدا بودن) باید قرار گیرد.

توابع عضو `const` برخلاف توابع عضو غیر `const` توسط متغیرهای `const` از کلاس نیز می‌توانند فراخوانی شوند.

^۱. Const member functions

مثال ۱)

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

class MyClass {
    int x;
public:
    MyClass(int x): x(x) {}
    int get_x() const { return x; }
};

void f(const MyClass &m){
    cout << m.get_x() << '\n'; // ok
}

int main() {
    MyClass m(4);
    f(m);
}
```

خروجی برنامه به صورت زیر است:

4

دو تابع در کلاس را نیز میتوان طوری سربارگذاری کرد که یکی از آنها از نوع `const` و دیگری غیر `const` باشد. در این صورت شی `const`, تابع عضو `const` و شی غیر `const`, تابع غیر `const` را فراخوانی خواهد کرد.

برای مثال:

```
class MyClass {
public:
    void f() const { cout << "f\n"; }
    void f() { cout << "f const\n"; }
};

MyClass m;
```

```
const MyClass m2;

m.f(); // f
m2.f(); // f const
```

متغیر عضو **mutable**

متغیرهای عضو غیر **static** و غیر **const** یک کلاس می‌توانند به صورت **mutable** تعریف شوند. مقادیر متغیرهای **mutable**، توسط تابع عضو **const** می‌توانند تغییر کنند.

تعریف این گونه متغیر با واژه‌ی **mutable** صورت می‌گیرد و به هر دو صورت زیر صحیح است:

```
mutable data_type member_variable;
data_type mutable member_variable;
```

مثال (۲)

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

class MyClass {
    mutable int a;
public:
    MyClass(int a): a{a} {}
    void f() const {
        a++; // ok
    }
    int get_a() const { return a; }
};

int main() {
    MyClass m(10);
    m.f();
    cout << m.get_a() << '\n';
}
```

خروجی برنامه به صورت زیر است:

11

استفاده از {} در مقداردهی اولیهی شی

در گذشته دیدیم برای مقداردهی متغیر انواع موجود مثل int، میتوان از = ()، {} و {} استفاده کرد. برای مثال تمام دستورات زیر صحیح و برابرند:

```
int a = 10;
int a(10);
int a{10};
int a = {10};
```

در این قسمت، بیشتر به تفاوت {} با () و = میپردازیم زیرا در اکثر مکانها عملکرد {} با {} یکی است.

همانگونه که پیشتر گفته شد یکی از امتیازات {} در مقداردهی متغیرها از انواع اساسی مثل int، جلوگیری از تبدیل narrow است. برای مثال در دستور زیر:

```
int a = 1.5; // a = 1
```

کمپایلر بدون خطا دستور را کمپایل میکند و a را با 1 مقداردهی میکند. هنگام استفاده از {}, کمپایلر در تبدیل narrow خطا صادر میکند. برای مثال:

```
int a{1.5}; // error!
```

کاربرد دیگر {} در مقداردهی اعضای غیر static یک کلاس (یا struct) است. در C98++، مقداردهی اولیهی این اعضا به طور مستقیم فقط با = ممکن بود. برای مثال:

```
class MyClass{
private:
    int a = 1;
    int b(1); // error
};
```

اکنون با {} نیز میتوان این کار را انجام داد:

```
class MyClass{
```

```
private:
    int b{10};
};
```

یکی از دلایل اصلی اضافه شدن {} به C++, یکسان کردن چند syntax جداگانه برای مقداردهیهای مختلف، در عین حال ارتقای کیفیت کد (با جلوگیری از تبدیل narrow است.

از {} در مقداردهی اولیه‌ی شی کلاس (یا struct) نیز (علاوه بر {}) میتوان استفاده کرد.

مثال ۳

```
#include <iostream>
using namespace std;
class MyClass{
public:
    MyClass(int a, int b){
        cout << a << '\t' << b << endl;
    }
};

int main() {
    MyClass M01(2,3);
    MyClass M02{3,4};
}
```

خروجی برنامه به صورت زیر است:

2	3
3	4

به طور کلی مقادیر {} را در std::initializer_list میتوان ذخیره کرد.
برای مثال:

```
std::initializer_list<int> il = {1,2,3,4};
```

موجود **<initializer_list>** در همین std::initializer_list

است. با استفاده از شی `std::initializer_list`، تعداد دلخواهی آرگومان (از یک نوع) را میتوان به توابع پاس داد. پس از دریافت این شی در توابع، از اعضای `end`، `size` و `begin` برای ایجاد حلقه در لیست میتوان استفاده کرد. مقدار بازگشتی این اعضا، با نوع `std::initializer_list<T>::iterator` قابل ذخیره است.

استفاده از متغیرهای ایتریتور، مانند اشارهگر صورت میگیرد (با `*` برای دیفرنس و ...).

مثال (۴)

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

void f(initializer_list<int> il){
    for(initializer_list<int>::iterator p = il.begin();
        p!=il.end(); ++p) {
        cout << *p << '\t';
    }
    cout << endl;

    for(auto const& i : il) {
        cout << i << '\t';
    }
    cout << endl;
}

int main() {
    f({1,2,3,4,5});
}
```

خروجی برنامه به صورت زیر است:

1	2	3	4	5
1	2	3	4	5

توضیح) در دستورات `for`، برای آشنایی با نحوه دریافت، از اسم کامل نوع ایتریتور

استفاده کردیم؛ در هر دو `for` از واژهٔ `auto` برای اعلان متغیر میتوان استفاده کرد. علاوه بر توابع، در پارامتر سازنده نیز میتواند نوشته شود. زمانی که یک سازنده با پارامتر `std::initializer_list` و سازندهای دیگر با پارامتر عادی موجود باشند، با قرارگیری `{}` در مقداردهی اولیه‌ی شی، برنامه، سازنده با پارامتر `std::initializer_list` را اجرا میکند. برای مثال:

```
class MyClass{
public:
    MyClass(int a){
        cout << "int a\n";
    }
    MyClass(initializer_list<int> li){
        cout << "initializer_list\n";
    }
};

MyClass M0{1};
```

برنامه خروجی "initializer_list" را خواهد داشت.
انتخاب سازنده با پارامتر از نوع `initializer_list` بین چند سازنده، به گونه‌ای است که هر نوع فراخوانی با `{}` که امکان اجرای آن را ممکن سازد (حتی با تبدیل `implicit`) باعث اجرای این سازنده میشود. به عنوان یک نکته، کلاسی که شامل این دو سازنده است به طوری که پارامتر یکی از سازندها `initializer_list` و دیگری سازنده‌ی پیش فرض باشد، مثل:

```
class MyClass {
public:
    MyClass();
    MyClass(std::initializer_list<int> il);
};
```

با نوشتن دستور زیر:

```
MyClass myObject();
```

هیچ کدام از سازندها اجرا نمی‌شود و به جای آن یک تابع اعلان خواهد شد. اعلان تابع، یک عملکرد کمپایلرها است (برای اطلاعات بیشتر به ضمیمهٔ ۵ مراجعه کنید). با ایجاد شی از طریق {} و پاس ندادن آرگومان در دستور بالا، برنامه سازندهٔ پیش فرض را اجرا می‌کند. مانند:

```
MyClass myObject{}; // first ctor
```

در صورت نیاز به فراخوانی سازنده با پارامتر std::initializer_list<>, باید {} در پرانتز نوشته شود. برای مثال:

```
MyClass myObject({});
```

سازندهٔ دوم را فراخوانی می‌کند.

ویژگی دیگر {}, جهت کوتاه‌نویسی در ساخت یا دریافت شی در پارامتر و یا عبارت بازگشتی تابع است. اگر تابعی موجود باشد که با فراخوانی، شی کلاس MyClass را برمی‌گرداند، مانند:

```
MyClass getObject(){
    return MyClass(1,2,3);
}
```

با {} به صورت زیر میتواند بازنویسی شود:

```
MyClass getObject(){
    return {1,2,3};
}
```

به همین ترتیب فراخوانی تابع زیر را:

```
void func(const MyClass &mo) { /* ... */ ;
func(MyClass(1,2,3));
```

به صورت زیر میتوان بازنویسی کرد:

```
void func(const MyClass &mo) { /* ... */ ;
func({1,2,3});
```

یکی از تفاوت‌های {} با {}, هنگام استفاده از واژهٔ auto برای اعلان یک متغیر است:

هنگام استفاده از `auto` برای تعیین نوع متغیر، اگر `initializer` در {} باشد، متغیر به صورت عادی و اگر در = {} قرار گرفته باشد، متغیر از نوع با مقادیر درون = {} اعلان می‌شود. برای مثال:

```
auto i{4}; // i as int with value of 4  
auto i = {4}; // i as std::initializer_list
```

کلاس (۴) (وراثت، اعضای **friend** و سربارگذاری اپراتور)^۱ در C++ می‌توان یک کلاس را مشتق کلاس دیگر قرار داد به این معنی که تمام اعضای public و protected کلاس پایه به کلاس مشتق - بدون بازنویسی آن‌ها - اضافه می‌شوند. در این حالت می‌گوییم کلاس مشتق اعضای public یا protected کلاس پایه را به ارث برد است. پس:

به کلاسی که اعضای public و protected آن به ارث برد می‌شود کلاس پایه و به کلاسی که اعضای public و protected کلاس پایه را به ارث می‌برد کلاس مشتق گفته می‌شود.

برای تعریف یک کلاس از نوع کلاس مشتق، بعد از اعلان آن، یک access_specifier و نام کلاس پایه پس از (:) بین تعریف و اعلان آن باید قرار گیرد. با فرمت زیر:

```
class derived_class : access_specifier base_class
{
    // body of the class
};
```

توجه شود که access_specifier در بالا بیانگر این است که اعضای public یا protected کلاس پایه با چه سطح دسترسی در کلاس مشتق قرار می‌گیرند.

access specifier	معنی
public	اعضای public کلاس پایه در کلاس مشتق به صورت protected و اعضای public به صورت protected قرار می‌گیرند.
protected	اعضای public و protected کلاس پایه در کلاس مشتق به صورت protected قرار می‌گیرند.

^۱. Inheritance

private	اعضای public و protected کلاس پایه در کلاس مشتق به صورت private قرار میگیرند.
----------------	--

اعضای **private** کلاس پایه نیز در هیچ نوع وراثتی به ارث برد نمیشوند.

مثال ۱) برنامهای که پیادهسازی وراثت را نشان میدهد:

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

class Person {
public:
  void info() {
    cout << "I can talk.\n";
    cout << "I can walk.\n";
  }
};

class Employee : public Person {
private:
  double salary;
public:
  Employee(double s): salary{s} {}
  void employee_info() {
    cout << "My salary is: " << salary <<
      '\n';
  }
};

class Student : public Person {
private:
  string year;
  string university;
public:
  Student(string y, string u): year{y}, university{u}
{}
```

```
void student_info() {
    cout << "I'm a " << year << " year student
studying at " << university << ".\n";
}
};

int main() {
Person p{};
Employee e{2000};
Student s{"first", "x"};
p.info();

cout << '\n';
e.info();
e.employee_info();

cout << '\n';
s.info();
s.student_info();
}
```

خروجی برنامه به صورت زیر است:

```
I can talk.
I can walk.

I can talk.
I can walk.
My salary is: 2000

I can talk.
I can walk.
I'm a first year student studying at x.
```

توضیح) چون کارمند و دانشجو به طور کلیتر یک شخصاند، از وراثت در این برنامه استفاده شد تا ویژگیهایی مشترک اشخاص را در هر کلاس کارمند و دانشجو بازنویسی

نشوند.

وراثت برای پیادهسازی مسائلی که یک یا چند کلاس با یک کلاس کلیتر قابل بیانند استفاده میشود. برای مثال فرض کنید که در ساخت یک بازی چند نوع حریف وجود داشته دارد و یک سری ویژگی‌ها در بین تمام حریف‌ها مشترک باشد، مثل سرعت راه رفتن و برای عدم بازنویسی این صفات در تمام کلاس‌ها، می‌توان یک کلاس به نام **Enemy** ایجاد کرد تا تمام صفات و اعمال مشترک حریف‌ها را توصیف کند؛ سپس برای هر نوع حریف می‌توان یک کلاس جداگانه مشتق از **Enemy** تعریف کرد، تا موارد غیر مشترک نیز در آنها توصیف شوند.

مثال ۲) برنامه‌ای که ترتیب اجرای سازنده و مخرب کلاس‌های پایه و مشتق را نشان می‌دهد.

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

class Base {
public:
    Base(){ cout << "Base's constructor called!\n"; }
    ~Base(){ cout << "Base's destructor called!\n"; }
};

class Derived: public Base {
public:
    Derived(){cout << "Derived's constructor called!\n";}
    ~Derived(){cout << "Derived's destructor called!\n";}
};

int main() {
    Derived D0{};
}
```

خروجی برنامه به صورت زیر است:

```
BaseClass's constructor called!
DerivedClass's constructor called!
DerivedClass's destructor called!
BaseClass's destructor called!
```

در صورت پارامتردار بودن سازندهی کلاس پایه، در لیست انتساب دهندهی عضو کلاس مشتق باید مقداردهی شوند. در غیر این صورت سازندهی پیش فرض کلاس پایه (در صورت وجود) اجرا می شود. با فرمت زیر:

```
Derived(parameters) : Base{parameters}, ...
{
}
```

مثال (۳)

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

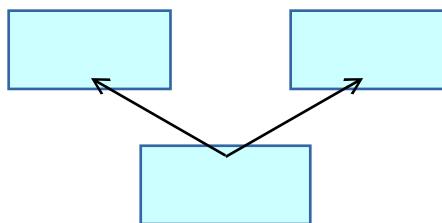
class Base {
public:
    Base(int a){
    }
};

class Derived: public Base {
private:
    int a;
public:
    Derived(int x , int y): Base{x}, a{y} {
    }
};

int main() {
    Derived D0{4, 10};
}
```

وراثت چندگانه^۱ و چندسطحی^۲

در صورت به ارث بردن یک کلاس از چند کلاس پایه آن را وراثت چندگانه می‌گویند.
دیاگرام آن به صورت زیر است:



مثال (۴)

```

#include <iostream>
using namespace std;

class Liquid {
private:
    double density;
public:
    Liquid(double d): density{d} {};
    void liquid_info() {
        cout << "Density: " << density << '\n';
    }
};

class Fuel {
private:
    double price;
public:
    Fuel(double r): price{r} {};
    void fuel_info() {
        cout << "Price: " << price << '\n';
    }
};
  
```

^۱. Multiple inheritance

^۲. Multilevel inheritance

```

class Petrol: public Liquid, public Fuel {
public:
    Petrol(double d, double r): Liquid{d}, Fuel{r}
    {};
    void petrol_info() {
        Liquid::liquid_info();
        Fuel::fuel_info();
        // prints other properties if defined for
petrol
    }
};

int main() {
    Petrol p{0.77,1.6};
    p.petrol_info();
}

```

خروجی برنامه به صورت زیر است:

```

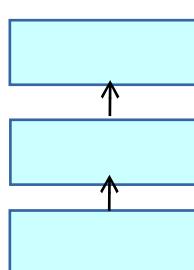
Density: 0.77
Price: 1.6

```

توضیح) به دلیل اینکه نفت هم یک مایع است و هم سوخت محسوب میشود از وراثت چندگانه در تعریف کلاس Petrol استفاده کردیم. در این مثال برای Liquid، متغیرهای دیگری برای ذخیره ویژگیهای دیگر مایع (علاوه بر چگالی) و یا در Fuel، توابعی برای افزایش یا کاهش قیمت ذخیره شده در price نیز میتوان با توجه به نیاز برنامه افزود.

در صورت به ارث برده شدن یک کلاس مشتق از کلاس مشتق دیگر آن را وراثت چندسطح می‌گویند.

دیاگرام آن به صورت زیر است:



مثال (۵)

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

class Person {
    //...
public:
    void print(){cout << "from Person";}
};

class Employee: public Person {
    //...
};

class Programmer: public Employee {
    //...
};

int main() {
    Programmer p{};
    p.print();
}
```

خروجی برنامه به صورت زیر است:

from Person

استفاده از وراثت چند سطحی و چندگانه مربوط به نوع طراحی برنامه می‌شود.

تابع و کلاس **friend**

در یک کلاس با تعیین کردن یک تابع از نوع **friend** به آن تابع اجازه دسترسی به

اعضای `protected` و `private` کلاس نیز داده میشود. با وجود اینکه اعلان تابع درون کلاس قرار میگیرد ولی تابع `friend` به عنوان یک عضو کلاس محسوب نیست.

برای تعیین تابع `friend`, باید واژه‌ی `friend` قبل از اعلان آن نوشته شود.
با فرمت:

```
class MyClass {
    friend returnType f(optional_params);
};

returnType f(optional_params){}
```

در این صورت تابع `friend`, بدون متغیر و به صورت `()x.f(args)` به جای `f` فراخوانی میشود. تعریف تابع `friend` در محل اعلان نیز میتواند قرار گیرد؛ در این صورت این تابع به صورت `inline1` در خواهد آمد. توابع عضو از کلاس‌های دیگر هم میتوانند از نوع تابع `friend` تعیین شوند. مثال:

```
class X {
    friend int Y::f(int);
};
```

مثال ۶

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

class MyClass {
    int x;
    friend void print(const MyClass &d ){
        cout << "x: " << d.x << '\n';
    }
}
```

¹ توضیح در فصل تابع ۲

```
public:
    MyClass(int x): x{x} {}
};

int main() {
    MyClass m{3};
    print(m);
}
```

خروجی برنامه به صورت زیر است:

x: 3

یکی از کاربردهای اصلی توابع **friend** زمانی است که یک تابع نیاز به دسترسی به دو کلاس را داشته باشد. برای مثال:

```
class A {
    friend void f(A &a, B &b);
};

class B {
    friend void f(A &a, B &b);
};

friend void f(A &a, B &b) { /* ... */ }
```

علاوه بر تابع، یک کلاس (union یا struct) را هم میتوان از نوع **friend** قرار داد که در این صورت، کلاس **friend** تعیین میشود. مانند:

```
class A {
    friend class B;
private:
    int x;
}

class B {
public:
    void f(A a) {
        a.x = 10; // ok
    }
}
```

{}

با استفاده از کلاس `friend`، میتوان دسترسی به اعضای `private` را فقط به کلاسهایی که نیاز دارند ایجاد کرد. در نتیجه از `public` کردن یکسری عضو فقط برای تعداد مشخصی از کلاس جلوگیری میشود. به این ترتیب با استفاده‌ی مناسب از کلاسهای `friend` میتوان کپسول کردن کلاس را ارتقا داد.

سربارگذاری اپراتورها^۱

در یک کلاس با سربارگذاری اپراتورها، میتوان برای اپراتورها زمانی که روی اشیای کلاس اعمال میشوند یک عملکرد تعريف کرد. برای مثال اگر یک کلاس برای کار با زمان نوشته باشیم، به طوری که شامل اعضای `seconds`, `minutes` و `hours` باشد، اعمال اپراتور + روی اشیای این کلاس مانند:

```
Time m3 = m1 + m2;
```

مناسب است که باعث جمع شدن اجزای `seconds`, `minutes` و `hours` از `m1` و `m2` شود و شی جدیدی با مقادیر جمع شده را برگرداند تا `m3` با این شی مقداردهی شود. سربارگذاری اپراتور به صورت زیر صورت میگیرد:

```
returnType operator+(optional_param);
```

که **0** اپراتور موجود در C++ است.

اپراتور سربارگذاری شده یک تابع است و مانند یک تابع نیز نوع بازگشتی، لیست پارامتر و بدن دارد.

مثال (۷)

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

class Cube {
    int x, y, z;
```

¹. Operator overloading

```

public:
    Cube(int x, int y, int z): x{x}, y{y}, z{z} {}
    int getVolume(){return x*y*z;}
    Cube operator+(const Cube &C0) {
        Cube temp{0,0,0};
        temp.x = x + C0.x;
        temp.y = y + C0.y;
        temp.z = z + C0.z;
        return temp;
    }
};

int main() {
    Cube box1{1,3,5}, box2{6,3,6};
    Cube box3 = box1 + box2;
    cout << box3.getVolume() << '\n';
}

```

خروجی برنامه به صورت زیر است:

462

(مثال ۸)

```

#include <iostream>
using namespace std;

class Line {
double m, b;
public:
    Line(double m, double b): m{m}, b{b} {}
    double operator()(double x){
        return m*x + b;
    }
};

int main() {
    Line l{2, 4.5};
    cout << l(2) << '\n';
}

```

توضیح) در این مثال برای کار کردن با خط در فضای دو بعدی یک کلاس تعریف کردیم.

با پاس دادن `X` از طریق اپراتور `()`، عرض خط برمیگردد. ظاهر استفاده از اپراتور سربارگذاری شده‌ی `()`، مانند ظاهر فراخوانی یک تابع است؛ به همین دلیل به نوع داده‌ای که `()` را سربارگذاری کرده باشد، شی تابع (`function object`) یا فانکتور (`functor`) میگویند.

با سربارگذاری اپراتورها میتوان رابط یک کلاس را به صورت واضحتر پیاده کرد. همچنین اپراتورها را طوری سربارگذاری کنید که با اپراتورهای موجود تطابق داشته باشند. برای مثال سربارگذاری اپراتورهای منطقی، مناسب است که با برگرداندن یک مقدار `bool` پیاده شوند.

اپراتورهای سربارگذاری، شده مانند تابع عضو، با اسم `operator0` نیز قابل فراخوانیاند. در مثال^۴، دو دستور زیر برابرند:

```
Cube box3 = box1.operator+(box2);
Cube box3 = box1 + box2;
```

تمام اپراتورها به جز اپراتورهای زیر قابل سربارگذاریاند:

<code>::</code>	<code>*</code>	<code>.</code>	<code>:</code>	<code>sizeof</code>
-----------------	----------------	----------------	----------------	---------------------

به این ترتیب اپراتور `&` نیز قابل سربارگذاری است. مانند:

```
struct MyClass {
    MyClass operator&() { ... }
};
```

در نتیجه این اپراتور میتواند به جای بازگردانی آدرس یک شی، عمل کاملاً متفاوتی را انجام دهد. برای دریافت آدرس شی چنین کلاس‌هایی، از اپراتور `std::addressof` موجود در `<memory>` استفاده میشود:

```
#include <memory>
MyClass m;
addressof(m);
```

در صورت سربارگذاری بودن `&` نیز آدرس اپراند خود را برمیگردد. نیاز به ذکر است که اپراتورها، لزوماً به عنوان توابع عضو کلاسها سربارگذاری نمیشوند. دو اپراتور پر تکراری که خارج از اعضای کلاس سربارگذاری میشوند، اپراتورهای `>` و

<> برای ورودی و خروجی است؛ زیرا اپراند سمت چپ این اپراتورها، اشیای استریم (mehr) قرار می‌گیرد. مثل:

```
cout << myObject << '\n';
```

همچنین به دلیل اینکه سربارگذاری اپراتورهای <> و << به طور معمول نیاز به دسترسی به اعضای private کلاس را دارد، تعریف سربارگذاری آنها از نوع friend تعیین می‌شود.

(مثال ۹)

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

class MyClass {
    int x;
    friend ostream& operator<<(ostream& out, const
        MyClass& o);
    friend std::istream& operator>>(istream& in,
        MyClass& o);
public:
    MyClass(int x) : x{x} {}
};

ostream& operator<<( ostream& out, const MyClass& m)
{
    out << m.x;
    return out;
}

istream& operator>>(istream& in, MyClass& m) {
    in >> m.x;
    return in;
}

int main() {
```

```
MyClass m{7};  
cout << m << '\n';  
cout << "Specify a value: ";  
cin >> m;  
cout << m << '\n';  
}
```

خروجی برنامه به صورت زیر است:

```
7  
Specify a value: 10  
10
```

توضیح) در این مثال اپراتورهای سربارگذاری شده، شی `istream` یا `ostream` قبل از اپراتور و شی `MyClass` بعد از آن را گرفته و عضو `x` را مقداردهی و یا مقدارگیری میکند. در آخر نیز شی جدید استریم برمیگردد تا در صورت وجود با اشیای دیگر استریم ترکیب شود.

کلاس (۵) (چندریختگی)^۱

اگر دو کلاس به نام های Base و Derived داشته باشیم به طوری که کلاس Derived از کلاس Base مشتق شده باشد، مثل:

```
class Base{
public:
    void f(){cout << "From Base\n";}
};

class Derived: public Base{
public:
    void f(){cout << "From Derived\n";}
};
```

مانند هر کلاس دیگر میتوان رفرنس یا اشارهگری از نوع Derived تعریف کرد که به شی Derived اشاره میکند و یا رفرنس داده میشود. برای کدهای بالا با نوشتن دستورات زیر:

```
Derived DObject;
Derived *DPtr = &DObject;
Derived &DRef = DObject;

DOPtr->f();
DORef.f();
```

برنامه خروجی "From Derived" را در خروجی نمایش میدهد. این خروجی یه این معناست که اشارهگر و رفرنس همانطور که پیش بینی میشد تابع عضو f() را از شی memberObj->member (برای یادآوری Derived فراخوانی میکند (برای یادآوری *j با است).

در C++ همچنین میتوان اشارهگر یا رفرنسی از نوع پایه تعریف کرد و آن را به شی کلاس مشتق اشاره داد. برای کدهای اول بخش دستورات زیر صحیح است:

¹. Polymorphism

```
Derived DObject;

Base *BPtr = &DObject;
Base &BRef = DObject;

(*BPtr).f();
BRef.f();
```

و برنامه خروجی "From Base" را در خروجی نمایش خواهد داد. این خروجی به این معناست که برنامه، تابع f() از کلاس Base را فراخوانی کرده. با وجود اینکه این دستورات صحیح اند، ممکن است خروجی که انتظار داشتید تولید نشده باشد. ولی با توجه به اینکه اشارهگر و رفرنس BPtr و BRef در نهایت از نوع کلاس Base اند و فقط به توابع عضو کلاس خود دسترسی دارند، خروجی منطقی ایجاد میشود. با اضافه کردن واژه‌ی virtual قبل از اعلان تابع f از کلاس پایه، در صورت وجود تابع عضو همانم و هم پارامتر در کلاس مشتق، برنامه، تابع عضو کلاس مشتق را فراخوانی خواهد کرد.

(۱) مثال

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

class Base {
public:
    virtual void f() { cout << "Base\n"; }
};

class Derived1: public Base {
public:
    void f() { cout << "Derived1\n"; }
};

class Derived2: public Base{
```

```

public:
    void f() { cout << "Derived2\n"; }
};

void g(Base &b) {
    b.f();
}

int main() {
    Derived1 d1;
    Derived2 d2;
    Base b;
    g(d1);
    g(d2);
    g(b);
}

```

توضیح) در این مثال تابع عضو `f`، در ۲ دستور `(d1).g()` و `(d2).g()` متناسب با شی کلاس مشتق فراخوانی میشود. در این حالت می گوییم تابع عضو `area` کلاس مشتق، تابع عضو `area` از کلاس پایه را نادیده (`override`) می گیرد. به این نوع الگوی شیگرایی، چندریختگی میگویند زیرا شی یک نوع (اشارهگر یا رفرنس کلاس پایه) توابع عضو مختلفی را بر اساس شی که به آن پاس داده میشود (اشیای کلاس مشتق) فراخوانی میکند. یکی از کاربردهای چندریختگی ذخیره کردن اشیای مختلف کلاس‌های مشتق در یک آرایه است؛ از آنجایی که تمام عناصر یک آرایه باید از یک نوع داده باشند، با اعلان یک آرایه از نوع اشارهگر کلاس پایه میتوان اشیا کلاسهای مشتق مختلف را ذخیره کرد. به عنوان مثال:

```
Vehicle *v[] = {&car, &bicycle};
```

در چندریختگی، با فراخوانی کردن تابع عضو `virtual`، بدنه‌ی توابعی که اجرا میشوند در حین برنامه مشخص خواهند شد؛ به همین دلیل به این ارزیابی `late binding`

گفته می‌شود. در مقابل آن، متغیرها و فراخوانی توابع عادی که در حین کمپایل مشخص می‌شوند، early binding نام دارند. به همین دلیل به چندريختگی، چندريختگی late binding نیز گفته می‌شود.

در C++ پیاده سازی چندريختگی توسط روشی به نام جدول مجازی (virtual table) صورت می‌گیرد. به دلیل اینکه آشنایی با آن در نحوه استفاده از چندريختگی تاثیری ندارد از پرداختن به آن پرهیز خواهد شد.

واژه‌ی virtual میتواند علاوه بر تابع عضو کلاس پایه، برای تابع عضو کلاس مشتق نیز نوشته شود.

تابع abstract و کلاس پایه‌ی pure virtual

اگر به جای نوشتن بدنی تابع virtual در کلاس پایه، اعلان آن مساوی با ۰ قرار گیرد (=۰)، تابع عضو از نوع pure virtual تعیین می‌شود. کلاس پایه‌ای که حداقل یک تابع pure virtual داشته باشد، کلاس پایه‌ی abstract (انتزاعی) نام دارد. برای مثال:

```
class Base{
public:
    virtual int f() = 0;
};
```

تابع f از نوع pure virtual است و کلاس Base از نوع کلاس abstract تعريف می‌شود. از کلاس abstract نمیتوان مستقیم شی ایجاد کرد. برای مثال برای کلاس Base دستور زیر غلط است:

```
Base b; // error
```

با وجود این میتوان از کلاس abstract اشاره‌گر یا رفرنس تعريف کرد. در این حالت باید تعريف توابع عضو pure virtual در کلاس مشتق قرار گیرند تا نادیده گرفته (override) شوند.

مثال (۲)

```

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

class Polygon {
public:
    virtual double area() const = 0;
};

class Rectangle: public Polygon {
private:
    double width, length;
public:
    Rectangle(double w, double l): width{w}, length{l}
{}
    double area() const {
        return width * length;
    }
};

class Triangle: public Polygon {
private:
    double base, height;
public:
    Triangle(double b, double h): base{b}, height{h}
{}
    double area() const {
        return (base * height / 2);
    }
};

double getArea(const Polygon *p) {
    return p->area();
}

```

```
int main() {
    Rectangle r{4.6,5.7};
    Triangle t{2.4,5.5};
    cout << getArea(&r) << '\n';
    cout << getArea(&t) << '\n';
}
```

خروجی برنامه به صورت زیر است:

26.22
6.6

توضیح) این برنامه شامل کلاس **Polygon** برای کار با چند ضلعی و دو کلاس مشتق **Rectangel** برای مستطیل و **Triangel** برای مثلث است. کلاس‌های مشتق با تابع **area** مساحت شکل خود را بر می‌گردانند. این برنامه همچنین شامل یک تابع به اسم **getArea** است که یک شی از نوع اشاره‌گر کلاس پایه (**Polygon**) می‌گیرد. با پاس دادن اشیای مختلف کلاس مشتق به دلیل اینکه **area** از کلاس پایه از نوع **virtual pure** است، تابع عضو **area** متناسب با شی مشتق فراخوانی می‌شود. در بعضی از پیاده سازی‌های چندريختگی، ساخت شی مستقیماً از کلاس پایه بیمعنی است. مثلاً در مثال ۲ برای دریافت محیط یک چند ضلعی، باید نوع آن مشخص باشد. به همین دلیل تابع عضو **area()** از کلاس **Polygon** را از نوع **pure virtual** تعیین کردیم تا هر کلاس مشتق، آن را متناسب با شکل خود پیاده کند. ABC (Abstract Base Class) به طور مخفف **abstract** به کلاس پایه می‌گفته شود.

dynamic_cast و اپراتور RTTI¹

RTTI قابلیت C++ برای تعیین نوع شی هنگام اجرای برنامه است. RTTI، دو اپراتور زیر را در دسترس قرار میدهد:

¹. Run-time type information

typeid .۱
dynamic_Cast .۲

typeid: نوع شی را هنگام اجرا برمی‌گرداند. این اپراتور میتواند برای تعیین مقادیر چندریخت (اشاره‌گر یا رفرنس کلاس پایه به مشتق) استفاده شود؛ در غیر این صورت مقادیر غیر چندریخت در حین کمپایل تعیین می‌شوند. این اپراتور در هدر <**typeinfo**> قرار دارد و نوع بازگشتی آن از نوع **&const type_info** است. این رفرنس با فراخوانی تابع عضو **name()**، میتواند نام شی را برگرداند.
مثال ۳) برنامه‌ای که کاربرد **typeid** را برای مقادیر چندریخت و غیرچندریخت نشان میدهد.

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

class Base {
public:
    virtual void func() {}
};

class Derived : public Base {};

int main() {
    Derived* pd = new Derived;
    Base* pb = pd;
    int i = 10;
    cout << typeid(pb).name() << '\n';
    cout << typeid(*pb).name() << '\n';
    cout << typeid(pd).name() << '\n';
    cout << typeid(*pd).name() << '\n';
    delete pd;
    cout << typeid(i).name() << '\n';
}
```

برنامه را کمپایل و اجرا کنید تا خروجی مشابه زیر تولید شود:

```
Base
Derived
Derived
Derived
i
```

توجه شود که برای تعیین نوع اشارهگر در صورت دیرفرنس نشدن، حاصل type_info برای نوع اشارهگر تعیین میشود نه شیای که به آن اشاره میشود. همچنین اگر نوع اشارهگر که دیرفرنس میشود، به typeid اشاره کند، استثنای std::bad_typeid را برمیگرداند (با استثنایها در آینده آشنا خواهیم شد). dynamic_cast در چندریختگی، تبدیل اشارهگر مشتق به پایه به صورت خودکار است. upcast صورت میگیرد و upcast نام دارد. تاکنون در تمام مثالهای چندریختگی از استفاده کردیم. برای مثال وقتی دستور زیر را مینویسیم:

```
Base *b{new Derived};
```

تبدیل upcast صورت میگیرد. تبدیل downcast بر عکس upcast است به این معنی که زمانی صورت میگیرد که اشارهگر کلاس پایه به اشارهگر کلاس مشتق تبدیل میشود. برخلاف upcast باشد توسط برنامهنویس گست شود. برای گست کردن اشیای چندریخت از dynamic_cast استفاده میشود. dynamic_cast بر خلاف static_cast صحیح بودن گست را در حین اجرای برنامه چک میکند. dynamic_cast در صورت عدم امکان گست، برای اشارهگرها و برای رفرنسها، استثنای std::bad_cast را (به دلیل عدم وجود nullptr در رفرنسها) برمیگرداند.

