Learn C++

Jacob Bishop

2022-09-01

1 C++ Basics

1.1 Statements and the structure of a program

```
1 #include <iostream>
2
3 int main()
4 {
5    std::cout << "Hello world!";
6    return 0;
7 }
```

行 1 是一个被称为预处理器指令的特殊类型。其代表可以使用 iostream 库的所有内容，该库为 C++ 标准库的一部分，用于读写 console。该库用于第五行的 `std::cout` 。
行 3 告诉编译器用户准备定义一个 **main** 函数，所有的 C++ 程序必须拥有一个 **main** 函数否则会连接失败。
行 4 和 7 告诉编译器 **main** 的函数主体。
行 5 为 **main** 中的第一个声明，也是第一个声明在运行程序时被执行。`std::cout` (character output) 以及 `<<` 运算符使用户可以发送单词或者数字输出到终端。
行 6 为返回声明。当一个可执行程序结束运行时，程序会返回一个值给操作系统，用以通知其运行是否成功。值 0 意味着“所有运行皆正常！”。该行为程序运行的最后一个声明。

1.3 Introduction to objects and variables

对象与变量

所有的电脑都有内存，被称为 **RAM** (random access memory 的简称) 供程序使用。可以吧 RAM 看作是一系列的数字标记后的信箱，其在程序运行时可以存储一部分数据。被存储于内存的数据单个部分，被称为**值**。

C++ 中，直接的内存访问是不被孤立的。相对的，用户通过一个对象间接的访问内存。一个**对象**是一块存储（通常为内存）包含了值或者其他关联的属性。编译器与操作系统是如何为对象分配内存的问题超出了这个课程。但是关键点在于，与其说*获取编号 7532 信箱存储的值*，不如说*获取该对象的值*。这意味着用户可以专注于使用对象来存储与获取值，并不用再担心哪块的内存被使用了。
对象可以被命名或者匿名。一个命名的对象为一个**变量**，该对象的名称叫做一个标识符。

1.4 Variable assignment and initialization

初始化

- Default initialization：当没有初始值时，称为**默认初始化**。大多数情况下，默认初始化会给变量留下一个待定值。
- Copy initialization：当提供了一个初始值在等号后，称为**拷贝初始化**。继承于 C 语言。在现代 C++ 中不再常用。
- Direct initialization：当初始值在圆括号中提供，称为**直接初始化**。同拷贝初始化一样，现代 C++ 中不再常用。
- Brace initialization：现代 C++ 初始化变量的方式是使用花括号，称为**大括号初始化**，也可被称为**统一初始化** uniform initialization 或者**列表初始化** list initialization。

```
int width { 5 };    // 推荐，直接大括号初始化
int height = { 6 }; // 拷贝大括号初始化
int depth {};       // 值初始化
```

值初始化与零初始化

当一个变量使用空的大括号初始化时，**值初始化**便发生了。大多数情况下，**值初始化**将初始化变量为零（或者空，如果给定的类型更适合时）。这种情况下被称为**零初始化**。
何时使用 0 vs ?
使用显式初始化值，如果用户真正的使用该值。

```
int x { 0 };    // 显式初始化值为 0
std::cout << x; // 使用 0 值
```

使用值初始化如果值是临时的且会被覆盖的。

```
int x {};       // 值初始化
std::cin >> x;  // 用户立即替换该值
```

初始化多个变量

```
int a = 5, b = 6;         // 拷贝初始化
int c( 7 ), d( 8 );       // 直接初始化
int e { 9 }, f { 10 };    // 推荐，大括号初始化
```

1.7 Keywords and naming identifiers

alignas	do	reinterpret_cast
alignof	double	requires (since C++20)
and	dynamic_cast	return
and_eq	else	short
asm	enum	signed
auto	explicit	sizeof
bitand	export	static
bitor	extern	static_assert
bool	false	static_cast
break	float	struct
case	for	switch
catch	friend	template
char	goto	this
char8_t (since C++20)	if	thread_local
char16_t	inline	throw
char32_t	int	true
class	long	try
compl	mutable	typedef
concept (since C++20)	namespace	typeid
const	new	typename
consteval (since C++20)	noexcept	union
constexpr	not	unsigned
constinit (since C++20)	not_eq	using
const_cast	nullptr	virtual
continue	operator	void
co_await (since C++20)	or	volatile
co_return (since C++20)	or_eq	wchar_t
co_yield (since C++20)	private	while
decltype	protected	xor
default	public	xor_eq
delete	register	

2 Functions and Files

2.1 Introduction to functions

用户自己编写的函数被称为**用户定义函数** user-defined functions。
函数调用 function call 为一种表达式，用于告知 CPU 暂停现在的函数并执行另一个函数。
用于初始化函数调用的函数被称为**调用者** caller，而被调用的函数被称为 **callee** 或 **called function**。

```
return-type identifier() // 函数头（告知编译器函数的存在）
{
    // 函数体（告知编译器函数用于做什么）
}
```

2.7 Forward declarations and definitions

使用前向声明

前向声明 forward declaration 允许用户在真正定义标识符之前告知编译器该标识符的存在。
通过**函数声明** function declaration 的语句（也可称为**函数原型**）来编写函数的前向声明：

```
int add(int x, int y);  // 函数声明包含了返回值，名称，入参类型，以及分号。没有函数体!
```

那么可以这样使用：

```
#include <iostream>

int add(int x, int y);

int main()
{
    std::cout << "The sum of 3 and 4 is: " << add(3, 4) << '\n';
    return 0;
}

int add(int x, int y)
{
    return x + y;
}
```

其他类型的前向声明

前向声明经常用于函数上。然而，前向声明也可以用于其它的 C++ 标识符，例如变量或者用户定义类型。它们有与函数声明不同的语法，后续章节将会讲到。

声明 vs. 定义

定义为真正实现了（函数或者类型）或者实例化了（变量）标识符。为了满足 **linker**，定义是必须的。如果标识符没有提供定义，linker 则会报错。
单定义规则 one definition rule（简写 ODR）是 C++ 中熟知的规则。拥有三个部分：

- 给定的一个文件中，函数、变量、类型或者模板只允许有一种定义。
- 给定的一个程序中，变量或者普通函数只有一种定义。区别在于程序可以拥有若干文件（下个章节）。
- 类型，模板，内联函数以及内联变量允许有独立的定义于不同的文件中（后续章节）。

声明为语句用于告知编译器标识符的存在以及其类型信息。

2.9 Naming collisions and an introduction to namespaces

命名空间是一个域允许用户在其中声明名称以达到消除歧义的目的。命名空间提供一个区域范围（称为**命名空间域** namespace scope）。
在命名空间内，所有名称必须具有唯一性，否则命名冲突还是会出现。

2.10 Introduction to the preprocessor

翻译与预处理器

当编译器编译代码时，用户可能期望编译器能完全按照写好的代码来进行编译。实际上并非如此。
比编译更提前的是，代码文件会通过一个**翻译**阶段。很多事情会发生在该阶段。一个被应用的代码文件可以被称为一个**翻译单元**。
其中翻译阶段中最值得注意的是预处理器。预处理器最好被视为一个独立的程序用于操作每个代码文件。
当预处理器运行时，它扫描整个代码文件（从上至下），寻找预处理器指令。**预处理器指令**（通常被称为指令 directives）开始与 # 号结束于新的一行（而不是分号）。这些指令告诉预处理器执行指定的文本操作任务。注意预处理器不能理解 C++ 的语法，相反的是，其拥有自身的语法（在某些情况下与 C++ 语法相似，其他情况下则不同）。
预处理器的输出通过若干的翻译阶段，然后才被编译。注意预处理器无论如何也不会修改原始的代码文件，更确切来说，所有通过预处理器的文本变化只会发生在每次编译代码文件时的临时内存里或是临时文件上。

Includes

当 `#include` 一个文件，预处理器用引用的文件内容替换掉 #include 指令。该引用的内容则被预处理（相对于文件的其他部分）并被编译。

宏定义

`#define` 指令可以用于创建一个宏。在 C++ 中，一个**宏**是一个定义了如何输入文本转为替换的输出文本的规则。
有两种基本的宏类型：*类对象宏* object-like macros 以及 *类函数宏* function-like macros。
*类函数宏*类似于函数，服务于相同的目的。这里不做讨论，因为它们的使用通常被视为有风险的，同时它们能完成的东西，普通函数也能胜任。
*类对象宏*可以被定义为以下两种方式中的一种：

```
#define identifier
#define identifier substitution_text
```

前者没有替换文本，而后者有。它们都是预处理器指令（非语句），因此不需要以分号结束。

有替代文本的类对象宏

当预处理器遇到这类指令时，任何在其之后出现的这个标识符都将会被替换为*替换文本* substitution_text。标识符通常使用全大写，并使用下划线来表示空格。

```
#include <iostream>

#define MY_NAME "Alex"

int main()
{
    std::cout << "My name is: " << MY_NAME;

    return 0;
}
```

预处理器转换上述代码为：

```
// iostream 的内容在此被插入

int main()
{
    std::cout << "My name is: " << "Alex";

    return 0;
}
```

带有替代文本的类对象被使用为（在 C 中）一种指定名称为字面值的方式。这不再必要了，因为 C++ 拥有更好的方法了。此类类对象通常只在历史代码中出现。
推荐避免该类宏，因为有更适合的做法（将会在 4.13 中提到）。

无替代文本的类对象宏

```
#define USE_YEN
```

这种形式的宏正如所见：在之后出现该标识符的地方，用无做替换。
这看起来很没用，实际上确实对于文本替换而言是无用的。然而，这种形式的指令通常不是这么用的。稍后会提到用法。
有别于有替代文本的类对象宏，这种形式的宏通常是可以被接受的。

条件编译

条件编译预处理指令允许用户有条件的选择编译与否。有不同类型的条件编译指令，这里只覆盖三种最常用的：`#ifdef` ，`#ifndef` 以及 `#endif` 。
`#ifdef` 预处理器指令允许预处理器检查标识符在之前是否被 `#define` 了。如果是，则介于 `#ifdef` 与 `#endif` 之间的代码将被编译。如果否，则代码被忽略。

```
#include <iostream>

#define PRINT_JOE

int main()
{
#ifdef PRINT_JOE
    std::cout << "Joe\n"; // 将会被编译，因为 PRINT_JOE 被定义了
#endif

#ifdef PRINT_BOB
    std::cout << "Bob\n"; // 将会被忽略，因为 PRINT_BOB 没有被定义
#endif

    return 0;
}
```

`#ifndef` 则与 `#ifdef` 相反。

#if 0

```
#include <iostream>

int main()
```

```
{
    std::cout << "Joe\n";

#if 0 // 从这里开始不做编译
    std::cout << "Bob\n";
    std::cout << "Steve\n";
#endif // 直到这里

    return 0;
}
```

定义的域

指令处理是先于编译的，根据文件顺序而从上至下执行的。
观察一下程序：

```
#include <iostream>

void foo()
{
#define MY_NAME "Alex"
}

int main()
{
    std::cout << "My name is: " << MY_NAME;

    return 0;
}
```

尽管 `#define MY_NAME "Alex"` 看起来像是定义在 *foo* 函数中，预处理器并不会察觉，因为它不理解 C++ 概念例如函数。因此，该程序与无论在函数前后定义 `#define MY_NAME "Alex"` 的程序无异。对于普遍的阅读性而言，应该让 `#define` 标识符与函数外。
一旦预处理器结束时，文件中所有的标识符会被丢弃。这意味着指令仅仅作用于其定义位置至文件结束。同一个项目中，一个代码文件中的指令不会影响其他的代码文件。

2.11 Header files

使用标准库的头文件

如图 1 所示:

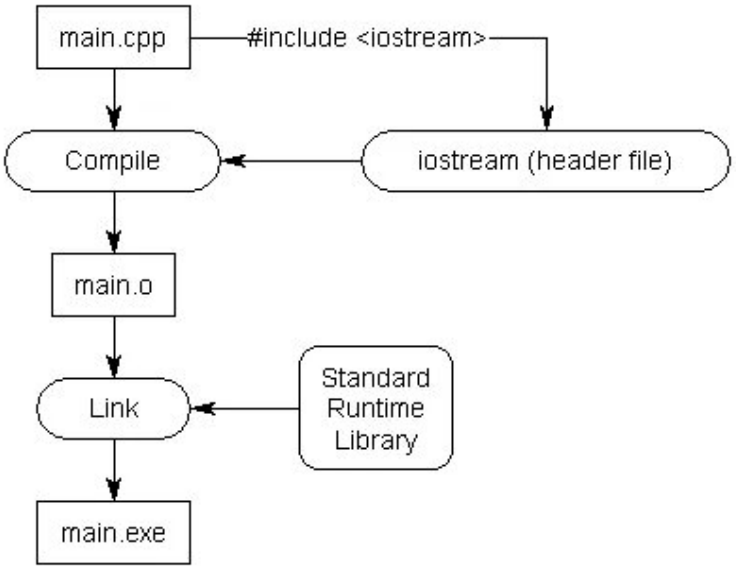

图 1: IncludeLibrary

编写自定义的头文件

add.cpp:

```
int add(int x, int y)
{
   return x + y;
}
```

main.cpp:

```
#include <iostream>

int add(int x, int y); // 前向声明使用函数原型

int main()
{
   std::cout << "The sum of 3 and 4 is " << add(3, 4) << '\n';
   return 0;
}
```

编写头文件非常简单，因为只包含了两部分：

1. 头文件保护符，下一章将会讨论。
2. 头文件的实际内容，其应该为前置定义的标识符，使得其他文件可以看到。

项目中添加一个头文件类似于添加一个源文件（2.8 的多文件程序将提到）。
头文件经常与代码文件配对，头文件为相关的代码文件提供前向声明。
add.h：

```
// 1. 需要一个头文件保护符在这里，将在下一章讲解，这里现在暂时忽略它

// 2. 这里是 .h 文件的内容，也是声明开始的地方
int add(int x, int y); // 函数原型 add.h -- 不要忘记分号!
```

main.cpp：

```
#include "add.h" // 插入 add.h 的内容到该处。注意这里使用双引号。
#include <iostream>

int main()
{
    std::cout << "The sum of 3 and 4 is " << add(3, 4) << '\n';
    return 0;
}
```

add.cpp：

```
#include "add.h" // 插入 add.h 的内容到该处。注意这里使用双引号。

int add(int x, int y)
{
    return x + y;
}
```

当预处理器处理 `#include "add.h"` 行时，拷贝 add.h 的内容至当前文件的该处。因为 *add.h* 包含了函数 *add* 的前向声明，前向声明将会被拷贝到 *main.cpp*。最终的结果是程序的功能与手动添加前向声明在 *main.cpp* 文件顶部的效果一直。

源文件应该包含其配对的头文件

在 C++ 中，代码文件的最佳时间是 `#include` 它们配对的头文件（如果存在的话）。上面的例子中 `add.cpp` 包含了 `add.h` 。

尖括号 vs 双引号

有可能会好奇为什么 `iostream` 使用的是尖括号，而 `add.h` 使用的是双引号。因为在不同的路径下可能存在相同名称的头文件。两种方式的使用可以帮助预处理器在何处寻找头文件。

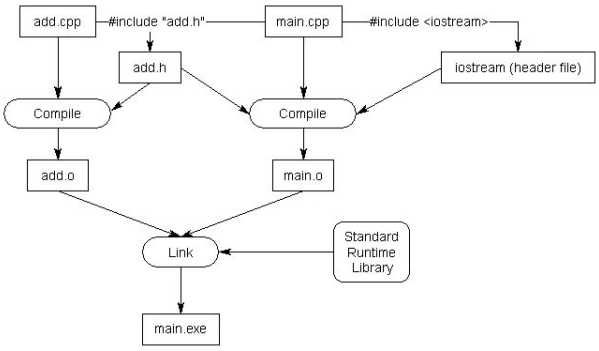

图 2: IncludeHeader

当使用尖括号时，则是告诉预处理器该头文件不是由用户编写的。预处理器将会寻找只指定于 `include directories` 路径下的头文件。`include directories` 由用户的项目/IDE 设置/编译器设置配置，通常默认路径中的头文件由用户编译器和/或操作系统所包含。预处理器不会在项目的源代码路径中寻找。
当使用双引号时，则是告诉预处理器头文件是由用户编写的。预处理器将会首先寻找当前路径。如果不能找到匹配的头文件，则会再从 `include directories` 中寻找。

头文件可能包含其他的头文件

最佳实践:
每个文件应该显式的 `#include` 所有其需要编译的头文件。不要依赖包含了其他头文件的头文件。

头文件的 #include 顺序

最佳实践:
为了最大化编译器发现缺失的 includes，请按照以下顺序进行 #includes:

1. 匹配的头文件
2. 本项目中其他的头文件
3. 第三方库的头文件
4. 标准库的头文件

其中每个组的头文件根据字母顺序进行排序。

这样的话，如果用户自定义的头文件缺失了三方库或者标准库的 #include，则很可能会产生编译错误，因此可以进行修复。
其他的最佳实践：

- 总是包含头文件保护符（详见下一章节）
- 不要在头文件中定义变量以及函数（全局变量除外 – 之后的章节将会覆盖）
- 头文件和与之关联的源文件拥有一样的名字（例如 grades.h 用于匹配 grades.cpp）
- 每个头文件需要有特定的任务，并且越独立越好。例如用户讲所有关联 A 功能的声明放入 A.h 中，所有关联 B 功能的声明放入 B.h 中。这样用户只需要关心 A 时 include A.h，而不会获得任何关联 B 的事务
- 注意头文件需要显式的在代码文件中 include 所需要的功能
- 所有用户编写的头文件应该可以自身被编译（应该 #include 所有其所需的依赖）
- 仅 #include 所需要的
- 不要 #include .cpp 文件

2.12 Header guards

通过**头文件保护符**这样的一种机制可以避免重复定义的问题（也可被称为 **include guard**）。通过以下形式，头文件保护符可以条件编译指令：

```
#ifndef SOME_UNIQUE_NAME_HERE
#define SOME_UNIQUE_NAME_HERE

// 用户定义（以及某些类型的定义）于此

#endif
```

当该头文件被 #included，预处理器检查 *SOME_UNIQUE_NAME_HERE* 是否之前就被定义了。如果是第一次引入该头文件，此时 *SOME_UNIQUE_NAME_HERE* 还未被定义。因此 #define *SOME_UNIQUE_NAME_HERE* 并且引入文件内容。如果该头文件再次被引入至同一个文件，*SOME_UNIQUE_NAME_HERE* 已经被定义了，该头文件则被忽略（多亏了 #ifndef）。
举个列子：
square.h:

```
#ifndef SQUARE_H
#define SQUARE_H

int getSquareSides()
{
    return 4;
}

#endif
```

geometry.h:

```
#ifndef GEOMETRY_H
#define GEOMETRY_H

#include "square.h"

#endif
```

main.cpp:

```
#include "square.h"
#include "geometry.h"

int main()
{
    return 0;
}
```

在预处理器解决完所有的 #include 指令后，程序会类似于这样：

main.cpp:

```
// Square.h 被 main.cpp include
#ifndef SQUARE_H // square.h 被 main.cpp include
#define SQUARE_H // SQUARE_H 获得定义

// 其所有的内容在此被 include
int getSquareSides()
{
    return 4;
}

#endif // SQUARE_H

#ifndef GEOMETRY_H   // geometry.h 被 main.cpp include
#define GEOMETRY_H
#ifndef SQUARE_H     // square.h 被 geometry.h include, SQUARE_H 之前已经被定义过了
#define SQUARE_H     // 因此没有内容在此被 include

int getSquareSides()
{
```

```
    return 4;
}

#endif // SQUARE_H
#endif // GEOMETRY_H

int main()
{
    return 0;
}
```

头文件保护符不阻止头文件被不同的代码文件引用多次

square.h:

```
#ifndef SQUARE_H
#define SQUARE_H

int getSquareSides()
{
    return 4;
}

int getSquarePerimeter(int sideLength); // getSquarePerimeter 的前置定义

#endif
```

square.cpp:

```
#include "square.h"  // square.h 在这里被引入一次

int getSquarePerimeter(int sideLength)
{
    return sideLength * getSquareSides();
}
```

main.cpp:

```
#include "square.h" // square.h 在这里又被引入一次
#include <iostream>

int main()
{
    std::cout << "a square has " << getSquareSides() << " sides\n";
    std::cout << "a square of length 5 has perimeter length " << getSquarePerimeter(5) << '\n';

    return 0;
}
```

注意 *square.h* 同时被 *main.cpp* 和 *square.cpp* 引入了，这就意味着 *square.h* 的内容也被分别引入进了两个代码文件。最终结果就是项目可以被编译但是 linker 会告知有多个 *getSquareSides* 标识符的定义。

避免这种情况最好的办法是简单的放置函数定义进一个.cpp 文件，因此头文件仅需包含前置定义：

square.h:

```
#ifndef SQUARE_H
#define SQUARE_H

int getSquareSides(); // getSquareSides 的前置定义
int getSquarePerimeter(int sideLength); // getSquarePerimeter 的前置定义

#endif
```

square.cpp:

```
#include "square.h"

int getSquareSides() // getSquareSides 的实际定义
{
return 4;
}

int getSquarePerimeter(int sideLength)
{
   return sideLength * getSquareSides();
}
```

main.cpp:

```
#include "square.h" // square.h 在此也被引用
#include <iostream>

int main()
{
   std::cout << "a square has " << getSquareSides() << "sides\n";
   std::cout << "a square of length 5 has perimeter length " << getSquarePerimeter(5) << '\n';

   return 0;
}
```

#pragma once

现代编译器提供了另外一个更简单的头文件保护符，那就是使用 *#pragma* 指令：

```
#pragma once

// 用户代码
```

`#pragma once` 更加简短以及不易出错。对于大多数项目而言 `#pragma once` 就可以很好的工作了。然而 `#pragma once` 并不属于 C++ 的官方语言（并且将永远不是，因为它不能在所有情况下都提供可靠性）。

2.13 How to design your first programs

设计步骤 1：定义目标

用一至两句换表达用户所需的结果，例如；

- 允许用户管理一组用户名称以及关联的电话号码。
- 生成随机的地下城且具有好看的洞穴。
- 生成一组拥有高分红的股票推荐。
- 为从塔顶落下的球需要多长时间着陆而建模。

设计步骤 2：定义需求

需求是对于约束以及能力双方而言的一个好听的词，前者是结果所需的约束条件（例如预算，时间轴，空间，内存等等），后者是程序展示的可以满足用户需求的能力。注意需求类似于专注于”什么（what）“，而不是”如何（how）“。
例如：

- 电话号码需要被存储，所以它们之后可以被重播。
- 随机的地下城应该总是包含一种通关方式。
- 股票推荐应该利用历史的价格数据。
- 用户可以进入塔楼的更高层。
- 需要 7 天之内的可测试版本。
- 程序应该在用户提交需求后的 10 秒之内产生结果。
- 程序应该仅有 0.1% 的崩溃率。

一个简单的问题可能会引出很多需求，同时解决方案并不会”结束“知道满足所有的需求。

设计步骤 3：定义工具，目标，以及备用计划

如果是一个有经验的程序员，则会有其他的一些步骤在此，例如：

- 定义目标架构以及/或者操作系统用于运行程序。
- 决定哪些工具能被使用。
- 决定是个人编写还是集体编写程序。
- 定义测试/反馈/发布的策略。
- 定义如何备份代码。

设计步骤 4：拆解困难问题成细小的简单问题

真实世界中，用户所执行的任务通常来说很复杂。直接尝试解决这些任务具有很大挑战，这种情况下需要使用**自上而下**的方法来解决问题。也就是说，不是直接解决单个复杂的任务，而是拆解任务至若干子任务，它们每个都可以简单的独立完成。如果子任务解决起来还是很困难，则继续拆解，最终得到可控的最小任务。

设计步骤 5：构想出事件的流程

项目有了结构之后便是要决定如何将所有的任务关联起来，即决定执行事件的序列。

实现步骤 1：规划 main 函数

例如：

```
int main()
{
//    doBedroomThings();
//    doBathroomThings();
//    doBreakfastThings();
//    doTransportationThings();

    return 0;
}
```

或者：

```
int main()
{
    // Get first number from user
    // getUserInput();

    // Get mathematical operation from user
```

```
    // getMathematicalOperation();

    // Get second number from user
    // getUserInput();

    // Calculate result
    // calculateResult();

    // Print result
    // printResult();

    return 0;
}
```

实现步骤 2：实现每个函数

每个函数需要：

1. 定义函数原型（输入和输出）
2. 编写函数
3. 测试函数

实现步骤 3：最终测试

测试整个项目。

其他编写程序时的建议

- **程序总是可以简单的启动。**
- **逐步增加功能。**
- **同一时间关注一块领域。**
- **编写代码的同时做好测试。**
- **不要预想着前期代码就完美。**

3 Debugging C++ Programs

3.2 The debugging process

一旦问题被识别，调试问题通常由五个步骤组成：

1. 寻找问题的根本原因（通常是不能运行的那一行代码）
2. 确保理解为什么该状况会发生
3. 决定如何修复该状况
4. 修复导致该状况的问题
5. 重新测试确保问题已被解决，并且没有新的问题产生

4 Fundamental Data Types

4.1 Introduction to fundamental data types

C++ 对于很多不同的数据类型具有内置的支持。这些被称为**基本数据类型** fundamental data types，不过也经常被不正式的称为 **basic types**，**primitive types**，或者 **built-in types**。

Types	Category	Meaning	Example
float double long double	Floating Point	a number with a fractional part	3.14159
bool	Integral (Boolean)	true or false	true
char wchar_t char8_t (C++20) char16_t (C++11) char32_t (C++11)	Integral (Character)	a single character of text 'c'	
short int long long long (C++11)	Integral (Integer)	positive and negative whole numbers, including 0	64
std::nullptr_t (C++11)	Null Pointer	a null pointer	nullptr
void	Void	no type	n/a

4.3 Object sizes and the sizeof operator

对象大小

现代机器的内存通常由成字节大小的单位构成，每个字节的内存都有位移的地址。

基本数据类型大小

Category	Type	Minimum Size	Note
boolean	bool	1 byte	
character	char wchar_t char16_t char32_t	1 byte 1 byte 2 bytes 4 bytes	Always exactly 1 byte
integer	short int long long long	2 bytes 2 bytes 4 bytes 8 bytes	
floating point	float double long double	4 bytes 4 bytes 8 bytes	

sizeof 操作符

sizeof 操作符是一个一元操作符，其入参为一个类型或者变量，其返回值则是字节大小。

```
#include <iostream>

int main()
{
    std::cout << "bool:\t\t" << sizeof(bool) << " bytes\n";
```

```
6     std::cout << "char:\t\t" << sizeof(char) << " bytes\n";
7     std::cout << "wchar_t:\t" << sizeof(wchar_t) << " bytes\n";
8     std::cout << "char16_t:\t" << sizeof(char16_t) << " bytes\n";
9     std::cout << "char32_t:\t" << sizeof(char32_t) << " bytes\n";
10    std::cout << "short:\t\t" << sizeof(short) << " bytes\n";
11    std::cout << "int:\t\t" << sizeof(int) << " bytes\n";
12    std::cout << "long:\t\t" << sizeof(long) << " bytes\n";
13    std::cout << "long long:\t" << sizeof(long long) << " bytes\n";
14    std::cout << "float:\t\t" << sizeof(float) << " bytes\n";
15    std::cout << "double:\t\t" << sizeof(double) << " bytes\n";
16    std::cout << "long double:\t" << sizeof(long double) << " bytes\n";
17
18    return 0;
19 }
```

打印结果:

```
bool:           1 bytes
char:           1 bytes
wchar_t:        2 bytes
char16_t:       2 bytes
char32_t:       4 bytes
short:          2 bytes
int:            4 bytes
long:           4 bytes
long long:      8 bytes
float:          4 bytes
double:         8 bytes
long double:    8 bytes
```

4.14 Compile-time constants, constant expressions, and constexpr

常数表达

常数表达 constant expression 是一种表达式，可以被编译器在编译期计算。常数表达中，所有的值需要在编译期可知（同时所有的操作符以及函数都必须支持编译器计算）。
当编译器遇到一个常数表达时，则在编译期计算表达式，接着用其计算结果替换掉表达式。

编译期常数

编译期常数是在编译期计算出的常数。例如'1'，'2.3' 以及"Hello, world!" 都可以是编译期常数。
那么常数变量呢？常数变量可以是编译期常数，也可以不是。

编译期常数变量

一个常数变量的初始化是常数表达式，那么它是编译期变量。

```
#include <iostream>

int main()
{
  const int x { 3 };  // x 是一个编译期常数
  const int y { 4 };  // y 是一个编译期常数

  const int z { x + y }; // x + y 是一个编译期表达式

  std::cout << z << '\n';

  return 0;
}
```

因为 x 与 y 它们的初始值是常数表达式，那么它们则是编译期常数，这意味着 `x + y` 同样是常数表达式。因此当编译器工作时，会计算 `x + y` 的值，并将常数表达式替换为计算的结果 7 。

运行时常数变量

任何通过非常数表达式的 const 变量皆为运行时常数。**运行时常数** runtime constants 是在运行时才能计算得出的常数。

```
#include <iostream>

int getNumber()
{
    std::cout << "Enter a number: ";
    int y{};
    std::cin >> y;

    return y;
}

int main()
{
    const int x{ 3 };           // x 是一个编译期常数

    const int y{ getNumber() }; // y 是一个运行时常数

    const int z{ x + y };       // x + y 是一个运行时表达式
    std::cout << z << '\n';     // 这同样也是一个运行时表达式

    return 0;
}
```

尽管 y 是一个 const，其初始值（即 'getNumber()' 的返回值）直到运行时才知道。因此，y 是运行时常数，而不是编译期常数。因此 `x + y` 也是一个运行时表达式。

constexpr 关键字

当用户声明一个常数变量，编译期将会隐式的追踪其是否为一个运行时或者是编译期常数。多数情况下，这不会作为优化的目标，但是有一些奇怪的案例中，C++ 需要编译期常数而非运行时常数（稍后将会提到）。
因为编译期常数通常允许更好的优化（同时很少的坏处），在可能的情况下，用户通常会选择使用编译期常数。
使用 const 时，变量会成为那种常数是根据初始化是否为编译期表达式来决定的。因为它们的定义看起来都一样，用户很容易误认一个运行时常数为编译期常数。上面的例子中，很难知道 y 是否为编译期常数 – 除非用户去检查 `getNumber()` 的返回类型。
幸运的是，用户可以从编译期中获得帮助，来确保编译时常数符合预期。为了达到这个目的，用户需要使用 `constexpr` 关键字而不是 `const` 变量声明。**constexpr**（常数表达式 constant expression 的简称）变量只允许是编译期常数。如果其初始值不是一个常数表达式，编译器则会报错。

```
#include <iostream>

int five()
{
    return 5;
}

int main()
{
    constexpr double gravity { 9.8 }; // ok: 9.8 是一个常数表达式
    constexpr int sum { 4 + 5 };      // ok: 4 + 5 是一个常数表达式
    constexpr int something { sum };  // ok: sum 是一个常数表达式

    std::cout << "Enter your age: ";
    int age{};
    std::cin >> age;

    constexpr int myAge { age };      // 编译错误: age 不是一个常数表达式
    constexpr int f { five() };       // 编译错误: five() 的返回值不是一个常数表达式

    return 0;
}
```

最佳实践:

- 任何变量在初始化后，不会变化的并且其初始化器在编译期可知，则为 `constexpr` 。

- 任何变量在初始化后，不会变化的并且其初始化器在编译期不可知，则为 `const` 。

常数合并常数的子表达

观察以下案例：

```
#include <iostream>

int main()
{
  constexpr int x { 3 + 4 }; // 3 + 4 是一个常数表达式
  std::cout << x << '\n';    // 这是运行时表达式

  return 0;
}
```

`3 + 4` 是常数表达式，因此编译器会在编译期计算并将其结果 `7` 替换掉表达式。编译器之后将会优化 `x` ，替换 `std::cout << x << 'n'` 为 `std::cout << 7 << 'n'` 。其返回表达式则在运行时执行。
然而，因为 `x` 只会用到一次，用户很可能在编写代码的时候这样写：

```
#include <iostream>

int main()
{
  std::cout << 3 + 4 << '\n'; // 这是运行时表达式

  return 0;
}
```

由于 `std::cout << 3 + 4 << 'n'` 不是一个常数表达式。那么有理由怀疑常数的子表达式 `3 + 4` 是否还是可以在编译期优化。这里的回答通常是“是”。编译器一直能够优化常数表达式，也包含了那些可以在编译期就决定好了的变量（编译期常数以及 constexpr 变量）。
让变量成为 constexpr 可以确保这些变量在编译时就能知道，因此有资格获得合并常数用于它们的表达式（即使在非常数表达式中）。

5 Operators

5.1 Operator precedence and associativity

运算符优先级

一个表达式中带有多个运算符可以被称为**组合表达式** compound expression。

运算符结合律

在一个表达式中，如果两个运算符拥有相同优先级并且互相关联，运算符**结合律**则会告知编译器是从左至右或是从右至左做计算。

Prec Ass	Operator	Description	Pattern
1 L->R	::	Global scope (unary)	name
	::	Namespace scope (binary)	class_name member_name
2 L->R	()	Parentheses	(expression)
	()	Function call	function_name(params)
	()	Initialization	type name(expression)
	()	Uniform initialization (C++11)	type name(expression)
	type()	Functional cast	new_type(expression)
	type()	Functional cast (C++11)	new_typeexpression
	[]	Array subscript	pointer[expression]
	.	Member access from object	object.member_name
	− >	Member access from object ptr	object_pointer->member_name
	++	Post-increment	lvalue++
	−−	Post-decrement	lvalue–
	typeid	Run-time type information	typeid(type) or typeid(expr)
	const_cast	Cast away const	const_cast<type>(expr)
	dynamic_cast	Run-time type-checked cast	dynamic_cast<type>(expr)
	reinterpret_cast	Cast one type to another	reinterpret_cast<type>(expr)
	static_cast	Compile-time type-checked cast	static_cast<type>(expr)
	sizeof...	Get parameter pack size	sizeof···(expression)
	noexcept	Compile-time exception check	noexcept(expression)
	alignof	Get type alignment	alignof(Type)
3 R->L	+	Unary plus	+expression
	−	Unary minus	-expression
	++	Pre-increment	++lvalue
	−−	Pre-decrement	–lvalue
	!	Logical NOT	!expression
		Bitwise NOT	expression
	(*type*)	C-style cast	(new_type)expression
	sizeof	Size in bytes	sizeof(type) or sizeof(expr)
	co_await	Await asynchronous call	co_await expression
	&	Address of	&lvalue
	*	Dereference	*expression
	new	Dynamic memory allocation	new type
	new[]	Dynamic array allocation	new type[expression]
	delete	Dynamic memory deletion	delete pointer
	delete[]	Dynamic array deletion	delete[] pointer
4 L->R	− > *	Member pointer selector	object_pointer->*pointer_to_member
	.*	Member object selector	object.*pointer_to_member
5 L->R	*	Multiplication	expression * expression
	/	Division	expression / expression
	%	Modulus	expression % expression
6 L->R	+	Addition	expression + expression
	−	Subtraction	expression - expression
7 L->R	<<	Bitwise shift left	expression « expression
	>>	Bitwise shift right	expression » expression
8 L->R	<=>	Three-way comparison	expression <=> expression
9 L->R	<	Comparison less than	expression < expression
	<=	Comparison less than or equals	expression <= expression
	>	Comparison greater than	expression > expression
	>=	Comparison greater than or equals	expression >= expression
10 L->R	==	Equality	expression == expression
	! =	Inequality	expression != expression
11 L->R	&	Bitwise AND	expression & expression
12 L->R	∧	Bitwise XOR	expression ∧ expression
13 L->R	|	Bitwise OR	expression | expression
14 L->R	&&	Logical AND	expression && expression
15 L->R	||	Logical OR	expression || expression
16 R->L	*throw*	Throw expression	throw expression
	co_yield	Yield expression	co_yield expression
	? :	Conditional	expression ? expression : expression
	=	Assignment	lvalue = expression
	* =	Multiplication assignment	lvalue *= expression
	/ =	Division assignment	lvalue /= expression
	% =	Modulus assignment	lvalue %= expression
	+ =	Addition assignment	lvalue += expression
	− =	Subtraction assignment	lvalue -= expression
	<<=	Bitwise shift left assignment	lvalue «= expression
	>>=	Bitwise shift right assignment	lvalue »= expression
	&	Bitwise AND assignment	lvalue &= expression
	| =	Bitwise OR assignment	lvalue |= expression
	∧ =	Bitwise XOR assignment	lvalue ∧ = expression
17 L->R	,	Comma operator	expression , expression

5.6 Relational operators and floating point comparisons

关系操作符 relational operators 是用于比较两个值的操作符。一共有六种：

操作符	符号	形式	操作
Greater than	>	x > y	true if x is greater than y, false otherwise
Less than	<	x < y	true if x is less than y, false otherwise
Greater than or equals	>=	x >= y	true if x is greater than or equal to y, false otherwise
Less than or equals	<=	x <= y	true if x is less than or equal to y, false otherwise
Equality	==	x == y	true if x equals y, false otherwise
Inequality	!=	x != y	true if x does not equal y, false otherwise

比较计算后的浮点值会出问题

考虑以下程序：

```
#include <iostream>

int main()
{
    double d1{ 100.0 - 99.99 }; // should equal 0.01 mathematically
    double d2{ 10.0 - 9.99 }; // should equal 0.01 mathematically

    if (d1 == d2)
        std::cout << "d1 == d2" << '\n';
    else if (d1 > d2)
        std::cout << "d1 > d2" << '\n';
    else if (d1 < d2)
        std::cout << "d1 < d2" << '\n';

    return 0;
}
```

变量 d1 与 d2 应该都为 0.01，然而程序打印了非预期的结果：

```
d1 > d2
```

如果在调试器中检查 d1 与 d2 的值，那么可能会得到 d1 = 0.0100000000000005116 以及 d2 = 0.0099999999999997868。两个数字都接近 0.01，但是 d1 要大一些，而 d2 要小一些。
如果需要高精度，那么使用任何关系操作符来比较浮点数是很危险的。这是因为浮点数并不精确，且对其进行小的取整可能会导致非预期的结果。

浮点数等式

使用等式操作符（!= 以及 ==）的问题更大。考虑操作符 == 是在运算成员完全相等的情况下返回 true。由于最小的取整错误都会导致两个浮点数并不相等，操作符 == 具有返回 false 的

风险，即便预期是 true 的情况。
因为这个原因，浮点运算成员使用这类操作符应该通常都被避免。

6 Scope, Duration, and Linkage

6.1 Compound statements(blocks)

组合声明 compound statement（同时也可称为**块** block，或**块声明** block statement）是一组零个或以上的声明被编译器视为单个声明。
代码块开始与 { 符号并结束于 } 符号，其中包含了可执行的声明。只要单个声明可以存在的地方，代码块就可以在那里存在。代码块不需要以分号结束。

6.2 User-defined namespaces and the scope resolution operator

自定义命名空间

C++ 允许用户通过 `namespace` 关键字来定义自己的命名空间。这样的命名空间被称为**用户定义命名空间** user-defined namespaces。

```cpp
#include <iostream>

namespace foo // 定义命名空间 foo
{
    // doSomething() 属于 foo 命名空间
    int doSomething(int x, int y)
    {
        return x + y;
    }
}

namespace goo // 定义命名空间 goo
{
    // doSomething() 属于 goo 命名空间
    int doSomething(int x, int y)
    {
        return x - y;
    }
}

int main()
{
    std::cout << foo::doSomething(4, 3) << '\n'; // 使用 foo 命名空间的 doSomething()
    std::cout << goo::doSomething(4, 3) << '\n'; // 使用 goo 命名空间的 doSomething()
    return 0;
}
```

若干命名空间（同名）也被允许

在不同的地方（不同的文件下，或者是同一文件下的不同位置），声明若干个命名空间是合法的。所有拥有相同名称的不同声明处，可被视为命名空间的组成部分。

circle.h:

```
#ifndef CIRCLE_H
#define CIRCLE_H

namespace basicMath
{
    constexpr double pi{ 3.14 };
}

#endif
```

growth.h:

```
#ifndef GROWTH_H
#define GROWTH_H

namespace basicMath
{
    // 常数 e 同样也是 basicMath 命名空间的一部分
    constexpr double e{ 2.7 };
}

#endif
```

main.cpp:

```
#include "circle.h" // basicMath::pi
#include "growth.h" // basicMath::e

#include <iostream>

int main()
{
    std::cout << basicMath::pi << '\n';
    std::cout << basicMath::e << '\n';

    return 0;
}
```

当分隔代码至若干文件时，用户可以在头文件与源文件使用命名空间。

add.h:

```
#ifndef ADD_H
#define ADD_H

```

```
namespace basicMath
{
    // 函数 add() 是 basicMath 命名空间的一部分
    int add(int x, int y);
}

#endif
```

add.cpp:

```
#include "add.h"

namespace basicMath
{
    // 定义函数 add()
    int add(int x, int y)
    {
        return x + y;
    }
}
```

main.cpp:

```
#include "add.h" // 使用 basicMath::add()

#include <iostream>

int main()
{
    std::cout << basicMath::add(4, 3) << '\n';

    return 0;
}
```

嵌套命名空间

```
#include <iostream>

namespace foo
{
    namespace goo // goo is a namespace inside the foo namespace
    {
        int add(int x, int y)
        {
            return x + y;
        }
    }
}

```

```cpp
int main()
{
    std::cout << foo::goo::add(1, 2) << '\n';
    return 0;
}
```

C++17 之后还可以这样声明命名空间：

```cpp
#include <iostream>

namespace foo::goo // goo 是一个在 foo 命名空间里的命名空间（C++17 风格）
{
  int add(int x, int y)
  {
    return x + y;
  }
}

int main()
{
    std::cout << foo::goo::add(1, 2) << '\n';
    return 0;
}
```

命名空间别名

C++ 允许用户创建**命名空间别名**，即可以临时缩短很长一串的命名空间成为更短的名称：

```cpp
#include <iostream>

namespace foo::goo
{
    int add(int x, int y)
    {
        return x + y;
    }
}

int main()
{
    namespace active = foo::goo; // active 现在指向 foo::goo

    std::cout << active::add(1, 2) << '\n'; // 这即是 foo::goo::add()

    return 0;
} // active 的别名在此处结束
```

6.6 Internal linkage

本地变量，我们称“一个标识符的链接决定了其它对该名称的声明，是否指向同一个对象”，那么本地变量是 `no linkage` 的。
全局变量以及函数标识符既可以是 `internal linkage` 也可以是 `external linkage` 。本章将会讲解 `internal linkage` ，下一章讲解 `external linkage` 。
一个 **internal linkage** 的标识符仅可以在单个文件中可视以及可用，并不能被其它文件访问（也就是说，它没有被暴露给 linker）。这就意味着如果两个文件拥有同名的标识符且是 internal linkage 的，那么它们则被视为独立。

带有 internal linkage 的全局变量

带有 internal linkage 的全局变量有时候会被称为 **内部变量** internal variables。
使用 `static` 关键字可以让非常数全局变量 internal。

```
static int g_x; // 非常数全局变量默认是 external linkage 的，通过 static 关键字使其变为
    internal linkage

const int g_y { 1 }; // const 全局变量默认是 internal linkage 的
constexpr int g_z { 2 }; // constexpr 全局变量默认是 internal linkage 的

int main()
{
    return 0;
}
```

带有 internal linkage 的函数

由于 linkage 是一个标识符的属性（而不是变量），函数标识符也有同样的 linkage 属性。函数默认是 external linkage（详见下一章），但是可以通过 `static` 关键字设置成 internal linkage。
add.cpp:

```
// 函数本声明为 static，现在只能在本文件中可用
// 在其他文件中通过前向声明尝试访问它则会失败
static int add(int x, int y)
{
    return x + y;
}
```

main.cpp:

```
#include <iostream>

int add(int x, int y); // 函数 add 的前向声明

```

```
5  int main()
6  {
7      std::cout << add(3, 4) << '\n';
8
9      return 0;
10 }
```

6.7 External linkage and variable forward declarations

上一章中讲到 `internal linkage` 限制了标识符仅可以使用在单个文件中。这一章将探讨 `external linkage` 。

带有 `external linkage` 的标识符同时可视并可用于其定义的文件，以及其他代码文件（通过前向声明）。这种情况下，带有 `external linkage` 的标识符是真正意味上的“全局”，可在项目的任何位置使用。

函数默认带有 external linkage

2.8 中学到的用户可以在其他的文件中，从其定义的地方调用函数。这是因为函数默认是 external linkage 的。

为了在其他文件中调用，用户必须为需要调用的函数放置一个前向声明 `forward declaration` 。前向声明可以告诉编译器函数的存在，linker 则关联函数调用至其真正定义的地方。

全局变量带有 external linkage

带有 external linkage 的全局变量有时被称为 **external variables**。用户可以使用 `extern` 关键字使一个全局变量 external（以便其它文件访问）。

变量通过 extern 关键字前向声明

为了在其它文件中可以使用到 external 的全局变量，用户在其他文件中必须放置该变量的前向声明 `forward declaration` 。而对于变量而言，为其创建一个前向声明也是通过 `extern` 关键字（并且不带初始值）。

文件作用域 vs. 全局作用域

“文件作用域”与“全局作用域”这两个概念有可能搞混淆，这是因为它们都是被非正式的使用。

考虑以下程序：

global.cpp:

```
1 int g_x { 2 }; // 默认为 external linkage
2 // g_x 在此离开作用域
```

main.cpp:

```
extern int g_x; // 前向声明 g_x -- g_x 可以被用在这里（超出 global.cpp 的文件作用域）

int main()
{
    std::cout << g_x << '\n'; // 应该打印 2

    return 0;
}
// g_x 的前向声明在这里结束
```

变量 g_x 在 global.cpp 中拥有文件作用域 – 它可以使用在定义起始的位置，直到文件结束，但是它不可以直接在 global.cpp 外可视。
在 main.cpp 中，g_x 的前置定义同样也有文件作用域 – 它可以使用在定义起始的位置，直到文件结束。
然而，非正式的“文件作用域”更经常被用于带有 internal linkage 的全局变量，而“全局作用域”被用于带有 external linkage 的全局变量（因为它们可以被用于程序的任何地方，只需要有正确的前向声明）。

6.9 Sharing global constants across multiple files (using inline variables)

在一些应用中，特定的有意义的常量会在整个项目代码中用到。它们可能包含了物理学或是数学不变的常数（例如 或者阿佛加德罗常量），或是应用的“调参”值（例如摩擦系数或者重力系数）。不需要在每个文件中重新定义这些常数，而是仅集中声明它们一次并在任何需要的地方使用。这样的话，如果需要修改的时候仅需要改一处地方。

作为内部变量的全局常数

在 C++17 之前，以下步骤是最方便以及常用的解决方案：

- 创建一个包含这些常量的头文件
- 头文件中，定义命名空间
- 在命名空间内添加所有需要的常量（确保它们都是 *constexpr*）
- 在需要的地方 #include 头文件

constants.h:

```
#ifndef CONSTANTS_H
#define CONSTANTS_H

```

```
// 自定义命名空间用于保存常量
namespace constants
{
    // 常量默认为 internal linkage
    constexpr double pi { 3.14159 };
    constexpr double avogadro { 6.0221413e23 };
    constexpr double myGravity { 9.2 }; // m/s^2 -- 这个星球的重力小一点
    // ... 其它相关常量
}
#endif
```

使用范围解析操作符 scope resolution operator (::) 与命名空间（位于操作符左侧）以及定义的变量名称在右侧，访问定义好的常量：

main.cpp:

```
#include "constants.h" // 本文件中 include 了这些常量的拷贝

#include <iostream>

int main()
{
    std::cout << "Enter a radius: ";
    int radius{};
    std::cin >> radius;

    std::cout << "The circumference is: " << 2.0 * radius * constants::pi << '\n';

    return 0;
}
```

作为外部变量的全局常数

上述的方法有几个缺点。

如果是简单的（适合小型程序），每次 constant.h 被 #include 到不同的代码文件，每个变量都被拷贝到代码文件中。因此如果 constants.h 被 include 到了 20 个不同文件中，每个变量就被拷贝了 20 次。头文件保护符并不会停止这事发生，它们仅仅阻止头文件在同一个文件中被 include 多次，而不是阻止头文件被多个不同的文件 include。这就带来了两个挑战：

1. 改变一个常数将需要每个引用其的文件重新编译，这就导致了大项目的编译时间大大延长。
2. 如果常数所需空间很大并且无法被优化，则会使用大量的内存。

避免上述问题的方法之一是使这些常量变为外部变量，因为我们可以拥有一个单变量（初始化一次）并在所有文件中共享。这个方法，用户需要定义常量与一个.cpp 文件中（为了确保定义只存在于一个处），并放置前向声明在头文件中（以便其它文件引入）。

注意：这里使用 const 而不是 constexpr，因为 constexpr 变量不可以被前向声明，即便他们有 external linkage。这是因为编译器需要在编译时知道这个变量的值，而前向声明并不提供该信息。

constants.cpp:

```cpp
#include "constants.h"

namespace constants
{
    // 真实全局变量
    extern const double pi { 3.14159 };
    extern const double avogadro { 6.0221413e23 };
    extern const double myGravity { 9.2 }; // m/s^2
}
```

constants.h:

```cpp
#ifndef CONSTANTS_H
#define CONSTANTS_H

namespace constants
{
    // 由于真实的变量在命名空间中，前向声明同样也需要在命名空间内
    extern const double pi;
    extern const double avogadro;
    extern const double myGravity;
}

#endif
```

使用代码与之前的方法相同：

```cpp
#include "constants.h" // 引用所有的前向声明

#include <iostream>

int main()
{
    std::cout << "Enter a radius: ";
    int radius{};
    std::cin >> radius;

    std::cout << "The circumference is: " << 2.0 * radius * constants::pi << '\n';

    return 0;
}
```

因为全局标识的常数需要命名空间（避免与其他的全局命名空间中的标识符冲突），“g_”前缀的命名则不再需要了。

然而，这个方法也有一些缺陷。首先这些常数仅仅在它们定义处（`constants.cpp`）被视为编译时的常数。其他文件中，编译器只能看到前向声明，即不定义常量（且必须有 linker 解决）。这就意味着在其他文件中，它们被视为运行时常数而不是编译期常数。因此在 `constants.cpp` 之外，这些变量不能被用于任何需要编译期常数的地方。其次，因为编译期常数通常会比运行时常数得到更多的优化，编译器对这些常数的优化支持就没有这么多了。
备注：变量为了被视为编译期的可用内容，例如数组大小，编译器必须知道变量的定义（而不仅仅是前向定义）。
因为编译器对每个源文件都是单独编译的，它仅可以通过源文件（也包含了任何引用的头文件）中所在的变量看到其定义。例如，定义在 `constants.cpp` 中的变量在编译器编译 `main.cpp` 时是不可见的。因此，`constexpr` 变量不能被分离至头文件以及源文件中，它们必须被定义在头文件中。

作为内联变量的全局常数 C++17

C++17 引入了一个新的概念称为 `incline variables`。C++ 中，称为 `inline` 的以及进化到意为“若干定义皆被允许”。因此，一个 ** **内联变量** inline variable 就是变量允许被定义在若干文件中且不危害唯一性规则。内联全局变量默认拥有 external linkage。
linker 将会合并一个变量的所有内联定义成为单个变量定义（因此符合单个定义规则）。这使得用户可以在头文件中定义变量，同时使变量被视为在某个.cpp 文件中的唯一定义。也就是说有一个常量被 #include 到了 10 个代码文件中。没有 inline，用户会得到 10 个定义。有了 inline，编译器则会选择一个定义为典型定义 canonical definition，因此用户只会得到一个定义。这就意味着省下了 9 个常量的内存。
这些常数在所有被引用的文件中，同样也保留了 constexpr 功能，所以他们可以被用于任何 constexpr 值所需要的地方。相比于运行时常数（或者非 const）变量，constexpr 值则可以被编译器高度优化。
内联变量拥有两个主要的约束需要被遵守：

1. 所有的内联变量的定义需要时唯一的（否则未定义行为会出现）。
2. 内联变量定义（不是前向声明）必须出现在任何文件中使用的变量。

constants.h:

```
#ifndef CONSTANTS_H
#define CONSTANTS_H

// 自定义命名空间用于保存常量
namespace constants
{
    inline constexpr double pi { 3.14159 }; // 注意：现在是 inline constexpr
```

```
    inline constexpr double avogadro { 6.0221413e23 };
    inline constexpr double myGravity { 9.2 }; // m/s^2
    // ... 其他相关变量
}
#endif
```

main.cpp:

```
#include "constants.h"

#include <iostream>

int main()
{
    std::cout << "Enter a radius: ";
    int radius{};
    std::cin >> radius;

    std::cout << "The circumference is: " << 2.0 * radius * constants::pi << '\n';

    return 0;
}
```

用户可以引用 `constants.h` 至多个代码文件，这些变量仅会被初始化一次并可在所有代码文件中共享。

这个方法仍然保留了在任何变量修改后，需要所有引用其的文件重新进行编译的缺陷。如果需要总是改变常数（例如调参的目的）并且导致更长的编译时间，移动常变的常数至其自己的头文件中（为了减少 #include 的次数）可能会更有帮助。

最佳实践：如果需要全局常数，同时可以使用 C++17 编译器，那么推荐在头文件中定义内联变量。

提醒：使用 `std::string_view` 为 `constexpr` 字符串。

6.14 Constexpr and consteval functions

Constexpr 函数可以在编译期被计算

一个带有 **constexpr** 函数的返回值，有可能会在编译期被计算。那么只需要在函数的返回值前面加上一个 `constexpr` 关键字就可以使一个函数成为 constexpr 函数。

```
#include <iostream>

constexpr int greater(int x, int y) // constexpr 函数
{
    return (x > y ? x : y);
}

```

```cpp
int main()
{
    constexpr int x{ 5 };
    constexpr int y{ 6 };

    // 稍后会解释这里为何使用变量 g
    constexpr int g { greater(x, y) }; // 将在编译期被计算

    std::cout << g << " is greater!\n";

    return 0;
}
```

具备编译期计算的能力，一个函数必须拥有一个 constexpr 的返回类型同时不调用任何非 constexpr 函数。除此之外，调用函数者必须是 constexpr 参数（例如 constexpr 变量或者字面值）。

Constexpr 函数是隐式内联的

因为 constexpr 函数有可能在编译期被计算，在其被调用处，编译器必须知道其所有的定义。这就意味着在若干文件中调用的 constexpr 函数需要其定义也被引用至各个文件中 – 这通常违反了单定义规则。为了避免这种情况，constexpr 函数是隐式内联的，使得它们从单独定义规则中豁免。
结果就是 constexpr 函数通常定义在头文件中，这样他们可以被 #include 到任何需求全定义的.cpp 文件中。

Constexpr 函数同样也可以在运行时被计算

```cpp
#include <iostream>

constexpr int greater(int x, int y)
{
    return (x > y ? x : y);
}

int main()
{
    int x{ 5 }; // 非 constexpr
    int y{ 6 }; // 非 constexpr

    std::cout << greater(x, y) << " is greater!\n"; // 将在运行时被计算

    return 0;
}
```

Constexpr 函数什么时候会在编译期执行?

```cpp
#include <iostream>

constexpr int greater(int x, int y)
{
    return (x > y ? x : y);
}

int main()
{
    constexpr int g { greater(5, 6) };              // case 1: 编译期计算
    std::cout << g << " is greater!\n";

    int x{ 5 }; // 非 constexpr
    std::cout << greater(x, 6) << " is greater!\n"; // case 2: 运行时计算

    std::cout << greater(5, 6) << " is greater!\n"; // case 3: 有可能是编译期计算，也有可能是运行时计算，根据编译器优化等级决定

    return 0;
}
```

判断 constexpr 函数是编译期计算还是运行时计算

C++20 之前是没有合适的标准语言工具的。
在 C++20 中，`std::is_constant_evaluated()`（定义在 <type_traits> 头文件中）返回一个 `bool` 表示当前函数调用是否执行在常数上下文中。这可以与条件声明结合起来，使一个函数的行为根据编译期或运行时做出不同的行为。

```cpp
#include <type_traits> // 内含 std::is_constant_evaluated

constexpr int someFunction()
{
    if (std::is_constant_evaluated()) // 如果是编译期计算
        // 做什么
    else // 运行时计算
        // 做其他
}
```

Consteval C++20

C++20 引入关键字 **consteval**，用以表示一个函数必须在编译时计算，否则会返回编译错误。这种函数被称为 **immediate functions**。

```cpp
#include <iostream>
```

```
2
3  consteval int greater(int x, int y) // 函数为 consteval
4  {
5      return (x > y ? x : y);
6  }
7
8  int main()
9  {
10     constexpr int g { greater(5, 6) };              // ok: 编译期计算
11     std::cout << greater(5, 6) << " is greater!\n"; // ok: 编译期计算
12
13     int x{ 5 }; // 非 constexpr
14     std::cout << greater(x, 6) << " is greater!\n"; // error: consteval 函数必须在编译期计算
15
16     return 0;
17 }
```

最佳实践：如果因为某种原因（例如性能）函数必须在编译期执行，请使用 **consteval**。

使用 consteval 使 constexpr 在编译期执行 C++20

consteval 这类函数的缺点就是不能在运行时计算，使得它们相较于 constexpr 函数少了一些灵活性。因此，拥有一个方便的方法迫使 constexpr 函数在编译期计算（即使其返回值被用在不需要的常数表达式中），那么可以尽可能的编译期计算，并且不可能时运行时计算。
consteval 函数提供了一种方式使其成为可能，使用一个干净的帮助函数：

```
1  #include <iostream>
2
3  // 使用 abbreviated 函数模版 (C++20) 以及 `auto` 返回值使得该函数可以对任何类型值生效
4  consteval auto compileTime(auto value)
5  {
6      return value;
7  }
8
9  constexpr int greater(int x, int y) // 函数为 constexpr
10 {
11     return (x > y ? x : y);
12 }
13
14 int main()
15 {
16     std::cout << greater(5, 6) << '\n';              // 编译期可能执行
17     std::cout << compileTime(greater(5, 6)) << '\n'; // 编译期执行
18
19     int x { 5 };
20     std::cout << greater(x, 6) << '\n';              // 仍然可以在运行时调用 constexpr 版本
21
22     return 0;
```

```
}
```

上述之所以行得通是因为 constexpr 函数需要常数表达式作为入参 – 因此如果我们使用 constexpr 函数的返回值作为 consteval 函数的入参，constexpr 函数就必须在编译期执行了！
注意 consteval 函数返回的是值，而这可能在运行时效率不高（如果值是一些拷贝起来很昂贵的类型，例如 std::string），在编译期的上下文中倒是无所谓因为 consteval 函数的整个调用过程最终会由计算后的返回值替换掉。
相关内容：

- 章节 8.8 中讲解 `auto` 。
- 章节 8.15 中讲解 abbreviated 函数模版（`auto` 参数）。

6.15 Unnamed and inline namespaces

C++ 支持两种变体的命名空间值得了解。

无名（匿名）命名空间

一个**无名命名空间**（也被称为**匿名命名空间**）是一种没有定义名称的命名空间，例如：

```
#include <iostream>

namespace // 无名命名空间
{
    void doSomething() // 只可以在本文件中访问
    {
        std::cout << "v1\n";
    }
}

int main()
{
    doSomething(); // 可以在不使用命名空间前缀的情况下调用 doSomething()

    return 0;
}
```

在 `unnamed namespace` 中声明的所有内容都将被视为其父命名空间的一部分。因此，即使 `doSomething` 函数定义在 `unnamed namespace` 中，函数本身是可以在父命名空间中方芬（在本案例中即 `global namespace` ），这也是为何可以在 `main` 里调用 `doSomething` 而不需要任何限定符。
虽然看上去这里的 `unnamed namespace` 没什么用，但是其中的所有标识符都可以被视为带有 `internal linkage` ，这就意味着对于文件外而言它们是不可见的。

对于函数而言，无名命名空间的作用等同于 `static functions` ，即：

```
#include <iostream>

static void doSomething() // 仅在此文件中可以被访问
{
    std::cout << "v1\n";
}

int main()
{
    doSomething(); // 可以不使用命名空间前缀调用 doSomething()

    return 0;
}
```

`unnamed namespace` 通常而言用于大量内容需要确保定义在指定文件中，这样简化了需要对内容中的每一条加上 `static` 声明。`unnamed namespace` 同样维护 `user-defined types`（之后章节将覆盖）在本地文件中，这是没有其他别的代替方案可以做到的。

内联命名空间

内联命名空间通常用作于放置版本的内容信息。类似于 `unnamed namespace` ，所有定义在 `inline namespace` 中的内容视为父命名空间的一部分。然而 `inline namespace` 不会提供 `internal linkage` 。

只需要 `inline` 关键字就可以定义内联命名空间：

```
#include <iostream>

inline namespace v1 // 定义一个名为 v1 的内联命名空间
{
    void doSomething()
    {
        std::cout << "v1\n";
    }
}

namespace v2 // 定义一个名为 v2 的命名空间
{
    void doSomething()
    {
        std::cout << "v2\n";
    }
}

int main()
{
    v1::doSomething(); // 调用 v1 的 doSomething()
```

```
22      v2::doSomething(); // 调用 v2 的 doSomething()
23
24      doSomething(); // 调用内联命名空间的 doSomething(), 即 v1
25
26      return 0;
27  }
```

下次在修改版本信息时，可以直接去掉 v1 的 `inline` 并在 v2 命名空间前加上，这样使得 `doSomething()` 直接调用的是 v2 版本。

7 Control Flow and Error Handling

7.1 Control flow introduction

流控制声明的分类

Category	Meaning	Implemented in C++ by
Conditional statements	Conditional statements cause a sequence of code to execute only if some condition is met.	If, switch
Jumps	Jumps tell the CPU to start executing the statements at some other location.	Goto, break, continue
Function	calls Function calls are jumps to some other location and back.	Function calls, return
Loops	Loops tell the program to repeatedly execute some sequence of code zero or more times, until some condition is met.	While, do-while, for, ranged-for
Halts	Halts tell the program to quit running.	std::exit(), std::abort()
Exceptions	Exceptions are a special kind of flow control structure designed for error handling.	Try, throw, catch

8 Type Conversion and Function Overloading

8.5 Explicit type conversion (casting) and static_cast

8.1 中讨论了编译器可以隐式转换值类型通过一个 `implicit type conversion` 系统。当用户希望提升一个值类型至更宽泛的类型，使用隐式转换是可以的。
很多新手尝试这样做：

```
double d = 10 /4; // 整数除法，d 初始值为 2.0
```

因为 `10` 和 `4` 的类型皆为 `int` ，整数除法被执行，表达式计算 `int` 值为 `2` 。该值再经历数值转换成 `double` 值 `2.0` 。这样很可能不是符合预期。
字面值的情况下用户可以手动改为带有小数的值，然而如果是变量，如下：

```
int x { 10 };
int y { 4 };
double d = x / y; // 结果还是 2.0
```

好在 C++ 有不同的**类型转换操作符** type casting operators（通常被称为 **casts**）使用户可以请求编译器进行类型转换。因为转换是用户显式的请求，这些类型转换通常被称为**显式类型转换** explicit type conversion（与隐式类型转换相对）。

类型 casting

C++ 提供五中类型的 casts：`C-style casts` ，`static casts` ，`const casts` ，`dynamic casts` 以及 `reinterpret casts` 。后面的四种方式有时被称为 **named casts**。
相关内容：18.10 中讲解 dynamic casts。
警告：除非有很好的理由，否则不要用 const casts 以及 reinterpret casts。

C-style casts

在标准 C 编程里，casts 是通过 () 操作符，需要转换的类型放置在圆括号中。

```
#include <iostream>

int main()
{
    int x { 10 };
    int y { 4 };


    double d { (double)x / y }; // 转换 x 为 double 使其可以做除法
    std::cout << d; // 打印 2.5

    return 0;
}
```

尽管 `C-style cast` 看起来是单 cast，在不同的上下文中仍然会产生不同结果。它可以包含 `static casts` ，`const casts` 或者 `reinterpret casts` （后两者之前提过是要尽量避免的）。因此 `C-style cast` 会有使用不当的风险并产生非预期的行为。
最佳实践：避免使用 `C-style cast` 。

static_cast

C++ 引入了一个 casting 操作符称为 **static_cast**，可以被用于值转换。

```cpp
#include <iostream>

int main()
{
    int x { 10 };
    int y { 4 };

    // static cast x 至 double 类型用于除法计算
    double d { static_cast<double>(x) / y };
    std::cout << d; // 打印 2.5

    return 0;
}
```

最佳实践：推荐 static_cast。

8.6 Typedefs and type aliases

Type 别名

C++ 中，**using** 是一个用于对现有数据类型创建别名的关键词。

```cpp
using Distance = double; // 定义 Distance 实则为 double 类型

Distance milesToDestination{ 3.4 }; // 定义变量
```

Naming type aliases

历史原因，没有关于类型如何命名的一致性。通常有三种常见的命名转换：

- 类型别名带有“_t”后缀（“type”的缩写）。这类转换通常用于标准库的全局域类型名称（例如 `size_t` 以及 `nullptr_t` ）。这类转换有 C 继承而来，在用户自定义类型（有时别的类型）别名是尤为常见，但是这不太符合现代 C++ 的习惯。
- 类型别名带有“_type”后缀。这类转换用于一些标准库类型（例如 `std::string` ）用于嵌套类型的别名（例如 `std::string::size_type` ）。但是还有一些嵌套类型完全没有后缀（例如 `std::string::iterator` ），因此这种用法最好是非连贯的。

- 不带后缀的类型别名。现代 C++ 中使用无后缀。大写字母可以帮助从变量以及函数名称（小写字母开头）中辨别类型名称，并防止它们命名冲突。

```
void printDistance(Distance distance); // Distance 在别处被定义
```

最佳实践：自定义类型使用大写的方式并且不带后缀（除非有非常特别的原因）。

类型别名的作用域

类型别名标识符与变量标识符的作用域相同。如果需要跨文件使用，可以在头文件中定义：

```
#ifndef MYTYPES
#define MYTYPES

    using Miles = long;
    using Speed = long;

#endif
```

Typedefs

typedef（“type definition”的缩写）是创建类型别名的旧方法。

```
// The following aliases are identical
typedef long Miles;
using Miles = long;
```

因为向后兼容的原因 Typedefs 仍然在 C++ 中存在，不过在现代 C++ 中，它们已经大量的被类型别名给替换了。
最佳实践：使用类型别名而不是 typedefs。

8.9 Introduction to function overloading

函数重载简介

函数重载 function overloading 允许用户创建若干同名函数，只要每个标识符命名的函数拥有不同的类型（或者可以以别的方式差异化函数）。每个函数共享一个名称（在同作用域下）被称为**重载函数**。

```
int add(int x, int y) // 整型版本
{
    return x + y;
}

double add(double x, double y) // double 型版本
{
```

```
8       return x + y;
9   }
10
11  int main()
12  {
13      return 0;
14  }
```

重载解析简介

另外当一个重载函数被调用，编译器则会尝试根据入参的差异匹配函数调用至正确的函数。这被称为**重载解析** overload resolution。

```
1   #include <iostream>
2
3   int add(int x, int y)
4   {
5       return x + y;
6   }
7
8   double add(double x, double y)
9   {
10      return x + y;
11  }
12
13  int main()
14  {
15      std::cout << add(1, 2); // 调用 add(int, int)
16      std::cout << '\n';
17      std::cout << add(1.2, 3.4); // 调用 add(double, double)
18
19      return 0;
20  }
```

8.10 Function overload differentiation

重载还是是如何区分的

Function property	Used for differentiation	Notes
Number of parameters	Yes	
Type of parameters	Yes	Excludes typedefs, type aliases, and const qualifier on value parameters. Includes ellipses.
Return type	No	

对于成员函数，额外的函数等级限定符也会被考虑到：

Function-level qualifier	Used for overloading
const or volatile	Yes
Ref-qualifiers	Yes

根据入参数的重载

```
int add(int x, int y)
{
    return x + y;
}

int add(int x, int y, int z)
{
    return x + y + z;
}
```

根据参数类型的重载

```
int add(int x, int y); // 整数
double add(double x, double y); // 浮点
double add(int x, double y); // 混合
double add(double x, int y); // 混合
```

暂时还没有覆盖到省略号的内容，但是省略号参数被视为唯一类型的参数：

```
void foo(int x, int y);
void foo(int x, ...); // 与 foo(int, int) 不同
```

函数返回值类型的不同不被认为是有区分

```
int getRandomValue();
double getRandomValue();
```

上述会导致编译器报错。

8.11 Function overload resolution and ambiguous matches

```
#include <iostream>

void print(int x)
{
    std::cout << x << '\n';
}

void print(double d)
{
    std::cout << d << '\n';
}

int main()
```

```
{
    print('a'); // char 不匹配 int 或 double
    print(5L); // long 不匹配 int 或 double

    return 0;
}
```

而 char 或 long 可以被隐式类型转换成 int 或 double 。那么它们分别用到哪种转换呢?

解析重载函数调用

如果一个函数调用的是重载函数，编译器会经过以下一系列规则来决定使用最为匹配的函数。每个步骤中，编译器会对入参应用不同类型的类型转换。每次转换被应用时，编译器会检查是否有相匹配的重载函数。在所有类型的转换被应用后以及所有的匹配被检查后，这个过程就结束了。这样会得到以下三种结果中的一个：

- 没有匹配的函数被找到。编译器移动至下一个步骤。
- 单个匹配的函数被找到。该函数被认为是最好的匹配。匹配过程现在完成嘞，剩下的步骤不再执行。
- 多个匹配函数被找到。编译器会报匹配有歧义的错误。之后会讨论这种结果。

参数匹配序列

第一步，编译器尝试找到最匹配的。其中分为两个阶段。首先，编译器查找是否有重载函数，检查调用者的入参类型是否严格匹配重载函数的入参类型。
编译器将一些 trivial 的转换应用于函数调用的参数。**trivial conversions** 是一系列固定的转换规则用于修改类型（而不修改值本身）达到找到匹配的目的。

```
void print(const int)
{
}

void print(double)
{
}

int main()
{
    int x { 0 };
    print(x); // x trivially 转换成 const int

    return 0;
}
```

进阶：转换一个非引用类型至一个引用类型（或者相反）同样也是 trivial conversion。
通过 trivial conversion 的匹配被视为精确匹配。
第二步，如果没有确切匹配被找到，编译器则会尝试应用数值提升 numeric promotion 来匹配入参。

```
void print(int)
{
}

void print(double)
{
}

int main()
{
    print('a'); // 提升用于匹配 print(int)
    print(true); // 提升用于匹配 print(int)
    print(4.5f); // 提升用于匹配 print(double)

    return 0;
}
```

第三步，如果通过数值提升 numeric promotion 没有匹配上，编译器则会尝试对入参应用数值转换 numeric conversions。

```
#include <string> // std::string

void print(double)
{
}

void print(std::string)
{
}

int main()
{
    print('a'); // 'a' 转换后匹配 print(double)

    return 0;
}
```

上述例子中，因为没有 `print(char)`（精准匹配），也没有 `print(int)`（promotion 匹配），`'a'` 则被数值转换为 double 进而匹配 `print(double)`。
重点：应用数值提升 numeric promotions 的匹配优先于任何应用数值转换 numeric conversions 的匹配。
第四步，如果通过数值转化 numeric conversion 也没有匹配上，编译器会尝试寻找用户自定义

的转换。尽管暂时还没有讲到用户自定义转换，某些类型（例如，类）可以定义隐式转换成其他类型。

```
// 暂时还没有讲到类
class X // 这里定义了名为 X 的类型
{
public:
    operator int() { return 0; } // 这里是 X 到 int 的用户自定义转换
};

void print(int)
{
}

void print(double)
{
}

int main()
{
    X x; // 这里创建类型为 X (名为 x) 的对象
    print(x); // x 通过用户自定义转换，被转换成 int 类型

    return 0;
}
```

相关内容：14.11 重载类型转换 overloading typecasts。
第五步，如果用户自定义转换没有匹配上，编译器则会匹配使用省略号的函数。
相关内容：12.6 省略号（以及为什么要避免它）。
第六步，如果上述都没有匹配上，编译器则会报无法找到匹配函数的错误。

模棱两可的匹配

在没有重载函数的情况下，每个函数调用会找到该函数，或者匹配失败同时报编译器错误。
在有重载函数的情况下，则会有第三种可能发生：`ambiguous match` 可能被找到。**模棱两可匹配** ambiguous match 出现在编译期找到两个或以上的函数可以同时匹配。这时编译器会停止匹配并且报模棱两可函数调用的错误。
例如：

```
void print(int x)
{
}

void print(double d)
{
}
```

```
8
9  int main()
10 {
11     print(5L); // 5L 是 long 类型
12
13     return 0;
14 }
```

另一个模棱两可匹配:

```
1  void print(unsigned int x)
2  {
3  }
4
5  void print(float y)
6  {
7  }
8
9  int main()
10 {
11     print(0); // int 可以被数值转换成 unsigned int 或者 float
12     print(3.14159); // double 可以被数值转换为 unsigned int 或者 float
13
14     return 0;
15 }
```

解决模棱两可匹配

由于模棱两可匹配是编译期错误，因此在编译程序前需要消除歧义。这里有几种方法:

1. 总是定义简单的重载函数，其入参类型是调用函数的类型。
2. 或者是显式转换歧义入参成为函数定义的类型。
3. 如果参数是字面值，使用后缀来确保字面值是正确的类型。

8.13 Function templates

C++ 模版简介

C++ 中，模版系统用于简化创建可以用作于不同数据类型的函数（或类）的过程。
模版 template 描述函数或者类的样子。不同于普通的定义（即所有类型需要被定义），模版中可以使用多个占位符类型。占位符类型代表着一些类型在编写模版时暂时未知，但是在之后会被提供。
一旦模版被定义，编译器可以使用模版来生成所需要的若干重载函数（或类），每个都使用不同的真实类型!

最后得到了几乎所有的函数或类（每组都有不同的类型），而仅仅只需创建和维护单个模版，编译器则做了其他所有的活。
重点：编译器可以使用单个模版生成一组相关函数或类，每个使用一组不同的类型。
重点：在编写模版时，其可以作用于暂时还不存在的类型。这使得模版代码兼具了灵活性以及未来的扩展性。

函数模板

函数模版是类函数定义的，使用于生成一个或多个重载函数，并带有不同真实类型的集合。也就是说可以让用户创建函数可以作用于不同的类型。
当创建函数模版时，使用占位符类型（同样也被称为**模版类型** template types）来对应任何参数类型，返回类型，或是函数体中的类型。
最佳实践：使用单个大写字母（起始于 T）用于命名模版类型（例如 T，U，V，等等）。
模版参数声明：

```
template <typename T> // 模版参数声明在此
T max(T x, T y) // 函数模版定义 max<T>
{
    return (x > y) ? x : y;
}
```

8.14 Function template instantiation

使用函数模版

```
#include <iostream>

template <typename T>
T max(T x, T y)
{
    return (x > y) ? x : y;
}

int main()
{
    std::cout << max<int>(1, 2) << '\n'; // 实例化并调用函数 max<int>(int, int)

    return 0;
}
```

由函数模版（带模版类型）创建函数（带指定类型）的过程称为**函数模版实例化** function template instantiation（或简称**实例化** instantiation）。当这个过程在函数调用时发生，被称为**隐式实例**

化 implicit instantiation。实例化函数通常也被成为**函数实例** function instance（或简称**实例** instance）或者**模版函数** template function。函数实例在任何层面都是普通函数。

```cpp
#include <iostream>

// 函数模版的声明（不需要更多的定义了）
template <typename T>
T max(T x, T y);

template<>
int max<int>(int x, int y) // 生成的函数 max<int>(int, int)
{
    return (x > y) ? x : y;
}

int main()
{
    std::cout << max<int>(1, 2) << '\n'; // 实例化 max<int>(int, int) 并调用

    return 0;
}
```

另一个例子：

```cpp
#include <iostream>

template <typename T>
T max(T x, T y) // max(T, T) 的函数模版
{
    return (x > y) ? x : y;
}

int main()
{
    std::cout << max<int>(1, 2) << '\n';    // 实例化并调用 max<int>(int, int)
    std::cout << max<int>(4, 3) << '\n';    // 调用已经实例化过的 max<int>(int, int)
    std::cout << max<double>(1, 2) << '\n'; // 实例化并调用 max<double>(double, double)

    return 0;
}
```

模版参数推导

```cpp
#include <iostream>

template <typename T>
T max(T x, T y)
{
    std::cout << "called max<int>(int, int)\n";
```

```
    return (x > y) ? x : y;
}

int max(int x, int y)
{
    std::cout << "called max(int, int)\n";
    return (x > y) ? x : y;
}

int main()
{
    std::cout << max<int>(1, 2) << '\n'; // 选择 max<int>(int, int)
    std::cout << max<>(1, 2) << '\n';    // 推导 max<int>(int, int) （非模版函数不被考虑）
    std::cout << max(1, 2) << '\n';      // 调用函数 max(int, int)

    return 0;
}
```

最佳实践：使用函数模版时更推荐普通函数调用的语法。

在多文件中使用函数模版

Max.h:

```
#ifndef MAX_H
#define MAX_H

template <typename T>
T max(T x, T y)
{
    return (x > y) ? x : y;
}

#endif
```

Foo.cpp:

```
#include "Max.h" // 调用 max<T, T>() 的模版定义
#include <iostream>

void foo()
{
  std::cout << max(3, 2) << '\n';
}
```

main.cpp:

```
#include "Max.h" // 引入 max<T, T>() 的模版定义
#include <iostream>

```

```
4  void foo(); // 函数 foo 的前向声明
5
6  int main()
7  {
8      std::cout << max(3, 5) << '\n';
9      foo();
10
11     return 0;
12 }
```

泛型编程

因为模版类型可以被任何实际类型替换，模版类型有时也被成为**泛型类型**。同样因为模版可以无关特定类型，编写模版有时也被成为**泛型编程**。

8.15 Function templates with multiple template types

使用 static_cast 转换参数来匹配类型

```
1  #include <iostream>
2
3  template <typename T>
4  T max(T x, T y)
5  {
6      return (x > y) ? x : y;
7  }
8
9  int main()
10 {
11     std::cout << max(static_cast<double>(2), 3.5) << '\n'; // 转换 int 至 double 用以调用 max(
       double, double)
12
13     return 0;
14 }
```

提供实际类型

```
1  #include <iostream>
2
3  template <typename T>
4  T max(T x, T y)
5  {
6      return (x > y) ? x : y;
7  }
8
```

```
int main()
{
    std::cout << max<double>(2, 3.5) << '\n'; // 提供了实际类型 double，编译器不再使用模版类型推导

    return 0;
}
```

带有若干模版类型参数的函数模版

```
#include <iostream>

template <typename T, typename U> // 使用两个模版类型参数 T 和 U
T max(T x, U y) // x 为 T 类型， y 为 U 类型
{
    return (x > y) ? x : y; // 这里出现了缩窄变化的问题
}

int main()
{
    std::cout << max(2, 3.5) << '\n';

    return 0;
}
```

```
#include <iostream>

template <typename T, typename U>
auto max(T x, U y)
{
    return (x > y) ? x : y;
}

int main()
{
    std::cout << max(2, 3.5) << '\n';

    return 0;
}
```

简化函数模版 C++20

C++20 引入关键字 `auto` 新用法：当 `auto` 关键字用于普通函数的参数类型，编译器会自动转换函数成为函数模版，其 auto 参数变为独立模版类型参数。这个方法创建的函数模版被称为 ** **简化函数模版** **。
例如，在 C++20 中简写：

```
auto max(auto x, auto y)
{
    return (x > y) ? x : y;
}
```

代替之间的：

```
template <typename T, typename U>
auto max(T x, U y)
{
    return (x > y) ? x : y;
}
```

如果想要每个模版类型参数作为独立类型，推荐这种去除模版参数声明的形式，使得代码更简洁已读。

最佳实践：如果每个 auto 参数都需要独立的模版类型时，请使用简写函数模版（在语言标准设为 C++20 或更新的情况下）。

9 Compound Types: References and Pointers

9.1 Introduction to compound data types

组合数据类型

组合数据类型（有时也被成为**复合数据类型** composite data types）是一种由基础数据类型（或者其他组合类型）构成的数据类型。每种组合数据类型都拥有其唯一的属性。
C++ 支持下列组合数据类型：

- Functions
- Arrays
- Pointer types:
  - Pointer to object
  - Pointer to function
- Pointer to member types:
  - Pointer to data member
  - Pointer to member function
- Reference types:
  - L-value references
  - R-value references
- Enumerated types:
  - Unscoped enumerations
  - Scoped enumerations
- Class types:
  - Structs
  - Classes
  - Unions

9.2 Value categories (lvalues and rvalues)

表达式的属性

为了帮助判断表达式如何被计算以及可以在哪里被使用，C++ 中所有的表达式都有两个属性：类型和值类别。

表达式的类型

```cpp
#include <iostream>

int main()
{
    auto v1 { 12 / 4 }; // int / int => int
    auto v2 { 12.0 / 4 }; // double / int => double

    return 0;
}
```

注意表达式的类型必须可以在编译期被决定（否则类型检查与类型推导不能工作）– 然而，表达式的值既可以在编译期（如果表达式为 constexpr）或是运行时（非 constexpr）。

表达式的值类别

表达式（或子表达式）的**值类别**表明一个表达式是否解析为一个值，一个函数，或者其他类型的对象。
C++11 之前，只有两种值类别：`lvalue` 和 `rvalue` 。
C++11 中，添加了三种新的值类别（`glvalue` ，`prvalue` 和 `xvalue` ）用于支持名为 `move sematics` 的新特性。

左值与右值表达式

lvalue（发音 ell-value，为 left value 或 locator value 的缩写，有时写为 l-value），是计算为可识别的对象或者函数（或 bit-field）的表达式。
C++ 标准中的“identity”并没有被很好的定义。实体 entity（例如对象或函数）拥有 identity 可以从其他类似的实体中区分出来（通常的做法是比较实体的地址）。
带有 identities 的实体可以通过标识符，引用，或者指针来进行访问，同时相较于单个表达式或声明而言，它们通常都拥有较长的生命周期。

```cpp
#include <iostream>

int main()
{
```

```
    int x { 5 };
    int y { x }; // x 为左值表达式

    return 0;
}
```

由于常数被引用进语言中，左值则有两种子类型：**可变 lvalue** 以及**不可变 lvalue**。

```
#include <iostream>

int main()
{
    int x{};
    const double d{};

    int y { x }; // x 为可变左值表达式
    const double e { d }; // d 为不可变左值表达式

    return 0;
}
```

rvalue（发音 arr-value，为 right value 的缩写，有时写为 r-value）是非 l-value 的表达式。常见的 rvalues 包括字面值（除了 C-style 的字符串字面值，它是 lvalues）以及函数和操作符的返回值。rvalues 不能被区分（意味着它们必须立刻被使用掉），同时仅存在于其表达式所在的作用域。

```
#include <iostream>

int return5()
{
    return 5;
}

int main()
{
    int x{ 5 }; // 5 为右值表达式
    const double d{ 1.2 }; // 1.2 为右值表达式

    int y { x }; // x 为可变左值表达式
    const double e { d }; // d 为不可变左值表达式
    int z { return5() }; // return5() 为右值表达式（因为其返回值是由值返回的）

    int w { x + 1 }; // x + 1 为右值表达式
    int q { static_cast<int>(d) }; // 转换 d 为 int 是右值表达式

    return 0;
}
```

关联内容：所有的左值和右值表达式可以在这里找到。

左值转换至右值

```
int main()
{
    int x{ 1 };
    int y{ 2 };

    x = y; // y 为可变左值而不是右值，这是合法的

    return 0;
}
```

这里合法是因为左值被隐式转换成右值，因此左值可以被使用在右值所需要在的地方。
考虑以下代码：

```
int main()
{
    int x { 2 };

    x = x + 1;

    return 0;
}
```

这里变量 `x` 被用作于两个不同的上下文中。在分配符左侧，`x` 是左值表达式用于计算变量 x。在分配符右侧，`x + 1` 为右值表达式用于计算出值 `3` 。
现在覆盖了左值的概念，接下来学习第一个组合类型：`lvalue reference` 。
重要概念：判断左值与右值的经验法则：左值表达式是计算变量或其他可辨识对象，它们的存在超出表达式范围；右值表达式计算字面值或函数与操作符的返回值，它们在表达式结束后舍弃。

9.3 lvalue references

C++ 中**引用** references 是已有对象的别称。一旦一个引用被定义了，任何应用于引用的操作则会应用到被引用的对象其本身。
这就意味着用户可以使用引用来读取或者修改被引用的对象。尽管引用刚开始看起来有点笨拙无用或者多余的，引用却在 C++ 中的任何地方都有用到。
也可以对函数创建引用，尽管这不常见。
现代 C++ 包含两种类型的引用：`lvalue references` 以及 `rvalue references` 。本章讲解前者。
相关内容：Rvalue references 则会在 `move semantics` 章节中讲到。

左值引用类型

lvalue 引用（通常称为 **引用**因为 C++11 之前只有一种引用）充当已有对象（例如变量）的别名。

```
int      // 一个普通的 int 类型
int&     // 一个对于 int 对象的左值引用
double&  // 一个对于 double 对象的左值引用
```

左值引用变量

使用左值引用类型，用户可以创建一个左值引用变量。**左值引用变量**是一种充当左值（通常是别的变量）引用的变量。
创建一个左值引用变量，只需要简单的定义一个带有左值引用类型的变量：

```
#include <iostream>

int main()
{
    int x { 5 };    // x 是一个普通的整数变量
    int& ref { x }; // ref 是一个左值引用变量，现在用作于 x 变量的别名

    std::cout << x << '\n';  // 打印 x (5)
    std::cout << ref << '\n'; // 打印 x 通过 ref (5)

    return 0;
}
```

上述例子中，`int&` 类型定义 `ref` 作为 int 的左值引用，接着初始化左值表达式 `x`。这样 `ref` 和 `x` 可以同义的被使用。
在编译器的角度，无论 & 符号是放在类型名称上（`int& ref` 还是变量名称上（`int &ref` 都没有差别，选择何种写法与风格有关。现代 C++ 程序员更加偏向前者，即 & 与类型绑定，因为这样可以更清楚知道引用是类型部分的信息，而不是标识符的信息。
最佳实践：定义引用时将 & 符号放在类型边上（而不是引用变量名字）。
进阶：对于已经知道指针的用户而言，& 符号在这个上下文中不代表“什么的地址”，而是“什么的左值引用”。

通过左值引用修改值

```
#include <iostream>

int main()
{
    int x { 5 }; // 普通整数变量
```

```
    int& ref { x }; // ref 是左值引用变量，现在用作于于变量 x 的别名

    std::cout << x << ref << '\n'; // 打印 55

    x = 6; // x 现在为 6

    std::cout << x << ref << '\n'; // 打印 66

    ref = 7; // 被引用的对象 (x) 现在为 7

    std::cout << x << ref << '\n'; // 打印 77

    return 0;
}
```

上述例子中，`ref` 为 `x` 的别名，因此可以通过 `x` 或 `ref` 修改 `x` 的值。

左值引用的初始化

与常量相同，所有引用都必须被初始化。
当一个引用被初始化了一个对象（或函数），可以说其**绑定**了该对象（或函数）。这个绑定的过程被称为**引用绑定**。被引用的对象（或函数）有时被称为**指示物** referent。
左值引用必须绑定至一个可变左值。

```
int main()
{
    int x { 5 };
    int& ref { x }; // 可行：左值引用绑定至一个可变的左值

    const int y { 5 };
    int& invalidRef { y };  // 不可行： int 引用不可被绑定至 double 变量
    int& invalidRef2 { 0 }; // 不可行： double 引用不可被绑定至 int 变量

    return 0;
}
```

`void` 的左值引用是不被允许的（它指向哪里?）。

引用不可以被再复位 reseated（改变引用的对象）

C++ 中的引用一旦初始化后不可以**再复位**，意为不可以改变其引用的对象。
新手 C++ 程序员经常通过赋值让引用指向另一个变量从而 reseat 引用。这样可以被编译和运行 – 只不过跟预期不一样。考虑以下程序：

```
#include <iostream>

int main()
```

```
{
    int x { 5 };
    int y { 6 };

    int& ref { x }; // ref 现在为 x 的别名

    ref = y; // 赋值 6（y 的值）至 x（被 ref 引用）
    // 上述代码并不会改变 ref 的引用至变量 y ！！！

    std::cout << x << '\n'; // 用户预期这里打印 5，实际上打印 6

    return 0;
}
```

当一个引用在表达式中计算，它处理的是其引用的对象。所以 `ref = y` 不会改变 `ref` 引用到 `y`。相反，因为 `ref` 是 `x` 的别名，表达式则视为 `x = y` 的计算，因为 `y` 表示的是值 `6`，`x` 则被赋值了 `6`。

左值引用的作用域与持续性

与普通变量规则相同：

```
#include <iostream>

int main()
{
    int x { 5 }; // 普通整数
    int& ref { x }; // 变量的引用

     return 0;
} // x 和 ref 在此消亡
```

引用与被引用者拥有独立的生命周期

除了一种情况（下一章详解）以外，引用的生命周期与其被引用的生命周期是独立的。换言之，下面两种情况都是对的：

- 在对象被引用前，引用可以被销毁。
- 被引用的对象可以在被引用时被销毁。

悬垂引用

当一个被引用的对象在被引用之前销毁了，那么引用指向的对象不再存在，这种引用被成为**悬垂引用**。访问悬垂引用会导致未被定义的行为。
悬垂引用很容易避免，不过之后的章节中会讲到在实际代码中何时容易出现。

引用非对象

在 C++ 中引用不是对象。一个引用不需要存在或者占用存储。如果可能的话，编译器将会通过替换被引用者至引用者来优化所有的引用。然而这也不总是可能的，因为如果可能，引用则会需要存储。
这也意为着“引用变量”这个词并不准确，因为变量都是带有名称的对象，而引用并不是对象。

9.4 Lvalue references to const

上一章节中讨论了左值引用只可以绑定到可变的左值上。这就意味着下面代码不合法：

```
int main()
{
    const int x { 5 }; // x 是一个不可变的 (const) 左值
    int& ref { x }; // 错误: ref 不可以绑定不可变左值

    return 0;
}
```

const 的左值引用

定义左值引用时使用 `const` 关键字，即告诉左值引用其引用的对象为 const。这种引用被称为 **const** 值的左值引用（有时也称为 **reference to const** 或者 **const reference**）。

```
int main()
{
    const int x { 5 };     // x 为不可变左值
    const int& ref { x }; // 正确: ref 是 const 值的左值引用

    return 0;
}
```

因为 const 左值引用视为 const 的引用，它们可以被访问但是不可以被修改：

```
#include <iostream>

int main()
{
    const int x { 5 };     // x 为不可变左值
    const int& ref { x }; // 正确: ref 是 const 值的左值引用

    std::cout << ref << '\n'; // 正确: 可以访问 const 对象
    ref = 6;                  // 错误: 不可以修改 const 对象

    return 0;
}
```

通过可变左值初始化 const 左值引用

const 左值引用可以绑定可变左值。这种情况下，当通过引用访问时，被引用的对象视为 const（尽管其根本的对象为非 const）：

```
#include <iostream>

int main()
{
    int x { 5 };              // x 为可变左值
    const int& ref { x };     // 正确：可以绑定可变左值给 const 引用

    std::cout << ref << '\n'; // 正确：可以通过 const 引用访问对象
    ref = 7;                  // 错误：不可以通过 const 引用修改对象

    x = 6;                    // 正确：x 为可变左值，仍然可以通过原有的标识符进行修改

    return 0;
}
```

最佳实践：更加推荐 `lvalue references to const` 而不是 `lvalue references to non-const` 除非需要通过引用来修改对象。

通过右值初始化 const 左值引用

const 左值引用也可以绑定右值：

```
#include <iostream>

int main()
{
    const int& ref { 5 }; // 可行： 5 为右值

    std::cout << ref << '\n'; // 打印 5

    return 0;
}
```

上述代码中，一个临时的对象被创建后用右值进行了初始化，const 引用绑定临时对象。
临时对象（有时也被称为**匿名对象**）是一个为临时使用（接着销毁）而在单个表达式中创建的对象。临时对象完全没有作用域（合理，因为域是标识符的属性，而临时对象没有标识符）。这就意味着临时对象仅可以在其创建的地方使用，因为没有其他别的办法指向它。

const 引用绑定临时对象延长了临时对象的生命周期

临时对象通常在其被创建的表达式结束时被销毁。

然而，考虑一下上述例子中如果临时对象创建时其右值 `5` 在初始化 `ref` 的表达式结束时被销毁。`ref` 引用将会变为左悬垂（引用的对象被销毁），这时尝试访问 `ref` 则会得到未定义行为。

为了避免这样悬垂引用的情况，C++ 有一个特别的规则：当 const 左值引用绑定了临时对象，临时对象的声明周期被延长到与引用的生命周期相匹配。

```
#include <iostream>

int main()
{
    const int& ref { 5 }; // 临时对象的生命周期被延长与 ref 一致

    std::cout << ref << '\n'; // 因此可以在此处安全使用

    return 0;
} // ref 和临时对象在此同时被销毁
```

重点：左值引用仅可以绑定可变左值。const 左值引用可以绑定可变左值，不可变左值，以及右值。这使得它们成为更加灵活的引用类型。

那么为什么 C++ 运行 const 引用绑定右值呢？请看下一章节。

9.5 Pass by lvalue reference

上一章介绍了左值引用以及 const 左值引用。单独来看它们似乎不是很有用 – 当可以使用变量本身的时候为什么还给变量创建别名呢？

首先在一些上下文中，函数参数是被拷贝成为函数的入参：

```
#include <iostream>

void printValue(int y)
{
    std::cout << y << '\n';
} // y 在此销毁

int main()
{
    int x { 2 };

    printValue(x); // x 被值传递（拷贝）进入参数 y （廉价）

    return 0;
}
```

一些对象的拷贝很昂贵

大多数由标准库提供的类型（例如 `std::string`）为 `class types`。类类型的拷贝通常是很昂贵的。可能的话，应该避免不必要的对象拷贝，特别是拷贝之后又被立刻销毁。
考虑以下代码：

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

void printValue(std::string y)
{
    std::cout << y << '\n';
} // y 在此销毁

int main()
{
    std::string x { "Hello, world!" }; // x 为 std::string

    printValue(x); // x 被值传递（拷贝）进入参数 y（昂贵）

    return 0;
}
```

引用传递

调用函数时，避免昂贵拷贝的一种方式是使用 `pass by reference`。当使用 ** 传递引用 ** 时，需要定义函数的入参为引用类型（或者 const 引用类型）而不是普通类型。当函数被调用时，每个引用入参绑定合理的参数。因为引用充当参数的别名，那么就不再需要进行参数拷贝。

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

void printValue(std::string& y) // 类型改为 std::string&
{
    std::cout << y << '\n';
} // y 在此销毁

int main()
{
    std::string x { "Hello, world!" };

    printValue(x); // x 被引用传递进入参数 y（廉价）

    return 0;
}
```

引用传递进行参数的值修改

```cpp
#include <iostream>

void addOne(int y) // y 为 x 的拷贝
{
    ++y; // 这里修改 x 的拷贝，而不是 x 对象
}

int main()
{
    int x { 5 };

    std::cout << "value = " << x << '\n';

    addOne(x);

    std::cout << "value = " << x << '\n'; // x 没有被改变

    return 0;
}
```

```cpp
#include <iostream>

void addOne(int& y) // y 绑定对象 x
{
    ++y; // 这里修改对象 x
}

int main()
{
    int x { 5 };

    std::cout << "value = " << x << '\n';

    addOne(x);

    std::cout << "value = " << x << '\n'; // x 被改变了

    return 0;
}
```

非 const 的引用传递仅接受可变左值引用

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

void printValue(int& y) // y 进接受可变左值
```

```
{
    std::cout << y << '\n';
}

int main()
{
    int x { 5 };
    printValue(x); // 正确：x 为可变左值

    const int z { 5 };
    printValue(z); // 错误：z 为不可变左值
    printValue(5); // 错误：5 为右值

    return 0;
}
```

传递 const 引用提供同样的便利（避免参数拷贝），同时也保证函数 * 不会 * 改变其引用的值。

```
void addOne(const int& ref)
{
    ++ref; // 不被允许：ref 是 const
}
```

最佳实践：推荐传递 const 引用而不是非 const 引用，除非有特别的理由需要做额外的事情（例如函数需要修改入参的值）。

混合值传递与引用传递

```
#include <string>

void foo(int a, int& b, const std::string& c)
{
}

int main()
{
    int x { 5 };
    const std::string s { "Hello, world!" };

    foo(5, x, s);

    return 0;
}
```

何时引用传递

最佳实践：基础类型使用值传递，类（或结构体）类型使用 const 引用传递。

值传递与引用传递的开销

不是所有的类类型都需要引用传递。
有两个关键点可以帮助我们理解何时需要值传递 vs 引用传递：
首先，拷贝对象的开销普遍与两点成正比：

- 对象的大小。对象使用越多的内存就需要更长的时间拷贝。
- 任何其它的准备开销。一些类类型需要额外准备的当它们被实例化时（例如打开文件或者数据库，或者分配特定的动态内存用以存储对象中的变量）。这些准备开销必须在每个对象拷贝时花费时间。

另一方面，为对象绑定引用总是迅速的（与拷贝基础类型的速度相同）。
其次，通过引用访问对象对比直接访问对象会稍微昂贵一些。通过变量的标识符，编译器可以直接获取分配给变量的内存地址并访问值。通过引用时，通常会有额外的一步：编译器必须先判定被引用的是哪个对象。编译器有时也会优化代码使用对象传递值。这就意味着，对象被引用传递生成的代码通常会慢于值传递所生成的代码。
现在可以回答为什么不在所有的地方进行引用传递：

- 对于廉价的对象拷贝，拷贝的开销与绑定的开销差不多，因此推荐值传递因为代码生成会快一些。
- 对于昂贵的对象拷贝，拷贝占据了大部分开销，因此推荐传递（const）引用来避免拷贝。

最佳实践：当对象拷贝廉价时，推荐值传递；对象拷贝昂贵时，推荐 const 引用传递；如果不确定，推荐后者。
最后一个问题是，如何定义“廉价拷贝”？这里没有绝对的答案，因为这随着编译器，使用案例，架构等变化。然而可以设立一个简单的规则：如果一个对象使用两个或更少的“单词”的内存（一个“单词”约等于一个内存地址大小）以及没有准备开销，视为廉价拷贝。
下面代码定义了一个宏用于判断一个类型（或对象）是有两个或者更少的内存地址：

```
#include <iostream>

// 返回 true 如果类型（或者对象）使用两个或更少的内存地址
#define isSmall(T) (sizeof(T) <= 2 * sizeof(void*))

struct S
{
    double a, b, c;
};

int main()
{
```

```
    std::cout << std::boolalpha; // 打印 true 或 false 而不是 0 或 1
    std::cout << isSmall(int) << '\n'; // true
    std::cout << isSmall(double) << '\n'; // true
    std::cout << isSmall(S) << '\n'; // false

    return 0;
}
```

然而很难知道一个类类型的对象是否有准备开销。最好是假设大部分的标准库都有准备开销。

9.6 Introduction to pointers

address-of 操作符（&）

尽管变量使用的内存地址默认不会暴露给用户，还是可以访问其信息。**address-of** 操作符（&）返回其内存地址：

```
#include <iostream>

int main()
{
    int x{ 5 };
    std::cout << x << '\n';  // 打印 x 变量的值
    std::cout << &x << '\n'; // 打印 x 变量的内存地址

    return 0;
}
```

对象使用若干字节地址的情况下，address-of 将会返回对象使用的第一个字节的地址。

解引用操作符（*）

单独获取变量的地址并不是很有用。
最有用的做法是通过地址访问其所存储的值。**解引用操作符** dereference operator（*）（同时偶尔会被称为 **indirection operator**）返回给定内存地址的值作为左值：

```
#include <iostream>

int main()
{
    int x{ 5 };
    std::cout << x << '\n';  // 打印变量 x 的值
    std::cout << &x << '\n'; // 打印变量 x 的内存地址

    std::cout << *(&x) << '\n'; // 打印变量 x 在内存地址的值（括号是不需要的，仅方便阅读）

    return 0;
}
```

指针

指针是一个对象用于存储**内存地址**（通常来说是其他的变量）作为其值。这允许用户存储其它的对象在之后使用。
题外话：现代 C++ 中，指针有时会被称为“裸指针 raw pointers”或者“dumb pointers”，这是为了区别于“智能指针 smart pointers”（后续章节详解）。
与引用类型的声明相似，指针类型的声明使用星号（*）：

```
int;  // 普通 int
int&; // int 值的左值引用

int*; // int 值的指针（保存一个整数值的地址）
```

创建指针类型的变量如下：

```
int main()
{
    int x { 5 };    // 普通变量
    int& ref { x }; // 整数（绑定 x）的引用

    int* ptr;       // 整数的指针

    return 0;
}
```

注意星号是声明指针的语法一部分，而不是使用解引用操作符。
最佳实践：当声明一个指针类型，把星号放在类型后。

指针实例化

与普通变量一样，指针默认 * 不会 * 被初始化的。一个没有被初始化的指针有时会被称为**野指针**。野指针包含一个垃圾地址，对野指针进行解引用将会造成未定义行为。正因如此，应该总是让指针初始化一个已知值。
最佳实践：总是初始化指针。

```
int main()
{
    int x{ 5 };

    int* ptr;        // 一个未初始化指针（存储了垃圾地址）
    int* ptr2{};     // 一个空指针（下一章节讨论）
    int* ptr3{ &x }; // 一个初始化了变量 x 地址的指针

    return 0;
}
```

因为指针保存地址，当初始化或为指针分配值时，值必须为一个地址。通常来说，指针用来存储另一个变量的地址（可以使用 address-of 操作符（&）来获取）。
一旦指针存储了其他对象的地址，则可以使用解引用操作符（*）来访问该地址的值了。例如：

```cpp
#include <iostream>

int main()
{
    int x{ 5 };
    std::cout << x << '\n'; // 打印变量 x 的值

    int* ptr{ &x }; // ptr 保存 x 的地址
    std::cout << *ptr << '\n'; // 使用解引用操作符打印 ptr 保存的地址的值

    return 0;
}
```

指针与赋值

指针赋值有两种不同的方式：

1. 修改指针指向（通过分配一个新地址）
2. 修改其指向的值（解引用指针后赋值一个新值）

```cpp
#include <iostream>

int main()
{
    int x{ 5 };
    int* ptr{ &x }; // ptr 实例化指向 x

    std::cout << *ptr << '\n'; // 打印指向的地址的值（x 的地址）

    int y{ 6 };
    ptr = &y; // // 修改 ptr 指向 y

    std::cout << *ptr << '\n'; // 打印指向的地址的值（y 的地址）

    return 0;
}
```

```cpp
#include <iostream>

int main()
{
    int x{ 5 };
```

```
    int* ptr{ &x }; // ptr 实例化指向 x

    std::cout << x << '\n';    // 打印 x 的值
    std::cout << *ptr << '\n'; // 打印指向的地址的值（x 的地址）

    *ptr = 6; // ptr 存储地址的对象被赋值 6（注意 ptr 在这里被解引用了）

    std::cout << x << '\n';
    std::cout << *ptr << '\n'; // 打印指向的地址的值（x 的地址）

    return 0;
}
```

指针行为很想左值引用

```
#include <iostream>

int main()
{
    int x{ 5 };
    int& ref { x };  // x 的引用
    int* ptr { &x }; // x 的指针

    std::cout << x;
    std::cout << ref;           // 使用引用打印 x 的值（5）
    std::cout << *ptr << '\n';  // 使用指针打印 x 的值（5）

    ref = 6; // 使用引用修改 x 的值
    std::cout << x;
    std::cout << ref;           // 使用引用打印 x 的值（6）
    std::cout << *ptr << '\n';  // 使用指针打印 x 的值（6）

    *ptr = 7; // 使用指针修改 x 的值
    std::cout << x;
    std::cout << ref;           // 使用引用打印 x 的值（7）
    std::cout << *ptr << '\n';  // 使用指针打印 x 的值（7）

    return 0;
}
```

指针与引用还有一些其它的不同值得注意：

- 引用必须被初始化，指针则不需要（但是也应该初始化）
- 引用不是对象，指针则是
- 引用不可被重复位（修改自身引用其它别的），指针可以被修改其指向

- 引用必须总是绑定一个对象，指针可以指向空（下一章节讲解案例）
- 引用是“安全的”（悬垂引用之外），指针固有危险（下一章节详解）

address-of 操作符返回一个指针

值得注意的是 address-of 操作符（&）不会作为字面值返回其引用的对象。相反的是，其返回一个指针包含了其被引用者的地址，而该指针的类型有参数派生出来（例如 `int` 地址会返回 `int` 指针的地址）。

```
#include <iostream>
#include <typeinfo>

int main()
{
  int x{ 4 };
  std::cout << typeid(&x).name() << '\n'; // 打印 &x 的类型（int *）

  return 0;
}
```

而 gcc 这里打印的是“pi”（pointer to int）。因为 typeid().name() 的返回结果是编译器所决定的，用户的编译器有可能打印不同的东西，但是都是相同的意思。

指针的大小

指针的大小是根据编译后的可执行架构所决定的 – 32-bit 的可执行使用 32-bit 的内存地址，指针的大小为 32 bits（4 个字节）；64-bit 的可执行，指针的大小为 64 bits（8 个字节）。

悬垂指针

与悬垂引用类似，**悬垂指针** dangling pointer 是指针存储地址的对象不再有效（例如对象被销毁）。解引用一个悬垂指针会导致未定义的行为。
以下是创建悬垂指针的例子：

```
#include <iostream>

int main()
{
    int x{ 5 };
    int* ptr{ &x };

    std::cout << *ptr << '\n'; // 有效

    {
        int y{ 6 };
```

```
        ptr = &y;

        std::cout << *ptr << '\n'; // 有效
    } // y 离开作用域，ptr 现在悬垂了

    std::cout << *ptr << '\n'; // 解引用一个悬垂指针导致未定义行为

    return 0;
}
```

9.7 Null pointers

除了内存地址，指针还可以保存一个额外的东西：null 值。**null 值**（简略为 **null**）是一个特别的值意为没有值。当一个指针存储了 null 值，意味着没有指向任何东西。这样的指针被称为**空指针** null pointer。

创建空指针最简单的方法是使用值实例化：

```
int main()
{
    int* ptr {}; // ptr 现在是一个空指针，没有保存任何地址

    return 0;
}
```

最佳实践：如果不初始化一个有效的对象地址时，初始化空指针。

```
#include <iostream>

int main()
{
    int* ptr {}; // ptr 现在是一个空指针，没有保存任何地址

    int x { 5 };
    ptr = &x; // ptr 指向对象 x（不再是空指针）

    std::cout << *ptr << '\n'; // 通过解引用 ptr 打印值

    return 0;
}
```

nullptr 关键字

类似于关键字 `true` 和 `false` 代表布尔字面值，**nullptr** 关键字代表一个空指针的字面值。使用 `nullptr` 可以显式初始化或赋值空值给一个指针。

```
int main()
{
```

```
    int* ptr { nullptr }; // 可以使用 nullptr 初始化一个空指针

    int value { 5 };
    int* ptr2 { &value }; // ptr2 是一个有效指针
    ptr2 = nullptr; // 可以赋值 nullptr 使指针变为空指针

    someFunction(nullptr); // 也可以传递 nullptr 给函数，其入参是指针类型

    return 0;
}
```

最佳实践：当需要空指针字面值用于初始化，赋值或者传递空指针给函数式，使用 `nullptr` 。

解引用空指针导致未定义的行为

```
#include <iostream>

int main()
{
    int* ptr {}; // 创建一个空指针
    std::cout << *ptr << '\n'; // 解引用空指针

    return 0;
}
```

警告：任何时候使用指针时，需要额外的消息代码中没有解引用空指针或者悬垂指针，因为都会导致未定义的行为（可能导致程序崩溃）。

检查空指针

```
#include <iostream>

int main()
{
    int x { 5 };
    int* ptr { &x };

    // 指针转换为布尔值 false 如果它们为 null，反之为 true
    if (ptr == nullptr) // 显式测试等式
        std::cout << "ptr is null\n";
    else
        std::cout << "ptr is non-null\n";

    int* nullPtr {};
    std::cout << "nullPtr is " << (nullPtr==nullptr ? "null\n" : "non-null\n"); // 显式测试等式

    return 0;
}
```

使用 nullptr 避免悬垂指针

```
// 假设 ptr 是一个指针有可能为空指针
if (ptr) // 如果 ptr 不是一个空指针
    std::cout << *ptr << '\n'; // 可以被解引用
else
    // 做一些其它的事情，只要不是解引用 ptr（打印错误，等等）
```

最佳实践：一个指针应该要么存储有效对象的地址，要么设为 nullptr。这样只需要测试其是否为 null，可以假设任何非空指针都有效。

如果可能的话，使用引用而不是指针

最佳实践：推荐引用而不是指针，除非由指针提供的额外功能需要被用到。

9.8 Pointers and const

const 值的指针

const 值的指针 pointer to a const value（有时简称为 `pointer to const`）是指向常量值的一种（非 const）指针。

```
int main()
{
    const int x{ 5 };
    const int* ptr { &x }; // 可以：ptr 指向 "const int"

    *ptr = 6; // 不可以：不能修改 const 值

    return 0;
}
```

然而，因为 const 值的指针本身不是 const（仅仅是指向 const 值），所以可以通过赋值新地址的方式修改指针：

```
int main()
{
    const int x{ 5 };
    const int* ptr { &x }; // 可以：ptr 指向 const int x

    const int y{ 6 };
    ptr = &y; // 可以：ptr 现在指向 const int y

    return 0;
}
```

类似于 const 值的引用，const 值的指针也可以指向非 const 变量。const 值的指针将其指向的值视为常量值，无论对象其初始化定义为 const 与否：

```
int main()
{
    int x{ 5 }; // 非 const
    const int* ptr { &x }; // ptr 指向 "const int"

    *ptr = 6;  // 不允许： ptr 指向 "const int"，因此不可以通过 ptr 修改值
    x = 6; // 允许：通过非 const 标识符 x 访问，值仍然为非 const

    return 0;
}
```

const 指针

同样的可以使指针本身作为常量值。**const 指针**是一种在初始化后其地址不可修改的指针。

```
int main()
{
    int x{ 5 };
    int* const ptr { &x }; // 在星号后的 const 意味着是一个 const 指针

    return 0;
}
```

与通常 const 变量相同，const 指针必须初始化，其值不可以通过赋值被修改：

```
int main()
{
    int x{ 5 };
    int y{ 6 };

    int* const ptr { &x }; // 可以：const 指针初始化 x 的地址
    ptr = &y; // 错误：一旦初始化，const 指针不可改变

    return 0;
}
```

然而，因为被指向的 * 值 * 是非 const，可以通过解引用 const 指针修改值：

```
int main()
{
    int x{ 5 };
    int* const ptr { &x }; // ptr 总是指向 x

    *ptr = 6; // 可以：指向的值为非 const

    return 0;
}
```

指向 const 值的 const 指针

最后，通过放置 `const` 关键字在类型星号的前后，可以声明一个 **const pointer to a const value**：

```cpp
int main()
{
    int value { 5 };
    const int* const ptr { &value }; // 指向 const 值的 const 指针

    return 0;
}
```

9.9 Pass by address

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

void printByValue(std::string val) // 函数参数为 str 的拷贝
{
    std::cout << val << '\n'; // 通过拷贝进行打印
}

void printByReference(const std::string& ref) // 函数参数为绑定 str 的引用
{
    std::cout << ref << '\n'; // 通过引用进行打印
}

void printByAddress(const std::string* ptr) // 函数参数为存储 str 地址的指针
{
    std::cout << *ptr << '\n'; // 通过解引用指针进行打印
}

int main()
{
    std::string str{ "Hello, world!" };

    printByValue(str); // 值传递，拷贝 str
    printByReference(str); // 引用传递，不会拷贝 str
    printByAddress(&str); // 地址传递，不会拷贝 str

    std::string* ptr { &str }; // 定义一个存储 str 地址的指针变量
    printByAddress(ptr); // 地址传递，不会拷贝 str

    return 0;
}
```

地址传递允许函数修改入参值

当通过对象的地址进行传递，函数获取对象的地址，可以通过解引用进行访问。因为这个地址属于真实的对象（而不是对象的拷贝），如果函数入参的指针非 const，函数可以通过指针修改入参的值：

```
#include <iostream>

void changeValue(int* ptr) // 注意：ptr 在本示例中指向非 const
{
    *ptr = 6; // 修改值为 6
}

int main()
{
    int x{ 5 };

    std::cout << "x = " << x << '\n'; // 打印 5

    changeValue(&x); // 传递 x 地址给函数

    std::cout << "x = " << x << '\n'; // 打印 6

    return 0;
}
```

Null 检查

```
#include <iostream>

void print(int* ptr)
{
    if (ptr) // 如果 ptr 不是一个空指针
    {
        std::cout << *ptr;
    }
}

int main()
{
  int x{ 5 };

  print(&x);
  print(nullptr);

  return 0;
}
```

大多数情况下，与上述例子相反的判断更有效：测试函数入参是否为空作为先决条件：

```
#include <iostream>

void print(int* ptr)
{
  if (!ptr) // 如果 ptr 为空指针，提前结束函数
      return;

  // 到了这里可以认为 ptr 是有效的，因此不在需要测试或嵌套了

  std::cout << *ptr;
}

int main()
{
  int x{ 5 };

  print(&x);
  print(nullptr);

  return 0;
}
```

如果空指针根本不可以传递至函数，则加上 `assert` （这里 asserts 用于记录永远不会出现）：

```
#include <iostream>
#include <cassert>

void print(const int* ptr) // const int 指针
{
  assert(ptr); // debug 模式下如果传递的是空指针则程序失败（因为这永远不会出现）

  //（可选）在生产模式下处理错误，如果真实发生了也不会导致程序崩溃
  if (!ptr)
    return;

  std::cout << *ptr;
}

int main()
{
  int x{ 5 };

  print(&x);
  print(nullptr);

  return 0;
}
```

推荐传递（const）引用

注意上述例子中的 `print` 函数处理空值不是很理想 – 即直接退出函数。那么为什么允许用户传递空值呢？传递引用有着与传递指针相同的便利，而不用承担解引用空指针的风险。
相较于传递地址，传递 const 引用还有一些额外的优势。
首先，因为传递的对象必须拥有地址，仅左值可以被传递地址（右值没有地址）。传递 const 引用更加的灵活，既可以接受左值也可以接受右值：

```
1
2  // ... 与最开头的函数一致
3
4  int main()
5  {
6      printByValue(5);     // 有效，但是需要拷贝
7      printByReference(5); // 有效，因为入参是 const 引用
8      printByAddress(&5);  // 错误，不可以获取 address-of 右值
9
10     return 0;
11 }
```

其次，传递引用的语法更加自然，仅需要传递字面值或者对象。传递地址却需要加上 & 符号（调用时）以及 * 符号（函数编写时）。
最佳实践：现代 C++ 中，大部分的事务都可以通过传递地址完成。遵循这条规则：“尽可能使用传递引用，必须时才传递地址”。

“可选”入参的地址传递

```
1  #include <iostream>
2  #include <string>
3
4  void greet(std::string* name=nullptr)
5  {
6      std::cout << "Hello ";
7      std::cout << (name ? *name : "guest") << '\n';
8  }
9
10 int main()
11 {
12     greet(); // 还不知道是哪个用户
13
14     std::string joe{ "Joe" };
15     greet(&joe); // 现在知道了用户是 joe
16
17     return 0;
18 }
```

不过大多数情况下，使用重载函数会更好一些：

```cpp
#include <iostream>
#include <string>
#include <string_view>

void greet(std::string_view name)
{
    std::cout << "Hello " << name << '\n';
}

void greet()
{
    greet("guest");
}

int main()
{
    greet(); // 还不知道是哪个用户

    std::string joe{ "Joe" };
    greet(joe); // 现在知道了用户是 joe

    return 0;
}
```

这样做有更多的好处：不再需要担心解引用空值，以及可以直接传递字符串字面值。

修改指针参数的指向

当传递地址给函数时，地址从入参中被拷贝进指针参数（没有问题，因为拷贝地址很快）。现在考虑以下案例：

```cpp
#include <iostream>

// [[maybe_unused]] 去除编译器警告：ptr2 设置了但是没有被使用
void nullify([[maybe_unused]] int* ptr2)
{
    ptr2 = nullptr; // 使函数参数变为空指针
}

int main()
{
    int x{ 5 };
    int* ptr{ &x }; // ptr 指向 x

    std::cout << "ptr is " << (ptr ? "non-null\n" : "null\n"); // 打印 ptr is non-null

    nullify(ptr);

    std::cout << "ptr is " << (ptr ? "non-null\n" : "null\n"); // 打印 ptr is non-null
```

```
    return 0;
}
```

如上所见，在函数内修改函数入参指针的地址不会影响参数的地址（`ptr` 仍然指向 `x`）。当函数 `nullify()` 被调用时，`ptr2` 获取了传递进来的地址的拷贝（即 `ptr` 所存储的地址，即 `x` 的地址）。当函数修改 `ptr2` 的指向时，仅仅影响的是拷贝后的 `ptr2`。
那么想要函数修改指针所存储的地址需要怎么做呢?