(۴) مثال

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

class Base {
public:
```

```

virtual ~Base() {}
virtual void name() {}
};

class Derived: public Base {
public:
    void name() {}
};

int main() {
    Base* b1 = new Base;
    Base* b2 = new Derived;

    if(Derived* d = dynamic_cast<Derived*>(b1)) {
        cout << "downcast from b1 to d successful\n";
        d->name(); // safe to call
    }

    if(Derived* d = dynamic_cast<Derived*>(b2)) {
        cout << "downcast from b2 to d successful\n";
        d->name(); // safe to call
    }

    delete b1;
    delete b2;
}

```

خروجی برنامه به صورت زیر است:

downcast from b2 to d successful

توضیح) در این برنامه با استفاده از اپراتور `dynamic_cast`, اشارهگر به `Base`, اشارهگر `Derived` به شی `Derived` کست میشود. هنگام کردن `b1` به اشارهگر `Derived`, به دلیل اینکه اشارهگر به شی `Derived` اشاره نمیکند, کست صورت نمیگیرد و `nullptr` برمیگردد. با استفاده از اپراتور `dynamic_cast` علاوه بر کردن, برای تبدیل بین شی شاخه

های مختلف وراثت نیز میتواند استفاده شود؛ این اپراتور در صورت امکان شی جدید را بر میگرداند در غیر این صورت کست صورت نخواهد گرفت. `dynamic_cast` همچنین چند محدودیت نیز دارد؛ برای مثال فقط برای وراثت `public` امکان کست کردن وجود دارد و برای وراثت `protected` و `private` امکان پذیر نیست؛ البته این موارد نیز به ندرت پیش می آیند.

مخرب `virtual`

در چندريختگی، زمانی که شی کلاس مشتق توسط اشارهگر کلاس پایه از بین میرود، مخرب کلاس مشتق فراخوانی نمیشود. این عملکرد را در مثال زیر بررسی میکنیم. مثال (۵)

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

class Base {
public:
    Base(){cout<<"Base constructor called\n";}
    ~Base(){cout<<"Base destructor called\n";}

};

class Derived: public Base {
public:
    Derived(){cout<<"Derived constructor called\n";}
    ~Derived(){cout<<"Derived destructor called\n";}

};

int main() {
    Base* p = new Derived;
    delete p;
}
```

خروجی برنامه به صورت زیر است:

```
Base constructor called
Derived constructor called
Base destructor called
```

این نوع عملکرد امکان بروز خطای undefined behavior را فراهم میکند. برای حل این مشکل باید سازنده کلاس پایه به صورت virtual تعیین شود. در مثال ۵ با اعلان مخرب کلاس Base به صورت زیر:

```
virtual ~Base() { /* ... */ ;
```

خروجی زیر تولید میشود:

```
Base constructor Called
Derived constructor called
Derived destructor called
Base destructor called
```

در این حالت، مخرب کلاس پایه نیز فراخوانی میشود. به صورت یک قاعده‌ی کلی، زمانی که کلاس یک یا چند تابع عضو virtual دارد، مخرب آن را نیز از نوع virtual تعیین کنید.

سازنده‌ی virtual

در C++, به طور مستقیم قابلیتی تحت عنوان سازنده‌ی virtual وجود ندارد؛ به این معنی که هنگام ساختن شی در چندریختگی نوع کلاس مشتق باید مشخص باشد. با این وجود به صورت غیر مستقیم میتوان با توابعی با اسم‌هایی مثل `create()` و `clone()` از نوع virtual، امکان ساخت و یا کپی شی بدون مشخص بودن کلاس مشتق فراهم شود.

مثال ۶

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

```
class Polygon {
public:
    virtual ~Polygon () { }

    virtual Polygon* clone() const = 0;
    virtual Polygon* create() const = 0;
};

class Circle : public Polygon{
public:
    Circle* clone() const; // Covariant Return
Types
    Circle* create() const; // Covariant Return
Types
    //...
};

Circle* Circle::clone() const { return new
Circle(*this); }
Circle* Circle::create() const { return new Circle(); }

void f(Polygon& s) {
    Polygon* s2 = s.clone();
    Polygon* s3 = s.create();

    delete s2;
    delete s3;
}

int main() {
    Circle s;
    f(s);
}
```

توضیح) در این مثال، توابع `clone()` و `create()` در کلاس پایه را به صورت تعیین کردیم. در این مثال، تابع `f` میتواند شی هر نوع کلاس مشتق شده را

دريافت کند و بدون مشخص بودن نوع آن، آن را کپی و در ساخت شی از آن استفاده کند. توجه شود که نوع بازگشتی توابع نادیده گرفته شده در کلاس مشتق، به جای کلاس پایه (Polygon)، از نوع کلاس مشتق (Circle) نوشته شده است. اين نوع مقدار بازگشتی، نوع بازگشتی کوواريانت^۱ نام دارد که در کمپایلرهای جديد شناخته شده است (برای اطلاعات بيشتر به ضميمه ۴ مراجعه کنيد)؛ در اين مثال در صورت عدم پشتيبانی کمپایلر، آن را به Polygon تغيير دهيد.

واژه‌ی **override**

فرض کنيد دستورات زير در برنامه موجود باشد:

```
class Base{
    virtual void a(int);
    virtual void b(int);
};

class Derived: public Base{
    virtual void a(float);
    virtual void b(char);
};
```

در اين دستورات پaramترهای توابع عضو Derived سهوا اشتباه نوشته شدهاند. در اجرای برنامه، زمانی که چندریختگی اتفاق بيفتد، به دليل اينكه توابع کلاس مشتق با کلاس پایه يكی نيستند، توابع جديد اعلن می‌شوند و بدون خطا چند ریختگی صورت نمیگيرد. با استفاده از واژه‌ی **override** بعد از اعلن تابع به طور مختص به کمپایلر اعلام می‌شود که می‌خواهيم اين تابع، تابع پایه را نادیده بگيرد. در صورت عدم انجام اين کار خطا صادر خواهد شد. مثال:

```
class Base{
    virtual void a(int);
```

^۱. Covariant Return Types

```

virtual void b(int)
};

class Derived: public Base{
    virtual void a(int) override;
    virtual void b(int) override;
};

```

واژهی final

از واژهی **final** در تعریف کلاس‌ها می‌توان استفاده کرد که در صورت قرارگیری، جلوی مشتق شدن از کلاس را می‌گیرد. این واژه همچنین برای توابع **virtual** می‌تواند استفاده شود که در این صورت از نادیده‌گرفته شدن (**override**) آن در کلاس مشتق جلوگیری می‌شود. برای مثال:

```

class Class1 final {
    // code
};
class Class2: public Class 1 // error
{};


```

: و یا

```

class Class1 {
    virtual void f();
};

class Class2: public Class1 {
    void f() final;
};


```

```

class Class3: public Class2{
    void f(); // error
};


```

در این دستورات به دلیل اینکه تابع **f()** از **Class2**، تابع **f()** از **Class1** را نادیده می‌گیرد، به طور خودکار از نوع **virtual** قرار می‌گیرد. سپس با قرار دادن آن از نوع

`final`، از نادیده‌گرفته شدن آن در توسط تابع `f()` از `Class3` جلوگیری خواهد شد. نیاز به ذکر است که واژه‌های `final` و `override` در مکانهای خاص برای کمپایلر معنی میدهند و جز مشخصه‌های رزرو شده برای کمپایلر نیستند. در نتیجه اعلان یک متغیر، تابع و ... به اسم `final` و `override` صحیح است:

```
void override() { /* ... */} // ok
int final; // ok
```

کلاس (۶) (توابع عضو خاص)

در گذشته مشاهده کردیم اگر یک کلاس ساده مانند کلاس زیر داشته باشیم:

```
class MyClass {  
    int x;  
public:  
    MyClass(int x): x{x} {}  
    void print_x() const { cout << x << '\n';}  
};
```

امکان کپی اشیای این کلاس در اشیای کلاس‌های دیگر کلاس، با استفاده سازنده وجود دارد. برای مثال:

```
MyClass m{4};  
MyClass m2{m};  
m2.print_x(); // 4
```

به سازندهای که به طور خودکار توسط کمپایلر تولید میشود و امکان کپی اشیای کلاس را فراهم میکند، سازندهی کپی گفته میشود. اگر یک کلاس از تخصیص منابع دینامیکی استفاده کند، با اجرای سازندهی کپی، فقط آدرس اشارهگر کپی میشود و فضای دینامیکی مورد اشاره‌ی آن کپی نخواهد شد.^۱ برای مثال اگر کلاس زیر را جهت ذخیره و کار با رشتهها تعریف کنیم^۲:

```
#include <cstring>  
class MyString {  
    char* cstr;  
public:  
    MyString(const char* arg)  
        : cstr(new char[std::strlen(arg)+1]) {  
            std::strcpy(cstr, arg);  
    }
```

¹. Shallow copy

² این مثال صرفا جهت آشنایی با مبحث این فصل است. کلاس std::string، کلاس کاملی برای کار و اعمال روی رشتهها است.

```
~MyString() {
    delete[] cstr;
}
};
```

سازندهی کپی باید توسط برنامه نویس تعریف شود تا عمل مناسب جهت کپی منبع دینامیک مورد اشاره‌ی `cstr` را انجام دهد^۱. این سازنده میتواند به صورت زیر پیاده شود:

```
MyString(const MyString& other) {
    cstr = new char[strlen(other.cstr) + 1];
    std::strcpy(cstr, other.cstr);
}
```

بعد از ساخت شی مناسب است امکان کپی آن با اپراتور `=` نیز وجود داشته باشد. برای مثال:

```
MyString M01("string");
MyString M02("");
M02 = M01;
```

برای دستیابی به این عملکرد، اپراتور `=` در کلاس `MyString` به صورت زیر میتواند سربارگذاری شود:

```
MyString& operator=(const MyString& other) {
    char* tmp_cstr = new char[strlen(other.cstr) + 1];
    std::strcpy(tmp_cstr, other.cstr);
    delete[] cstr;
    cstr = tmp_cstr;
    return *this;
}
```

این اپراتور سربارگذاری شده، کپی **assignment**² نام دارد. توجه شود که استفاده از `=` هنگام تعریف متغیر، سازندهی کپی را اجرا میکند (نه کپی

¹. Deep copy

². Copy assignment

assignment را). در دستورات زیر شرح داده شده:

```
MyString M01{"string"}; // copy ctor
MyString M02("string"); // copy ctor
MyString M03 = "string"; // copy ctor
MyString M04{};
M02 = M01; // copy assignment
```

فرض کنیم تابعی به اسم make_MyString() در برنامه تعریف شده باشد که با فراخوانی، شی MyString را برمیگرداند. در این صورت نوشتمن دستور زیر:

```
MyString M0;
M0 = make_MyString();
```

برای کلاس MyString مراحل زیر را انجام میدهد:

۱. با فراخوانی تابع، یک شی موقت (temporary) تولید میشود و مقدارش با سازندھی copy در شی M0 کپی میشود.
۲. مقدار از پیش ذخیره شده شی M0 از بین میرود.
۳. شی موقت بازگشته از تابع نیز از بین میرود.

به عنوان مثالی دیگر در مورد اشیای موقت، نتیجه‌ی اپراتورهای + و ... است. مثلاً اگر اپراتور + برای کلاس MyString سربارگذاری شود، دستور سوم ایجاد شی موقت میشود:

```
MyString M01("1"), M02("2");
MyString M03 = M01 + M02;
```

با توجه به اینکه شی موقت در نهایت پس از پایان دستور از بین میرود، منطقی است که به جای کپی شدن فضای آن، فضایش انتقال یابد. این قابلیت، به طور کلی انتقال فضا نام دارد و با استفاده از رفرنس rvalue که در C++11 معرفی شده امکان پذیر است.

جهت یادآوری lvalue در عمل، مقداریست که آدرسش قابل دریافت و یا از نوع رفرنس lvalue باشد؛ در غیر این صورت عبارت از نوع rvalue است. عبارتهای مشخص شده lvalue هستند:

```

int a; // lvalue
int& get_a(){
    return a;
}
get_a(); // lvalue

struct A{
    int x;
};
A A0; // lvalue
A0.x; // lvalue

```

و مقادیر مشخص شده زیر rvalue هستند:

```

10; // rvalue
struct A{
    int a;
}
A().a; // rvalue

int a{10}, b{2};
a + b; // rvalue

```

برای تعریف کامل دسته بندی مقادیر میتوانید به ضمیمهٔ ۶ مراجعه کنید.

برای افزودن قابلیت انتقال، دو عضو سازندهٔ انتقال و اپراتور انتقال **assignment** نیز باید تعریف شوند. این اعضا برای کلاس **MyString** به صورت زیر قابل پیادهسازی هستند:

```

// move ctor
MyString(MyString&& other): cstr(other.cstr){
    other.cstr = nullptr;
}

//move assignment
MyString& operator=(MyString&& other) {
    if(this != &other)

```

```

{
    delete[] cstr;
    cstr = other.cstr;
    other.cstr = nullptr;
}
return *this;
}

```

توجه فرمایید که در انتقال `if` از انتقال یک شی به خودش جلوگیری میکند.

همچنانی امکان انتقال عبارات `lvalue` نیز وجود دارد؛ برای این کار از تابع `std::move` موجود در `<utility>` استفاده میشود. برای `MyString` زیر کاربرد `std::move` را نشان میدهد:

```

#include <utility>
...
MyString M0("string");
MyString M02(std::move(M0));

```

تابع `std::move`، آرگومان `lvalue` خود را به `rvalue` تبدیل میکند که در نهایت باعث فراخوانی عضو انتقال شی میشود. با وجود اینکه خروجی تابع `std::move` از نوع `rvalue` است، مقدار آن یک مقدار موقت (`temporary`) نیست. به همین دلیل کمیته C++ چنین مقداری که قابل انتقال هست ولی مقدار موقت ندارد `xvalue` نامگذاری کرده است. مقادیر `xvalue` جزئی از مقادیر `rvalue` هستند.

نکتهی قابل ذکر در مورد رفرنس `rvalue` این است که با اینکه این رفرنسها به یک مقدار `rvalue` ارجاع داده میشوند، با این حال خود میتوانند هم `rvalue` و هم از نوع `lvalue` باشند. رفرنس `rvalue` در صورت داشتن اسم یک عبارت `lvalue` است.

مثل رفرنس `a` در دستور زیر:

```
int&& a{10}; // lvalue
```

در صورت عدم داشتن نام، رفرنس `rvalue` یک نوع `rvalue` است. مثل فراخوانی تابع

زیر:

```
int&& f() { /* ... */ }
f(); // rvalue
```

در نتیجه اگر تابعی به صورت زیر داشته باشیم که با رفرنس rvalue، متغیر MyString میسازد:

```
void usesMyString(MyString&& S1){
    MyString S2 = S1; // executes copy ctor
}
...
usesMyString(MyString("String"));
```

هنگام ساخت شی S2، سازنده‌ی کپی اجرا می‌شود و شی S1 انتقال نمی‌یابد. برای انتقال رفرنس rvalue، مانند هر lvalue دیگر از std::move() باید استفاده کرد:

```
void usesMyString(MyString&& S1){
    MyString S2 = std::move(S1); // executes move
ctor
}
```

توجه شود که از std::move() جهت بازگشت مقدار از تابع نباید استفاده شود. برای مثال:

```
MyString make_MyString() {
    MyString S1("string");
    return std::move(S1); // bad
}
```

زیرا با نوشتن std::move(), از بهینه سازی RV0¹ که توسط کمپایلر صورت می‌گیرد، جلوگیری می‌شود. بهینه سازی RV0 در صورت امکان، عبارت بازگشتی را به صورت خودکار انتقال میدهد. به همین ترتیب تابع make_MyString() باید بدون std::move() نوشته شود. همچنین مقدار بازگشتی تابعی که شی برمی‌گرداند (به نام تابع فکتوری) نباید از نوع رفرنس rvalue به شی ایجاد شده در تابع (شی اتوماتیک)

¹. Return value optimization

باشد. در این صورت خطای Undefined behaviour ایجاد می‌شود. برای مثال:

```
MyString &&make_MyString() // don't do this
{
    return MyString("string");
}
```

تابع عضو خاص^۱

در گذشته بررسی کردیم اگر هیچ سازنده‌های در کلاس اعلان نشود، کمپایلر به طور خودکار یک سازنده بدون پارامتر (سازنده‌ی پیش فرض) و با بدنیهی خالی ایجاد می‌کند. علاوه بر سازنده‌ی پیشفرض، ۵ تابع عضو زیر نیز در صورت نیاز و با شرایط خاص ایجاد می‌شوند. این توابع عبارت‌اند از:

- مخرب
- سازنده‌ی کپی
- کپی assignment
- سازنده‌ی انتقال
- انتقال assignemnt

شرایط ایجاد این اعضاء در جدول زیر بررسی شده:

تابع عضو	شرط ایجاد	تعریف پیش فرض
سازنده‌ی پیشفرض	اگر سازنده دیگری اعلان نشده باشد.	بدنهی خالی
مخرب	اگر مخرب اعلان نشده باشد.	بدنهی خالی
سازنده‌ی کپی	اگر اعضای انتقال اعلان نشده باشند.	کپی اعضا
assignemnt کپی	اگر اعضای انتقال اعلان نشده باشند.	کپی اعضا
سازنده‌ی انتقال	اگر اعضای انتقال، کپی و مخرب	انتقال اعضا

^۱. Special member functions

	اعلان نشده باشند.	
انتقال assignment	اگر اعضای انتقال، کپی و مخرب اعلان نشده باشد.	انتقال اعضا

(مثال)

```

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

class MyClass{
private:
    int x;
public:
    MyClass(int x): x(x) {}
    int get_x() const {
        return x;
    }
};

int main() {
    MyClass 01(10);
    MyClass 02(01);
    cout << 02.get_x() << '\n';
    MyClass 03 = move(02);
    cout << 03.get_x() << '\n';
    cout << 02.get_x() << '\n';
}

```

خروجی برنامه به صورت زیر است:

```

10
10
10

```

توضیح) در این مثال با اعلان نکردن توابع عضو خاص، کمپایلر آنها را به طور خودکار ایجاد میکند. در این برنامه وجود اعضای کپی و انتقال اشیای کلاس بررسی میشوند.

توجه شود که اعضای انتقال تولید شده، اعضا را نظیر انتقال میدهند. هنچنین در C++, انتقال دادن متغیرها از نوعهای پایه‌ای مثل `int`, `double` و ... باعث کپی آنها می‌شود؛ در نتیجه با انتقال شی ۰۲ در این مثال، فضای متغیر `x` از ۰۲ بعد از انتقال هنوز باقی خواهد ماند.

توابع عضو خاص - علاوه بر کلاس‌هایی که از تخصیص دینامیک حافظه استفاده می‌کنند - در کلاس‌هایی که نیاز به عملکرد خاص دارند نیز باید توسط برنامه نویس تعریف شوند. برای مثال:

```
class MyClass{
public:
    ...
    ...
    ~MyClass(){
        makeLog("MyClass object destructed.");
    }
};
```

توجه شود در این کلاس با تعریف مخرب، از تولید اعضای انتقال توسط کمپایلر جلوگیری می‌شود. توابع عضو خاص که بر اساس شرایط گفته شده ایجاد نشده‌اند می‌توان به صورت زیر تولید شوند:

```
memberFunction_declarartion = default;
```

در کلاس `MyClass`, تولید اعضای انتقال که با اعلان مخرب ایجاد نمی‌شوند، به صورت زیر می‌توانند به برنامه اضافه شوند:

```
MyClass(MyClass&&) = default;
MyClass& operator=(MyClass&&) = default;
```

همچنین با تعریف اعضای انتقال، تولید خودکار اعضای کپی نیز متوقف می‌شود. در صورت نیاز به این اعضا، دستورات زیر نیز باید به کلاس اضافه شوند:

```
MyClass(const MyClass&) = default;
MyClass& operator=(const MyClass&) = default;
```

در نتیجه تعریف کامل کلاس `MyClass` که نیاز به مخرب غیر پیش فرض دارد به

صورت زیر میشود:

```
class MyClass{
...
public:
    MyClass() {};
    MyClass(MyClass&&) = default;
    MyClass& operator=(MyClass&&) = default;
    MyClass(const MyClass&) = default;
    MyClass& operator=(const MyClass&) = default;
    ~MyClass(){
        makeLog("MyClass object destructed.");
    }
};
```

برای جلوگیری از تولید اعضای خاص، از `delete` بهجای `default` میتوان استفاده کرد. به صورت:

`memberFunction_declaration = delete;`

اگر در کلاسی فقط اعضای انتقال معنیدار باشند، به صورت زیر میتواند از تولید اعضای کپی جلوگیری کرد:

```
class uncopyable{
public:
    uncopyable() {};
    uncopyable(uncopyable&&) = default;
    uncopyable& operator=(MyClass&&) = default;
    uncopyable(const uncopyable&) = delete;
    uncopyable& operator=(const uncopyable&) = delete;
    ~uncopyable(){
        makeLog("uncopyable object destructed.");
    }
};
```

البته در این مثال به دلیل وجود اعضای انتقال، بدون `delete` قرار دادن اعضای کپی نیز این اعضا به صورت خودکار ایجاد نمیشوند؛ در این موقع برای واضحتر کردن عملکرد کلاس و جلوگیری از نتایج غیرمنتظره توصیه میشود که اعضای مورد نیاز را با

افزوده و یا با `delete default` حذف کنید.

اکنون به علت وجود شرایط تولید و یا عدم تولید توابع عضو خاص میپردازیم. در C98++، که اعضای انتقال معرفی نشده بودند، اعضای سازنده‌ی کپی، کپی assignment و مخرب به تولید یک دیگر وابسته نبودند؛ زیرا اهمیت Rule of three مورد توجه قرار نگرفت. rule of three یک مفهوم در C98++ بود که در کدنویسی کلاسها بیان میکرد: در صورتی که یکی از سه اعضای مخرب، سازنده‌ی کپی و کپی assignment توسط برنامه نویس تعریف شوند، هر سه عضو نیز باید توسط برنامه نویس ایجاد شوند. منطق این قانون این است که در صورت نیاز به تعریف یکی از سه اعضاً احتمالاً کلاس از منابع خارجی استفاده میکند؛ به همین ترتیب فرآیندهای مناسب برای ایجاد deep copy نیز باید توسط برنامه نویس ایجاد شود. در C++11 با توجه به این قانون، کمیته‌ی C++ توابع عضو اضافه شده (سازنده‌ی انتقال و انتقال assignment) را به گونه‌ای معرفی کرد که با ایجاد آنها توسط برنامه نویس، تولید بقیه‌ی اعضای خاص به طور خودکار انجام نشود. تولید این اعضا به دلیل ایجاد مشکل در کمپایل کدهای قدیمی، برای ۳ عضو خاص C98++ قرار نگرفت. در C++11 به بعد Rule of five و Rule of zero نیز معرفی شدند. Rule of five میکند با تعریف ۳ عضو مخرب، سازنده‌ی کپی و کپی assignment از تولید خودکار اعضای انتقال جلوگیری میشود، در نتیجه در صورت نیاز به اعضای انتقال، هر ۵ عضو باید توسط برنامه نویس ایجاد شوند. Rule of zero نیز بیان میکند در کلاسی که ۵ عضو خاص توسط برنامه نویس تعریف شده‌اند، کلاس، منبع را باید به طور انحصاری مدیریت کند. در کلاس‌های دیگر نیز باید هیچ یک از این ۵ عضو تعریف شود. مثال:

```
class rule_of_zero {
private:
    std::vector<int> v;
    int i;
};
```

این کلاس از منابع خارجی استفاده نمیکند، به این ترتیب با تعریف نکردن هیچ یک از ۵ عضو، با انتقال و کپی اشیای این کلاس، اعضایشان نظیر به نظری کپی/انتقال میباشد. نکته‌ی قابل ذکر در مورد اعضای انتقال و کپی این است که با انتقال شی، لزوماً شی منتقل نمیشود. ابتدا نحوه عملکرد `rvalue` را یادآوری میکنیم و سپس به بررسی این عبارت میپردازیم. به طور کلی با رفرنس `lvalue` به صورت `const` میتوان به یک مقدار `rvalue` ارجاع کرد؛ برای مثال:

```
int const& i = 5;
```

این نوع رفرنس، توانایی ایجاد تغییر مقدار مورد ارجاعش را ندارد. برای رفرنس `i` دستور زیر غلط است:

```
i = 6; // error
```

رفرنس `rvalue` نیز میتواند به یک مقدار `rvalue` ارجاع شود:

```
int &&i = 5;
```

با این تفاوت که توانایی تغییر شی مورد ارجاع را دارد؛ برای `i`:

```
i = 6; // ok
```

در صورتی که دوتابع سربارگذاری شوند و یکی رفرنس `rvalue` و دیگری رفرنس `lvalue` از نوع `const` را بگیرد، با پاس دادن `rvalue`، برنامه تابع با پارامتر `rvalue` را فراخوانی میکند. برای مثال:

```
void print(int const& i){
    std::cout << "i = int const& \n";
}

void print(int &&i){
    std::cout << "i = int && \n";
}

print(10);
```

خروجی زیر را خواهد داشت:

```
i = int &&
```

نتیجه‌ی این عملکرد این است که با پاس دادن `rvalue` به شی، صرفاً مقدار منتقل نمی‌شود؛ زیرا این مقادیر فقط زمانی منتقل می‌شوند که سازندھی انتقال، سازندھی کپی را سربارگذاری کرده باشد. در غیر این صورت توسط سازندھی کپی نیز می‌توانند دریافت شوند.

(مثال)

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

class MyClass{
public:
    MyClass() {};
    MyClass(const MyClass& other) {
        cout << "copy ctor executed\n";
    };
    MyClass& operator=(const MyClass& other){
        cout << "copy assignment executed\n";
    }
    ~MyClass() = default;
};

int main() {
    MyClass 01;
    MyClass 02 = move(01);
}
```

خروجی برنامه به صورت زیر است:

```
copy ctor executed
```

با سعی به انتقال مقدار `rvalue`، از کمپایلر درخواست می‌شود تا در صورت وجود اعضای انتقال آن را منتقل کند در غیر این صورت کپی صورت می‌گیرد. با حذف کردن اعضای

انتقال با `delete`، کمپایلر تولید اعضای کپی را نیز متوقف میکند؛ در این صورت با سعی به انتقال شی پیغام خطای صادر میکند:

```
class MyClass{
public:
    ...
    MyClass(MyClass&&) = delete;
    MyClass& operator=(MyClass&&) = delete;

};

MyClass 01{};
MyClass 02 = move(01); // error
```

دادهای اساسی C++ نیز همانطور که گفته شد، اساساً فاقد فرآیندهای انتقالاند و با سعی به انتقال آنها، باعث ایجاد کپی از آنها میشود. برای مثال:

```
int i{4};
int i2 = std::move(i);
cout << i << '\n'; // 4
cout << i2 << '\n'; // 4
```

واژه‌ی **static**

هر شی در برنامه یک طول عمر^۱ دارد. چهار طول عمر اشیا عبارتند از:

۱. اتوماتیک: فضای یک شی با طول عمر اتوماتیک با آغاز بلوک اشغال و با خروج از آن آزاد می‌شود. متغیر **a** در تابع زیر، نمونه‌ای از این نوع شی است:

```
void f() {  
    int a;  
}
```

۲. دینامیک: اشیایی که با تخصیص حافظه‌ی دینامیک ایجاد و آزاد می‌شوند از نوع این طول عمراند. برای مثال:

```
int *a = new int;  
delete a;
```

۳. موقت: فضای اشیای موقت با عبارتی درون عبارت دیگر ایجاد می‌شوند و با اتمام اجرای عبارت بیرونی نیز از بین می‌روند. برای مثال:

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

۴. استاتیک (**static**): فضای شی استاتیک با شروع برنامه ساخته و با پایان آن آزاد خواهد شد. متغیری که خارج از هر تابعی اعلام شده، نمونه‌ای از یک شی با این نوع طول عمر است. برای مثال:

```
int a;  
int main(){  
}
```

برای اعلان یک متغیر از نوع استاتیک که در یک بلوک اعلان شده، از واژه‌ی **static** استفاده می‌شود. برای مثال:

```
void f() {  
    static int i;  
}
```

^۱. Storage duration

مقداردهی اولیه متغیرهای static فقط بار نخستی که برنامه به اعلان آنها برسد صورت می‌گیرد.

(مثال ۱)

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

void f(){
    static int counter = 1;
    cout << counter << ' ';
    counter++;
}

int main() {
    for(int i = 0; i < 5; ++i) {
        f();
    }
}
```

خروجی برنامه به صورت زیر است:

1 2 3 4 5

توضیح) فضای متغیر استاتیک counter، در طول برنامه ثابت میماند. همانطور که از خروجی برنامه مشخص است، مقداردهی اولیه‌ی متغیر a با ۱ فقط در اولین فراخوانی تابع صورت می‌گیرد و در فراخوانی‌های بعدی فقط آخرین مقدار آن در دسترس قرار میگیرد.

کاربرد دیگر واژه‌ی static در اعلان اعضای یک کلاس است. در تعریف کلاس زیر:

```
class A{
public:
    int x;
};
```

برای متغیر عضو x در هر شی ایجاد شده از کلاس A، فضای جدایگانهای اشغال میشود.

اگر بخواهیم یک متغیر تعریف کنیم که فضای آن در تمام اشیای یک کلاس مشترک باشد و مقدار آن بین تمام اشیا یکسان بماند، باید متغیر عضو را با واژه `static` اعلان کنیم. برای مثال:

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

class A{
public:
    static int x; // declaration
};

int A::x = 4; // definition
int main() {
    A a, a2;
    a.x++;
    cout << a.x << "=" <<
        a2.x << '\n';
}
```

خروجی برنامه به صورت زیر است:

5=5

توضیح) در این برنامه مقدار متغیر عضو استاتیک با یک شی (a) افزایش میابد و مقدار آن هنگام دسترسی با شی دیگر نیز تغییر کرده است. متغیر عضوی که از نوع `static` تعیین شده، فقط اعلان آن قرار میگیرد. برای استفاده از آن، خارج از کلاس (بدون واژه `static`) باید تعریف شود. در این مثال خط زیر متغیر a را تعریف و مقداردهی اولیه میکند:

```
int A::x = 4;
```

اگر متغیر با تعریف مقداردهی اولیه نشود، برنامه آن را با ۰ مقداردهی میکند. برای مثال:

```
int A::x;
```

برای دسترسی به متغیر عضو static، علاوه بر شی، از نام کلاس همراه با: نیز میتوان استفاده کرد. در این مثال، سه عبارت زیر برابرند:

```
a.x;
a2.x;
A::x;
```

اگر عضو static عدد صحیح یا enum از نوع const باشد، در کلاس مستقیم با = یا {} میتواند مقداردهی اولیه شود. در این صورت نیاز به تعریف جدایگانه آن در خارج از کلاس نیست. برای مثال:

```
class A {
    const static int x = 10;
}
```

همچنین اعضای استاتیک زمانی که constexpr باشند نیز میتوانند در کلاس با مقادیر ثابت مقداردهی اولیه شوند. برای مثال:

```
class A{
public:
    constexpr static int ar[]{1,2,3};
    constexpr static int i = 1;
};
```

از C++17، متغیر عضو static که با واژه‌ی inline نوشته شده باشد نیز می‌تواند در خود کلاس تعریف و مقداردهی اولیه شود. برای مثال:

```
class A{
public:
    inline static int i = 1;
};
```

تابع عضو static

تابع عضوی که از نوع static تعریف شده، مانند متغیر عضو static، میتواند بدون اشیای کلاس به آن دسترسی ایجاد شود. این نوع توابع، دسترسی به اعضای غیر

کلاس را ندارند. توابع عضو یک کلاس زمانی از نوع `static` اعلان می‌شوند که عمل آنها کلی باشد و وابسته به مقادیر غیر `static` کلاس نباشند. مثال ۲) برنامه‌ای که از تابع عضو `static` برای نمایش یک متغیر `static` کلاس استفاده می‌کند.

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

class A{
public:
    inline static int counter = 0;
    A(){
        counter++;
    }
    static void printCounter(){
        cout << counter << '\n';
    }
};

int main() {
    A a,a2,a3;
    A::printCounter();
}
```

خروجی برنامه به صورت زیر است:

3

توضیح) این برنامه با ایجاد هر شی یکی به مقدار `counter` اضافه می‌کند و در نهایت با تابع `printCounter` این متغیر نمایش می‌یابد. توابع `static` به `this` نمی‌توانند دسترسی پیدا کنند زیرا این نوع توابع بین تمام اشیا به اشتراک گذاشته می‌شوند. واژه‌ی `static`, در تعیین اتصال (linkage) اجزا نیز کاربرد دارد که در فصل آینده بررسی می‌کنیم.

چند فایل کردن کد منبع

در پروژه‌های بزرگ C++ کدهای منبع برنامه را برای کدنویسی و مدیریت آسانتر به چند فایل تقسیم می‌کنند. این فایل‌ها میتوانند شامل کلاسها، توابع و عناصر دیگر C++ باشند. اگر دو فایل به اسم `main.cpp` و فایل `A.cpp` (که تابع `main` در آن قرار دارد) موجود باشد، با کمپایل برنامه، ابتدا فایل `main.cpp` و سپس `A.cpp` کمپایل و به فایل شی¹ تبدیل می‌شوند. سپس فایلهای اشیای ایجاد شده، در مرحله‌ی لینک کردن² به هم متصل می‌شوند. به دلایل زیر ساختی هر فایل شی، از عناصر موجود در فایلهای اشیای دیگر اطلاع ندارد. برای استفاده از این اعضاء، اعلان اجزای مورد نیاز در هر فایلی که از آنها استفاده می‌کند باید قرار گیرد. برای مثال:

```
// A.cpp
void f(){
    // body
}
```

```
// main.cpp
void f();

int main() {
    f();
}
```

اگر تابع `f` (از فایل `A.cpp`) علاوه بر `main.cpp` در فایلهای دیگر نیز استفاده شود، اعلان آن در هر فایلی که از آن استفاده می‌کند نیز باید افزوده شود. برای جلوگیری از بازنویسی اعلانها در هر فایل `.cpp`، از فایل `hدر` (معمولاً با پسوند `.h`) استفاده می‌شود. در فایل `hدر`، اعلان اجزا (به طور کلیتر، واسطه‌ها) قرار می‌گیرد و تعریف آنها در یک فایل `.cpp` قرار خواهد گرفت. برای مثال:

```
// A.h
void f();
```

¹. Object file

². Linking

```
// A.cpp
void f(){
    // body
}
```

اکنون فایلهای `A.cpp` و `B.cpp` اگر به تابع `f` نیاز داشته باشند، به جای اعلان جداگانه آن در هر فایل، با دستور `#include` به صورت زیر:

```
// main.cpp
#include "A.h"

int main() {
    f();
}
```

```
// b.cpp
#include "A.h"
void g(){
    f();
}
```

اعلان تابع `f` در هر کدام از فایل‌ها افزوده می‌شود. نحوهی عملکرد `#include` مشابه کپی و پیسیت کردن محتوای فایل مقابله آن در فایلی که دستور در آن نوشته شده است. البته در این مثال استفاده از فایل هدیر، تفاوت محسوسی با اعلان مستقیم `f` در فایلهای `main.cpp` و `b.cpp` دیده نمی‌شود. ولی در فایلهایی که شامل تعداد زیادی تعریف است، راحتی استفاده از فایل هدیر محسوس خواهد شد. دستور `#include` یک دستور پیشپردازnde است. دستورات پیشپردازnde قبل از کمپایل دستورات اصلی C++ و در مرحلهای به اسم پیشپردازش اجرا می‌شوند. به فایل منبع (`.c`، `.cpp`) و هدیرهایی که در مرحلهای پیشپردازش (با `#include`) به آن افزوده می‌شود، واحد ترجمه^۱ (به طور مخفف TU) گفته می‌شود.