通过引用传递地址?

正如可以通过引用传递普通变量一样，可以传递指针的引用:

```
#include <iostream>

void nullify(int*& refptr) // refptr 现在是指针的引用
{
    refptr = nullptr; // 使函数参数变为空指针
}

int main()
{
    int x{ 5 };
    int* ptr{ &x }; // ptr 指向 x

    std::cout << "ptr is " << (ptr ? "non-null\n" : "null\n"); // 打印 ptr is non-null

    nullify(ptr);

    std::cout << "ptr is " << (ptr ? "non-null\n" : "null\n"); // 打印 ptr is null
    return 0;
}
```

因为 `refptr` 现在是指针的引用，当 `ptr` 作为参数时，`refptr` 绑定 `ptr`。这就意味着 `refptr` 的任何改动都将作用于 `ptr` 上。

其实只有值传递

通过了引用传递，地址传递，值传递的学习，那么现在来看看如何做减法。
因为编译器总是能优化整个引用，很多时候实际需要引用的情况是不存在的。引用通常被编译器实现成指针。这就意味着在这个场景下，传递引用实际上仅仅传递的是地址（通过隐式的解引用来访问引用）。
同时之前的课程中提到了传递地址仅仅是拷贝调用者的地址给到被调用的函数 – 即以值传递的方式传递地址。
因此，可以总结 C++ 真正传递的是值! 传递地址（和引用）仅存在于需要解引用修改值的时候，这种时候是不可以用通常的值作为入参。

9.11 Return by reference and return by address

返回引用

返回引用可以避免返回类型的拷贝。

```
std::string&       returnByReference(); // 返回一个已存在的 std::string 的引用（廉价）
const std::string& returnByReferenceToConst(); // 返回一个已存在的 std::string 的 const 引用
    （廉价）
```

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

const std::string& getProgramName() // 返回一个 const 引用
{
    static const std::string s_programName { "Calculator" }; // static duration，在程序结束后销
    毁

    return s_programName;
}

int main()
{
    std::cout << "This program is named " << getProgramName();

    return 0;
}
```

返回引用的对象必须在函数返回后存在

使用引用作为返回时有一个严峻的警告：程序员必须确保被引用的对象存在的时间比返回引用时要长。否则，返回的引用会变成悬垂引用（对象被销毁了），使用这样的引用将导致未定义行为。

现代编译器在遇到尝试返回本地变量的引用时，会报警或者返回编译错误，但是编译器有时在遇到更复杂的情况下可能会遇到问题。

不要返回 non-const local static 变量的引用

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

const int& getNextId()
{
    static int s_x{ 0 }; // 注意：变量为 non-const
    ++s_x; // 生成下一个 id
    return s_x; // 返回其引用
```

```
}

int main()
{
    const int& id1 { getNextId() }; // id1 为引用
    const int& id2 { getNextId() }; // id2 为引用

    std::cout << id1 << id2 << '\n';

    return 0;
}
```

上述代码返回 22 。这是因为 id1 与 id2 引用的是同一个对象（static 变量 s_x ），因此当任何（例如 getNextId() ）修改了值，所有的引用访问了已修改的值。另一个常见的问题是当返回一个 static local 的 const 引用时，没有标准化的方法重置 s_x 置默认值。这样的程序要么使用非自然的方法（例如重置参数），或者只能由退出后重启来进行重置。
最佳实践：避免返回 non-const local static 变量的引用。

通过拷贝返回的引用进行赋值/初始化普通变量

如果函数返回引用，且引用用于初始化或者给非引用变量进行赋值，那么返回值将被拷贝（就像是直接返回值）。

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

const int& getNextId()
{
    static int s_x{ 0 };
    ++s_x;
    return s_x;
}

int main()
{
    const int id1 { getNextId() }; // id1 现在是普通变量，其获取 getNextId() 返回的引用并进行值拷贝
    const int id2 { getNextId() }; // id2 现在是普通变量，其获取 getNextId() 返回的引用并进行值拷贝

    std::cout << id1 << id2 << '\n';

    return 0;
}
```

上述代码中，getNextId() 返回引用，但是 id1 与 id2 为非引用变量。这种情况下，返回的引用值被拷贝成正常变量。因此打印 12 。

另外也要注意如果一个程序返回悬垂引用，引用在拷贝之前变为悬垂，也会带来未定义行为：

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

const std::string& getProgramName() // 返回 const 引用
{
    const std::string programName{ "Calculator" };

    return programName;
}

int main()
{
    std::string name { getProgramName() }; // 拷贝悬垂引用
    std::cout << "This program is named " << name << '\n'; // 未定义行为

    return 0;
}
```

返回引用参数的引用是可以的

有几种情况下返回对象的引用是有意义的。
如果入参就是引用，那么返回入参引用是安全的，例如：

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

const std::string& firstAlphabetical(const std::string& a, const std::string& b)
{
  return (a < b) ? a : b;
}

int main()
{
  std::string hello { "Hello" };
  std::string world { "World" };

  std::cout << firstAlphabetical(hello, world) << '\n';

  return 0;
}
```

调用者可以通过引用修改值

当 non-const 引用作为参数被传递至函数，函数则可以使用该引用进行值修改。
同样的，当 non-const 引用被返回，调用者可以使用该引用进行值修改。

```cpp
#include <iostream>

// 接收两个 non-const 引用，返回最大的那个
int& max(int& x, int& y)
{
    return (x > y) ? x : y;
}

int main()
{
    int a{ 5 };
    int b{ 6 };

    max(a, b) = 7; // 修改最大的 a 或 b 至 7

    std::cout << a << b << '\n';

    return 0;
}
```

返回地址

返回地址与返回引用，功能基本一致，前者返回的是对象的指针，后者返回的则是对象的引用。返回地址与返回引用一样需要额外的警惕 – 返回地址的这个对象必须在函数返回后存活着，不然调用者则会接收到一个悬垂指针。

相比于返回引用，返回地址的最大优势是在没有合适的对象返回时，可以让函数返回 `nullptr`。

而返回地址最大的劣势是，调用者必须记得在解引用之前进行 `nullptr` 的检查，否则空指针解引用则会带来未定义行为。正因为这样的危险，才更加推荐返回引用而不是返回地址，除非返回“无对象”是真实需要的。

最佳实践：推荐返回引用而不是返回地址，除非需求“无对象”（使用 `nullptr`）的返回。

9.12 Type deduction with pointers, references, and const

`auto` 关键字推导时会丢弃 `const` 限定符：

```cpp
const double foo()
{
    return 5.6;
}

int main()
{
    const double cd{ 7.8 };
```

```
9
10      auto x{ cd };     // double（const 被丢弃）
11      auto y{ foo() }; // double（const 被丢弃）
12
13      return 0;
14  }
```

通过添加 `const` 限定符可以重新应用 const：

```
1   const double foo()
2   {
3       return 5.6;
4   }
5
6   int main()
7   {
8       const double cd{ 7.8 };
9
10      const auto x{ cd };     // const double（const 重新应用）
11      const auto y{ foo() }; // const double（const 重新应用）
12
13      return 0;
14  }
```

类型推导丢弃引用

除了丢弃 const 限定符，类型推导也会丢弃引用：

```
1   #include <string>
2
3   std::string& getRef(); // 某返回引用的函数
4
5   int main()
6   {
7       auto ref { getRef() }; // 类型推导为 std::string（而不是 std::string&）
8
9       return 0;
10  }
```

与丢弃 const 限定符一样，如果需要推导的类型成为一个引用，可以进行重新应用：

```
1   #include <string>
2
3   std::string& getRef(); // 某返回引用的函数
4
5   int main()
6   {
7       auto ref1 { getRef() };  // std::string（引用被丢弃）
8       auto& ref2 { getRef() }; // std::string&（引用重新应用）
9
```

```
    return 0;
}
```

高等 const 与低等 const

高等 const 是 const 限定符直接应用于对象本身：

```
const int x;    // 该 const 应用于 x，因此它是高等 const
int* const ptr; // 该 const 应用于 ptr，因此它是高等 const
```

相反，**低等 const** 则是 const 限定符应用于对象的引用或者指针：

```
const int& ref; // 该 const 应用于被引用的对象，因此它是低等 const
const int* ptr; // 该 const 应用于被指向的对象，因此它是低等 const
```

const 值的引用永远是低等 const，而指针可以拥有高等，低等，或者两者兼具的 const：

```
const int* const ptr; // 左 const 低等，右 const 高等
```

当提到类型推导丢弃 const 限定符，它仅仅丢弃高等 const，而低等 const 是不会被丢弃的。接下来看几个例子。

类型推导与 const 引用

如果初始化的是 const 的引用，引用首先被丢弃（再重新应用，如果合适），接着任何高等 const 从结果中丢弃。

```
#include <string>

const std::string& getRef(); // 某返回 const 引用的函数

int main()
{
    auto ref1{ getRef() }; // std::string（引用被丢弃，接着高等 const 在结果中被丢弃）

    return 0;
}
```

上述例子中，因为 `getRef()` 返回 `const std::string&`，引用会先被丢弃，留下 `const std::string`。该 const 为高等，因此也被丢弃，留下推导类型 `std::string`。
可以进行重新应用：

```
#include <string>

const std::string& getRef(); // 某返回 const 引用的函数

int main()
{
    auto ref1{ getRef() };        // std::string（高等 const 与引用被丢弃了）
```

```
    const auto ref2{ getRef() };  // const std::string（高等 const 被重新应用，引用被丢弃了）

    auto& ref3{ getRef() };       // const std::string&（引用被重新应用，低等 const 未被丢弃）
    const auto& ref4{ getRef() }; // const std::string&（引用与 const 都被重新应用）

    return 0;
}
```

`ref1` 和 `ref2` 没有问题。问题在于 `ref3`，通常来说引用会被丢弃，但是因为重新应用了引用，它没有被丢弃。这意味着类型仍然是 `const std::string&`。同时因为 const 为低等 const，没有被丢弃。因此这里的推导类型是 `const std::string&`。

`ref4` 类似于 `ref3`，除了重新应用了 `const` 限定符。因为类型已经被推导成 const 的引用，重新应用 `const` 在这里是多余的。在此使用 `const` 可以显式的清楚知道结果为 const（而 `ref3` 的案例中推导的 const，其结果是隐式的且不明显的）。

最佳实践：如果需要 const 引用，重新应用 `const` 即便它不是被严格的需求，这可以使代码更加清晰并且防止错误。

类型推导与指针

不同于引用，类型推导不会丢弃指针：

```
#include <string>

std::string* getPtr(); // 某返回指针的函数

int main()
{
    auto ptr1{ getPtr() }; // std::string*

    return 0;
}
```

也可以用星号来表示类型推导：

```
#include <string>

std::string* getPtr(); // 某返回指针的函数

int main()
{
    auto ptr1{ getPtr() };  // std::string*
    auto* ptr2{ getPtr() }; // std::string*

    return 0;
}
```

auto 与 auto* 的不同之处

当使用 auto 作为指针类型的初始化，auto 的类型推导包含了指针。因此上述例子中的 ptr1 ，替换 auto 的类型是 std::string* 。
当使用 auto* 作为指针类型的初始化，类型推导则 * 不会 * 包含指针 – 指针再之后被重新应用在推导的类型上。因此 ptr2 ，替换 auto 的类型是 std::string ，指针再重新应用。
多数情况下，两者的效果是一致的（ptr1 与 ptr2 都被推导为 std::string* ）。
然而它们之间还有一些区别。首先，auto* 必须被指定为指针初始化，否则编译器会报错：

```
#include <string>

std::string* getPtr(); // 某返回指针的函数

int main()
{
    auto ptr3{ *getPtr() };      // std::string（因为解引用了 getPtr()）
    auto* ptr4{ *getPtr() };     // 不能被编译（initializer 不是一个指针）

    return 0;
}
```

类型推导与 const 指针

由于指针不会被丢弃，不需要担心它。但是对于指针，还有 const 指针以及指向 const 的指针需要考虑，同样也有 auto vs auto* 。与引用类似，只有高等 const 在指针类型推导时会被丢弃。
来看一个简单的例子：

```
#include <string>

std::string* getPtr(); // 某返回指针的函数

int main()
{
    const auto ptr1{ getPtr() };  // std::string* const
    auto const ptr2 { getPtr() }; // std::string* const

    const auto* ptr3{ getPtr() }; // const std::string*
    auto* const ptr4{ getPtr() }; // std::string* const

    return 0;
}
```

当使用 auto const 或 const auto 时，可以说，“让任何推导类型都为 const”。因此 ptr1 与 ptr2 的推导类型是 std::string* const 。类似于 const int 和 int const 表明的是同样一件事。

然而当使用 `auto*` 时，const 限定符的顺序则是有作用的。`const` 在左侧意为“让推导的指针类型指向 const”，而 `const` 在右侧意为“让推导的指针类型为 const 指针”。因此 `ptr3` 成为了指向 const 的指针，`ptr4` 成为了 const 指针。

接下来看一个例子 initializer 作为指向 const 的 const 指针：

```
#include <string>

const std::string* const getConstPtr(); // 某返回指向 const 的 const 指针的函数

int main()
{
    auto ptr1{ getConstPtr() };  // const std::string*
    auto* ptr2{ getConstPtr() }; // const std::string*

    auto const ptr3{ getConstPtr() };  // const std::string* const
    const auto ptr4{ getConstPtr() };  // const std::string* const

    auto* const ptr5{ getConstPtr() }; // const std::string* const
    const auto* ptr6{ getConstPtr() }; // const std::string*

    const auto const ptr7{ getConstPtr() };  // 错误：const 限定符不可以被二次应用
    const auto* const ptr8{ getConstPtr() }; // const std::string* const

    return 0;
}
```

`ptr1` 与 `ptr2` 很直接。高等 const 被丢弃，低等 const 保留，因此这两种情况的最终类型皆为 `const std::string*` 。

`ptr3` 与 `ptr4` 同样很直接。高等 const 被丢弃，但是被重新应用，因此这两种情况皆为 `const std::string* const` 。

`ptr5` 与 `ptr6` 类似于之前的例子。高等 const 被丢弃。对于 `ptr5` 而言，`auto* const` 重新应用了高等 const，其最终类型为 `const std::string* const` 。而对于 `ptr6` 而言，`const auto*` 重新应用的 const 为其所指向的（在这里已经是 const 了），因此最终类型为 `const std::string*` 。

`ptr7` 应用了 const 两次，这是不被允许的，因此导致编译错误。

`ptr8` 则是在指针的两侧都重新应用了 const（这是允许的因为 `auto*` 必须为一个指针类型），因此其返回的类型是 `const std::string* const` 。

最佳实践：如果需要一个 const 指针，重新应用 `const` 限定符即使不是真正严格的需求，这样使得意图更为明确以及可以防止错误。

10 Compound Types: Enums and Structs

10.2 Unscoped enumerations

枚举

枚举 enumeration（也被称为 **enumerated type** 或是 **enum**）是一种包含了所有可能值并且每个值被定义为符号常量（称为 **enumerator**）的组合型数据类型。
C++ 支持两种类型的枚举：无范围枚举 unscoped enumerations（本章讲解），以及有范围枚举 scoped enumerations（下一章讲解）。

无范围枚举

通过 `enum` 关键字定义无范围枚举。

```
// 定义一个名为 Color 的无范围枚举
enum Color
{
    // 这些为成员
    // 这些象征性的常量定义了所有该枚举可能出现的值，通过逗号分隔而不是分号
    red,
    green,
    blue, // 最后一个逗号是可选的，推荐使用
}; // 枚举定义的最后必须由分号结束

int main()
{
    // 定义一些枚举类类型 Color 的变量
    Color apple { red };   // apple 为 red
    Color shirt { green }; // shirt 为 green
    Color cup { blue };    // cup 为 blue

    Color socks { white }; // 错误：white 不是 Color 枚举的成员
    Color hat { 2 };       // 错误：2 不是 Color 枚举的成员

    return 0;
}
```

枚举以及其成员的命名

为了方便枚举类都由大写字母开头（与成语定义类型相同）。
警告：枚举不需要被命名，但是未命名的枚举在现代 C++ 中应该避免。

枚举的类型是独特类型

每个被创建的枚举的类型被视作**独特类型** distinct type，意味着编译器可以从其他类型中做出区分（不同于 typedefs 或者类型别名，它们被视为与其所别名的类型为非独特类型）。
因为枚举类型是独特类型，枚举成员为枚举中的一部分，因此成员不可以用于其他枚举类型的对象：

```
enum Pet
{
    cat,
    dog,
    pig,
    whale,
};

enum Color
{
    black,
    red,
    blue,
};

int main()
{
    Pet myPet { black }; // 编译错误：black 不是 Pet 的成员
    Color shirt { pig }; // 编译错误：pig 不是 Color 的成员

    return 0;
}
```

使用枚举

因为枚举成员是具有描述性质的，它们对于增强代码可读性有非常大的作用。枚举类型最好是用于有一小个集合而又有关联的常量，对象只需要一次涵盖这些值即可。

```
enum DaysOfWeek
{
    sunday,
    monday,
    tuesday,
    wednesday,
    thursday,
    friday,
    saturday,
};

enum CardinalDirections
```

```
{
    north,
    east,
    south,
    west,
};

enum CardSuits
{
    clubs,
    diamonds,
    hearts,
    spades,
};
```

有时函数可以返回状态码给调用者来只是函数的执行是否顺利或是遇到何种错误。传统的做法是用一些小的负值来代表何种错误：

```
int readFileContents()
{
    if (!openFile())
        return -1;
    if (!readFile())
        return -2;
    if (!parseFile())
        return -3;

    return 0; // success
}
```

然而直接使用值并没有任何描述性，因此使用枚举类型将会是另一种更好的方式：

```
enum FileReadResult
{
    readResultSuccess,
    readResultErrorFileOpen,
    readResultErrorFileRead,
    readResultErrorFileParse,
};

FileReadResult readFileContents()
{
    if (!openFile())
        return readResultErrorFileOpen;
    if (!readFile())
        return readResultErrorFileRead;
    if (!parseFile())
        return readResultErrorFileParse;

    return readResultSuccess;
```

```
}
```

接着调用者可以测试函数的返回值，相比于返回数值，这样便于更好的理解：

```
if (readFileContents() == readResultSuccess)
{
    // do something
}
else
{
    // print error message
}
```

枚举类型也可以作为函数入参：

```
enum SortOrder
{
    alphabetical,
    alphabeticalReverse,
    numerical,
};

void sortData(SortOrder order)
{
    if (order == alphabetical)
        // 根据字母顺序排序
    else if (order == alphabeticalReverse)
        // 根据字母反序顺序排序
    else if (order == numerical)
        // 根据数值排序
}
```

无范围枚举的作用域

无范围枚举之所以有这么个名字是因为它们把成员的名称放入了与枚举其自身所定义的作用域中（而不是创建一个新的作用域，类似于命名空间那样）。

```
enum Color // enum 定义在全局命名空间
{
    red, // 因此 red 也定义在全局命名空间
    green,
    blue,
};

int main()
{
    Color apple { red };
。
    return 0;
}
```

这就会导致拥有相同成员的若干枚举会发生命名冲突：

```
enum Color
{
    red,
    green,
    blue, // blue 放入了全局命名空间
};

enum Feeling
{
    happy,
    tired,
    blue, // 错误：命名冲突
};

int main()
{
    Color apple { red };
    Feeling me { happy };

    return 0;
}
```

避免命名冲突

```
enum Color
{
    color_red,
    color_blue,
    color_green,
};

enum Feeling
{
    feeling_happy,
    feeling_tired,
    feeling_blue, // 不再有冲突
};

int main()
{
    Color paint { color_blue };
    Feeling me { feeling_blue };

    return 0;
}
```

而更好的做法是给它们提供各自的命名空间：

```cpp
namespace color
{
    // Color 枚举与成员都定义在 color 命名空间内
    enum Color
    {
        red,
        green,
        blue,
    };
}

namespace feeling
{
    enum Feeling
    {
        happy,
        tired,
        blue, // feeling::blue 不会与 color::blue 冲突
    };
}

int main()
{
    color::Color paint { color::blue };
    feeling::Feeling me { feeling::blue };

    return 0;
}
```

最佳实践：推荐枚举放入有名称的空间（例如命名空间或者类），这样就不会污染全局的命名空间了。

枚举的对比

```cpp
#include <iostream>

enum Color
{
    red,
    green,
    blue,
};

int main()
{
    Color shirt{ blue };

    if (shirt == blue)
```

```
        std::cout << "Your shirt is blue!";
    else
        std::cout << "Your shirt is not blue!";

    return 0;
}
```

10.3 Unscoped enumeration input and output

上一章讲到了枚举成员是符号常量，但是没有提到它们是整数型符号常量。也就是说，枚举可以存储整数值。

```
#include <iostream>

enum Color
{
    black, // 赋值 0
    red, // 赋值 1
    blue, // 赋值 2
    green, // 赋值 3
    white, // 赋值 4
    cyan, // 赋值 5
    yellow, // 赋值 6
    magenta, // 赋值 7
};

int main()
{
    Color shirt{ blue }; // 这里实际上存储的是整数 2

    return 0;
}
```

显式的定义成员值也是可以的。这里的整数值可以为正为负，也可以共享同一个值。任何没定义的成员将被赋予一个较前成员的值增加 1 的整数：

```
enum Animal
{
    cat = -3,
    dog,         // 赋值 -2
    pig,         // 赋值 -1
    horse = 5,
    giraffe = 5, // 与 horse 值相同
    chicken,     // 赋值 6
};
```

注意这里的 `horse` 与 `giraffe` 共享同一个值。当这种情况出现时，枚举成员变不再是独立的了 – `horse` 与 `giraffe` 变成可以交换的了。尽管 C++ 允许这样的情况发生，但是最好

还是避免。
最佳实践：避免显式赋值枚举成员，除非有令人信服的原因。

无范围枚举隐式转换整数值

```
#include <iostream>

enum Color
{
    black, // 赋值 0
    red, // 赋值 1
    blue, // 赋值 2
    green, // 赋值 3
    white, // 赋值 4
    cyan, // 赋值 5
    yellow, // 赋值 6
    magenta, // 赋值 7
};

int main()
{
    Color shirt{ blue };

    std::cout << "Your shirt is " << shirt; // 这是做什么呢?

    return 0;
}
```

因为枚举类型存储整数值，那么打印内容：`Your shirt is 2` 。

打印枚举名称

通常的做法：

```
// Using if-else for this is inefficient
void printColor(Color color)
{
    if (color == black) std::cout << "black";
    else if (color == red) std::cout << "red";
    else if (color == blue) std::cout << "blue";
    else std::cout << "???";
}
```

然而使用一系列的 if-else 声明效率低下，因为在匹配找到之前还需要若干比较。一个高效的做法是使用 switch 声明：

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

```
3
4  enum Color
5  {
6      black,
7      red,
8      blue,
9  };
10
11
12 // 后面会展示 C++17 的更好的版本
13 std::string getColor(Color color)
14 {
15     switch (color)
16     {
17     case black: return "black";
18     case red:   return "red";
19     case blue:  return "blue";
20     default:    return "???";
21     }
22 }
23
24 int main()
25 {
26     Color shirt { blue };
27
28     std::cout << "Your shirt is " << getColor(shirt) << '\n';
29
30     return 0;
31 }
```

这样打印出 `Your shirt is blue` 。

然而这个版本仍然没有效率，因为每次函数被调用时都需要创建并返回 `std::string` （非常的昂贵）。

在 C++17 中，一个更好的方法是使用 `std::string_view` 来代替 `std::string` 。`std::string_view` 允许用户返回字符串字面量并且拷贝起来没有那么的昂贵。

```
1  #include <iostream>
2  #include <string_view> // C++17
3
4  enum Color
5  {
6      black,
7      red,
8      blue,
9  };
10
11 constexpr std::string_view getColor(Color color) // C++17
12 {
```

```
    switch (color)
    {
    case black: return "black";
    case red:   return "red";
    case blue:  return "blue";
    default:    return "???";
    }
}

int main()
{
    Color shirt{ blue };

    std::cout << "Your shirt is " << getColor(shirt) << '\n';

    return 0;
}
```

10.4 Scoped enumerations (enum classes)

尽管无范围枚举在 C++ 中是独立类型，它们并不是类型安全的，有时候甚至会允许用户做没有意义的行为。

```
#include <iostream>

int main()
{
    enum Color
    {
        red,
        blue,
    };

    enum Fruit
    {
        banana,
        apple,
    };

    Color color { red };
    Fruit fruit { banana };

    if (color == fruit) // 编译器将会作整数来比较 color 和 fruit
        std::cout << "color and fruit are equal\n"; // 并得出它们相等!
    else
        std::cout << "color and fruit are not equal\n";

    return 0;
```

```
}
```

有范围枚举

上述问题的解决方案是使用**有范围枚举** scoped enumeration（C++ 中通常被称为 **enum class**）。有范围枚举类似于无范围枚举，不过主要有两个最大的不同：它们是强类型的（即不会隐式转换成整数），以及强作用域的（枚举成员仅仅放置在枚举自身域内）。
创建有范围枚举可以使用 `enum class`，其余的部分则跟无范围枚举的定义一致：

```
#include <iostream>
int main()
{
    enum class Color // "enum class" 定义为有范围的枚举
    {
        red, // red 被视为 Color 作用域的一部分
        blue,
    };

    enum class Fruit
    {
        banana, // banana 被视为 Fruit 作用域的一部分
        apple,
    };

    Color color { Color::red }; // 注意：red 不再是可以直接被访问到的，必须使用 Color::red
    Fruit fruit { Fruit::banana }; // 注意：banana 不再是可以直接被访问到的，必须使用 Fruit::
     banana

    if (color == fruit) // 编译错误：编译器不知道如何比较两个不同的类型
        std::cout << "color and fruit are equal\n";
    else
        std::cout << "color and fruit are not equal\n";

    return 0;
}
```

额外话：`class` 关键字（同样也有 `static` 关键字），在 C++ 中是一个最超载的关键字，根据不同的语境会有不同的意思。尽管有范围枚举使用了 `class` 关键字，其也不被视为“class 类”（即为 structs，classes，以及 unions 所保留的）。

有范围枚举定义自己的作用域

有别于无范围枚举，即枚举成员与枚举在同一个作用域中，有范围枚举使其成员仅仅处于其自身定义的作用域中。换言之，有范围枚举作用类似于命名空间中的枚举成员。内建的命名空间减少了全局命名空间污染，以及潜在的命名冲突问题。

有范围枚举不会隐式转换成整数

不同于无范围枚举，有范围枚举不会隐式转换成整数。
最佳实践：更加推荐有范围枚举而不是无范围枚举，除非是有令人信服的原因。
`using enum` 声明（C++20）
在 C++20 中，`using enum` 声明可以引入所有的枚举成员至当前作用域。当使用的是 enum class 类型，这使得用户可以在无需添加前缀的情况下访问 enum class 的成员。
这在拥有很多标识符，以及重复的前缀（例如在 switch 声明中），用起来非常的方便：

```cpp
#include <iostream>
#include <string_view>

enum class Color
{
    black,
    red,
    blue,
};

constexpr std::string_view getColor(Color color)
{
    using enum Color; // 将所有的 Color 枚举成员引入当今作用域（C++20）
    // 现在可以不再使用 Color:: 作为前缀来访问枚举成员了

    switch (color)
    {
    case black: return "black"; // 注意：black 而不是 Color::black
    case red:   return "red";
    case blue:  return "blue";
    default:    return "???";
    }
}

int main()
{
    Color shirt{ Color::blue };

    std::cout << "Your shirt is " << getColor(shirt) << '\n';

    return 0;
}
```

10.5 Introduction to structs, members, and member selection

结构体 struct 是程序定义类型，允许用户打包若干变量在同一个类型内。

定义结构体

```
struct Employee
{
    int id {};
    int age {};
    double wage {};
};
```

`struct` 关键字告诉编译器正在定义一个名为 `Employee` 的结构体。
在花括号内，定义该结构体内所包含的变量。这些变量是结构体的一部分，被称为**数据成员** data members（或**成员变量** member variables）。
最后以分号结束结构体类型的定义。

定义结构体对象

```
Employee joe; // Employee 为类型，joe 为变量名
```

访问成员

```
struct Employee
{
    int id {};
    int age {};
    double wage {};
};

int main()
{
    Employee joe;

    return 0;
}
```

上述例子中，`joe` 意为整个结构体对象（包含了成员变量）。访问成员变量需要使用**成员选择操作符** member selection operator（`.`）介于结构体变量名与成员名之间。

```
#include <iostream>

struct Employee
{
    int id {};
    int age {};
    double wage {};
};

```

```
int main()
{
    Employee joe;

    joe.age = 32;

    std::cout << joe.age << '\n';

    return 0;
}
```

```
#include <iostream>

struct Employee
{
    int id {};
    int age {};
    double wage {};
};

int main()
{
    Employee joe;
    joe.id = 14;
    joe.age = 32;
    joe.wage = 60000.0;

    Employee frank;
    frank.id = 15;
    frank.age = 28;
    frank.wage = 45000.0;

    int totalAge { joe.age + frank.age };

    if (joe.wage > frank.wage)
        std::cout << "Joe makes more than Frank\n";
    else if (joe.wage < frank.wage)
        std::cout << "Joe makes less than Frank\n";
    else
        std::cout << "Joe and Frank make the same amount\n";

    // Frank 升职了
    frank.wage += 5000.0;

    // 今天是 Joe 的生日
    ++joe.age; // 使用 pre-increment 来增加 Joe 的年龄

    return 0;
}
```

10.6 Struct aggregate initialization

数据成员默认不被初始化

```cpp
#include <iostream>

struct Employee
{
    int id; // 注意：这里没有初始化
    int age;
    double wage;
};

int main()
{
    Employee joe; // 注意：这里也没有初始化器
    std::cout << joe.id << '\n';  // 出现未定义行为

    return 0;
}
```

什么是聚合?

通用的编程中，**聚合数据类型** aggregate data type（也被称为 **aggregate**）是一种包含若干数据类型的任意类型。有些类型的聚合允许成员拥有不同的类型（例如 structs），而其它的需要所有成员必须只有一种类型（例如 arrays）。
C++ 中，一个 aggregate 的定义更窄一些，也更复杂一些：
- 是一种 class 类型（struct，class，或者 union），或者一种 array 类型（内建的 array 或者 ‘std::array‘）。- 没有 private 或者 protected non-static 数据类型。- 没有用户声明或者继承的构造器。- 没有基础类。- 没有虚拟成员函数。

struct 的聚合初始化

因为普通变量近可以存储单个类型，用户仅需要提供单个 initializer，而 struct 可以拥有若干成员，因此当定义 struct 类型的对象是，需要一种可以同时初始化若干成员的方法。
Aggregates 使用的初始化方法被称为**聚合初始化** aggregate initialization。用户仅需要提供一个 **initializer list** 作为初始化器，并且其中由逗号分隔初始值，便可以进行 struct 初始化了。
与普通变量可以拷贝初始化 copy-list initialized，直接初始化 direct initialized，或者列表初始化一样 list initialized，聚合初始化也拥有这三种形态的：

```cpp
struct Employee
{
    int id {};
```

```
    int age {};
    double wage {};
};

int main()
{
    Employee frank = { 1, 32, 60000.0 }; // 拷贝列表初始化使用等号 + 花括号
    Employee robert ( 3, 45, 62500.0 );  // 直接初始化使用圆括号（C++20）
    Employee joe { 2, 28, 45000.0 };     // 列表初始化使用使用花括号（推荐）

    return 0;
}
```

上述的每个初始化都是**依成员方向初始化**的，这就意味着 struct 中每个成员是依照声明的顺序进行初始化的。
最佳实践：推荐使用（非拷贝）花括号初始化。

列表初始化中的缺失初始化器

```
struct Employee
{
    int id {};
    int age {};
    double wage {};
};

int main()
{
    Employee joe { 2, 28 }; // joe.wage 将会被初始化成 0.0

    return 0;
}
```

Const structs

```
struct Rectangle
{
    double length {};
    double width {};
};

int main()
{
    const Rectangle unit { 1.0, 1.0 };
    const Rectangle zero { }; // 值初始化所有成员

```

```
    return 0;
}
```

指定式的初始化器（C++20）

C++20 新增了一种称为 **designated initializers** 的初始化 struct 的方式。

```
struct Foo
{
    int a{ };
    int b{ };
    int c{ };
};

int main()
{
    Foo f1{ .a{ 1 }, .c{ 3 } }; // 可行：f.a = 1, f.b = 0（值初始化），f.c = 3
    Foo f2{ .b{ 2 }, .a{ 1 } }; // 错误：初始化顺序与结构体声明的顺序不匹配

    return 0;
}
```

Designated initializers 很好因为它们提供了某种程度的自文档的形式可以帮助用户避免不小心弄错初始化顺序。然而，designated initializers 同样也使得列表初始化变得非常散乱，因此这里不推荐其为最佳实践。

最佳实践：当对 aggregate 添加新成员，安全的做法是在其定义的底部添加，这样可以确保初始化时，其它成员不会错位。

通过 initializer 列表赋值

```
struct Employee
{
    int id {};
    int age {};
    double wage {};
};

int main()
{
    Employee joe { 1, 32, 60000.0 };

    joe.age  = 33;      // Joe 过了个生日
    joe.wage = 66000.0; // 也获得了升职

    return 0;
}
```

上述对于单个成员赋值是没有问题的，但是如果要更新多个成员不是那么方便。

```
struct Employee
{
    int id {};
    int age {};
    double wage {};
};

int main()
{
    Employee joe { 1, 32, 60000.0 };
    joe = { joe.id, 33, 66000.0 }; // Joe 过了个生日也获得了升职

    return 0;
}
```

通过 designated initializers 赋值（C++20）

```
struct Employee
{
    int id {};
    int age {};
    double wage {};
};

int main()
{
    Employee joe { 1, 32, 60000.0 };
    joe = { .id = joe.id, .age = 33, .wage = 66000.0 }; // Joe 过了个生日也获得了升职

    return 0;
}
```

10.7 Default member initialization

当定义 struct（或 class）类型时，可以给每个成员提供默认初始值作为类型定义的一部分。这个过程被称为**非 static 成员初始化** non-static member initialization，同时初始值被称为**默认成员初始化器** default member initializer。

```
struct Something
{
    int x;       // 无初始化值（坏）
    int y {};    // 默认值初始化
    int z { 2 }; // 显式默认值
};

```

```
int main()
{
    Something s1; // s1.x 没有被初始化， s1.y 为 0，以及 s1.z 为 2

    return 0;
}
```

显式初始化值优先于默认值

```
struct Something
{
    int x;
    int y {};
    int z { 2 };
};

int main()
{
    Something s2 { 5, 6, 7 }; // 为 s2.x，s2.y，以及 s2.z 显式初始化（默认值没有被使用）

    return 0;
}
```

当默认值存在时在初始化列表中缺失的初始化器

```
struct Something
{
    int x;
    int y {};
    int z { 2 };
};

int main()
{
    Something s3 {}; // 值初始化 s3.x，s3.y 与 s3.z 使用默认值

    return 0;
}
```

总是为成员提供默认值

```
struct Fraction
{
  int numerator { }; // 应该在这里使用 { 0 } 但是因为是例子，因此这里用值初始化
  int denominator { 1 };
```

```
};

int main()
{
  Fraction f1;          // f1.numerator 值初始化 0, f1.denominator 默认为 1
  Fraction f2 {};       // f2.numerator 值初始化 0, f2.denominator 默认为 1
  Fraction f3 { 6 };    // f3.numerator 初始化为 6, f3.denominator 默认为 1
  Fraction f4 { 5, 8 }; // f4.numerator 初始化为 5, f4.denominator 初始化为 8

  return 0;
}
```

最佳实践：为所有成员提供默认值。这样可以确保在即使变量的定义没有包含初始化列表的情况下，所有成员也可以被初始化，。

aggregates 的默认初始化 vs 值初始化

```
Fraction f1;          // f1.numerator 值初始化为 0, f1.denominator 默认为 1
Fraction f2 {};       // f2.numerator 值初始化为 0, f2.denominator 默认为 1
```

最佳实践：如果没有显式的初始化值提供给 aggregate，那么推荐使用值初始化（通过空的花括号初始化器）而不是默认初始化（不带花括号）。

10.8 Struct passing and miscellany

structs 传递（通过引用）

使用 structs 而不是单个变量的最大好处就是可以传递整个 struct 给需要其成员的函数。structs 通常通过（const）引用传递来避免拷贝。

```
#include <iostream>

struct Employee
{
    int id {};
    int age {};
    double wage {};
};

void printEmployee(const Employee& employee) // 注意这里传递引用
{
    std::cout << "ID:   " << employee.id << '\n';
    std::cout << "Age:  " << employee.age << '\n';
    std::cout << "Wage: " << employee.wage << '\n';
}

int main()
```

```
{
    Employee joe { 14, 32, 24.15 };
    Employee frank { 15, 28, 18.27 };

    // 打印 Joe 的信息
    printEmployee(joe);

    std::cout << '\n';

    // 打印 Frank 的信息
    printEmployee(frank);

    return 0;
}
```

返回 structs

设想如果需要一个函数返回三维的笛卡尔坐标：

```
#include <iostream>

struct Point3d
{
    double x { 0.0 };
    double y { 0.0 };
    double z { 0.0 };
};

Point3d getZeroPoint()
{
    // 创建一个变量并返回
    Point3d temp { 0.0, 0.0, 0.0 };
    return temp;
}

int main()
{
    Point3d zero{ getZeroPoint() };

    if (zero.x == 0.0 && zero.y == 0.0 && zero.z == 0.0)
        std::cout << "The point is zero\n";
    else
        std::cout << "The point is not zero\n";

    return 0;
}
```

返回未命名 structs

上述的 ‘getZeroPoint()‘ 创建了一个命名对象（‘temp‘）并返回，这里可以改进一下返回临时（未命名）对象：

```
Point3d getZeroPoint()
{
    return Point3d { 0.0, 0.0, 0.0 }; // 返回未命名的 Point3d
}
```

这种情况下，临时的 Point3d 被创建，拷贝后返回给调用者，然后在表达式结束时销毁。那么在函数有显式的返回值（‘Point3d‘）而不是类型推导（‘auto‘ 作为返回值）时，甚至可以省略返回时的声明：

```
Point3d getZeroPoint()
{
    // 已经指定了函数的返回值类型
    // 因此这里便不在需要再次声明
    return { 0.0, 0.0, 0.0 }; // 返回未命名的 Point3d
}
```

注意因为该案例中返回的都是 0 值，可以使用空花括号来返回一个值初始化的 Point3d：

```
Point3d getZeroPoint()
{
    return {};
}
```

带有程序定义成员的 structs

C++ 中，structs（以及 classes）可以拥有程序定义类型的成员。这里有两种方法。
首先定义一个程序定义类型（全局作用域中），接着使用其作为另一个程序定义类型的成员：

```
#include <iostream>

struct Employee
{
    int id {};
    int age {};
    double wage {};
};

struct Company
{
    int numberOfEmployees {};
    Employee CEO {}; // Employee 为 Company struct 内部的 struct
};

int main()
```

```
{
    Company myCompany{ 7, { 1, 32, 55000.0 } }; // 嵌套的初始化列表用于初始化 Employee
    std::cout << myCompany.CEO.wage; // 打印 CEO 薪水
}
```

其次，类型同样也可以嵌套在另一个类型内部：

```
#include <iostream>

struct Company
{
    struct Employee // 通过 Company::Employee 进行访问
    {
        int id{};
        int age{};
        double wage{};
    };

    int numberOfEmployees{};
    Employee CEO{}; // Employee 为 Company struct 内部的 struct
};

int main()
{
    Company myCompany{ 7, { 1, 32, 55000.0 } }; // 嵌套的初始化列表用于初始化 Employee
    std::cout << myCompany.CEO.wage; // 打印 CEO 薪水
}
```

这种做法更常见于 classes，详见 13.17。

Struct 大小与数据结构偏移

通常而言，struct 的大小是所有成员大小之和，不过也不总是如此！

```
#include <iostream>

struct Foo
{
    short a {};
    int b {};
    double c {};
};

int main()
{
    std::cout << "The size of Foo is " << sizeof(Foo) << '\n';

    return 0;
}
```

在很多操作系统上，short 为 2 个字节，int 为 4 个字节，已经 double 为 8 哥字节，因此预期的 ‘sizeof(Foo)‘ 应该为 2 + 4 + 8 = 14 字节。然而在作者机器上打印的是：The size of Foo is 16。一个 struct 的大小只能说其最小是所有变量大小之和，但是往往会更大！因为性能的原因，编译器偶尔会在 struct 之间增加间隔（被称为 **padding**）。
上述的 ‘Foo‘ 中，编译器无形的添加 2 个字节在 ‘a‘ 后，使得结构体变成了 16 字节而非 14。这实际上会对 struct 大小产生很大的影响：

```
#include <iostream>

struct Foo1
{
    short a{};
    short qq{}; // 注意：qq 在此定义
    int b{};
    double c{};
};

struct Foo2
{
    short a{};
    int b{};
    double c{};
    short qq{}; // 注意：qq 在此定义
};

int main()
{
    std::cout << "The size of Foo1 is " << sizeof(Foo1) << '\n';
    std::cout << "The size of Foo2 is " << sizeof(Foo2) << '\n';

    return 0;
}
```

注意 ‘Foo1‘ 与 ‘Foo2‘ 拥有相同成员，唯一的区别是 ‘qq‘ 定义的顺序不同。程序打印：

```
The size of Foo1 is 16
The size of Foo2 is 24
```

10.9 Member selection with pointers and references

struct 指针的成员选择

然而，使用成员选择操作符（.）在 struct 指针上不起作用：

```
#include <iostream>

struct Employee
```

```
{
    int id{};
    int age{};
    double wage{};
};

int main()
{
    Employee joe{ 1, 34, 65000.0 };

    ++joe.age;
    joe.wage = 68000.0;

    Employee* ptr{ &joe };
    std::cout << ptr.id << '\n'; // 编译错误：不能在指针上使用操作符 \acode{.}

    return 0;
}
```

对于普通变量或者引用而言，可以直接访问对象。然而因为指针存储的是地址，因此需要先对指针进行解引用拿到对象后才能访问。所以访问 struct 指针的成员可以这样做：

```
#include <iostream>

struct Employee
{
    int id{};
    int age{};
    double wage{};
};

int main()
{
    Employee joe{ 1, 34, 65000.0 };

    ++joe.age;
    joe.wage = 68000.0;

    Employee* ptr{ &joe };
    std::cout << (*ptr).id << '\n'; // 不太合适，但是至少工作：先解引用再使用成员选择

    return 0;
}
```

然而这有点丑陋，特别是还需要用括号包住解引用操作符以便使解引用优先于成员选择符。为了有更清晰的语法，C++ 提供了**从指针选择成员符（->）**（有时称为 **arrow operator**）用于从指向对象的指针中选择成员：

```
#include <iostream>
```

```

struct Employee
{
    int id{};
    int age{};
    double wage{};
};

int main()
{
    Employee joe{ 1, 34, 65000.0 };

    ++joe.age;
    joe.wage = 68000.0;

    Employee* ptr{ &joe };
    std::cout << ptr->id << '\n'; // 更好：使用 -> 选择指向对象指针的成员

    return 0;
}
```

成员混合指针与非指针

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

struct Paw
{
    int claws{};
};

struct Animal
{
    std::string name{};
    Paw paw{};
};

int main()
{
    Animal puma{ "Puma", { 5 } };

    Animal* ptr{ &puma };

    // ptr 是指针，使用 ->
    // paw 非指针，使用 .

    std::cout << (ptr->paw).claws << '\n';

```

```
    return 0;
}
```

注意 `(ptr -> paw).claws` 的圆括号不是必须的，因为 `->` 和 `.` 两个符号都是从左至右计算的，加上括号可以增加可读性。

10.10 Class templates

类似于函数模板用于实例化函数，**类模板** class template 用于实例化 class 类型。

```
#include <iostream>

template <typename T>
struct Pair
{
    T first{};
    T second{};
};

int main()
{
    Pair<int> p1{ 5, 6 };        // 实例化 Pair<int> 并创建对象 p1
    std::cout << p1.first << ' ' << p1.second << '\n';

    Pair<double> p2{ 1.2, 3.4 }; // 实例化 Pair<double> 并创建对象 p2
    std::cout << p2.first << ' ' << p2.second << '\n';

    Pair<double> p3{ 7.8, 9.0 }; // 创建对象 p3 使用之前定义过的 Pair<double>
    std::cout << p3.first << ' ' << p3.second << '\n';

    return 0;
}
```

在函数中使用 class 模板

```
#include <iostream>

template <typename T>
struct Pair
{
    T first{};
    T second{};
};

template <typename T>
constexpr T max(Pair<T> p)
{
```

```
    return (p.first > p.second ? p.first : p.second);
}

int main()
{
    Pair<int> p1{ 5, 6 };
    std::cout << max<int>(p1) << " is larger\n"; // 显式调用 max<int>

    Pair<double> p2{ 1.2, 3.4 };
    std::cout << max(p2) << " is larger\n"; // 调用 max<double> 使用模板参数推导（推荐）

    return 0;
}
```

带有模板参数与非模板参数成员的 class 模板

```
template <typename T>
struct Foo
{
    T first{};    // 会被替换的 T 类型
    int second{}; // 永远是 int 类型
};
```

带有若干模板类型的 class 模板

```
#include <iostream>

template <typename T, typename U>
struct Pair
{
    T first{};
    U second{};
};

template <typename T, typename U>
void print(Pair<T, U> p)
{
    std::cout << '[' << p.first << ", " << p.second << ']';
}

int main()
{
    Pair<int, double> p1{ 1, 2.3 };
    Pair<double, int> p2{ 4.5, 6 };
    Pair<int, int> p3{ 7, 8 };

    print(p2);
```

```
23
24      return 0;
25  }
```

std::pair

因为 pair 数据类型非常常见，C++ 标准库中包含一个名为 `std::pair` （`<utility>` 头文件）的 class 模板：

```
1  #include <iostream>
2  #include <utility>
3
4  template <typename T, typename U>
5  void print(std::pair<T, U> p)
6  {
7      std::cout << '[' << p.first << ", " << p.second << ']';
8  }
9
10 int main()
11 {
12     std::pair<int, double> p1{ 1, 2.3 };
13     std::pair<double, int> p2{ 4.5, 6 };
14     std::pair<int, int> p3{ 7, 8 };
15
16     print(p2);
17
18     return 0;
19 }
```

在若干文件中使用 class 模板

pair.h:

```
1  #ifndef PAIR_H
2  #define PAIR_H
3
4  template <typename T>
5  struct Pair
6  {
7      T first{};
8      T second{};
9  };
10
11 template <typename T>
12 constexpr T max(Pair<T> p)
13 {
14     return (p.first > p.second ? p.first : p.second);
```

```
}

#endif
```

foo.cpp:

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

void foo()
{
    Pair<int> p1{ 1, 2 };
    std::cout << max(p1) << " is larger\n";
}
```

main.cpp:

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

void foo(); // 函数 foo() 的前向声明

int main()
{
    Pair<double> p2 { 3.4, 5.6 };
    std::cout << max(p2) << " is larger\n";

    foo();

    return 0;
}
```

10.11 Class template argument deduction (CTAD) and deduction guides

Class 模板参数推导（CTAD）（C++17）

从 C++17 开始，当一个对象从 class 模板中实例化时，编译器可以根据对象的初始化类型中推导模板类型（CTAD）：

```
#include <utility> // std::pair

int main()
{
    std::pair<int, int> p1{ 1, 2 }; // 显式指定 std::pair<int, int> (C++11 onward)
    std::pair p2{ 1, 2 };           // CTAD 用于推导 std::pair<int, int> (C++17)

    return 0;
}
```

CTAD 仅作用于没有模板参数时。因此以下两种做法都是错误的：

```cpp
#include <utility> // std::pair

int main()
{
    std::pair<> p1 { 1, 2 };    // 错误：太少的模板参数，两个参数都没有被推导
    std::pair<int> p2 { 3, 4 }; // 错误：太少的模板参数，第二个参数没有被推导

    return 0;
}
```

模板参数推导指南（C++17）

多数情况下，CTAD 都可以正常工作。然而在特定情况下，编译器可能会需要一些额外的帮助用于理解如何正确的推导出模板类型。
以下的代码在 C++17 中不能被编译：

```cpp
// 自定义 Pair 类型
template <typename T, typename U>
struct Pair
{
    T first{};
    U second{};
};

int main()
{
    Pair<int, int> p1{ 1, 2 }; // 可以：显式的指定模板参数
    Pair p2{ 1, 2 };           // C++17 中编译错误

    return 0;
}
```

这是因为在 C++17 中，CTAD 并不知道如何为聚合类型进行模板参数推导。为了解决这个问题，可以提供编译器一个**推导指南**，这样可以提示编译器如何在给定的 class 模板中进行正确的模板参数推导。

```cpp
template <typename T, typename U>
struct Pair
{
    T first{};
    U second{};
};

// 这里是 Pair 的推导指南
// Pair 对象通过参数类型 T 与 U 实例化成为 Pair<T, U>
template <typename T, typename U>
Pair(T, U) -> Pair<T, U>;

```

```
13 int main()
14 {
15     Pair<int, int> p1{ 1, 2 }; // 显式指定模板参数 Pair<int, int>
16     Pair p2{ 1, 2 };     // CTAD 推导出 Pair<int, int>
17 
18     return 0;
19 }
```

`Pair` class 的推导指南很简单，现在看一看它是如何工作的。

```
1 template <typename T, typename U>
2 Pair(T, U) -> Pair<T, U>;
```

首先，使用相同的模板类型定义 `Pair` 类。这很容易理解，因为如果推导指南告诉编译器如何为 `Pair<T, U>` 进行类型推导，需要同时定义 `T` 和 `U` 。其次，在箭头的右侧是帮助编译器推导的类型。这里就是希望编译器可以为对象推导出类型 `Pair<T, U>` 。最后再箭头的左侧告诉编译器什么类型的声明是它需要知道的。

11 Arrays, Strings, and Dynamic Allocation

11.1 Arrays

数组 array 是一种聚合类型允许用户通过单个标识符访问若干同样类型的变量。

```
int testScore[30]{}; // 分配 30 个整数变量在固定的数组
```

任何 array 变量的声明中，使用方括号（‘[]‘）告诉编译器这是一个数组变量（而不是普通变量），同时有多少个变量要分配（**数组长度** array length）。
上述例子声明了一个名为 testScore 并带有 30 长度的固定数组。**固定数组**的长度在编译时已知。
当 testScore 实例化后，30 个整数将被分配。
数组中每个变量被称为**元素**。元素没有独立的名称，而是通过**下标操作符** subscript operator（[]）以及**下标** subscript（或 index）来访问数组名称获取元素。这个过程被称为 **subscripting** 或 **indexing** 数字。

```
#include <iostream>

int main()
{
    int prime[5]{};
    prime[0] = 2;
    prime[1] = 3;
    prime[2] = 5;
    prime[3] = 7;
    prime[4] = 11;

    std::cout << "The lowest prime number is: " << prime[0] << '\n';
    std::cout << "The sum of the first 5 primes is: " << prime[0] + prime[1] + prime[2] + prime
     [3] + prime[4] << '\n';

    return 0;
}
```

最佳实践：显式实例化数组（即便元素类型是 self-initializing 的）。
规则：使用数组时请确保下标不要超出范围！

11.3 Arrays and loops

当一个循环用于访问每个元素时，通常被称为**遍历**整个数组。

```
constexpr int scores[]{ 84, 92, 76, 81, 56 };
constexpr int numStudents{ static_cast<int>(std::size(scores)) };
// const int numStudents{ sizeof(scores) / sizeof(scores[0]) }; // 当 C++17 不可用时请使用这种
    方式
int totalScore{ 0 };

```

```
// 使用循环计算总分数
for (int student{ 0 }; student < numStudents; ++student)
    totalScore += scores[student];

auto averageScore{ static_cast<double>(totalScore) / numStudents };
```

```
#include <iostream>
#include <iterator> // std::size

int main()
{
    constexpr int scores[]{ 84, 92, 76, 81, 56 };
    constexpr int numStudents{ static_cast<int>(std::size(scores)) };

    int maxScore{ 0 }; // 保存最大的分数
    for (int student{ 0 }; student < numStudents; ++student)
    {
        if (scores[student] > maxScore)
        {
            maxScore = scores[student];
        }
    }

    std::cout << "The best score was " << maxScore << '\n';

    return 0;
}
```

11.4 Sorting an array using selection sort

可以使用标准库中的 std::swap() 方法来交换两个元素。

```
#include <iostream>
#include <utility>

int main()
{
    int x{ 2 };
    int y{ 4 };
    std::cout << "Before swap: x = " << x << ", y = " << y << '\n';
    std::swap(x, y); // 将 x 的值与 y 的值进行交换
    std::cout << "After swap:  x = " << x << ", y = " << y << '\n';

    return 0;
}
```

C++ 中的选择排序

```cpp
#include <iostream>
#include <iterator>
#include <utility>

int main()
{
  int array[]{ 30, 50, 20, 10, 40 };
  constexpr int length{ static_cast<int>(std::size(array)) };

  // Step through each element of the array
  // (except the last one, which will already be sorted by the time we get there)
  for (int startIndex{ 0 }; startIndex < length - 1; ++startIndex)
  {
    // smallestIndex is the index of the smallest element we've encountered this iteration
    // Start by assuming the smallest element is the first element of this iteration
    int smallestIndex{ startIndex };

    // Then look for a smaller element in the rest of the array
    for (int currentIndex{ startIndex + 1 }; currentIndex < length; ++currentIndex)
    {
      // If we've found an element that is smaller than our previously found smallest
      if (array[currentIndex] < array[smallestIndex])
        // then keep track of it
        smallestIndex = currentIndex;
    }

    // smallestIndex is now the smallest element in the remaining array
                // swap our start element with our smallest element (this sorts it into the correct place)
    std::swap(array[startIndex], array[smallestIndex]);
  }

  // Now that the whole array is sorted, print our sorted array as proof it works
  for (int index{ 0 }; index < length; ++index)
    std::cout << array[index] << ' ';

  std::cout << '\n';

  return 0;
}
```

std::sort

因为数组排序很常见，C++ 标准库在 `<algorithm>` 中提供了 `std::sort` 函数。

```cpp
#include <algorithm> // for std::sort
#include <iostream>
#include <iterator> // for std::size

```

```
int main()
{
  int array[]{ 30, 50, 20, 10, 40 };

  std::sort(std::begin(array), std::end(array));

  for (int i{ 0 }; i < static_cast<int>(std::size(array)); ++i)
    std::cout << array[i] << ' ';

  std::cout << '\n';

  return 0;
}
```

11.8 Pointers and arrays

当一个固定数组用于一个表达式中时，固定数值则会 **decay**（被隐式转换）成为一个指向数组第一个元素的指针。

```
#include <iostream>

int main()
{
    int array[5]{ 9, 7, 5, 3, 1 };

    // 打印数组第一个元素的地址
    std::cout << "Element 0 has address: " << &array[0] << '\n';

    // 打印数组 decay 后的指针的值
    std::cout << "The array decays to a pointer holding address: " << array << '\n';


    return 0;
}
```

打印结果为：

```
Element 0 has address: 0042FD5C
The array decays to a pointer holding address: 0042FD5C
```

一个数组和一个数组的指针是完全相同的，这是 C++ 中一个常见的谬误，它们并不相同。上述的案例中，数组的类型是“int[5]”，并且它的“值”是数组元素本身。而数组的指针类型是“int”，其值为数组第一个元素的地址。

数组中所有的元素仍然可以通过指针进行访问，但是由数组类型派生出来的信息（例如数组的长度）不可被指针所访问。

不过实际上，大多数情况下，用户可以将固定数组视为指针。

```
int array[5]{ 9, 7, 5, 3, 1 };

// 数组解引用返回第一个元素
std::cout << *array; // 打印 9!

char name[]{ "Jason" }; // C-style 字符串（同样也是一个数组）
std::cout << *name << '\n'; // 打印 'J'
```

注意我们实际上并没有对数组自身解引用。数组（类型 `int[5]`）被隐式转换成了一个指针（类型 `int*`），解引用了指针获取了在其内存地址的值（即数组第一个元素的值）。

同样也可以让指针指向数组：

```
#include <iostream>

int main()
{
    int array[5]{ 9, 7, 5, 3, 1 };
    std::cout << *array << '\n'; // 打印 9

    int* ptr{ array };
    std::cout << *ptr << '\n'; // 打印 9

    return 0;
}
```

这是因为数组退化成为了一个 `int*` 类型的指针，新的指针（同样也是 `int*`）拥有同样的类型。

这里有一些例子来解释，固定数组与指针的不同点。

最主要的区别是再是用 sizeof() 操作符。对固定数组使用时，其返回整个数组的大小（数组长度 元素大小）；而对指针使用时，返回的是指针的大小。

```
#include <iostream>

int main()
{
    int array[5]{ 9, 7, 5, 3, 1 };

    std::cout << sizeof(array) << '\n'; // 打印 sizeof(int) * array length

    int* ptr{ array };
    std::cout << sizeof(ptr) << '\n'; // 打印指针大小

    return 0;
}
```

结果：

```
20
4
```

第二个区别在于使用 address-of 操作符（& ）。获取指针的地址返回的是指针变量的内存地址；获取数组地址返回的是整个数组的指针，该指针同样指向数组的第一个元素，但是类型信息是不同的。

```
#include <iostream>

int main()
{
    int array[5]{ 9, 7, 5, 3, 1 };
    std::cout << array << '\n';   // 类型 int[5]，打印 009DF9D4
    std::cout << &array << '\n';  // 类型 int(*)[5]，打印 009DF9D4

    std::cout << '\n';

    int* ptr{ array };
    std::cout << ptr << '\n';     // 类型 int*，打印 009DF9D4
    std::cout << &ptr << '\n';    // 类型 int**，打印 009DF9C8

    return 0;
}
// h/t to reader PacMan for this example
```

11.2 中提到过拷贝大数组是非常昂贵的，因此在传递数组至函数时 C++ 不会对数组进行拷贝，而是衰退成指针，该指针被传递：

```
#include <iostream>

void printSize(int* array)
{
    // 数组在这里被视为指针
    std::cout << sizeof(array) << '\n'; // 打印的是指针的大小，而不是数组的大小!
}

int main()
{
    int array[]{ 1, 1, 2, 3, 5, 8, 13, 21 };
    std::cout << sizeof(array) << '\n'; // 打印 sizeof(int) * array length

    printSize(array); // 数组参数在这里衰退成指针

    return 0;
}
```

注意，即使参数声明是固定数组，数组被隐式转换成指针照样发生：

```
#include <iostream>

// C++ 将隐式转换 array[] 至 *array
void printSize(int array[])
{
```

```
    // array 在这里被视为指针，而不是固定数组
    std::cout << sizeof(array) << '\n'; // 打印的是指针的大小，而不是数组的大小!
}

int main()
{
    int array[]{ 1, 1, 2, 3, 5, 8, 13, 21 };
    std::cout << sizeof(array) << '\n'; // will print sizeof(int) * array length

    printSize(array); // 数组参数在这里衰退成指针

    return 0;
}
```

上述例子中，C++ 隐式转换参数数组语法（`[]`）成为指针语法（`*`），这就意味着下面两个函数声明是完全相同的：

```
void printSize(int array[]);
void printSize(int* array);
```

最佳实践：推荐使用指针语法（`*`）。
最后值得注意的是当数组是结构体或者类里面的一部分的时候，传递整个结构体或类至函数时，它们是不会衰退成指针的。

11.9 Pointer arithmetic and array indexing

指针算数

C++ 允许对指针进行整数加减法的操作。如果 `ptr` 指向一个整数，那么 `ptr + 1` 则是 ptr 在内存中的下一个整数的地址；`ptr - 1` 则是上一个整数的地址。
注意 `ptr + 1` 不会返回 `ptr` 之后的**内存地址**，而是**下一个类型的对象**的内存地址。
当计算指针算数表达式的结果时，编译器总是会乘上对象的大小。这就叫做 **scaling**。

```
#include <iostream>

int main()
{
    int value{ 7 };
    int* ptr{ &value };

    std::cout << ptr << '\n';
    std::cout << ptr+1 << '\n';
    std::cout << ptr+2 << '\n';
    std::cout << ptr+3 << '\n';

    return 0;
}
```

打印类似于:

```
0012FF7C
0012FF80
0012FF84
0012FF88
```

而下面的代码:

```
#include <iostream>

int main()
{
    short value{ 7 };
    short* ptr{ &value };

    std::cout << ptr << '\n';
    std::cout << ptr+1 << '\n';
    std::cout << ptr+2 << '\n';
    std::cout << ptr+3 << '\n';

    return 0;
}
```

打印类似于:

```
0012FF7C
0012FF7E
0012FF80
0012FF82
```

指针算数，数组，以及 indexing 背后的魔法

```
#include <iostream>

int main()
{
     int array[]{ 9, 7, 5, 3, 1 };

     std::cout << &array[1] << '\n';  // 打印数组元素 1 的内存地址
     std::cout << array+1 << '\n';    // 打印数组指针 + 1 的内存地址

     std::cout << array[1] << '\n';   // 打印 7
     std::cout << *(array+1) << '\n'; // 打印 7（注意这里圆括号是必须的）

    return 0;
}
```

打印类似于:

```
0017FB80
0017FB80
7
7
```

也就是说，当编译器看到下标操作符（`[]`），则会转化数组成为指针加法！归纳就是，当 n 是整数时，`array[n]` 等同于 `*(array + n)`。下标操作符看起来更好看同时更加简单易用（不再需要记住圆括号）。

使用指针遍历数组

```
#include <iostream>
#include <iterator> // std::size

bool isVowel(char ch)
{
    switch (ch)
    {
    case 'A':
    case 'a':
    case 'E':
    case 'e':
    case 'I':
    case 'i':
    case 'O':
    case 'o':
    case 'U':
    case 'u':
        return true;
    default:
        return false;
    }
}

int main()
{
    char name[]{ "Mollie" };
    int arrayLength{ static_cast<int>(std::size(name)) };
    int numVowels{ 0 };

    // name + arrayLength 为数组中最后一个 char 的内存地址
    //
    for (char* ptr{ name }; ptr != (name + arrayLength); ++ptr)
    {
        if (isVowel(*ptr))
        {
            ++numVowels;
        }
```

```
    }

    std::cout << name << " has " << numVowels << " vowels.\n";

    return 0;
}
```

由于元素计算很常见，标准库提供了 `std::count_if` 方法：

```
#include <algorithm>
#include <iostream>
#include <iterator> // std::begin 与 std::end

bool isVowel(char ch)
{
    switch (ch)
    {
    case 'A':
    case 'a':
    case 'E':
    case 'e':
    case 'I':
    case 'i':
    case 'O':
    case 'o':
    case 'U':
    case 'u':
        return true;
    default:
        return false;
    }
}

int main()
{
    char name[]{ "Mollie" };

    // 遍历所有的元素并计数
    //
    auto numVowels{ std::count_if(std::begin(name), std::end(name), isVowel) };

    std::cout << name << " has " << numVowels << " vowels.\n";

    return 0;
}
```

`std::begin` 返回一个从第一个元素开始的迭代器（指针）；`std::end` 返回一个从最后一个元素开始的迭代器（指针），该迭代器仅作为标记，访问它会导致未定义行为，因为它并没有指向一个真实的元素。

`std::begin` 与 `std::end` 仅仅作用于已知大小的数组。如果数组衰退成指针，则可以手动计算其起始。

```
// nameLength 是数组的元素数
std::count_if(name, name + nameLength, isVowel)

// 不要使用以下代码。访问非法索引会导致未定义行为
// std::count_if(name, &name[nameLength], isVowel)
```

11.11 Dynamic memory allocation with new and delete

C++ 支持三种基本类型的内存分配：

- **静态内存分配** static memory allocation 发生在静态与全局变量。这些类型的变量内存仅在程序运行时分配一次，并持续程序整个过程。
- **自动内存分配** automatic memory allocation 发生在函数参数与本地变量。这些类型的变量内存在进入相关的代码块时进行分配，在代码块结束时释放。
- **动态内存分配** dynamic memory allocation 本文详解。

静态分配和自动分配有两点相同：

- 变量/数组的大小在编译期就必须知道
- 内存分配和内存释放是自动的（当变量实例化/销毁时）

动态内存分配是运行程序时向操作系统请求内存的一种方式。这些内存不是从有限的栈内存而来，而是由操作系统所管理的更大的**堆**内存提供。

动态分配单个变量

为了动态分配 * 单个 * 变量，需要使用 **new** 操作符：

```
new int; // 动态分配一个整数（并丢弃其值）
```

上述是向操作系统请求一个整数的内存。new 操作符创建使用了该内存的对象，并返回一个包含了其被分配的内存地址的指针。
通常来说会将返回值分配给用户自己的指针变量，这样便于之后访问：

```
int* ptr{ new int };
```

可以通过指针访问内存间接进行操作：

```
*ptr = 7; // 赋值 7 至内存地址
```

初始化一个动态分配的变量

当动态分配一个变量时，可以选择直接初始化或者标准初始化：

```
int* ptr1{ new int (5) };   // 使用直接初始化
int* ptr2{ new int { 6 } }; // 使用标准初始化
```

删除单个变量

当使用完动态分配的变量时，需要显式的告诉 C++ 释放内存。对于单个变量而已，使用 **delete** 操作符：

```
// 假设 ptr 之前已经被 new 操作符分配过了
delete ptr;     // 将 ptr 指向的内存还给操作系统
ptr = nullptr;  // 设置 ptr 成为空指针
```

悬垂指针

C++ 不会对内存释放或是删除指针的值做任何的保证。大多数情况下，操作系统返回的内存总是包含与之前其返回的值一致，同时指针会指向被内存释放的地址。
指向被内存释放地址的指针被称为**悬垂指针** dangling pointer。间接通过或删除悬垂指针将会发生未定义行为。

```
#include <iostream>

int main()
{
    int* ptr{ new int }; // 动态分配整数
    *ptr = 7; // 值放入内存地址

    delete ptr; // 返回操作系统的内存，ptr 现在是悬垂指针

    std::cout << *ptr; // 间接通过悬垂指针将会发生未定义行为
    delete ptr; // 再次进行内存释放将会发生未定义行为

    return 0;
}
```

释放内存可能会创建若干个悬垂指针：

```
#include <iostream>

int main()
{
    int* ptr{ new int{} }; // 动态分配整数
    int* otherPtr{ ptr };  // otherPtr 现在指向同一个内存地址

```

```
    delete ptr; // 返回操作系统的内存，ptr 和 otherPtr 现在都是悬垂指针
    ptr = nullptr; // ptr 设为 nullptr

    // 然而 otherPtr 仍然还是一个悬垂指针!

    return 0;
}
```

最佳实践：设置被删除的指针为空指针，除非该指针会立刻离开作用域。

操作符 new 可能会失败

当向操作系统请求内存时，在很罕见的情况下，操作系统有可能没有任何内存可以被分配。默认情况下，如果失败 *bad_alloc* 异常会被抛出。如果这个异常不能被正确的处理，那么程序则会被终结。

多数情况下，抛出异常是非预期的，因此有另一种方式的 new 可以在内存不能被分配时返回空指针。通过在 new 关键字与分配的类型之间增加常量 std::nothrow：

```
int* value { new (std::nothrow) int }; // 当整数分配失败时，值会被设为空指针
```

注意如果尝试间接通过该指针，未定义行为将出现（大概率程序崩溃）。因此最佳实践是检查所有的内存请求来确保它们成功了再去使用内存分配：

```
int* value { new (std::nothrow) int{} }; // 请求一个整数的内存
if (!value) // 处理返回 null 时的情况
{
    // 错误处理
    std::cerr << "Could not allocate memory\n";
}
```

空指针与动态内存分配

在处理动态内存分配时，空指针（设为 nullptr 的指针）实际上很有用。

```
// 如果 ptr 没有被分配，则进行分配
if (!ptr)
    ptr = new int;
```

删除空指针没有影响，因此不需要下面的代码：

```
if (ptr)
    delete ptr;
```

而是可以直接：

```
delete ptr;
```

内存泄漏

动态分配的内存会一直存在直到显式的释放内存或者程序结束（操作系统进行清理）。然而，用于保存动态分配地址的指针遵循着普通的作用域规则。这样的不匹配会导致有趣的问题：

```
void doSomething()
{
    int* ptr{ new int{} };
}
```

这个函数进行了整数的内存分配，但是永远没有被删除。因为指针变量仅仅是普通的变量，当函数结束时，ptr 离开作用域。正因为 ptr 是唯一保存动态分配地址的变量，当 ptr 被销毁时便不再有指向动态分配内存的引用。这就意味着程序“失去了”动态分配的内存。结果该动态分配的整数不能再被删除。

这被称为**内存泄漏**。内存泄漏发生在将内存还给操作系统之前，程序丢失了动态分配内存的地址，操作系统不能使用该内存，因为该内存被视为仍然在程序中使用。

尽管离开作用域会导致内存泄漏，还有其他一些方法也会导致内存泄漏：

```
int value = 5;
int* ptr{ new int{} }; // 分配内存
ptr = &value; // 丢失旧地址，导致内存泄漏
```

上述可以在重新赋值前删除指针进行修复：

```
int value{ 5 };
int* ptr{ new int{} }; // 分配内存
delete ptr;   // 将内存还给操作系统
ptr = &value; // 重新赋值
```

同样的通过双分配也会导致内存泄漏：

```
int* ptr{ new int{} };
ptr = new int{}; // 丢失旧地址，导致内存泄漏
```

11.12 Dynamically allocating arrays

除了动态分配单个值，也可以动态分配数组。不同于固定数组需要在编译期知道数组长度，动态分配一个数组可以在运行时选择长度。

使用 new 和 delete（通常为 new[] 与 delete[]）来进行数组的动态分配。

```
#include <iostream>

int main()
{
    std::cout << "Enter a positive integer: ";
    int length{};
    std::cin >> length;
```

```
8
9       int* array{ new int[length]{} }; // 使用 array new。注意这里的 length 不再需要是常量!
10
11      std::cout << "I just allocated an array of integers of length " << length << '\n';
12
13      array[0] = 5;
14
15      delete[] array; // 使用 array delete 释放内存
16
17      // 这里不需要设定 array 成为 nullptr/0，因为这里马上出作用域
18
19      return 0;
20  }
```

动态分配数组的 length 必须为可以被转换成 `std::size_t` 的类型。实际上使用 `int` 长度也是可以的，因为 `int` 会被转换成 `std::size_t` 。

初始化动态分配数组

```
1  int* array{ new int[length]{} };
```

数组重新设定长度

动态分配数组使得用户可以在分配是设定数组长度。然而 C++ 没有提供内置的重设已被分配的数组长度的方法。可以用别的方法来绕过这个限制，例如重新分配一个新的数组，拷贝所有元素，然后删除旧数组。然而这很容易导致错误，特别是元素类型是 class（因为有特殊的规则用于管理它们的创建）。
因此不推荐这么做。如果需要这样的功能，C++ 在标准库中提供了 `std::vector` 的可变大小的数组。

11.13 For-each loops

在 11.3 中使用了 *for* 循环来遍历数组中的所有元素：

```
1  #include <iostream>
2  #include <iterator> // std::size
3
4  int main()
5  {
6      constexpr int scores[]{ 84, 92, 76, 81, 56 };
7      constexpr int numStudents{ std::size(scores) };
8
9      int maxScore{ 0 };
10     for (int student{ 0 }; student < numStudents; ++student)
11     {
```

```
        if (scores[student] > maxScore)
        {
            maxScore = scores[student];
        }
    }

    std::cout << "The best score was " << maxScore << '\n';

    return 0;
}
```

有另一种简单且安全的循环被称为 **for-each** 循环（也被称为 **range-based for-loop**），可以遍历数组中的所有元素（或者其它 list-type 结果）。

For-each 循环

for-each 声明的语法类似于：

```
for (element_declaration : array)
   statement;
```

```
#include <iostream>

int main()
{
    constexpr int fibonacci[]{ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 };
    for (int number : fibonacci) // 遍历 fibonacci 数组
    {
       std::cout << number << ' '; // 访问数组元素
    }

    std::cout << '\n';

    return 0;
}
```

For each 循环以及 auto 关键字

由于元素声明需要与数组元素类型一致，这里使用 `auto` 关键字是一个好主意，让 C++ 帮助进行类型推导。

```
#include <iostream>

int main()
{
    constexpr int fibonacci[]{ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 };
    for (auto number : fibonacci) // 类型为 auto，number 的类型由数组进行推导
    {
```

```
        std::cout << number << ' ';
    }

    std::cout << '\n';

    return 0;
}
```

For each 循环以及引用

```
std::string array[]{ "peter", "likes", "frozen", "yogurt" };
for (auto element : array) // 元素将被拷贝
{
    std::cout << element << ' ';
}
```

这就意味着遍历到的每个数组元素都会被拷贝，这就非常的昂贵，大多数时候仅仅只需要引用原来的元素。幸运的是可以使用引用：

```
std::string array[]{ "peter", "likes", "frozen", "yogurt" };
for (auto& element: array) // & 符号可以使元素成为引用，阻止拷贝发生
{
    std::cout << element << ' ';
}
```

上述例子，元素会变成引用，另外任何对元素的修改都会影响被遍历的数组。因此有时候使用 `const` 引用意为只需要可读功能：

```
std::string array[]{ "peter", "likes", "frozen", "yogurt" };
for (const auto& element: array) // 元素现在是 const 引用
{
    std::cout << element << ' ';
}
```

最佳实践：在 for-each 循环元素声明中，如果元素不是基础类型，因为性能原因，推荐使用引用或者 `const` 引用。

For-each 不能作用于数组指针

由于需要遍历整个数组，for-each 需要知道数组的大小，这就意味着数组长度已知。因为数组会退化成指针而丢失大小，for-each 则不能工作！

```
#include <iostream>

int sumArray(const int array[]) // array 为指针
{
    int sum{ 0 };
```

```

    for (auto number : array) // 编译错误，array 的大小不可知
    {
        sum += number;
    }

    return sum;
}

int main()
{
     constexpr int array[]{ 9, 7, 5, 3, 1 };

     std::cout << sumArray(array) << '\n'; // array 在这里退化成指针

     return 0;
}
```

动态数组也因为同样的原因不能工作。

如何获取当前元素的索引?

For-each 循环*没有提供*直接获取数组当前元素索引的方法。这是因为 *for-each* 循环还可以用于很多不能直接索引的结构（例如链表）。

从 C++20 开始，range-based for-loops 可以使用**初始声明** init-statement，即与普通 for 循环一样。可以使用初始声明来手动创建索引计数从而不去污染 for-loop 中的函数。

初始声明的语法如下：

```
for (init-statement; element_declaration : array)
   statement;
```

```
#include <iostream>

int main()
{
    std::string names[]{ "Alex", "Betty", "Caroline", "Dave", "Emily" }; // 学生名字
    constexpr int scores[]{ 84, 92, 76, 81, 56 };
    int maxScore{ 0 };

    for (int i{ 0 }; auto score : scores) // i 是当前元素的索引
    {
        if (score > maxScore)
        {
            std::cout << names[i] << " beat the previous best score of " << maxScore << " by "
    << (score - maxScore) << " points!\n";
            maxScore = score;
        }

```

```
        ++i;
    }

    std::cout << "The best score was " << maxScore << '\n';

    return 0;
}
```

打印:

```
Alex beat the previous best score of 0 by 84 points!
Betty beat the previous best score of 84 by 8 points!
The best score was 92
```

需要注意的是如果有 `continue` 在循环内，`++i` 会被跳过而导致非预期结果，因此需要确保 `i` 在触发 `continue` 之前自增。
在 C++20 之前，变量 `i` 需要在循环外部声明，这可能在函数内部有另一个 `i` 的时候导致命名冲突。

11.14 Void pointers

void 指针，又称通用指针，是一种特殊类型的指针可以用作于指向任何数据类型的对象!

```
void* ptr; // ptr 为 void pointer
```

void 指针可以指向任何数据类型的对象:

```
int nValue;
float fValue;

struct Something
{
    int n;
    float f;
};

Something sValue;

void* ptr;
ptr = &nValue; // 有效
ptr = &fValue; // 有效
ptr = &sValue; // 有效
```

然而因为 void 指针不知道它所指向的对象类型，解引用 void 指针是不合法的。相反，void 指针必须先强制转换成其它类型的指针才可以进行解引用操作。

```
int value{ 5 };
void* voidPtr{ &value };

```

```
// std::cout << *voidPtr << '\n'; // 不合法：void 指针解引用

int* intPtr{ static_cast<int*>(voidPtr) }; // 强制转换为 int 指针

std::cout << *intPtr << '\n'; // 解引用
```

接下来显而易见的问题就是：如果一个 void 指针不知道其指向的是什么，如何进行强制转换呢？根本上而言，用户需要一直进行追踪。

```
#include <iostream>
#include <cassert>

enum class Type
{
    tInt, // 注意：这里不能使用 int，因为是关键字
    tFloat,
    tCString
};

void printValue(void* ptr, Type type)
{
    switch (type)
    {
    case Type::tInt:
        std::cout << *static_cast<int*>(ptr) << '\n'; // 强制转换 int 指针并解引用
        break;
    case Type::tFloat:
        std::cout << *static_cast<float*>(ptr) << '\n'; // 强制转换 float 指针并解引用
        break;
    case Type::tCString:
        std::cout << static_cast<char*>(ptr) << '\n'; // 强制转换 char 指针（无解引用）
        // std::cout 将 char* 视为 C-style 字符串
        // 如果在这里解引用了，则只会打印单个字符
        break;
    default:
        assert(false && "type not found");
        break;
    }
}

int main()
{
    int nValue{ 5 };
    float fValue{ 7.5f };
    char szValue[]{ "Mollie" };

    printValue(&nValue, Type::tInt);
    printValue(&fValue, Type::tFloat);
    printValue(szValue, Type::tCString);
```

```
41
42     return 0;
43 }
```

混合 void 指针

void 指针可以设置空值：

```
1 void* ptr{ nullptr };
```

最佳实践：通常而言，除了必须使用时，应该避免使用 void 指针。

11.16 An introduction to std::array

之前的章节谈到过固定与动态数组，尽管它们都内置于 C++，它们都有缺陷：固定数组衰退成为指针，丢失数组长度信息；动态数组内存释放的麻烦，以及重置数组长度所面临的风险。为了解决这些问题，C++ 标准库包含了功能性的数组：`std::array` 以及 `std::vector` 。本章讲解前者，下一章节讲解后者。

简介 std::array

`std::array` 提供固定数组的功能且传递给函数式不会退化，其定义于 `<array>` 头文件中的 `std` 命名空间。

```
1 #include <array>
2
3 std::array<int, 3> myArray; // 声明一个长度为 3 的整数数组
```

可以使用 initializer lists 或者 list initialization 来进行 `std::array` 初始化：

```
1 std::array<int, 5> myArray = { 9, 7, 5, 3, 1 }; // initializer list
2 std::array<int, 5> myArray2 { 9, 7, 5, 3, 1 };  // list initialization
```

有别于内建的固定数组，`std::array` 不能省略数组长度：

```
1 std::array<int, > myArray { 9, 7, 5, 3, 1 };  // 非法：必须提供数组长度
2 std::array<int> myArray { 9, 7, 5, 3, 1 };    // 非法：必须提供数组长度
```

然而从 C++17 开始，允许省略类型和长度。它们可以被同时省略，但是不能仅省略其中之一，同时数组必须是显式的初始化：

```
1 std::array myArray { 9, 7, 5, 3, 1 }; // 类型被推导成 std::array<int, 5>
2 std::array myArray { 9.7, 7.31 };     // 类型被推导成 std::array<double, 2>
```

从 C++20 开始，可以指定元素类型同时省略数组长度：

```
1 auto myArray1 { std::to_array<int, 5>({ 9, 7, 5, 3, 1 }) }; // 指定类型与长度
2 auto myArray2 { std::to_array<int>({ 9, 7, 5, 3, 1 }) };    // 仅指定类型，推导长度
3 auto myArray3 { std::to_array({ 9, 7, 5, 3, 1 }) };         // 推导类型与长度
```

不幸的是，std::to_array 比起直接创建 std::array 而言更昂贵，因为它实际上会从 C-style 数组中拷贝所有元素到 std::array 。正因如此，应该避免使用 std::to_array ，特别是创建很多次的环境下（例如在一个循环中）。

也可以对数组进行赋值：

```
std::array<int, 5> myArray;
myArray = { 0, 1, 2, 3, 4 };    // okay
myArray = { 9, 8, 7 };          // okay, 元素 3 和 4 设置为零!
myArray = { 0, 1, 2, 3, 4, 5 }; // 不允许，超出范围
```

使用下标操作符访问 std::array 的值：

```
std::cout << myArray[1] << '\n';
myArray[2] = 6;
```

与内建的固定数组一样，下标操作符不会有任何的范围检查，如果提供了无效的索引，未定义行为会发生。

std::array 提供了另一种元素访问的方式（at() 函数）带有范围检查（运行时）：

```
std::array myArray { 9, 7, 5, 3, 1 };
myArray.at(1) = 6;  // 数组元素 1 有效，设置第二个元素为 6
myArray.at(9) = 10; // 数组元素 9 无效，抛出运行时异常
```

长度与排序

size() 函数可以用于获取 std::array 的长度：

```
std::array myArray { 9.0, 7.2, 5.4, 3.6, 1.8 };
std::cout << "length: " << myArray.size() << '\n';
```

因为当传递给函数时 std::array 不会衰退成指针，size() 函数可用：

```
#include <array>
#include <iostream>

void printLength(const std::array<double, 5>& myArray)
{
    std::cout << "length: " << myArray.size() << '\n';
}

int main()
{
    std::array myArray { 9.0, 7.2, 5.4, 3.6, 1.8 };

    printLength(myArray);

    return 0;
}
```

注意标准库使用“size”意为数组长度 – 不要与 `sizeof()` 固定数组搞混淆（即返回数组的内存大小）。这里面的命名并不一致。

最佳实践：总是传递 `std::array` 的引用，或者 `const` 引用。

因为长度总是已知的，range-based for-loops 也可以用于 `std::array`：

```
std::array myArray{ 9, 7, 5, 3, 1 };

for (int element : myArray)
    std::cout << element << ' ';
```

可以使用 `std::sort` 对 `std::array` 进行排序，位于 `<algorithm>` 头文件：

```
#include <algorithm> // for std::sort
#include <array>
#include <iostream>

int main()
{
    std::array myArray { 7, 3, 1, 9, 5 };
    std::sort(myArray.begin(), myArray.end());       // 前向排序
    // std::sort(myArray.rbegin(), myArray.rend()); // 后向排序

    for (int element : myArray)
        std::cout << element << ' ';

    std::cout << '\n';

    return 0;
}
```

传递不同长度的 std::array 给函数

```
#include <array>
#include <cstddef>
#include <iostream>

// printArray 是一个模板函数
template <typename T, std::size_t size> // 模板化元素类型与长度
void printArray(const std::array<T, size>& myArray)
{
    for (auto element : myArray)
        std::cout << element << ' ';
    std::cout << '\n';
}

int main()
{
    std::array myArray5{ 9.0, 7.2, 5.4, 3.6, 1.8 };
```

```
17      printArray(myArray5);
18
19      std::array myArray7{ 9.0, 7.2, 5.4, 3.6, 1.8, 1.2, 0.7 };
20      printArray(myArray7);
21
22      return 0;
23  }
```

通过 size_type 手动索引 std::array

```
1   #include <array>
2   #include <iostream>
3
4   int main()
5   {
6       std::array myArray { 7, 3, 1, 9, 5 };
7
8       // std::array<int, 5>::size_type 是 size() 的返回值!
9       for (std::array<int, 5>::size_type i{ 0 }; i < myArray.size(); ++i)
10          std::cout << myArray[i] << ' ';
11
12      std::cout << '\n';
13
14      return 0;
15  }
```

更好的可读性:

```
1   #include <array>
2   #include <cstddef> // std::size_t
3   #include <iostream>
4
5   int main()
6   {
7       std::array myArray { 7, 3, 1, 9, 5 };
8
9       for (std::size_t i{ 0 }; i < myArray.size(); ++i)
10          std::cout << myArray[i] << ' ';
11
12      std::cout << '\n';
13
14      return 0;
15  }
```

更好的方式是避免对 `std::array` 进行手动索引，而是用 range-based for-loops (或迭代器)。

结构体数组

```cpp
#include <array>
#include <iostream>

struct House
{
    int number{};
    int stories{};
    int roomsPerStory{};
};

int main()
{
    std::array<House, 3> houses{};

    houses[0] = { 13, 4, 30 };
    houses[1] = { 14, 3, 10 };
    houses[2] = { 15, 3, 40 };

    for (const auto& house : houses)
    {
        std::cout << "House number " << house.number
                  << " has " << (house.stories * house.roomsPerStory)
                  << " rooms\n";
    }

    return 0;
}
```

初始化：

```cpp
// 与预期一致
std::array<House, 3> houses { // houses 的初始化器
    { // 额外一层花括号用于初始化位于 std::array 结构体内部的 C-style 的数组成员
        { 13, 4, 30 }, // initializer for array element 0
        { 14, 3, 10 }, // initializer for array element 1
        { 15, 3, 40 }, // initializer for array element 2
     }
};
```

11.17 An introduction to std::vector

上一章节介绍了 `std::array` 提供了 C++ 内建固定数组的功能以及更安全和使用的方式。
类似的，C++ 标准库提供了更安全和便利的动态数组的功能，名为 `std::vector` 。
不同于与固定数组类似的 `std::array` ， `std::vector` 拥有额外的能力。

简介 std::vector

```cpp
#include <vector>

// 不需要声明长度
std::vector<int> array;
std::vector<int> array2 = { 9, 7, 5, 3, 1 };  // 使用 initializer list 初始化数组 (before C++11)
std::vector<int> array3 { 9, 7, 5, 3, 1 };    // 使用 uniform initialization 初始化数组

// 如 std::array，C++17 开始可以省略类型
std::vector array4 { 9, 7, 5, 3, 1 }; // 推导成 std::vector<int>
```

与 `std::array` 类似，可以通过下标访问也可以通过 `at()` 函数访问：

```cpp
array[6] = 2;     // 无边界检查
array.at(7) = 3;  // 有边界检查
```

自清理防止内存泄漏

当向量变量离开作用域时，会自动释放其内存。

```cpp
void doSomething(bool earlyExit)
{
    int* array{ new int[5] { 9, 7, 5, 3, 1 } }; // 通过 new 分配内存

    if (earlyExit)
        return; // 退出函数，但是没有调用内存释放

    // 做一些事情

    delete[] array; // 永远没有被调用
}
```

然而如果 `array` 是 `std::vector` 的话内存泄漏便不会发生，这是因为一旦 `array` 离开作用域，内存就会立刻被释放（无论函数是否提前退出），这就使得 `std::vector` 更加的安全。

Vectors 记忆其长度

不同于内建的动态数组，不知道其自身的长度，std::vector 一直追踪其自身长度。可以通过 `size()` 函数查询向量的长度：

```cpp
#include <iostream>
#include <vector>

void printLength(const std::vector<int>& array)
{
    std::cout << "The length is: " << array.size() << '\n';
}

```

```
int main()
{
    std::vector array { 9, 7, 5, 3, 1 };
    printLength(array);

    std::vector<int> empty {};
    printLength(empty);

    return 0;
}
```

如同 `std::array` ， `size()` 返回一个嵌套类型 `size_type` ，为一个非负整数。

修改向量的长度

变化内建的动态数组是非常复杂的，而 `std::vector` 则非常简单，只需要调用 `resize()` 函数。

```
#include <iostream>
#include <vector>

int main()
{
    std::vector array { 0, 1, 2 };
    array.resize(5); // 设置长度为 5

    std::cout << "The length is: " << array.size() << '\n';

    for (int i : array)
        std::cout << i << ' ';

    std::cout << '\n';

    return 0;
}
```

打印:

```
The length is: 5
0 1 2 0 0
```

向量也可以变小:

```
#include <vector>
#include <iostream>

int main()
{
    std::vector array { 0, 1, 2, 3, 4 };
    array.resize(3); // 设置长度为 3
```

```
8
9       std::cout << "The length is: " << array.size() << '\n';
10
11      for (int i : array)
12          std::cout << i << ' ';
13
14      std::cout << '\n';
15
16      return 0;
17  }
```

打印:

```
1  The length is: 3
2  0 1 2
```

初始化特定大小的向量

```
1  #include <iostream>
2  #include <vector>
3
4  int main()
5  {
6      // 使用直接初始化，可以创建一个有 5 个元素的向量，每个元素都为 0。
7      // 如果使用的是大括号初始化，向量拥有 1 个为 5 的元素。
8      std::vector<int> array(5);
9
10     std::cout << "The length is: " << array.size() << '\n';
11
12     for (int i : array)
13         std::cout << i << ' ';
14
15     std::cout << '\n';
16
17     return 0;
18 }
```

11.18 Introduction to iterators

遍历整个数组（或者其他结构）的数据在编程中非常常见。现阶段覆盖了多种方式：通过循环以及索引（`for-loops` 以及 `while lopps`），通过指针和指针算数，以及通过 `range-based for-loops`：

```
1  #include <array>
2  #include <cstddef>
3  #include <iostream>
4
```

```cpp
int main()
{
    // C++17 中，变量 data 的类型被推导为 std::array<int, 7>
    // 如果报编译错误，详见下方的警告
    std::array data{ 0, 1, 2, 3, 4, 5, 6 };
    std::size_t length{ std::size(data) };

    // 带有显式索引的 while 循环
    std::size_t index{ 0 };
    while (index < length)
    {
        std::cout << data[index] << ' ';
        ++index;
    }
    std::cout << '\n';

    // 带有显式索引的 for 循环
    for (index = 0; index < length; ++index)
    {
        std::cout << data[index] << ' ';
    }
    std::cout << '\n';

    // 带有指针的 for 循环（注意：ptr 不能为 const，因为需要增加它）
    for (auto ptr{ &data[0] }; ptr != (&data[0] + length); ++ptr)
    {
        std::cout << *ptr << ' ';
    }
    std::cout << '\n';

    // ranged-based 的 for 循环
    for (int i : data)
    {
        std::cout << i << ' ';
    }
    std::cout << '\n';

    return 0;
}
```

警告：上述例子使用了一个 C++17 的特性名为 `class template argument deduction` 用于从模版变量的初始化中推导模版参数。例子中，当编译器看到 `std::array data{0, 1, 2, 3, 4, 5, 6};` 时，会推导成 `std::array<int, 7> data {...}` 。

迭代器

迭代器 iterator 是一个设计用来穿越整个容器的对象（例如，数组中的值，或者字符串中的字符），提供了访问每个元素的能力。
一个容器可能会提供不同种类的迭代器。例如，一个数组可以提供一个前向迭代器以及一个反方向的后向迭代器。
一旦合适类型的迭代器被创建，程序员便可以使用迭代器所提供的接口来穿越并访问元素，而不再需要担心使用哪种方式或者数据是如何存储于容器中的。因为 C++ 迭代器通常使用相同的接口（操作符 ++ 来移动至下一个元素）并访问（操作符 来访问当前元素），用户可以使用始终如一的方法来遍历不同类型的容器。

指针作为迭代器

最简单的迭代器类型就是一个指针，（使用指针算法）作用于有序的内存：

```cpp
#include <array>
#include <iostream>

int main()
{
    std::array data{ 0, 1, 2, 3, 4, 5, 6 };

    auto begin{ &data[0] };
    // 注意指向超出了最后一个元素
    auto end{ begin + std::size(data) };

    // 带有指针的 for-loop
    for (auto ptr{ begin }; ptr != end; ++ptr) // ++ 用于移动至下一个元素
    {
        std::cout << *ptr << ' '; // 解引用获取当前元素的值
    }
    std::cout << '\n';

    return 0;
}
```

标准库迭代器

遍历是常见的操作，因此所有的标准库容器都直接提供了遍历的支持。不需要用户计算起始以及终止点，可以直接通过函数 `begin()` 与 `end()` 来获取起始与终止点：

```cpp
#include <array>
#include <iostream>

int main()
```

```
{
    std::array array{ 1, 2, 3 };

    // 向数组请求起始与终止点（通过各自的成员函数）
    auto begin{ array.begin() };
    auto end{ array.end() };

    for (auto p{ begin }; p != end; ++p) // ++ 移动至下一个元素
    {
        std::cout << *p << ' '; // 解引用获取当前元素的值
    }
    std::cout << '\n';

    return 0;
}
```

`iterator` 头文件同样也包含了泛型方法（`std::begin` 与 `std::end`）：

```
#include <array>
#include <iostream>
#include <iterator> // std::begin 与 std::end

int main()
{
    std::array array{ 1, 2, 3 };

    // 使用 std::begin 以及 std::end 来获取起始点以及终止点
    auto begin{ std::begin(array) };
    auto end{ std::end(array) };

    for (auto p{ begin }; p != end; ++p) // ++ 移动至下一个元素
    {
        std::cout << *p << ' '; // 解引用获取当前元素的值
    }
    std::cout << '\n';

    return 0;
}
```

现在暂时不用担心迭代器的类型，之后的章节中会详细讲解。这里的关键点在于迭代器处理了遍历容器的所有细节。用户仅需要做四件事情：起始点，终止点，操作符 ++ 移动迭代器至下一个元素（或者结束），以及操作符 用于获取当前值。

迭代器失效（悬垂迭代器）

类似于指针已经引用，如果元素已经被遍历过了，迭代器也可以成为“悬垂”。当这种情况发生时可以说改迭代器已经**失效** invalidated 了。访问单个迭代器会导致未定义行为。

```
#include <iostream>
```

```cpp
#include <vector>

int main()
{
    std::vector v{ 1, 2, 3, 4, 5, 6, 7 };

    auto it{ v.begin() };

    ++it; // 移动至下一个元素
    std::cout << *it << '\n'; // ok: 打印 2

    v.erase(it); // 擦除现在正在遍历的元素

    // erase() 使迭代器擦除元素（以及其后面的元素）
    // 因此迭代器 "it" 不再有效

    ++it; // 未定义行为
    std::cout << *it << '\n'; // 未定义行为

    return 0;
}
```

11.19 Introduction to standard library algorithms

由 C++ 算数标准库提供的功能主要分为三种类型：

- **Inspectors** – 用于观察（但不修改）容器中的数据。例如查询以及计数。
- **Mutators** – 用于修改容器中的数据。例如排序以及乱序。
- **Facilitators** – 用于由数据成员生成结果。例如乘法，或者以何种形式的方式排序。

使用 std::find 通过值寻找元素

```cpp
#include <algorithm>
#include <array>
#include <iostream>

int main()
{
    std::array arr{ 13, 90, 99, 5, 40, 80 };

    std::cout << "Enter a value to search for and replace with: ";
    int search{};
    int replace{};
    std::cin >> search >> replace;

```

```
    // 省略输入检查

    // std::find 返回一个迭代器，指向找到的元素（或是容器的末尾）
    // 用一个变量存储它，使用类型推导做该迭代器的类型（因为不关心）
    auto found{ std::find(arr.begin(), arr.end(), search) };

    // 算法在找不到目标的情况下返回迭代器的末尾。
    // 使用成员函数 end() 访问
    if (found == arr.end())
    {
        std::cout << "Could not find " << search << '\n';
    }
    else
    {
        // 替换值
        *found = replace;
    }

    for (int i : arr)
    {
        std::cout << i << ' ';
    }

    std::cout << '\n';

    return 0;
}
```

使用 std::find_if 寻找匹配符合某种条件的元素

```
#include <algorithm>
#include <array>
#include <iostream>
#include <string_view>

// 函数将返回 true 如果元素匹配
bool containsNut(std::string_view str)
{
    // 如果没有找到子字符串 std::string_view::find 返回 std::string_view::npos
    // 否则返回子字符串出现的索引。
    return (str.find("nut") != std::string_view::npos);
}

int main()
{
    std::array<std::string_view, 4> arr{ "apple", "banana", "walnut", "lemon" };

    // 检索数组查看是否有元素包含 “nut” 子字符串
```

```
    auto found{ std::find_if(arr.begin(), arr.end(), containsNut) };

    if (found == arr.end())
    {
        std::cout << "No nuts\n";
    }
    else
    {
        std::cout << "Found " << *found << '\n';
    }

    return 0;
}
```

使用 std::count 与 std::count_if 统计出现次数

`std::count` 统计所有出现的元素；`std::count_if` 统计所有符合条件的元素。

```
#include <algorithm>
#include <array>
#include <iostream>
#include <string_view>

bool containsNut(std::string_view str)
{
    return (str.find("nut") != std::string_view::npos);
}

int main()
{
    std::array<std::string_view, 5> arr{ "apple", "banana", "walnut", "lemon", "peanut" };

    auto nuts{ std::count_if(arr.begin(), arr.end(), containsNut) };

    std::cout << "Counted " << nuts << " nut(s)\n";

    return 0;
}
```

使用 std::sort 进行自定义排序

`std::sort` 可以接收一个函数作为第三个参数用于实现自定义排序。该函数接收两个用作比较的参数，如果第一个参数排序在第二个参数之前返回 true。默认情况下 `std::sort` 进行的是升序排序。

```
#include <algorithm>
#include <array>
```

```cpp
#include <iostream>

bool greater(int a, int b)
{
    // 排序 @a 在 @b 之前，如果 @a 大于 @b
    return (a > b);
}

int main()
{
    std::array arr{ 13, 90, 99, 5, 40, 80 };

    // 传递 greater 函数给 std::sort
    std::sort(arr.begin(), arr.end(), greater);

    for (int i : arr)
    {
        std::cout << i << ' ';
    }

    std::cout << '\n';

    return 0;
}
```

使用 std::for_each 对一个容器中所有元素操作

`std::for_each` 接收一个列表作为输入，并应用自定义函数于每个元素。

```cpp
#include <algorithm>
#include <array>
#include <iostream>

void doubleNumber(int& i)
{
    i *= 2;
}

int main()
{
    std::array arr{ 1, 2, 3, 4 };

    std::for_each(arr.begin(), arr.end(), doubleNumber);

    for (int i : arr)
    {
        std::cout << i << ' ';
    }

```

```
21      std::cout << '\n';
22
23      return 0;
24  }
```

执行顺序

注意标准库中的大部分算法并不保证执行的顺序。这类的算法，只关心用户传递的函数，并不假设特定顺序，因为不同的编译器顺序的调用可能不同。

下列算法保证顺序执行：`std::for_each`，`std::copy`，`std::copy_backward`，`std::move` 与 `std::move_backward`。

最佳实践：除非有特别的指出，不要假设标准库的算法能确保执行的顺序。

12 Functions

12.1 Function Pointers

创建一个非 const 函数指针的语法：

```
// fcnPtr 是一个无入参并返回整数的函数指针
int (*fcnPtr)();
```

fcnPtr 可以指向任何匹配其类型的函数。
*fcnPtr 的圆括号是必须的，因为 `int* fcnPtr()` 会被解释为一个无入参并返回整数且名为 fcnPtr 函数的前向声明。
如果是一个 const 函数指针，只需要将 const 放在星号后：

```
int (*const fcnPtr)();
```

如果把 const 放在 int 之前，则会解释为指向的函数其返回值为 const int 类型。

赋值函数指针

函数指针可以通过函数进行初始化（同时非 const 函数指针还可以被赋值函数）。

```
int foo()
{
    return 5;
}

int goo()
{
    return 6;
}

int main()
{
    int (*fcnPtr)(){ &foo };  // fcnPtr 指向函数 foo
    fcnPtr = &goo;            // fcnPtr 现在指向了函数 goo

    return 0;
}
```

注意函数指针的类型（入参与返回类型）必须匹配函数的类型：

```
// 函数原型
int foo();
double goo();
int hoo(int x);

// 函数指针赋值
int (*fcnPtr1)(){ &foo };     // 可以
```

```
int (*fcnPtr2)(){ &goo };      // 错误 -- 返回类型不匹配!
double (*fcnPtr4)(){ &goo };   // 可以
fcnPtr1 = &hoo;                // 错误 -- fcnPtr1 没有入参，但是 hoo() 有入参
int (*fcnPtr3)(int){ &hoo };   // 可以
```

不同于基础类型，如果需要 C++ 将隐式转换一个函数成为一个函数指针（因此用户不需要使用 address-of 操作符（&）来获取函数地址）。然而，并不会隐式转换函数指针成为 void 指针，或者反过来。

函数指针可以被初始化或赋值 nullptr：

```
int (*fcnptr)() { nullptr };  // 可以
```

使用函数指针调用函数

首先是通过显式解引用：

```
int foo(int x)
{
    return x;
}

int main()
{
    int (*fcnPtr)(int){ &foo }; // 使用函数 foo 初始化 fcnPtr
    (*fcnPtr)(5); // 通过 fcnPtr 调用函数 foo(5)

    return 0;
}
```

第二种方法是通过隐式解引用：

```
int foo(int x)
{
    return x;
}

int main()
{
    int (*fcnPtr)(int){ &foo }; // 使用函数 foo 初始化 fcnPtr
    fcnPtr(5); // 通过 fcnPtr 调用函数 foo(5)

    return 0;
}
```

可以看到隐式解引用的方法与普通函数调用一致，因为普通函数名称就是函数的指针！

有一个有趣的地方值得注意：默认参数并不会通过函数指针调用而工作。默认参数在编译时决定（也就是说在不提供参数时，编译器在编译时替换为默认参数），而函数指针作用于运行时。因此在调用函数指针时，默认参数不能被决定。这种情况下，用户必须显式传递任何默认参数。

另外注意因为函数指针可以被设为 nullptr，那么在调用之前进行断言或者条件测试是一个好主意。

```
int foo(int x)
{
    return x;
}

int main()
{
    int (*fcnPtr)(int){ &foo }; // 使用函数 foo 初始化 fcnPtr
    if (fcnPtr) // 确保 fcnPtr 不是空指针
        fcnPtr(5); // 否则这会导致未定义行为

    return 0;
}
```