^۱. Translation Unit

اگر نام فایلی که مقابله `#include` قرار می‌گیرد در <> نوشته شود، برنامه فایل هدر را از کتابخانه‌ی استاندارد به برنامه می‌افزاید. اگر نام فایل در "" قرار گیرد، فایل نوشته شده از دایرکتوری مقابله `#include` اضافه خواهد شد (در صورت وجود فقط نام یک فایل، دایرکتوری هدر از دایرکتوری فایل جست و جو می‌شود).

مثال ۱) در این مثال با افزودن یک کلاس در فایل مجزا به برنامه با استفاده از `qt` آشنا می‌شویم. IDE‌های دیگر روند مشابهی دارند.

۱. ابتدا یک پروژه ایجاد کرده و سپس از منوی `File` or `creator` `New File` را انتخاب کنید.

۲. گزینه‌ی `C++` و سپس `C Class++` را از قسمت راست انتخاب کنید و روی ... کلیک کنید. با این کار فایل جدیدی شامل یک کلاس ایجاد می‌شود.

۳. در قسمت `Class Name` واژه‌ی `MyClass` را برای کلاس نوشته و روی `Next` کلیک کنید.

۴. روی `Finish` کلیک کنید.

دستورات زیر را در فایل‌های ایجاد شده تایپ کنید:

```
// main.cpp
#include <iostream>
#include "myclass.h"
using namespace std;

int main() {
    MyClass m;
    m.f();
}
```

```
myclass.cpp:
#include "myclass.h"
#include <iostream>
MyClass::MyClass() {}
```

```

void MyClass::f()
{
    std::cout << "f()\n";
}

myclass.h:
#ifndef MYFILE_H
#define MYFILE_H
class MyClass
{
public:
    MyClass();
    void f();
};
#endif // MYFILE_H

```

خروجی برنامه به صورت زیر است:

4100

توضیح) همانطور که از خروجی دیده می شود، در myfile.cpp از اپراتور تفکیک ناحیه (::) برای تعریف اجزای کلاس استفاده شده است.
در مثال ۱، سه دستور پیشپردازندگی زیر که در فایل .h. به طور خودکارتوسط IDE اضافه میشود:

```

#ifndef MYCLASS_H
#define MYCLASS_H

// content

#endif

```

روشی است که در آن از چند بار افزوده شدن محتوای یک هدیر با استفاده از ماکروها جلوگیری می شود. به این صورت که ابتدا با افزوده شدن این هدیر، ماکروی

در فایل تعريف می‌شود و محتوا (content) نیز به فایل افزوده خواهد شد. در صورت سعی به افزودن دوباره‌ی این هدر، به دلیل اینکه ماکروی MYCLASS_H در فایل از قبل تعريف شده است، ساختار شرط پیشپردازنه اجرا نمی‌شود و محتوای content بار دیگر افزوده نخواهد شد. همچنین نام ماکروی MYCLASS_H کاملاً اختیاری است ولی معمولاً با حروف بزرگ نوشته می‌شود.

در مورد هدرهای کتابخانه‌ی استاندارد لازم به ذکر است که پسوند .h نام آنها نباید نوشته شود. در صورت قرار دادن .h کمپایلر، هدر را از کتابخانه‌ی استاندارد C جستجو می‌کند. مثلاً:

```
#include <string>
```

هدر string را که مختص C++ است به فایل افزوده می‌شود، ولی:

```
#include <string.h>
```

هدر string زبان C که کاملاً هدری متفاوتی است افزوده می‌شود. همچنین برای استفاده از هدرهای زبان C به جای قرار دادن .h. می‌توان حرف c را قبل از نام آن قرار داد. مثلاً:

```
#include <cstring>
```

که معادل:

```
#include <string.h>
```

است؛ با این تفاوت که هدرهایی که با C شروع می‌شوند (مثل cstring) اجزای آنها در std namespace قرار دارند. در نتیجه استفاده از این هدرها نسبت به .h. توصیه می‌شود (زیرا اسم اجزای هدر مستقیماً در فایل قرار نخواهد گرفت).

اتصال (linkage)

یک اسم در برنامه که بیانگر یک شی، رفرنس، تابع، نوع، قالب، namespace و یا شمارنده است، به طور کلی می‌تواند ویژگی‌ای به اسم اتصال (linkage) داشته باشد.

اتصال (linkage)، یکسان بودن اسم را در ناحیه‌های مختلف بیان می‌کند. متغیرهایی که در یک بلوک بدون واژه‌ی `extern` اعلام شده‌اند، اتصال ندارند. این اسمها فقط در بلوکی که اعلام می‌شوند قابل استفاده‌اند. در صورت اعلام یک جز هم نام با اسم اجزای بدون اتصال، اعلام جدیدی با یک فضای متفاوت در برنامه ایجاد می‌شود. برای مثال:

```
void f(){
    int x = 2;
    cout << x << '\n';
}

void f2(){
    int x = 3;
    cout << x << '\n';
}

f(); // 2
f2(); // 3
```

اتصال داخلی (internal linkage) به این معنی است که اسم، فقط در واحد ترجمه‌ای (TU) که در آن اعلام می‌شود قابل مشاهده است. متغیر و توابعی که با واژه‌ی `const` در ناحیه‌ی `namespace` اعلام می‌شوند از این دسته‌اند. متغیرهای `static` (و `constexpr`) به طور پیش فرض و اسمهایی که در `namespace` بدون اسم اعلام می‌شوند با اتصال داخلی‌اند. برای مثال:

```
namespace {
    int i;
}

static int i;
const int i2;

static void f();

int main(){
```

```
// ...
}
```

اسمی که اتصال خارجی (external linkage) دارد به این معنی است که توسط لینکر قابل دسترس است و میتواند بین TU‌های مختلف استفاده شود. هر متغیر غیر static و بدون static، متغیر اعلام شده با extern و توابعی که اعلام آنها بدون const صورت گرفته، از این نوعاند. برای مثال:

```
int i;
extern const int i2;
void f();

int main() {
    //...
}
```

به طور خلاصه با واژه‌ی static و extern میتوان اتصال را به طور خاص تغییر داد. اتصال پیش فرض برای اسمهای غیر static، extern و برای const، static است. یک متغیر و تابع معمولی در برنامه به تعداد دلخواه میتواند اعلام شود ولی تعریف آنها فقط یکبار میتواند نوشته شود^۱. برای مثال:

```
int f();
int f();
int f();
int f(){ return 1; } // ok
```

:

```
int f(){ return 1; }
int f(){ return 2; } // error
```

در مثال ۲، اگر در هدر، مستقیم تعریف تابع را مینوشتیم مانند:

```
void f(){
    cout << "f()\n";
}
```

¹. One Definition Rule (ODR)

با `#include` شدن آن در چند فایل، تعریف `f()` نیز چند بار در برنامه قرار میگیرد و در نتیجه منجر به بروز خطای لینکر میشود. برای مثال:

```
A.h
extern const int i = 1; //error
void f(){ // error
    // ...
}
```

با استفاده از واژه‌ی `inline` در تعریف تابع یا متغیر، امکان تعریف یکسان تابع یا متغیر را در چند واحد ترجمه (TU) ممکن میکند. در نتیجه با قرار دادن تعریف اجزای `inline` در هر دو فایل، امکان `#include` کردن آنها در چند فایل، بدون خطای لینکر، امکان پذیر میشود. برای مثال:

```
A.h
inline void f(){
    // ...
}
inline extern const int i;
```

"extern" C

در خیلی از موقع نیاز به استفاده از کدهای C در C++ و بالعکس است. این اتصال به آسانی صورت نمیگیرد؛ زیرا برای مثال در C++ (برخلاف C) امکان سربارگذاری توابع وجود دارد و کمپایلر، بین توابع سربارگذاری شده نمیتواند با اسم تابع - به عنوان یک اسم منحصر به فرد - به توابع دسترسی پیدا کند. در نتیجه کمپایلر نام آنها را با کمپایل تغییر میدهد¹. برای مثال توابع زیر:

```
void f(int i) {};
void f(bool b) {};
```

به توابع زیر میتوانند تغییر کنند:

```
void __f_i(int i) {};
void __f_b(bool b) {};
```

¹. Name mangling

همچنین کمپایلرهای مختلف با تغییر نام توابع، لزوماً اسمهای یکسانی تولید نمیکنند. برای تعیین توابع و متغیرهای C++ (با اتصال خارجی) به طوری که در کدهای C قابل اجرا باشند (تغییر نام توابع صورت نگیرد و ...)، ابتدای اعلان آنها باید واژه‌ی `extern` باشد (قرار گیرد. برای مثال:

```
extern "C" void f();
void f() {
    // ...
}
```

و یا به صورت زیر اعلان چند جز را میتوان از نوع C تعیین کرد:

```
extern "C" {
    void f();
    void g();
}
```

برای استفاده از توابع C در C++ نیز اعلان آنها را با "C" تعیین شود:

```
extern "C" void c_func(int);

void cpp_func(){
    c_func();
}
```

به همین دلیل برای `#include` کردن هدرهای C (غیر از هدرهای C سیستم مثل `cstring`، `cstdio` و ...) که کدهای همراه آن با کمپایلر C کمپایل شده، باید دستور `extern "C"` را در C در قرار داد. برای مثال:

```
extern "C" {
    #include "c_header.h"
}

int main() {
    c_func();
}
```

در این صورت اعلان محتوای "extern "C" با `c_header.h` تعیین می‌شود. به طور مشابه کل محتوای هدیر را میتوان در دستورات زیر قرار داد:

```
#ifdef __cplusplus
extern "C" {
#endif

// header's content
```

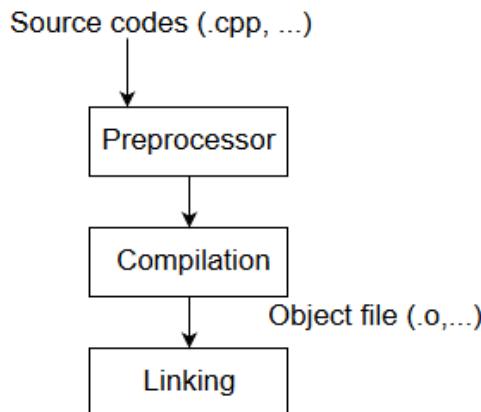
```
#ifdef __cplusplus
}
#endif
```

با `#include` شدن این هدیر، اعلانهای آن در صورتی از نوع "extern "C" تعیین می‌شوند که برنامه با کمپایلر C++ کمپایل شود (ماکروی `cplusplus` در آن تعریف شده باشد) در غیر این صورت اعلان آنها به صورت عادی قرار می‌گیرد.

کمپایل برنامه‌های چند فایلی و کتابخانه^۱

کمپایل برنامه‌های نوشته شده در C++ و تولید خروجی نهایی در چند مرحله انجام می‌شود. همانطور که گفته شد ابتدا دستورات پیشپردازنه (`#include`, `#define`, `#` و ...) در مرحله‌ای به اسم پیشپردازش اجرا و پردازش می‌شوند. سپس کمپایلر خروجی پیشپردازنه را دریافت و فایلهای شی (object files) تولید می‌کند. در مرحله‌ی آخر لینکر با استفاده از فایلهای شی یک خروجی قابل اجرا تولید می‌کند.

¹. Library



برای مثال اگر برنامه شامل فایلهای زیر باشد:

```

point.h
#ifndef POINT_H
#define POINT_H
#include <iostream>

class Point {
private:
    double x, y;
public:
    Point(double x, double y);
    Point();
    double get_x() const;
    double get_y() const;
    friend std::ostream& operator<< (std::ostream
&out, const Point &point);
};

#endif // POINT_H

point.cpp
#include "point.h"
#include <iostream>
Point::Point(double x, double y): x(x), y(y) {
}
  
```

```

Point::Point(): Point(0,0) {
}
double Point::get_x() const {
    return x;
}

double Point::get_y() const {
    return y;
}

std::ostream& operator<< (std::ostream &out, const
Point &point) {
    out << "(" << point.x << ","
        << point.y << ")\n";
    return out;
}

main.cpp
#include <iostream>
#include "point.h"
using namespace std;

int main() {
    Point p1{0,0}, p2{4,6};
    cout << p1 << '\n';
    cout << p2 << '\n';
}

```

برای کمپایل با g++ از دستورات زیر استفاده میشود:

```

g++ -c -o main.o main.cpp
g++ -c -o point.o point.cpp

```

این دستورات فایل‌های شی main.o و point.o تولید میکنند. در g++ گزینهی -c بیانگر ایجاد خروجی فایل شی و -o نیز اسم فایل خروجی را مشخص میکند. سپس با

دستور زیر:

```
g++ -o program point.o main.o
```

فایلهای شی به هم لینک داده میشوند و خروجی `program` تولید میشود. در سیستم عاملهای شبیه یونیکس، سپس با دستور زیر برنامه‌ی `program` اجرا میشود (در دایرکتوریای که `program` قرار دارد):

```
./program
```

همچنین تولید فایل شی و لینک آنها میتوان در یک دستور و به صورت زیر انجام شود:

```
g++ -o program main.cpp point.cpp
```

در C++ قسمتهایی از برنامه مثل کلاس‌ها و توابع در صورتی که در چند برنامه قابل استفاده باشند، در یک واحدی به اسم کتابخانه قرار میگیرند. که در این صورت مدیریت، تغییر و استفاده از آنها ساده‌تر خواهد شد. تا اینجا از کتابخانه‌ی استاندارد C++ استفاده کردیم که به طور پیش فرض در برنامه قابل استفاده است. در این قسمت به نحوی ایجاد یک کتابخانه‌ی جدید و استفاده از آن میپردازیم.

کتابخانه‌ها به دو دسته تقسیم میشوند:

۱. static

۲. shared

کتابخانه‌های static فقط هنگام کمپایل قابل لینک و اتصال به برنامه‌هاند. کتابخانه‌های shared - در صورت نیاز - هنگام اجرا به برنامه متصل میشوند. کتابخانه‌ی static باعث وابستگی برنامه به کتابخانه‌های خاصی بر روی سیستم نمی‌شود؛ ولی کتابخانه‌ی shared اگر به دلایلی مثل به روز رسانی، دچار مشکل شود، تمام برنامه‌های وابسته به آن نیز با مشکل مواجه خواهند شد. از نظر حجم، کتابخانه‌ی static حجم برنامه را بالا میبرد ولی shared نخواهد برد.

به طریقه‌ی اتصال کتابخانه‌ی `shared` در حین اجرا، اتصال دینامیک و کتابخانه‌ی `static` در حین کمپایل کمپایل، اتصال استاتیک می‌گویند. کتابخانه‌های `shared` در لینوکس معمولاً پسوند `.so` در `macOS` `.dylib` و در ویندوز `.dll` دارند. پسوند کتابخانه‌ی `static` همچنین در لینوکس و `macOS` `.a` و در ویندوز `.lib` است.

ساخت کتابخانه‌ی `static` در لینوکس^۱

در مثال قسمت قبل (`point.h`, `point.cpp`, `main.cpp`)، با دستورات زیر از `point.cpp` میتوان یک کتابخانه‌ی `static` ایجاد کرد.

```
g++ -c -o point.o point2t.cpp  
ar rcs libpoint.a print.o
```

پس از ایجاد کتابخانه‌ی `libpoint.a`، با دستور زیر به برنامه لینک داده می‌شود:

```
g++ -o program main.cpp -lpoint
```

با `-L` میتوان مسیر کتابخانه موجود غیر از مسیر استاندارد را تعیین کرد برای مثال:

```
g++ -o program main.cpp -L/home/imam/lib -lpoint
```

توجه شود در شرایط واقعی چندین فایل شی در کتابخانه‌ی `static` قرار می‌گیرند. در اینجا هدف آشنایی اولیه با کتابخانه‌ها است.

اکنون به نحوی ایجاد کتابخانه‌ی `shared` می‌پردازیم. برای این کار نیز از مثال `shared` استفاده می‌کنیم. با دستور زیر شی `print.o` به یک کتابخانه‌ی `point` تبدیل می‌شود:

```
g++ -fPIC -shared print.o -o libpoint.so
```

حال کتابخانه‌ی ایجاد شده میتواند به برنامه لینک داده شود:

```
g++ -o program -lpoint
```

^۱ توضیحات این بخش با IDE نیز قابل پیاده‌سازی‌اند.

نکته‌ی قابل ذکر این است که جهت استفاده‌ی برنامه از کتابخانه‌ی **shared** در حین اجرای برنامه، باید مسیر دریافت کردن آن مشخص باشد. به طور پیش فرض کتابخانه‌ها از دایرکتوریهای از قبل تعیین شده - مثل `/usr/local/lib` - جستجو و افروزه می‌شوند. با دستور زیر:

```
export LD_LIBRARY_PATH=/path/
```

جستجو و دریافت کتابخانه از دایرکتوری تعیین شده صورت خواهد گرفت. برای مشاهده‌ی کتابخانه‌های **shared** متصل به برنامه از دستور `ldd` مانند زیر می‌توان استفاده کرد:

```
ldd ./program
```

خروجی `ldd` برای برنامه‌ای که شامل کتابخانه‌ی **shared** به اسم `libpoint.so` است:

صورت زیر می‌شود:

```
iman@linux-32bi:~/projects/temp> ldd ./a.out
    linux-vdso.so.1 (0x00007ffd3c2af000)
libsharedprint.so => ./libpoint.so
(0x00007f6346786000)
    libstd++C.so.6 => /usr/lib64/libstd++C.so.6
(0x00007f634645e000)
    libm.so.6 => /lib64/libm.so.6
(0x00007f634615d000)
    libgcc_s.so.1 => /lib64/libgcc_s.so.1
(0x00007f6345f46000)
    libc.so.6 => /lib64/libc.so.6
(0x00007f6345b9e000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f6346989000)
```

فرق `dll` ویندوز با `so` در لینوکس این است که در ویندوز، محتوای کتابخانه به طور پیش فرض خارج از آن قرار نمی‌گیرد و با دستور:

```
__declspec(dllexport)
```

که قبل از یک داده، تابع یا کلاس قرار میگیرد، باعث صادر (export) شدن آن خارج از کتابخانه خواهد شد. کتابخانه صادر شده هم از داخل و هم از خارج از آن قابل دسترس است. با دستور زیر:

```
_declspec(dllexport)
```

جز به داخل کتابخانه وارد (import) میشود که باعث خواهد شد فقط درون کتابخانه قابل دسترس شود. با استفاده از دستورات پیشپردازندگی زیر:

```
#ifdef LIBRARY_EXPORTS  
#    define DLLEXPORT __declspec(dllexport)  
#else  
#    define DLLEXPORT __declspec(dllimport)  
#endif
```

و اعلان به صورت زیر:

```
DLLEXPORT int f();
```

از آنجایی که فایل هدر هم توسط کاربر و هم توسط کمپایلر استفاده میشود میتوان با روش بالا ماکروی LIBRARY_EXPORTS را در کمپایل کتابخانه تعریف کرد که دستور صادر شدن اجزا جایگزین DLLEXPORT میشود.

namespace

تقریبا در تمامی مثال‌های این کتاب از عبارت:

```
using namespace std;
```

استفاده کردیم. در این بخش به namespace‌ها و نحوه‌ی ایجاد و استفاده از آن‌ها می‌پردازیم.

با استفاده از namespace می‌توان مجموعه‌ای از دستورات شامل کلاس‌ها، اشیا و توابع را دسته بندی کرد. فرمت تعریف namespace به صورت زیر است:

```
namespace id {  
    // classes, objects and functions  
}
```

یک namespace به نام id تعریف می‌شود. دسترسی به اعضای namespace با اپراتور :: و همراه با نام namespace صورت می‌گیرد.
مانند:

```
id::member
```

(۱) مثال

```
#include <iostream>  
using namespace std;  
  
namespace np1 {  
    int a;  
}  
namespace np2 {  
    int a;  
}  
int main() {  
    np1::a = 4;  
    np2::a = 6;  
    cout << "a from np1 namespace = " << np1::a << '\n';  
    cout << "a from np2 namespace = " << np2::a << '\n'
```

```
n';
}
```

خروجی برنامه به صورت زیر است:

```
a from np1 namespace = 4
a from np2 namespace = 6
```

توضیح) همانطور که خروجی دیده میشود، دو متغیر با نامهای یکسان (a) تعریف شد ولی به دلیل آنکه در دو namespace مجزا قرار داشتنند، کمپایلر پیغام خطأ صادر نکرد.

واژه‌ی using

با استفاده از واژه‌ی **using** می‌توان عضو یا اعضای یک namespace را در محدوده‌ای از برنامه معرفی کرد. به این معنا که دسترسی به عضو یا اعضای معرفی شده بدون نیاز به اپراتور `::`، امکان پذیر میشود.
با فرمت زیر:

```
using id::member;
```

(۲) مثال

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

namespace np1{
    int a;
}
using np1::a;

int main(){
    a = 4;
    cout << a << '\n';
}
```

خروجی برنامه به صورت زیر است:

4

همچنانین تمام اعضای یک namespace را می‌توان با استفاده از عبارت `using namespace` به برنامه معرفی نمود. با فرمت:

using namespace id;

مثال ۳

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

namespace np1{
    int a;
    void print_var(){ cout << a << '\n'; }
}

using namespace np1;

int main(){
    a = 4;
    print_var();
}
```

خروجی برنامه به صورت زیر است:

4

توجه شود که پس از معرفی عضو (یا اعضا) با واژه‌ی `using`، همچنان از اپراتور `::` همراه با نام `namespace` برای دسترسی به آن عضو (اعضا) می‌توان استفاده کرد. در مثال قبل به جای دستورات زیر:

```
a = 4;
print_var();
```

میتوانستیم از دستورات زیر استفاده کنیم:

```
np1::a = 4;
```

```
np1::print_var();
```

در C++ تمام توابع، اشیا و کلاس‌های موجود در کتاب خانه‌ی استاندارد در یک `std` به نام `namespace` قرار دارند. در مثال‌های کتاب زمانی که نیاز به اجزای `std` مثل `cout` شد، دستور `using namespace std;` را در برنامه قرار میدادیم تا در هر استفاده از این اعضاء، عبارت `std::` را قبل از نام آنها نویسیم. البته معرفی کل یک `namespace` در برنامه‌های بزرگ توصیه نمی‌شود؛ زیرا برای مثال اگر یک اسمی در برنامه اعلام شود به طوری که به طور اتفاقی همنام با یکی از عناصر موجود در `namespace` `list` و یا `map` از `std` باشد، کمپایلر با خطأ مواجه خواهد شد. در این حالت با نوشته نشدن عبارت:

```
using namespace std;
```

این ابهام از بین می‌رود و فقط با `std::` به اعضای `std` دسترسی ایجاد می‌شود.

الپس^۱ یک `namespace`

یک `namespace` می‌تواند الپس یک `namespace` دیگر تعریف شود. به این معنی که تمام اعضای `namespace` دوم در `namespace` اول نیز قرار خواهند گرفت. با فرمت زیر:

```
namespace np1 // existing namespace
{
    // members
}
namespace np2 = np1;
```

مثال ۴) مثال ساده‌ای از الپس `:namespace`

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

¹ Alias

```
namespace np1 {
    int a;
}
namespace np2 = np1;
int main() {
    np2::a = 4;
    cout << np2::a << '\n';
}
```

خروجی برنامه به صورت زیر است:

4

تعريف namespace در namespace

در C++ امکان تعریف یک namespace در یک namespace دیگر وجود دارد که namespace نام دارد. در این نوع namespace nested namespaces به اعضای namespace بیرونی دسترسی مستقیم دارد ولی بر عکس آن صحیح نیست.

مثال:

```
namespace Outer {
    int a = 10;
    namespace Inner {
        int b = 20;
        int function1() {return a;}
    }
    int function2() {return Inner::b;}
}
```

از C++17 به بعد، nested namespace زیر را:

```
namespace A {
    namespace B {
        namespace C {
        }
    }
}
```

```
}
```

به صورت زیر میتوان تعریف کرد:

```
namespace A:::B:::C {
}
```

inline namespace

اعضای nested namespace در inline namespace مانند اعضای
namespace بیرونی (برخلاف اعضای داخلی) عمل میکنند. برای مثال:

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

namespace A {
    inline namespace B {
        void f() { cout << "string\n"; }
    }
}

int main() {
    A::f(); // string
}
```

برای مدیریت نسخههای کتابخانه استفاده میشود. مثلا: inline namespace

```
namespace X {
    namespace v10{
        void f() { /* ... */ }
    }
    inline namespace v11 {
        void f() { /* ... */ }
    }
}
```

اگر تابع `f()` در `v11`, یک نسخهی جدیدتر از تابع موجود در `v10` باشد, با قرار گرفتن آن, دستور زیر:

```
X::f();
```

به تابع `f()` از نسخهی `v11` تلقی خواهد کرد. با اضافه شدن نسخهی جدید (مثلا `v12`) و `inline` قرار گیری آن، `f(X)` در این حالت به تابع `f()` از نسخهی جدیدتر (`v12`) رجوع خواهد کرد. در صورت نیاز به به تابع `f` از نسخهای قدیمی، باید اسم `namespace` همراه با `::` به طور مختص نوشته شود. برای مثال:

```
X::v10::f();
X::v11::f();
```

^۱ پیشپردازنده

دستورات پیشپردازنده^۲ یک سری دستور هستند که توسط پیشپردازنده و پیش از دستورات دیگر در C++ اجرا می‌شوند. این دستورات با علامت هش (#) شروع می‌شوند و با یک خط جدید (بدون ؛) نیز به پایان میرسند. دستور پیشپردازنده‌ای که در تمام مثالهای کتاب استفاده کردیم، دستور `#include` میباشد که در گذشته به عملکرد آن نیز پرداختیم.

دستورات پیشپردازنده شامل ۵ دسته میشوند:

۱. تعریف ماکرو^۳
۲. ساختار شرطی
۳. دستور `#error`
۴. دستور `#line`
۵. افزودن کد منبع با `#include`

تعریف ماکرو

ماکروها در کد منبع واژگانی هستند که در صورت وجود، جایشان با یک جایگزین عوض می‌شود. تعریف ماکرو با دستور پیشپردازنده‌ی `#define` و با فرمت زیر صورت میگیرد:

`#define macro replacement`

نام ماکرو و `replacement` مقداری است که در کد منبع جایگزین ماکرو می‌شود.

(۱) مثال

`#include <iostream>`

^۱. Preprocessor

^۲. Preprocessor directives

^۳. Macro

```
using namespace std;

#define name "Iman"
#define age 20
int main() {
    cout << "name = " << name << " and age = " << age
    << '\n';
}
```

خروجی برنامه به صورت زیر است:

```
name = Iman and age = 20
```

توضیح) پس از اتمام مرحله‌ی پیش پردازش، مقادیر "Iman" و 20 به ترتیب با **name** و **age** جایگزین می‌شوند.

مقدادر جایگزینی ماکروها نوع داده‌ی خاصی ندارند و فقط در کد منبع جایگزین ماکروها می‌شوند. همچنین یک ماکرو میتواند بدون مقدار جایگزینی نیز تعریف شود. برای مثال:

```
#define X
```

در گذشته، یک نمونه از کاربردهای این ماکروها را در گارد **#include** مشاهده کردیم.

ماکروهای با پارامتر

ماکروها را با پارامتر می‌توان تعریف کرد که در این صورت مشابه یک تابع عمل خواهند کرد. این نوع ماکروها، ماکروهای شبه تابع^۱ نام دارند. تابع زیر را برای مثال:

```
int square(int a) {
    return a*a;
}
```

با ماکرو می‌تواند به صورت زیر بازنویسی کرد:

```
#define square(a) (a*a).
```

^۱. Function-like macro

در هنگام کار با توابع ماکرو توجه کنید که استفاده از پرانتز برای تعیین اولویتها باعث جلوگیری از خطاهای احتمالی میشود. تابع ماکرو همچنین میتواند با ساختار شرطی یک مقدار را جایگزین کند. برای مثال تابع:

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

با ماکروی زیر میتواند بازنویسی شود^۱:

```
#define max(a, b) ((a < b) ? (a) : (b))
```

مثال ۲) برنامه‌ای که برای تشخیص بزرگی ۲ مقدار، از تابع ماکرو استفاده می‌کند:

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

#define max(a,b) ((a > b) ? (a) : (b))
int main() {
    int a = 3, b = 5;
    cout << "Maximum value is " << max(a,b) << '\n';
}
```

خروجی برنامه به صورت زیر است:

```
Maximum value is 5
```

در توابع ماکرو، از ۲ اپراتور # و ## برای مقدار جایگزین آن‌ها می‌توان استفاده کرد. # مقدار رشته‌ای عبارت را جایگزین می‌کند. برای مثال عبارت:

```
#define str(a) #a
cout << str(Dennis) << '\n';
```

¹. اپراتور ?: اپراتور شرطی نام دارد.

`cout` را به دستور زیر تبدیل می‌کند:

```
cout << "Dennis" << '\n';
```

اپراتور `##` ۲ پارامتر اطراف خود را به هم می‌چسباند (۲ عبارت باید با فاصله نوشته شوند).

برای مثال:

```
#define concat(a,b) a ## b
concat(c, out) << "Dennis\n";
```

`cout` را به دستور زیر تبدیل می‌کند:

```
cout << "Dennis\n";
```

ماکروهای از پیش تعریف شده^۱

برای کمپایلرها یکسری ماکروها از پیش تعریف شده است. این ماکروها هم شامل ماکروهای استاندارد و هم غیر استاندارد می‌شود. پرکاربردترین ماکروهای استاندارد عبارت‌اند از:

- `DATE` تاریخ کمپایل را با فرمت `YYYY:MM:DD` جایگزین می‌کند.
 - `TIME` ساعت کمپایل را با فرمت `HH:MM:SS` جایگزین می‌کند.
 - `FILE` نام فایل منبعی که در آن نوشته می‌شود را جایگزین می‌کند.
 - `cplusplus` در کمپایلرهای `C++` تعریف شده است و مقدار آن برابر با نسخه‌ی استانداردی است که برای کمپایل برنامه استفاده می‌شود (`C++17` برای `201703L`).
 - یکی از کاربردهای این ماکرو بررسی استفاده از دستورات مختلف، مناسب با استاندارد استفاده شده برای کمپایل است.
 - `LINE` شماره خطی که این ماکرو در آن نوشته شده است را جایگزین می‌کند.
- در صورت نیاز به لیست کامل، به رفرانس `C++` مراجعه کنید.

¹. Predefined macros

برخی از ماکروهای از پیش تعریف شده‌ی غیر استاندارد - که فقط مختص یک سیستم یا بسته‌ی کمپایلاند - جهت آشنایی عبارت اند از:

- در اکثر سیستم‌های شبه یونیکس، unix
- BSD در سیستم‌های bsd با کرنل 3.4 به بالا تعریف شده است.

توجه کنید که از دستورات پیش‌پردازنده‌ها بیش از موقع ضروری نباید استفاده شود. وجود تعداد زیادی این دستورات، زمان مورد نیاز برای کمپایل را بالا می‌برند و خوانایی کد را نیز کاهش دهنند.

دستورات شرطی^۱ (**#ifdef** ، **#ifndef** ، **#if** ، **#endif** ، **#else#** و **#elif**)

این دستورات یک سری کد را با شرطی خاص به کد منبع می‌افزایند. دستور #
این دستورات مخفف if defined است و فرمت آن به صورت زیر است:

```
#ifdef MACRO
// controlled text
#endif
```

که در صورت تعریف مacroی MACRO پس از خروج از مرحله‌ی پیش پردازش controlled text را به کد منبع می‌افزاید. endif# نیز پایان ساختار شرط را در پیش‌پردازنهای مشخص می‌کند.

دستور #ifndef مخفف if not defined است و برعکس #ifdef عمل می‌کند: در صورت عدم تعریف مacro دستورات را به کد منبع می‌افزاید.

دستورات #if و #elif و #else # می‌توانند یک عبارت ثابت مثل لیترال‌های عدد صحیح و یا مacroی تعریف شده با #define را به عنوان تست شرط بپذیرند. با فرمت زیر:

```
#if value
```

^۱. Conditional directives

```
// ...
#ifndef another_value
// ...
#else
// ...
#endif
```

مثال:

```
#if SIZE>200
#undef SIZE
#define SIZE 200
#elif SIZE<50
#undef SIZE
#define SIZE 50
#else
#undef SIZE
#define SIZE 100
#endif
```

دستورات شرطی پیشپردازندۀ اغلب به ۲ دلیل نوشته می‌شوند:

۱. زمانی که کد منبع برنامه‌ها در سیستم‌های مختلف نیازمند خروجی‌های متفاوت باشد. مثلاً همانطور که گفته شد یک سری ماکروها در سیستم‌های خاصی تعریف شده‌اند؛ در نتیجه با ساختارهای شرطی می‌توان تعریف یک ماکرو را بررسی کرد و در صورت تعریف، کدهای مورد نیاز به برنامه افزوده شوند.
۲. با یک کد منبع، دو برنامه‌ی مجزا کمپایل شود. برای مثال در یکی از برنامه‌ها، داده‌های ایجاد شده در یک فرایند - جهت اشکالزدایی - در خروجی چاپ شوند.

#error دستور

کمپایلر رویه‌ی کمپایل را در صورت مشاهده‌ی این دستور متوقف می‌کند. فرمت آن به صورت زیر است:

```
#error optional_message
```

که optional_message پیام خطای اختیاری است که کمپایلر با متوقف کردن کمپایل در خروجی چاپ می‌کند. نوشتن این پیام اختیاری است. یکی از کاربردهای این دستور اطمینان از استفاده از کمپایلر C++ است:

```
#ifndef __cplusplus
#error Use ++C Compiler..
#endif
```

در صورت عدم تعریف __cplusplus، اگر کد منبع با کمپایلر غیر از کمپایلر C++ - مثل C - شروع به کمپایل شده، با رسیدن به دستور #error متوقف خواهد شد.

دستور #line

زمانی که کد منبع را کمپایل می‌کنیم اگر یک سری خطای خطا را توسط کمپایلر مشاهده شود، کمپایلر نام فایل و شماره‌ی خط خطا را در خروجی نمایش می‌دهد. با استفاده از دستور #line می‌توان هم نام فایل و هم شماره‌ی خط خطا را تغییر داد. با فرمت زیر:

```
#line lineNumber "FileName"
```

شماره‌ی خطی است که شمارش خطهای کد منبع - از خط بعد از این دستور - با مقدار آن صورت می‌گیرد. FileName نیز نام فایلی است که به جای نام اصلی فایل چاپ خواهد شد.

مثال ۳) برنامه‌ای که شامل یک فرمان نادرست است و پیغام خطای ناشی از کمپایل آن، با دستور #line تغییر می‌یابد:

```
#line 22 "Calc"
int main(){ // line 22
    5 = 4; // line 23
}
```

قالب تابع

قالب‌ها اساس برنامه نویسی جنریک (عمومی) هستند و این امکان را به برنامه میدهند که به صورت مستقل از نوع داده‌ی خاصی کد نویسی کنیم.

برای مثال اگر بخواهیم یک تابع ساده بنویسیم به طوری که ۲ آرگومان از نوع `int` بگیرد و جمع آن‌ها را برگرداند باید تابعی مشابه زیر تعریف کنیم:

```
int sum(int a, int b){  
    return a+b;  
}
```

حال فرض کنید بخواهیم این تابع مقادیر `double` بگیرد و جمع آن‌ها را نیز از نوع `double` برگرداند. برای این کار، تابع باید به صورت زیر سربارگذاری شود:

```
double sum(double a, double b){  
    return a+b;  
}
```

و همچنین در صورت نیاز برای پارامترهای `std::string` (یا هر نوع داده‌ی دیگر) نیز یک تابع باید ایجاد و در موقعیت مناسب فراخوانی شود. برای جلوگیری از این کار در `C++` قالب‌ها وجود دارند.

قالب تابع بلوکی مانند تابع است که میتواند در پارامتر خود انواع داده‌های عمومی^۱ داشته باشد؛ به این معنا که هنگام فراخوانی این انواع داده‌ها بر اساس نوع آرگومانها به نوع داده‌ی مناسب تبدیل می‌شوند. ایده‌ی کلی استفاده از قالبها ایجاد توابع عمومی بدون استفاده از نوع داده‌ی خاصی برای همه یا بعضی از پارامترها، مقادیر تعریف شده در بدنه و یا نوع بازگشتی تابع است.

فرمت تعریف قالب تابع به صورت زیر است:

```
template <class T> returnType f() {  
//...  
}
```

¹. Generic types

یا:

```
template <typename T> returnType f() {
//...
}
```

در بالا T یک پارامتر قالب است که می‌تواند جایگزین انواع داده‌های مختلف شود. tfunc نیز نام قالب تابع است. واژه‌ی typename با class معنی یکسان دارد؛ از نظر تاریخی در نسخه‌های اولی C++ برای عدم معرفی واژه جدید (در نتیجه زیاد نشدن واژه‌های از پیش تعریف شده) فقط واژه class در قالبها وجود داشت. کمیته‌ی C++ برای رفع ابهام بین واژه‌ی class در قالبها و کلاس‌ها، علاوه بر class واژه‌ی typename را نیز اضافه کرد.

مثالی از اعلان قالب تابع:

```
template <typename T> void f(T x){
//...
}
```

در اینجا T پارامتر قالب و f قالب تابع است. فرض کنیم قالب تابع f دو بار به صورت زیر فراخوانی شود:

```
f ('c'); // c is a char variable
f (5.3); // 5.2 is a float variable
```

در اولین فراخوانی پارامتر قالب T به نوع داده‌ی char و در دومین فراخوانی به float تبدیل می‌شود. انواع داده‌های که جایگزین پارامتر قالب می‌شوند آرگومان قالب نام دارد. درمثال بالا float و char آرگومان‌های قالب هستند.

قالب تابع به این علت قالب نام دارند که مانند یک قالب عمل می‌کنند؛ به این معنی که بسته به نوع فراخوانی، تابع مناسب برای نوع آرگومان‌ها تولید می‌کند.

پارامتر قالب معمولاً با یک حرف بزرگ نام گذاری می‌شود. مثل T1، T2 و

(۱) مثال

```
#include <iostream>
using namespace std;
template<typename T> void printExpr(const T& x){
    cout << x << '\n';
}

int main(){
    int a = 4;
    char c = 's';
    printExpr(a);
    printExpr(c);
    printExpr(10);
}
```

خروجی برنامه به صورت زیر است:

```
4
s
10
```

قالب تابع با بیش از یک پارامتر قالب برای تعریف قالب تابع با بیش از یک پارامتر باید، در هنگام تعریف، پارامترهای قالب را تفکیک شده با کاما نوشت. با فرمت زیر:

```
template <typename T1, typename T2, ...> func;
```

(۲) مثال

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

template <typename T1, typename T2>
void printVars(T1 a, T2 b){
    cout << a << '\n';
    cout << b << '\n';
}
int main(){
```

```

int a = 4;
char b = 'C';
printVars(a,b);
}

```

خروجی برنامه به صورت زیر است:

```

4
C

```

در قالب تابع، لازم نیست همه‌ی پارامترهای تابع از نوع پارامتر قالب باشند. پارامترها می‌توانند از انواع دیگر مانند **int** و ... نیز قرار گیرند. مثلاً^(۳) برنامه‌ای که شامل یک قالب تابع به نام **printNTimes** است و دو آرگومان می‌گیرد و آرگومان اول را به تعداد آرگومان دوم نمایش می‌دهد:

```

#include <iostream>
using namespace std;

template <typename T>
void printNTimes(const T &x, int n){
    for(int i = 0; i < n; ++i) {
        cout << i << ". " << x << '\n';
    }
}

int main(){
    printNTimes("string",3);
    printNTimes(3,2);
}

```

خروجی برنامه به صورت زیر است:

```

0. string
1. string
2. string
1. 3
2. 3

```

سربارگذاری قالب تابع

قالبهای توابع، توابع عادی را میتوانند سربارگذاری کنند. در این صورت اگر فراخوانی تابع عادی قابل اجرا باشد، اجرای آن نسبت به قالب تابع اولویت مییابد.

مثال (۴)

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

template <typename T> void f(T x){
    cout << "template.\n";
    cout << x << '\n';
}
void f(int x){
    cout << "nontemplate.\n";
    cout << x << '\n';
}

int main(){
    f(4);
    f(1.5);
}
```

خروجی برنامه به صورت زیر است:

```
nontemplate.
4
template.
1.5
```

با سربارگذاری قالبهای توابع با تابع عادی میتوان تعریف قالب تابع را برای یک یا چند نوع خاص مجزا کرد. برای مثال:

```
template <typename T> void print(const T &x) {
    cout << x << '\n';
```

```

}

void print(bool x){
    cout << boolalpha << x << '\n';
}

print(3);      // 3
print(false);   // false

```

نتیجه‌گیری^۱ انواع در قالبها

زمانی که قالب تابع فراخوانی می‌شود برنامه سعی به تعیین انواع قالب می‌کند. به این تعیین نوع، نتیجه‌گیری نوع داده گفته می‌شود. در یک مثال ساده با دستورات زیر:

```

template<typename T>
void f(T x) {
// ...
}

int v = 10;
f(v)

```

نوع T به int، نتیجه‌گیری می‌شود. در صورتی که پارامتر رفرنس نباشد (مانند مثال بالا)، اگر نوع آرگومان رفرنس باشد، ابتدا قسمت رفرنس نوعش نادیده گرفته می‌شود؛ سپس اگر عبارت const بود، بودن آن نیز نادیده گرفته می‌شود. در این نوع قالب تابع، دریافت آرگومان به منظور کپی آنها نوشته شده به همین دلیل نیز پارامتر آن از نوع const تعیین نمی‌شود.

برای مثال:

```

template<typename T>
void f(T param){}

```



```

int x = 5;
const int constx = x;

```

^۱. Deduction

```
const int& refx = x;

f(x) // T = int
f(constx); // T = const int
f(refx); // T = int
```

در حالت دوم اگر پارامتر یک رفرنس باشد، مانند:

```
template<typename T> void f(T& x);
```

قسمت رفرنس آرگومان حذف میشود سپس نوع داده‌ی T در پارامتر به صورت نظیر به نظیر تعیین میشود.

برای مثال:

```
template<typename T> void f(T& x){}
```

```
int x = 5;
const int constx = x;
const int& refx = x;
```

```
f(x) // T = int
f(constx); // T = const int
f(refx); // T = const int
```

در حالت آخر اگر رفرنس **(&& rvalue)** در قالب تابع نوشته شود، نوعی به اسم رفرنس ارسالی^۱ ایجاد میشود. برای مثال:

```
template <typename T> void f(T&& x){}
```

نتیجه‌گیری در این نوع رفرنس به گونه‌ای است که اگر آرگومان پاس داده شده باشد، آرگومان قالب از نوع رفرنس **lvalue** قرار میگیرد؛ در غیر این صورت نتیجه‌گیری عادی صورت خواهد گرفت. برای مثال:

```
template <typename T> void f(T&& x){}
int i;
f(i); // f<int&>(int& &&)
```

¹. Forwarding reference

```
f(0); // f<int>(int&&)
```

در دستورات بالا در اولین فراخوانی با تعیین T به $\&int$ ، در پارامتر f همانطور که در توضیحات مقابلش نوشته شده، ایجاد نوع رفرنس به رفرنس میشود. در C++، امکان تعریف رفرنس به رفرنس وجود ندارد و با نوشته شدن چنین عبارتی مستقیماً توسط برنامه نویس، ایجاد خطأ میشود. با این وجود در مکانهایی مثل نتیجه‌گیری بالا که عبارت رفرنس به رفرنس ایجاد میشود، کمپایلر با روش زیر رفرنس به رفرنس را در نهایت به یک رفرنس تبدیل میکند (TR نوع رفرنس به T است):

TR	R
$T\&$	$\& -> T\&$
$T\&$	$\&\& -> T\&$
$T\&\&$	$\& -> T\&$
$T\&\&$	$\&\& -> T\&\&$

این عملکرد حذف شدن رفرنس¹ نام دارد. به این ترتیب در فراخوانی:

```
f(i); // f<int&>(int& &&) -> f<int&>(int&)
```

عبارت $\&int \&\& &int$ به $\&int$ تبدیل میشود. عملکرد رفرنس ارسالی به این ترتیب، به نحوی است که اگر عبارت $lvalue$ به قالب تابع پاس داده شود، پارامتر از نوع رفرنس $lvalue$ ، در غیر این صورت از نوع رفرنس $rvalue$ تعیین میشود. کاربرد اصلی رفرنس ارسالی در پاسداهن آرگومانهای دریافت شده به توابع دیگر است. با تعریفی که تاکنون از قالب تابع با رفرنس ارسالی داریم، با وجود اینکه هم میتوان $rvalue$ و هم $lvalue$ به آن پاس داد، پارامتر یک عبارت $lvalue$ است؛ در نتیجه به عنوان یک عبارت $lvalue$ نیز به توابع دیگر پاس داده میشود. در این تابع برای حفظ کردن عبارت $lvalue$ بودن آرگومان، میتوان پارامتر به صورت زیر به $\&\&T$ گست کرد. برای مثال:

```
void f1(int& x){
    cout << "x refers to lvalue\n";
```

¹. Reference collapsing

```

}

void f1(int&& x){
    cout << "x refers to rvalue\n";
}

template<typename T> void f2(T&& x){
    f1(static_cast<T&&>(x));
}

int x{1};
f2(x); // x refers to lvalue
f2(1); // x refers to rvalue

```

نحوه‌ی حفظ شدن نوع آرگومان با کست کردن به $\&\&T$ به گونه‌ای است که اگر مقدار $\&A$ به قالب تابع پاس داده شود، نوع T به $\&A$ تبدیل می‌شود و با کست کردن به $\&\&T$ ، همان نوع باقی می‌ماند ($\&A \&\& A$)؛ اگر آرگومان پاس داده شده $rvalue$ باشد نوع T به A تبدیل می‌شود و با کست کردن به $\&\&T$ ، به $\&\&A$ تبدیل می‌شود که به نوع اصلی خود برمی‌گردد. به عمل دریافت آرگومان (ها) و استفاده از آن برای فراخوانی یک تابع دیگر، با حفظ ماندن نوعشان ($rvalue$) یا $\&A$ بودن/ $const$ بودن، ارسال کامل^۱ گفته می‌شود. اکنون که با نحوه‌ی ارسال کامل با کست کردن به $\&\&T$ آشنا شدیم، به بررسی $std::forward$ می‌پردازیم. این قالب تابع، در `<utility>` قرار دارد و کست کردن به $\&\&T$ را با `utility` کوتاه‌تر انجام میدهد. برای مثال دو دستور زیر در قالب تابعی که رفرنس ارسالی دریافت می‌کند برابراند:

```

f1(static_cast<T&&>(x));
f1(std::forward<T>(x));

```

(آرگومان قالب $std::forward$ ، به جای $\&\&T$ ، خود T است). در مثال زیر، ارسال

^۱. Perfect forward

کامل را با `std::forward` در یک برنامه مشاهده میکنیم.

(۵) مثال

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

void f1(int& x){
    cout << "x refers to lvalue\n";
}

void f1(int&& x){
    cout << "x refers to rvalue\n";
}

template<typename T> void f2(T&& x){
    f1(forward<T>(x));
}

int main() {
    int x = 1;
    f2(x);
    f2(1);
}
```

خروجی برنامه به صورت زیر است:

```
x refers to lvalue
x refers to rvalue
```

لازم به ذکر است که نتیجهگیری در `auto` نیز به روش نتیجهگیری در قالبها صورت می‌گیرد با این تفاوت که در دستور زیر:

```
auto x1 = 10;
```

`x1` از نوع `int` تعریف میشود و در دستور زیر:

```
auto x2 = {10};
```

۲۰ اعلان خواهد شد.
با یک عنصر `<std::initializer_list<int>` از نوع

در قالب‌های توابع با وجود بعضی از شرایط، نتیجه‌گیری پارامترها به طور خودکار امکان پذیر نیست. برای مثال:

```
template<typename T> void f(){
    T x = 10;
    cout << x << '\n';
}
f(); // error
```

در این نوع قالبها، به صورت زیر می‌توان نوع پارامترهای قالب را به طور مختص تعیین کرد:

```
func < argument-list > ( parameter-list ) ;
```

برای مثال:

```
template<typename T> void f(){
    T x = 10;
    cout << x << '\n';
}
f<int>();
```

در قالب‌هایی که نتیجه‌گیری صورت می‌گیرد نیز از این روش برای مقداردهی پارامتر(های) قالب می‌توان استفاده کرد. مثال:

```
template<typename T> void f(T x){
    cout << x << '\n';
}

f<double>(10);
```

در این مثال با فراخوانی عادی `f`، مثل `f(10)` نوع `T` به `int` تبدیل می‌شود که با تعیین آن به صورت `double`، از این نوع نتیجه‌گیری جلوگیری می‌شود. در مثالی دیگر اگر به قالب تابع `f`، رشته به صورت لیترال پاس داده شود (مثل `(f("aaa"))`) نوع `T` به

`std::string` تبدیل میشود؛ در این حالت نیز برای تعیین `T` به `char* const` میتوان به صورت زیر عمل کرد:

```
f<std::string>("aaa");
```

پارامتر غیر نوع و قالب قالب

پارامترهای قالب علاوه بر نوع داده بودن، به صورت **غیر نوع**^۱ و یا **قالب قالب**^۲ نیز میتوانند باشند. پارامتر غیر نوع به جای نوع داده، بیانگر مقدار است. این پارامترها، با نوشه شدن نوع داده‌ی خاص (مثل `int` و ...) در پارامتر قالب تعیین میشوند. مقادیری که با این نوع پارامترها جایگزین میشوند باید از نوع ثابت باشند و کمپایلر نیز این قالبها را در حین کمپایل مقداردهی میکند. مثال زیر برنامه‌ای است که از پارامترهای غیر نوع استفاده میکند.

(۶) مثال

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

template<int n, typename T>
void printNTimes(const T& x){
    for(int i = 0; i < n; ++i)
        cout << x << ' ';
    cout << '\n';
}

int main() {
    printNTimes<5,int>(1);
    printNTimes<5>(2);
}
```

خروجی برنامه به صورت زیر است:

- ^۱. Non-type template parameters.
- ^۲. Template template parameters.

1	1	1	1	1
2	2	2	2	2

توضیح) در اولین فراخوانی، همهی پارامترهای قالب مشخص و در فراخوانی دوم از آرگومان پاس داده شده (۲) به نوع T در برنامه نتیجه‌گیری می‌شود. اگر در این مثال قالب تابع `printNTimes` را مانند گذشته مینوشتیم (که مقدار n به عنوان پارامتر تابع دریافت می‌شد) خروجی مشابهی به وجود می‌آمد؛ با این تفاوت که چون پارامترهای غیر نوع در حین کمپایل ارزیابی می‌شوند، در مکانهایی که نیاز به این نوع ارزیابی است (مانند تعیین عناصر آرایه) مورد استفاده قرار می‌گیرند.

در C++17، نوع پارامترهای غیر نوع میتواند با `auto` نیز تعیین شوند. مثال:

```
template<auto n> void f() {
    //...
};  
f<10>();
```

قالب کلاس و متغیر

قالب کلاس^۱

قالب کلاس جز دیگر قالبهای است که امکان ایجاد مجموعهای از انواع داده را فراهم میکند.
فرمت تعریف قالب کلاس و اعلان شی از آن به صورت زیر است:

```
template <class type> class A{  
};  
A <DataType> objName;
```

نوع قالب کلاس، برخلاف قالب تابع، باید با ساخت شی و به صورت مختص تعیین شود.
در C++17 با مقداردهی اولیه‌ی شی نیز این کار امکان پذیر است.

مثال ۱) برنامه‌ای که شامل یک قالب کلاس است و در سازنده خود دو مقدار را گرفته و
با آن‌ها دو متغیر عضو کلاس را مقداردهی می‌کند. همچنین یک تابع به نام
printMax برای نمایش بزرگترین مقدار شی وجود دارد:

```
#include <iostream>  
using namespace std;  
  
template <class T> class MyClass{  
private:  
    T a, b;  
public:  
    MyClass(T a, T b): a{a}, b{b} {}  
    void printMax(){  
        cout << (a>b?a:b) << '\n';  
    }  
};  
  
int main(){  
    MyClass<int> m{3,7};  
    m.printMax();
```

^۱. Class template

}

خروجی برنامه به صورت زیر است:

7

تعریف تابع عضو قالب کلاس - مانند تابع عضو عادی - خارج از کلاس امکان‌پذیر است.
با فرمت زیر:

```
template <parameter-list> class id{
public:
    returnType function();
}

template <parameter-list> returnType
class<T>::function (){
    //...
}
```

قالب کلاس مانند قالب تابع، می‌تواند چند پارامتر قالب داشته باشد.

به صورت زیر:

```
template <typename T1, typename T2, ...> class
MyClass{};
```

(۲) مثال

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

template<typename T1, typename T2> class MyClass{
private:
    T1 a;
    T2 b;
public:
    MyClass(T1 a, T2 b):a{a}, b{b} { }
    void print(){
        cout << "a = " << a << '\n';
        cout << "b = " << b << '\n';
    }
}
```

```

    }
};

int main(){
    MyClass<int, char> m{1,'c'};
    m.print();
}

```

خروجی برنامه به صورت زیر است:

```
a = 1
b = c
```

در قالب‌های کلاس می‌توان انواع داده‌های پیش فرض را با `=` در لیست پارامتر قالب برای یک سری از آنها تعریف کرد. با عدم تعیین آرگومان(های) قالب، این آرگومانهای پیش فرض در نظر گرفته می‌شوند.

مثال (۳)

```

template<typename T1 = int, typename T2 = char>
class MyClass {
private:
    T1 a;
    T2 b;
public:
    MyClass(T1 a, T2 b): a{a}, b{b} { }
    void print(){
        cout << "a = " << a << '\n';
        cout << "b = " << b << '\n';
    }
};

MyClass<> m(1,'c');
MyClass<int, float> m2(4,3.5);
m.print();
m2.print();

```

خروجی زیر را به همراه خواهد داشت:

```
a = 1
b = c
a = 4
b = 3.5
```

در مثال بالا برای ساخت شی P0، آرگومان قالب را ننوشتیم و کمپایلر به طور پیش فرض به ترتیب `int` و `char` را برای پارامترها قرار داد. اگر یک پارامتر پیش فرض تعیین شود، تمام پارامترهای بعد از آن (در لیست قالب) نیز باید از نوع پیش فرض قرار گیرند.

اختصاصی کردن قالب کلاس^۱

تعریف قالب کلاس، مانند قالب تابع، به ازای نوع(های) داده‌ی خاصی می‌تواند مجزا شود. قالب کلاس تخصصی شده از نظر اسم با قالب کلاس هم نام است. همچنین قالب کلاس اختصاصی شده به عنوان یک کلاس مجزا در برنامه شناخته می‌شود، به این معنی که میتواند اعضای متفاوتی داشته باشد.

با فرمت زیر:

```
template <typename T> class MyClass{};
template <> MyClass<specialDataType>{};
```

مثال ۴) برنامه‌ای که شامل قالب کلاس تخصصی است:

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

template <typename T>
class MyClass{
private:
    T a;
public:
    MyClass(char a): a{a} {
        cout << "obj constructed for generic types\n";
    }
}
```

^۱. Class template specialization

```

    }
};

template <> class
MyClass<char>{
private:
    char a;
public:
    MyClass(char x): a{x}{
        cout << "obj constructed for char\n";
    }
    void print_from_char(){
        cout << "printing from specialized obj for
char\n";
    }
};

int main() {
    MyClass<int> m{2};
    MyClass<char> m2{'x'};
    m2.print_from_char();
}

```

خروجی برنامه به صورت زیر است:

```

obj constructed for generic types
obj constructed for char
printing from specialized obj for char

```

علاوه بر امکان تخصصی کردن تمام پارامترهای قالب کلاس (مثل قالب تابع)، میتواند قسمتی از پارامترهای قالب کلاس را فقط اختصاصی کرد.^۱ با ساختاری مشابه زیر:

```

template<typename T1, typename T2, int I>
class MyClass {};// primary template

template<typename T, typename T2, int I>
class MyClass<T*, T2, I> {};// partial

```

¹. Partial template specialization

specialization: when T1 is ptr

(۵) مثال

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

template<class T1, class T2 >
class MyClass { // primary template
public:
    MyClass(){
        cout << "generic MyClass object created.\n";
    }
};

template<class T>
class MyClass<T, bool> { // partially specialized
public:
    MyClass(){
        cout << "2nd argument is bool.\n";
    }
};

template<class T>
class MyClass<T, T*> { // partially specialized
public:
    MyClass(){
        cout << "2nd argument is pointer of 1st arg.\n";
    }
};

int main() {
    MyClass<int,int> m;
    MyClass<int,bool> m2;
    MyClass<int,int*> m3;
}
```

خروجی برنامه به صورت زیر است:

```
generic MyClass object created.  
2nd argument is bool.  
2nd argument is pointer of 1st arg.
```

هنگام تعریف عضو خارج از کلاس اختصاصی شده، قسمت `<>template` نوشته نمیشود:

```
template<typename T>  
class MyClass {  
  
};  
  
template<>  
class MyClass<int> {  
public:  
    void f();  
};  
  
void MyClass<int>::f() {  
    // body  
}
```

به طور کلی اعضای یک کلاس (شامل `struct` و ...) را - جدا از قالب بودن خود کلاس - میتوان به صورت قالب تعریف کرد. برای مثال:

```
class MyClass{  
public:  
    template<typename T> void f(T x){  
        // ...  
    }  
};
```

مخرب و سازندهی کپی را نمیتوان به صورت قالب تابع اعلام کرد. قالب اعضای تابع نمیتوانند `virtual` باشد و همچنین نمیتوانند توابع کلاس پایه‌ی خود را، زمانی که `virtual` را نادیده بگیرند (`override`). اگر یک تابع عضو قالب و یک تابع عادی

در یک کلاس موجود و با فراخوانی آنها هردو قابل اجرا باشند، برنامه تابع عضو غیر قالب را فراخوانی میکند مگر در حالتی که لیست آرگومانهای قالب در حین فراخوانی نوشته شود. برای تعریف قالب عضوها خارج از کلاس، باید دو بار قسمت `template` را نوشت (یکی برای کلاس و دیگری برای عضو). برای مثال:

```
template<typename T>
class MyClass {
    template<typename T2> void f(T2 a);
};

template<typename T>
template<typename T2>
void MyClass<T>::f(T2 a) {
    //...
}
```

قالب متغیر^۱

از C++14 به بعد، امکان تعریف قالب متغیر نیز وجود دارد.

مثال ۶

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

template<typename T> T pi = T(3.141592);

int main() {
    cout << pi<float> << '\t' << pi<int> << '\n';
}
```

خروجی برنامه به صورت زیر است:

3.14159 3

^۱. Variable template

قبل از C++14، پیاده‌سازی قالب متغیر با قرار دادن یک متغیر عضو `static` در قالب کلاس یا `struct` صورت می‌گرفت. این نوع پیاده‌سازی علاوه بر طولانی بودن تعریف آن، استفاده از آن نیز syntax طولانی‌تری دارد. برای مثال اگر قالب متغیر `pi` در مثال ۶ با قالب کلاس ایجاد شود، دسترسی به آن به صورت زیر انجام می‌شود:

```
pi<double>::value
```

ولی با قالب متغیر به شکل زیر صورت می‌گیرد:

```
pi<double>
```

اعلان alias

همانطور که در فصول پیش پرداخته شد با `typedef` می‌توان نام جدیدی را برای نوع داده‌ی موجود اعلان کرد. برای مثال بهجای استفاده از نوع داده‌ی زیر:

```
std::vector<std::vector<std::vector<double>>>
```

در برنامه، با `typedef` می‌توان نام جدیدی برای نوع داده به صورت زیر ایجاد کرد:

```
typedef std::vector<std::vector<std::vector<double>>>
vec3d_double;
```

در نتیجه نوع داده‌ی طولانی فقط یک بار نوشته می‌شود.

در استانداردهای جدید، برای تعریف نام جدید برای نوع داده‌ی موجود، از واژه‌ی `using` نیز می‌توان استفاده کرد. به صورت:

```
using aliasType = existingType;
```

پس دو دستور زیر در استانداردهای جدید برابرند:

```
typedef std::vector<std::vector<std::vector<double>>>
vec3d_double;
```

```
using vec3d_double =
std::vector<std::vector<std::vector<double>>>;
```

فرق اصلی استفاده از `typedef` با اعلان `alias` اینکه ایجاد قالب از اعلان

است (اصطلاحاً قالب alias). برای مثال:

```
template <typename T>
using vec3d =
std::vector<std::vector<std::vector<double>>>;

vec3d<double> v;
vec3d<int> v2;
```

enable_if و SFINAE

اگر یک قالب تابع و یک تابع معمولی هر دو با نام add داشته باشیم:

```
int add(int x1, int x2) {
    return x1 + x2;
}
```

```
template <typename T>
typename T::add_return_type add(T x1, T x2) {
    return x1 * x2;
}
```

با فراخوانی add با دستور:

```
add(4,10);
```

تابع معمولی با نوع بازگشتی int فراخوانی میشود؛ زیرا در قالب تابع با پاس دادن ۴ و ۱۰ و تعیین شدن T از نوع int، نوع بازگشتی قالب (int::add_return_type) تعیین شده است و برنامه (بدون ایجاد خطا) تابع معمولی را فراخوانی میکند. در قالب تابع به طور کلی اگر جایگذاری نوع نتیجه‌گیری شده برای قالب (و نوع بازگشتی آن) صحیح نباشد قالب به ازای آن نوع از لیست توابع قابل اجرا، بدون خطا، حذف میشود؛ این عملکرد SFINAE^۱ نام دارد. با وجود اینکه SFINAE یک عملکرد کمپایلر است، معمولاً از آن به عنوان یک قابلیت C++ استفاده میشود. یکی از مهمترین کاربردهای

¹. Substitution Failure Is Not An Error

تابع `std::enable_if` تابع `type_traits` موجود در هدر است. این قالب تابع تعريف مشابه زير دارد:

```
template<bool B, typename T = void>
struct enable_if {
};

template<typename T>
struct enable_if<true, T> {
    typedef T type;
}
```

و میتوان به صورت زير در قالب تابعها استفاده شود:

```
template <typename T, typename U>
typename enable_if<is_integral<T>::value &&
is_integral<U>::value, T>::type
add(T x1, U x2) {
    cout << "returning ints\n";
    return x1 + x2;
}
```

```
add(1,2);
```

در اين قالب تابع با پاس دادن دو مقدار غير اعشار مثل ۱ و ۲، شى از تعريف دوم استفاده ميکند و درنتيجه عضو `type`، از نوع آرگومان اول ايجاد ميشود. با صحيح بودن شرط `enable_if` (عددی بودن آرگومانها) و جايگذاري نوع آرگومان اول، قالب تابع صحيح خواهد بود در نتيجه به ازاي مقادير عددی اجرا خواهد شد.

در C++14 به بعد نوع `enable_if_t` زير تعريف شده است:

```
template <bool B, typename T = void>
using enable_if_t = typename enable_if<B, T>::type;
```

در نتيجه قالب تابع `add` برای آرگومانهای `int` میتواند به صورت زير نوشته شود:

```
template <typename T, typename U>
enable_if_t<is_integral<T>::value &&
```

```
is_integral<U>::value, T>
add(T x1, U x2) {
    cout << "returning ints\n";
    return x1 + x2;
}
```

عملکرد enable_if_t را در یک برنامه نیز بررسی میکنیم.

مثال (۷)

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

template <typename T, typename U>
enable_if_t<is_integral<T>::value &&
is_integral<U>::value, T>
add(T x1, U x2) {
    cout << "returning int.\n";
    return x1 + x2;
}

template <typename T, typename U>
enable_if_t<is_floating_point<T>::value &&
is_floating_point<U>::value, T>
add(T x1, U x2) {
    cout << "returning float.\n";
    return x1 + x2;
}

int main(){
    add(5,10);
    add(1.1,3.3);
}
```

خروجی برنامه به صورت زیر است:

```
returning int.
```

`returning float.`

قالب variadic

قالب variadic تابع یا کلاسی است که تعداد داخواهی از آرگومان دریافت میکند. تعریف یک قالب تابع variadic به صورت زیر است:

```
template <typename... Args>
void f(Args... args) { }
```

در تعریف قالب تابع `f`, قسمت `Args` بسته‌ی پارامتر قالب^۱ و `args` بسته‌ی پارامتر^۲ تابع نام دارد که به معنی مجموعه‌ای از پارامترها است (نامهای `Args` و `args` اختیاریاند). در قالب تابع variadic جهت استفاده از بسته‌ی پارامتر تابع، از ساختارهای حلقه مثل `for` نمیتوان استفاده کرد؛ بلکه با ساختار بازگشتی باید تفکیک و استفاده شوند. برای مثال:

```
template<typename T>
T sum(T t) {
    return t;
}
```

```
template <typename T, typename... Args>
T sum(T t, Args... args) {
    return t + sum(args...); }
```

این قالب تابع جهت جمع کردن تعداد دلخواهی از آرگومانها استفاده میشود. برای مثال:
`sum(4,3,5,6,7); // 25`

```
string s1 = "aa", s2 = "bb";
sum(s1,s2); // aabb
```

اجرای قالب تابع `sum` به گونه‌ای است که با عبارت `sum(args...)`, قالب `sum` با یک

^۱. Template parameter pack

^۲. Function parameter pack

پارامتر کمتر از قبل اجرا میشود. زمانی که فقط یک پارامتر باقی ماند، قالب تابع `sum` اول (با یک پارامتر) اجرا میشود؛ که نقش حالت پایه (`base case`) در بازگشتی را دارد. برای مثال برای فراخوانی زیر:

```
sum(1,2,3,4);
```

قالبها به صورت زیر اجرا میشوند:

```
T sum(T, Args ...) [with T = int; Args = {int, int, int}]  
T sum(T, Args ...) [with T = int; Args = {int, int}]  
T sum(T, Args ...) [with T = int; Args = {int}]  
T sum(T) [with T = int]
```

std::map و std::vector، std::array

stl¹ قسمتی از کتابخانه‌ی استاندارد است که با قالب‌ها نوشته شده است. stl شامل کامپونتهايی به اسم الگوريتم، کانتينر (container)، فانکتور (functor)، آيتريتور (iterator) است. قسمت کانتينر اين کتابخانه شامل انواع داده‌هاي برای ذخیره‌ی مجموعه‌ای از عناصر مibashد. ابتدا با نوع داده‌ی پركاربرد آن، آشنا ميشويم.

std::vector

شی std::vector مانند آرایه، متغيری برای ذخیره‌ی مجموعه‌ای از عناصر از یک نوع داده است؛ با اين تفاوت که سايز اشيای vector بخلاف آرایه مি�تواند هنگام اجرا تغيير کند. با اپراتورهای سربارگذاري شده در vector، دسترسی و اعمال دیگر روی عناصر تقریبا مشابه آرایه صورت میگیرد. از نظر سرعت شی vector مقداری کندتر از آرایه است.

بعد از اضافه کردن هدر <vector> فرمت تعريف متغير از vector به صورت زير است:

```
vector <dataType> v;
```

نوع داده‌ی عناصر و v متغير dataType است. برای افزودن عنصر به vectorتابع عضو push_back(expr) موجود است که مقدار expr را به انتهای vector میافزاید. برای مثال دستورات زير:

```
vector <int> a;
a.push_back(10); // a = {10}
a.push_back(20); // a = {10, 20}
```

یک شی vector از نوع int ايجاد ميکنند و طی اجرا، ۲ عنصر را به انتهای آن میافزایند.

¹. Standard Template Library

برای حذف عنصر از `vector` از تابع عضو `pop_back()` میتوان استفاده کرد. با فرآخوانی این تابع عضو، آخرین عنصر متغیر حذف میشود.

در استانداردهای جدید C++ با استفاده از {} میتوان شی `vector` را مقداردهی کرد. مقادیر درون {} با `vector` توسط `std::initializer_list` دریافت میشوند.

برای مثال دستور زیر:

```
vector <int> v{1,3,45,213};
```

یک شی `vector` از نوع `int` با عناصر 1, 3 و ... ایجاد میکند.

(مثال ۱)

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

int main() {
    vector <int> v = {2, 4 ,3 ,3};
    for(int i = 0; i < v.size(); ++i) {
        cout << v[i] << '\t';
    }
    cout << '\n';
}
```

خروجی برنامه به صورت زیر است:

2	4	3	3
---	---	---	---

توضیح) تابع عضو `size()`، تعداد عناصر `vector` را برمیگرداند. با اپراتور سربارگذاری شدهی []، مانند آرایه به عناصر `vector` دسترسی پیدا میشود و مقدار آنها نمایش مییابند. توجه شود نوع مقدار بازگشته `size()` از نوع `unsigned int` (یا `std::vector<T>::size_type`) است که میتواند لزوماً نباشد. در نتیجه ممکن است مقدار قابل ذخیره در `i` (از نوع `int`) کمتر از سایز متغیر

باشد. روش صحیحتر ایجاد حلقه در متغیر `v` به صورت زیر است:

```
for(vector<int>::size_type i = 0; i < v.size(); ++i)
{
    cout << v[i] << '\t';
}
```

در اکثر سیستمهای نوع `size_t` با `size_type` یکی است.