函数作为参数传递至其它函数

函数指针一个最有用的地方在于它可以作为参数传递进其它函数中。作为参数的函数有时候会被称为**回调函数** callback functions。
这里是之前章节中的选择排序：

```
#include <utility> // std::swap

void SelectionSort(int* array, int size)
{
    // 遍历数组中的每一个元素
    for (int startIndex{ 0 }; startIndex < (size - 1); ++startIndex)
    {
        // 最小数值的索引
        int smallestIndex{ startIndex };

        // 查询数组中剩下元素的最小值
        for (int currentIndex{ startIndex + 1 }; currentIndex < size; ++currentIndex)
        {
            // 如果当前元素小于之前找到的元素（由 startIndex + 1 开始）
            if (array[smallestIndex] > array[currentIndex]) // 比较在这里完成
            {
                // 本次遍历中最新的最小值
                smallestIndex = currentIndex;
            }
        }

        // 交换起始元素与最小的元素
        std::swap(array[startIndex], array[smallestIndex]);
    }
}
```

现在替换比较函数，比较函数可以是这样：

```
bool ascending(int x, int y)
{
    return x > y; // 如果第一个元素大于第二个元素则进行交换
}
```

那么就可以这样替换：

```
#include <utility>

void SelectionSort(int* array, int size)
{
    for (int startIndex{ 0 }; startIndex < (size - 1); ++startIndex)
    {
        int smallestIndex{ startIndex };

        for (int currentIndex{ startIndex + 1 }; currentIndex < size; ++currentIndex)
        {
            if (ascending(array[smallestIndex], array[currentIndex]))
            {
                smallestIndex = currentIndex;
            }
        }

        std::swap(array[startIndex], array[smallestIndex]);
    }
}
```

函数指针类似于：

```
bool (*comparisonFcn)(int, int);
```

最终完整的代码：

```
#include <utility> // std::swap
#include <iostream>

// 注意用户自定义比较函数位于第三个参数
void selectionSort(int* array, int size, bool (*comparisonFcn)(int, int)) {
  // 遍历数组中的每一个元素
  for (int startIndex{0}; startIndex < (size - 1); ++startIndex) {
    // 最小/大数值的索引
    int bestIndex{startIndex};

    // 查询数组中剩下元素的最小/大值
    for (int currentIndex{startIndex + 1}; currentIndex < size; ++currentIndex) {
      // 如果当前元素小于/大于之前找到的元素（由 startIndex + 1 开始）
      if (comparisonFcn(array[bestIndex], array[currentIndex])) // 比较在这里完成
      {
        // 本次遍历中最新的最小/大值
        bestIndex = currentIndex;
```

```
18        }
19    }
20
21    // 交换起始元素与最小/大的元素
22    std::swap(array[startIndex], array[bestIndex]);
23  }
24}
25
26// 升序的比较函数
27bool ascending(int x, int y)
28{
29    return x > y;
30}
31
32// 降序的比较函数
33bool descending(int x, int y)
34{
35    return x < y;
36}
37
38// 用于打印元素
39void printArray(int* array, int size)
40{
41    for (int index{ 0 }; index < size; ++index)
42    {
43        std::cout << array[index] << ' ';
44    }
45
46    std::cout << '\n';
47}
48
49int main()
50{
51    int array[9]{ 3, 7, 9, 5, 6, 1, 8, 2, 4 };
52
53    // 根据 descending() 进行降序排序
54    selectionSort(array, 9, descending);
55    printArray(array, 9);
56
57    // 根据 ascending() 进行升序排序
58    selectionSort(array, 9, ascending);
59    printArray(array, 9);
60
61    return 0;
62}
```

让函数指针拥有类型别名

函数指针的语法非常的丑陋，然而可以使用类型别名使其看起来更像是普通变量：

```
using ValidateFunction = bool(*)(int, int);
```

那么不需要再像这样的使用：

```
bool validate(int x, int y, bool (*fcnPtr)(int, int)); // 丑陋
```

可以这么做：

```
bool validate(int x, int y, ValidateFunction pfcn) // 简洁
```

使用 std::function

另一种定义并存储函数指针的方式是使用 `std::function`，其位于 `<functional>` 头文件。

```
#include <functional>

// std::function 方法返回布尔值且接受两个整数参数
bool validate(int x, int y, std::function<bool(int, int)> fcn);
```

更新之前的例子：

```
#include <functional>
#include <iostream>

int foo()
{
    return 5;
}

int goo()
{
    return 6;
}

int main()
{
    std::function<int()> fcnPtr{ &foo }; // 声明函数指针
    fcnPtr = &goo; // fcnPtr 现在指向函数 goo
    std::cout << fcnPtr() << '\n'; // 与通常一样调用函数

    return 0;
}
```

std::function 的类型别名有利于可读性：

```
using ValidateFunctionRaw = bool(*)(int, int);          // 原始函数指针的别名
using ValidateFunction = std::function<bool(int, int)>; // std::function 的别名
```

从 C++17 开始，CTAD 可以由初始化器推导 std::function 的模板参数，上述例子中可以将 `std::function<int()> fcnPtr{ &foo };` 修改为 `std::function fcnPtr{ &foo };`，并让编译器计算出模板类型。然而 CTAD 不能作用于类型别名，这是因为没有提供初始化器。

函数指针的类型接口

与推断普通变量的类型一样，`auto` 关键字也可以推断函数指针的类型：

```
#include <iostream>

int foo(int x)
{
    return x;
}

int main()
{
    auto fcnPtr{ &foo };
    std::cout << fcnPtr(5) << '\n';

    return 0;
}
```

不过缺陷就是，所有关于函数参数以及返回值的细节都被隐藏了，这样在调用的时候，或者使用其返回值时，很容易造成错误。

12.2 The stack and the heap

程序使用的内存通常被分隔成不同的区域，被称为段 segments：

- 代码段 code segment（也被称为文本段 text segment），是编译好的项目的存储。代码段通常只能可读。
- bss 段（也被称为非初始化数据段），是零初始化全局以及静态变量的存储。
- 数据段（也被称为初始化数据段），是初始化全局以及静态变量的存储。
- 堆，是动态分配变量的存储。
- 调用栈，是函数参数，本地变量，以及其它函数相关信息的存储。

堆 segment

堆 segment（也被熟知为“free store”）追踪动态内存分配的内存使用。
在 C++ 中，当使用 new 操作符分配内存，内存会被分配在应用程序的堆 segment 中。

```
int* ptr { new int };       // ptr 在堆中被指定了 4 字节
int* array { new int[10] }; // array 在堆中被指定了 40 字节
```

当一个动态分配变量被删除时，内存被“返回”给堆，接着可以被重新分配给未来的内存请求。需要留意的是删除一个指针并没有删除变量，只是仅仅将关联的内存还给操作系统。
堆有好处也有坏处：

- 堆上分配的内存相对来说比较慢。
- 被分配到的内存一直占用直到被指定释放内存（小心内存泄漏），或是程序终止（操作系统进行清理）。
- 动态分配内存必须通过指针来进行访问。解引用指针相比于直接访问变量会更慢一些的。
- 因为对是一个很大的内存池，大数组，结构体，或者类可以被分配到这里。

调用栈

调用栈 call stack（通常指代“the stack”）扮演更加有趣的角色。调用栈在程序启动时就持续追踪所有被激活的函数（那些已经被调用但是还没有结束），并且处理所有函数参数以及本地变量的分配。
调用栈被实现为栈数据结构。因此在谈到它们是如何工作之前，必须要理解清楚什么是一个栈数据结构。

栈数据结构

数据结构是程序用来管理数据的机制，使得数据可以被有效的利用。我们已经学习了若干数据结构，例如数组和结构体。它们都提供了便利的存储数据以及访问数据的机制。还有很多别的数据结构在程序中普遍使用，它们之间相当多的都已经被标准库实现了，栈就是其中之一。
在计算机程序中，栈是一种可以存储若干变量（如同数组）的容器数据类型。然而数组允许用户无需顺序的方式（**随机访问** random access）访问以及修改元素，栈却有其限制。栈有下列三种操作方式：

1. 查看栈顶层元素（通常由函数 `top()` 进行调用，有时称为 `peek()` ）
2. 移除栈顶层元素（通过 `pop()` 函数）
3. 放置新元素在栈顶层（通过 `push()` 函数）

栈是后进 last-in 先出 first-out（LIFO）结构。

调用栈 segment

调用栈 segment 是用于存储调用栈的内存。在应用启动时，main() 函数由操作系统推入到调用栈中，接着程序继续执行。
当遇到函数调用时，函数推入调用栈。当当前函数结束时，函数从调用栈上移除（这个过程有时被称为 **unwinding the stack**）。因此，观察当前调用栈的函数，可以看到所用被调用的函数来获取当前真正执行的点位。

栈溢出

栈的大小是有限的，因此只能存储有限的信息。如果程序尝试放入超过栈所能承受的信息时，**栈溢出** stack overflow 则会发生。即所有的栈都被分配了 – 后续的分配会溢出到别的部分的内存中。

12.3 std::vector capacity and stack behavior

之前的讨论过 std::vector 可以作为一个兼具记住长度以及有需要是动态调整大小的数组。尽管这些是 std::vector 最有用最通用的地方，然而它还有一些额外的属性与能力可以用作他出。

```
int* array{ new int[10] { 1, 2, 3, 4, 5 } };
```

上述代码表示数组长度为 10，即使只使用了 5 个元素进行分配。然而如果希望遍历这些初始化后的元素，并保留未使用的位置呢？这种情况下需要分开追踪到底有多少元素从被分配的元素中被“使用”了。有别于内建数组或者 std::array，仅记住长度，std::vector 包含两种分开的属性：长度与容量。在 std::vector 的上下文中，**长度** length 是数组中有多少元素被使用了，而**容量** capacity 是有多少元素被分配进了内存。

```
#include <vector>
#include <iostream>

int main()
{
    std::vector array { 0, 1, 2 };
    array.resize(5); // 设定长度为 5

    std::cout << "The length is: " << array.size() << '\n';

    for (auto element: array)
        std::cout << element << ' ';

    return 0;
};
```

打印：

```
The length is: 5
0 1 2 0 0
```

上述代码使用 `resize()` 函数设定向量长度为 5，也就是告诉数组只使用数组的前五位元素。然而有趣的问题是：该数组的容量是多少?

可以通过 `capacity()` 函数进行查询：

```
#include <vector>
#include <iostream>

int main()
{
    std::vector array { 0, 1, 2 };
    array.resize(5); // set length to 5

    std::cout << "The length is: " << array.size() << '\n';
    std::cout << "The capacity is: " << array.capacity() << '\n';
}
```

打印：

```
The length is: 5
The capacity is: 5
```

为什么要区分长度和容量呢? std::vector 在需要的时候会释放其内存，不过它倾向于这么做，因为数组的重设大小从计算上而言是昂贵的。考虑以下代码：

```
#include <vector>
#include <iostream>

int main()
{
  std::vector array{};
  array = { 0, 1, 2, 3, 4 }; // okay, array length = 5
  std::cout << "length: " << array.size() << "  capacity: " << array.capacity() << '\n';

  array = { 9, 8, 7 }; // okay, array length is now 3!
  std::cout << "length: " << array.size() << "  capacity: " << array.capacity() << '\n';

  return 0;
}
```

打印：

```
length: 5  capacity: 5
length: 3  capacity: 5
```

注意尽管赋值了更新的数组给向量，向量并没有释放内存（容量仍然是 5）。修改长度很容易，因为当前只有 3 个元素可用。

数组下标和 `at()` 都是基于长度而言的，而不是容量。

std::vector 栈的行为

如果下标操作符和 `at()` 函数是基于数组长度的，同时容量也总是大于长度的，为什么还需要担心容量呢？尽管 std::vector 可以被用作为动态数组，也可以用作为栈。只需要使用 3 个函数匹配栈的操作：

- `push_back()` 推入元素进栈 - `back()` 返回栈的顶层元素 - `pop_back()` 移除栈的顶层元素

```
#include <iostream>
#include <vector>

void printStack(const std::vector<int>& stack)
{
  for (auto element : stack)
    std::cout << element << ' ';
  std::cout << "(cap " << stack.capacity() << " length " << stack.size() << ")\n";
}

int main()
{
  std::vector<int> stack{};

  printStack(stack);

  stack.push_back(5); // push_back() 推入元素
  printStack(stack);

  stack.push_back(3);
  printStack(stack);

  stack.push_back(2);
  printStack(stack);

  std::cout << "top: " << stack.back() << '\n'; // back() 返回最后一个元素

  stack.pop_back(); // pop_back() 移除元素
  printStack(stack);

  stack.pop_back();
  printStack(stack);

  stack.pop_back();
  printStack(stack);

  return 0;
}
```

打印：

```
(cap 0 length 0)
5 (cap 1 length 1)
5 3 (cap 2 length 2)
5 3 2 (cap 3 length 3)
top: 2
5 3 (cap 3 length 2)
5 (cap 3 length 1)
(cap 3 length 0)
```

因为向量的重设大小非常的昂贵，可以提前使用 `reserve()` 函数告诉向量分配特定大小的空间：

```
#include <vector>
#include <iostream>

void printStack(const std::vector<int>& stack)
{
  for (auto element : stack)
    std::cout << element << ' ';
  std::cout << "(cap " << stack.capacity() << " length " << stack.size() << ")\n";
}

int main()
{
  std::vector<int> stack{};

  stack.reserve(5); // 设定容量（至少）为 5

  printStack(stack);

  stack.push_back(5);
  printStack(stack);

  stack.push_back(3);
  printStack(stack);

  stack.push_back(2);
  printStack(stack);

  std::cout << "top: " << stack.back() << '\n';

  stack.pop_back();
  printStack(stack);

  stack.pop_back();
  printStack(stack);

  stack.pop_back();
  printStack(stack);
```

```
38
39   return 0;
40 }
```

打印：

```
(cap 5 length 0)
5 (cap 5 length 1)
5 3 (cap 5 length 2)
5 3 2 (cap 5 length 3)
top: 2
5 3 (cap 5 length 2)
5 (cap 5 length 1)
(cap 5 length 0)
```

向量可能分配额外容量

当向量重设大小时，有可能会分配更多的容量。这是为了让额外的元素有“喘息空间”，来减少重设空间操作的次数：

```
#include <vector>
#include <iostream>

int main()
{
  std::vector v{ 0, 1, 2, 3, 4 };
  std::cout << "size: " << v.size() << "  cap: " << v.capacity() << '\n';

  v.push_back(5); // 增加另一元素
  std::cout << "size: " << v.size() << "  cap: " << v.capacity() << '\n';

  return 0;
}
```

有可能打印：

```
size: 5  cap: 5
size: 6  cap: 7
```

如何分配额外容量是基于编译器的实现所决定的。

12.4 Recursion

C++ 中的**递归函数** recursive function 是一种调用自身的函数。

递归终结条件

递归终结 recursive termination 是当条件达满足时，控制函数停止调用自己。

```cpp
#include <iostream>

void countDown(int count)
{
    std::cout << "push " << count << '\n';

    if (count > 1) // 终结条件
        countDown(count-1);

    std::cout << "pop " << count << '\n';
}

int main()
{
    countDown(5);
    return 0;
}
```

记忆化算法

记忆化 memoization 缓存所有昂贵函数调用，在当同样输入再次发生时，其结果可以直接被返回。

```cpp
#include <iostream>
#include <vector>

int fibonacci(int count)
{
  // 使用 static std::vector 缓存计算结果
  static std::vector results{ 0, 1 };

  // 如果该结果已被计算过，则直接使用缓存中的结果
  if (count < static_cast<int>(std::size(results)))
    return results[count];

  // 否则结算新值并添加至缓存
  results.push_back(fibonacci(count - 1) + fibonacci(count - 2));
  return results[count];
}

int main()
{
  for (int count { 0 }; count < 13; ++count)
    std::cout << fibonacci(count) << ' ';

  return 0;
}
```

12.5 Command line arguments

命令行参数 command line arguments 是在程序启动时，由操作系统传递给程序的可选字符串参数。程序可以使用它们作为输入（或者忽略它们）。与函数参数提供输入传递给函数的原理相似，用户或程序输入命令行参数至程序。

```
int main(int argc, char* argv[])
```

或者

```
int main(int argc, char** argv)
```

推荐使用第一种方式。

argc 是一个整数参数包含了传递给程序的参数数量（argc = argument count）。argc 至少为 1，因为第一个参数总是程序自身的名称。每个由用户提供的命令行参数会使 argc 增加 1.

argv 是真实参数值（argv = argument values，尽管正确的名称应该是“argument vectors”）。argv 是 C-style 字符串的数组。数组的长度为 argc。

```
// Program: MyArgs
#include <iostream>

int main(int argc, char* argv[])
{
    std::cout << "There are " << argc << " arguments:\n";

    // 循环每个参数并打印其索引与值
    for (int count{ 0 }; count < argc; ++count)
    {
        std::cout << count << ' ' << argv[count] << '\n';
    }

    return 0;
}
```

处理数值类型参数

```
#include <iostream>
#include <sstream> // std::stringstream
#include <string>

int main(int argc, char* argv[])
{
  if (argc <= 1)
  {
    // 有些操作系统上 argv[0] 为空字符而不是程序名称
    // 这需要条件判断 argv[0] 是否为空
    if (argv[0])
```

```
12        std::cout << "Usage: " << argv[0] << " <number>" << '\n';
13      else
14        std::cout << "Usage: <program name> <number>" << '\n';
15
16      return 1;
17    }
18
19    std::stringstream convert{ argv[1] }; // 名为 convert 的 stringstream，使用 argv[1] 进行初始
        化
20
21    int myint{};
22    if (!(convert >> myint)) // 转换
23      myint = 0; // 如果转换失败，给 myint 一个默认值
24
25    std::cout << "Got integer: " << myint << '\n';
26
27    return 0;
28  }
```

12.7 Introduction to lambdas (anonymous functions)

```
1  #include <algorithm>
2  #include <array>
3  #include <iostream>
4  #include <string_view>
5
6  // 如果元素匹配，函数返回 true
7  bool containsNut(std::string_view str)
8  {
9      // 如果找不到子字符串，std::string_view::find 返回 std::string_view::npos，
10     // 否则返回子字符串的索引。
11     return (str.find("nut") != std::string_view::npos);
12 }
13
14 int main()
15 {
16     std::array<std::string_view, 4> arr{ "apple", "banana", "walnut", "lemon" };
17
18     // 检索 array 查看元素是否包含 "nut" 子字符串
19     auto found{ std::find_if(arr.begin(), arr.end(), containsNut) };
20
21     if (found == arr.end())
22     {
23         std::cout << "No nuts\n";
24     }
25     else
26     {
27         std::cout << "Found " << *found << '\n';
```

```
    }

    return 0;
}
```

这里根本的问题是 `std::find_if` 需要用户传递一个函数指针。正因如此，需要强制定义一个只会使用一次的函数，且必须要有函数名，且必须放入全局作用域中（因为函数不可以嵌套!）。

Lambdas 来拯救

一个 **lambda 表达式 **（也称 **lambda** 或者 ** 闭包 closure**）允许用户在函数中定义匿名函数。嵌套是很重要的，因为它允许用户既可以避免命名空间的污染，还可以定义与使用位置更近的函数（提供额外的内容）。
C++ 中 lambdas 的语法是一个奇怪的东西，需要花一点时间习惯它：

```
[ captureClause ] ( parameters ) -> returnType
{
    statements;
}
```

- 如果不需要捕获变量，捕获从句可以为空。- 如果不需要参数，参数列表可以为空也可以省略。- 返回类型是可选地，且如果省略了，则视为 `auto` （这样会使用类型推导来决定返回类型）。
那么使用 lambda 改写后的代码如下：

```
#include <algorithm>
#include <array>
#include <iostream>
#include <string_view>

int main()
{
  constexpr std::array<std::string_view, 4> arr{ "apple", "banana", "walnut", "lemon" };

  const auto found{ std::find_if(arr.begin(), arr.end(),
                            [](std::string_view str) // 此为 lambda，无捕获从句
                            {
                              return (str.find("nut") != std::string_view::npos);
                            }) };

  if (found == arr.end())
  {
    std::cout << "No nuts\n";
  }
  else
  {
    std::cout << "Found " << *found << '\n';
  }
```

```
24
25   return 0;
26 }
```

lambda 的类型

上述例子中，定义了一个 lambda。那样的使用方式有时被称为 ** 函数字面量 function literal**。然而，在其使用的地方编写的 lambda，有时会让代码变得难以阅读。命名的 lambda 可以让代码便于阅读。

例如下面的代码切片中使用 `std::all_of` 来检查所有元素是否为偶数：

```
1 // 坏：需要阅读 lambda 才能理解发生了什么
2 return std::all_of(array.begin(), array.end(), [](int i){ return ((i % 2) == 0); });
```

这样可以提高可读性：

```
1 // 好：可以存储 lambda 为一个命名变量，并传递其至函数
2 auto isEven{
3   [](int i)
4   {
5     return ((i % 2) == 0);
6   }
7 };
8
9 return std::all_of(array.begin(), array.end(), isEven);
```

进阶：实际上，lambdas 并不是函数（这也是为什么它们可以避开 C++ 不支持嵌套函数的限制）。它们是特殊的名为函子 functor 的对象。函子是一种包含了重载 `operator()` 的对象，使得它们可以像函数那样被调用。

有几种存储 lambda 的方式。如果 lambda 有空的捕获从句（即 [] 里没有任何东西），可以使用通常的函数指针。`std::function` 或者通过 `auto` 关键字的类型推导同样也可用（即使 lambda 拥有捕获从句）。

```
1  #include <functional>
2
3  int main()
4  {
5    // 一个普通函数指针。仅作用在空捕获从句
6    double (*addNumbers1)(double, double){
7      [](double a, double b) {
8        return (a + b);
9      }
10   };
11
12   addNumbers1(1, 2);
13
14   // 使用 std::function。lambda 可以拥有非空的捕获从句
```

```
  std::function addNumbers2{ // 注意：在 C++17 之前，使用 std::function<double(double, double)>
    [](double a, double b) {
      return (a + b);
    }
  };

  addNumbers2(3, 4);

  // 使用 auto。存储带有其真实类型的 lambda
  auto addNumbers3{
    [](double a, double b) {
      return (a + b);
    }
  };

  addNumbers3(5, 6);

  return 0;
}
```

仅有的使用 lambda 真实类型的方法是 `auto` 。`auto` 拥有着不需要比较 `std::function` 的便利。

不幸的是，在 C++20 之前，用户不能总是使用 `auto` 。以防真实的 lambda 是未知的（例如因为传递 lambda 给一个函数作为入参，且由调用者决定传递什么样的 lambda），不可以在无比较的情况下使用 `auto` 。这种情况下 `std::function` 反而可以使用。

```
#include <functional>
#include <iostream>

// 不知道 fn 会是什么。std::function 可用作于通常函数以及 lambdas。
void repeat(int repetitions, const std::function<void(int)>& fn)
{
  for (int i{ 0 }; i < repetitions; ++i)
  {
    fn(i);
  }
}

int main()
{
  repeat(3, [](int i) {
    std::cout << i << '\n';
  });

  return 0;
}
```

如果给 `fn` 使用 `auto` ，函数的调用者不会知道 `fn` 的入参以及返回类型。当缩写函数模版

在 C++20 里添加了，这个限制才被打破。
另外，因为它们本质是模板，拥有 `auto` 参数的函数不再可以被分隔进头文件与源文件。
规则：当初始化 lambda 变量时，使用 `auto` ；而不可通过 lambda 进行初始化的时候，使用 `std::function` 。

泛型 lambdas

大多数情况下，lambda 参数与普通函数参数无异。
一个显著的区别是自从 C++14 之后可以使用 `auto` 的参数（注意：C++20，普通函数也可以使用 `auto` 参数了）。当一个 lambda 拥有一个或以上的 `auto` 参数，编译器将从调用 lambda 处推断参数所需要的类型。
因为带有一个或以上 `auto` 的 lambdas 可以用作于广泛的类型，它们通常被称为 ** 泛型 lambdas**。
进阶：在 lambda 上下文中使用时，`auto` 是模版参数的缩写。

```
#include <algorithm>
#include <array>
#include <iostream>
#include <string_view>

int main()
{
  constexpr std::array months{ // C++17 之前使用 std::array<const char*, 12>
    "January",
    "February",
    "March",
    "April",
    "May",
    "June",
    "July",
    "August",
    "September",
    "October",
    "November",
    "December"
  };

  // 查询两个连续的拥有相同开头字母的月份
  const auto sameLetter{ std::adjacent_find(months.begin(), months.end(),
                                      [](const auto& a, const auto& b) {
                                        return (a[0] == b[0]);
                                      }) };

  // 确保两个月份被找到
  if (sameLetter != months.end())
```

```
  {
    // std::next 返回 sameLetter 的下一个遍历器
    std::cout << *sameLetter << " and " << *std::next(sameLetter)
              << " start with the same letter\n";
  }

  return 0;
}
```

上述例子中使用 `auto` 参数来记录字符串的 `const` 引用。因为所有字符串类型允许通过 `operator[]` 访问它们的独立字符，所以不需要关心使用者是否传达的是一个 `std::string`，C-style 字符串，或者是其它。这允许用户编写一个可以接受上述任何类型的 lambda，意味着如果之后替换了 `months` 的类型，不需要再重写 lambda。

然而，`auto` 并不总是最好的选择：

```
#include <algorithm>
#include <array>
#include <iostream>
#include <string_view>

int main()
{
  constexpr std::array months{ // C++17 之前使用 std::array<const char*, 12>
    "January",
    "February",
    "March",
    "April",
    "May",
    "June",
    "July",
    "August",
    "September",
    "October",
    "November",
    "December"
  };

  // 统计多少个月份是 5 个字母组成的
  const auto fiveLetterMonths{ std::count_if(months.begin(), months.end(),
                                       [](std::string_view str) {
                                         return (str.length() == 5);
                                       }) };

  std::cout << "There are " << fiveLetterMonths << " months with 5 letters\n";

  return 0;
}
```

这个例子中，使用 `auto` 将推断一个 `const char*` 的类型。C-style 字符串并不容易处理（除了使用 `operator[]` ）。这种情况下，推荐显示定义参数为 `std::string_view` ，这样允许用户更方便的操作（例如查询 string_view 其长度，即使用户传递的是 C-style 数组）。

泛型 lambdas 与静态变量

有一件值得注意的是唯一的 lambda 会因为 `auto` 的解释，被生成不同类型：

```
#include <algorithm>
#include <array>
#include <iostream>
#include <string_view>

int main()
{
  // 打印值并统计 @print 被调用了多少次
  auto print{
    [](auto value) {
      static int callCount{ 0 };
      std::cout << callCount++ << ": " << value << '\n';
    }
  };

  print("hello"); // 0: hello
  print("world"); // 1: world

  print(1); // 0: 1
  print(2); // 1: 2

  print("ding dong"); // 2: ding dong

  return 0;
}
```

返回类型推导以及尾随返回类型

如果使用了类型推导，一个 lambda 的返回类型会由自身内部的 `return` 声明推导出来，lambda 中的所有返回声明必须返回同样类型（否则编译器无法知道使用哪个）。

```
#include <iostream>

int main()
{
  auto divide{ [](int x, int y, bool intDivision) { // 注意：没有值得返回类型
    if (intDivision)
      return x / y; // 返回类型为 int
    else
```

```
      return static_cast<double>(x) / y; // 错误: 返回类型与前一个返回类型不同
  } };

  std::cout << divide(3, 2, true) << '\n';
  std::cout << divide(3, 2, false) << '\n';

  return 0;
}
```

这会产生编译错误，因为第一个返回声明（int）的返回类型鱼第二个返回声明（double）的返回类型不匹配。

针对返回不同类型的问题有两种方法：

1. 显式的强制转换类型使得所有返回类型匹配 1. 显式指定 lambda 的返回类型，并让编译器进行隐式转换

第二个方法通常来说更好一些：

```
#include <iostream>

int main()
{
  // 注意: 显式指定这里的返回为 double
  auto divide{ [](int x, int y, bool intDivision) -> double {
    if (intDivision)
      return x / y; // 将会进行隐式转换, 将结果转换成 double
    else
      return static_cast<double>(x) / y;
  } };

  std::cout << divide(3, 2, true) << '\n';
  std::cout << divide(3, 2, false) << '\n';

  return 0;
}
```

标准库函数对象

对于常用的操作（例如加法，否定，或者比较），用户不需要编写 lambdas，因为标准库已经有很多可调用对象，它们都定义于 `<functional>` 头文件。

```
#include <algorithm>
#include <array>
#include <iostream>
#include <functional> // std::greater

int main()
{
  std::array arr{ 13, 90, 99, 5, 40, 80 };
```

```

  // 传递 std::greater 给 std::sort
  std::sort(arr.begin(), arr.end(), std::greater{}); // 注意: 需要花括号进行实例化

  for (int i : arr)
  {
    std::cout << i << ' ';
  }

  std::cout << '\n';

  return 0;
}
```

12.8 Lambda captures

捕获从句 capture clause 用于（间接的）给予 lambda 访问周边作用域的可用变量。

```
#include <algorithm>
#include <array>
#include <iostream>
#include <string_view>
#include <string>

int main()
{
  std::array<std::string_view, 4> arr{ "apple", "banana", "walnut", "lemon" };

  std::cout << "search for: ";

  std::string search{};
  std::cin >> search;

  // 捕获 @search                                    vvvvvvv
  auto found{ std::find_if(arr.begin(), arr.end(), [search](std::string_view str) {
    return (str.find(search) != std::string_view::npos);
  }) };

  if (found == arr.end())
  {
    std::cout << "Not found\n";
  }
  else
  {
    std::cout << "Found " << *found << '\n';
  }

  return 0;
```

```
}
```

捕获是如何工作的?

上述代码中，lambda 看起来像是直接访问 `main` 中的 `search` 变量，实际上并不是这样。lambdas 可能看起来像是嵌套代码块，但是实际上有细微的差别（而这细微的差别却很重要）。
当一个 lambda 定义被执行时，对于每个其需要捕获的变量会在 lambda 中进行拷贝。这些被拷贝的变量皆在外部作用域进行初始化。
因此上述例子中，当 lambda 对象被创建，lambda 获取自身所需变量 `search` 的拷贝。该 `search` 拥有与 `main` 中的 `search` 相同的值，所以 lambda 的行为像是在访问 `main` 中的 `search` 而实际上并不是。
而拷贝的变量拥有相同的名称，却不需要拥有与原始变量相同的类型（这在之后章节中会讲到）。
重点：被 lambda 捕获的变量是外部作用域变量的拷贝，而不是真实变量。
进阶：尽管 lambdas 看起来像是函数，它们实际上是可以像函数那样调用的对象（名为**函子** functors – 之后的章节会讲解如何创建用户自定义函子）。当编译器遇到 lambda 定义，lambda 对象被实例化，lambda 的成员也被初始化。

捕获默认为 const 值

默认情况下，变量是被 `const value` 所捕获。这就意味着当 lambda 创建时，lambda 捕获外部作用域中变量的常数拷贝，也就是说 lambda 不允许修改变量。

```
#include <iostream>

int main()
{
  int ammo{ 10 };

  // 定义一个 lambda 并由 “shoot” 变量进行存储
  auto shoot{
    [ammo]() {
      // 非法，ammo 作为 const 拷贝被捕获
      --ammo;

      std::cout << "Pew! " << ammo << " shot(s) left.\n";
    }
  };

  // 调用 lambda
  shoot();

  std::cout << ammo << " shot(s) left\n";

```

```
  return 0;
}
```

可变捕获

mutable 关键字从所有被捕获的变量上移除 `const` 限制。

```
#include <iostream>

int main()
{
  int ammo{ 10 };

  auto shoot{
    // 在参数列表后添加 mutable
    [ammo]() mutable {
      // 现在可以修改 ammo 了
      --ammo;

      std::cout << "Pew! " << ammo << " shot(s) left.\n";
    }
  };

  shoot();
  shoot();

  std::cout << ammo << " shot(s) left\n";

  return 0;
}
```

警告：因为被捕获的变量是 lambda 对象的成员，它们的值会持续存在于若干 lambda 调用！

引用捕获

```
#include <iostream>

int main()
{
  int ammo{ 10 };

  auto shoot{
    // 这里不再需要 mutable 了
    [&ammo]() { // &ammo 意味着 ammo 是被引用捕获的
      // 修改 ammo 将会影响 main 的 ammo
      --ammo;

      std::cout << "Pew! " << ammo << " shot(s) left.\n";
```

```
    }
  };

  shoot();

  std::cout << ammo << " shot(s) left\n";

  return 0;
}
```

使用引用捕获统计 `std::sort` 进行了多少次比较:

```
#include <algorithm>
#include <array>
#include <iostream>
#include <string>

struct Car
{
  std::string make{};
  std::string model{};
};

int main()
{
  std::array<Car, 3> cars{ { { "Volkswagen", "Golf" },
                             { "Toyota", "Corolla" },
                             { "Honda", "Civic" } } };

  int comparisons{ 0 };

  std::sort(cars.begin(), cars.end(),
    // 引用捕获 @comparisons
    [&comparisons](const auto& a, const auto& b) {
      // 引用捕获，可以不使用 "mutable" 关键字修改 comparisons
      ++comparisons;

      // 排序 make
      return (a.make < b.make);
  });

  std::cout << "Comparisons: " << comparisons << '\n';

  for (const auto& car : cars)
  {
    std::cout << car.make << ' ' << car.model << '\n';
  }

  return 0;
}
```

捕获若干变量

由逗号分隔捕获若干变量，这样可以混合包含值捕获或是引用捕获：

```
int health{ 33 };
int armor{ 100 };
std::vector<CEnemy> enemies{};

// 值捕获 health 与 armor，引用捕获 enemies
[health, armor, &enemies](){};
```

默认捕获

有时显式列举捕获的变量很麻烦，如果修改了 lambda，有可能会忘记添加或移除被捕获的变量。幸运的是，可以授权编译器帮助用户自动生成需要捕获的变量列表。

默认捕获 default capture（也称为 **capture-default**）捕获所有在 lambda 中提及的变量，而没有提及的变量则不会捕获。

使用 `=` ，进行值默认捕获。

使用 `&` ，进行引用默认捕获。

```
#include <algorithm>
#include <array>
#include <iostream>

int main()
{
  std::array areas{ 100, 25, 121, 40, 56 };

  int width{};
  int height{};

  std::cout << "Enter width and height: ";
  std::cin >> width >> height;

  auto found{ std::find_if(areas.begin(), areas.end(),
                           [=](int knownArea) { // 值默认捕获 width 与 height
                             return (width * height == knownArea); // 因为它们在这里被提及
                           }) };

  if (found == areas.end())
  {
    std::cout << "I don't know this area :(\n";
  }
  else
  {
    std::cout << "Area found :)\n";
  }
```

```
28
29   return 0;
30 }
```

默认捕获可以与普通捕获混合。可以值捕获一些变量并引用捕获其它，但是每个变量只能被捕获一次。

```
1  int health{ 33 };
2  int armor{ 100 };
3  std::vector<CEnemy> enemies{};
4
5  // 值捕获 health 与 armor，引用捕获 enemies
6  [health, armor, &enemies](){};
7
8  // 引用捕获 enemies，其余的值默认捕获
9  [=, &enemies](){};
10
11 // 值捕获 armor，其余的引用默认捕获
12 [&, armor](){};
13
14 // 非法，已经说了引用默认捕获所有变量
15 [&, &armor](){};
16
17 // 非法，已经说了值默认捕获所有变量
18 [=, armor](){};
19
20 // 非法，armor 出现了两次
21 [armor, &health, &armor](){};
22
23 // 非法，默认捕获必须位于捕获组的第一个元素
24 [armor, &](){};
```

在 lambda 捕获中定义新的变量

有时候用户希望在捕获变量时允许微小的修改，或者是声明一个新的只在 lambda 中可见的变量。那么可以在 lambda 捕获时，定义一个没有指定类型的变量。

```
1  #include <array>
2  #include <iostream>
3  #include <algorithm>
4
5  int main()
6  {
7    std::array areas{ 100, 25, 121, 40, 56 };
8
9    int width{};
10   int height{};
11
```

```
  std::cout << "Enter width and height: ";
  std::cin >> width >> height;

  // 这里存储了 areas，但是用户输入 width 与 height
  // 需要在进行查询之前，计算 area
  auto found{ std::find_if(areas.begin(), areas.end(),
                           // 声明一个新的只在 lambda 中可见的变量
                           // userArea 的类型自动被推导为 int
                           [userArea{ width * height }](int knownArea) {
                             return (userArea == knownArea);
                           }) };

  if (found == areas.end())
  {
    std::cout << "I don't know this area :(\n";
  }
  else
  {
    std::cout << "Area found :)\n";
  }

  return 0;
}
```

`userArea` 只会在 lambda 定义时计算一次。被计算的值会存储在 lambda 对象内，该值在每次调用时相同。如果一个 lambda 是可变的，同时修改了定义在捕获的变量，那么原始值则被覆盖。

最佳实践：仅在值很短并且类型显而易见时，在捕获时初始化变量。否则最好将变量定义在 lambda 外，然后捕获它。

悬垂捕获变量

变量在 lambda 定义时被捕获。如果是引用捕获在 lambda 之前消亡了，那么 lambda 持有的则变成悬垂引用。

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

// 返回一个 lambda
auto makeWalrus(const std::string& name)
{
  // 引用捕获 name，返回 lambda
  return [&]() {
    std::cout << "I am a walrus, my name is " << name << '\n'; // 未定义行为
  };
}

```

```
int main()
{
  // sayName 是一个由 makeWalrus 返回的 lambda
  auto sayName{ makeWalrus("Roofus") };

  // 调用 lambda 函数
  sayName();

  return 0;
}
```

调用 `makeWalrus` 时，从字符串字面量“Roofus”创建了临时的 `std::string` 。`makeWalrus` 中的 lambda 引用捕获了字符串。该临时字符串在 `makeWalrus` 返回时消亡，但是 lambda 仍然指向它。接着调用 `sayName` 时，悬垂引用被访问了，导致了未定义行为。

注意这同样也发生在 `name` 值传递给 `makeWalrus` 时，因为 `name` 还是会在 `makeWalrus` 结束时消亡，同样的留下悬垂引用。

警告：引用捕获变量时要额外的小心，特别是引用默认捕获。被捕获变量的存活时间必须比 lambda 长。

如果希望在 lambda 使用时，被捕获的 `name` 有效，需要通过值捕获的方式（要么显式声明要么使用默认的值捕获）。

可变 lambdas 的无意识的拷贝

因为 lambdas 是对象，它们可以被拷贝。在某些情况下会导致一些问题。

```
#include <iostream>

int main()
{
  int i{ 0 };

  // 创建一个名为 count 的 lambda
  auto count{ [i]() mutable {
    std::cout << ++i << '\n';
  } };

  count(); // 唤起 count

  auto otherCount{ count }; // 创建一个 count 的拷贝

  // 唤起 count 与其拷贝
  count();
  otherCount();

  return 0;
}
```

打印:

```
1
2
2
```

上述代码没有打印 1，2，3，而是打印 2 两次。当作为 `count` 的拷贝，创建 `otherCount` 时，用户创建了带有当前状态的 `count` 的拷贝。因此 `otherCount` 拷贝 `count` 时，它们拥有自己的 `i` 。

再来看一个不是那么明显的例子:

```
#include <iostream>
#include <functional>

void myInvoke(const std::function<void()>& fn)
{
    fn();
}

int main()
{
    int i{ 0 };

    // 自增并打印 @i 的本地拷贝
    auto count{ [i]() mutable {
      std::cout << ++i << '\n';
    } };

    myInvoke(count);
    myInvoke(count);
    myInvoke(count);

    return 0;
}
```

打印:

```
1
1
1
```

这展示了与之前代码同样的问题。当 `std::function` 创建了 lambda，`std::function` 内在拷贝了 lambda 对象。因此调用 `fn()` 实际上是在执行 lambda 的拷贝，而不是真实的 lambda。如果传递的是可变 lambda，同时希望避免拷贝发生，那么有两种选择。第一个选择是使用非捕获 lambda – 上述例子中，移除捕获并使用静态本地变量进行状态追踪。但是静态本地变量很难被追踪，同时导致代码可读性降低。另一个更好的选择是在创建 lambda 第一时间就阻止其拷贝的发生。但是由于用户不能改变 `std::function` （或者其它标准库函数与对象）的实现，那么该怎么做呢?

幸运的是，C++ 提供了一个名为 `std::reference_wrapper` 的便捷类型（同样也定义在 `<functional>` 头文件中），它允许用户传递普通类型而实际上是引用。更方便的是，`std::reference_wrapper` 可以通过 `std::ref()` 函数创建。通过包裹 lambda 在一个 `std::reference_wrapper` 中，任何时候任何人尝试拷贝该 lambda，都会拷贝 lambda 的引用而不是真实的对象。

下面是通过 `std::ref` 进行代码升级：

```
#include <iostream>
#include <functional>

void myInvoke(const std::function<void()>& fn)
{
    fn();
}

int main()
{
    int i{ 0 };

    // 自增并打印 @i 的本地拷贝
    auto count{ [i]() mutable {
      std::cout << ++i << '\n';
    } };

    // std::ref(count) 确保 count 被视为引用
    // 因此，任何对 count 的拷贝实际上都是拷贝引用
    // 确保只有一个 count 存在
    myInvoke(std::ref(count));
    myInvoke(std::ref(count));
    myInvoke(std::ref(count));

    return 0;
}
```

现在的打印符合预期了：

```
1
2
3
```

注意即使 `invoke` 值获取 `fn` 时，输出也不会有变化，因为通过 `std::ref` 创建的 `std::function` 并不创建 lambda 的拷贝。

规则：标准库的函数可能会拷贝函数对象（提醒：lambdas 是函数对象）。如果用户希望提供带有可变捕获变量的 lambdas 时，使用 `std::ref` 传递它们的引用。

最佳实践：尝试不使用可变 lambdas。不可变的 lambdas 更容易理解也不需要面对上述问题，而且在增加并行执行时会增加更多危险的问题。

13 Basic Object-oriented Programming

13.2 Classes and class members

类

面向对象编程的世界中，既希望自定义类型不仅仅存储数据，同样也希望提供处理数据的函数。
C++ 中通过关键字 **class** 可以定义一个新的 program-defined 类型。
在 C++ 中，类与结构体本质上相同，实际上，下述代码中的结构体与类完全一样：

```
struct DateStruct
{
    int year {};
    int month {};
    int day {};
};

class DateClass
{
public:
    int m_year {};
    int m_month {};
    int m_day {};
};
```

注意仅有一个明显的差别是 *public*：类中的关键字。下一章讨论其功能。
类似于结构体的定义，类的定义也不会分配任何内存，它仅仅定义着是类的模样。

成员函数

除了存储数据，类（结构体）可以包含函数！在类中定义的函数被称为**成员函数** member functions。成员函数既可以定义在类定义的内部，也可以是外部。

```
class DateClass
{
public:
    int m_year {};
    int m_month {};
    int m_day {};

    void print() // 定义一个名为 print() 的成员函数
    {
        std::cout << m_year << '/' << m_month << '/' << m_day;
    }
};

int main()
```

```
{
    DateClass today { 2020, 10, 14 };

    today.m_day = 16;
    today.print();

    return 0;
}
```

当调用“today.print()”时，编译器解释 `m_day` 为 `today.m_day`，`m_month` 为 `today.m_month` 以及 `m_year` 为 `today.m_year`。

通过这样的方式，关联对象被隐式传递到成员函数。正因如此，它通常被称为 **the implicit object**。

为成员变量添加“m_”前缀可以帮助其从函数参数或者成员函数中的本地变量中区分开来。

最佳实践：类命名以大写字母开头。

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

class Employee
{
public:
    std::string m_name {};
    int m_id {};
    double m_wage {};

    // 打印员工信息
    void print()
    {
        std::cout << "Name: " << m_name <<
                "  Id: " << m_id <<
                "  Wage: $" << m_wage << '\n';
    }
};

int main()
{
    // 定义两个员工
    Employee alex { "Alex", 1, 25.00 };
    Employee joe { "Joe", 2, 22.25 };

    // 打印员工信息
    alex.print();
    joe.print();

    return 0;
}
```

对于非成员函数而言，一个函数不能调用在其“之后”定义的函数（除非有前向声明）：

```
void x()
{
// 不可以在此调用 y() 除非编译器已经看到了 y() 的前向声明
}

void y()
{
}
```

对于成员函数而言，这个限制不再存在：

```
class foo
{
public:
     void x() { y(); } // 可以在此调用 y()，尽管 y() 定义在类的尾部
     void y() { };
};
```

成员类型

除了成员变量以及成员函数，类还可以拥有**成员类型**或者**嵌套类型**（包括类型别称）。

```
class Employee
{
public:
    using IDType = int;

    std::string m_name{};
    IDType m_id{};
    double m_wage{};

    void print()
    {
        std::cout << "Name: " << m_name <<
            "  Id: " << m_id <<
            "  Wage: $" << m_wage << '\n';
    }
};
```

最佳实践：面对只有数据的结构时，使用 struct 关键字；面对既有数据又有函数的对象而言，使用 class 关键字。

已经使用过类的例子

```
#include <string>
#include <array>
#include <vector>
```

```cpp
#include <iostream>

int main()
{
    std::string s { "Hello, world!" }; // 实例化一个字符串类对象
    std::array<int, 3> a { 1, 2, 3 }; // 实例化一个数组类对象
    std::vector<double> v { 1.1, 2.2, 3.3 }; // 实例化一个向量类对象

    std::cout << "length: " << s.length() << '\n'; // 调用成员函数

    return 0;
}
```

13.3 Public vs private access specifiers

一个结构体或类的**公有成员** public members 可以被任何人直接进行访问，包括其外部的代码皆可访问；一个结构体或类的**私有成员** private members 仅可以被累不的成员所访问。

访问说明符

尽管类成员默认情况下是私有的，可以通过 `public` 关键字使它们公用：

```cpp
class DateClass
{
public: // 注意这里使用了 public 关键字，以及冒号
    int m_month {}; // 公有，可以被任何人访问
    int m_day {}; // 公有，可以被任何人访问
    int m_year {}; // 公有，可以被任何人访问
};

int main()
{
    DateClass date;
    date.m_month = 10; // 可以，因为 m_month 是公有的
    date.m_day = 14;  // 可以，因为 m_day 是公有的
    date.m_year = 2020;  // 可以，因为 m_year 是公有的

    return 0;
}
```

混合访问说明符

最佳实践：一般情况下，成员变量私有，成员函数公有，除非有更好的理由不这么做。

```cpp
#include <iostream>

class DateClass // 默认情况下，成员都是私有的
```

```
{
    int m_month {}; // 私有，仅可被其它成员访问
    int m_day {}; // 私有，仅可被其它成员访问
    int m_year {}; // 私有，仅可被其它成员访问

public:
    void setDate(int month, int day, int year) // 公有，可以被任何人访问
    {
        // setDate() 可以访问私有成员，因为其本身就是类的成员
        m_month = month;
        m_day = day;
        m_year = year;
    }

    void print() // 公有，可以被任何人访问
    {
        std::cout << m_month << '/' << m_day << '/' << m_year;
    }
};

int main()
{
    DateClass date;
    date.setDate(10, 14, 2020); // 可以，因为 setDate() 是共有的
    date.print();               // 可以，因为 print() 是共有的
    std::cout << '\n';

    return 0;
}
```

一组类的公有成员通常被归为**公有接口** public interface。

访问控制是基于每个类的

```
#include <iostream>

class DateClass
{
    int m_month {};
    int m_day {};
    int m_year {};

public:
    void setDate(int month, int day, int year)
    {
        m_month = month;
        m_day = day;
        m_year = year;
    }
```

```
16
17      void print()
18      {
19          std::cout << m_month << '/' << m_day << '/' << m_year;
20      }
21
22      // 注意额外新增的函数
23      void copyFrom(const DateClass& d)
24      {
25          // 注意我们可以直接访问 d 的成员
26          // 因为 d 的类型与现在作用域所属的类型相同!
27          m_month = d.m_month;
28          m_day = d.m_day;
29          m_year = d.m_year;
30      }
31  };
32
33  int main()
34  {
35      DateClass date;
36      date.setDate(10, 14, 2020); // 可以, 因为 setDate() 是共有的
37
38      DateClass copy {};
39      copy.copyFrom(date);        // 可以, 因为 copyFrom() 是公有的
40      copy.print();
41      std::cout << '\n';
42
43      return 0;
44  }
```

C++ 中有一个细微的差别经常被忽略或者误解，那就是访问控制是对于每个类的，而不是每个对象而言的。这就意味着当函数访问类的私有成员，它可以访问任何该类的对象中的所有私有成员。

上述例子中，`copyFrom()` 是 `DateClass` 的成员，即可以访问 `DateClass` 的私有成员。这就意味着 `copyFrom()` 不可以直接访问其操作（拷贝）的隐式对象的私有成员，同样也意味着它直接访问了入参类型为 `DateClass` d 的私有成员！如果参数 d 是其他类型，这就不行了。

重温结构体与类

上述提到了访问说明符，现在可以讲一下 C++ 中一个类与一个结构体的真实区别了：类的成员默认都是私有的，而结构体的成员默认公有的。

13.4 Access functions and encapsulation

封装

面向对象编程里，**封装** encapsulation（也称为**信息隐藏** Information hiding）是一种将实现的细节隐藏的步骤。用户通过公有接口访问对象，而不需要知道其内部是如何实现的。

- 既可以便于使用也可以减少程序的复杂度。
- 帮助保护数据以及防止滥用。
- 利于调试

访问函数

访问函数是简短的公有函数，其职责是获取或修改私有成员变量。
访问函数通常有两种：getters 以及 setters。**Getters**（有时也称为 **accessors**）是用于返回私有成员变量的函数；**Setters**（有时也称为 **mutators**）是用于设置私有成员变量的函数。

```
class Date
  {
    private:
    int m_month;
    int m_day;
    int m_year;

    public:
    int getMonth() { return m_month; } // month 的 getter
    void setMonth(int month) { m_month = month; } // month 的 setter

    int getDay() { return m_day; } // day 的 getter
    void setDay(int day) { m_day = day; } // day 的 setter

    int getYear() { return m_year; } // year 的 getter
    void setYear(int year) { m_year = year; } // year 的 setter
  };
```

一个简单的例子，getter 返回非 const 引用：

```
#include <iostream>

class Foo
{
private:
int m_value{ 4 };

public:
int& getValue() { return m_value; } // 返回非 const 引用
```

```
};

int main()
{
    Foo f;                      // f.m_value 初始化为 4
    f.getValue() = 5;           // 使用非 const 引用赋值 m_value 为 5
    std::cout << f.getValue(); // 打印 5

    return 0;
  }
```

最佳实践：getters 应该返回值或者 const 引用。

访问函数的关注点

创建类的时候考虑以下几点：

- 如果类的成员没有任何的外部访问需求，则不要提供该成员的访问函数
- 如果类的成员需要被外部访问，则需要考虑是否暴露 behavior 还是 action （例如是一个 `setAlive(bool)` 的 setter，还是实现 `kill()` 函数）
- 如果都不是，考虑是否仅可以提供一个 getter

13.5 Constructors

当类（或结构体）的所有成员都是公有时，可以使用列表初始化来进行**聚合初始化** aggregate initialization 来初始化类（或结构体）。

```
class Foo
{
public:
    int m_x {};
    int m_y {};
};

int main()
{
    Foo foo { 6, 7 }; // 列表初始化

    return 0;
}
```

然而一旦让任意一个成员变量为私有时，便不再允许使用聚合初始化了。这也符合常理：如果不能直接访问变量（因为其为私有），则不应该允许直接初始化它。

构造函数

构造函数是一种特殊的类成员函数，在创建一个该类的对象时会被自动调用。构造函数通常通过用户提供的值进行初始化类的成员变量，或者是构造类时任何必要的步骤（例如打开文件或者数据库）。
在一个构造函数被执行后，对象应该处于一个被定义好的，以及可使用的状态。
有别于普通的成员函数，构造函数的命名需要遵守规定的规则：

- 与类同名（大小写一致）
- 没有返回值（void 也不行）

默认构造函数以及默认初始化

一个不带有任何参数的构造函数（或者所有参数带有默认值）被称为**默认构造器**。如果用户没有提供初始化值，则调用默认构造函数。

```
#include <iostream>

class Fraction
{
private:
    int m_numerator {};
    int m_denominator {};

public:
    Fraction() // 默认构造函数
    {
        m_numerator = 0;
        m_denominator = 1;
    }

    int getNumerator() { return m_numerator; }
    int getDenominator() { return m_denominator; }
    double getValue() { return static_cast<double>(m_numerator) / m_denominator; }
};

int main()
{
    Fraction frac{}; // 调用 Fraction() 默认构造函数
    std::cout << frac.getNumerator() << '/' << frac.getDenominator() << '\n';

    return 0;
}
```

值初始化

上述代码，使用了值初始化 value-initialization 初始化类对象。

```
Fraction frac {}; // 使用空的花括号进行值初始化
```

同样也可以使用默认初始化 default-initialization 来初始化类对象。

```
Fraction frac; // 默认初始化 default-initialization，调用默认构造函数
```

大多数情况下，值初始化与默认初始化一个类对象会返回同样的结果：默认构造函数被调用。大多数程序员偏向使用默认初始化而不是值初始化一个类对象。这是因为当使用值初始化时，在一些确定的情况下，编译器调用默认构造函数之前，会零初始化 zero-initialize 类成员，这就带来一些性能损失（C++ 程序员不会关心他们不使用的特性）。

```
#include <iostream>

class Fraction
{
private:
    // 移除初始化器
    int m_numerator;
    int m_denominator;

public:
    // 移除默认构造函数

    int getNumerator() { return m_numerator; }
    int getDenominator() { return m_denominator; }
    double getValue() { return static_cast<double>(m_numerator) / m_denominator; }
};

int main()
{
    Fraction frac;
    // frac 没有被初始化，访问其成员会导致未定义的行为
    std::cout << frac.getNumerator() << '/' << frac.getDenominator() << '\n';

    return 0;
}
```

最佳实践：推荐值初始化 value-initialization 而不是默认初始化 default-initialization。

使用带有参数的构造函数进行直接初始化和列表初始化

由于默认构造函数方便了用户可以确保以合理的默认值进行初始化，而多数时候又希望类的实例化可以带有指定的入参。幸运的是，构造函数同样也可以带有参数。

```
#include <cassert>
```

```
2
3  class Fraction
4  {
5  private:
6      int m_numerator {};
7      int m_denominator {};
8
9  public:
10     Fraction() // 默认构造函数
11     {
12          m_numerator = 0;
13          m_denominator = 1;
14     }
15
16     // 带有两个参数的构造函数，一个带有默认值
17     Fraction(int numerator, int denominator=1)
18     {
19         assert(denominator != 0);
20         m_numerator = numerator;
21         m_denominator = denominator;
22     }
23
24     int getNumerator() { return m_numerator; }
25     int getDenominator() { return m_denominator; }
26     double getValue() { return static_cast<double>(m_numerator) / m_denominator; }
27 };
```