مثال ۲) در این مثال نحوه استفاده از `for` مبتنی بر بازه در `vector`‌ها نشان داده شده است.

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

int main() {
    vector <int> v{2, 4, 3, 3};
    v.push_back(24);
    v.push_back(222);
    v.pop_back();
    for(const auto& i : v)
        cout << i << ' ';
    cout << '\n';
}
```

خروجی برنامه به صورت زیر است:

```
2 4 3 3 24
```

مثال ۳) در این مثال نحوه استفاده از بعضی از عضوها و اپراتورهای سربارگذاری شدهی `vector` را مشاهده میکنیم:

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

int main() {
    vector<int> v{1,2};
```

```

vector<int> v2(v); // copy
if(!v.empty()){
    v.clear();
    cout << "v is now empty.";
}
if(v == v2)
    cout << "v == v2 \n";
}

```

خروجی برنامه به صورت زیر است:

v is not empty.

برای اشاره به عناصرهای `vector` مانند آرایه نمیتوان مستقیم از آدرس عناصرها استفاده کرد. برای این کار از متغیر `vector<T>::iterator` استفاده میشود. اپراتورهای این نوع، طوری سربارگذاری شده که مانند یک اشاره‌گر عمل میکند (با `++` به عنصر بعدی، با `*` به مقدار عنصر و ... دست یافته میشود).

با فراخوانی تابع عضو `begin()` و `end()` از متغیر `vector` نیز به ترتیب اولین و آخرین ایتریتور برمیگردد.

فرمت اعلان ایتریتور برای `vector` به صورت زیر است:

`vector<dataType>::iterator it;`

در ایتریتور مانند اشاره‌گر، با اضافه کردن عدد `n`، به عنصر `n`ام از ایتریتور دسترسی ایجاد میشود.

(۴) مثال

```

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

int main() {
    vector<int> v{2,4,3,3};
    for(vector<int>::iterator it = v.begin(); it != v.end(); ++it) {

```

```

    cout << *it << ' ';
}
cout << '\n';
}

```

خروجی برنامه به صورت زیر است:

2 4 3 3

توضیح) تابع عضو `end()`، مکان یکی بعد از آخرین عنصر `vector` را برمیگرداند؛ به همین دلیل در `for`، تمام عنصرها در خروجی نمایش مییابند.
در مثال ۲ هنگام استفاده از `for` مبتنی بر بازه، متغیر `i` نیز به یک ایتریتور تبدیل می‌شود.

در کانتینرها علاوه بر نوع `iterator` و `const_iterator` نیز موجوداند. این نوع ایتریتورها به ترتیب برای عدم تغییر مقادیر و برای ایجاد حلقه بر عکس در عناصر کانتینر استفاده میشوند. نوع `const_reverse_iterator` نیز به صورت ترکیبی از آن دو قرار دارد.
در کانتینرها به طور مشابه علاوه بر `begin()` و `end()` از توابع عضو `rbegin()` و `rend()` برای بازگردانی ایتریتور `reverse_iterator` و `cbegin()` و `cbegin()` و `crend()` و `crbegin()` نیز برای دریافت `const_iterator` و از `const_reverse_iterator` به اولین و آخر عنصر کانتینر استفاده میشود. برای مثال:

```

vector<int> v{1,2,3,4};
for(auto rit = v.rbegin(); rit != v.rend(); ++rit)
    cout << *rit << ' ';

```

خروجی ۱ ۲ ۳ ۴ را ایجاد میکند.

توابع آزاد `std::end` و `std::begin`

همانطور که مشاهده کردیم از توابع عضو `begin()` و `end()` برای دریافت ایتریتور به

اولین و آخرین عنصر `vector` و بقیه‌ی کانتینرهای `stl` میتوان استفاده کرد. به دلیل اینکه این توابع، توابع عضواند، از آنها برای دریافت ایتریتور از آرایه‌هایی معمولی نمیتوان استفاده کرد. نوشتمن دستورات زیر در نتیجه صحیح نیست:

```
template<typename T> void f( const T& t ) {
    for(auto it = t.begin(); it != t.end(); ++it){
        // ...
    }
}

vector<int> v = {1,2,3,4};
int a[] = {1,2,3,4};

f(v); // ok
f(a); // error
```

برای حل این مشکل و کلی کردن تعریف قالب `f`، از توابع آزاد `std::begin` و `std::end` موجود در `<iterator>` میتوان استفاده کرد. قالب تابع `f` به صورت زیر میتواند بازنویسی شود:

```
template<typename T> void f(const T& t ) {
    for(auto it = begin(t); it != end(t); ++it){
        // ...
    }
}

std::rbegin(), std::rend(), std::crend() و std::cbegin(), std::cend(), std::crbegin
به طور مشابه توابع آزاد،
```

نیز موجوداند.

برای افزودن یا حذف کردن عنصر از متغیر `vector` در ایندکس خاص، از توابع عضو `insert` و `erase` استفاده میشود. تابع `erase` به عنوان آرگومان یک ایتریتور به `insert` عنوان مکان ایندکس میگیرد و عنصر مورد اشاره‌ی آن را حذف میکنند. تابع `insert`

علاوه بر ایتریتور، یک مقدار برای اضافه یا کم کردن از متغیر نیز دریافت میکند.

(مثال ۵)

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

int main() {
    vector<int> v{1,1,1,1};
    v.insert(v.begin() + 1, 2);
    v.erase(v.begin());
    for(const auto& i: v)
        cout << i << ' ';
    cout << '\n';
}
```

خروجی برنامه به صورت زیر است:

2 1 1 1

امکان پاس دادن ۲ آرگومان به `erase()` نیز وجود دارد. در این صورت بازهی بین آنها حذف میشود. برای مثال:

```
vector<int> v{1,1,1,1};
v.erase(v.begin(), v.begin() + 2); // {1,1}
```

هنگام اعلان متغیر از `vector`، با پاس دادن یک عدد به سازندهی آن با () میتوان سایز اولیهی `vector` را تعیین کرد. مقادیر عنصرها در این روش با سازندهی پیش فرض مقداردهی میشوند (در `int` با مقدار ۰، در `std::string` با رشتی خالی و ...). اگر ۲ آرگومان پاس داده شود به تعداد آرگومان اول سایز اولیه و با آرگومان دوم عنصرش مقداردهی میشود. برای مثال:

```
vector<int> v(4); // {0,0,0,0}
vector<int> v2(4, 1); // {1,1,1,1}
```

با اعلان `vector` از `vector` میتوان آرایهی دو بعدی را پیادهسازی کرد. برای مثال:

```
vector<vector<int>> a;
```

یک متغیر به اسم `a` اعلان میکند که هر ایندکس، خود یک شی `vector` از `int` است. این شی در عمل مانند یک آرایه‌ی دو بعدی نیز با `[[[]]]` به ایندکس مورد نظر دسترسی ایجاد میشود.

از C++11، دو عضو `emplace_back` و `emplace` به `vector` اضافه شده. این عضوها مانند `insert` و `push_back` برای افزودن عنصر به `vector` قرار گرفته‌اند با این تفاوت که آرگومانهای خود مستقیماً بر روی فضا کانتینر ایجاد میکند. آرگومانهایی که این توابع دریافت میکنند از نوع عناصر `vector` نیستند؛ بلکه آرگومانهای دریافتی را با قالب `variadic` به سازنده‌ی عنصر ارسال میکند. برای مثال، عملکرد `push_back` در دستورات زیر:

```
vector<MyClass> v;
v.push_back(MyClass{10}); // copy
```

به این صورت است که ابتدا شی موقت از `MyClass` ایجاد میشود و سپس به متغیر `v` انتقال مییابد. سپس در آخر نیز شی موقت از بین میرود. با استفاده از `emplace_back` به صورت زیر، مستقیماً شی `MyClass` در کانتینر ساخته میشود و از کپی یا انتقال اضافه جلوگیری خواهد شد:

```
vector<MyClass> v;
v.emplace_back(10);
```

با پاس دادن مقادیر غیر از `rvalue` به اعضای `emplace`، عملکرد مشابه‌ای مانند ایجاد خواهد شد (هر دو شی را کپی میکنند):

```
vector<MyClass> v;
MyClass m{10};
```

```
v.push_back(m); // copy
v.emplace_back(m); // copy
```

عضو `emplace_back` نیز مانند `emplace` است، با این تفاوت که به عنوان آرگومان اول، یک ایتریتور قبل از عنصری که میخواهد ساخته شود میگیرد.

عضو `emplace_back` در کانتینرهایی که عضو `push_back` دارند موجود است. به طور مشابه، اعضای `emplace` و `emplace_front` نیز در کانتینرهایی که را شامل میشوند، قرار گرفته‌است.

در `stl` برای ذخیره توالیه عناصر، غیر از `vector` از دو نوع دادهی `list` و `linked list` (لیست پیوندی) است (به ترتیب که پیاده سازی `forward_list` (سینگل-لینکد لیست) و `doubly-linked list` (دبلیو-لینکد لیست) میتوان استفاده کرد. این نوعها برای ذخیره مقادیری استفاده میشوند که بیشتر افزودن/کاستن آنها از مکانهای غیر از ابتدا و انتهای کانتینر صورت میگیرد. نوعهای `stack` و `queue` نیز به ترتیب برای استفاده از نوع داده با `LIFO` و `FIFO` قرار دارند. نوع دادهی آخر `deque` (دبلیو-اند-کوئی) است؛ افزودن/کاستن در این نوع فقط از اول و آخر کانتینر صورت میگیرد.^۱

std::array

تاکنون برای ذخیره توالیه عناصر با سایز ثابت از آرایه استفاده کردیم. یکی از ضعفهای آرایه‌ها، تبدیل خودکار آنها به اشارهگر و از بین رفتن اطلاعات سایز آنهاست. همچنین مدیریت فضای آرایه‌های دینامیک، در برنامه‌های بزرگ نیز دشوار است. از `C++11` به بعد، برای حل این ضعفها یک کانتینر به اسم `std::array` اضافه شده است. بعد از افزودن هدر `<array>` ایجاد متغیر از آن به صورت زیر میشود:

```
std::array<dataType, size> a;
```

مقداردهی اولیه‌ی آن با `{}` میتواند به ۲ صورت انجام شود:

^۱ برای ذخیره توالی مقادیر به صورت پیشفرض و در اکثر مواقع از `std::vector` استفاده میشود. نحوهی استفاده از کانتینرهای دیگر مشابه `std::vector` است و در رفرانس `C++` موجود است.

```
std::array<dataType, size> a{{1,2,3,4}};
std::array<dataType, size> a{1,2,3,4}; // since C++14
پیادهسازی قالب std::array خود شامل یک آرایه‌ی عادی است. مانند:
```

```
template<typename T, std::size_t n>
struct array {
    T a[n];
    // ...
}
```

در نتیجه با مقداردهی اولیه، آرایه‌ی آن با {} مقداردهی می‌شود. در دستور زیر: std::array<dataType, size> a{{1,2,3,4}}; {}‌های داخلی در برای آرایه‌ی عضو قرار می‌گیرند. از C++14 به بعد می‌توان فقط یک {} قرار داد. مثل:

```
std::array<dataType, size> a{1,2,3,4};
```

مثال ۶

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

void printSize(const array<int, 5> &m) {
    cout << "size: " << m.size(); // no decay
    cout << '\n';
}

int main() {
    array<int, 5> m{1, 2, 3, 4, 5};
    printSize(m);

    for(const auto& i : m)
        cout << i << ' ';
    cout << '\n';

    auto it = m.begin() + 2; // iterator
```

```

    cout << *it << '\n';
}

```

خروجی برنامه به صورت زیر است:

```

size: 5
1 2 3 4 5
3

```

به علت ثابت بودن سایز `std::array`، این متغیرها برخلاف `std::vector` شامل توابع `push_back`، `emplace_back` و ... نیستند.

`std::map`

`std::map` مانند `std::vector` یک نوع داده‌ی دیگر در `stl` است. استفاده از عناصر `map`، به صورت مجموعه‌ای از زوجهای `key-value` صورت می‌گیرد. یک `key` حداکثر یکبار در عناصر `map` میتواند قرار گیرد (منحصر به فرد است). `map` در بعضی از زبانهای برنامه نویسی (مثل `Python`) به اسم دیکشنری وجود دارد. به دلیل `key-value` گفته می‌شود (در مقابل بودن کانتینر وابسته (`associative`) کانتینر متوالی (`vector` (sequence) مثل `vector`)) مثل `map` اعلان متغیر از `std::map`، بعد از افزودن هدر `<map>` به صورت زیر می‌شود:

```
std::map<keyType, mappedType> m;
```

نوع داده‌ی `key` و `mappedType` مقدارهای `keyType` و `mappedType` است. `map` شده است.

برای مقداردهی اولیه‌ی `map` مانند `vector` از {} استفاده می‌شود. برای مثال:

```
map<int, string> m{{1, "s1"},  
                    {4, "s2"},  
                    {9, "s3"}};
```

سپس اضافه کردن عنصر با تابع عضو `insert()` به دو صورت زیر میتواند انجام شود:

```
m.insert( std::make_pair(key, data) );
```

`m.insert(std::pair<keyType, datatype> (key, data));`
 مثال (۷)

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

int main() {
    map<string, string> m;
    m.insert(make_pair("a","Linux"));
    m.insert(make_pair("b","Windows"));
    cout << m["a"] << '\n'; // or .at()
    cout << m["b"] << '\n';
}
```

خروجی برنامه به صورت زیر است:

```
Linux
Windows
```

برای ایجاد حلقه در `map` و نمایش عناصر با `for` مبتنی بر بازه، به صورت زیر عمل میشود:

```
for(auto& i : m) {
    cout << m.first << " : " << m.second << '\n';
}
```

پس از برابر شدن هر `i` (در `for`) با عناصرهای `map`، نوع `i` از نوع `std::pair` با عنصرهای `map` (در `for`) با عناصرهای `map` میشود؛ با اجرای اعضای `first` و `second` با `i` به `map` دسترسی پیدا میشود. در صورت استفاده از ایتریتور، برای اجرای این عضوها از اپراتور `->` (یا `*`) باید استفاده کرد. برای مثال:

```
map<string, string>::iterator i = m.begin();
cout << i->first << ":" << i->second; // a:linux
```

با اپراتور `[]` علاوه بر دریافت رفرنس به مقدار `map` شده، برای اضافه کردن عنصر به آن

نیز استفاده میشود. برای مثال:

```
m["d"] = "freeBSD";
```

تفاوت [] با `insert()` در این است که هنگام استفاده از []، اگر `key` وجود داشته باشد، مقدار آن با مقدار جدید جایگزین میشود و در غیر این صورت یک `key` جدید ایجاد میشود. ولی `insert()` همیشه یک `key` و `data` جدید میافزاید. لازم به ذکر است مقدار بازگشتی `insert()` نیز یک `pair` است که عضو `first`، یک `false` به عنصر اضافه شده یا از قبل موجود و `second` یک `true` یا `false` به معنی افزوده شدن عنصر و `false` به معنی وجود داشتن عنصر از قبل است. برای مثال:

```
auto ret = m.insert(make_pair(4, "freeBSD"));
if (!ret.second)
    cout << "key already exist with value of "
        << (ret.first)->second << '\n';
```

از فرم جدید `if` با `init` میتوان دستورات بالا را به صورت زیر نوشت (C++17):

```
if (auto ret = m.insert(make_pair(2, "freeBSD")); !ret.second)
    cout << "key already exist with value of "
        << (ret.first)->second << '\n';
```

با استفاده از این فرم اسم متغیر `ret` در کل ناحیه‌ای که `if` در آن نوشته شده قرار نمیگیرد و فقط در خود `if` قابل دسترس میشود.

برای جست و جو یک `key` در `map` از تابع عضو `find` استفاده میشود. این تابع اگر `key` مورد نظر را پیدا کرد، ایتریتور به عنصر آن برمیگرداند؛ در غیر این صورت `map::end` را بخواهد گرداند. برای مثال:

```
auto ret = m.find("a");
if(ret != m.end())
    cout<<"key 'a' found\n";
else
```

```
cout << "key 'a' not found!\n";
```

این دستورات را نیز با ساختار جدید `if` با `init` میتوان بازنویسی کرد.
برای حذف کردن عنصر از `map` از تابع عضو `erase` استفاده میشود. این تابع به عنوان آرگومان هم میتواند ایتریتور (یا بازهای از ایتریتورها) بگیرد و هم `.key`.

مثال:

```
map<string, int> products{{"bread", 103}, {"meat", 42}};
products.erase("meat"); // {{"bread", 103}}
products.erase(products.begin()); // {}
```

عنصرها در متغیر `map` توسط `key` و از کوچک به بزرگ مرتب میشوند. برای مثال:

```
map<int, string> m{{5, "string1"}, {2, "string2"}, {9, "string3"}};
for(auto &i:m) {
    cout << i.first << " : " << i.second;
    cout << '\n';
}
```

خروجی زیر را خواهد تولید میکند:

```
2 : string2
5 : string1
9 : string3
```

برای `key`هایی که از نوع `string` و `int` این ترتیب بر اساس حروف الفبا است.
روش مرتب سازی `map` از پارامتر سوم قالب آن مشخص میشود که به طور پیش فرض `std::less` قرار دارد (جزعی از کامپوننت الگوریتم است).
مرتب سازی دلخواه را در صورت نیاز میتوان با تعیین نوع سومین پارامتر قالب با نوع شی قابل فراخوانی، انجام داد. این شی این نوع باید ۲ پارامتر `key` را بگیرند و یک مقدار `bool` برگرداند. برای مثال دستورات زیر:

```
struct customSort {
    bool operator()(const int &l, const int &r) {
```

```

        return (l > r);
    }
};

map<int, string, customSort> m;
m = {{5, "string1"}, 
      {2, "string2"}, 
      {9, "string3"}};

for(auto& i : m){
    cout << i.first << ' ' << i.second << '\n';
}

```

را بر اساس بزرگی به کوچکی key های int نمایش میدهند. در یک map، key ها میتوانند شی از نوع داده‌ی ایجاد شده در برنامه باشند. در چنین map‌هایی باید فرآیند مرتبسازی (پیش فرض و یا تعیین شده) برای نوع داده تعریف شده باشد.

(۸) مثال

```

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

class MyClass {
    int x;
public:
    MyClass(int x): x{x} {}
    const int& getx() const {return x;}
};

struct customSort {
    bool operator()(const MyClass &l, const MyClass &r) {
        return l.getx() > r.getx();
    }
}

```

```

};

int main() {
    map<MyClass, int, customSort> m;

    m.insert(make_pair<MyClass, int>(MyClass{3}, 1));
    m.insert(make_pair<MyClass, int>(MyClass{14}, 2));
    m.insert(make_pair<MyClass, int>(MyClass{7}, 3));

    for(const auto& i: m) {
        cout << i.first.getx() << " : "
            << i.second << '\n';
    }
}

```

خروجی برنامه به صورت زیر است:

```

14 : 2
7 : 3
3 : 1

```

توضیح) در کلاس MyClass امکان سربارگذاری < برای مرتب سازی پیش فرض نیز وجود دارد.

یک کانتینر وابسته (associative) ای دیگر در stl، unordered_map است. key ها در unordered_map مرتبسازی نمیشوند؛ این نوع داده با جدول هش (hash table) پیاده میشود و معمولاً نیاز به ذخیره بیشتر حافظه نسبت به map دارند. از طرفی سرعت دسترسی به عناصر unordered_map سریعتر است. رابط این نوع کانتینر تقریبا مشابه map است.

نوعهای داده‌ی دیگری به اسم (multimap، unordered_multimap) در stl وجود دارند. این کانتینرها میتوانند چند عنصر را با keyهای یکسان ذخیره کنند.

عضو آخر `set`, `unordered_set`, `unordered_multiset` است. در این نوع کانتینرها، `key` نیز هست. به علت محدودتر بودن کاربرد کانتینرهای غیر `associative` در بین کانتینرها، به توضیح آنها نمیپردازیم. اعضای `map` از `map` و ... در این کانتینرها نیز مفاهیم مشابهای با `insert`, `iterator` دارند.

کار با فایل

پیش از این با ورودی و خروجی در کنسول به کمک `cin` و `cout` آشنا شدیم. در این قسمت با ورودی و خروجی در فایلها آشنا می‌شویم. برای این منظور در `<fstream>` کلاس‌های زیر:

`ofstream`: برای نوشتن بر روی فایل،

`ifstream`: برای خواندن از فایل،

`fstream`: هم برای خواندن و نوشتن فایل

موجوداند. پس از ساخت شی از یکی از کلاس‌های مذکور برای ورود و خروج، ابتدا باید فایل مورد نظر با تابع `open()` باز شود. اعلان تابع `open()` به صورت زیر است:

void open(const char *I, ios::mode);

اولین آرگومان، آدرس فایل و دومین آرگومان مُد باز کردن فایل است که می‌تواند شامل موارد زیر باشد:

flag	توضیح
<code>ios::in</code>	فایل را برای خواندن باز می‌کند. (ورود)
<code>ios::out</code>	فایل را برای نوشتن باز می‌کند. (خروج)
<code>ios::app</code>	خروجی، به انتهای فایل افزوده می‌شود.
<code>ios::ate</code>	فایل را باز می‌کند و برنامه را برای خواندن و نوشتن به انتها فایل می‌برد.
<code>ios::trunc</code>	اگر محتوایی در فایل وجود داشت، ابتدا آن را پاک می‌کند.
<code>ios::binary</code>	فایل را به صورت باینری باز می‌کند.

فرق `ios::app` با `ios::ate` این است که `ios::app` قبل از هر خواندن/نوشتن به انتهای فایل می‌رود؛ ولی `ios::ate` فقط باز کردن فایل به انتهای فایل می‌رود و

پس از باز شدن، میتوان به مکان‌های دیگر فایل نیز رجوع شود.
با اپراتور **|** می‌توان بیش از یک مد را مشخص کرد. برای مثال:

```
F0.open("file.bin", ios::out | ios::app |
ios::binary);
```

شی F0، فایل file.bin را با سه مد باز می‌کند.
اشیای ifstream و ofstream به طور پیش فرض مدهای زیر را دارند:

کلاس	پیش فرض mode
ofstream	ios::out ios::trunk
iostream	ios::in
fstream	ios::in ios::out

باز کردن فایل با سازنده نیز امکان‌پذیر است.

بعد از باز کردن فایل – مانند cout و cin – از اپراتور <> برای خروجی و <> برای ورودی استفاده می‌شود.

مثال ۱) برنامه‌ای که از شی ofstream استفاده می‌کند:

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

int main() {
    ofstream of{"data.txt"};
    of << "text1\n";
    of << "text2\n";
}
```

با کمپایل و اجرای دستورات، فایل data.txt در دایرکتوری برنامه ایجاد و سپس رشته‌های text1 و text2 در آن نوشته می‌شوند. مانند:



توضیح) با از بین رفتن شی `fstream`‌ها، فایل باز شده به طور خودکار بسته میشود. با این وجود، جهت بستن زودتر فایل قبل از بین رفتن شی، از تابع عضو `close()` میتوان استفاده کرد. با تابع `is_open()` نیز میتوان باز یا بسته بودن فایل توسط شی `false` را بررسی کرد. این تابع در صورت باز بودن `true` و در غیر این صورت `false` برمیگردد.

با پاس دادن نام فایل به `(open()`) یا سازنده، فایل در دایرکتوری برنامه ایجاد میشود. برای ایجاد فایل در دایرکتوری دیگر، اسم فایل را باید همراه با آدرسش پاس داد. مثلاً در سیستم عامل‌های شبیه یونیکس، با پاس دادن:

`/tmp/data.txt`

برنامه، فایل `data.txt` را در دایرکتوری `/tmp` ایجاد میکند. کاربران ویندوز برای آدرسی دهی دایرکتوری با `\`، باید دو بک اسلش \\\ قرار دهند. در این صورت با توجه به `escape character`، اولین بک اسلش نادیده و دومین بک اسلش در آدرس فایل قرار میگیرد. در ویندوز با / نیز میتوان آدرس فایل را تعیین کرد.

مثال ۲) اگر فایل `data.txt` را در دایرکتوری برنامه به صورت زیر داشته باشیم:

data.txt:

20 Richard 10

15	Linus	20
5	Ken	30

به طوری که داده‌ها به ترتیب ستون‌ها به صورت id، اسم و سن وارد شده‌اند، با دستورات زیر داده‌ها در خروجی نمایش می‌یابند:

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main() {
    int id, age;
    string name;
    ifstream ifo{"data.txt"};
    ifo >> id >> name >> age;
    cout << id << ' ' << name << ' ' << age << '\n';
    ifo >> id >> name >> age;
    cout << id << ' ' << name << ' ' << age << '\n';
    ifo >> id >> name >> age;
    cout << id << ' ' << name << ' ' << age << '\n';
}
```

برای جلوگیری از چند بار نوشته شدن دستورات خواندن، از while به صورت زیر می‌توان استفاده کرد:

```
while(ifo >> id >> name >> age) {
    cout << id << ' ' << name << ' ' << age << '\n';
}
```

اشیای ifstream شامل تابع عضو seekg() و ifstream شامل seekp() هستند که جهت تغییر مکان در فایل استفاده می‌شوند. اولین آرگومان، فاصله را به صورت بایت از آرگومان دوم ایجاد می‌کند. آرگومان دوم می‌تواند:

ios::beg از اول فایل،
ios::cur از مکان فعلی در فایل،
ios::end از آخر فایل باشد.

باشد. در صورت نوشته نشدن آرگومان دوم، `ios::beg` به طور پیش فرض قرار می‌گیرد.

با توابع `tell()` و `tellp()` می‌توان به ترتیب موقعیت خروجی و ورودی فعلی فایل را دریافت کرد.

مثال ۳) در این مثال یک فایل به اسم `a.txt` در دایرکتوری برنامه ایجاد و با توابع عضو `seekp()` و `tellp()` نیز به قسمتهای مختلف فایل رجوع می‌شود.

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

int main(){
    std::ofstream of{"a.txt"};
    of << 1 << '\n';
    of << 2 << '\n';
    of << 3 << '\n';
    auto pos = of.tellp();
    of.seekp(pos - 3);
    of << "string\n";
}
```

توضیح) پس از اجرای برنامه، محتوای فایل `a.txt` به صورت زیر می‌شود:

```
1
string
```

تابع (۲)

در این فصل به ناگفته‌های توابع میپردازیم.

constexpr

در گذشته مشاهده کردیم که با استفاده واژه‌ی **constexpr** می‌توان متغیرهایی را اعلان کرد که مقدارشان در حین کمپایل مشخص می‌شود. این نوع مقادیر، امتیازاتی نسبت به مقادیر معمولی دارند: می‌توانند در تعیین تعداد عناصر **std::array**، مقادیر شمارنده و ... استفاده شوند. اکنون با کاربرد **constexpr** در تابع آشنا می‌شویم.

تابع **constexpr** برخلاف متغیر **constexpr** همیشه مقدار مشخص در حین کمپایل برنمی‌گرداند. تابع **constexpr** فقط زمانی این نوع مقادیر را برمی‌گرداند که با چنین مقادیری فراخوانی شوند. در غیر این صورت مانند یک تابع عادی عمل خواهد کرد. برای مثال فرض کنید در برنامه نیاز به نوشتن تابعی شود تا مقداری برای تعیین سایز اشیای **std::array**، با انجام یکسری محاسبه برگرداند. چنین تابعی را می‌توان به صورت زیر نوشت^۱:

```
constexpr int f(int a, int b){  
    // int result = ...;  
    return result;  
}
```

و سپس می‌توان یک **std::array** به صورت زیر اعلان کرد:

```
std::array<int, f(2,2)> a;
```

تابعی که مقدار **constexpr** برمی‌گرداند محدود به دریافت و بازگشت انواع لیترال‌ها است. در C++ می‌توان تمام لیترال‌های موجود به جز **void** و انواع داده‌هایی که با **constexpr** ایجاد شده‌اند را به عنوان پارامتر توسط این تابع دریافت کرد. مثال (۱) در این مثال با تعریف کلاسی که با مقادیر **constexpr** مقداردهی و

^۱ در صورتی که با کمپایل دستورات با خطأ مواجه می‌شوید، کمپایلر شما از C++14 پشتیبانی نمی‌کند. در C++11 تابع **constexpr** فقط می‌توانند شامل یک دستور و آنهم دستور **return** باشد.

مقدارگیری میشود آشنا می‌شویم.

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

class Class{
public:
    constexpr Class(int aVal): a{aVal}
    {}
    constexpr int geta() const {return a;}
private:
    int a;
};

int main(){
    constexpr Class c{10};
    array<int, c.geta()> a; // 10
}
```

توضیح) در این مثال اعلان سازنده را به صورت `constexpr` نوشتیم زیرا مقادیری که با آنها شی مقداردهی اولیه می‌شود در حین کمپایل مشخصاند. به همین ترتیب توابع `getter` و `setter` میتوانند چنین مقادیری را برگردانند.

تابع `constexpr` یک سری محدودیت نیز دارد؛ در C++14 نمیتواند شامل موارد زیر باشد:

1. اعلان `asm`
2. دستور `goto`
3. دستوری که شامل `label` است به جز `case` و `.default`
4. بلوک `try`
5. تعریف متغیر از نوع غیر لیترال

¹ اعلان `asm` قابلیتی است که امکان استفاده از کد اسمبلی در برنامه‌ی C++ را ممکن می‌کند.

6. تعریف متغیر `thread static` یا `constexpr`

7. تعریف متغیر بدون مقداردهی اولیه

در صورت عدم وجود این موارد، تابع را به صورت `constexpr` تعریف کنید.
همچنین کمپایلر برای متغیرهایی که از نوع `constexpr` تعریف شوند، میتواند بهینهسازی نیز انجام دهد. پس به عنوان یک قاعده کلی متغیرها و توابع را به صورت `constexpr` اعلان کنید مگر اینکه با محدودیتهای مذکور مواجه شوید.

اشارهگر تابع

در C و C++ امکان اشاره به یک تابع وجود دارد. این نوع اشارهگرهای اشارهگر تابع نام دارد. اگر تابعی مانند زیر داشته باشیم:

```
int sum(int a, int b){  
    return a + b;  
}
```

میتوان به صورت زیر یک اشارهگر اعلان کرد و به سپس به تابع `sum` اشاره داد:

```
int (*sumPtr)(int, int);  
sumPtr = &sum;
```

برای دستیابی به آدرس تابع، `&` نیز میتواند قرار نگیرد:

```
int (*sumPtr)(int, int);  
sumPtr = sum;
```

سپس برای فراخوانی تابع با اشارهگر، به دو صورت میتوان زیر عمل کرد:

```
sumPtr(1,4);  
(*sumPtr)(1,4);
```

(۲) مثال

```
#include <iostream>  
using namespace std;  
  
int sum(int a, int b){
```

```

return a + b;
}

int main() {
    int (*sumPtr)(int, int) = &sum;
    cout << sumPtr(2,4) << '\n';
}

```

خروجی برنامه به صورت زیر است:

6

به این ترتیب فرمات اعلان اشارهگر تابع به صورت زیر است:

```
funcReturnType (*funcName)(parameters);
```

هنگام فراخوانی یک تابع به اشارهگر توجه کنید که در صورتی که تابع، پارامتر پیش فرض داشته باشد، این مقادیر مقداردهی نمیشوند. زیرا پارامترهای پیش فرض هنگام کمپایل مقداردهی میشوند؛ ولی فراخوانی یک تابع با اشارهگر، هنگام اجرا ارزیابی خواهد شد. در این صورت مقادیر پارامترها مستقیم هنگام فراخوانی با اشارهگر باید تعیین شوند.

برای مثال:

```

int func(int a, int b = 10) {
    // definition
}
funcPtr(5, 10);

```

در برنامه نویسی یکی از کاربردهای این اشارهگرهای، پاس دادن آنها به توابع دیگر است. زیرا توابعی که این نوع اشارهگرهای را دریافت میکنند میتوانند توابع مورد اشاره‌ی آنها را – در صورت یکسان نبودن ناحیه آنها نیز – اجرا کنند. این توابع، توابع callback نام دارند.

مثال ۳

```

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

```

```

void func(const string& s, void (*callback)(string
const&)){
    callback(s);
}

void print(const string& s){
    cout << "##" << s << '\n';
}

int main() {
    void (*fptr)(string const&) = &print;
    func("string", fptr);
}

```

خروجی برنامه به صورت زیر است:

****string**

در ادامه‌ی این فصل با **std::function** آشنا خواهیم شد. شی این قالب کلاس علاوه بر توابع، در ذخیره‌ی تمام اشیای قابل فراخوانی (مثل شی کلاسی که اپراتور () را سربارگذاری کرده باشد) قابل استفاده است. **std::function** جایگزین مناسب برای اشاره‌گر تابع در C++ مدرن است.

تابع **inline**

تابعی که به صورت **inline** تعریف می‌شود هنگام فراخوانی، بدن‌اش مستقیم در محل فراخوانی قرار می‌گیرد. برای مثال:

```

inline int add(int a,int b){
    return a + b;
}

```

برای تعریف تابع به صورت **inline**، این واژه، باید قبل از تعریف تابع قرار گیرد و اعلان آن مانند اعلان یک تابع عادی نوشته می‌شود. تابعی که در خود کلاس تعریف شده، به

طور خودکار از نوع `inline` تعریف میشود.