注意现在有两个构造函数：一个是默认构造函数，一个是带有两个入参的构造函数。它们可以和平的共存是因为函数重载。实际上可以定义很多个构造函数，只要它们都是唯一的签名（入参的数量与类型）。

那么如何使用带有参数的构造函数呢？非常简单，使用列表初始化或者直接初始化：

```
1  Fraction fiveThirds{ 5, 3 };  // 列表初始化 list initialization，调用 Fraction(int, int)
2  Fraction threeQuarters(3, 4); // 直接初始化 Direct initialization，也调用 Fraction(int, int)
```

同样的，这里推荐列表初始化。在后面章节会说明原因（模板与 `std::initializer_list`）何时使用直接初始化。有一些其它特别的构造函数可能会导致花括号初始化不一样，这种情况下需要使用直接初始化。之后的章节会进行讲解。

注意构造器的第二个参数带有默认值，因为下面代码也是合法的：

```
1  Fraction six{ 6 }; // 调用 Fraction(int, int) 构造函数，第二个参数使用默认值 1
```

最佳实践：推荐使用花括号进行类对象的初始化。

使用等号与类进行拷贝初始化

与基础变量相似，类也可以被拷贝初始化。

```
Fraction six = Fraction{ 6 }; // 拷贝初始化 Fraction，将调用 Fraction(6, 1)
Fraction seven = 7; // 拷贝初始化 Fraction。编译器将尝试找到转换 7 成为一个 Fraction 的方式，将唤起 Fraction(7, 1) 构造函数
```

然而建议不要使用这样的方式进行初始化类，因为可能会变得更低效。尽管直接初始化，列表初始化，以及拷贝初始化都可以作用于基础类型，类的拷贝初始化却不一致（尽管最终结果通常一致）。后续章节将会探讨更多的细节。

减少构造函数

上述带有两个构造函数的代码可以简化成：

```
#include <cassert>

class Fraction
{
private:
    int m_numerator {};
    int m_denominator {};

public:
    // 默认构造函数
    Fraction(int numerator=0, int denominator=1)
    {
        assert(denominator != 0);

        m_numerator = numerator;
        m_denominator = denominator;
    }

    int getNumerator() { return m_numerator; }
    int getDenominator() { return m_denominator; }
    double getValue() { return static_cast<double>(m_numerator) / m_denominator; }
};
```

隐式生成默认构造函数

如果类没有构造函数，C++ 则会为类自动生成一个公有的默认构造器。有时这被称为**隐式构造函数** implicit constructor（或者隐式生成构造函数）。

```
class Date
{
private:
    int m_year{ 1900 };
    int m_month{ 1 };
    int m_day{ 1 };

```

```
    // 没有用户提供的构造函数，编译器生成默认构造器
};

int main()
{
    Date date{};

    return 0;
}
```

如果类拥有其它的构造函数，那么隐式生成构造函数则不会被提供：

```
class Date
{
private:
    int m_year{ 1900 };
    int m_month{ 1 };
    int m_day{ 1 };

public:
    Date(int year, int month, int day) // 普通的非默认构造函数
    {
        m_year = year;
        m_month = month;
        m_day = day;
    }

    // 没有隐式构造函数提供，因为用户已经定义了
};

int main()
{
    Date date{}; // 错误：不可以实例化对象，因为默认构造函数不存在，同时编译器也不会生成它
    Date today{ 2020, 1, 19 }; // today 被初始化为 Jan 19th, 2020

    return 0;
}
```

在拥有其它构造函数的情况下希望拥有默认构造函数，可以选择为所有入参添加默认值，或是显式的定义一个默认构造函数。

还有第三种选择：使用 `default` 关键字告诉编译器创建一个默认构造函数：

```
class Date
{
private:
    int m_year{ 1900 };
    int m_month{ 1 };
    int m_day{ 1 };

public:
```

```
    // 告诉编译器创建一个默认构造函数，即使提供了其他用户定义的构造函数
    Date() = default;

    Date(int year, int month, int day) // 普通的非默认构造函数
    {
        m_year = year;
        m_month = month;
        m_day = day;
    }
};

int main()
{
    Date date{}; // date 被初始化为 Jan 1st, 1900
    Date today{ 2020, 10, 14 }; // today 被初始化为 Oct 14th, 2020

    return 0;
}
```

最佳实践：如果 `class` 拥有构造函数，同时还需要默认构造函数时（例如因为所有的成员使用非 static 成员初始化），使用 `= default` 。

包含类成员的类

```
#include <iostream>

class A
{
public:
    A() { std::cout << "A\n"; }
};

class B
{
private:
    A m_a; // B 作为成员变量，包含了 A

public:
    B() { std::cout << "B\n"; }
};

int main()
{
    B b;
    return 0;
}
```

构造函数笔记

很多新程序员会迷惑构造函数是否会创建对象。它们不会 – 编译器设置好对象的内存分配先于构造函数的调用。
构造函数实际上服务于两个目的。

1. 构造函数决定谁被允许创建一个类（class）类型的对象。也就是说一个类的对象仅可以在匹配构造函数时被创建。
2. 构造函数可以用于初始化对象。一个构造函数是否初始化是由程序员所决定的。在语法上让一个构造函数不进行初始化是合法的（构造函数仍然允许对象被创建）。

然而，与初始化所有本地变量的最佳实践类似，在创建对象时初始化所有成员变量也是最佳实践。这可以通过一个构造函数或者非 static 成员初始化完成。
最佳实践：总是初始化对象中的所有成员变量。
最后，构造函数仅意在创建对象时用于初始化。用户不应该调用一个构造函数来重新初始化一个已存在的对象。虽然它有可能通过编译，但是结果可能并不是用户预期的那样（相反的是，编译器将创建一个临时对象然后再销毁它）。

13.6 Constructor member initializer lists

上一章节中，为了简化，在构造函数中使用了分配操作符用于初始化类成员。

```
class Something
{
private:
    int m_value1 {};
    double m_value2 {};
    char m_value3 {};

public:
    Something()
    {
        // 这些都是赋值，不是初始化
        m_value1 = 1;
        m_value2 = 2.2;
        m_value3 = 'c';
    }
};
```

当类的构造函数被执行时，m_value1，m_value2 以及 m_value3 被创建。接着构造函数的函数体运行使成员变量被赋值。这就类似于以下代码的非面向对象 C++ 的流程：

```
int m_value1 {};
double m_value2 {};
```

```
char m_value3 {};

m_value1 = 1;
m_value2 = 2.2;
m_value3 = 'c';
```

虽然 C++ 中是合法的语法，但是它并不是一个好的风格（同时比起初始化要低性能）。
然而在上一章节中，一些数据类型（例如 const 与引用变量）必须在其声明处进行初始化，考虑以下代码：

```
class Something
{
private:
    const int m_value;

public:
    Something()
    {
        m_value = 1; // 错误: const 变量不能被赋值
    }
};
```

在构造函数中，对 const 或者引用成员变量赋值是不可能的。

成员初始化列表

为了解决这个问题，C++ 提供了一种名为*成员初始化列表* member initializer list（通常被称为“member initialization list”）的初始化类成员变量的方法（与其在变量创建后赋值）。不要把它们与用于对数组赋值的初始化列表搞混淆。
在 1.4 章节中讲解了可以通过三种方法初始化变量：拷贝，直接，以及标准化初始化。

```
int value1 = 1;     // copy initialization
double value2(2.2); // direct initialization
char value3 {'c'};  // uniform initialization
```

使用一个初始化列表几乎与直接初始化或标准化初始化相同。
使用初始化列表改写上面的代码：

```
#include <iostream>

class Something
{
private:
    int m_value1 {};
    double m_value2 {};
    char m_value3 {};

public:
```

```
    Something() : m_value1{ 1 }, m_value2{ 2.2 }, m_value3{ 'c' } // 初始化成员变量
    {
    // 这里不再需要赋值
    }

    void print()
    {
         std::cout << "Something(" << m_value1 << ", " << m_value2 << ", " << m_value3 << ")\n"
    ;
    }
};

int main()
{
    Something something{};
    something.print();
    return 0;
}
```

成员初始化列表在构造函数参数后参数插入。由冒号（:）开始，接着列出每个初始化的变量，且由逗号分隔。

注意在构造函数体内不再需要赋值，因为初始化列表替换了这个功能。同时注意初始化列表没有分号结尾。

当然了，当允许调用者传递初始化值时，构造函数更有用了：

```
#include <iostream>

class Something
{
private:
    int m_value1 {};
    double m_value2 {};
    char m_value3 {};

public:
    Something(int value1, double value2, char value3='c')
        : m_value1{ value1 }, m_value2{ value2 }, m_value3{ value3 } // 直接初始化成员变量
    {
    // 这里不再需要赋值
    }

    void print()
    {
         std::cout << "Something(" << m_value1 << ", " << m_value2 << ", " << m_value3 << ")\n"
    ;
    }

};
```

```
23
24 int main()
25 {
26     Something something{ 1, 2.2 }; // value1 = 1, value2=2.2, value3 为默认值 'c'
27     something.print();
28     return 0;
29 }
```

最佳实践：使用成员初始化列表来初始化类成员变量，而不是赋值。

初始化 const 成员变量

类可以包含 const 成员变量，它们与普通的 const 变量表现一致 – 必须被初始化，同时它们的值之后不能再被改变。
可以使用构造函数的成员初始化列表来初始化 const 成员（与非 const 成员相似），同时初始化值可以为常量或非常量。

```
1  #include <iostream>
2
3  class Something
4  {
5  private:
6      const int m_value;
7
8  public:
9      Something(int x) : m_value{ x } // 直接初始化 const 成员
10     {
11     }
12
13     void print()
14     {
15         std::cout << "Something(" << m_value << ")\n";
16     }
17 };
18
19 int main()
20 {
21     std::cout << "Enter an integer: ";
22     int x{};
23     std::cin >> x;
24
25     Something s{ x };
26     s.print();
27
28     return 0;
29 }
```

规则：const 成员变量必须被初始化。

通过成员初始化列表初始化数组成员

考虑一个带有数字成员的类:

```
class Something
{
private:
    const int m_array[5];

};
```

在 C++11 之前，用户只能通过成员初始化列表来零初始化一个数组成员:

```
class Something
{
private:
    const int m_array[5];

public:
    Something(): m_array {} // 零初始化数组成员
    {
    }

};
```

而从 C++11 开始，用户可以使用标准化初始化来初始化一个数组成员:

```
class Something
{
private:
    const int m_array[5];

public:
    Something(): m_array { 1, 2, 3, 4, 5 } // 使用标准化初始化来初始化一个数组成员
    {
    }

};
```

初始化成员变量为类

成员初始化列表同样也可以用于初始化类成员:

```
#include <iostream>

class A
{
public:
    A(int x = 0) { std::cout << "A " << x << '\n'; }
};
```

```

class B
{
private:
    A m_a {};
public:
    B(int y)
        : m_a{ y - 1 } // 调用 A(int) 构造函数来初始化成员 m_a
    {
        std::cout << "B " << y << '\n';
    }
};

int main()
{
    B b{ 5 };
    return 0;
}
```

格式化初始化列表

C++ 提供给用户了很多关于如何格式化初始化列表的灵活性，并且可以完全随用户的喜好。如果初始化列表与函数在同一列，那么把所有东西放在一行是可以的：

```
class Something
{
private:
    int m_value1 {};
    double m_value2 {};
    char m_value3 {};

public:
    Something() : m_value1{ 1 }, m_value2{ 2.2 }, m_value3{ 'c' } // 所有东西在一行
    {
    }
};
```

如果初始化列表没有与函数名同一行，那么应该缩进在下一行：

```
class Something
{
private:
    int m_value1;
    double m_value2;
    char m_value3;

public:
    Something(int value1, double value2, char value3='c') // 这一行已经有很多东西了
```

```
        : m_value1{ value1 }, m_value2{ value2 }, m_value3{ value3 } // 所以可以把所有东西缩进再下一行
    {
    }

};
```

如果初始化不在单独一行（或者初始化很重要），则可以每个都占据一行：

```
class Something
{
private:
    int m_value1 {};
    double m_value2 {};
    char m_value3 {};
    float m_value4 {};

public:
    Something(int value1, double value2, char value3='c', float value4=34.6f) // 这一行已经有很多东西了
        : m_value1{ value1 } // 每个都占据一行
        , m_value2{ value2 }
        , m_value3{ value3 }
        , m_value4{ value4 }
    {
    }

};
```

初始化列表顺序

出乎意料的是，在初始化列表中的变量初始化的顺序与指定的顺序并不相同。而是根据它们如何在类中声明的顺序一致。
为了更好的结果，下列建议应该被遵守：

1. 在成员变量依赖其他成员变量先初始化时，不要使用初始化列表（换言之，确保成员变量能正确的初始化即使初始化顺序是不同的）。

2. 初始化列表中的初始化变量顺序应该与类中声明的顺序一致。这并非是严格需要的，只要前一条建议被遵循，但在所有警告都打开的情况下，编译器可能会给出顺序不一致的警告。

总结

成员初始化列表允许用户初始化成员而不是赋值成员。这是唯一初始化成员需要值的方法，例如 const 或者引用成员，同时相较于在构造函数体内赋值有更良好的性能。成员初始化列表可以同时作用于基础类型以及类成员。

13.7 Non-static member initialization

当编写一个拥有若干构建函数的类时，在每个构造函数中都指定默认值则导致了冗余代码。如果更新一个成员的默认值，则必须更新所有构造函数。
那么可以给定普通函数成员变量（非 static 关键字）默认初始值：

```
#include <iostream>

class Rectangle
{
private:
    double m_length{ 1.0 }; // m_length 默认值为 1.0
    double m_width{ 1.0 };  // m_width 默认值为 1.0

public:
    void print()
    {
        std::cout << "length: " << m_length << ", width: " << m_width << '\n';
    }
};

int main()
{
    Rectangle x{}; // x.m_length = 1.0, x.m_width = 1.0
    x.print();

    return 0;
}
```

非 static 成员初始化（也被称为 in-class 成员初始化）提供默认值使得构造函数在没有初始化值的时候使用它们。
然而注意构造函数仍然决定哪种对象会被创建。考虑以下代码：

```
#include <iostream>

class Rectangle
{
private:
    double m_length{ 1.0 };
    double m_width{ 1.0 };

public:

    // 注意：本例子没有提供默认构造函数

    Rectangle(double length, double width)
        : m_length{ length },
          m_width{ width }
    {
```

```
        // m_length 与 m_width 通过构造函数初始化（默认值并未被使用）
    }

    void print()
    {
        std::cout << "length: " << m_length << ", width: " << m_width << '\n';
    }

};

int main()
{
    Rectangle x{}; // 不能编译，没有默认构造函数存在，即使成员都拥有默认的初始化值

    return 0;
}
```

如果提供了默认初始化值，同时构造函数通过成员初始化列表初始化了成员，则成员初始化列表优先。

```
#include <iostream>

class Rectangle
{
private:
    double m_length{ 1.0 };
    double m_width{ 1.0 };

public:

    Rectangle(double length, double width)
        : m_length{ length },
          m_width{ width }
    {
        // m_length 与 m_width 通过构造函数初始化（默认值并未被使用）
    }

    Rectangle(double length)
        : m_length{ length }
    {
        // m_length 有构造函数初始化
        // m_width 使用默认值 (1.0)
    }

    void print()
    {
        std::cout << "length: " << m_length << ", width: " << m_width << '\n';
    }

```

```
};

int main()
{
    Rectangle x{ 2.0, 3.0 };
    x.print();

    Rectangle y{ 4.0 };
    y.print();

    return 0;
}
```

注意初始化成员使用非 static 成员初始化必须要使用等号或者是花括号（标准化）初始化 – 圆括号初始化在这里不生效：

```
class A
{
    int m_a = 1;  // 可以（拷贝初始化）
    int m_b{ 2 }; // 可以（标准初始化）
    int m_c(3);   // 不可以（圆括号初始化）
};
```

规则：推荐使用非 static 成员初始化为成员变量提供默认值。

13.8 Overlapping and delegating constructors

带有重复功能的构造函数

当实例化一个新对象时，对象的构造函数是隐式调用的。对于一个类拥有若干重复功能的构造函数是很常见的。考虑以下代码：

```
class Foo
{
public:
    Foo()
    {
        // code to do A
    }

    Foo(int value)
    {
        // code to do A
        // code to do B
    }
};
```

该类拥有两个构造函数：一个默认构造函数，一个带有 int 入参的构造函数。因为“code to do A”的构造函数在两个构造函数中都被需要，这里代码就重复了。

显而易见的方案并不能工作

```
class Foo
{
public:
    Foo()
    {
        // code to do A
    }

    Foo(int value)
    {
        Foo(); // 使用上述构造函数完成 A（不工作）
        // code to do B
    }
};
```

这是因为 `Foo()` 实例化了一个新的 `Foo` 对象，接着立刻被销毁。

委托构造函数

构造函数允许调用同一个类下的构造函数。这个过程被称为**委托构造函数** delegating constructors（或者 **constructor chaining**）。
要让一个构造函数调用同一个类下的另一个构造函数，只需要简单的在成员初始化列表中调用即可。这是直接调用另一个构造函数的可以接受的方案：

```
class Foo
{
private:

public:
    Foo()
    {
        // code to do A
    }

    Foo(int value): Foo{} // 使用 Foo() 默认构造函数完成 A
    {
        // code to do B
    }

};
```

另一个使用委托构造函数的例子：

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

```

class Employee
{
private:
    int m_id{};
    std::string m_name{};

public:
    Employee(int id=0, std::string_view name=""):
        m_id{ id }, m_name{ name }
    {
        std::cout << "Employee " << m_name << " created.\n";
    }

    // 使用委托构造函数来最小化冗余代码
    Employee(std::string_view name) : Employee{ 0, name }
    { }
};
```

这里有几点关于委托构造函数需要注意的地方。首先，使用委托构造函数的构造函数自身不允许做任何成员初始化。因此构造函数可以是委托或者是初始化，而不能是同时两者。
其次一个构造函数委托另一个构造函数，可以被委托回到第一个构造函数。这就形成了无限循环，将导致程序用尽栈空间并崩溃。确保所有委托构造函数使用的都是非委托构造函数，可以避免这种情况。
最佳实践：如果有若干拥有相同功能的构造函数，使用委托构造函数来避免重复代码。

使用普通成员函数进行设置

由于构造函数仅可以初始化或者委托，那么如果默认构造函数做了一些共同的初始化，这就带来了挑战。考虑以下代码：

```
class Foo
{
private:
    const int m_value { 0 };

public:
    Foo()
    {
         // 代码用于做一些共同的设置任务（例如打开文件夹或者连接数据库）
    }

    Foo(int value) : m_value { value } // 必须初始化 m_value 因为它是 const
    {
        // 那么如何获取 Foo() 中共同的初始化代码呢 ？
    }

```

```
};
```

`Foo(int)` 构造函数仅可以初始化 `m_value`，或者是委托 `Foo()` 访问设置代码

```
#include <iostream>

class Foo
{
private:
    const int m_value { 0 };

    void setup() // setup 是私有的，因此仅可在构造函数中使用
    {
        // 代码用于做一些共同的设置任务（例如打开文件夹或者连接数据库）
        std::cout << "Setting things up...\n";
    }

public:
    Foo()
    {
        setup();
    }

    Foo(int value) : m_value { value } // 必须初始化 m_value 因为它是 const
    {
        setup();
    }

};

int main()
{
    Foo a;
    Foo b{ 5 };

    return 0;
}
```

重置类对象

有时候可能会发现编写一个成员函数（例如名为 `reset()`）用来重置一个类对象至初始状态。由于有可能已经拥有了默认构造函数用于初始化，用户可能会尝试直接在 `reset()` 函数中调用该默认构造函数。然而直接调用会带来未定义行为，正如之前所示那样，因此这是行不通的。如果一个类是可赋值的（意味着它拥有一个可用的赋值操作符），那么可以创建一个新的类对象，然后覆盖原有值：

```
#include <iostream>

```

```
class Foo
{
private:
    int m_a{ 5 };
    int m_b{ 6 };


public:
    Foo()
    {
    }

    Foo(int a, int b)
        : m_a{ a }, m_b{ b }
    {
    }

    void print()
    {
        std::cout << m_a << ' ' << m_b << '\n';
    }

    void reset()
    {
        *this = Foo(); // 创建新 Foo 对象，接着使用赋值来覆盖隐式对象
    }
};

int main()
{
    Foo a{ 1, 2 };
    a.reset();

    a.print();

    return 0;
}
```

关联内容：`this` 指针将在 13.10 中覆盖，对类进行赋值将在 14.15 中覆盖。

13.9 Destructors

在类中，**析构函数** destructor 是另一种在类对象被销毁时调用的特殊成员函数。构造函数被设计用于初始化一个类，而析构函数被设计用于帮助清理。

当一个对象普通的离开作用域，或者使用 `delete` 关键字删除动态分配的对象时，类的析构函数会自动被调用（如果存在的话），用于在对象从内存中被移除之前，清理任何必要的内容。对于简单类而言（那些仅初始化普通成员变量的），结构函数是不需要的，因为 C++ 将自动清除

内存。
然而如果类对象存储了任何资源（例如动态内存，或者文件，或者数据库连接），或者用户需要在对象被摧毁之前做任何的维护，那么结构函数正是最佳的地方，因为它通常是对象被摧毁前最后的一件事。

析构函数命名

与构造函数类似，结构函数也有特定的规则：

- 必须与类名称相同，并在前加上波浪号（~）。
- 不能带有参数。
- 没有返回值。

通常来说用户不应该直接显式的调用结构函数（因为它在对象销毁之前会被自动调用），因为只有很罕见的情况需要对一个对象进行若干次清理。然而，结构函数可能安全的调用其它成员函数，因为对象并没有被摧毁，直到析构函数被执行完毕。

析构函数案例

```
#include <iostream>
#include <cassert>
#include <cstddef>

class IntArray
{
private:
  int* m_array{};
  int m_length{};

public:
  IntArray(int length) // 构造函数
  {
    assert(length > 0);

    m_array = new int[static_cast<std::size_t>(length)]{};
    m_length = length;
  }

  ~IntArray() // 析构函数
  {
    // 动态删除之前分配过的数组
    delete[] m_array;
  }
```

```
25
26    void setValue(int index, int value) { m_array[index] = value; }
27    int getValue(int index) { return m_array[index]; }
28
29  int getLength() { return m_length; }
30  };
31
32  int main()
33  {
34    IntArray ar ( 10 ); // 分配 10 个整数
35    for (int count{ 0 }; count < ar.getLength(); ++count)
36      ar.setValue(count, count+1);
37
38    std::cout << "The value of element 5 is: " << ar.getValue(5) << '\n';
39
40    return 0;
41  } // ar 在这里被摧毁，因此 ~IntArray() 析构函数在这里被调用
```

构造函数与析构函数的时机

正如之前提到的，构造函数在对象创建时被调用，析构函数在对象被摧毁时调用。下面例子中，在它们中使用 `cout` 声明来验证：

```
1  #include <iostream>
2
3  class Simple
4  {
5  private:
6      int m_nID{};
7
8  public:
9      Simple(int nID)
10         : m_nID{ nID }
11     {
12         std::cout << "Constructing Simple " << nID << '\n';
13     }
14
15     ~Simple()
16     {
17         std::cout << "Destructing Simple" << m_nID << '\n';
18     }
19
20     int getID() { return m_nID; }
21 };
22
23 int main()
24 {
25     // 在栈上分配一个 Simple
```

```
    Simple simple{ 1 };
    std::cout << simple.getID() << '\n';

    // 动态分配一个 Simple
    Simple* pSimple{ new Simple{ 2 } };

    std::cout << pSimple->getID() << '\n';

    // 之前动态分配的 pSimple，因此需要删除它
    delete pSimple;

    return 0;
} // simple 在这里离开作用域
```

上述代码打印：

```
Constructing Simple 1
1
Constructing Simple 2
2
Destructing Simple 2
Destructing Simple 1
```

RAII

RAII (Resource Acquisition Is Initialization) 是一种编程技术，凭此技术资源的使用与对象的生命周期自动绑定（例如，非动态分配对象）。在 C++ 中，RAII 通过类的构造函数与析构函数来实现。一个资源（例如内存，文件或者数据库连接）通常在对象的构造函数中是需要的（通过构造函数，在对象创建后生成是符合逻辑的）。该资源在对象存活期间是可使用的，在对象被销毁时释放于析构函数中。RAII 最关键的优点就是在拥有资源的对象自动进行清理，防止资源泄漏（例如内存没有被释放）。

13.10 The hidden ”this” pointer

一个经常被面向对象新手问到的问题，“当一个成员函数被调用时，C++ 是如何追踪是哪个对象被调用了呢？”。答案就是 C++ 利用了一个隐藏的名为“this”的指针！

被隐藏的 *this 指针

```
simple.setID(2);
```

尽管调用 `setID()` 函数看起来像是只有一个参数，实际上有两个！当被编译时，编译器转换 `simple.setID(2);` 成为：

```
setID(&simple, 2); // 注意 simple 从前置的一个对象变为了一个函数的入参！
```

注意这仅仅是一个标准的函数调用，同时 simple 对象现在被地址传递成为函数的一个入参。不过这也只是一半答案。因为函数调用现在有了额外的参数，成员函数定义需要进行修改来接受（并使用）这个参数。结果就是以下的成员函数：

```
void setID(int id) { m_id = id; }
```

被转换为：

```
void setID(Simple* const this, int id) { this->m_id = id; }
```

当编译器编译普通成员函数时，它隐式添加名为“this”的新参数给函数。**this 指针 ** 是一个隐藏的 const 指针用于存储被调用函数的对象的地址。
这里还有一个细节需要关注。那就是在成员函数内部，任何类成员（函数与变量）同样也需要更新，这样它们才能指向被调用成员函数的对象。在它们每个前添加“this ->”前缀很容易。因此函数 `setID()`，`m_id`（类成员变量）可以被转换为 `this->m_id`。因此当“this”指向 simple 的地址，this->m_id 成为 shimple.m_id。
把它们全部联系起来就是：

1. 当调用 `simple.setID(2)` 时，编译器实际上调用的是 `setID(&simple, 2)`。
2. 在 `setID()` 中，“this”指针存储了 simple 对象的地址。
3. 任何在 `setID()` 中的成员需要“this->”前缀。因此当 `m_id = id` 时，编译器实际上执行的是 `this->m_id = id`，这种情况下将 simple.m_id 更新至 id。

“this”总是指向其被操作的对象

```
int main()
{
    Simple A{1};  // this = &A 在 Simple 的构造函数内
    Simple B{2};  // this = &B 在 Simple 的构造函数内
    A.setID(3);   // this = &A 在成员函数 setID 内
    B.setID(4);   // this = &B 在成员函数 setID 内

    return 0;
}
```

显式引用“this”

```
class Something
{
private:
    int data;

```

```
public:
    Something(int data)
    {
        this->data = data; // this->data 为成员，data 为本地参数
    }
};
```

串联成员函数

```
#include <iostream>

class Calc
{
private:
    int m_value{};

public:
    Calc& add(int value) { m_value += value; return *this; }
    Calc& sub(int value) { m_value -= value; return *this; }
    Calc& mult(int value) { m_value *= value; return *this; }

    int getValue() { return m_value; }
};

int main()
{
    Calc calc{};
    calc.add(5).sub(3).mult(4);

    std::cout << calc.getValue() << '\n';
    return 0;
}
```

13.11 Class code and header files

在类定义外定义成员函数

在类变得更长以及更复杂的时候，所有成员函数定义在类的内部会导致类变得难以管理以及使用。使用一个已经写好的类只需要明白其共有接口（共有成员函数），而不是类在底下是怎么工作的。成员函数实现的细节正是这些。

幸运的是 C++ 提供一种分离类的“声明”部分与“实现”部分的的方法。这便是定义类成员函数在类所定义的外部。为了这么做，只需要简单的使用普通函数那样去定义成员函数，并使得类名称作为该函数的前缀，并由操作符（:: ）隔开。

```
class Date
```

```
{
private:
    int m_year;
    int m_month;
    int m_day;

public:
    Date(int year, int month, int day);

    void SetDate(int year, int month, int day);

    int getYear() { return m_year; }
    int getMonth() { return m_month; }
    int getDay()  { return m_day; }
};

// Date 构造函数
Date::Date(int year, int month, int day)
{
    SetDate(year, month, day);
}

// Date 成员函数
void Date::SetDate(int year, int month, int day)
{
    m_month = month;
    m_day = day;
    m_year = year;
}
```

这以及很直接了。因为访问函数通常只需要一行，它们通常留在内定义里，即使它们可以被移到外面去。

这里是另一个例子包含了通过成员初始化列表定义的外部定义的构造函数：

```
class Calc
{
private:
    int m_value = 0;

public:
    Calc(int value=0);

    Calc& add(int value);
    Calc& sub(int value);
    Calc& mult(int value);

    int getValue() { return m_value; }
};

```

```
Calc::Calc(int value): m_value{value}
{
}

Calc& Calc::add(int value)
{
    m_value += value;
    return *this;
}

Calc& Calc::sub(int value)
{
    m_value -= value;
    return *this;
}

Calc& Calc::mult(int value)
{
    m_value *= value;
    return *this;
}
```

放置类定义于头文件中

按照传统，类定义放在与类同名的头文件中，而定义在类外部的成员函数放在与类同名的.cpp文件中。

Date.h:

```
#ifndef DATE_H
#define DATE_H

class Date
{
private:
    int m_year;
    int m_month;
    int m_day;

public:
    Date(int year, int month, int day);

    void SetDate(int year, int month, int day);

    int getYear() { return m_year; }
    int getMonth() { return m_month; }
    int getDay()  { return m_day; }
};

```

```
#endif
```

Date.cpp:

```
#include "Date.h"

// Date 构造函数
Date::Date(int year, int month, int day)
{
    SetDate(year, month, day);
}

// Date 成员函数
void Date::SetDate(int year, int month, int day)
{
    m_month = month;
    m_day = day;
    m_year = year;
}
```

现在任何其他头文件或者代码文件希望使用 Date 类仅需简单的 `#include "Date.h"` 。注意 Date.cpp 同样也需要被编译到任何使用了 Date.h 的项目中，这样 linker 才能知道 Date 是如何实现的。

定义类在一个头文件中会违反 one-definition 规则吗?

这并不会。如果头文件拥有正确的头文件保护符，它不可能在同一文件中被引入多次。

类型（包括类）是免于 one-definition 规则的。因此 #including 类进若干代码文件中不会有问题（如果有问题，类就没有那么有用了）。

定义成员函数在头文件中会违反 one-definition 规则吗?

这得看情况。成员函数定义在类定义处被视为隐式内联。内联函数免于 one-definition 规则。这意味着定义小型的成员函数（例如访问函数）在类定义处是没有问题的。

成员函数定义在类的外部被视为普通函数，也服从了 one-definition 的规则。因此，这些函数应该被定义于代码文件，而不是头文件。有一个例外是模板函数，它也被视为隐式内联。

那么什么是应该定义在头文件 vs cpp 文件，以及什么是在类定义内 vs 外部?

将所有成员函数定义在头文件的类中，虽然可以编译，还是有很多弊端。首先如上提到的，类定义被塞满了很凌乱。其次，如果修改了任何头文件中的代码，则需要重新编译所有引用该头文件的文件。这会带来涟漪效果，即一个微小的改动会导致整个程序需要被重新编译（很慢）。如果只是修改了.cpp 文件的代码，只需要重新编译该文件即可!

因此，推荐以下做法:

- 对于仅使用在一个文件中并没有复用的类，定义在需要使用的单个.cpp 文件中。
- 对于需要在若干文件使用的类或者是普通的复用，定义在与类名称相同的.h 文件中。

- 小型的成员函数（小型的构造函数或析构函数，访问函数，等等）可以在类中定义。
- 非小型成员函数应该定义在与类同名的.cpp 文件中。

成员函数的默认参数应该被定义在类定义处（在头文件中），它们可以被任何 #includes 该头文件的看到。
在编写库时，分隔类定义与类实现是非常常见的。
除开开源软件（.h 与.cpp 文件同时被提供），大多数三方库仅提供头文件，以及预编译的库文件。这么做有以下几个原因：1）link 预编译库要比每次需要重新编译快很多；2）单个预编译的库可以被很多应用共享，而编译的代码被编译成每一个使用它的可执行文件（文件大小膨胀）；3）知识版权（不希望代码被偷窃）。
将文件分割成声明（头文件）以及实现（代码文件）不仅仅是好的形式，在创建用户自定义库的时候也变得更简单了。

13.12 Const class objects and member functions

Const 类

实例化对象的时候也可以使用 const 关键字。初始化由类的构造函数完成。

```
const Date date1;                   // 使用默认构造函数进行初始化
const Date date2(2020, 10, 16);     // 带参数初始化
const Date date3 { 2020, 10, 16 };  // 带参数初始化 (C++11)
```

一旦一个 const 类对象通过构造函数被初始化，尝试修改成员变量是不被允许的，因为这违背了 const 对象规则。这里包含了直接修改成员变量（如果它们是共有的），或者是调用成员函数设置成员变量。

```
class Something
{
public:
    int m_value {};

    Something(): m_value{0} { }

    void setValue(int value) { m_value = value; }
    int getValue() { return m_value ; }
};

int main()
{
    const Something something{}; // 调用默认构造函数

    something.m_value = 5; // 编译错误：违背 const
    something.setValue(5); // 编译错误：违背 const
```

```
18
19      return 0;
20  }
```

Const 成员函数

一个 **const 成员函数**确保不会修改对象或者调用任何非 const 成员函数（因为它们有可能修改对象）。

```
1  class Something
2  {
3  public:
4      int m_value {};
5
6      Something(): m_value{0} { }
7
8      void resetValue() { m_value = 0; }
9      void setValue(int value) { m_value = value; }
10
11     int getValue() const { return m_value; } // 注意 const 关键字在参数列表后，函数体前
12 };
```

对于成员函数定义在类定义外的情况，const 关键字必须同时定义在类定义的函数协议上，以及函数定义上。

```
1  class Something
2  {
3  public:
4      int m_value {};
5
6      Something(): m_value{0} { }
7
8      void resetValue() { m_value = 0; }
9      void setValue(int value) { m_value = value; }
10
11     int getValue() const; // 注意这里额外的 const 关键字
12 };
13
14 int Something::getValue() const // 以及这里
15 {
16     return m_value;
17 }
```

另外，任何 const 成员函数尝试修改一个成员变量或者调用非 const 成员函数会导致变异错误。

```
1  class Something
2  {
3  public:
4      int m_value {};
```

```
    void resetValue() const { m_value = 0; } // 编译错误，const 函数不能修改成员变量
};
```

最佳实践：令任何不修改类变量的成员函数为 const，这样它们可以被调用为 const 对象。

传递 const 引用用于创建 const 对象

尽管实例化 const 类对象是一种创造 const 对象的方法，另一种常用的方法是传递 const 引用对象用于创建 const 对象。

```
class Date
{
private:
    int m_year {};
    int m_month {};
    int m_day {};

public:
    Date(int year, int month, int day)
    {
        setDate(year, month, day);
    }

    // setDate() cannot be const, modifies member variables
    void setDate(int year, int month, int day)
    {
        m_year = year;
        m_month = month;
        m_day = day;
    }

    // The following getters can all be made const
    int getYear() const { return m_year; }
    int getMonth() const { return m_month; }
    int getDay() const { return m_day; }
};

// note: We're passing date by const reference here to avoid making a copy of date
void printDate(const Date& date)
{
    std::cout << date.getYear() << '/' << date.getMonth() << '/' << date.getDay() << '\n';
}

int main()
{
    Date date{2016, 10, 16};
    printDate(date);

```

```
    return 0;
}
```

const 成员不可以返回非 const 成员的引用

当一个成员函数是 const 时，隐藏的 this 指针同样也是 const 的，这就意味着所有在函数内的成员都被视为 const。因此，一个 const 成员函数不可以返回一个非 const 成员的引用，因为这样允许了调用者使用非 const 访问该 const 成员。const 成员函数可以返回 const 成员的引用。详见下一章节。

重载 const 与非 const 函数

最后，尽管这么做不是很常见，还是有可能会重载一个函数变成带有 const 与非 const 版本的同一函数。这可以工作是因为 const 限定符被视为函数签名的一部分，因此两个函数的区别只在于只因 const 视为不同。

```
#include <string>

class Something
{
private:
    std::string m_value {};

public:
    Something(const std::string& value=""): m_value{ value } {}

    const std::string& getValue() const { return m_value; } // getValue() 的 const 对象（返回 const 引用）
    std::string& getValue() { return m_value; } // getValue() 的非 const 对象（返回非 const 引用）
};
```

那么 const 版本的函数将会被调用于任何 const 对象，而非 const 版本将会被调用与任何非 const 对象：

```
int main()
{
  Something something;
  something.getValue() = "Hi"; // 调用非 const getValue()

  const Something something2;
  something2.getValue(); // 调用 const getValue()

  return 0;
}
```

13.13 Static member variables

之前的章节中讲到过 static 变量会一直保留其值，即使在离开作用域后也不会被销毁。
static 关键之应用在全局变量时拥有了另一个含义 – 给予它们 internal linkage（即限制它们在所定义的文件范围外可视/使用）属性。因为全局变量通常来说是需要避免的，因此 static 关键之很少用作这里。

static 成员变量

在应用于类的时候，static 关键字多了两种用法：static 成员变量，以及 static 成员函数。幸运的是这俩用法都很直接。在这一章节里将讲解 static 成员变量，下一章节讲解 static 成员函数。

```
#include <iostream>

class Something
{
public:
    static int s_value;
};

int Something::s_value{ 1 };

int main()
{
    Something first;
    Something second;

    first.s_value = 2;

    std::cout << first.s_value << '\n';
    std::cout << second.s_value << '\n';
    return 0;
}
```

打印：

```
2
2
```

static 成员并不与类对象关联

尽管用户可以通过类的对象来访问 static 成员，实际上在没有任何该类的对象实例化时 static 成员依然存在！类似于全局变量，它们在程序启动时就被创建，在程序结束时被销毁。
因此，最好是将 static 成员视为属于类的本身，而不是该类的对象。因为 ‘s_value‘ 独立于任何类对象，它可以直接使用类名称以及作用域解析操作符来进行访问（即 ‘Something::s_value‘）：

```cpp
#include <iostream>

class Something
{
public:
    static int s_value; // 声明 static 成员变量
};

int Something::s_value{ 1 }; // 定义 static 成员变量（稍后讨论）

int main()
{
    // 注意：这里没有实例化任何 Something 类型的对象

    Something::s_value = 2;
    std::cout << Something::s_value << '\n';
    return 0;
}
```

最佳实践：通过类名称访问 static 成员（使用作用域解析操作符）而不是通过类对象来进行访问（使用成员选择操作符）。

定义以及初始化 static 成员变量

当在类的内部定义一个 static 成员变量，那么就是在告诉编译器 static 成员变量的存在，而不是真正的定义它（更像是前向声明）。因为 static 成员变量不是类对象的一部分（而是类似于被视为全局变量，并在程序启动时被初始化），用户必须显式的在类的外部定义 static 成员，即全局作用域。

上述的列子中，通过这一行：

```cpp
int Something::s_value{ 1 }; // 定义 static 成员变量
```

这一行代码有两个目的：实例化 static 成员变量（如同全局变量），并可选的初始化它。这个例子中，提供了初始化值 1. 如果没有提供初始化，C++ 将初始化值为 0。

注意这个 static 成员定义不属于访问控制：用户可以定义并初始化变量即使其定义在类中是私有的（或者受保护的）。

如果类是定义在.h 文件中，那么 static 成员定义通常要放在该类关联的代码文件中（例 Something.cpp）。如果类是定义在.cpp 文件中，那么 static 成员定义通常直接放在类的下方。不要将 static 成员定义在头文件中（类似于全局变量，如果头文件被多次引入，则会导致若干定义而产生 linker 错误）。

static 成员变量的内联初始化

上述方法存在一些捷径。

首先，当 static 成员是一个 const 整数类型（包括 char 与 bool）或者是一个 const 枚举，那么 static 成员可以在类定义内部初始化：

```
class Whatever
{
public:
    static const int s_value{ 4 }; // 一个 static const int 可以直接被定义以及初始化
};
```

其次，从 C++17 开始，可以通过内联声明的方法，初始化非 const static 成员在类定义中：

```
class Whatever
{
public:
    static inline int s_value{ 4 }; // 一个 static inline int 可以直接被定义以及初始化（C++17）
};
```

案例

为什么在类内部使用 static 变量呢？一个有用的案例是分配一个唯一 ID 给每个类的实例：

```
#include <iostream>

class Something
{
private:
    static inline int s_idGenerator { 1 };  // C++17
    // static int s_idGenerator;            // C++14 或更早
    int m_id { };

public:
    Something()
    : m_id { s_idGenerator++ } // 从 id 生成器中获取下一个值
    {}

    int getID() const { return m_id; }
};

// 如果是 C++14 或更早版本，用户需要在类定义的外部，初始化非 const static 成员。
// 注意这里定义并初始化了 s_idGenerator，即使上述的声明是私有的。
// 这样做是可以的，因为定义并不属于访问控制。
// int Something::s_idGenerator { 1 }; // 通过 ID 生成器获取值 1（C++14 或更早版本）

int main()
{
    Something first;
    Something second;
    Something third;

```

```
    std::cout << first.getID() << '\n';
    std::cout << second.getID() << '\n';
    std::cout << third.getID() << '\n';
    return 0;
}
```

打印:

```
1
2
3
```

因为 ‘s_idGenerator‘ 被所有的 ‘Something‘ 对象所共享，当 ‘Something‘ 对象被创建时，构造函数获取 ‘s_idGenerator‘ 当前值并为下一个对象增加值。这就保证了每个实例化的 ‘Something‘ 对象都获取了唯一 id（顺序创建的）。这样可以方便数组中包含若干项的调试，因为提供了一种用来分辨同样类型对象的方法!
static 成员变量在类需要利用内部检索表时也很有用（例如用于存储一系列预计算值的数组）。通过 static 可以使检索表在所有对象中只存在一份，而不是每个对象都实例化一份检索表。这样可以显著的减少内存使用。

13.14 Static member functions

上一章讲解了 static 成员变量，知道了它们是属于类的而不是该类对象的。如果 static 成员变量是公有的，则可以通过类名以及作用域解析操作符直接访问它们。那么如果 static 成员变量是私有的呢?
考虑以下例子:

```
class Something
{
private:
    static int s_value;

};

int Something::s_value{ 1 }; // 初始化，这是可以的，即使 s_value 的声明是私有的

int main()
{
    // 那么这里该如何访问私有的 Something::s_value 呢?
}
```

上述情况在 main 中不可以直接访问 `Something::s_value` ，因为它是私有的。通常而言可以通过公有成员函数来访问私有成员。而创建一个普通的公有成员函数来访问 `s_value` 首先需要初始化一个该类的对象才能使用该公有函数! 可以有更好的办法，那就是使函数 static。
类似于 static 成员变量，static 成员函数也没有依附于任何对象。

```cpp
#include <iostream>

class Something
{
private:
    static int s_value;
public:
    static int getValue() { return s_value; } // static 成员函数
};

int Something::s_value{ 1 }; // 初始化

int main()
{
    std::cout << Something::getValue() << '\n';
}
```

static 成员函数没有 *this 指针

static 成员函数有两个有趣的怪异之处值得注意。首先，因为 static 成员函数没有依附于任何对象，它们没有 *this* 指针！想象一下也很合理 – *this* 指针指向的总是对象。static 成员函数并不作用于对象，因此 *this* 指针是不需要的。
其次，static 成员函数可以直接访问其它的 static 成员（变量或函数），但是不能访问非 static 成员。这是因为非 static 成员必须属于一个类对象，而 static 成员函数与对象无关！

另一个案例

static 成员函数可以定义在类声明的外边。这与普通的成员函数一样：

```cpp
#include <iostream>

class IDGenerator
{
private:
    static int s_nextID; // static 成员的声明

public:
     static int getNextID(); // static 函数的声明
};

// 在类的外边定义 static 成员。注意这里不要使用 static 关键字。
// 在 1 生成 IDs
int IDGenerator::s_nextID{ 1 };

// 在类的外边定义 static 函数。注意这里不要使用 static 关键字。
int IDGenerator::getNextID() { return s_nextID++; }
```

```
18
19 int main()
20 {
21     for (int count{ 0 }; count < 5; ++count)
22         std::cout << "The next ID is: " << IDGenerator::getNextID() << '\n';
23
24     return 0;
25 }
```

打印:

```
1 The next ID is: 1
2 The next ID is: 2
3 The next ID is: 3
4 The next ID is: 4
5 The next ID is: 5
```

一个关于所有带有 static 成员的类的警告

注意当编写带有 static 成员的类时，尽管“pure static classes”（也被称为“monostates”）很有用，但是同时也带来了一些缺点。

首先，因为所有的 static 成员只会实例化一次，因此没有办法获得多个纯 static 类（不进行拷贝以及重命名类的情况下）。例如如果需要两个独立的 IDGenerator 对象，在一个纯 static 类的情况下，这是不可能的。

其次，全局变量是危险的，因为任何微小的代码可以修改其值，并破坏了另一处看似无关的代码。这同样也是纯 static 类的问题，因为所有成员都属于类（而不是类的对象），而类的声明通常拥有全局作用域，一个纯 static 类基本上等于把函数以及变量声明在了全局可访问的命名空间，也就带来了全局变量所拥有的缺点。

C++ 不支持 static 构造函数

如果通过构造函数可以初始换普通成员，那么对其延伸就意味着可以通过 static 构造函数来初始化 static 成员函数。虽然有一些现代语言确实支持 static 构造函数来达到这个目的，但是不幸的是 C++ 并不是它们这些现代语言其中之一。

如果 static 变量可以被直接初始化，则不需要构造函数：可以在定义的地方直接初始化 static 成员变量（即使是私有的）。上面的例子就是这么做的。以下是另一个例子：

```
1 class MyClass
2 {
3 public:
4   static std::vector<char> s_mychars;
5 };
6
7 std::vector<char> MyClass::s_mychars{ 'a', 'e', 'i', 'o', 'u' }; // 在定义处初始化 static 变量
```

如果初始化 static 成员变量需要执行代码（例如循环），则有很多不同的笨拙的方法来完成它。一种对所有变量，无论是否 static 的方法是使用 lambda 并立刻调用它。

```
class MyClass
{
public:
    static std::vector<char> s_mychars;
};

std::vector<char> MyClass::s_mychars{
  []{ // lambdas 在没有参数时可以省略参数列表
      // 在 lambda 内可以声明另一个 vector 并使用循环
      std::vector<char> v{};

      for (char ch{ 'a' }; ch <= 'z'; ++ch)
      {
          v.push_back(ch);
      }

      return v;
  }() // 立马调用 lambda
};
```

下列代码看起来更像是通常的构造函数。然而，它有一点诡异，且应当永远不需要它。

```
class MyClass
{
public:
  static std::vector<char> s_mychars;

  class init_static // 定义一个名为 init_static 的嵌套类
  {
  public:
    init_static() // 初始化构造函数初始化 static 变量
    {
      for (char ch{ 'a' }; ch <= 'z'; ++ch)
      {
        s_mychars.push_back(ch);
      }
    }
  };

private:
  static init_static s_initializer; // 使用这个 static 对象来确保 init_static 构造函数被调用
};

std::vector<char> MyClass::s_mychars{}; // 定义 static 成员变量
MyClass::init_static MyClass::s_initializer{}; // 定义 static 初始化，其调用 init_static 构造函
    数并初始化 s_mychars
```

当 static 成员 s_initializer 被定义，init_static() 默认构造函数会被调用（因为 s_initializer 类型为 init_static）。可以使用这个构造函数来初始化任何 static 成员变量。这个方法的好处是所有的初始化代码通过 static 成员都被隐藏在原始类中。

13.15 Friend functions and classes

从这一章开始就一直在宣传数据私有的概念。然而还是偶尔发现在某些情况下，有某类的外部进行更紧密的操作的操作需求。例如，希望一个类存储数据，一个函数（或者其它类）用于展示该数据。尽管存储类和展示部分的代码为了维护方便被分隔，后者与前者的细节联系的却非常紧密。就结果而言，展示部分的代码并没有从隐藏存储类细节中获益。
这种情况下有两种选择：

1. 展示部分的代码使用存储类暴露出来的公有函数。然而这有若干潜在的缺陷。首先公有成员函数必须被定义，这就耗费了时间，并且使得存储类的接口杂乱无章。其次，存储类在暴露给展示部分的代码时，可能需要一些根本不想暴露给其他使用者的函数。也就是没有办法做到“这个函数值暴露给展示类”。

2. 第二种选择就是使用友元类以及友元函数给与展示部分的代码访问存储类私有的细节。这样可以直接访问存储类的所有私有成员和函数，同时隔离其它的使用者！

友元函数

友元函数是一种被视为类成员并可以访问私有成员的函数。在所有其他方面，友元函数就像是普通函数那样。一个友元函数可以是一个普通函数，或者是另一个类的成员函数。要声明一个友元函数，仅需在函数原型前使用 *friend* 关键字。在类的私有或者公有部分声明友元函数并没差异。

```
class Accumulator
{
private:
    int m_value { 0 };

public:
    void add(int value) { m_value += value; }

    // 让 reset() 函数成为该类的友元函数
    friend void reset(Accumulator& accumulator);
};

// reset() 现在是 Accumulator 类的友元函数
void reset(Accumulator& accumulator)
{
    // 可以访问 Accumulator 对象的私有数据
```

```
17      accumulator.m_value = 0;
18  }
19
20  int main()
21  {
22      Accumulator acc;
23      acc.add(5); // accumulator 加 5
24      reset(acc); // 重置 accumulator 至 0
25
26      return 0;
27  }
```

这个例子中，声明了 `reset()` 函数获取 Accumulator 类的对象，设置 `m_value` 值为 `0`。因为 `reset()` 不是 Accumulator 类的成员，普通的 `reset()` 不能访问 Accumulator 的私有成员。然而因为 Accumulator 特别声明了 `reset()` 函数为该类的友元函数，`reset()` 函数被给予了访问 Accumulator 私有成员的权限。

注意这里传递了一个 Accumulator 对象给 `reset()`。这是因为 `reset()` 不是一个成员函数。它没有 *this 指针，也没有一个可用的 Accumulator 对象，除非提供了。

这里有另一个例子：

```
1   #include <iostream>
2
3   class Value
4   {
5   private:
6       int m_value{};
7
8   public:
9       Value(int value)
10          : m_value{ value }
11      {
12      }
13
14      friend bool isEqual(const Value& value1, const Value& value2);
15  };
16
17  bool isEqual(const Value& value1, const Value& value2)
18  {
19      return (value1.m_value == value2.m_value);
20  }
21
22  int main()
23  {
24      Value v1{ 5 };
25      Value v2{ 6 };
26      std::cout << std::boolalpha << isEqual(v1, v2);
27
```

```
    return 0;
}
```

这个例子中，声明了 `isEqual()` 函数为 Value 类的友元函数。`isEqual()` 获取两个 Value 对象作为参数。因为 `isEqual()` 是 Value 类的友元函数，它可以访问 Value 对象的所有私有成员。这个情况下，比较两个对象，返回 true 如果它们相等。

若干友元

一个函数还可以同时做若干类的友元函数：

```
#include <iostream>

class Humidity;

class Temperature
{
private:
    int m_temp {};

public:
    Temperature(int temp=0)
        : m_temp { temp }
    {
    }

    friend void printWeather(const Temperature& temperature, const Humidity& humidity);
};

class Humidity
{
private:
    int m_humidity {};

public:
    Humidity(int humidity=0)
        : m_humidity { humidity }
    {
    }

    friend void printWeather(const Temperature& temperature, const Humidity& humidity);
};

void printWeather(const Temperature& temperature, const Humidity& humidity)
{
    std::cout << "The temperature is " << temperature.m_temp <<
       " and the humidity is " << humidity.m_humidity << '\n';
}
```

```
38
39 int main()
40 {
41     Humidity hum{10};
42     Temperature temp{12};
43
44     printWeather(temp, hum);
45
46     return 0;
47 }
```

这里有两点需要注意的是：第一，因为 `printWeather` 是两个类的友元函数，它可以访问两者对象的私有数据。其次，注意第一行：

```
1 class Humidity;
```

这是一个类原型用于告诉编译器之后要定义一个 Humidity 的类。没有这一行的话编译器会说不知道什么是 Humidity。类的原型与函数原型的作用一致 – 告诉编译器有一个这样的东西并在之后会定义。然而不同于函数的是，类没有返回类型或者参数，所以类原型总是简单的 `class ClassName` 即可。

友元类

同样的，一整个类是另一个类的友元也是可以的。这样让所有友元类的成员都能访问另一个类的私有成员。

```
1  #include <iostream>
2
3  class Storage
4  {
5  private:
6      int m_nValue {};
7      double m_dValue {};
8  public:
9      Storage(int nValue, double dValue)
10         : m_nValue { nValue }, m_dValue { dValue }
11     {
12     }
13
14     // 使 Display 类成为 Storage 类的友元
15     friend class Display;
16 };
17
18 class Display
19 {
20 private:
21     bool m_displayIntFirst;
22
```

```
public:
    Display(bool displayIntFirst)
        : m_displayIntFirst { displayIntFirst }
    {
    }

    void displayItem(const Storage& storage)
    {
        if (m_displayIntFirst)
            std::cout << storage.m_nValue << ' ' << storage.m_dValue << '\n';
        else // 先展示 double
            std::cout << storage.m_dValue << ' ' << storage.m_nValue << '\n';
    }
};

int main()
{
    Storage storage{5, 6.7};
    Display display{false};

    display.displayItem(storage);

    return 0;
}
```

关于友元类还有额外的几点需要注意。首先，即使 `Display` 是 `Storage` 的友元类，`Display` 并没有直接访问 `Storage` 对象的 this 指针。其次，并不意味着 `Storage` 是 `Display` 的友元。如果想要两个类互为友元，那么需要互相声明对方为友元。最后，如果类 A 是类 B 的友元，类 B 是 C 的友元，那么也不意味着 A 是 C 的友元。

使用友元函数与类的时候要注意，因为它允许友元函数或类违反封装。如果类的细节被改变了，那么友元的细节也将被强制改变。最后，应该最小程度的限制使用友元函数或类。

友元成员函数

与其让整个类成为友元，可以让单个成员函数成为友元。这与让一个普通函数成为友元类似，不过使用的是带有 className::prefix 的成员函数的名称。

然而实际上这比预期的样子要奇怪一些。上述例子用 Display::displayItem 一个友元函数：

```
#include <iostream>

class Display; // Display 类的前向声明

class Storage
{
private:
  int m_nValue {};
```

```
  double m_dValue {};
public:
  Storage(int nValue, double dValue)
    : m_nValue { nValue }, m_dValue { dValue }
  {
  }

  // 使 Display::displayItem 成员函数作为 Storage 类的友元
  friend void Display::displayItem(const Storage& storage); // 错误：Storage 还未看到 Display 类的完整定义
};

class Display
{
private:
  bool m_displayIntFirst {};

public:
  Display(bool displayIntFirst)
    : m_displayIntFirst { displayIntFirst }
  {
  }

  void displayItem(const Storage& storage)
  {
    if (m_displayIntFirst)
      std::cout << storage.m_nValue << ' ' << storage.m_dValue << '\n';
    else
      std::cout << storage.m_dValue << ' ' << storage.m_nValue << '\n';
  }
};
```