```
class MyClass {
public:
    void f1(){ // inline
    }
    void f2();
};

inline void MyClass::f2() // inline
{}
```

توجه شود `inline` قرار دادن تابع صرفا باعث قرارگیری بدنی آن در محل اعلان نمیشود. بلکه به کمپایلر پیشنهاد میکند که این تابع را به این صورت تعریف کند. کمپایلر با اعمال بهینهسازی، توابع غیر `inline` را نیز میتواند `inline` و یا توابع `inline` را به صورت عادی تعریف کند. در تئوری با قرارگرفتن بدنی تابع `inline` در مکان فراخوانی، سرعت اجرای برنامه بالا میرود ولی حجم برنامه نیز افزایش پیدا میکند. تاثیر `inline` علاوه بر احتمال افزایش سرعت اجرا، امکان تعریف یکسان تابع در چند مکان فراخوانی، سرعت اجرای برنامه بالا میرود ولی حجم برنامه نیز افزایش پیدا میکند. در عمل به این معنیست که میتوان `translation unit (TU)` تابع را در هدر تعریف کرد؛ در نتیجه با افزوده شدن این هدر در چند TU، کمپایلر در لینک ایجاد خطای نمیکند. در C++17، متغیرهای `inline` نیز افزوده شده‌اند. متغیری `inline` از نوع `static` و یا در ناحیه‌ی `namespace` قراردارد میتواند از نوع `inline` تعریف شود:

```
struct MyStruct
{
    inline static int n = 1;
};
```

به این صورت امکان تعریف متغیرها `inline` در هدر وجود دارد به طوری که با ایجاد چند تعریف آن (با چند بار افزوده شدن هدر) خطای لینکر ایجاد نمیشود. همچنین توابعی که از نوع `constexpr` تعریف شده‌اند، به طور خودکار `inline` نیز هستند.

حذف تابع با **delete**

اگر اعلان تابعی مانند زیر داشته باشیم:

```
void getInt(int a);
```

این تابع میتواند با آرگومان های غیر از **int** نیز فراخوانی شود. برای مثال فراخوانی های زیر صحیح نند:

```
getInt(true);
getInt('a');
getInt(1.5);
```

دلیل این عملکرد تبدیل خودکار مقادیر **double**، **bool**، **char** به **int** است که توسط کمپایلر صورت میگیرد. در برنامه اگر این تابع فقط به ازای پارامترهای **int** معنیدار باشد، میتوان اعلان تابع برای مقادیر غیر **int** را با **delete** حذف کرد. برای مثال:

```
void getInt(int a);
void getInt(bool) = delete;
void getInt(char) = delete;
void getInt(double) = delete;
```

توجه شود که با حذف تابع به ازای آرگومان **double**، تابع برای **float** نیز حذف میشود چون کمپایلر تبدیل **float** به **double** را به **int** ترجیح میدهد و با فراخوانی تابع با مقدار **float**، برنامه سعی به اجرای تابع **delete** شده میکند.

همانطور که گفته شد در C++ 5، تابع عضو در کلاسها به صورت خودکار و توسط کمپایلر ایجاد میشوند. در شی بعضی از کلاس ها مثل استریمها، عملکرد دقیق سازندهی کپی و یا مثلاً انتقال مشخص نیست. در این گونه کلاسها میتوان از **delete** برای حذف این توابع استفاده کرد. برای مثال:

```
class uncopyable{
public:
    uncopyable(const uncopyable&) =delete;
```

```
uncopyable& operator=(const uncopyable&) =delete;
};
```

decltype(auto) و decltype

اپراتور `decltype` نوع عبارت را برمیگرداند. برای مثال:

```
int i = 0;
decltype(i) j = 1; // j as int
```

نحوهی بازگردانی نوع در `decltype` به گونهای است که اگر عبارت، یک `identifier` یا عضو کلاس (یا `struct`) که با `.` یا `->` دسترسی ایجاد شده باشد، نوع متغیر را برمیگرداند. برای مثال:

```
int i = 0;
const int i2 = 0;
```

```
class MyClass {
public:
    int x;
};

decltype(i) // int
decltype(i2) // const int
decltype(MyClass::x) // int
```

در غیر این صورت اگر عبارت `&decltype(e)`، `e` از نوع `T` باشد، `T` باشد، `&&T` از نوع `xvalue` در غیر این صورت از نوع `lvalue` و از نوع `T` باشد، `T` باشد، `&T` از نوع `prvalue` است که `T` را برمیگرداند. برای مثال:

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

```
int x1 = 1, x2 = 2;
```

```
template<typename T>
class MyClass {
//...
```

```
public:
T& operator[](size_t index);

};

MyClass<int> m; // decltype(v) is vector<int>

decltype(f(x)) // int
decltype(m[0] == 10) // int&
decltype(x1 + x2); // int
```

در C++11 یکی از کاربردهای `decltype` در تعیین نوع بازگشتی قالب‌های توابعی بود که مقدار بازگشتی آنها وابسته به پارامترهای قالب بودند. برای مثال در تعریف زیر:

```
template <typename T, typename U>
auto add(T const &t, U const &u) -> decltype(t+u) {
    return t+u;
}
```

نوع بازگشتی قالب بر اساس نوع جمع پارامترهای `t` و `u` تعیین می‌شود. توجه شود اینگونه تعیین نوع (با `->` بعد از لیست پارامتر)، نوع بازگشتی `trailing` نام دارد و مختص `decltype` نیست. از این روش تعیین نوع در توابع عادی که از نوع `auto` تعیین شده‌اند نیز می‌توان استفاده کرد. برای مثال:

```
auto f() -> int {
    return 1;
}
```

از C++14 به بعد نوشتن نوع بازگشتی `trailing` اختیاری شده است. در نتیجه قالب تابع `add` را بدون `decltype` می‌توان به صورت زیر نوشت:

```
template <typename T, typename U>
auto add(T const &t, U const &u) {
    return t+u;
}
```

این کاربرد decltype البته در لامبدهایی که پارامتر آنها از نوع `auto` است مورد استفاده قرار میگیرد (بخش بعد).

هنگام استفاده از `auto` برای تعیین نوع در دستورات زیر:

```
const int& ci = 10;
auto ci2 = ci; //ci2 as int
```

نوع `ci2` از طریق نتیجه‌گیری نوع `auto` تعیین میشود در نتیجه نوع `ci2` بدون `const` و رفرنس تعیین خواهد شد. با استفاده از `decltype(auto)`، میتوان نتیجه‌گیری نوع در `auto` را از طریق قوانین `decltype` انجام داد. برای مثال:

```
const int& ci = 10;
decltype(auto) ci2 = ci; //ci2 as const int&
```

متغیر `ci3` از نوع `&const int` تعیین میشود. استفاده اصلی `decltype(auto)` در تعیین نوع بازگشتی توابع ارسالی یا توابعی است که توابع دیگر را فراخوانی میکنند. برای مثال:

```
int f1();
int& f2();
```

```
decltype(auto) use_f1() {return f1();}
decltype(auto) use_f2() {return f2();}
```

تابع `f1` را از نوع عادی و `use_f2` را از نوع رفرنس برمیگرداند. در تابع ارسالی:

```
template<class... Args>
decltype(auto) f(Args&&... args) {
    return f2(std::forward<Args>(args)...);
```

به این ترتیب میتوان آن را از نوع `(auto)`، بدون دانستن رفرنس یا رفرنس نبودن مقدار بازگشتی تعیین کرد.

عبارت‌های لامبدا^۱

لامبدها، توابعی بیاسماند که مستقیم در یک محل تعریف و استفاده – یا به طور معمولتر – به توابع دیگر پاس داده می‌شوند.
دستور زیر یک لامبда ایجاد می‌کند:

```
[] (int a, int b){return a + b;}(2,3);
```

[] نماد تابع لامبدا، (int a, int b) پارامترهای آن و

دستور بازگشت لامبda است. (۲,۳) آرگومان‌هایی است که به لامبda پاس داده می‌شود.

مثال ^۴) برنامه‌ای که از لامبda استفاده می‌کند:

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

int main() {
    cout << [](int a, int b){return a + b;}(2,1) << '\n';
}
```

خروجی برنامه به صورت زیر است:

3

فرمت تعریف تابع لامبda به صورت زیر است:

[capture list] (parameter list) **mutable** noexcept ->
ret {function body}

که: [] و لیست دریافت^۲ (عبارتی درون ۲ قلاب) می‌تواند یکی از ۳ نوع زیر باشد:

: [] متغیرهای محلی در لامبda دریافت نمی‌شوند.

: [&] متغیرهای محلی با رفرنس در بدنه‌ی لامبda دریافت می‌شوند.

: [=] متغیرهای محلی با مقدارشان دریافت می‌شوند.

¹ Lambda function

² capture list

`return ret` نوع بازگشته لامبدا است. اگر نوشته نشود برنامه نوع آن را از روی مقدار تعیین میکند.

اگر متغیر ذخیرهسازی آن از نوع اتوماتیک نباشد، میتواند توسط لامبدا بدون لیست دریافتی دریافت شود:

```
int x = 10;
int main() {
    [] () {cout << x << '\n';} ();
}
```

در لامبدا، مقدار متغیرهای گرفته با مقدار (بر خلاف رفرنس) به طور پیش فرض نمیتواند تغییر کند. برای مثال:

```
int x;
[&]() {x = 2;}(); // OK
[=]() {x = 3;}(); // error
```

برای تغییر مقادیر این متغیرها، از واژه‌ی `mutable` در تعریف لامبدا باید استفاده کرد:

```
int x;
[&]() {x = 2;}(); // OK
[=]() mutable {x = 3;}(); // ok
```

همچنین یکی از انواع لیست دریافتی میتواند برای متغیرها تعیین شود و برای متغیرهای دیگر استثنا قابل شد. برای مثال:

```
int x = 10, y = 10;
[&, y] () mutable {x++; y++;} ();
cout << x << '\n'; // 11
cout << y << '\n'; // 10
```

در C++17 در محل `constexpr`، میتوان واژه‌ی `mutable` قرار داد. قرارگیری این عبارت به کمپایلر اعلان میکند فراخوانی تابع، یک فراخوانی `constexpr` است. اگر این واژه قرار نگیرد و لامبدا تمام شرایط `constexpr` را داشته باشد نیز، لامبda از نوع `constexpr` تعریف میشود.

عبارت `noexcept` در صورت قرارگیری در تعریف لامبدا، به کمپایلر اعلان می‌شود که لامبدا ایجاد استثنای نمی‌کند (بررسی در فصل آینده). قرارگیری این عبارت اختیاری است. مثلاً^(۵) برنامه‌ای که شامل لامبادایی است که از لیست دریافتی جهت دسترسی به متغیرهای خارج از آن استفاده می‌کند:

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

int main() {
    int y = 0;
    [&](int x) { if (y > x){
        cout << "y > x";
    } else {
        cout << "y < x";
    }
} (2);
cout << '\n';
}
```

خروجی برنامه به صورت زیر است:

```
y < x
```

تابع `std::sort` یک تابع از هدیر `<algorithm>` است که برای مرتب سازی کانتینرها استفاده می‌شود. به تابع `std::sort`، در صورت پاس دادن ۲ آرگومان به طوری که اولین آرگومان، ایتریتور به اولین عنصر و دومین آرگومان، ایتریتور به آخرین عنصر آرایه باشد، این تابع، عناصر کانتینر را به طور صعودی مرتب می‌کند. برای مثال:

```
int a[] = {1, 5, -4, 14};
size_t s = sizeof(a) / sizeof(a[0]);
sort(a, a + s);

for(const auto& i: a) {
    cout << i << ' ';
} // -4 1 5 14
```

تابع `std::sort` میتواند به عنوان پارامتر سوم یک شی تابع برای مقایسه دریافت کند. این تابع باید به صورتی باشد که اگر اولین آرگومان این تابع کوچکتر از دومی باشد باید `true` برگرداند. برای تعریف این تابع مقایسه به جای تعریف یک تابع عادی و قرار دادن آن در کل برنامه، میتوان در محل فراخوانی `std::sort`، یک لامبدا تعریف کرد و به آن پاس داد.

مثال ۶) برنامه‌ای که کاربرد لامبدا را هنگام استفاده از تابع `std::sort` نشان میدهد.

```
#include <iostream>
#include <vector>
#include <cmath>
#include <algorithm>
using namespace std;

int main(){
    vector<int> v = {-4, 4, -2, 6, -8};
    sort(v.begin(), v.end(), [](int a, int b) {
        return abs(b) > abs(a);
    });
    for(const auto& i : v) {
        cout << i << ' ';
    }
    cout << '\n';
}
```

خروجی برنامه به صورت زیر است:

-2 -4 4 6 -8

توضیح) در این مثال از یک لامبدا برای مرتب کردن عناصر `std::vector` استفاده میشود به طوری که عناصر آن را بر اساس مقدار قدر مطلق آنها مرتب میکند. همانطور که در اول بخش گفته شد در حالت ساده نوع بازگشتی لامبدا توسط کمپایلر و به طور خودکار نتیجهگیری میشود. نتیجهگیری نوع در لامبدا، مشابه یک تابع عادی است که نوع بازگشتی آن از نوع `auto` تعیین شده. برای مثال:

```
[](int a, int b){ return a + b; } (3,4); // 7
```

نوع بازگشتی آن از نوع `int` تعیین میشود. با این حال در صورت نیاز میتوان نوع بازگشتی آن را با `-> T` بعد از پارامتر آن تعیین کرد (مثلاً زمانی که نوع `return`ها متفاوت است)، مثلاً:

```
[](int a) -> double {
    if(a==3){
        return 1; // int
    } else {
        return 1.1; // double
    }
} (4);
```

نوع داده‌ی پارامترهای لامبدا را می‌توانند از نوع `auto` نیز تعیین کرد. برای مثال ۵، تعریف لامبدا زیر صحیح است:

```
[](auto a, auto b){return a + b;} (2,1);
```

در نتیجه برای دریافت آرگومان با ارسال کامل میتوان از `decltype` به صورت زیر استفاده کرد:

```
[](auto&& x){
    f(std::forward<decltype(x)>(x));
} (expr);
```

به طور کلی در لیست دریافتی لامبدها، متغیر نیز میتوان اعلان کرد. مثلاً:

```
[x = 10] () {
    cout << x << '\n';
}();
```

به این نوع لیست دریافتی، لیست دریافتی عمومی¹ گفته میشود. از این نوع لیست دریافتی در انتقال اشیا به لامبدا میتوان استفاده کرد:

```
unique_ptr<int> u(new int(10));
[u{move(u)}] () {
```

```
    cout << *u << '\n';
}();
```

خروجی زیر را خواهد داشت:

10

از لیست دریافتی عمومی جهت قرار دادن نام جدید برای متغیرهای خارج از لامبدا نیز استفاده میشود:

```
int i = 4;
[newi = i] () {
    cout << newi << '\n';
}();
```

به طور کلی لامبدا زمانی کاربرد دارد که بتوان آن را ذخیره کرد. با ذخیره‌ی لامبدا، آن را میتوان به توابع دیگر پاس داد، در کانتینر ذخیره کرد و ... برای ذخیره‌ی لامبدا از `<functional>` موجود در هدر `std::function` استفاده میشود. اشیای این قالب کلاس برای ذخیره‌ی لامبدا، تابع عادی و به طور کلی هر شی قابل فراخوانی استفاده میشود.

مثال (۷) برنامهای که چند روش استفاده از `std::function` را نشان میدهد:

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

void print_func(int i) {
    cout << i << '\n';
}

struct PrintNum {
    void operator()(int i) const{
        cout << i << '\n';
    }
};
```

```
int main() {
    function<void()> f1 = []() { cout << "f1\n"; };
    f1();

    function<void(int)> f2 = print_func;
    f2(20);

    function<void(int)> f3 = PrintNum();
    f3(30);
}
```

خروجی برنامه به صورت زیر است:

```
f1
20
30
```

در صورتی که یک لامبدا بدون لیست دریافتی تعریف شود به جای `std::function` میتوان آن را با اشارهگر تابع ذخیره کرد. به دلیل کاربرد کم چنین اشارهگری از بررسی مثالی در این مورد پرهیز میکنیم.

هنگام ذخیرهی لامبدا مانند مقادیر، به جای نوشتتن نوع آن مثل `<()function<void>` از واژهی `auto` میتوان استفاده کرد. برای مثال:

```
auto add = [](int a, int b){ return a + b; };

void f(std::function<bool(T)> lf); // error
```

نیاز به ذکر است که از شی `std::function` در قالبها، برای دریافت لامبدا با انواع مختلف نمیتوان به صورت زیر عمل کرد:

```
template<typename T>
```

```
void f(std::function<bool(T)> lf); // error
```

دلیل محدودیت قالبها این است که کمپایلر برای یک لامبدا، یک شی از یک کلاس به اسم نوع گلوژر¹ - که به صورت خودکار ایجاد میشود - میسازد. همچنین نوع گلوژرها منحصر به فرد است به این معنی که کمپایلر برای دو لامبدا با ظاهر یکسان، دو نوع

¹ Closure type

کلوزر یکسان ایجاد نمی‌کند. برای مثال:

```
auto l1 = [](int x) {return 10;};
auto l2 = [](int x) {return 10;};
std::is_same< decltype(l1), decltype(l2)>::value; // false
```

سپس اشیای ساخته شده از انواع کلوزر به صورت خودکار به شی تبدیل می‌شوند. در قالب تابعها تبدیل خودکار در نظر گرفته نمی‌شود، به همین دلیل امکان دریافت لامبدا به ازای انواع مختلف پارامتر T وجود ندارد. در چنین مواردی نوع T باید به صورت مختص تعیین شود و یا پارامتر قالب جدایی برای دریافت لامبدا نوشته شود. اشیای موقت ساخته شده از نوع کلوزر در هنگام اجرای برنامه، کلوزر^۱ نام دارند.

(۸) مثال

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

template<typename T, typename Function>
vector<T> filter(vector<T> const& v, Function f) {
    vector<T> tmp;
    for(auto const &i: v) {
        if(f(i)) {
            tmp.push_back(i);
        }
    }
    return tmp;
}

template<typename T>
void print_vec(const vector<T>& v) {
    for(auto const&i : v) {
        cout << i << ' ';
    }
}
```

¹. Closure

```

    }
    cout << '\n';
}

int main() {
    vector<int> v = {1,2,3,4,5};
    vector<int> result1 = filter(v, [](int x) {return
(x > 2);});
    vector<int> result2 = filter(v, [](int x) {return
(x > 4);});

    print_vec(result1);
    print_vec(result2);
}

```

خروجی برنامه به صورت زیر است:

```
3 4 5
5
```

توضیح) این برنامه شامل قالب تابعی است که شی هرنوع `vector` ای را با یک لامبدا دریافت میکند و با توجه به لامبدادای بولین، شی فیلتر شده‌ی مناسب را از نوع `vector` بر میگردد.

از `std::function` برای ذخیره کردن اعضای یک کلاس (یا `struct`) هم میتوان استفاده کرد:

```

struct MyStruct {
    MyStruct(int num) : num(num) {}
    void print_add(int i) {
        cout << num+i << '\n';
    }
    int num;
};

MyStruct MS0{1};

```

```
function<void(MyStruct&, int)> f =
&MyStruct::print_add;
f(MS0, 5);

function<int(MyStruct&)> n = &MyStruct::num;
cout << "num: " << n(MS0) << '\n';
```

خروجی زیر را خواهد داشت:

```
6
n = 1
```

همانطور که از خروجی دیده میشود برای ذخیره‌ی یک عضو در `std::function` باید شی کلاس (یا `struct`) به عنوان پارامتر توسط متغیر دریافت شود.

اکنون به `std::bind` میپردازیم. `std::bind` یک تابع دریافت میکند و یک شی تابع را با مقادیر پیشفرض یا جابه‌جایی شده‌ی پارامترها برمیگرداند. (مثال ۹)

```
#include <iostream>
#include <functional>
using namespace std;
using namespace std::placeholders;

void f1(int p1, int p2, int p3) {
    cout << p1 << ' ' << p2 << ' ' << p3;
    cout << '\n';
}

int main() {
    auto f2 = bind(f1, _3, _2, _1);
    f2(1, 2, 3);
}
```

خروجی برنامه به صورت زیر است:

3 2 1

توضیح) اولین ورودی `std::bind`, تابع (یا هر شی قابل فراخوانی) است. آرگومانهای بعدی `std::bind`, نحوی قرارگیری و یا مقادیر تابعیست که برمیگردد. `_1`, `_2` و ... نیز اعضای `placeholders` در برنامه معرفی شدند. در این برنامه تابع بازگشته (از نوع `std::function`) تابع `f1` را با پارامترها به صورت بر عکس اجرا میکند.

مثالهای دیگر از `std::bind`:

```
auto f1 = [](int a, int b) {cout << a << ' ' << b;};
auto f2 = bind(f1, 10, _1);
f2(2,3); // 10 2
```

در C++14, کاربرد `std::bind` به اندازه‌ی C++11 نیست: چون هنگام تعریف لامبда در C++14 بر خلاف C++11, امکان انتقال متغیرها در لیست دریافتی وجود دارد. دستور C++11 زیر:

```
int v = 4;
auto f1 = std::bind(f, std::move(v));
```

در C++14 به صورت زیر و بدون `std::bind` نوشته میشود:

```
int v = 4;
auto f1 = [v{std::move(v)}]() { f(std::move(v)); };
```

همچنین در C++11 عبارتها نمیتوانند در لامبدا دریافت شوند و فقط مشخصه‌ها این قابلیت را دارند. در C++14 این محدودیت برداشته شده. دستور C++11 زیر:

```
auto f1 = std::bind(f, a + b);
```

در C++14 به صورت زیر میتواند نوشته شود:

```
auto f1 = [sum = a + b]() { f(sum); };
```

عبارت‌های لامبدا در C++, پایه‌ی برنامه نویسی تابعیاند. در این پارادایم برنامه نویسی، محاسبات به عنوان ارزیابی توابع و بدون تغییر داده‌های ثابت صورت میگیرد. در برنامه

نویسی تابعی همچنین توابع را مانند مقادیر میتوان به توابع دیگر پاس داد، از توابع بازگرداند و به عنوان مثالی ساده اگر یک شی `std::vector` متشکل از یکسری عناصر `int` در برنامه موجود باشد و نیاز به ایجاد یک شی جدید از `std::vector` با مقادیر به توان ۲ رسیدهی عناصر شی اول شود، در حالت عادی از ساختار `for` استفاده میکنیم. برای مثال:

```
vector<int> v = {1,2,3,4,5};
vector<int> result;
for(const auto& i : v){
    result.push_back(i * i);
}
```

با استفاده از لامبدا و `std::transform` دستورات بالا را میتوان به صورت زیر بازنویسی کرد (این عمل در برنامه نویسی تابعی نگاشت (`map` نام دارد):

```
vector<int> v = {1,2,3,4,5};
vector<int> result;
std::transform(v.begin(), v.end(),
              std::back_inserter(result),
              [] (const int i) { return i*i;});
```

قالب تابع `std::transform` ابتدا و انتهای یک بازه، ابتدای بازهی مقصد و یک شی تابع (مثل لامبدا) را میگیرد و تابع را ببروی تمام عناصر بازهی مبدأ اعمال میکند؛ سپس نتیجه‌ی حاصل شده در بازهی مقصد ذخیره میشود. قالب تابع `std::back_inserter` یک نوع خاصی از ایتریتور است که برای تغییر مقدار عناصر در `<algorithm>` استفاده میشود.

استثناء¹

اگر در برنامه یک تابع با پارامتر `int` موجود باشد به طوری که به ازای آرگومان ۰ عملکرد غلط دارد، در گذشته از `if` و بازگشت یک مقدار خطأ (مثال -۱) برای عدم اجرای تابع استفاده میکردیم. مانند:

```
int f(int a){  
    if(a == 0) {  
        cout << "wrong input\n";  
        return -1;  
    }  
    //...  
}
```

این نوع بررسی مقدار پارامتر چندین ضعف دارد؛ برای مثال، دستور `return -1;` که به منظور مشخص کردن ورودی غلط به تابع نوشته شده، در بعضی از توابع، امکان تمایز آن با مقدار بازگشتهی ۱- در حالت عادی وجود ندارد. برای مثال در تابع:

```
double division(double a, double b){  
    if(b == 0) {  
        cout << "can't divide by 0\n";  
        return -1;  
    }  
    return a/b;  
}
```

که ۲ عدد را گرفته و حاصل تقسیم اولی بر دومی را برمی‌گرداند، در صورتی که -۱ از تابع برگردد، معلوم نیست که پارامتر دوم ۰ بوده و یا حاصل تقسیم مقدار ۱ شده است. در C++ مدیریت خطاها را با استفاده از یک مکانیسم بهتری به اسم مدیریت استثنای میتوان انجام داد. در این ساختار، قسمتی از برنامه که وقوع خطأ را به برنامه گزارش میدهد از مقدار بازگشتهی تابع متمایز میشود. همچنین خطاهایی که در اجرای برنامه رخ میدهند لزوماً مربوط ورودیهای غلط نیست؛ خطای ناشی از برقراری ارتباط با دیتابیس،

¹. Exception

مشکل سطح دسترسی فایل هنگام باز کردن آنها و ... همگی از این نوع خطاهای هستند. ایجاد استثنای با دستور `throw` صورت میگیرد. بعد از این واژه، مقداری را که با ایجاد استثنای میخواهیم تولید شود قرار میدهیم. برای مثال:

```
throw -1;
```

همچنین یک دستور `throw` باید در بلوک `try` و `catch` اجرا شود. بلوک `try` برای بررسی وجود استثنای قرار میگیرد. این بلوک زمانی که ایجاد استثنای مشاهده کرد، آن را جهت مدیریت، به بلوک `catch` مناسب انتقال میدهد.

برای مثال:

```
try{
    throw 5;
} catch(int e) {
    // ...
}
```

بلوک `catch` باید مستقیماً بعد از `try` قرار گیرد. همچنین برخلاف `try` میتوان بیش از یک بلوک `catch` بعد از `try` قرار داد.

مثال ۱) در این برنامه، یکتابع جهت محاسبه حاصل تقسیم دو عدد تعریف شده. در صورتی که پارامتر دوم این تابع صفر باشد (صفر بودن مقسوم علیه)، یک استثنای از نوع رشته بر میگرداند.

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

double division(double a, double b){
    if(b == 0)
        throw "can't divide by 0\n";
    else
        return a/b;
}
```

```
int main() {
    try {
        cout << division(4, 5) << '\n';
        cout << division(4, 0) << '\n';
    } catch(const char *e) {
        cout << e << '\n';
    }
}
```

خروجی برنامه به صورت زیر است:

```
0.8
can't divide by 0
```

توضیح) در تابع `division` متناسب با نوع عبارت بعد از `throw` (رشته)، پارامتر `blok` `catch` را از نوع `const char *` تعیین کردیم تا در این `blok` قابل دریافت باشد.

در اعلان پارامتر `catch`، میتوان فقط نوع استثنای را نوشت. در این صورت `blok` `catch` فقط با ایجاد استثنای و بدون دریافت مقدار آن اجرا میشود. برای مثال:

```
try {
    throw 1;
} catch(int){
    cout << "int exception was caught\n";
}
```

بلوک `try` به ازای نوعهای مختلف `throw` های مختلف اجرا کند. برای مثال:

```
try{

} catch (int e) { // catchs ints

} catch (MyClass e){ // catchs MyClass objects

} // ...
```

همچنین استثنایی که توسط `throw` ایجاد میشود میتواند از هر نوعی که بتوان آن را کپی کرد و یا انتقال داد باشد (مثلاً شی `MyClass` در دستورات بالا). زمانی که یک استثنا `throw` شده در برنامه نهایتاً نتواند دریافت شود، برنامه با اجرای `std::terminate()` متوقف میشود. برای مثال:

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

int main() {
    throw 4;
    cout << "program is already terminated.\n";
}
```

در این دستورات، خط شامل `cout` اجرا نمیشود.

مثال (۲)

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

void f(){
    try {
        throw 1;
        cout << "this line won't be printed\n";
    } catch(int){
        cout << "exception caught from f\n";
    }
}

int main() {
    try {
        f();
    } catch(int) {
        cout << "exception caught from main\n";
    }
}
```

خروجی برنامه به صورت زیر است:

```
exception caught from f
```

توضیح) در این برنامه، استثنا توسط بلوک `catch` موجود در تابع `main` دریافت نمیشود: زیرا بلوک `catch` در `f` این کار را زودتر انجام خواهد داد. همانطور که در این مثال دیده میشود، چند بلوک `try-catch` میتوانند درون هم قرار گیرند و نحوه اجرای `catch` را نیز مشاهده کردیم: با ایجاد استثنا در بلوک `try`، برنامه با نادیده گرفتن ادامه‌ی دستورات، به اولین بلوک `catch`ی که بتواند آن را دریافت کند منتقل میشود. با اجرای بلوک `catch` در این فرآیند، متغیرهای اتوماتیکی که از ابتدای بلوک `try` ساخته شده‌اند بر خلاف ترتیب اشغال آنها در حافظه از بین میروند. این فرآیند، `stack unwinding` نام دارد. برای مثال:

```
try{
    int a, b;
    throw 1;
} catch(int e){
    //...
}
```

با ایجاد استثنا با دستور `throw 1`، برنامه به بلوک `catch` انتقال می‌یابد و به ترتیب متغیرهای `b` و `a` نیز از بین می‌روند. این نحوه عملکرد بلوکهای `try` و `catch` در بعضی از برنامه‌ها مجر به نشت حافظه (`memory leak`) می‌شود. برای مثال در تابع `main` زیر:

```
int main(){
    try {
        int *a{new int};
        throw 10;
        delete a; // never gets executed!
    } catch(int) {
        // ...
    }
}
```

}

دستور `throw` قبل از دستور `delete`; اجرا میشود در نتیجه شی مورد اشاره‌ی `a` از بین نخواهد رفت. این نوع نشت حافظه در برنامه‌هایی که در اول بلوک `try` منابع را ایجاد و در آخر `try` آنها را آزاد میکنند رخ خواهد داد. زیرا در بلوک‌های بزرگ (مثل اکثر توابع) احتمال ایجاد یک استثنا در بین این دستورات کم نیست.

در C++ برای جلوگیری از نشت حافظه با بوجود آمدن استثنا، از تکنیک^۱ RAI¹ استفاده میشود. در تکنیک RAI¹، منابعی که در یک شی میخواهند اشغال شوند در سازنده و آزاد کردن آنها در مخرب صورت میگیرد. در این حالت شی اتوماتیک این انواع داده‌ها، با ایجاد استثنا و از بین رفتنشان، منابعی که اشغال کرد هاند را نیز به طور خودکار (با فرآخوانی مخربشان) آزاد میکنند.

مثال (۳)

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

class MyClass{
    int *a;
public:
    MyClass(): a{new int} {
        cout << "created\n";
    }
    ~MyClass() {
        delete a;
        cout << "destroyed\n";
    }
};

int main() {
    try{
        MyClass m;
```

¹. Resource Acquisition Is Initialization

```

throw 1;
} catch(int) {
    cout << "exception caught\n";
}
}

```

خروجی برنامه به صورت زیر است:

```

created
destroyed
exception caught

```

توضیح) در این مثال با اجرای دستور `throw 1`; منبع تخصیص یافته در شی `m` به طور خودکار و با اجرای مخرب آزاد میشود. در این تکنیک، مخرب به طور مستقل از ایجاد استثنای اجرا خواهد شد.

برای دریافت استثنای `catch` از هر نوع داده، با قرار دادن الیپسیس^۱ (سه نقطه (...)) در پارامتر آن این کار میسر میشود. برای مثال:

```

try {
    // code
}
catch (int) { cout << "an int caught."; }
catch (const char*) { cout << "a string caught"; }
catch (...) { cout << "default exception"; }

```

در صورت وجود بلوک `catch(...)` در بین چند بلوک `catch`، باید این بلوک آخرین آنها نوشته شود. برای مثال، دستورات زیر غلط است:

```

try{
    throw 1;
} catch(...){ // error
}
} catch(int) {
}

```

^۱. Ellipsis

اگر بلوک `catch` نتواند خطا را به طور کامل مدیریت کند، میتواند آن را دوباره `throw` کند تا در سطح بالاتری از `try` مدیریت شود. کردن دوباره خطا توسط عبارت `throw;` بدون قرار دادن مقداری مقابله آن صورت می‌گیرد. برای مثال:

```
try {
    // ...
}

} catch(err){
    log("error msg");
    throw;
}
```

دستور `throw;` و یا در تابعی که در `catch` فراخوانی می‌شود می‌تواند نوشته شود. در بلوک `catch(...)` نیز معمولاً دستور `throw;` وجود دارد. بعضی از توابع در برنامه استثنای ایجاد نمی‌کنند. اینگونه توابع را با قرار دادن واژه‌ی `noexcept` بعد از پارامتر آنها می‌توان از نوع توابع `noexcept` تعیین کرد. برای مثال:

```
void f() noexcept;
```

توابع `noexcept` به کمپایلر این امکان را میدهد که خروجی بهینه تولید کنند. به علاوه، برنامه نویس نیز مطلع می‌شود آیا نیاز به مدیریت خطا با قرار دادن فراخوانی این توابع در `try` هست یا خیر. واژه‌ی `noexcept` هم در اعلان و هم در تعریف تابع باید نوشته شود. اگر تابع `noexcept` در حین اجرا استثنای ایجاد کند، برنامه با اجرای `(std::terminate)` متوقف می‌شود. توجه شود ایجاد استثنای در توابع `noexcept` با کمپایل برنامه چک نمی‌شود و فقط در اجرای آن‌ها ایجاد مشکل خواهد کرد. تعیین کننده‌ی `noexcept` با `()` نیز میتوان نوشته شود. در این فرم اگر عبارت درون `()` باشد تابع از نوع `noexcept` و در غیر این صورت از نوع تابع عادی تعیین می‌شود.

برای مثال:

```
template <typename T>
void f(T &x) noexcept(is_fundamental<T>::value);
```

نوشتن noexcept بدون پرانتز به معنی noexcept(true) است. عبارت درون پرانتز () نیز باید از نوع ثابت باشد.

علاوه بر تعیینکننده‌ی noexcept، یک اپراتور به اسم noexcept() نیز موجود است. اگر عبارت این اپراتور استثنا ایجاد نکند true و در غیر این صورت false برمیگردد. برای مثال:

```
void f(int a) noexcept(noexcept(g(a)));
```

اپراتور noexcept(g(a))، ایجاد استثنا در تابع g را به ازای ورودی a بررسی میکند؛ در صورت برقراری، مقدار true برمیگردد و با true بودن پرانتز مقابل تعیین کننده‌ی noexcept، تابع f را از نوع noexcept تعیین میکند. در غیر این صورت تابع f از نوع عادی تعیین خواهد شد.

به عنوان یک قاعده‌ی کلی، هر تابعی که در برنامه استثنا ایجاد نمیکند باید از نوع throw noexcept تعیین شود. توجه شود اگر بدنه‌ی تابعی مستقیم شامل دستور

نباشد، بازهم میتواند استثنا ایجاد کند. برای مثال تابع زیر:

```
void f(){
    vector<int> v(50);
}
```

دستور شامل اعلان vector میتواند منجر به تولید استثنا شود. در نتیجه این توابع باید از نوع noexcept تعیین شوند.

کلاس std::exception

با کلاس std::exception میتوان مدیریت خطاهای را با رابط ثابت انجام داد. برای مثال اگر اپراتور new در تخصیص فضای دچار خطا شود، استثنا را از نوع std::bad_alloc برمیگرداند. کلاس std::bad_alloc ۲۵ کلاس دیگر

مشتق از `std::exception` که در کتابخانه‌ی استاندارد برای ایجاد استثناء استفاده می‌شود. کلاس `std::exception` در هدر `<exception>` قرار دارد.

مثال ۴)

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

int main() {
    try {
        int *a{new int[1000000000000000]}; // خطای alloc
    } catch(exception &e){
        cout << e.what() << '\n';
    }
}
```

خروجی برنامه به صورت زیر است:

`std::bad_alloc`

توضیح) در بلوک `try` یک دستور تخصیص حافظه با `new` نوشته‌یم که باعث `throw` شدن خطای `std::bad_alloc` می‌شود. توجه شود که پارامترهای از نوع `std::exception` از نوع رفرنس اعلان می‌شوند (مانند `e` در این مثال) تا اشیای مشتق شده از آن نیز (`std::bad_alloc`) قابل دریافت باشند. در این مثال پس از دریافت شی `std::bad_alloc`، متن خطأ را با تابع عضو `what()` نمایش دادیم.

در برنامه به طور کلی توصیه می‌شود که در صورت نیاز به تعریف یک کلاس مجرزا برای کار با استثناهای آن را طوری تعریف کنید که مشتق از `std::exception` از `std::exception` یا از کلاس‌های دیگر `<exception>` باشند. در این صورت `catch` کردن خطأ از این نوع، با شی `std::exception` می‌تواند صورت گیرد.

مثال ۶) در این مثال، یک کلاس جهت کار با استثناهای تعریف و مشتق از `std::exception`

میکنیم. تابع `what()` این کلاس جهت بازگردانی یک پیام دلخواه نادیده گرفته شده (`override`) شده است:

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

class CustomException: public exception {
    virtual const char* what() const noexcept {
        return "Custom error message goes here.";
    }
};

int main() {
    try {
        throw CustomException();
    } catch(exception &e){
        cout << e.what() << '\n';
    }
}
```

خروجی برنامه به صورت زیر است:

Custom error message goes here.

توضیح) یک کلاس برای مدیریت استثنای در یک برنامه کاربردی، صرفا برای نمایش یک رشته دلخواه ایجاد نمی‌شود. برای این کار از استثنای `std::runtime_error` میتوان استفاده کرد:

```
throw std::runtime_error("custom msg");
```

این کلاس در سازنده‌ی خود یک رشته دلخواه می‌گیرد و مانند بقیه کلاس‌های استثنای `std::exception` مشتق شده است. هدف از تعریف یک کلاس جداگانه برای استثنایها، مدیریت خطاهایی است که متناسب با برنامه نیاز به عملکرد خاصی نیز دارند.

تابع `division` از مثال ۱ را میتوان با کلاس‌های استاندارد بازنویسی کرد:

```
double division(double a, double b){
    if(b != 0)
        return a/b;
    else
        throw std::invalid_argument{"can't divide by
0"};
}
```

جهت اطلاع از بقیه‌ی کلاس‌های استاندارد استثنای میتوانید به رفرانس C++ مراجعه کنید.

اینواریانت^۱

اگر یک کلاس ساده برای پیاده کردن آرایه با کلاس‌ها - مثل `std::array` - داشته باشیم (برای سادگی `array` از نوع قالب نیست):

```
class array {
    int s;
    int* a;
public:
    array(int s);
    int& operator[](int i);
    // ...
};

array::array(int size){
    if(size < 0) throw std::length_error{};
    a = new int[size];
    s = size;
}

int& array::operator[](int i){
    if(0 <= i && i < s)
```

^۱. Invariant

```

throw std::out_of_range{"Subscript operator"};
return a[i];
}

```

اپراتور [] در این کلاس، رفرنس به عنصر مورد نظر را با فرض اینکه **a** به آرایه‌ای از **int** ها با سایز **S** اشاره می‌کند برمی‌گرداند. توابع عضو دیگر این کلاس نیز با این فرض نوشته خواهند شد. به دستوراتی که با فرض درست بودن آنها در کلاس استفاده می‌شوند، اینواریانت گفته می‌شود. در این کلاس، اگر سایز تعیین شده منفی باشد و یا هنگام تخصیص حافظه با **new** مشکلی ایجاد شود، سازنده با ایجاد استثناء، شی ساخته خواهد شد. در نتیجه اپراتور [] که برای دسترسی به عناصرهای مختلف آرایه سربارگذاری شده، فقط صحیح بودن شماره‌ی عنصر مورد نظر را (بین ۰ و آخرین عنصر) بررسی می‌کند و سپس بدون نیاز به بررسی صحیح بودن آرایه (اینواریانت)، عنصر مورد نظر از آن را برمی‌گرداند. در یک کلاس به طور کلی، سازنده وظیفه ایجاد کردن اینواریانتها و توابع آن وظیفه حفظ کردن اینواریانتها بعد از خروج برنامه از آنها را دارد.

static_assert و **assert**

با ایجاد استثناء، بروز خطأ به برنامه گزارش می‌شود. از **static_assert** و **assert** می‌توان برای گزارش خطأ به برنامه نویس استفاده کرد. این خطاهای معمولاً از نوع خطاهاییست که از نظر منطقی غیر ممکناند و در صورت وجود، مشکلات اساسی در برنامه وجود دارند. **static_assert** برای بررسی شرط در حین اجرا و **assert** در حین کمپایل استفاده می‌شوند. همچنانیں شرایطی که در **assert** و **static_assert** بررسی می‌شوند با فرض این است که به طور پیش فرض صحیحاند. برای مثال:

```

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

void f(int *p){
    assert(p != nullptr);
}

```

```
// ...
}

int main(){
    int a{10};
    int *p1 = &a, *p2;
    f(p1);
    f(p2);
}
```

برنامه را کمپایل کنید تا خروجی مشابه زیر تولید شود:

```
6: void f(int*): Assertion `p != nullptr' failed.
```

توضیح) با فرض اینکه تعریف تابع `f` فقط برای اشارهگرهای غیر `nullptr` صحیح است، در این مثال با `assert` این شرط را در حین اجرا بررسی کردیم. یک ماکرو از `<cstdlib>` است؛ در نتیجه باید بدون `std::` استفاده شود. این ماکرو در صورت غلط بودن شرط، اسم خط، تابع و ... را نمایش می‌دهد و برنامه را نیز از اجرا متوقف می‌کند. `assert`ها به طور کلی برای دیباگ کردن برنامه استفاده می‌شوند و در هنگام کمپایل نهایی با تعریف ماکروی `NDEBUG` با دستور زیر:

```
#define NDEBUG
```

اجرای `assert`ها نیز متوقف شود. بدون تعریف مستقیم ماکروی `NDEBUG` نیز در کمپایلرهای مختلف می‌توان آن را در هنگام کمپایل تعریف شده قرار داد. مثلاً در `g++` با قرار دادن گزینه‌ی `-NDEBUG` مانند:

```
g++ main.cpp -o main -NDEBUG
```

ماکروی `NDEBUG` در برنامه تعریف می‌شود. همچنین به دلیل اینکه از `assert` فقط برای دیباگ کردن استفاده می‌شود، باید شامل عباراتی باشند که اشیای برنامه را تغییر می‌دهند و یا به طور کلی اثر جانبی ایجاد می‌کنند: زیرا خارج از دیباگ کردن اجرا نخواهند شد. مثل:

```
assert(x = f()); // bad
```

در assert با قرار دادن && در شرط میتوان در صورت غلط بودن، رشتهای را نیز در خروجی نمایش داد. مثلاً:

```
assert(a == b && "a and b should be equal.");
```

مثال ۵) در این مثال یک نوع داده ایجاد میشود که فقط پارامترهای قالب pod را میپذیرد.^۱ از static_assert برای بررسی یک شرط در حین کمپایل استفاده میشود.

```
#include <cassert>
#include <type_traits>
using namespace std;

template<typename T>
class MyClass {
    static_assert(std::is_pod<T>::value, "T should be
POD");
};

class A {
    int x1;
public:
    int x2;
};

int main(){
    MyClass<int> m1;
    MyClass<A> m2;
}
```

برنامه را کمپایل کنید تا خروجی مشابه زیر تولید شود:

```
error: static assertion failed: T should be POD
```

توضیح) همانطور که در خروجی دیده میشود با غلط بودن شرط، رشتهای در خروجی نمایش مییابد و کمپایل متوقف میشود. در C++17 قسمت پیام static_assert میباشد و کمپایل متوقف نمیشود. در اختیاری است.

¹ انواع pod مخفف (Plain Old Data) انواعی هستند که با انواع مورد استفاده در زبان C سازگارند.

از استثناهای به طور معمول برای بررسی قسمتهای خارجی برنامه (مثلاً توابع `public` و `assert`) و از کلاسها برای قسمتهای داخلی و جهت مطلع کردن برنامه نویس قرار میگیرند.

کار با حافظه (۲) (اشاره‌گرهای هوشمند)^۱

- اشاره‌گرهای عادی (raw pointers) چندین ضعف دارند. مهمترین آنها عبارت‌اند از:
۱. اعلان آنها مشخص نمی‌کند که اشاره‌گر به یک شی و یا به یک آرایه اشاره می‌کند. این ویژگی علاوه بر تفاوت در نحوه استفاده از آنها، در نحوه آزاد کردن آنها (با `delete` و یا `delete []`) نیز تاثیر می‌گذارد.
 ۲. اعلان آنها مشخص نمی‌کند کجای برنامه وظیفه‌ی آزاد کردن فضای آن را بر عهده دارد؛ به عبارت دیگر کدام قسمت از برنامه مالک آن فضا است.
 ۳. تضمینی در اجرای دستور `delete` (نوشته شده) نیست (برای مثال در یک تابع ممکن است قبل رسیدن به دستور `delete`، به دستور `return` برسد. البته چنین مشکلاتی از طرف برنامه نویس ایجاد می‌شود ولی در برنامه‌هایی که شامل تعداد زیادی اشاره‌گر است، این مشکلات نادر نیستند).
 ۴. در حالت عادی امکان تشخیص اشاره‌گر معلق (dangling pointer) وجود ندارد. این نوع اشاره‌گرهای زمانی ایجاد می‌شوند که شی مورد اشاره از بین رفته ولی اشاره‌گر هنوز به فضای آن اشاره می‌کند.
 ۵. با اعلان اشاره‌گر `raw` به عنوان متغیرهای عضو، به طور معمول پیاده‌سازی توابع عضو خاص (شامل سازنده‌ی `copy`، انتساب `copy`، ...) باید توسط برنامه نویس صورت گیرد و کمپایلر تولید نخواهد کرد.

برای حل این مشکلات در هدر `<memory>`، اشاره‌گرهایی تحت عنوان اشاره‌گرهای هوشمند اضافه شدند، که شامل:

unique_ptr .1
weak_ptr .2
shared_ptr .3

است. اشاره‌گرهای هوشمند با قالب کلاس نوشته شده‌اند و خود شامل اشاره‌گر عادی (`raw`)‌اند. اشاره‌گرهای هوشمند مدیریت فضاهای دینامیک ایجاد شده را توسط مدل

^۱. Smart pointers

RAII مدیریت میکنند: به این معنی که فرآیند `delete` به طور خودکار و با ازبین رفتن شی صورت میگیرد. اپراتورهای این اشیا طوری سربارگذاری شده که استفاده از آنها مشابه اشارهگرهای `raw` است (* و -> دیفرنس میکند و ...)

`std::unique_ptr`

شی `unique_ptr` مالکیت فضای مورد اشاره را به صورت انحصاری (`exclusive`) کنترل میکند. به این معنا که در هر زمان، فقط یک شی `unique_ptr` میتواند به آن فضا اشاره کند. زمانی که شی `unique_ptr` از بین میرود (از ناحیه خارج میشود)، فضای مورد اشاره آن به طور خودکار آزاد خواهد شد.

اعلان متغیر از `std::unique_ptr` به صورت انجام میگیرد:

```
#include <memory>
...
std::unique_ptr<dataType> p;
```

نوع دادهایست که `p` میخواهد اشاره کند.

(مثال ۱)

```
#include <iostream>
#include <memory>
using namespace std;
int main() {
    unique_ptr<int> p(new int);
    *p = 100;
    cout << *p << '\n';
}
```

خروجی برنامه به صورت زیر است:

100

توضیح) فضای ایجاد شده با `new`, به سازنده شی `p` پاس داده میشود. سپس اشارهگر `p` نیز به طور خودکار فضای ایجاد شده را `delete` میکند.

مثال ۲) مثالی که نحوه استفاده از شی `unique_ptr` را برای ذخیره آرایه نشان میدهد:

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

int main() {
    constexpr int size = 10;
    unique_ptr<int[]> a(new int[size]);

    for (size_t i = 0; i < 10; ++i) {
        a[i] = i;
    }

    for (size_t i = 0; i < size; ++i) {
        cout << i << ":" << a[i] << '\n';
    }
}
```

خروجی برنامه به صورت زیر است:

```
0: 0
1: 1
2: 2
3: 3
4: 4
```

توضیح) با وجود اینکه برای ذخیره مجموعه‌ای از عناصر، جایگزینهای مناسبتری مثل `C` وجود دارد، از این نوع آرایه برای `api` های زبان `C++` (یا قدیمی) که اشاره‌گر `raw heap` به فضای `std::vector` `std::array` برمیگردانند میتوان استفاده کرد. لازم به ذکر است شی `std::array` یا `std::vector` حجم بیشتری نسبت به آرایه‌ی ذخیره شده با `unique_ptr` اشغال میکنند (با تفاوت چند بایت). در نتیجه در مواقعي که منابع سیستم کم و تعداد اشیای مورد نیاز زیاد است میتواند جاگزین مناسبی

باشد.

با فراخوانی تابع عضو `reset()` بدون پارامتر، میتوان شی مورد اشاره‌ی `unique_ptr` را `delete` کرد. با فراخوانی `reset()` و پاس دادن یک اشاره‌گر، علاوه‌به `delete` شدن فضای شی `unique_ptr`، به فضای جدید پاس داده شده اشاره خواهد کرد. همچنین با فراخوانی تابع عضو `release()` شی `unique_ptr`، اشاره‌گر مورد ذخیره‌ی خود را برمی‌گرداند و به `nullptr` اشاره خواهد کرد.

مثال (۳)

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

int main() {
    unique_ptr<string> p1(new string("string"));
    unique_ptr<string> p2(p1.release());
    unique_ptr<string> p3;
    p3.reset(p2.release());
    cout << *p3 << '\n';
    if(!p1){
        cout << "p1's object is destructed\n";
    }
}
```

خروجی برنامه به صورت زیر است:

```
string
p1's object is destructed.
```

توضیح) عملکرد توابع عضو مذکور در این برنامه بررسی شدند. در شی `unique_ptr` اپراتور `bool` در صورت اشاره‌ی اشاره‌گر به فضای `true` و در غیر این صورت `false` را برمی‌گرداند. از این اپراتور در ساختار `if` استفاده شد. توجه شود در صورتی که اشاره‌گر بازگشتی از تابع `release()` در اشاره‌گر عادی

ذخیره شود، مسئولیت `delete` کردن فضای برگشته به عهده‌ی برنامه نویس خواهد بود و به طور خودکار انجام نمی‌شود. برای مثال:

```
auto p = p1.release();
delete p;
```

همچنین شی `unique_ptr` را فقط میتوان انتقال داد و کپی کردن آن امکان‌پذیر نیست زیرا در این حالت فضای شی چندبار `delete` خواهد شد. برای مثال:

```
std::unique_ptr<int> a(new int);
std::unique_ptr<int> b;
b = std::move(a);
```

شی `a` به `b` منتقل می‌شود و `a` برابر با `nullptr` قرار خواهد گرفت. با تابع عضو `release` نیز فرآیند مشابهی انجام می‌شود. برای مثال:

```
std::unique_ptr<int> a(new int);
std::unique_ptr<int> b;
b.reset(a.release());
```

در حین ساخت شی از `unique_ptr` میتوان با پاس دادن تابع (یا شی تابع و لامبدا)، فرآیند از بین رفتن شی را به طور دلخواه تغییر داد. برای مثال در دستورات زیر:

```
auto customDeleter = [](Class *po)
{
    makeLog(po);
    delete po;
}
unique_ptr<Class, decltype(customDeleter)> p(new Class);
```

از `delete` شدن فضای `p`، با تابع فرضی `makeLog` لاغ گرفته می‌شود. همانطور که در این دستورات دیده می‌شود، نوع `deleter` غیر پیش فرض باید به عنوان آرگومان دوم قالب `std::unique_ptr` نوشته شود. تمام `delete` های غیر پیشفرض، فضای مورد اشاره را از طریق اشاره‌گر دریافت می‌کنند و فرآیند مناسب را پیش از

کردن انجام میدهند. این فرآیند میتواند شامل `log` گرفتن، بستن ارتباط شبکه، بستن فایل و ... باشد.

`()std::make_unique`

به جای ساخت شی با استفاده از `new` مانند:

```
unique_ptr<int> p(new int);
```

میتوان با تابع `std::make_unique()` شی `unique_ptr` ایجاد کرد:

```
unique_ptr<int> p(make_unique<int>());
```

هنگام استفاده از `make_unique` را `unique_ptr` به دلیل اینکه این تابع، شی `unique_ptr` را برمیگرداند، اعلان و مقداردهی اولیه‌ی شی `unique_ptr` را با نوع `auto` میتوان انجام داد. برای نمونه:

```
std::unique_ptr<Class> p1(new Widget);
auto p2(std::make_unique<Class>());
```

در دستور دوم `auto` جایگزین شده است.

یک محدودیت دارد: امکان مشخص کردن تابع `deleter` وجود ندارد؛ ولی با `new` این کار امکان پذیر است. برای مثال:

```
auto CustomDeleter = []( MyClass *m ) { ... };
unique_ptr< MyClass, decltype(CustomDeleter)> p(new MyClass, CustomDeleter)
```

مثال ۴) برنامه‌ای که ذخیره‌ی اشارهگر `unique_ptr` در یک `vector` را نشان میدهد.

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

int main() {
```

```

vector<unique_ptr<int>> v;
unique_ptr<int> u = make_unique<int>(5);
v.push_back(move(u));
cout << *v[0] << '\n';
}

```

خروجی برنامه به صورت زیر است:

5

یکی از کاربردهای معمول `unique_ptr` در نوع بازگشتی توابع است. برای مثال در تابعی میتواند استفاده شود که آرگومان (از نوع `int`, `string` و ...) دریافت میکند و با توجه به `id` و یا متن رشته، اشارهگر کلاس پایه به شی مشتق مناسب را برمیگرداند. این نوع توابع، متدهای فکتوری (`factory methods`) نام دارند. همچنین این توابع غیر از آرگومان ذکر شده (برای تشخصی شی مشتق مناسب)، آرگومانهای مورد نیاز برای ساخت شی مشتق را نیز دریافت میکنند.

در شی `unique_ptr` چند تابع عضو، غیر از مواردی که بررسی کردیم موجود است؛ تابع عضو `get`، اشارهگر شی مورد اشاره را برمیگرداند. این تابع عضو در پاس دادن اشارهگر `raw` به یک تابع (مثلاً تابع C) مورد استفاده قرار میگیرد. تابع عضو `swap` شی را برمیگرداند. تابع عضو `get_deleter`, `deleter` و `unique_ptr` را با هم جابجا میکند.

std::shared_ptr

فضای مورد اشاره‌ی شی `shared_ptr` برخلاف `unique_ptr` میتواند همزمان مورد اشاره‌ی چند `shared_ptr` قرار گیرد. فرمت تعریف شی `shared_ptr` به صورت زیر است:

```

#include <memory>
...
std::unique_ptr<dataType> p;

```

مثال ۵) برنامه‌ای که شامل یک اشارهگر `shared_ptr` است:

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

int main() {
    shared_ptr<int> p(new int);
    *p = 44;
    cout << *p << '\n';
}
```

خروجی برنامه به صورت زیر است:

44

عملکرد `unique_ptr` مانند `shared_ptr` در شی `release()` و `reset()` است.

برنامه فضای مورد اشاره‌ی `shared_ptr` را زمانی `delete` میکند که آخرین شی `shared_ptr` که به آن فضا اشاره میکند از بین رود. برنامه این ارزیابی را با استفاده از یک شمارنده‌ی شی `shared_ptr` انجام میدهد. این شمارنده، عدد رفرنس^۱ نام دارد و با اجرا شدن سازنده‌ی شی `shared_ptr` یکی به مقدارش افزوده و با اجرای مخرب یکی از مقدارش کم میشود. اپراتور کپی `assignment` میتواند مقدار عدد رفرنس را هم زیاد و هم کم کند. به این معنی که وقتی عبارت زیر را مینویسیم:

`p1 = p2;`

و `p1` و `p2` به دو شی جداگانه اشاره کنند، از عدد رفرنس شی `p1` کاهش و به افزوده میشود.

زمانی که برنامه با کم کردن عدد رفرنس آن را به صفر برساند، منبع مورد اشاره‌ی `shared_ptr` را `delete` میکند. امکان انتقال اشیای `shared_ptr` (با سازنده یا اپراتور `=`) مانند `unique_ptr` وجود دارد؛ در این نوع فرآیند، مقدار عدد رفرنس کم یا

¹. Reference count

زیاد نمیشود (زیرا با فرآیند انتقال، شی به فضای جدید و شی قبلی به `nullptr` اشاره خواهد داد). با فراخوانی تابع `use_count()` به مقدار عدد رفرنس میتوان دست یافت.

مثال ۶)

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

int main(){
    shared_ptr<int> p1(new int);
    shared_ptr<int> p2 = p1;
    cout << p1.use_count() << '\n';
    cout << p2.use_count() << '\n';
}
```

خروجی برنامه به صورت زیر است:

```
2
2
```

حجم شی `shared_ptr` به دلیل شامل عدد رفرنس است، دوبرابر حجم یک اشارهگر `raw` فضا اشغال میکند (زیرا ۲ اشارهگر باید ذخیره کند). همچنین مقدار عدد رفرنس میتواند همزمان در چند `thread` کم و زیاد شود؛ به همین دلیل این فرآیند به صورت `atomic`^۱ انجام میشود. فرآیندهای `atomic` کنترل از غیر `atomic` است. برای شی `shared_ptr`، مانند `unique_ptr` میتوان `deleter` غیر پیش فرض تعیین کرد. با این تفاوت که در `shared_ptr`، `deleter` جز نوع اشارهگر نیست. برای مثال:

```
auto customDeleter = [](Class *po){
    makeLog(po);
    delete po;
```

^۱ در این کتاب به فرآیندهای `atomic` و به طور کلی پردازش موازی پرداخته نمیشود.

```
}
```

```
unique_ptr<Class, decltype(customDeleter)> up(new
Class, customDeleter);
shared_ptr<Class> sp(new Class, customDeleter);
```

امتیاز طراحی `shared_ptr` نسبت به `unique_ptr` در یکسان بودن نوع `deleter`ها (با `shared_ptr` امیتوان اشیای مختلف `shared_ptr` از یک نوع را (با `deleter`های متفاوت) در یک کانتینر ذخیره کرد و یا به یک توابع پاس داد. در شی `shared_ptr` برخلاف `unique_ptr`، سایز `shared_ptr` تاثیری در حجم شی `deleter` ندارد. همچنین تابع عضو `make_shared` مشابه `make_unique` ولی برای شی `shared_ptr` است. ویژگی و محدودیتهای این تابع نیز مانند `make_unique` است. برای مثال:

```
auto p = make_shared<int>(5);
```

همانطور که پیشتر گفته شد، شی `shared_ptr` شامل دو اشارهگر است، یکی از آنها به فضای مورد اشاره و دیگری به عدد رفرنس اشاره می‌کند. اشارهگر دوم علاوه بر عدد رفرنس به قسمت بزرگتری به اسم `block` کنترل^۱ اشاره میکند. `block` کنترل غیر از عدد رفرنس شامل کپی `deleter` (اگر قرار گرفته باشد)، تخصیص دهنده و شمارندهی `weak` (ادامه‌ی فصل) نیز میشود. `block` کنترل با `make_shared` و یا پاس دادن اشارهگر `raw` به سازنده‌ی شی `shared_ptr` ایجاد میشود (و یا از طریق مقداردهی توسط شی `shared_ptr.unique_ptr`. زیرا شی `shared_ptr` با مقداردهی توسط شی `unique_ptr` سازگار است، ولی بر عکس آن صحیح نیست). به این ترتیب برای جلوگی از ایجاد چند `block` کنترل و ایجاد چند عدد رفرنس جداگانه، از یک اشارهگر `raw` نباید در ساخت چند شی مختلف `shared_ptr` استفاده شود. برای مثال:

```
int *i = new int;
```

¹. Control block

```
shared_ptr<int> s1(i);
shared_ptr<int> s2(i); // undefined behaviour
```

زیرا در این صورت با وجود چند عدد رفرنس مجزا، با صفر شدن هر کدام برنامه سعی در کردن فضای اشاره‌گر (در اینجا `i`) میکند. در چند بار `delete` شدن یک فضا، ایجاد `undefined behaviour` میشود. در صورت نیاز به پاس دادن اشاره‌گر `raw` (مثلًاً تعیین `make_shared` که با `deleter` امکان پذیر نیست)، خروجی `new` را مستقیم به سازنده شی `shared_ptr` ارسال کنید. برای مثال:

```
shared_ptr<int> s1(new int, customDeleter); // ok
```

در یکی از موارد دیگر که پاس دادن اشاره‌گر `raw` باعث بروز مشکل میشود، در ساخت شی `shared_ptr` در تابع عضوی است که میخواهد به شی `this` اشاره کند. برای مثال در دستورات زیر^۱:

```
#include <memory>
#include <iostream>
using namespace std;
class MyClass {
public:
    shared_ptr<MyClass> f()
    {
        return shared_ptr<MyClass>(this); // bad
    }
};

int main(){
    shared_ptr<MyClass> p1(new MyClass);
    shared_ptr<MyClass> p2 = p1->f(); // error
}
```

به دلیل اینکه در تابع عضو `f` برای ساخت شی `shared_ptr` از اشاره‌گر `this` استفاده میشود، شیای را این تابع برمیگردد عدد رفرنس متفاوتی نسبت به شی اصلی

^۱ این مبحث جز قسمتهای پیشرفته و با کاربرد خاص C++ است؛ لذا در صورت تمایل از آن بگذرید.

دارد؛ زیرا با پاس دادن اشارهگر `shared_ptr`، یک بلوک کنترل جدید ایجاد میشود. برای حمل این مشکل، باید کلاس را از `std::enable_shared_from_this` مشتق کرد؛ سپس در محل مورد نیاز تابع `shared_ptr` فراخوانی شود. مثال قبل به صورت زیر بازنویسی میشود:

```
#include <memory>
#include <iostream>
using namespace std;
class MyClass: public enable_shared_from_this<MyClass> {
public:
    shared_ptr<MyClass> f() {
        return shared_from_this(); // OK
    }
};

int main(){
    shared_ptr<MyClass> p1(new MyClass);
    shared_ptr<MyClass> p2 = p1->f();
}
```

توجه شود که کلاس باید از قالب کلاس `enable_shared_from_this` و با نوشتن نام آن در `<>` مقابلش مشتق شود.^۱ تابع `shared_from_this()`، از نظر ساختاری، `shared_ptr` جدیدی میسازد که بلوک کنترل آن با شی `unique_ptr` فعلی یکی است. طراحی این تابع به گونه‌ای است که روی بلوک کنترل شی `unique_ptr` فعلی تکیه میکند.

`std::weak_ptr`

اشارهگر `shared_ptr`، تکمیل کننده‌ی `std::weak_ptr` است. این اشارهگر

^۱ این نوع مشتق کردن CRTP (Curiously Recurring Template Pattern) نام دارد.

میتواند به فضای `shared_ptr` موجود اشاره کند ولی مقدار عدد رفنس شی `shared_ptr` را تغییر نمیدهد.
مثال (۷)

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

int main(){
    shared_ptr<int> p = make_shared<int>();
    weak_ptr<int> w(p);
    cout << p.use_count() << '\n';
}
```

خروجی برنامه به صورت زیر است:

1

اگر فضای `shared_ptr` که `weak_ptr` به آن اشاره میکند از بین رود، شی `weak_ptr` اصطلاحاً معلق میشود (به این معنا که به فضایی اشاره میکند که از بین رفته است). برای بررسی معلق بودن شی `weak_ptr` از تابع عضو `expired()` میتوان استفاده کرد. پس از بررسی با این تابع عضو، برای استفاده از فضای `weak_ptr` میتوان تابع عضو `lock()` را فراخوانی کرد (زیرا خود شی `weak_ptr` اپراتورهای دیفرنس کردن ندارد). تابع عضو `lock()`، شی `shared_ptr` را به فضا - و یا `nullptr` را در صورت معلق بودن - برمیگرداند.
مثال (۸)

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

int main(){
    shared_ptr<int> p = make_shared<int>(10);
```

```

weak_ptr<int> w(p);
if(!w.expired()){
    shared_ptr<int> p2 = w.lock();
    cout << *p2 << '\n';
}
}

```

خروجی برنامه به صورت زیر است:

10

برای ساخت `shared_ptr` از `weak_ptr` غیر از `lock()` با پاس دادن شی `weak_ptr` به سازنده‌ی شی `shared_ptr` نیز میتوان استفاده کرد. اگر در این روش شی `weak_ptr` معلق باشد، استثنای `std::bad_weak_ptr` برخواهد گشت. برای مثال:

```

shared_ptr<int> p = make_shared<int>(10);
weak_ptr<int> w(p);
p = nullptr;
shared_ptr<int> p2(w); // std::bad_weak_ptr thrown.

```

یکی از استفاده‌های `weak_ptr` در موقعی است که دو کلاس داشته باشیم و اشیای هر کدام شامل اشارهگری است که به شی دیگری اشاره میکند.

برای مثال:

```

#include <memory>
#include <iostream>
using namespace std;
class B;
class A {
public:
    shared_ptr<B> b;
    ~A() { cout << "~A()"; }
};

class B {

```

```

public:
    shared_ptr<A> a;
    ~B() { cout << "~B()"; }
};

int main() {
    auto ap = std::make_shared<A>();
    auto bp = std::make_shared<B>();
    ap->b = bp;
    bp->a = ap;
}

```

با کمپایل و اجرای این دستورات، کمپایلر پیغامهای مخرب کلاسها را اجرا نمیکند؛ که به این معنیست که اشارهگرهای `a` و `b` از بین نمیروند. در این مثال، شی دو کلاس، شامل `shared_ptr` هایی است که به فضاهم اشاره میکنند. هنگام از بین رفتن این `shared_ptr` ها، از بین رفتن هر کدام وابسته به از بین رفتن دیگریست و اصطلاحاً رفرنس سیکلی^۱ ایجاد میشود. با تغییر یکی از اعضای `shared_ptr` به `weak_ptr` این وابستگی از بین میرود (در طراحی، شی کلاسی را که مالک (owner) شی دیگر است را `shared_ptr` نگه دارید و دیگری را به `weak_ptr` تغییر دهید). مثال صحیح آن چنین است:

```

#include <memory>
#include <iostream>
using namespace std;
class B;
class A {
public:
    shared_ptr<B> b;
    int x{10};
    ~A() { cout << "~A()" << '\n'; }
};

class B {

```

^۱. Cyclic reference

```

public:
    weak_ptr<A> a;
    void printx(){
        auto t = a.lock();
        if(t)
            cout << "x = " << t->x << '\n';
    }
    ~B() { cout << "~B()\n"; }
};

int main() {
    auto ap = std::make_shared<A>();
    auto bp = std::make_shared<B>();
    ap->b = bp;
    bp->a = ap;
    bp->printx();
}

```

خروجی برنامه به صورت زیر است:

```
x = 10
~A()
~B()
```

تولید اعداد تصادفی^۱

قبل از C++11، تولید اعداد تصادفی (راندم) در C++ با کتابخانه‌ی زبان C صورت می‌گرفت. از C++11، هدیری تحت عنوان <random> افزوده شده، که چندین مزیت نسبت به کتابخانه‌ی C دارد که در این فصل به آنها می‌پردازیم. به دلیل اینکه تولید اعداد تصادفی با کتابخانه‌ی C به طور گسترده در کدهای قدیمی C++ استفاده شده، ابتدا به بررسی آن می‌پردازیم.

برای تولید اعداد تصادفی از طریق کتابخانه‌ی C ابتدا هدیر <cstdlib> باید به برنامه افزوده شود. این هدیر تابع std::rand() را در دسترس قرار میدهد. با فراخوانی RAND_MAX() یک عدد تصادفی بین بازه ۰ و RAND_MAX برمی‌گردد. std::rand یک ماکرو در cstdlib است که مقدار آن در کمپایلرهای مختلف متفاوت است.