为了使一个成员函数成为友元，编译器必须看到作为友元成员函数的那个类的所有定义（而不仅仅只是前向声明）。

幸运的是，解决这个问题最简单的做法就是将整个 `Display` 类的定义移动到 `Storage` 类的前方。

```
#include <iostream>

class Display
{
private:
  bool m_displayIntFirst {};

public:
  Display(bool displayIntFirst)
    : m_displayIntFirst { displayIntFirst }
  {
  }
```

```
13
14    void displayItem(const Storage& storage) // 错误：编译器并不知道什么是 Storage
15    {
16      if (m_displayIntFirst)
17        std::cout << storage.m_nValue << ' ' << storage.m_dValue << '\n';
18      else
19        std::cout << storage.m_dValue << ' ' << storage.m_nValue << '\n';
20  }
21  };
22
23  class Storage
24  {
25  private:
26    int m_nValue {};
27    double m_dValue {};
28  public:
29    Storage(int nValue, double dValue)
30      : m_nValue { nValue }, m_dValue { dValue }
31    {
32    }
33
34    // 使 Display::displayItem 成员函数作为 Storage 类的友元
35    friend void Display::displayItem(const Storage& storage); // 现在可以了
36  };
```

然而这有另一个问题。因为成员函数 `Display::displayItem()` 使用 `Storage` 作为引用参数，而刚刚将 `Storage` 的定义移动到了 `Display` 的后面。

幸运的是，只需要几步就可以解决这个问题。首先添加 `Storage` 类的前向声明。其次，移动 `Display::displayItem()` 的定义至类的外部，放在 `Storage` 类的完整定义之后。

```
1   #include <iostream>
2
3   class Storage; // Storage 类的前向声明
4
5   class Display
6   {
7   private:
8     bool m_displayIntFirst {};
9
10  public:
11    Display(bool displayIntFirst)
12      : m_displayIntFirst { displayIntFirst }
13    {
14    }
15
16    void displayItem(const Storage& storage); // 之前的前向声明在这里的使用需要
17  };
18
19  class Storage // Storage 类的全部定义
```

```
{
private:
  int m_nValue {};
  double m_dValue {};
public:
  Storage(int nValue, double dValue)
    : m_nValue { nValue }, m_dValue { dValue }
  {
  }

  // 使 Display::displayItem 成员函数成为 Storage 类的友元（需要看到 Display 类的所有定义）
  friend void Display::displayItem(const Storage& storage);
};

// 现在可以定义 Display::displayItem 了，它需要看到 Storage 类的所有定义
void Display::displayItem(const Storage& storage)
{
  if (m_displayIntFirst)
    std::cout << storage.m_nValue << ' ' << storage.m_dValue << '\n';
  else
    std::cout << storage.m_dValue << ' ' << storage.m_nValue << '\n';
}

int main()
{
  Storage storage(5, 6.7);
  Display display(false);

  display.displayItem(storage);

  return 0;
}
```

现在整个思路就清晰了：Storage 类的前向定义满足了声明 Display::displayItem() 的条件，接着 Display 的完整定义满足了声明 Display::displayItem() 作为 Storage 友元声明的条件，接着 Storage 类的完整定义满足了 Display::displayItem() 作为 Storage 成员函数的条件。

这看起来很痛苦 – 确实如此。幸运的是，需要这样的操作仅仅是因为想要定义所有的内容在单个文件中。一个更好的办法是将所有类定义都放在头文件中，而成员函数定义在相应的.cpp 文件中。这种方式，所有的类定义都讲可以立刻在.cpp 文件中看到，也就不再需要重新排列类或者函数了！

总结

友元函数/类是一种函数/类，被视为其他类成员，用于访问其他类私有成员的方式。这使得友元函数/类可以更加紧密的与其他类工作，而不需要其他类暴露自身的私有成员（例如通过访问函数）。
友元经常在定义重载操作符时使用（详见下一章），或者是两个或以上的类需要以一种更紧密的方式工作。
注意，在让指定的成员函数成为友元时，需要知道所有被友元的类的定义。

13.16 Anonymous objects

某些情况下，用户会需要一个临时变量，例如：

```
#include <iostream>

int add(int x, int y)
{
    int sum{ x + y };
    return sum;
}

int main()
{
    std::cout << add(5, 3) << '\n';

    return 0;
}
```

在 `add()` 函数中，注意 `sum` 变量仅仅用作于临时占位变量。它并没有多大贡献 – 相反，它仅作用于传递表达式的返回值。
其实还有一种更简单的方式，那就是使用匿名对象来编写 `add()` 函数。**匿名对象**本质上是一个无名的值。因为它们没有名字，所以没有办法在它们创建地之外引用它们。结果而言，它们拥有“表达式作用域”，意为它们被创建，计算，并摧毁在单个表达式中。
以下是使用匿名对象重写的 `add()` 函数：

```
#include <iostream>

int add(int x, int y)
{
    return x + y; // 匿名对象被创建用于存储并返回 x + y 的结果
}

int main()
{
    std::cout << add(5, 3) << '\n';
```

```
11
12      return 0;
13  }
```

当 `x + y` 被计算，其结果被存储于一个匿名对象中。匿名对象的拷贝接着将值返回给调用者，接着匿名对象被销毁。

这不仅仅作用于返回值，同样也作用于函数参数。

```
1  #include <iostream>
2
3  void printValue(int value)
4  {
5      std::cout << value;
6  }
7
8  int main()
9  {
10     int sum{ 5 + 3 };
11     printValue(sum);
12
13     return 0;
14 }
```

可以写成这样：

```
1  #include <iostream>
2
3  void printValue(int value)
4  {
5      std::cout << value;
6  }
7
8  int main()
9  {
10     printValue(5 + 3);
11
12     return 0;
13 }
```

匿名类对象

尽管之前的例子使用的都是内建的数据类型，但是为自定义的类构建匿名对象也是可行的。与普通创建对象类似，但是省略变量名。

```
1  Cents cents{ 5 }; // 普通变量
2  Cents{ 7 };       // 匿名对象
```

现在来看一个更好的例子：

```cpp
#include <iostream>

class Cents
{
private:
    int m_cents{};

public:
    Cents(int cents)
        : m_cents { cents }
    {}

    int getCents() const { return m_cents; }
};

void print(const Cents& cents)
{
   std::cout << cents.getCents() << " cents\n";
}

int main()
{
    Cents cents{ 6 };
    print(cents);

    return 0;
}
```

可以使用匿名对象简化上述代码:

```cpp
#include <iostream>

class Cents
{
private:
    int m_cents{};

public:
    Cents(int cents)
        : m_cents { cents }
    {}

    int getCents() const { return m_cents; }
};

void print(const Cents& cents)
{
   std::cout << cents.getCents() << " cents\n";
}

```

```
int main()
{
    print(Cents{ 6 }); // 注意：传递一个匿名 Cents 值

    return 0;
}
```

现在来看一个更复杂的例子：

```
#include <iostream>

class Cents
{
private:
    int m_cents{};

public:
    Cents(int cents)
        : m_cents { cents }
    {}

    int getCents() const { return m_cents; }
};

Cents add(const Cents& c1, const Cents& c2)
{
    Cents sum{ c1.getCents() + c2.getCents() };
    return sum;
}

int main()
{
    Cents cents1{ 6 };
    Cents cents2{ 8 };
    Cents sum{ add(cents1, cents2) };
    std::cout << "I have " << sum.getCents() << " cents.\n";

    return 0;
}
```

使用匿名值：

```
#include <iostream>

class Cents
{
private:
    int m_cents{};

public:
```

```
    Cents(int cents)
        : m_cents { cents }
    {}

    int getCents() const { return m_cents; }
};

Cents add(const Cents& c1, const Cents& c2)
{
    // 列表初始化查看函数的返回类型并根据它们创建正确的对象
    return { c1.getCents() + c2.getCents() }; // 返回匿名 Cents 值
}

int main()
{
    Cents cents1{ 6 };
    Cents cents2{ 8 };
    std::cout << "I have " << add(cents1, cents2).getCents() << " cents.\n";

    return 0;
}
```

实际上，因为 `cents1` 与 `cents2` 只被用在一处，可以更进一步的匿名：

```
#include <iostream>

class Cents
{
private:
    int m_cents{};

public:
    Cents(int cents)
        : m_cents { cents }
    {}

    int getCents() const { return m_cents; }
};

Cents add(const Cents& c1, const Cents& c2)
{
    return { c1.getCents() + c2.getCents() }; // 返回 Cents 值
}

int main()
{
    std::cout << "I have " << add(Cents{ 6 }, Cents{ 8 }).getCents() << " cents.\n"; // 打印
    Cents 值

    return 0;
```

```
}
```

总结

C++ 中，匿名对象主要用于传递或者返回值，避免了创建大量的临时变量。动态的内存分配也是如此匿名（这也是为什么它们的地址必须分配给指针，否则没有办法执行它们）。
同样值得注意的是因为匿名对象拥有表达式作用域，它们仅可以被使用一次（除非被绑定到左值引用，即延长临时对象的声明周期来匹配引用的生命周期）。如果需要多次引用一个表达式，那么则需要命名变量。

13.17 Nested types in classes

考虑以下代码：

```
#include <iostream>

enum class FruitType
{
  apple,
  banana,
  cherry
};

class Fruit
{
private:
  FruitType m_type {};
  int m_percentageEaten { 0 };

public:
  Fruit(FruitType type) :
    m_type { type }
  {
  }

  FruitType getType() const { return m_type; }
  int getPercentageEaten() const { return m_percentageEaten; }
};

int main()
{
  Fruit apple { FruitType::apple };

  if (apple.getType() == FruitType::apple)
    std::cout << "I am an apple";
  else
```

```
    std::cout << "I am not an apple";

  return 0;
}
```

代码没有错。但是因为枚举 FruitType 用于连接 Fruit 类，独立存在于类本身其实是有点怪异的。

嵌套类型

类似于函数和数据可以作为类的成员，C++ 中，类型也同样可以在类中被定义。只需要在类中定义类型并使用 public 访问说明符即可。

```
#include <iostream>

class Fruit
{
public:
  // 注意：移动 FruitType 进类中并位于 public 访问说明符之下
  // 同样也从 enum class 转换为 enum
  enum FruitType
  {
    apple,
    banana,
    cherry
  };

private:
  FruitType m_type {};
  int m_percentageEaten { 0 };

public:
  Fruit(FruitType type) :
    m_type { type }
  {
  }

  FruitType getType() const { return m_type; }
  int getPercentageEaten() const { return m_percentageEaten; }
};

int main()
{
  // 注意：可以通过 Fruit 来访问 FruitType 了
  Fruit apple { Fruit::apple };

  if (apple.getType() == Fruit::apple)
    std::cout << "I am an apple";
```

```
36   else
37     std::cout << "I am not an apple";
38
39   return 0;
40 }
```

14 Operator overloading

14.1 Introduction to operator overloading

8.9 函数重载中学到了函数的重载提供了一种机制来创建和解析函数调用同名的不同函数，其中只需要每个函数都有一个独特的函数原型。这允许用户创建函数的变体用于应对不同的数据类型，而无需为每一种变体进行独特的命名。
C++ 中操作符的实现如同函数。通过函数重载的方式来重载操作符函数，可以自定义操作符用作于不同的数据类型（包括用户自己的类）。使用函数重载来重载操作符被称为**操作符重载**。

操作符作为函数

思考以下例子：

```
1 int x { 2 };
2 int y { 3 };
3 std::cout << x + y << '\n';
```

编译器使用了内建版本的操作符（+）来进行整型运算 – 该函数使整数 x 与 y 相加并返回一个整数结果。当看见表达式 `x + y` 时，可以在脑海中转换成函数调用 `operator+(x, y)`（其中 operator+ 是函数的名称）。

解析重载操作符

当解析一个包含操作符的表达式时，编译器使用以下规则：

- 如果*所有的*运算是基础的数据类型，编译器会调用内建的操作符函数，如果该函数不存在，编译器则产生编译错误。
- 如果*任意的*运算时用户数据类型（例如，用户的类，或者枚举类），编译器将查看类型是否有匹配的重载操作符函数可以被调用。如果不能找到，则尝试转换一个或多个用户定义类型的运算成基础数据类型，以便使用匹配的内建操作符（通过重载类型强转，下一节讲详细讲述）。如果还是失败了，则产生变异错误。

操作符重载的限制是什么?

首先，C++ 中几乎任何存在的操作符都可以被重载。除了：条件判断符（?:），sizeof，作用于（::），成员选择符（.），指针成员选择符（.*），typeid，以及强转操作符。
其次，仅可以重载已存在的操作符。不可以创建新的符号或者对已有操作符重命名。例如不可以创建一个符号 ** 来做指数计算。

第三，重载操作符中的运算至少有一个必须是用户定义类型。这就意味着不可以重载加法操作符来进行 int 与 double 相加。然而可以重载加法操作符来进行 int 与 Mystring 的相加。
第四，不可以修改操作符支持的运算数量。
最后，所有的操作符维持其默认的优先级与结合律（无论它们做什么），这是不可变的。
一些新手程序员尝试重载位运算符 XOR 操作符（^）来做指数计算。然而 C++ 中操作符 ^ 相较于基础计算操作符，拥有一个更低级的优先级，这就导致运算的解析错误。
因为有优先级的问题，最好的办法就是在操作符原有的意图上进行重载。
最佳实践：当重载操作符时，最好是维持操作符原有的意图。
此外，因为操作符并没有描述性的名称，其意图往往不是很清楚。例如，操作符 + 对于字符串类型的拼接是合理的，但是操作符- 呢？它的意图是不清晰的。
最佳实践：如果重载的操作符其意义是不清晰且符合直觉的，那么应该使用命名函数。

14.2 Overloading the arithmetic operators using friend functions

C++ 中用到最多的操作符之一就有运算操作符 – 即加法操作符（+），减法操作符（-），乘法操作符（*）与除法操作符（/）。注意所有的运算操作符都是二元操作符 – 即需要两个运算对象 – 位于操作符的两侧。这四个操作符都以相同的方式的重载。
事实上有三种不同的方法用于重载操作符：以成员函数的方式，以友元函数的方式，以及普通函数的方式。本节将详细讲述友元函数的方式（因为它更符合直觉）。下一节讲述普通函数的方式，接着再讲解成员函数的方式。

使用友元函数重载操作符

考虑以下代码：

```
class Cents
{
private:
  int m_cents {};

public:
  Cents(int cents) : m_cents{ cents } { }
  int getCents() const { return m_cents; }
};
```

下面的例子展示了如何重载操作符加号（+）来使两个“Cents”对象相加：

```
#include <iostream>

class Cents
{
private:
  int m_cents {};
```

```
7
8  public:
9    Cents(int cents) : m_cents{ cents } { }
10
11   // 使用友元函数进行 Cents + Cents 的相加
12   friend Cents operator+(const Cents& c1, const Cents& c2);
13
14   int getCents() const { return m_cents; }
15 };
16
17 // 注意: 该函数不是一个成员函数!
18 Cents operator+(const Cents& c1, const Cents& c2)
19 {
20   // 使用 Cents 的构造函数与操作符+(int, int)
21   // 可以直接访问 m_cents 因为其为友元
22   return Cents{c1.m_cents + c2.m_cents};
23 }
24
25 int main()
26 {
27   Cents cents1{ 6 };
28   Cents cents2{ 8 };
29   Cents centsSum{ cents1 + cents2 };
30   std::cout << "I have " << centsSum.getCents() << " cents.\n";
31
32   return 0;
33 }
```

打印:

```
1 I have 14 cents.
```

重载减法操作符（-）同样也很简单:

```
1  #include <iostream>
2
3  class Cents
4  {
5  private:
6    int m_cents {};
7
8  public:
9    Cents(int cents) : m_cents{ cents } { }
10
11   // 使用友元函数进行 Cents + Cents 的相加
12   friend Cents operator+(const Cents& c1, const Cents& c2);
13
14   // 使用友元函数进行 Cents - Cents 的相减
15   friend Cents operator-(const Cents& c1, const Cents& c2);
16
```

```
  int getCents() const { return m_cents; }
};

// 注意：该函数不是一个成员函数！
Cents operator+(const Cents& c1, const Cents& c2)
{
  // 使用 Cents 的构造函数与操作符+(int, int)
  // 可以直接访问 m_cents 因为其为友元
  return Cents{c1.m_cents + c2.m_cents};
}

// 注意：该函数不是一个成员函数！
Cents operator-(const Cents& c1, const Cents& c2)
{
  // 使用 Cents 的构造函数与操作符-(int, int)
  // 可以直接访问 m_cents 因为其为友元
  return Cents(c1.m_cents - c2.m_cents);
}

int main()
{
  Cents cents1{ 6 };
  Cents cents2{ 2 };
  Cents centsSum{ cents1 - cents2 };
  std::cout << "I have " << centsSum.getCents() << " cents.\n";

  return 0;
}
```

重载乘法操作符（*）与除法操作符（/）同样也很简单。

友元函数可以定义在类的内部

尽管友元函数不是类的成员函数，它们仍然可以被定义在类的内部：

```
#include <iostream>

class Cents
{
private:
  int m_cents {};

public:
  Cents(int cents) : m_cents{ cents } { }

  friend Cents operator+(const Cents& c1, const Cents& c2)
  {
    return Cents{c1.m_cents + c2.m_cents};
  }
```

```
15
16    int getCents() const { return m_cents; }
17 };
18
19 int main()
20 {
21    Cents cents1{ 6 };
22    Cents cents2{ 8 };
23    Cents centsSum{ cents1 + cents2 };
24    std::cout << "I have " << centsSum.getCents() << " cents.\n";
25
26    return 0;
27 }
```

通常来说并不推荐这么做，因为重要的函数定义放在分开的.cpp 文件中更好。而之后的教程中这么使用是为了代码的连续性。

为不同类型的运算对象重载操作符

通常情况下，会有不同类型的运算对象的需要。例如 Cents(4) 与整数 6 相加返回 Cents(10)。当 C++ 解析 `x + y` 表达式时，`x` 变为第一个参数，`y` 变为第二个参数。当 `x` 与 `y` 的类型相同，那么 `x + y` 还是 `y + x` 并没有区别，相同类型的操作符 + 会被调用。然而当运算对象的类型不同时，`x + y` 与 `y + x` 并不相同。

例如，`Cents(4) + 6` 将调用操作符 +(Cents, int)，而 `6 + Cents(4)` 将调用操作符 +(int, Cents)。因此为不同类型的二元进行重载操作符，实际上是需要编写两个函数 – 以对应两种情况。

```
1  #include <iostream>
2
3  class Cents
4  {
5  private:
6    int m_cents {};
7
8  public:
9    Cents(int cents) : m_cents{ cents } { }
10
11   // 使用友元函数进行 Cents + int 的相加
12   friend Cents operator+(const Cents& c1, int value);
13
14   // 使用友元函数进行 int + Cents 的相加
15   friend Cents operator+(int value, const Cents& c1);
16
17
18   int getCents() const { return m_cents; }
19 };
```

```
20
21 // 注意：该函数不是一个成员函数！
22 Cents operator+(const Cents& c1, int value)
23 {
24   // 使用 Cents 的构造函数与操作符+(Cents&, int)
25   // 可以直接访问 m_cents 因为其为友元
26   return { c1.m_cents + value };
27 }
28
29 // 注意：该函数不是一个成员函数！
30 Cents operator+(int value, const Cents& c1)
31 {
32   // 使用 Cents 的构造函数与操作符+(int ,Cents&)
33   // 可以直接访问 m_cents 因为其为友元
34   return { c1.m_cents + value };
35 }
36
37 int main()
38 {
39   Cents c1{ Cents{ 4 } + 6 };
40   Cents c2{ 6 + Cents{ 4 } };
41
42   std::cout << "I have " << c1.getCents() << " cents.\n";
43   std::cout << "I have " << c2.getCents() << " cents.\n";
44
45   return 0;
46 }
```

注意两个重载函数有着相同的实现 – 这是因为它们做的事情是一样的，不同的只是参数的顺序。

另一个例子

接下来看另一个例子：

```
1  #include <iostream>
2
3  class MinMax
4  {
5  private:
6    int m_min {}; // 已经见到的 min 值
7    int m_max {}; // 已经见到的 max 值
8
9  public:
10   MinMax(int min, int max)
11     : m_min { min }, m_max { max }
12   { }
13
14   int getMin() const { return m_min; }
15   int getMax() const { return m_max; }
```

```
16
17    friend MinMax operator+(const MinMax& m1, const MinMax& m2);
18    friend MinMax operator+(const MinMax& m, int value);
19    friend MinMax operator+(int value, const MinMax& m);
20  };
21
22  MinMax operator+(const MinMax& m1, const MinMax& m2)
23  {
24    // 获取 m1 与 m2 中的最小值
25    int min{ m1.m_min < m2.m_min ? m1.m_min : m2.m_min };
26
27    // 获取 m1 与 m2 中的最大值
28    int max{ m1.m_max > m2.m_max ? m1.m_max : m2.m_max };
29
30    return { min, max };
31  }
32
33  MinMax operator+(const MinMax& m, int value)
34  {
35    // 获取 m 与 value 中的最小值
36    int min{ m.m_min < value ? m.m_min : value };
37
38    // 获取 m 与 value 中的最大值
39    int max{ m.m_max > value ? m.m_max : value };
40
41    return { min, max };
42  }
43
44  MinMax operator+(int value, const MinMax& m)
45  {
46    // 调用 operator+(MinMax, int)
47    return { m + value };
48  }
49
50  int main()
51  {
52    MinMax m1{ 10, 15 };
53    MinMax m2{ 8, 11 };
54    MinMax m3{ 3, 12 };
55
56    MinMax mFinal{ m1 + m2 + 5 + 8 + m3 + 16 };
57
58    std::cout << "Result: (" << mFinal.getMin() << ", " <<
59      mFinal.getMax() << ")\n";
60
61    return 0;
62  }
```

使用其他操作符来实现操作符

上述例子中，注意定义的操作符 +(int, MinMax) 中调用了操作符 +(MinMax, int)，即得出相同结果。这使得操作符 +(int, MinMax) 的实现变成了一行代码，大大的减少了重复的代码，也同样使得函数更容易理解。
通常可以通过调用其他重载操作符来定义重载操作符。当这样做可以简化代码时，这样的做法则是提倡的。

14.3 Overloading operators using normal functions

上一节中讲到了使用友元函数来重载操作符。这很方便因为可以给与直接访问需要操作的类内部成员的权利。然而当不需要访问时，可以使用普通函数来编写重载函数。

```
#include <iostream>

class Cents
{
private:
  int m_cents{};

public:
  Cents(int cents)
    : m_cents{ cents }
  {}

  int getCents() const { return m_cents; }
};

// 注意：该函数不是成员函数也不是友元函数！
Cents operator+(const Cents& c1, const Cents& c2)
{
  // 使用 Cents 构造函数与操作符+(int, int)
  // 这里并不需要直接访问私有成员
  return Cents{ c1.getCents() + c2.getCents() };
}

int main()
{
  Cents cents1{ 6 };
  Cents cents2{ 8 };
  Cents centsSum{ cents1 + cents2 };
  std::cout << "I have " << centsSum.getCents() << " cents.\n";

  return 0;
}
```

由于普通函数和友元函数效果完全一样（仅在访问私有成员时拥有不同等级），通常来说不会区分它们。一个区别在于友元函数声明在类内部也可以作为原型；而普通函数的版本，用户需要提供自己的函数原型。

Cents.h:

```
#ifndef CENTS_H
#define CENTS_H

class Cents
{
private:
  int m_cents{};

public:
  Cents(int cents)
    : m_cents{ cents }
  {}

  int getCents() const { return m_cents; }
};

// 需要显式提供操作符+ 的原型，这样使得其他文件夹知道该重载的存在
Cents operator+(const Cents& c1, const Cents& c2);

# endif
```

Cents.cpp:

```
#include "Cents.h"

// 注意：该函数不是成员函数也不是友元函数！
Cents operator+(const Cents& c1, const Cents& c2)
{
  // 使用 Cents 构造函数与操作符+(int, int)
  // 这里并不需要直接访问私有成员
  return { c1.getCents() + c2.getCents() };
}
```

main.cpp:

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

int main()
{
  Cents cents1{ 6 };
  Cents cents2{ 8 };
  Cents centsSum{ cents1 + cents2 }; // 如果在 Cents.h 中没有原型，这里会编译错误
  std::cout << "I have " << centsSum.getCents() << " cents.\n";

```

```
  return 0;
}
```

通常而言，如果存在成员函数可用的情况，普通函数比友元函数更为推荐（越少的函数解除类的内部越好）。然而，不要仅仅为了使用普通函数来重载操作符而去添加额外的访问函数，直接使用友元函数！
最佳实践：在不需要添加额外的访问函数的情况下，推荐使用普通函数而不是友元函数来重载操作符。

14.4 Overloading the IO operators

对于拥有若干成员变量的类而言，打印每个独立的变量在屏幕上令人厌倦。例如考虑以下类：

```
class Point
{
private:
    double m_x{};
    double m_y{};
    double m_z{};

public:
    Point(double x=0.0, double y=0.0, double z=0.0)
      : m_x{x}, m_y{y}, m_z{z}
    {
    }

    double getX() const { return m_x; }
    double getY() const { return m_y; }
    double getZ() const { return m_z; }
};
```

如果希望打印这个类的实力在屏幕上，需要如下：

```
Point point{5.0, 6.0, 7.0};

std::cout << "Point(" << point.getX() << ", " <<
    point.getY() << ", " <<
    point.getZ() << ')';
```

当然使用一个可复用的函数更有意义：

```
class Point
{
private:
    double m_x{};
    double m_y{};
    double m_z{};

public:
```

```
    Point(double x=0.0, double y=0.0, double z=0.0)
      : m_x{x}, m_y{y}, m_z{z}
    {
    }

    double getX() const { return m_x; }
    double getY() const { return m_y; }
    double getZ() const { return m_z; }

    void print() const
    {
        std::cout << "Point(" << m_x << ", " << m_y << ", " << m_z << ')';
    }
};
```

这样更好一点，但是仍然有缺陷。因为 `print()` 返回 `void`，它不能在输出声明中调用，因此需要这么做：

```
int main()
{
    const Point point{5.0, 6.0, 7.0};

    std::cout << "My point is: ";
    point.print();
    std::cout << " in Cartesian space.\n";
}
```

那如果是以下这样会更简单：

```
Point point{5.0, 6.0, 7.0};
cout << "My point is: " << point << " in Cartesian space.\n";
```

返回同样的结果，且不会破坏若干声明的输出，也不需要记住打印函数。
幸运的是，重载 « 操作符可以达到这样的效果！

重载操作符 «

重载操作符 « 类似于重载操作符 +（它们皆为二元操作符），只不过参数类型不同。
考虑这样一个表达式 `std::cout << point`。如果操作符为 «，那么运算对象呢？左侧的运算对象是 std::cout 对象，而右侧的运算对象则是 Point 类对象。std::cout 实际上是一个类型为 std::ostream 的对象。因此重载函数应该像这样：

```
// std::ostream is the type for object std::cout
friend std::ostream& operator<< (std::ostream& out, const Point& point);
```

为 Point 类实现操作符 « 非常的直接 – 因为 C++ 已经知道了如何使用操作符 « 输出 doubles，且所有成员也都是 doubles，那么可以简单的使用操作符 « 来输出 Point 的成员变量。

```cpp
#include <iostream>

class Point
{
private:
    double m_x{};
    double m_y{};
    double m_z{};

public:
    Point(double x=0.0, double y=0.0, double z=0.0)
      : m_x{x}, m_y{y}, m_z{z}
    {
    }

    friend std::ostream& operator<< (std::ostream& out, const Point& point);
};

std::ostream& operator<< (std::ostream& out, const Point& point)
{
    // 因为 operator<<  是 Point 类的友元，可以直接访问 Point 的成员。
    out << "Point(" << point.m_x << ", " << point.m_y << ", " << point.m_z << ')'; // 真实输出在这里完成

    return out; // 返回 std::ostream 使得可以串联调用 operator<<
}

int main()
{
    const Point point1{2.0, 3.0, 4.0};

    std::cout << point1 << '\n';

    return 0;
}
```

打印:

```
Point(2, 3.5, 4) Point(6, 7.5, 8)
```

重载操作符 »

同样的重载输入操作符也是可以的。其实现与输出操作符相似，重点在于了解 `std::cin` 是 `std::istream` 类型的对象。

```cpp
#include <iostream>

class Point
{
```

```
private:
    double m_x{};
    double m_y{};
    double m_z{};

public:
    Point(double x=0.0, double y=0.0, double z=0.0)
      : m_x{x}, m_y{y}, m_z{z}
    {
    }

    friend std::ostream& operator<< (std::ostream& out, const Point& point);
    friend std::istream& operator>> (std::istream& in, Point& point);
};

std::ostream& operator<< (std::ostream& out, const Point& point)
{
    // Since operator<< is a friend of the Point class, we can access Point's members directly.
    out << "Point(" << point.m_x << ", " << point.m_y << ", " << point.m_z << ')';

    return out;
}

std::istream& operator>> (std::istream& in, Point& point)
{
    // 因为 operator<< 是 Point 类的友元，可以直接访问 Point 的成员。
    // 注意参数 point 必须是非 const 的，因此可以使用输入值修改其类成员
    in >> point.m_x;
    in >> point.m_y;
    in >> point.m_z;

    return in;
}

int main()
{
    std::cout << "Enter a point: ";

    Point point;
    std::cin >> point;

    std::cout << "You entered: " << point << '\n';

    return 0;
}
```

假设用户输入了 3.0 4.5 7.26 ，则打印：

```
You entered: Point(3, 4.5, 7.26)
```

14.5 Overloading operators using member functions

在 14.2 中学习了使用友元函数重载算法操作符，同样也学习了使用普通函数重载操作符。很多操作符可以有另一种方式被重载：成员函数。
使用成员函数重载操作符类似于使用友元函数的方式，但是有几点要求：

- 重载操作符必须是左侧运算对象的成员函数
- 左侧运算对象变成隐式的 *this 对象
- 其它的运算对象变为函数入参

转换一个友元重载操作符成为成员重载操作符也很简单：

1. 重载操作符定义为成员函数而不是友元
2. 左侧参数被移除，因为该参数现在变为了隐式 *this 对象
3. 在函数体内，所有左侧参数的引用可以被移除

```
#include <iostream>

class Cents
{
private:
    int m_cents {};

public:
    Cents(int cents)
        : m_cents { cents } { }

    // Overload Cents + int
    Cents operator+ (int value);

    int getCents() const { return m_cents; }
};

// 注意：该函数是一个成员函数！
// 友元版本中的 cents 参数现在是隐式 *this 参数
Cents Cents::operator+ (int value)
{
    return Cents { m_cents + value };
}

int main()
{
    Cents cents1 { 6 };
```

```
28      Cents cents2 { cents1 + 2 };
29      std::cout << "I have " << cents2.getCents() << " cents.\n";
30
31      return 0;
32  }
```

不是所有东西都能以友元函数的方式重载

赋值（=），下标（[]），函数调用（()），以及成员选择（->）操作符必须以成员函数重载。

不是所有东西都能以成员函数的方式重载

14.4 重载 I/O 操作符中学习了重载操作符 «，它是不能作为成员函数的。因为重载的运算成员必须是左侧的成员。

使用普通、友元、或成员函数进行重载的时机

大多数情况下，C++ 允许用户自己决定到底是使用普通、友元、或成员函数来进行重载。然而总有一种情况是优于另外两个的。
当处理二元操作符且不需要修改左侧运算成员（例如操作符 +），普通函数或友元函数更为推荐，因为它可以用于所有的参数类型（即使左侧运算成员不是一个类对象，或者是一个不可变的类）。普通或友元函数可以从“对称性”获益，因为索引运算成员是显式参数（而不是左侧运算成员是 *this）。
当处理二元操作符且需要修改左侧运算成员时（例如操作符 +=），成员函数通常更为推荐。这些情况下，左侧运算成员通常是一个类类型，且拥被修改的对象通过 *this 指向是自然而然的。因为右侧运算成员变成了显式参数，被修改与被计算的是显而易见的。
一元运算通常也是作为成员函数进行重载，因为它们不需要参数。

14.6 Overloading unary operators +, -, and !

重载一元操作符

不同于迄今为止已经见到过得操作符，正号（+）、负号（-）以及逻辑非（!）操作符都是一元操作符，这就意味着它们仅可以对一个运算成员进行计算。因为它们运算的对象就是应用的对象，通常一元操作符重载的实现都是以成员函数的方式。这三种运算的实现行为一致。
现在看一下上一节所用的例子来进行操作符- 重载：

```
1  #include <iostream>
2
3  class Cents
4  {
```

```
private:
    int m_cents {};

public:
    Cents(int cents): m_cents{cents} {}

    // 以成员函数的方式重载 -Cents
    Cents operator-() const;

    int getCents() const { return m_cents; }
};

// 注意: 该函数为成员函数!
Cents Cents::operator-() const
{
    return -m_cents; // 由于返回类型是一个 Cents, 这里使用了 Cents(int) 构造函数, 进行了从 int 到 Cents 的一个隐式转换
}

int main()
{
    const Cents nickle{ 5 };
    std::cout << "A nickle of debt is worth " << (-nickle).getCents() << " cents\n";

    return 0;
}
```

这应该是很直接的。重载的负号操作符（-）是一个以成员函数方式实现的一元操作符，因此其不需要参数（它运算 *this 对象）。该函数返回一个与原始对象一样的对象，其值带有负号。因为操作符- 不修改 Cents 对象，我们可以（应该）使其成为一个 const 函数（有此可在 const Cents 对象上进行调用）。

注意这里的负号操作符- 与减法操作符- 应该是没有歧义的 – 因为它们的参数不同。

以下是另一个案例。操作符! 是逻辑非操作符 – 如果一个表达式运算类似于“true”，操作符! 将返回 false，反之亦然。我们通常会在布尔变量上看到这样的操作：

```
if (!isHappy)
  std::cout << "I am not happy!\n";
else
  std::cout << "I am so happy!\n";
```

对于整数，0 解析为 false，其它为 true，因此操作符! 应用在除零以外的整数上都为 true，零为 false。

延展这个概念，在“false“, ”zero“或者其他默认初始状态下，使用操作符! 解析为 true。

以下案例展示了重载了操作符- 以及操作符! 的自定义 Point 类：

```
#include <iostream>

```

```
class Point
{
private:
    double m_x {};
    double m_y {};
    double m_z {};

public:
    Point(double x=0.0, double y=0.0, double z=0.0):
        m_x{x}, m_y{y}, m_z{z}
    {
    }

    // 转换 Point 为与之相等的负值
    Point operator- () const;

    // 如果 point 是原点返回 true
    bool operator! () const;

    double getX() const { return m_x; }
    double getY() const { return m_y; }
    double getZ() const { return m_z; }
};

Point Point::operator- () const
{
    return { -m_x, -m_y, -m_z };
}

bool Point::operator! () const
{
    return (m_x == 0.0 && m_y == 0.0 && m_z == 0.0);
}

int main()
{
    Point point{}; // use default constructor to set to (0.0, 0.0, 0.0)

    if (!point)
        std::cout << "point is set at the origin.\n";
    else
        std::cout << "point is not set at the origin.\n";

    return 0;
}
```

打印:

```
point is set at the origin.
```

14.7 Overloading the comparison operators

在 5.6 的关系符以及浮点数比较中讨论了六种比较操作符。重载这些比较操作符相对来说比较简单，因为它们遵循相同的模式。

因为比较操作符同样都是二元操作符，并且不会修改左侧运算成员，因此我们将以友元函数的方式进行重载。

这里是一个重载了操作符 == 与操作符!= 的 Car 类案例：

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

class Car
{
private:
    std::string m_make;
    std::string m_model;

public:
    Car(std::string_view make, std::string_view model)
        : m_make{ make }, m_model{ model }
    {
    }

    friend bool operator== (const Car& c1, const Car& c2);
    friend bool operator!= (const Car& c1, const Car& c2);
};

bool operator== (const Car& c1, const Car& c2)
{
    return (c1.m_make == c2.m_make &&
            c1.m_model == c2.m_model);
}

bool operator!= (const Car& c1, const Car& c2)
{
    return (c1.m_make != c2.m_make ||
            c1.m_model != c2.m_model);
}

int main()
{
    Car corolla{ "Toyota", "Corolla" };
    Car camry{ "Toyota", "Camry" };

    if (corolla == camry)
        std::cout << "a Corolla and Camry are the same.\n";

```

```
    if (corolla != camry)
        std::cout << "a Corolla and Camry are not the same.\n";

    return 0;
}
```

上述代码非常的直观。

那么操作符 < 与操作符 > 呢？它们的直观意义并不明确，因此不去定义这些操作符更好。

最佳实践：仅去定义符合直觉的重载操作符。

然而有一个特例。如果希望对 Cars 列表进行排序呢？这种情况下，我们希望能重载比较操作符用于排序。例如对 Cars 重载操作符 < 意为根据字母顺序进行排序。

一些标准库中的容器类需要重载操作符 < 这样它们可以维护元素的排序。

这里是另一个重载了 6 个所有比较操作符的例子：

```
#include <iostream>

class Cents
{
private:
    int m_cents;

public:
    Cents(int cents)
  : m_cents{ cents }
  {}

    friend bool operator== (const Cents& c1, const Cents& c2);
    friend bool operator!= (const Cents& c1, const Cents& c2);

    friend bool operator< (const Cents& c1, const Cents& c2);
    friend bool operator> (const Cents& c1, const Cents& c2);

    friend bool operator<= (const Cents& c1, const Cents& c2);
    friend bool operator>= (const Cents& c1, const Cents& c2);
};

bool operator== (const Cents& c1, const Cents& c2)
{
    return c1.m_cents == c2.m_cents;
}

bool operator!= (const Cents& c1, const Cents& c2)
{
    return c1.m_cents != c2.m_cents;
}

bool operator< (const Cents& c1, const Cents& c2)
```

```
{
    return c1.m_cents < c2.m_cents;
}

bool operator> (const Cents& c1, const Cents& c2)
{
    return c1.m_cents > c2.m_cents;
}

bool operator<= (const Cents& c1, const Cents& c2)
{
    return c1.m_cents <= c2.m_cents;
}

bool operator>= (const Cents& c1, const Cents& c2)
{
    return c1.m_cents >= c2.m_cents;
}

int main()
{
    Cents dime{ 10 };
    Cents nickel{ 5 };

    if (nickel > dime)
        std::cout << "a nickel is greater than a dime.\n";
    if (nickel >= dime)
        std::cout << "a nickel is greater than or equal to a dime.\n";
    if (nickel < dime)
        std::cout << "a dime is greater than a nickel.\n";
    if (nickel <= dime)
        std::cout << "a dime is greater than or equal to a nickel.\n";
    if (nickel == dime)
        std::cout << "a dime is equal to a nickel.\n";
    if (nickel != dime)
        std::cout << "a dime is not equal to a nickel.\n";

    return 0;
}
```

最小化比较的冗余

上述例子可以注意到每个重载比较操作符有多么的相似。重载比较操作符倾向有高度的冗余，并且随着实现的复杂性增加，冗余随之增加。
幸运的是，很多比较操作符可以使用其他比较操作符来实现：

- 操作符!= 可以通过!(操作符 ==) 来实现

- 操作符 > 可以通过与之相反的 < 来实现
- 操作符 >= 可以通过!(操作符 <) 来实现
- 操作符 <= 可以通过!(操作符 >) 来实现

这就意味着我们仅需要实现操作符 == 以及操作符 <，那么其它的四个比较操作符可以根据这两个操作符来被定义！以下是更新后的 Cents 案例：

```
#include <iostream>

class Cents
{
private:
    int m_cents;

public:
    Cents(int cents)
        : m_cents{ cents }
    {}

    friend bool operator== (const Cents& c1, const Cents& c2);
    friend bool operator!= (const Cents& c1, const Cents& c2);

    friend bool operator< (const Cents& c1, const Cents& c2);
    friend bool operator> (const Cents& c1, const Cents& c2);

    friend bool operator<= (const Cents& c1, const Cents& c2);
    friend bool operator>= (const Cents& c1, const Cents& c2);

};

bool operator== (const Cents& c1, const Cents& c2)
{
    return c1.m_cents == c2.m_cents;
}

bool operator!= (const Cents& c1, const Cents& c2)
{
    return !(operator==(c1, c2));
}

bool operator< (const Cents& c1, const Cents& c2)
{
    return c1.m_cents < c2.m_cents;
}

bool operator> (const Cents& c1, const Cents& c2)
{
```

```
    return operator<(c2, c1);
}

bool operator<= (const Cents& c1, const Cents& c2)
{
    return !(operator>(c1, c2));
}

bool operator>= (const Cents& c1, const Cents& c2)
{
    return !(operator<(c1, c2));
}

int main()
{
    Cents dime{ 10 };
    Cents nickel{ 5 };

    if (nickel > dime)
        std::cout << "a nickel is greater than a dime.\n";
    if (nickel >= dime)
        std::cout << "a nickel is greater than or equal to a dime.\n";
    if (nickel < dime)
        std::cout << "a dime is greater than a nickel.\n";
    if (nickel <= dime)
        std::cout << "a dime is greater than or equal to a nickel.\n";
    if (nickel == dime)
        std::cout << "a dime is equal to a nickel.\n";
    if (nickel != dime)
        std::cout << "a dime is not equal to a nickel.\n";

    return 0;
}
```

如此便在需要修改的时候，仅需要更新操作符 == 以及操作符 < 而不是所有的六种比较操作符！

14.8 Overloading the increment and decrement operators

重载增加（++）与减少（--）操作符非常的直观，仅有一个小例外。它们实际上两个版本的增加与减少操作符：前置增加以及减少（例如 `++x; --y;`）以及后置的增加与减少（例如 `x++; y--;`）。

由于增加与减少操作符都是一元操作符，它们要修改操作成员，因此最好是作为成员函数进行重载。这里处理前置的版本，因为它们更直观。

重载前置增加与减少

```
#include <iostream>

class Digit
{
private:
    int m_digit;
public:
    Digit(int digit=0)
        : m_digit{digit}
    {
    }

    Digit& operator++();
    Digit& operator--();

    friend std::ostream& operator<< (std::ostream& out, const Digit& d);
};

Digit& Digit::operator++()
{
    // If our number is already at 9, wrap around to 0
    if (m_digit == 9)
        m_digit = 0;
    // otherwise just increment to next number
    else
        ++m_digit;

    return *this;
}

Digit& Digit::operator--()
{
    // If our number is already at 0, wrap around to 9
    if (m_digit == 0)
        m_digit = 9;
    // otherwise just decrement to next number
    else
        --m_digit;

    return *this;
}

std::ostream& operator<< (std::ostream& out, const Digit& d)
{
  out << d.m_digit;
  return out;
}
```

```
48
49 int main()
50 {
51     Digit digit(8);
52
53     std::cout << digit;
54     std::cout << ++digit;
55     std::cout << ++digit;
56     std::cout << --digit;
57     std::cout << --digit;
58
59     return 0;
60 }
```

打印:

```
1 89098
```

注意返回了 *this。重载的增加与减少操作符返回当前的隐式对象，这样若干操作符可以被”串联“起来。

重载后置增加与减少

通常而言，重名的函数可以通过不同的参数数量以及/或不同的参数类型来进行重载。然而考虑前置与后置的增加与减少操作符，它们都有相同的名字，并且都是接受同样类型的一个参数。那么如何区分它们的重载呢?

C++ 语言规范提供了一个特殊的方案：编译器检查重载操作符是否有一个 int 参数。如果有，操作符是后置重载；如果重载的操作符没有参数，操作符则是前置重载。

```
1  class Digit
2  {
3  private:
4      int m_digit;
5  public:
6      Digit(int digit=0)
7          : m_digit{digit}
8      {
9      }
10
11     Digit& operator++(); // 前置不带参数
12     Digit& operator--(); // 前置不带参数
13
14     Digit operator++(int); // 后置带有 int 参数
15     Digit operator--(int); // 后置带有 int 参数
16
17     friend std::ostream& operator<< (std::ostream& out, const Digit& d);
18 };
19
```

```
20 // 没有参数意味着前置 operator++
21 Digit& Digit::operator++()
22 {
23     // 如果处于 9，处理成 0
24     if (m_digit == 9)
25         m_digit = 0;
26     // 否则增加一个数
27     else
28         ++m_digit;
29 
30     return *this;
31 }
32 
33 // 没有参数意味着前置 operator--
34 Digit& Digit::operator--()
35 {
36     // 如果处于 0，处理成 9
37     if (m_digit == 0)
38         m_digit = 9;
39     // 否则减少一个数
40     else
41         --m_digit;
42 
43     return *this;
44 }
45 
46 // int 参数意味着后置 operator++
47 Digit Digit::operator++(int)
48 {
49     // 通过当前的 digit 创建一个临时的变量
50     Digit temp{*this};
51 
52     // 使用前置操作符来增加 digit
53     ++(*this); // 应用操作符
54 
55     // 返回临时结果
56     return temp; // 返回保存的状态
57 }
58 
59 // int 参数意味着后置 operator--
60 Digit Digit::operator--(int)
61 {
62     // 通过当前的 digit 创建一个临时的变量
63     Digit temp{*this};
64 
65     // 使用前置操作符来减少 digit
66     --(*this); // 应用操作符
67 
68     // 返回临时的结果
```

```
69      return temp; // 返回保存的状态
70  }
71
72  std::ostream& operator<< (std::ostream& out, const Digit& d)
73  {
74    out << d.m_digit;
75    return out;
76  }
77
78  int main()
79  {
80      Digit digit(5);
81
82      std::cout << digit;
83      std::cout << ++digit; // 调用 Digit::operator++();
84      std::cout << digit++; // 调用 Digit::operator++(int);
85      std::cout << digit;
86      std::cout << --digit; // 调用 Digit::operator--();
87      std::cout << digit--; // 调用 Digit::operator--(int);
88      std::cout << digit;
89
90      return 0;
91  }
```

打印:

```
1  5667665
```

这里有几个有趣的点。首先，注意通过整数虚拟参数区分了前置与后置操作符。其次由于虚拟参数并没有在函数实现中使用，因此不需要为其命名。这就是告诉编译器对待该变量为占位符，意味着不会因为声明了变量却不使用而产生警告。

其次，注意前置与后置操作符的功能一致 – 它们都是增加或减少对象。它们俩的区别在于返回的值。前置操作符返回被增加或减少的对象。因此重载它们非常的直接，仅需简单的增加或减少成员变量，接着返回 *this。

后置操作符需要返回的是增加或减少发生前的状态。这就带来了一点迷惑 – 如果要返回修改前的状态，那么增加或减少对象就不可能。也就是说如果在修改对象前返回对象，修改的行为将永远不会被调用。

这个问题的通常解决方案是使用临时变量来存储修改前的状态。那么对象可以被修改，最后临时变量被返回给调用者。这种方式下，调用者获取的是对象在修改之前的拷贝，而对象本身已经被修改了。注意这就意味着返回值必须是非引用的，因为不能返回本地变量的引用，它们会在函数结束时被销毁。同样注意这就意味着后置操作符通常效率会比前置操作符更低一些，因为整个过程中增加了临时变量的实例化，并且值返回而不是引用返回。

最后注意后置操作符都调用了前置操作符的函数。这样减少了重复代码，同样使得未来的修改变得更简单。

14.9 Overloading the subscript operator

在处理数组的时候，通常会使用下标操作符（[]）来索引其指定元素：

```
myArray[0] = 7; // 数组第一个元素设置为 7
```

然而考虑以下 `IntList` 类，其拥有一个数组的成员变量：

```
class IntList
{
private:
    int m_list[10]{};
};

int main()
{
    IntList list{};
    // 如何从 m_list 中访问元素?
    return 0;
}
```

由于 m_list 成员变量是私有的，我们不能直接访问它。这就意味着不能直接 get 或 set m_list 中的数值。那么问题该如何解决呢?
不使用操作符重载的情况下：

```
class IntList
{
private:
    int m_list[10]{};

public:
    void setItem(int index, int value) { m_list[index] = value; }
    int getItem(int index) const { return m_list[index]; }
};
```

可以工作，但是不是特别的用户友好。考虑以下案例：

```
int main()
{
    IntList list{};
    list.setItem(2, 3);

    return 0;
}
```

这里是设置第二个元素为 3 呢，还是第三个元素为 2? 不查看 `setItem()` 定义的情况下，明显是不清晰的。
当然也可以直接返回整个数组并使用操作符 [] 来访问元素

```
class IntList
{
```

```
private:
    int m_list[10]{};

public:
    int* getList() { return m_list; }
};
```

虽然可以工作，但是语法上很奇怪：

```
int main()
{
    IntList list{};
    list.getList()[2] = 3;

    return 0;
}
```

重载操作符 []

然而一个更好的方案是重载操作符 [] 使得可以访问 m_list 的元素。下标操作符必须以成员函数的方式来进行重载。一个重载的操作符 [] 函数总是接受一个参数：用户提供在方括号中间的值。

```
class IntList
{
private:
    int m_list[10]{};

public:
    int& operator[] (int index);
};

int& IntList::operator[] (int index)
{
    return m_list[index];
}
```

现在，任何使用对 IntList 类使用下标操作符 []，编译器将会返回 m_list 成员变量中相应的元素！这就使得直接对 m_list 进行 get 和 set 成为可能：

```
IntList list{};
list[2] = 3; // set 一个值
std::cout << list[2] << '\n'; // get 一个值

return 0;
```

这样子简单的语法以及理解都做到了。当 `list[2]` 被解析时，编译器首先检查是否存在一个重载的操作符 [] 函数。如果存在，那么传递方括号中的值作为参数传入函数。

注意尽管用户可以为函数提供默认值，然而使用操作符 [] 时不提供下标在方括号中并不认为是一个合理的语法，因此没有意义。
Tip：C++23 将为操作符 [] 添加若干下标的支持。

为什么操作符 [] 返回一个引用

让我们观察一下 `list[2] = 3` 是如何被解析的。由于下标操作符相较于赋值操作符拥有更高的优先级，`list[2]` 先被解析。`list[2]` 调用操作符 []，其被我们定义为返回 `list.m_list[2]` 的引用。因为操作符 [] 返回的是一个引用，它返回的是真实的 `list.m_list[2]` 数组元素。部分被解析的表达式变成了 `list.m_list[2] = 3`，即一个直接的整数赋值。
在 9.2 值分类（左值与右值）中，学到了在赋值声明左侧的任何值都必须是一个左值（即一个拥有真实内存地址的变量）。因为操作符 [] 可以在赋值声明中左侧使用（例如 `list[2] = 3`），那么操作符 [] 的返回值必须为左值。事实就是引用总是左值，因为仅可以使用引用变量的内存地址。因此返回引用满足了编译器所需要的返回左值。
思考一下如果操作符 [] 是值返回了一个整数而不是引用。`list[2]` 则会调用操作符 []，其返回的是 `list.m_list[2]` 的值。举个例子，如果 `m_list[2]` 本来拥有的值是 6，操作符 [] 将返回值 6。那么 `list[2] = 3` 将被解析为 `6 = 3`，这将毫无意义！如果这么做了，编译器则会抱怨：

```
error C2106: '=' : left operand must be l-value
```

处理 const 对象

上述的 IntList 例子中，操作符 [] 是非 const 的，因此我们可以用它作为左值来修改非 const 对象的状态。然而，如果 IntList 对象是 const 的呢？这种情况下是无法调用非 const 版本的操作符 []，因为这会允许潜在的修改一个 const 对象的状态。
好消息是我们可以分开定义一个非 const 版本以及一个 const 版本的操作符 []。非 const 版本将会使用非 const 对象，而 const 版本则使用 const 对象。

```
#include <iostream>

class IntList
{
private:
    int m_list[10]{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }; // 为了演示，这里给一些初始值

public:
    int& operator[] (int index);
    int operator[] (int index) const; // 也可以返回 const int& 如果返回的是一个非基础类型
};

int& IntList::operator[] (int index) // 对于非 const 对象而言：可以用于赋值
```

```
{
    return m_list[index];
}

int IntList::operator[] (int index) const // 对于 const 对象而言：仅可以用于访问
{
    return m_list[index];
}

int main()
{
    IntList list{};
    list[2] = 3; // 可行：调用非 const 版本的 operator[]
    std::cout << list[2] << '\n';

    const IntList clist{};
    clist[2] = 3; // 编译错误：调用 const 版本的 operator[]，即返回值。不可用于赋值，因为其为右值。
    std::cout << clist[2] << '\n';

    return 0;
}
```

错误检查

重载下标操作符的另一个好处就是我们可以安全的进行访问而不是直接访问数组。同侧人员，当访问数组时，下标操作符并不会检查索引是否有效。例如编译器不会抱怨以下代码：

```
int list[5]{};
list[7] = 3; // 索引 7 超出边界了！
```

然而如果知道数组的大小，我们可以使重载下标操作符进行检查确保索引在边界内：

```
#include <cassert> // assert()
#include <iterator> // std::size()

class IntList
{
private:
    int m_list[10]{};

public:
    int& operator[] (int index);
};

int& IntList::operator[] (int index)
{
    assert(index >= 0 && index < std::size(m_list));

```

```
    return m_list[index];
}
```

上述例子中使用了 `assert()` 函数（引用于 `cassert` 头文件中）来确保索引有效。如果在 assert 中的表达式解析为 false 时（意味着用户传入了无效索引），程序则会带上报错信息被终结，这就比另一种方式（污染内存）更好了。这可能是检查类似这种问题的最通常的办法了。

不要混淆对象的指针与重载操作符 []

如果尝试调用操作符 [] 在一个对象的指针上，C++ 将假设用户在尝试索引该类型的数组。考虑下列代码：

```
#include <cassert> // for assert()
#include <iterator> // for std::size()

class IntList
{
private:
    int m_list[10]{};

public:
    int& operator[] (int index);
};

int& IntList::operator[] (int index)
{
    assert(index >= 0 && index < std::size(m_list));

    return m_list[index];
}

int main()
{
    IntList* list{ new IntList{} };
    list [2] = 3; // 错误：这会假设我们在访问 IntLists 的数组的索引 2
    delete list;

    return 0;
}
```

因为我们不能对一个 IntList 赋值一个整数，这并不能通过编译。然而，如果赋值整数是有效的，这将被编译并运行，最终得到未定义的结果。

规则：确保没用尝试对一个对象的指针进行重载操作符 [] 的调用。

合理的语法将会是先解引用（确保在操作符 [] 之前使用括号，因为操作符 [] 的优先级比操作符 * 要高），接着再调用操作符 []：

```
int main()
```

```
{
    IntList* list{ new IntList{} };
    (*list)[2] = 3; // get our IntList object, then call overloaded operator[]
    delete list;

    return 0;
}
```