(۱) مثال

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

int main() {
    for(int i = 0; i < 5; ++i) {
        cout << rand() << '\n';
    }
}
```

برنامه را کمپایل و اجرا کنید تا خروجی مشابه زیر تولید شود:

```
1968078301
287724083
410622274
558519326
460165363
```

^۱ بطور دقیق‌تر اعداد شبیه تصادفی.

توضیح) با کمپایل و اجرای برنامه، پنج عدد تصادفی تولید میشود. با اجرای دوبارهی برنامه، خروجی های مشابهی تولید خواهد شد. برای ایجاد خروجیهای متفاوت در هر اجرا، باید تابع `std::rand()` با یک آرگومان `unsigned` فراخوانی شود (به این کار سید (seed) کردن گفته میشود). با پاس دادن مقداری ثابت در سید کردن مقادیری متفاوت از گذشته تولید میشود ولی این مقادیر باز هم با ثابت بودن مقدار پاس داده شده به `std::rand()`، یکسان خواهند بود. برای حل این مشکل به جای پاس دادن مقادیر ثابت، از خروجی تابع (`<ctime>` از `std::time(nullptr)`) در هر اجرا برای سید کردن `rand` استفاده میشود. تابع `std::time(nullptr)` در هر اجرا مقادیر متفاوتی را برمی‌گرداند که به سبب تولید اعداد تصادفی متفاوت در هر اجرا میشود.²

مثال ۲) بازنویسی مثال ۱ به طوری که در هر اجرا مقادیر جدیدی تولید میشود:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
int main() {
    srand(time(nullptr));
    for(int i = 0; i < 5; ++i) {
        cout << rand() << '\n';
    }
}
```

برنامه را کمپایل و اجرا کنید تا خروجی مشابه زیر تولید شود:

```
1796606603
117391392
1902428578
855479711
1181716800
```

² با فراخوانی تابع `std::time(nullptr)`، زمان گذشته از ۱۹۹۷ ۰۰:۰۰:۰۰ یا زانویه به ثانیه برمیگردد.

توضیح) توجه شود که تولید اعداد تصادفی یکسان همیشه نا مطلوب نیست. مثلاً در مقاله های علمی اگر نیاز به مشخص کردن اعداد تصادفی باشد، با مشخص کردن مقدار ثابت استفاده شده جهت سید کردن، برنامه، روی هر سیستمی که اجرا شود اعداد تصادفی یکسان تولید خواهد کرد (با شرط یکسان بودن کمپایلر).

به طور کلی در برنامه نویسی برای تولید اعداد تصادفی کمتر از n ، به طور گسترده از اپراتور $\%$ بر روی خروجی تابعی که اعداد تصادفی تولید میکند (`std::rand()`) در $C/C++$ استفاده میشود. برای مثال دستور زیر:

```
std::rand() % 6;
```

مقادیر 0، 1، 2، ... 5 را برمیگرداند (چون باقی مانده‌ی تقسیم هر عدد طبیعی بر 6، یک عدد بدون اعشار بین بازه‌ی بسته‌ی 0 تا 5 است). با استفاده از روش بالا به ازای هر n ، مقادیر بین 0 تا $n-1$ با احتمال کاملاً مساوی برنمیگردد. برابر نبودن احتمال بازگشت مقادیر را برای $n = 6$ اکنون بررسی می‌کنیم. فرض شود مقدار `RAND_MAX` برابر 32767 است (که در اکثر کمپایلرهای نیز همین مقدار است)؛ در این حالت کل مقادیر قابل بازگشت توسط `std::rand()` به صورت زیر میباشد:

0	1	2	3	4	5
6	7	8	.	.	.
.
36766	36767				

حال با اعمال $\%$ ، اعداد زیر برمیگردد:

0	1	2	3	4	5
0	1	2	.	.	.
.
0	1				

همانطور که در جدول دیده میشود احتمال بازگشت اعداد 0 و 1 مقداری بیشتر از بقیه اعداد است. فقط زمانی این احتمال برابر خواهد بود که باقی مانده‌ی تقسیم

RAND_MAX بر n ، برابر ۱ - شود (همهی خانه‌های جدول بالا پر شوند). غیر از مشکل برابر نبودن این احتمال، تابع `std::rand()` خود نیز در اکثر پیاده‌سازیها بهترین الگوریتم موجود نیست. اکنون به بررسی قابلیتهای C++11 به بعد برای تولید اعداد تصادفی میپردازیم که مشکلات مذکور را برطرف کرده و قابلیتهای بیشتری نیز اضافه کرده است.

عناصر موجود برای تولید اعداد تصادفی در C++11 به دو عنصر جداگانه تقسیم شده: ۱. انجین (موتور): نقش انجین، تولید بیت‌های تصادفی است؛ به گونه‌ای که احتمال تولید هر ۰، در حالت اپتیمال، برابر با هر ۱ است.

۲. توزیع: نقش توزیع، شکل بخشیدن به بیت‌های تصادفی تولید شده توسط انجین است. توزیع میتواند از بیت‌های ایجاد شده اعداد تصادفی طبیعی، اعشاری، بولین، اعداد تصادفی با وزنهای مختلف و ... ایجاد کند.

مثال ۳) در این برنامه یک تابع به نام `roll_a_die()` برای شبیه‌سازی تاس تعریف میشود:

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

int roll_a_die() {
    static std::mt19937 e;
    static uniform_int_distribution<int> d(1, 6);
    return d(e);
}

int main() {
    for(int i=0; i < 5; ++i) {
        cout << roll_a_die() << '\n';
    }
}
```

برنامه را کمپایل و اجرا کنید تا خروجی مشابه زیر تولید شود:

```
5 1 6 6 1
```

توضیح) در تابع `roll_a_die`، یک انجین از `std::mt19937` به صورت اعلان میشود (تا در هر فراخوانی تابع، یک متغیر جدید ایجاد نشود). انجین `std::mt19937` پیاده سازی الگوریتم ۳۲ بیت مرسن توویستر (Mersenne Twister) است که در طیف زیادی از سیستمها استفاده میشود. در `<random>`، انجین های دیگری غیر از `std::mt19937` وجود دارند که در ادامه بررسی میکنیم؛ ولی در اکثر برنامهها، استفاده از این انجین مناسبترین گزینه است. در ادامه، یک متغیر از توزیع `std::uniform_int_distribution` اعلان میشود و بازه مورد نظر نیز (بین ۱ و ۶) هنگام ساخت آن به سازندهاش ارسال میشود. در مثال ۳، انجین `e` مانند `std::rand()` زمانی که سید نشده، اعداد یکسانی در هر اجرا برمیگرداند. سید کردن `std::mt19937` با پاس دادن مقدار عددی (مثبت) به سازندهاش میتواند صورت گیرد. برای مثال:

```
std::std::mt19937 e{11};
```

به طوری که مانند سید کردن با `std::srand()`، اگر یک مقدار ثابت جهت سید کردن به سازندهاش ارسال شود، اعدادی تولید خواهد شد که به ازای این مقدار سید در هر اجرا ثابتاند. برای دریافت اعداد مختلف در هر اجرا میتوان از زمان (مثل `C++11 time(nullptr)`) استفاده کرد؛ ولی در `std::random_device` گزینه‌ی مناسبتری است. برای مثال:

```
std::random_device rdev{};  
std::mt19937 e{rdev()};
```

شی `std::random_engine` یک عدد تصادفی غیر قطعی تولید که جهت سید کردن انجین مورد استفاده قرار میگیرد. توجه شود که از خود شی `std::random_e ngine` مستقیماً برای تولید اعداد تصادفی نباید استفاده کرد (به دلیل سرعت پایین و

نیاز به اشغال فضای زیاد) بلکه برای سید کردن انجین فقط مناسب است. در نتیجه تعریف تابع roll_a_die() از مثال ۳ به گونه‌ای که در هر اجرا اعداد تصادفی جدیدی تولید می‌کند به صورت زیر می‌شود:

```
int roll_a_die() {
    static std::random_device rdev{};
    static std::mt19937 e{rdev()};
    static uniform_int_distribution<int> d(1, 6);
    return d(e);
}
```

مثال ۴ برنامه‌ای که یک سری عدد تصادفی را با وزن‌های مختلف تولید می‌کند. برای تعیین کردن وزن اعداد از توزیع std::discrete_distribution استفاده می‌شود. این توزیع یک عدد صحیح بین بازه‌ی [۰, n) با وزنهای تعیین شده تولید می‌کند.

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

int main() {
    vector<int> rand_tracker(4);
    random_device rd;
    mt19937 gen(rd());
    discrete_distribution<> d({4, 1, 1, 4});

    for(int n = 0; n<10000; ++n) {
        rand_tracker[d(gen)]++;
    }
    for(int i = 0; i < 4; ++i) {
        std::cout << i << " generated " <<
        rand_tracker[i] << " times\n";
    }
}
```

برنامه را کمپایل و اجرا کنید تا خروجی مشابه زیر تولید شود:

```
0 generated 3954 times
1 generated 964 times
2 generated 984 times
3 generated 4098 times
```

توضیح) در ساخت متغیر `std::discrete_distribution`, وزن ها از نوع `std::initializer_list` به سازنده ارسال میشوند. شاید احتمال تولید هر عدد تصادفی را برابر نسبت وزن $\frac{1}{4}$ به جمع کل وزن ها قرار میدهد. در مثال بالا، احتمال تولید عدد ۰، برابر با $\frac{1}{4} + \frac{1}{4} = \frac{1}{2}$ و یا ۰.۵ است. همچنین در برنامه، از یک `vector` به اسم `rand_tracker` برای شمارش و نمایش اعداد تولید شده استفاده میشود. همانطور که از خروجی قابل مشاهده است، برای اعداد ۰ و ۳ که وزن بیشتری قرار دادیم تعداد بیشتری نسبت به ۱ و ۲ تولید میشود.

در زمان نوشتن این کتاب انجین های زیر در کتابخانه استاندارد تعریف شدهاند:

الگوریتم	انجین (در <code>std</code>)
Linear congruential	<code>linear_congruential_engine</code>
Mersenne twister	<code>mersenne_twister_engine</code>
Subtract with carry (lagged Fibonacci)	<code>subtract_with_carry_engine</code>

هر انجین از نظر سرعت، فضای مورد نیاز و بازده متفاوت است. برای مثال الگوریتم `Mersenne twister` سرعت نسبتاً کم و نیاز به اشغال فضای زیاد دارد ولی طولانیترین توالی غیر تکراری را تولید میکند.

اعلان انجین `std::mersenne_twister_engine` به صورت زیر است:

```
template<
    class UIntType,
    size_t w, size_t n, size_t m, size_t r,
```

```
UIntType a, size_t u, UIntType d, size_t s,
UIntType b, size_t t,
UIntType c, size_t l, UIntType f
> class mersenne_twister_engine;
```

البته غیر از استفاده های خاص، از انجینهای از پیش تعریف شده استفاده میشود. بین انجین های از پیش تعریف شده، mt19937 و mt19937_64 که الایس های mersenne_twister_engine مناسبترین انجینها هستند. mt19937 از انجین mersenne_twister_engine به صورت زیر تعریف شده (صرفاً جهت آشنایی):

```
std::mersenne_twister_engine<std::uint_fast32_t, 32,
624, 397, 31,
0x9908b0df, 11,
0xffffffff, 7,
0x9d2c5680, 15,
0xefc60000, 18, 1812433253>
```

انجین از پیش تعریف شده دیگر، std::default_random_engine است که نوع آن بر اساس کمپایلرهای مختلف، متفاوت است. استاندارد C++، این انجین را بر اساس سرعت، حجم و ... و یا ترکیبی از آنها، بهترین انجین برای استفاده های معمولی و غیر حرفهای تعریف کرده است. پس به طور کلی در اکثر برنامهها از mt19937 و یا default_random_engine استفاده میشود.

همچنین یک سری انجین های دیگر موجود است که بر اساس انجین های تعریف شده کار می کنند. این نوع انجین ها، اداپتور انجین¹ نام دارند. عبارت اند از:

اداپتور انجین (در std)	توضیح
discard_block_engine	این اداپتور، مقداری از داده های تولید شده توسط انجین پایه‌ی خود را حذف میکند.

¹. Engine adaptors

independent_bits_engine	اداپتور، خروجی انجین پایه‌ی خود را در بلوکهایی با بیت‌های مشخص شده قرار میدهد.
shuffle_order_engine	اداپتور، خروجی انجین پایه‌ی خود را با ترتیب کنندیگر تولید می‌کند.

اعلان اداپتور‌ها، جهت استفاده از آن‌ها به صورت زیر است:

```
template < class Engine, size_t P, size_t R
> class discard_block_engine;

template <class Engine, std::size_t W, class UIntType
> class independent_bits_engine;

template <class Engine, std::size_t K> class
shuffle_order_engine;
```

مثالی از نحوه استفاده از آداپتورها:

```
// using namespace std;
mt19937 eng;
discard_block_engine< mt19937, 11, 5>
discard_eng(eng);
shuffle_order_engine< discard_block_engine< mt19937,
11, 5> , 50> shuffle_eng(discard_eng);
```

در کتابخانه‌ی استاندارد، ۲۰ توزیع تعریف شده (در صورت نیاز به لیست کامل، به رفرنس C++ مراجعه کنید). پر استفاده ترین آنها (در std) عبارت اند از:

Distribution	توضیح
uniform_int_distribution	مقادیر صحیح با توزیع مساوی در بازه تولید می‌کند.
uniform_real_distribution	مقادیر حقیقی با توزیع مساوی در بازه

	تولید میکند.
bernoulli_distribution	مقادیر بولین بر اساس توزیع برنولی تولید میکند.
normal_distribution	مقادیر حقیقی روی توزیع نرمال (گاووس) تولید میکند.
discrete_distribution	مقادیر طبیعی بر اساس احتمال تولید میکند.

چند نکته‌ی قابل ذکر در مورد انجینهای:

1. طوری از انجینهای استفاده کنید که با کمترین تعداد ممکن ساخته شوند. به عنوان مثال اگر در توابع کلاسها استفاده می‌شوند، انجین را به صورت متغیر عضو تعریف کنید.
2. در صورتی که در یک `thread` مورد استفاده قرار می‌گیرند، در هر `thread` شی جدآگانهای از آنها اعلان کنید.
3. در کدهای قدیمی که از `std::rand()`، `std::srand()` و ... جهت تولید اعداد تصادفی استفاده می‌شود، میتوان با تعریف ماکرو و تابع زیر از قابلیتهايی `C++11` بدون بازنویسی برنامه استفاده کرد:

```
#define RAND_MAX (engine().max() - engine().min())
void srand( unsigned s = 1u ){ engine().seed(s);}
int rand() { return engine()() - engine().min(); }
```

ضمیمه‌ها

ضمیمه (۱) کمپایلر برای C++

کاربران ویندوز، از آخرین نسخه‌ی Visual Studio که از C++14 پشتیبانی میکند میتوانند استفاده کنند. از میان کمپایلرهای متن باز، از معروف ترین و به روزترین آن‌ها کمپایلر GCC است. این کمپایلر برای سیستم عامل‌های شبه یونیکس نوشته شده است. معادل این کمپایلر در ویندوز تحت بسته‌ی MinGW موجود است که از وبسایت www.mingw.org قابل دریافت میباشد.

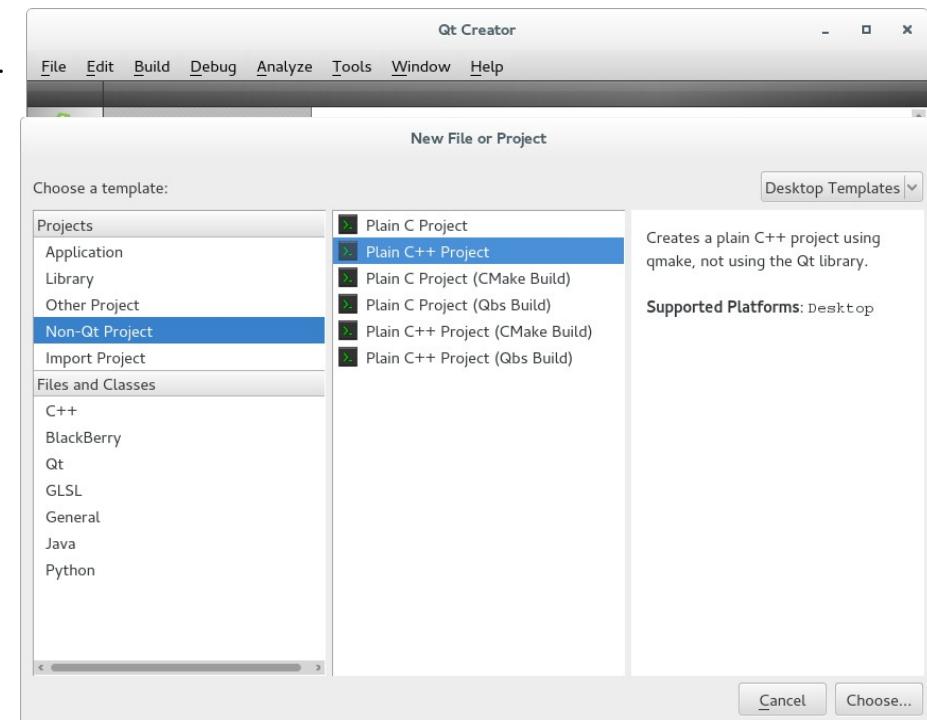
کاربران لینوکس و مک از آخرین نسخه‌ی GCC می‌توانند استفاده کنند که از آخرین استانداردها نیز پشتیبانی می‌کند. کمپایلر GCC شامل مجموعه‌ای از کمپایلرها برای زبانهای مختلف است که از g++, مختص کمپایل کدهای C++ قرار دارد. با استفاده از g++ هنگام کمپایل برنامه، به طور خودکار کتابخانه‌ها و پایانه‌ی مختص C++ استفاده می‌شود.

آشنایی با IDE

در این قسمت طریقه‌ی نصب qt creator نوشته شده؛ این IDE پس از آشنایی با مقدمات C++ و جهت کار با کتابخانه‌ی Qt - که امکانات زیادی در زمینه‌ی شبکه، واسطه گرافیکی (gui) و ... را فراهم می‌کند - را آسانتر می‌کند.

پس از نصب کمپایلر (مثلًا GCC برای لینوکس و مک و MinGW برای ویندوز)، برای نصب qt creator فایل نصبی Qt که شامل qt creator نیز می‌باشد را برای سیستم خود از وبسایت www.qt.io دریافت کنید. پس از نصب، مراحل زیر را برای ایجاد پروژه طی کنید:

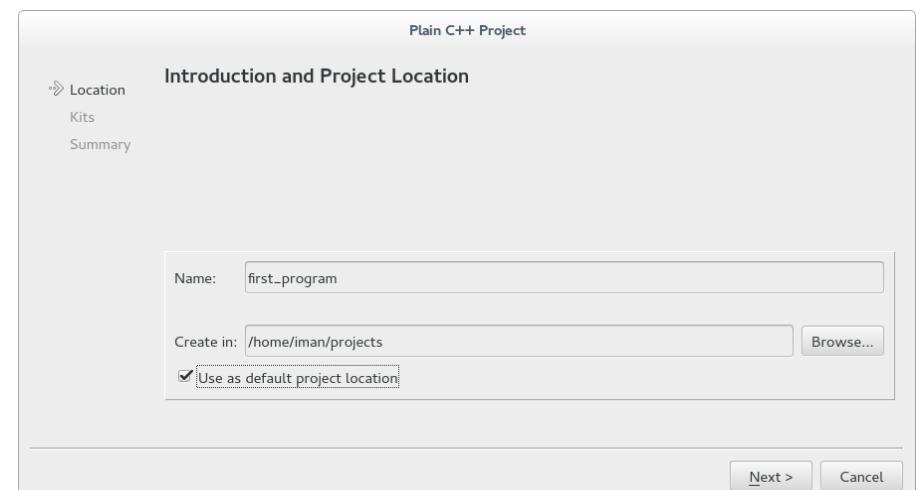
۱. ابتدا qt creator را اجرا کنید تا صفحه‌ای مشابه باز شود:



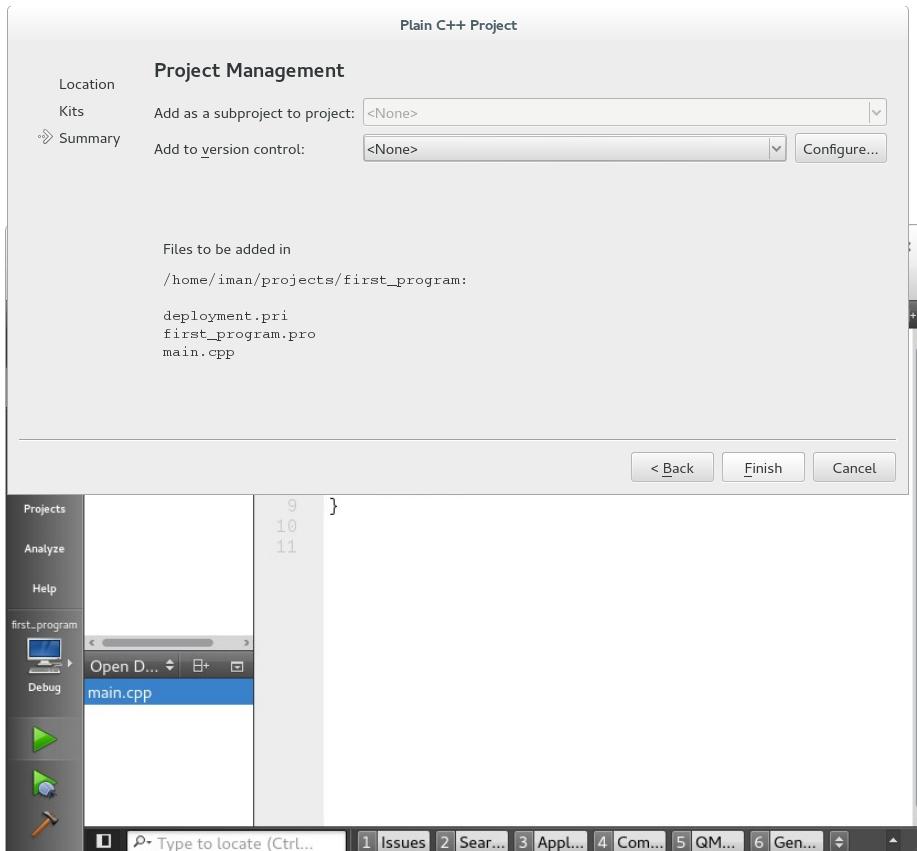
از منوی File گزینهی New file or Project را انتخاب کنید:

۳. از گزینه Non-Qt Project را از سمت راست انتخاب کنید سپس روی Choose... کلیک کنید:

۴. در مقابل Name: نام پروژه و مقابل Create in: آدرس پروژه را با ... انتخاب کنید و روی Next > بزنید:

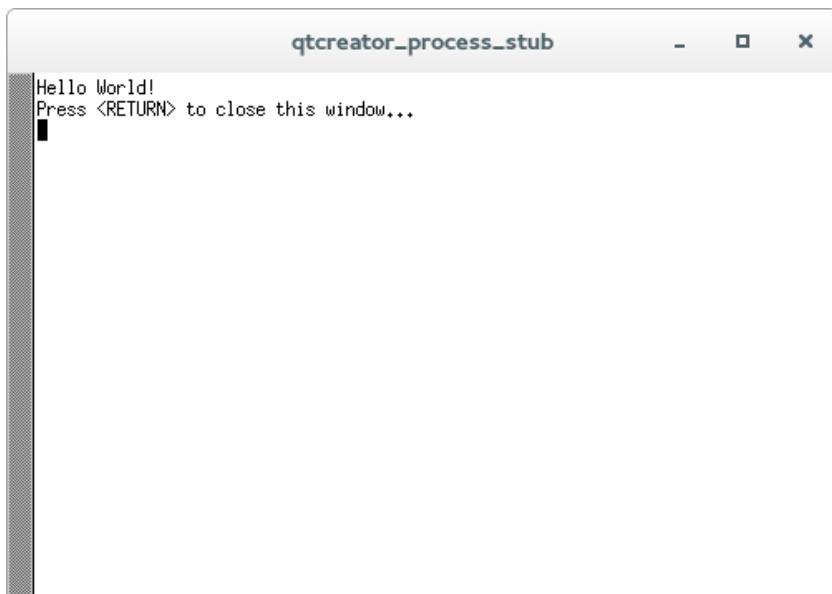


کمپایلر مورد نظر را انتخاب کنید و روی **Next** کلیک کنید.
 ۶. این صفحه مربوط به تنظیمات مربوط به برنامه‌های کنترل نسخه مانند git است. در صورت عدم نیاز به راهاندازی، بدون تغییر روی **Finish** کلیک کنید:



صفحه‌ی باز شده محیط برنامه نویسی ماست. در سمت چپ، فایل‌های مربوط به برنامه قرار دارند. از آنجا پوشه‌ی **Sources** را باز کنید تا فایل **main.cpp** مشاهده شود. این فایل تنها فایل برنامه است که توسط **qt creator** به صورت خودکار، ایجاد می‌شود. دستورات این فایل شامل برنامه‌ای است که در خروجی جمله معروف **Hello World** را چاپ می‌کند.

۷. بر روی لوگوی اجرا که در پایین سمت چپ qt creator است کلیک کنید تا برنامه کمپایل و اجرا شود:



کنسول سیستم شما ممکن است cmd ویندوز باشد ولی خروجی Hello World! مشابه بالا خواهد بود.

ضمیمه (۲) کمپایل با C++17، C++14 و C++11، آخرین استانداردهای پیاده شده در آن به طور در آخرین نسخه‌ی Visual Studio، آخرین استانداردهای پیاده شده در آن به طور خودکار قابل استفاده هستند. در صورت استفاده از `g++`، باید از `flag` زیر هنگام کمپایل، برای C++11:

`-std=C++11`

و یا:

`-std=C++14`

برای C++14 و یا:

`-std=++C1z`

برای کمپایل با استاندارد C++17 استفاده شود. برای مثال:

`g++ main.cpp -std=++C1z`

فایل منبع `main.cpp` را با استانداردهای C++17 کمپایل می‌کند.

در صورت استفاده از `qt creator` در فایل `pro` برای C++17 قرار دهید:
`CONFIG += ++C1z`

در صورت استفاده از IDE دیگر، این `flag` در تنظیمات `build` (یا در قسمتی مشابه) قابل افزودن است.

ضمیمه (۳) گرفتن آرگومان از `cli`

به جای پاس دادن آرگومان با `cin`، در مواردی پاس دادن مقادیر با اجرای برنامه مناسبتر است. مثلاً با دستور زیر در ویندوز:

```
program.exe 192.168.1.1 5000
```

یا در یونیکس به صورت:

```
./program 192.168.1.1 5000
```

در برنامه‌ای که آدرس `ip` و پورت را می‌گیرد و به سرور متصل می‌شود، استفاده از آرگومان `cli` به دلیل مشخص بودن آرگومانها در هنگام اجرای برنامه نسبت به استفاده از `cin` مناسبتر است. یکی از مشکلات `cin` در این است که در حین اجرا برنامه منتظر ورودی کاربر می‌شود که در این حالت نوشتن برنامه‌ای دیگر برای اتوماتیک کردن فرآیند اجرای برنامه دشوار می‌شود.

با اعلان `main` به صورت زیر:

```
int main(int argc, char** argv);
```

امکان پاس دادن آرگومان با اجرا از طریق `cli` فراهم می‌شود. در برنامه `argc` تعداد آرگومان‌ها و آرایه‌ی رشته از آرگومان‌ها است.
(مثال)

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

int main(int argc, char** argv) {
    for(auto i = 0; i < argc; ++i) {
        cout << *(argv[i]) << '\t';
    }
    cout << '\n';
}
```

با اجرا به صورت زیر، آرگومان‌ها در خروجی نمایش داده می‌شوند (برنامه‌ی `program`)

کمپایل شده در دایرکتوری است):

program.exe data1 data2 data3

یا در یونیکس به صورت:

./program data1 data2 data3

در برنامه‌هایی که از آرگومان `cli` استفاده می‌کنند، معمولاً از ساختاری شرطی برای اطمینان از دریافت آرگومان‌های مناسب استفاده می‌شود.

ضمیمه (۴) نوع بازگشتی کوواریانت (covariant) در صورتی که دو کلاس با نام‌های `Derived` و `Base` در برنامه داشته باشیم و `Derived` مشتق شده باشد، در بعضی از موقع (مثل سازندهی `virtual` از `Base` در `Derived`) کلاس مشتق، شامل تابع نادیده‌گرفته شده‌ای (`override`) می‌باشد که شی مشتق شده‌تری نسبت به `Base` را برمی‌گرداند. برای مثال:

```
class Base {
public:
    virtual Base* clone() const {
        return new Base();
    }
};

class Derived : public Base {
public:
    Base* clone() const {
        return new Derived();
    }
};
```

تعریف چنین تابع عضوی همانطور که انتظار می‌رود کاملاً صحیح است؛ زیرا اشاره‌گر به صورت `implicit` به اشاره‌گر `downcast` `Base*` می‌شود. ولی حالتی را در نظر بگیرید که شی `Derived` مشخص است و می‌خواهیم با `clone()` آن را کپی کنیم. در این حالت نمی‌توان دستورات زیر را برای مثال قبل نوشت:

```
Derived *d1 = new Derived();
Derived *d2 = d1->clone(); // error
```

یک راه حل در این مثال استفاده از `dynamic_cast` برای `upcast` کردن و استفاده از ساختار شرطی برای اطمینان از ایجاد گست است. برای مثال:

```
Derived* d = new Derived();
Base* b = d->clone();
Derived* d2 = dynamic_cast<Derived*>(b);
```

```
if(d2) {
    // checking for conversion success. if not, throw
    // exception, etc
}
```

در C++, تابع عضو نادیده‌گرفته شده در کلاس مشتق، می‌تواند شی مشتقشده‌تری نسبت به کلاس پایه‌ی خود را برگرداند؛ این نوع مقدار بازگشتی کوواریانت^۱ نام دارد. برای مثال کلاس‌های Base و Derived را به صورت زیر می‌توان بازنویسی کرد:

```
class Base {
public:
    virtual Base* clone() const {
        return new Base();
    }
};

class Derived : public Base {
public:
    Derived* clone() const {
        return new Derived();
    }
};
```

که در این حالت دستورات زیر بدون نیاز به گست کردن ممکن می‌شود:

```
Derived* d = new Derived;
Derived* d2 = d->clone(); // ok
```

استفاده از نوع بازگشتی کوواریانت برای وراثت private و protected و وراثت چندگانه نیز امکانپذیر است.

^۱ Covariant return type

ضمیمه (۵) در Most vexing parse C++

این عبارت یک قاعده در کمپایل C++ است که بیان میکند: در صورتی یک دستور هم به عنوان اعلان یک متغیر بطوری که با () مقداردهی اولیه شده و یک تابع معنی دهد، کمپایلر اعلان تابع را در نظر میگیرد.
(مثال)

```
#include <iostream>
using namespace std;
class MyClass{
    MyClass(){
        cout << "constructor called.\n";
    }
};

int main() {
    MyClass myObject();
}
```

با کمپایل و اجرای برنامه هیچ خروجی دریافت نمیشود. عدم خروجی رشته در کنسول به این معناست که شی myObject ساخته نشده و سازنده آن نیز اجرا نشده است. دلیل این عمل کرد خط زیر است:

```
MyClass myObject();
```

این دستور میتواند یک اعلان تابع معنی دهد که کمپایلر نیز این معنی را انتخاب میکند. در مثال بالا چون سازندهی کلاس MyClass بدون پارامتر است، برای جلوگیری از ابهام most vexing parse هنگام ساخت شی باید به صورت زیر عمل کرد:

```
MyClass myObject;
```

توجه شود که در عمل، most vexing parse در مکان هایی اتفاق میافتد که به راحتی مثال بالا قابل تشخیص نیست. در C++11 به بعد، ساخت شی با {} این ابهام را

ایجاد نمیکند. مثلاً برای کلاس MyClass دستور زیر:

```
MyClass myObject{};
```

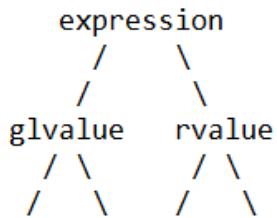
یک متغیر اعلان میکند.

از نظر تاریخی عبارت Scott اولین بار در کتاب most vexing parse مورد استفاده قرار گرفت. این تحت عنوان Meyers Effective STL (2001) تعريف شد.

ضمیمه(6) دسته بندی مقادیر

مقادیر در C++11 به بعد بر اساس دو ویژگی دسته بندی شده‌اند.
I: identity یا identity داشته باشد. I: identity نداشته باشد.
M: قابل انتقال باشد یا M: قابل انتقال نباشد.

ترکیب دو ویژگی فوق ۴ حالت ایجاد می‌کند که به دلیل بی کاربرد بودن IM (identity نداشته باشد و قابل انتقال نباشد)، ۳ حالت دیگر در نظر گرفته می‌شود. ۳ حالت بوجود آمده در دیاگرام زیر قابل مشاهده است.



سپس برای هر دسته اسمی انتخاب شده. انتخاب lvalue xvalue prvalue جایگزین اسم‌های انتخاب شده با حالت‌ها

موجود در دیاگرام بالا به صورت زیر می‌شود:
در لیست زیر اجزای هر دسته نوشته شده:

• lvalue: تعیین کننده‌ی یک تابع یا یک شی است. برای مثال:

```
"string"; // lvalue *
int a{}; // lvalue
int &a{10}; // lvalue
struct S{int a;};
S sobj; // lvalue
sobj.a; // lvalue **
```

* رشته برخلاف بقیه‌ی لیترال‌ها یک آرایه ذخیره می‌شود.
** اعضای غیر static از lvalue نیز lvalue است.

• xvalue (eXpiring value): به یک شی تلقی می‌شود که معمولاً در

پایان طول عمر آن است (تا فضایش جایجا شود مثلاً). برای مثال:

```
int a = 10;
std::move(a); // xvalue
```

- `xvalue` یا `lvalue` یا `gvalue` (generalized value) است.
- `rvalue` تعیین کننده‌ی یک `xvalue`, یک شی موقت, یا مقداری است که به یک شی وابسته نیست.
- یک `xvalue` است که `rvalue` (pure value) نیست.
- برای مثال:

```
42; // prvalue
int func();
func(); // prvalue
int a{};
&a; // prvalue
```

برای مقایسه، همان `lvalue` باقی مانده‌اند ولی `C03++` تغییر نام پیدا کرده. با وجود اینکه تعیین دسته‌ی یک `prvalue` میتواند مشکل باشد، در عمل میتوان از روش زیر برای مشخص `lvalue` و `rvalue` استفاده کرد:

1. اگر امکان دریافت آدرس یک `expression` وجود داشته باشد، یک `lvalue` است.
2. اگر نوع `expression` یک رفرنس `lvalue` (رفرنس عادی) باشد، عبارت یک `lvalue` است.
3. در غیر این صورت `rvalue` یک `expression` است.