这很丑并且容易出错。最好的办法就是不要对不必要的对象设置指针。

函数参数并不是必须为一个整数

如上所述 C++ 会传递方括号中间的值给到重载函数。大多数情况下会是一个整数值。然而这并不是必须的 – 实际上用户可以定义自己的重载操作符 [] 来获取任何所需要的类型。可以定义操作符 [] 获取 double，一个字符串，或者是任何其他类型。
一个荒谬的例子用于展示它是如何工作的：

```
#include <iostream>
#include <string_view> // C++17

class Stupid
{
private:

public:
  void operator[] (std::string_view index);
};

// 重载操作符[] 来打印什么东西并没有意义
// 但是这是简单的办法来展示函数参数可以为非整数
void Stupid::operator[] (std::string_view index)
{
  std::cout << index;
}

int main()
{
  Stupid stupid{};
  stupid["Hello, world!"];

  return 0;
}
```

正如预期，打印：

```
Hello, world!
```

当编写某种类的时候，重载操作符 [] 并获取字符串参数可以很有用，例如使用字符串作为索引。

14.10 Overloading the parenthesis operator

迄今为止我们学习到的所有重载操作符都是用户定义操作符参数的类型，而没有参数的数量（根据操作符的类型而固定）。例如，操作符 == 总是需要两个参数，而操作符! 总是一个。圆括号操作符（操作符 ()）是一个特别有趣的操作符，它允许用户同时设定参数的类型与数量。
有两件事需要记住：首先，圆括号操作符必须以成员函数的方式实现；其次，在非 OO 的 C++ 中，操作符 () 是用于调用函数的。在类的情况下，操作符 () 仅是一个与任何其他重载操作符一样的普通操作符，即调用一个函数（被称为操作符 ()）。

一个例子

让我们来看一个适用于重载这个操作符的例子：

```
class Matrix
{
private:
    double data[4][4]{};
};
```

Matrix 是线性代数的核心组件，通常用于几何建模以及 3D 图像计算。本例的矩阵是一个 4 乘 4 的二维 double 数组。
上一章讲到了可以重载操作符 [] 来提供直接访问私有的一维数组的能力。然而本例中，我们希望访问的是私有的二维数组。因为操作符 [] 局限于单个参数，索引二维数组的时候并不高效。然而由于操作符 () 可以接受任意数量的参数，我们可以声明一个版本的操作符 () 来接受两个整数索引参数，并且使用它们来访问二维数组。

```
#include <cassert> // for assert()

class Matrix
{
private:
    double m_data[4][4]{};

public:
    double& operator()(int row, int col);
    double operator()(int row, int col) const; // 用作于 const 对象
};

double& Matrix::operator()(int row, int col)
{
    assert(col >= 0 && col < 4);
    assert(row >= 0 && row < 4);

    return m_data[row][col];
}

```

```
double Matrix::operator()(int row, int col) const
{
    assert(col >= 0 && col < 4);
    assert(row >= 0 && row < 4);

    return m_data[row][col];
}
```

现在可以定义一个 Matrix 并访问其元素：

```
#include <iostream>

int main()
{
    Matrix matrix;
    matrix(1, 2) = 4.5;
    std::cout << matrix(1, 2) << '\n';

    return 0;
}
```

打印：

```
4.5
```

现在让我们再次重载操作符 ()，这一次不接受任何参数：

```
#include <cassert> // for assert()
class Matrix
{
private:
    double m_data[4][4]{};

public:
    double& operator()(int row, int col);
    double operator()(int row, int col) const;
    void operator()();
};

double& Matrix::operator()(int row, int col)
{
    assert(col >= 0 && col < 4);
    assert(row >= 0 && row < 4);

    return m_data[row][col];
}

double Matrix::operator()(int row, int col) const
{
    assert(col >= 0 && col < 4);
    assert(row >= 0 && row < 4);
```

```
25
26      return m_data[row][col];
27  }
28
29  void Matrix::operator()()
30  {
31      // reset all elements of the matrix to 0.0
32      for (int row{ 0 }; row < 4; ++row)
33      {
34          for (int col{ 0 }; col < 4; ++col)
35          {
36              m_data[row][col] = 0.0;
37          }
38      }
39  }
```

现在可以声明一个 Matrix 同时访问其元素：

```
1  #include <iostream>
2
3  int main()
4  {
5      Matrix matrix{};
6      matrix(1, 2) = 4.5;
7      matrix(); // erase matrix
8      std::cout << matrix(1, 2) << '\n';
9
10     return 0;
11 }
```

打印：

```
1  0
```

由于操作符 () 非常的灵活，它可以被用于很多不同的目的。然而这是强烈不建议的行为，因为 () 符号并没有给与任何有关操作符的指示。上述的例子中，清除的功能更好的做法是调用一个 `clear()` 或是 `erase()` 的函数，因为 `matrix.erase()` 相较于 `matrix()` 更便于理解。

函子的乐趣

操作符 () 同样也被广泛的用于实现 **函子** functor（或 **function object**），即类似于函数的类。函子比普通函数的好处在于函子可以存储数据于成员变量中（因为它们是类）。

```
1  #include <iostream>
2
3  class Accumulator
4  {
5  private:
6      int m_counter{ 0 };
```

```
7
8  public:
9      int operator() (int i) { return (m_counter += i); }
10 };
11
12 int main()
13 {
14     Accumulator acc{};
15     std::cout << acc(10) << '\n'; // prints 10
16     std::cout << acc(20) << '\n'; // prints 30
17
18     return 0;
19 }
```

注意使用 Accumulator 像是在做一个普通的函数调用，但是 Accumulator 类存储了一个积累的值。

这里有可能会疑惑为什么我们不能用普通函数以及 static 本地变量来保存函数调用后的数据来做同样的事情呢。可以，但是因为函数仅有一个全局实例，而始终只能使用一个将会是很大的限制。而函子，我们可以实例化若干个函子对象，并且非常自然的使用它们。

14.11 Overloading typecasts

在 8.5 显式类型转换（casting）以及 static_cast 中学习到了 C++ 允许用户转换类型。C++ 直到如何转换内建的数据类型。然而，它不知道如何转换任何用户定义的类。这就是重载类型转换操作符的意义所在。

用户定义转换 user-defined conversions 允许我们转换类成为另一种数据类型。

```
1  class Cents
2  {
3  private:
4      int m_cents;
5  public:
6      Cents(int cents=0)
7          : m_cents{ cents }
8      {
9      }
10
11     int getCents() const { return m_cents; }
12     void setCents(int cents) { m_cents = cents; }
13 };
```

这个类非常的简单：它存储了 cents 为整数，提供访问函数用以 get 和 set cents 的数量，同时也提供了一个构造函数用于转换 int 成为一个 Cents。

如果可以转换一个 int 成为 Cents，那么对于我们而言，转换 Cents 回一个 int 是否同样有意义？某些情况下，这可能不合理，但是现在这种情况下是合理的。

上述的例子中需要使用 `getCents` 来转换 Cents 变量成一个整数，因此可以使用 `printInt()` 来进行打印：

```
#include <iostream>

void printInt(int value)
{
    std::cout << value;
}

int main()
{
    Cents cents{ 7 };
    printInt(cents.getCents()); // print 7

    std::cout << '\n';

    return 0;
}
```

如果已经编写了大量的函数用于接受整数作为参数，代码再调用 `getCents()` 就会变得非常凌乱。

为了让事情变得简单，我们可以通过重载 int 的类型转换来提供一个用户定义的转换。这样使得可以直接转换 Cents 成为 int。以下代码展示了如何做到的：

```
class Cents
{
private:
    int m_cents;
public:
    Cents(int cents=0)
        : m_cents{ cents }
    {
    }

    // 重载 int 转换
    operator int() const { return m_cents; }

    int getCents() const { return m_cents; }
    void setCents(int cents) { m_cents = cents; }
};
```

有三点需要注意的地方：

1. 为了重载函数用于转换类成为一个 int，我们需要编写一个名为 `operator int()` 的函数在类中。注意在 `operator` 与需要转换的类型之间有一个空格。
2. 用户定义的转换不接受参数，因为没有办法给它们传递参数。

3. 用户定义的转换没有返回类型，C++ 假设用户会返回正确的类型。

现在可以像这样调用 `printInt()` 了：

```
#include <iostream>

int main()
{
    Cents cents{ 7 };
    printInt(cents); // 打印 7

    std::cout << '\n';

    return 0;
}
```

编译器将首先注意到函数 `printInt` 接受一个整数类型。接着注意到变量 cents 并不是一个 int。最后，编译器会查看用户是否提供了转换 Cents 成为 int 的方法。这里将调用操作符 int() 函数，即返回一个 int，接着返回的 int 将会被传递给 `printInt` 。
也可以显式的转换 Cents 变量成为一个 int：

```
Cents cents{ 7 };
int c{ static_cast<int>(cents) };
```

用户可以提供任何希望用到的数据类型转换，包括用户自定义的数据类型！
这里是一个新的称为 Dollars 的提供了重载 Cents 转换的类：

```
#include <iostream>

class Cents
{
private:
    int m_cents;
public:
    Cents(int cents=0)
        : m_cents{ cents }
    {
    }

    // 重载 int 转换
    operator int() const { return m_cents; }

    int getCents() const { return m_cents; }
    void setCents(int cents) { m_cents = cents; }
};

class Dollars
{
private:
```

```
    int m_dollars;
public:
    Dollars(int dollars=0)
        : m_dollars{ dollars }
    {
    }

    // 允许我们将 Dollars 转换成 Cents
    operator Cents() const { return Cents { m_dollars * 100 }; }
};

void printCents(Cents cents)
{
    std::cout << cents; // cents 将被隐式转换成为一个 int
}

int main()
{
    Dollars dollars{ 9 };
    printCents(dollars); // dollars 将被隐式转换成为一个 Cents

    std::cout << '\n';

    return 0;
}
```

打印:

```
900
```

14.12 The copy constructor

回忆初始化的类型

首先让我们回忆一下 C++ 提供了哪些类型的初始化：直接（圆括号）初始化，标准（花括号）初始化以及拷贝（等号）初始化。

```
#include <cassert>
#include <iostream>

class Fraction
{
private:
    int m_numerator{};
    int m_denominator{};

public:
    // Default constructor
    Fraction(int numerator=0, int denominator=1)
```

```
        : m_numerator{numerator}, m_denominator{denominator}
    {
        assert(denominator != 0);
    }

    friend std::ostream& operator<<(std::ostream& out, const Fraction& f1);
};

std::ostream& operator<<(std::ostream& out, const Fraction& f1)
{
  out << f1.m_numerator << '/' << f1.m_denominator;
  return out;
}
```

我们可以做一个直接初始化：

```
int x(5); // 直接初始化一个整数
Fraction fiveThirds(5, 3); // 直接初始化一个 Fraction，调用 Fraction(int, int) 构造函数
```

C++11 中可以标准初始化：

```
int x { 5 }; // 标准初始化一个整数
Fraction fiveThirds {5, 3}; // 标准初始化一个 Fraction，调用 Fraction(int, int) 构造函数
```

最后是拷贝初始化：

```
int x = 6; // 拷贝初始化一个整数
Fraction six = Fraction(6); // 拷贝初始化一个 Fraction，将调用 Fraction(6, 1)
Fraction seven = 7; // 拷贝初始化一个 Fraction。编译器将会尝试寻找转换 7 成为 Fraction 的方法，
    这会唤起 Fraction(7, 1) 构造函数。
```

使用直接和标准初始化，对象被直接创建。然而拷贝初始化会更复杂一些。我们将在下一节开始深入探讨拷贝初始化，现在为了更高效的学习，我们需要迂回一下。

拷贝构造函数

考虑以下代码：

```
#include <cassert>
#include <iostream>

class Fraction
{
private:
    int m_numerator{};
    int m_denominator{};

public:
    // 默认构造函数
    Fraction(int numerator=0, int denominator=1)
        : m_numerator{numerator}, m_denominator{denominator}
```

```
    {
        assert(denominator != 0);
    }

    friend std::ostream& operator<<(std::ostream& out, const Fraction& f1);
};

std::ostream& operator<<(std::ostream& out, const Fraction& f1)
{
  out << f1.m_numerator << '/' << f1.m_denominator;
  return out;
}

int main()
{
  Fraction fiveThirds { 5, 3 }; // 标准初始化一个 Fraction，调用 Fraction(int, int) 构造函数
  Fraction fCopy { fiveThirds }; // 标准初始化一个 Fraction -- 用哪个构造函数?
  std::cout << fCopy << '\n';
}
```

如果编译了程序会发现可以通过，并且打印：

```
5/3
```

让我们仔细研究程序是如何工作的。

变量 fiveThirds 被标准初始化，并调用了 `Fraction(int, int)` 构造函数。这里并不出奇，那么第二行呢？变量 fCopy 很明显也是一个初始化，构造函数肯定是被调用了的。那么这个被调用的构造函数是什么？

答案是调用了 Fraction 的拷贝构造函数。**拷贝构造函数** copy constructor 是一种特殊的构造函数，用于从一个已有对象（相同类型）来创建一个新的拷贝对象。类似于默认构造函数，如果用户不提供那么 C++ 将会创造一个公有的拷贝构造函数。因为编译器并不了解用户的类，默认情况下，被编译器创造的拷贝构造函数会使用名为成员向初始化的初始化方法。**成员向初始化** memberwise initialization 简单的意为每个成员的拷贝直接从类的成员中进行拷贝。上述例子中 `fCopy.m_numerator` 将会从 `fiveThirds.m_numerator` 开始进行初始化，以此类推。正如同显式定义一个默认构造函数那样，我们同样可以显式定义一个拷贝构造函数。

```
#include <cassert>
#include <iostream>

class Fraction
{
private:
    int m_numerator{};
    int m_denominator{};

public:
```

```
    // 默认构造函数
    Fraction(int numerator=0, int denominator=1)
        : m_numerator{numerator}, m_denominator{denominator}
    {
        assert(denominator != 0);
    }

    // 拷贝构造函数
    Fraction(const Fraction& fraction)
        : m_numerator{fraction.m_numerator}, m_denominator{fraction.m_denominator}
        // 注意：我们可以直接访问参数 fraction 的成员，因为处在 Fraction 类内部
    {
        // 这里不需要检查分母是否为 0，因为 fraction 必须是已有的合法 Fraction
        std::cout << "Copy constructor called\n"; // 仅用于证明其有效
    }

    friend std::ostream& operator<<(std::ostream& out, const Fraction& f1);
};

std::ostream& operator<<(std::ostream& out, const Fraction& f1)
{
  out << f1.m_numerator << '/' << f1.m_denominator;
  return out;
}

int main()
{
  Fraction fiveThirds { 5, 3 }; // 直接初始化一个 Fraction，调用 Fraction(int, int) 构造函数
  Fraction fCopy { fiveThirds }; // 直接初始化 -- 通过 Fraction 拷贝构造函数
  std::cout << fCopy << '\n';
}
```

打印：

```
5/3
```

例子中定义的拷贝构造函数使用了成员向的初始化，其功能等同于 C++ 提供的默认构造函数，只不过加了打印声明。

拷贝构造函数的参数必须是引用

对于拷贝构造函数而言，其参数必须是一个（const）引用。这有道理：如果是值传递，则需要拷贝构造函数去拷贝参数成为拷贝构造函数的入参（也就是无限递归）。

阻止拷贝 可以使拷贝构造函数私有来阻止拷贝：

```
#include <cassert>
#include <iostream>
```

```

class Fraction
{
private:
    int m_numerator{};
    int m_denominator{};

    // 拷贝构造函数（私有）
    Fraction(const Fraction& fraction)
        : m_numerator{fraction.m_numerator}, m_denominator{fraction.m_denominator}
    {
        // 这里不需要检查分母是否为 0，因为 fraction 必须是已有的合法 Fraction
        std::cout << "Copy constructor called\n"; // 仅用于证明其有效
    }

public:
    // 拷贝构造函数
    Fraction(int numerator=0, int denominator=1)
        : m_numerator{numerator}, m_denominator{denominator}
    {
        assert(denominator != 0);
    }

    friend std::ostream& operator<<(std::ostream& out, const Fraction& f1);
};

std::ostream& operator<<(std::ostream& out, const Fraction& f1)
{
  out << f1.m_numerator << '/' << f1.m_denominator;
  return out;
}

int main()
{
  Fraction fiveThirds { 5, 3 }; // 直接初始化一个 Fraction，调用 Fraction(int, int) 构造函数
  Fraction fCopy { fiveThirds }; // 拷贝初始化是私有的，这行编译错误
  std::cout << fCopy << '\n';
}
```

拷贝构造函数可能被省略 考虑下列代码：

```
#include <cassert>
#include <iostream>

class Fraction
{
private:
  int m_numerator{};
  int m_denominator{};
```

```

public:
    // 默认构造函数
    Fraction(int numerator=0, int denominator=1)
        : m_numerator{numerator}, m_denominator{denominator}
    {
        assert(denominator != 0);
    }

  // 拷贝构造函数
  Fraction(const Fraction &fraction)
    : m_numerator{fraction.m_numerator}, m_denominator{fraction.m_denominator}
  {
    std::cout << "Copy constructor called\n"; // 仅用于证明其有效
  }

  friend std::ostream& operator<<(std::ostream& out, const Fraction& f1);
};

std::ostream& operator<<(std::ostream& out, const Fraction& f1)
{
  out << f1.m_numerator << '/' << f1.m_denominator;
  return out;
}

int main()
{
  Fraction fiveThirds { Fraction { 5, 3 } };
  std::cout << fiveThirds;
  return 0;
}
```

现在来思考一下这个代码是如何工作的。首先，直接初始化了一个匿名的 Fraction 对象，使用了 `Fraction(int, int)` 构造函数。接着使用该匿名对象作为 Fraction fiveThirds 的初始对象。因为匿名对象是一个 Fraction，那么 fiveThirds 应该调用拷贝构造函数对么?
编译并运行这段程序，可能会认为答案如下:

```
copy constructor called
5/3
```

而实际上，更可能得到的答案是这样:

```
5/3
```

注意初始化一个匿名对象接着使用其直接初始化对象花费了两步（一是创建匿名对象，一是调用拷贝构造函数）。然而结果却是直接初始化，即仅花费了一步。
因为这个原因，这种情况下编译器允许优化调用拷贝函数并直接做一次的直接初始化。这个以优化为目的的省略特定拷贝（或移动）的步骤被称为 **elision**。

14.13 Copy initialization

考虑以下代码:

```
int x = 5;
```

该声明使用了拷贝初始化来初始化一个新建值为 5 的 x 变量。
而类的初始化稍微会复杂一些，因为它们使用了构造函数来进行初始化。本节将测试类的拷贝初始化的所有主题。

类的拷贝初始化

给定的 Fraction 类:

```
#include <cassert>
#include <iostream>

class Fraction
{
private:
    int m_numerator;
    int m_denominator;

public:
    // Default constructor
    Fraction(int numerator=0, int denominator=1)
        : m_numerator(numerator), m_denominator(denominator)
    {
        assert(denominator != 0);
    }

    friend std::ostream& operator<<(std::ostream& out, const Fraction& f1);
};

std::ostream& operator<<(std::ostream& out, const Fraction& f1)
{
  out << f1.m_numerator << '/' << f1.m_denominator;
  return out;
}
```

考虑以下:

```
int main()
{
    Fraction six = Fraction(6);
    std::cout << six;
    return 0;
}
```

如果尝试编译并运行它，会打印预期:

```
6/1
```

这种样式的拷贝初始化被解析成等同于如下的样式：

```
Fraction six(Fraction(6));
```

从上一节中学习到了，它潜在的调用了 `Fraction(int, int)` 以及 Fraction 的拷贝构造函数（可能会因为性能原因被省略）。然而由于省略并不是保证的（在 C++17 之前，省略不像现在是强制性的），所以最好是避免类的拷贝初始化，而是使用标准初始化。
最佳实践：避免使用拷贝初始化，而是使用标准初始化。

其它用到拷贝初始化的地方

还有其他一些地方又到了拷贝初始化，不过只有两个值得显式提及的。当传递或值返回一个类，这个过程使用了拷贝初始化。

```
#include <cassert>
#include <iostream>

class Fraction
{
private:
  int m_numerator;
  int m_denominator;

public:
    Fraction(int numerator=0, int denominator=1)
        : m_numerator(numerator), m_denominator(denominator)
    {
        assert(denominator != 0);
    }

  Fraction(const Fraction& copy) :
    m_numerator(copy.m_numerator), m_denominator(copy.m_denominator)
  {
    std::cout << "Copy constructor called\n"; // 仅用于证明其有效
  }

  friend std::ostream& operator<<(std::ostream& out, const Fraction& f1);
  int getNumerator() { return m_numerator; }
  void setNumerator(int numerator) { m_numerator = numerator; }
};

std::ostream& operator<<(std::ostream& out, const Fraction& f1)
{
  out << f1.m_numerator << '/' << f1.m_denominator;
  return out;
}
```

```

Fraction makeNegative(Fraction f) // 理想情况下应该使用 const 引用
{
    f.setNumerator(-f.getNumerator());
    return f;
}

int main()
{
    Fraction fiveThirds(5, 3);
    std::cout << makeNegative(fiveThirds);

    return 0;
}
```

上述代码中，函数 `makeNegative` 值接受一个 Fraction。当程序运行时：

```
Copy constructor called
Copy constructor called
-5/3
```

第一个拷贝构造函数调用发生在 fiveThirds 作为参数传递给 `makeNegative()` 的入参。第二次调用发生在当 `makeNegative()` 返回值被传递回 `main()`。

上述例子中，参数值传递以及返回值都不能被省略。然而在其他情况下，如果参数或返回值满足特定标准时，编译器可能会优化拷贝构造函数使其省略。例如：

```
#include <iostream>
class Something
{
public:
  Something() = default;
  Something(const Something&)
  {
    std::cout << "Copy constructor called\n";
  }
};

Something foo()
{
  return Something(); // 拷贝构造函数在这里与寻常情况一样被调用
}
Something goo()
{
  Something s;
  return s; // 拷贝构造函数在这里与寻常情况一样被调用
}

int main()
{
```

```
  std::cout << "Initializing s1\n";
  Something s1 = foo(); // 拷贝构造函数在这里与寻常情况一样被调用

  std::cout << "Initializing s2\n";
  Something s2 = goo(); // 拷贝构造函数在这里与寻常情况一样被调用
}
```

上面的代码中将会调用拷贝构造函数 4 次 – 然而由于拷贝省略，用户的编译器有可能将省略大部分或所有的情况。Visual Studio 2019 省略 3 次（没有省略 `goo()` 的返回），以及 GCC 省略所有的 4 次。

14.14 Converting constructors, explicit, and delete

默认情况下 C++ 视任何构造函数为隐式转换操作符。考虑下面代码：

```
#include <cassert>
#include <iostream>

class Fraction
{
private:
  int m_numerator;
  int m_denominator;

public:
  Fraction(int numerator = 0, int denominator = 1)
     : m_numerator(numerator), m_denominator(denominator)
  {
    assert(denominator != 0);
  }

  Fraction(const Fraction& copy)
    : m_numerator(copy.m_numerator), m_denominator(copy.m_denominator)
  {
    std::cout << "Copy constructor called\n";
  }

  friend std::ostream& operator<<(std::ostream& out, const Fraction& f1);
  int getNumerator() { return m_numerator; }
  void setNumerator(int numerator) { m_numerator = numerator; }
};

void printFraction(const Fraction& f)
{
  std::cout << f;
}

std::ostream& operator<<(std::ostream& out, const Fraction& f1)
```

```
{
  out << f1.m_numerator << '/' << f1.m_denominator;
  return out;
}

int main()
{
  printFraction(6);

  return 0;
}
```

尽管函数 `printFraction()` 预期一个 Fraction，我们提供的却是字面值整数 6。由于 Fraction 拥有一个构造函数将获取一个整数，编译器将隐式转化字面值 6 成为一个 Fraction 对象。这么做是借助于初始化 `printFraction()` 参数 f 时使用了 `Fraction(int, int)` 构造函数。因此上述代码打印：

```
6/1
```

这个隐式转换作用于所有的初始化（直接与拷贝）。

explicit 关键字

在 Fraction 案例中隐式转换是有意义的，但在其他情况下可能并不可取，或是导致非预期的行为：

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

class MyString
{
private:
  std::string m_string;
public:
  MyString(int x) // 分配 x 大小的字符串
  {
    m_string.resize(x);
  }

  MyString(const char* string) // 分配字符串用于存储字符串的值
  {
    m_string = string;
  }

  friend std::ostream& operator<<(std::ostream& out, const MyString& s);

};

```

```
std::ostream& operator<<(std::ostream& out, const MyString& s)
{
  out << s.m_string;
  return out;
}

void printString(const MyString& s)
{
  std::cout << s;
}

int main()
{
  MyString mine = 'x'; // 可编译并使用 MyString(int)
  std::cout << mine << '\n';

  printString('x'); // 可编译并使用 MyString(int)
  return 0;
}
```

上述例子中，用户尝试通过一个字符来初始化字符串。由于字符是整数家庭的一部分，编译器使用了转换的构造函数 `MyString(int)` 来隐式转换 char 成为 MyString。该程序打印 MyString 非预期的结果。同样的，调用 `printString('x')` 导致隐式转换返回同样的问题。

解决该问题的一种方法是使构造函数（以及转换函数）通过 explicit 关键字使它们变成显式的，做法是放置其在函数名前。构造函数与转换函数是显式的情况下将不会使用隐式构造函数或拷贝初始化：

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

class MyString
{
private:
  std::string m_string;
public:
  // explicit 关键字使该构造函数的隐式转换变为不合法
  explicit MyString(int x) // 分配 x 大小的字符串
  {
    m_string.resize(x);
  }

  MyString(const char* string) // 分配字符串用于存储字符串的值
  {
    m_string = string;
  }

  friend std::ostream& operator<<(std::ostream& out, const MyString& s);

```

```
};

std::ostream& operator<<(std::ostream& out, const MyString& s)
{
  out << s.m_string;
  return out;
}

void printString(const MyString& s)
{
  std::cout << s;
}

int main()
{
  MyString mine = 'x'; // 编译错误，因为 MyString(int) 现在是 explicit 的，且没有任何与之匹配的
  std::cout << mine;

  printString('x'); // 编译错误，因为 MyString(int) 不能用于隐式转换

  return 0;
}
```

上述的程序不能通过编译，因为 `MyString(int)` 变成了显式的了，且合适的转化构造函数不能被找到用于隐式转换'x' 成为一个 MyString。
然而注意让一个构造函数 explicit 仅阻止隐式转换。显式转换（通过 casting）仍然是被允许的：

```
std::cout << static_cast<MyString>(5); // 允许：显式转换 5 至 MyString(int)
```

直接或标准初始化仍然会转换参数来匹配（标准初始化不会做狭窄转换，但是会做其他类型的转换）：

```
MyString str{'x'}; // 允许：初始化参数可能仍然被隐式转换来进行匹配
```

最佳实践：explicit 构造函数与用户定义转换成员函数可以阻止隐式转换错误。

delete 关键字

MyString 的例子中，希望完全禁止'x' 被转换成 MyString（无论是隐式或显式，因为结果并不符合直觉）。一种办法是添加 `MyString(char)` 构造函数，并使其私有：

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

class MyString
{
private:
  std::string m_string;

```

```
  MyString(char) // MyString(char) 类型的对象不能在类的外部进行构造
  {
  }

public:
  // explicit 关键字使得该构造函数的隐式转换不合法
  explicit MyString(int x) // allocate string of size x
  {
    m_string.resize(x);
  }

  MyString(const char* string) // 分配字符串用于存储值
  {
    m_string = string;
  }

  friend std::ostream& operator<<(std::ostream& out, const MyString& s);

};

std::ostream& operator<<(std::ostream& out, const MyString& s)
{
  out << s.m_string;
  return out;
}

int main()
{
  MyString mine('x'); // 编译错误，因为 MyString(char) 是私有的
  std::cout << mine;
  return 0;
}
```

然而构造函数仍然可以在类的内部使用（私有访问仅对于调用该函数的非成员生效）。
一个更好的方案来解决这个问题就是使用“delete”关键字：

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

class MyString
{
private:
  std::string m_string;

public:
  MyString(char) = delete; // 任何使用该构造函数的都视为错误

  explicit MyString(int x)
  {
```

```
14      m_string.resize(x);
15    }
16
17    MyString(const char* string)
18    {
19      m_string = string;
20    }
21
22    friend std::ostream& operator<<(std::ostream& out, const MyString& s);
23
24  };
25
26  std::ostream& operator<<(std::ostream& out, const MyString& s)
27  {
28    out << s.m_string;
29    return out;
30  }
31
32  int main()
33  {
34    MyString mine('x'); // 编译错误，因为 MyString(char) 是 deleted 的
35    std::cout << mine;
36    return 0;
37  }
```

14.15 Overloading the assignment operator

赋值操作符 assignment operator（操作符 =）用于从一个对象拷贝值至另一个*已存在的对象*。

赋值 vs 拷贝构造函数

拷贝构造函数以及赋值操作符的目的几乎相等 – 都是拷贝一个对象至另一个对象。然而拷贝构造函数初始化新对象，而赋值操作符替换已有对象的内容。
它们之间的差异会让很多新手程序员产生疑惑，不过并没有那么的难。总结一下：

- 如果一个新对象必须在拷贝出现之前被创建，那么使用拷贝构造函数（注：这包含了值传递与返回对象）。
- 如果一个新对象不需要在拷贝前被创建，使用赋值操作符。

重载赋值操作符

重载赋值操作符（操作符 =）非常的直观，仅需要一个特定条款就可以实现。赋值操作符必须重载为成员函数。

```cpp
#include <cassert>
#include <iostream>

class Fraction
{
private:
  int m_numerator { 0 };
  int m_denominator { 1 };

public:
  // 默认构造函数
  Fraction(int numerator = 0, int denominator = 1 )
    : m_numerator { numerator }, m_denominator { denominator }
  {
    assert(denominator != 0);
  }

  // 拷贝构造函数
  Fraction(const Fraction& copy)
    : m_numerator { copy.m_numerator }, m_denominator { copy.m_denominator }
  {
    // 这里不需要检查分母是否为 0，因为 fraction 必须是已有的合法 Fraction
    std::cout << "Copy constructor called\n"; // 仅用于证明其有效
  }

  // 重载赋值操作符
  Fraction& operator= (const Fraction& fraction);

  friend std::ostream& operator<<(std::ostream& out, const Fraction& f1);

};

std::ostream& operator<<(std::ostream& out, const Fraction& f1)
{
  out << f1.m_numerator << '/' << f1.m_denominator;
  return out;
}

// 一个简单的实现（更好的实现详见后面）
Fraction& Fraction::operator= (const Fraction& fraction)
{
    // 进行拷贝
    m_numerator = fraction.m_numerator;
    m_denominator = fraction.m_denominator;

    // 返回已有对象使得可以串联该操作符
    return *this;
}

```

```
int main()
{
    Fraction fiveThirds { 5, 3 };
    Fraction f;
    f = fiveThirds; // 调用重载赋值
    std::cout << f;

    return 0;
}
```

打印:

```
5/3
```

现在来看非常的直观，重载操作符 = 返回 *this 使得若干赋值可以被串联起来:

```
int main()
{
    Fraction f1 { 5, 3 };
    Fraction f2 { 7, 2 };
    Fraction f3 { 9, 5 };

    f1 = f2 = f3; // 串联赋值

    return 0;
}
```

自赋值 self-assignment 的问题

从这里开始事情变得有趣了起来。C++ 允许自赋值:

```
int main()
{
    Fraction f1 { 5, 3 };
    f1 = f1; // self assignment

    return 0;
}
```

这将调用 `f1.operator=(f1)`，在上述的简单实现下所有的成员都会被赋值。这个特定的例子中，自赋值导致每个成员都被赋予其自身，没有整体的影响，除了浪费了时间。大多数情况下，自赋值并不需要做任何事情!
然而在赋值操作符需要动态分配内存时，自赋值将会变得很危险:

```
#include <iostream>

class MyString
{
private:
```

```
6   char* m_data {};
7   int m_length {};
8
9 public:
10  MyString(const char* data = nullptr, int length = 0 )
11    : m_length { length }
12  {
13    if (length)
14    {
15      m_data = new char[length];
16
17      for (int i { 0 }; i < length; ++i)
18        m_data[i] = data[i];
19    }
20  }
21  ~MyString()
22  {
23    delete[] m_data;
24  }
25
26  // 重载赋值
27  MyString& operator= (const MyString& str);
28
29  friend std::ostream& operator<<(std::ostream& out, const MyString& s);
30 };
31
32 std::ostream& operator<<(std::ostream& out, const MyString& s)
33 {
34   out << s.m_data;
35   return out;
36 }
37
38 // 一个简单的操作符= 的实现（不要使用）
39 MyString& MyString::operator= (const MyString& str)
40 {
41   // 如果当前字符串有数据存在，删除它
42   if (m_data) delete[] m_data;
43
44   m_length = str.m_length;
45
46   // 从 str 中拷贝数据至隐式对象
47   m_data = new char[str.m_length];
48
49   for (int i { 0 }; i < str.m_length; ++i)
50     m_data[i] = str.m_data[i];
51
52   // 返回现有的对象使其可被串联
53   return *this;
54 }
```

```
55
56 int main()
57 {
58   MyString alex("Alex", 5); // 遇到 Alex
59   MyString employee;
60   employee = alex; // Alex 是最新的雇员
61   std::cout << employee; // 雇员，念出你的名字
62
63   return 0;
64 }
```

首先运行上述程序将会看到打印“Alex”。现在运行以下：

```
1 int main()
2 {
3     MyString alex { "Alex", 5 }; // 遇到 Alex
4     alex = alex; // Alex 是他自己
5     std::cout << alex; // Alex，念出你的名字
6
7     return 0;
8 }
```

很大概率会获得垃圾输出。这是怎么回事?
思考当隐式对象以及传递给参数（str）都是变量 alex 的情况下，重载操作符 = 发生了什么。本例中 `m_data` 与 `str.m_data` 一样。首先发生的是函数检查隐式对象是否已经有一个字符串。如果有，它需要删除，这样不会有内存泄漏。本例 `m_data` 被分配了，所以函数删除了 `m_data` 。但是因为 str 与 *this 相同，我们希望拷贝的字符串已经被删除了，所以 `m_data`（以及 `str.m_data` ）变为悬垂。
之后又分配了新的内存给 `m_data`（以及 `str.m_data`)。因此结果就是从 `str.m_data` 拷贝进 `m_data` ，也就是拷贝了垃圾。

检测并处理自赋值

幸运的是我们可以坚持自赋值何时发生。以下是一个更新后的重载操作符 = 的实现：

```
1 MyString& MyString::operator= (const MyString& str)
2 {
3   // 自赋值检测
4   if (this == &str)
5     return *this;
6
7   // 如果当前字符串有数据存在，删除它
8   if (m_data) delete[] m_data;
9
10  m_length = str.m_length;
11
12  // 从 str 中拷贝数据至隐式对象
```

```
  m_data = new char[str.m_length];

  for (int i { 0 }; i < str.m_length; ++i)
    m_data[i] = str.m_data[i];

  // 返回现有的对象使其可被串联
  return *this;
}
```

通过检测因素对象的地址是否与传入对象的地址一致，可以让复制操作符立刻返回而不需要做额外的工作。
由于这是一个指针比较，它应该非常的快，且不需要操作符 == 被重载。

何时不去处理自赋值

通常自赋值检测在拷贝构造函数时跳过。因为被拷贝构造的对象是新建的，仅在新建的对象可以等于被拷贝的对象时是当尝试通过自身来初始化一个新定义的对象：

```
someClass c { c };
```

这种情况下，编译器会警告 `c` 是一个未初始化的变量。
其次，在类可以自然处理自赋值的时候，自赋值检查可能被省略。考虑 Fraction 类赋值操作符有一个自赋值保障：

```
// 一个更好的操作符= 实现
Fraction& Fraction::operator= (const Fraction& fraction)
{
    // 自赋值保护
    if (this == &fraction)
        return *this;

    // 进行拷贝
    m_numerator = fraction.m_numerator; // 可处理自赋值
    m_denominator = fraction.m_denominator; // 可处理自赋值

    // 返回已有对象使得可以串联该操作符
    return *this;
}
```

如果自赋值保护不存在，该函数将仍然在自赋值时进行正确操作（因为所有由函数完成的操作可以正确的处理自赋值）。
由于自赋值很罕见，一些著名的 C++ 专家建议省略自赋值保护，即使在类中可以获得好处。这里不建议这么做，因为我们相信代码防护是更好的实践，其次才是可选的优化。

拷贝与交换方言

一个处理自赋值问题更好的方法是通过拷贝与交换方言。详情请见 Stack Overflow。

默认赋值操作符

不同于其他操作符，编译器在用户没有提供自定义的复制操作符情况下，会提供默认的公有复制操作符。该赋值操作符会进行成员向的复制（即等效于成员向的默认拷贝构造函数的初始化）。与其他构造函数和操作符一样，可以通过私有化复制操作符或者 delete 关键字来阻止赋值：

```
#include <cassert>
#include <iostream>

class Fraction
{
private:
  int m_numerator { 0 };
  int m_denominator { 1 };

public:
    // 默认构造函数
    Fraction(int numerator = 0, int denominator = 1)
        : m_numerator { numerator }, m_denominator { denominator }
    {
        assert(denominator != 0);
    }

  // 拷贝构造函数
  Fraction(const Fraction &copy) = delete;

  // 重载赋值
  Fraction& operator= (const Fraction& fraction) = delete; // 通过赋值没有拷贝!

  friend std::ostream& operator<<(std::ostream& out, const Fraction& f1);

};

std::ostream& operator<<(std::ostream& out, const Fraction& f1)
{
  out << f1.m_numerator << '/' << f1.m_denominator;
  return out;
}

int main()
{
    Fraction fiveThirds { 5, 3 };
    Fraction f;
    f = fiveThirds; // 编译错误，操作符= 被删除了
    std::cout << f;

    return 0;
}
```

14.16 Shallow vs deep copying

浅拷贝

由于 C++ 并不熟悉用户的类，其所提供的默认拷贝构造函数与赋值操作符使用的拷贝方法是成员向的拷贝（同样也被称为**浅拷贝** shallow copy）。这就意味着 C++ 分别拷贝每个成员对象（使用重载的赋值操作符 =，以及拷贝构造函数的直接初始化）。当类很简单（例如没有包含任何动态内存分配）时，这样做没有任何问题。

例如再来看一下 Fraction 的例子：

```
#include <cassert>
#include <iostream>

class Fraction
{
private:
    int m_numerator { 0 };
    int m_denominator { 1 };

public:
    // 默认构造函数
    Fraction(int numerator = 0, int denominator = 1)
        : m_numerator{ numerator }
        , m_denominator{ denominator }
    {
        assert(denominator != 0);
    }

    friend std::ostream& operator<<(std::ostream& out, const Fraction& f1);
};

std::ostream& operator<<(std::ostream& out, const Fraction& f1)
{
  out << f1.m_numerator << '/' << f1.m_denominator;
  return out;
}
```

由编译器为该类提供的默认拷贝构造函数与默认赋值操作符如下：

```
#include <cassert>
#include <iostream>

class Fraction
{
private:
    int m_numerator { 0 };
    int m_denominator { 1 };

public:
```

```
    // 默认构造函数
    Fraction(int numerator = 0, int denominator = 1)
        : m_numerator{ numerator }
        , m_denominator{ denominator }
    {
        assert(denominator != 0);
    }

    // 隐式拷贝构造函数可能的实现
    Fraction(const Fraction& f)
        : m_numerator{ f.m_numerator }
        , m_denominator{ f.m_denominator }
    {
    }

    // 隐式赋值操作符可能的实现
    Fraction& operator= (const Fraction& fraction)
    {
        // 自赋值保护
        if (this == &fraction)
            return *this;

        // 进行拷贝
        m_numerator = fraction.m_numerator;
        m_denominator = fraction.m_denominator;

        // 返回已有对象使得可以串联该操作符
        return *this;
    }

    friend std::ostream& operator<<(std::ostream& out, const Fraction& f1)
    {
        out << f1.m_numerator << '/' << f1.m_denominator;
        return out;
    }
};
```

注意因为这些默认版本的对于该类而言恰好有效，因此这种情况下并没有理由去创造一个自定义的版本。

然而当设计的类处理动态分配内存时，成员向（浅）的拷贝则会带来大量的麻烦！这是因为浅拷贝指针仅拷贝了指针的地址 – 它并没有分配任何内存或是拷贝指向的内容！

下面来看一个例子：

```
#include <cstring> // for strlen()
#include <cassert> // for assert()

class MyString
{
```

```
private:
    char* m_data{};
    int m_length{};

public:
    MyString(const char* source = "" )
    {
        assert(source); // 确保 source 并非为空字符串

        // 找到字符串的长度
        // 加一作为终结符
        m_length = std::strlen(source) + 1;

        // 分配一个等长的缓存
        m_data = new char[m_length];

        // 拷贝参数字符串至内部的缓存
        for (int i{ 0 }; i < m_length; ++i)
            m_data[i] = source[i];
    }

    ~MyString() // 析构函数
    {
        // 需要释放字符串
        delete[] m_data;
    }

    char* getString() { return m_data; }
    int getLength() { return m_length; }
};
```

上述简单字符串类分配内存用于存储一个传入的字符串。注意暂未定义拷贝构造函数或是重载赋值操作符。因此 C++ 将提供进行浅拷贝的默认拷贝构造函数以及默认赋值操作符。拷贝构造函数类似于：

```
MyString::MyString(const MyString& source)
    : m_length { source.m_length }
    , m_data { source.m_data }
{
}
```

注意 m_data 是一个对于 source.m_data 的浅指针拷贝，意味着它们同时指向同一个地址。现在考虑以下代码：

```
#include <iostream>

int main()
{
    MyString hello{ "Hello, world!" };
```

```
    {
        MyString copy{ hello }; // 使用默认拷贝构造函数
    } // copy 是一个本地变量，所以在此被销毁。析构函数删除 copy 的字符串，即留下一个悬垂指针

    std::cout << hello.getString() << '\n'; // 这将带来未定义行为

    return 0;
}
```

虽然这段代码看起来无害，但是它包含了一个阴险的问题那就是导致程序出现未定义行为！
让我们逐行拆分该例子：

```
MyString hello{ "Hello, world!" };
```

本行无害。其调用了 MyString 构造函数，分配一些内存，设置 `hello.m_data` 使其指向它，接着拷贝字符串“Hello, world!”给它。

```
MyString copy{ hello }; // 使用默认拷贝构造函数
```

本行看起来也无害，但是这实际上是问题的来源！本行被解析时，C++ 将使用默认的拷贝构造函数（因为用户没有提供）。该拷贝构造函数将做一个浅拷贝，初始化 `copy.m_data` 与 `hello.m_data` 同样的地址。因此，`copy.m_data` 与 `hello.m_data` 指向了同样的内存！

```
} // copy 在此被销毁
```

当 copy 离开作用域，MyString 析构函数在 copy 上被调用。析构函数删除了动态分配的内存，即 `copy.m_data` 与 `hello.m_data` 同时指向的内存！因此，删除了 copy，同样也（不经意的）影响了 hello。变量 copy 被销毁，但是 `hello.m_data` 仍然指向被删除（无效）的内存！

```
std::cout << hello.getString() << '\n'; // 这将导致未定义行为
```

现在可以理解为什么程序有未定义行为了。我们删除了 hello 指向的字符串，并尝试打印已经不存在的内存上的值。
这个问题的源头在于拷贝构造函数所做的浅拷贝 – 在拷贝构造函数或是重载赋值操作符中浅拷贝指针的值几乎就是在自找麻烦。

深拷贝

该问题的一个解决方案是对于任何非空指针进行深拷贝。**深拷贝** deep copy 为拷贝分配内存并拷贝真实的值，这样拷贝存活于独立的内存中。以这样的方式，拷贝与源是独立的且不会互相影响。深拷贝需要用户编写自定义的拷贝构造函数以及重载赋值操作符。

```
// 假设 m_data 已被初始化
void MyString::deepCopy(const MyString& source)
{
    // 首先需要释放该字符串所存储的任何值！
    delete[] m_data;
```

```
6
7       // 因为 m_length 不是一个指针，我们可以浅拷贝它
8       m_length = source.m_length;
9
10      // m_data 是一个指针，所以它非空时需要深拷贝它
11      if (source.m_data)
12      {
13          // 为拷贝分配内存
14          m_data = new char[m_length];
15
16          // 进行拷贝
17          for (int i{ 0 }; i < m_length; ++i)
18              m_data[i] = source.m_data[i];
19      }
20      else
21          m_data = nullptr;
22  }
23
24  // 拷贝构造函数
25  MyString::MyString(const MyString& source)
26  {
27      deepCopy(source);
28  }
```

如所见这比浅拷贝涉及的内容更多！首先，需要确保源拥有字符串（行 11）。如果有则分配足够的内存用于存储字符串的拷贝（行 14）。最后需要手动的拷贝字符串（行 17 与 18）。

现在进行重载赋值操作符，它会有一点棘手：

```
1   // 赋值操作符
2   MyString& MyString::operator=(const MyString& source)
3   {
4       // 检查是否自赋值
5       if (this != &source)
6       {
7           // 现在进行深拷贝
8           deepCopy(source);
9       }
10
11      return *this;
12  }
```

注意赋值操作符与拷贝构造函数类似，但是有三个主要的不同点：

- 添加了一个自赋值检测
- 返回 *this 使得可以串联赋值操作符
- 需要显式的释放任何其存储的字符串（当 m_data 之后被释放时，不会有内存泄漏）。这步在 `deepCopy()` 中完成。

当重载赋值操作符被调用，被赋值的想有可能已包含了之前的值，我们需要确保在赋与新值内存之前清理它。对于非动态分配的变量（即固定大小）而言，我们不需要担心是因为新值将会覆盖旧值。然而对于动态分配的变量而言，我们需要在分配任何新内存之前，显式的释放所有的旧内存。如果不这么做，代码虽然不会崩溃，但是会带来内存泄漏，随即在每次进行分配时吃掉空闲内存。

一个更好的解决方案

标准库的类处理动态内存，例如 `std::string` 以及 `std::vector`，处理了它们所有的内存管理，并且重载了拷贝构造函数与赋值操作符使得拥有合理的深拷贝。因此相较于自定义的内存管理，用户使用它们初始化或复制时便可以像普通的基础变量那样！这使得类简单易用，减少了出错，也不需要用户花费时间在自定义的重载函数上！

总结

- 默认拷贝构造函数以及默认赋值操作符皆为浅拷贝，对于不含动态分配变量的类而言是可行的。
- 拥有动态分配变量的类的拷贝构造函数以及赋值操作符必须要进行深拷贝。
- 优先考虑使用标准库所提供的类而不是用户手动进行内存管理。

14.17 Overloading operators and function templates

在 8.14 函数模版实例化中讨论了编译器是如何使用函数模板来实例化函数并接着进行编译的。同样注意这些函数也可能不会编译，如果函数模版中的代码尝试执行一些真实类型不支持的操作（例如让整数值 `1` 与 `std::string` 相加）。
本章我们将参考几个例子，实例化的函数并不会编译因为真实类类型并不支持这些操作符，并且展示如何定义这些操作符使得实例化的函数能通过编译。

操作符，函数调用，以及函数模板

首先创建一个简单的类：

```
class Cents
{
private:
    int m_cents{};
public:
    Cents(int cents)
        : m_cents { cents }
    {
```

```
    }

    friend std::ostream& operator<< (std::ostream& ostr, const Cents& c)
    {
        ostr << c.m_cents;
        return ostr;
    }
};
```

并定义一个 max 函数模板:

```
template <typename T>
const T& max(const T& x, const T& y)
{
    return (x < y) ? y : x;
}
```

现在观察当尝试调用 max() 在对象类型为 Cents 上:

```
#include <iostream>

class Cents
{
private:
    int m_cents{};
public:
    Cents(int cents)
        : m_cents { cents }
    {
    }

    friend std::ostream& operator<< (std::ostream& ostr, const Cents& c)
    {
        ostr << c.m_cents;
        return ostr;
    }
};

template <typename T>
const T& max(const T& x, const T& y)
{
    return (x < y) ? y : x;
}

int main()
{
    Cents nickel{ 5 };
    Cents dime{ 10 };

    Cents bigger = max(nickel, dime);
```

```
    std::cout << bigger << " is bigger\n";

    return 0;
}
```

C++ 将为 `max()` 创建一个类似如下的模板实例：

```
template <>
const Cents& max(const Cents& x, const Cents& y)
{
    return (x < y) ? y : x;
}
```

接着将尝试编译该函数。看到这里的问题了吗？C++ 不知道当 `x` 与 `y` 是 `Cents` 类型的情况下，如何解析 `x < y`！因此，这会产生一个编译错误。

为了解决这个问题只需要简单的为任何想要使用 `max` 的类重载操作符 `operator<` 即可：

```
#include <iostream>

class Cents
{
private:
    int m_cents {};
public:
    Cents(int cents)
        : m_cents { cents }
    {
    }

    friend bool operator< (const Cents& c1, const Cents& c2)
    {
        return (c1.m_cents < c2.m_cents);
    }

    friend std::ostream& operator<< (std::ostream& ostr, const Cents& c)
    {
        ostr << c.m_cents;
        return ostr;
    }
};

template <typename T>
const T& max(const T& x, const T& y)
{
    return (x < y) ? y : x;
}

int main()
{
    Cents nickel{ 5 };
```

```
    Cents dime { 10 };

    Cents bigger = max(nickel, dime);
    std::cout << bigger << " is bigger\n";

    return 0;
}
```

正如预期，打印：

```
10 is bigger
```

另一个案例

现在看另外一个确实重载操作符而函数模板不能工作的例子。
下列函数模板将计算数组对象的平均值：

```
#include <iostream>

template <typename T>
T average(const T* myArray, int numValues)
{
    T sum { 0 };
    for (int count { 0 }; count < numValues; ++count)
        sum += myArray[count];

    sum /= numValues;
    return sum;
}

int main()
{
    int intArray[] { 5, 3, 2, 1, 4 };
    std::cout << average(intArray, 5) << '\n';

    double doubleArray[] { 3.12, 3.45, 9.23, 6.34 };
    std::cout << average(doubleArray, 4) << '\n';

    return 0;
}
```

打印：

```
3
5.535
```

可以看到内建类型都有效！
现在看一下调用该函数在 `Cents` 类上的表现：

```
#include <iostream>

template <typename T>
T average(const T* myArray, int numValues)
{
    T sum { 0 };
    for (int count { 0 }; count < numValues; ++count)
        sum += myArray[count];

    sum /= numValues;
    return sum;
}

class Cents
{
private:
    int m_cents {};
public:
    Cents(int cents)
        : m_cents { cents }
    {
    }
};

int main()
{
    Cents centsArray[] { Cents { 5 }, Cents { 10 }, Cents { 15 }, Cents { 14 } };
    std::cout << average(centsArray, 4) << '\n';

    return 0;
}
```

编译器变得狂暴并输出了成吨的错误信息！首个错误信息可能类似于这样：

```
error C2679: binary << : no operator found which takes a right-hand operand of type Cents (or
    there is no acceptable conversion)
```

记住 `average()` 返回一个 `Cents` 对象，且我们在尝试使用 `operator<<` stream 该对象至 `std::cout` 。然而我们并未为 `Cents` 类定义 `operator<<` 。现在尝试一下：

```
#include <iostream>

template <typename T>
T average(const T* myArray, int numValues)
{
    T sum { 0 };
    for (int count { 0 }; count < numValues; ++count)
        sum += myArray[count];

    sum /= numValues;
```

```
    return sum;
}

class Cents
{
private:
    int m_cents {};
public:
    Cents(int cents)
        : m_cents { cents }
    {
    }

    friend std::ostream& operator<< (std::ostream& out, const Cents& cents)
    {
        out << cents.m_cents << " cents ";
        return out;
    }
};

int main()
{
    Cents centsArray[] { Cents { 5 }, Cents { 10 }, Cents { 15 }, Cents { 14 } };
    std::cout << average(centsArray, 4) << '\n';

    return 0;
}
```

再次编译则获得另一个错误：

```
error C2676: binary += : Cents does not define this operator or a conversion to a type
    acceptable to the predefined operator
```

这个错误实际上是由函数模板实例创建时调用 `average(const Cents*, int)` 导致的。记住当调用一个模板函数时，编译器会“拓印”一个函数的拷贝，其参数类型（即占位符类型）会被替换为函数调用中的真实类型。这里是当 `T` 为 `Cents` 对象时的 `average()` 函数模板实例：

```
template <>
Cents average(const Cents* myArray, int numValues)
{
    Cents sum { 0 };
    for (int count { 0 }; count < numValues; ++count)
        sum += myArray[count];

    sum /= numValues;
    return sum;
}
```

之所以有报错信息的原因是因为这一行：

```
sum += myArray[count];
```

这种情况下，sum 是一个 Cents 对象，但是我们并未为 Cents 对象定义 operator+=！我们仍将定义该函数使得 average() 可以为 Cents 工作。

```
#include <iostream>

template <typename T>
T average(const T* myArray, int numValues)
{
    T sum { 0 };
    for (int count { 0 }; count < numValues; ++count)
        sum += myArray[count];

    sum /= numValues;
    return sum;
}

class Cents
{
private:
    int m_cents {};
public:
    Cents(int cents)
        : m_cents { cents }
    {
    }

    friend std::ostream& operator<< (std::ostream& out, const Cents& cents)
    {
        out << cents.m_cents << " cents ";
        return out;
    }

    Cents& operator+= (const Cents &cents)
    {
        m_cents += cents.m_cents;
        return *this;
    }

    Cents& operator/= (int x)
    {
        m_cents /= x;
        return *this;
    }
};

int main()
{
    Cents centsArray[] { Cents { 5 }, Cents { 10 }, Cents { 15 }, Cents { 14 } };
    std::cout << average(centsArray, 4) << '\n';
```

```
47
48      return 0;
49  }
```

最终代码通过编译并运行！打印结果：

```
1  11 cents
```

注意我们并未修改 `average()` 使其对类型 `Cents` 生效。我们仅定义了操作符为 `Cents` 类实现 `average()`，编译器为我们做了剩下的事情！

15 Move Semantics and Smart Pointers

15.1 Introduction to smart pointers and move semantics

考虑一个函数动态分配一个值:

```
void someFunction()
{
    Resource* ptr = new Resource(); // Resource 是一个结构体或类

    // 这里对 ptr 进行一些操作

    delete ptr;
}
```

尽管上述代码看起来非常直接，它也非常容易忘记释放 ptr。即使记得在函数末尾 delete ptr，但还是有很多可能导致函数提前退出而 ptr 没有被 delete。例如:

```
#include <iostream>

void someFunction()
{
    Resource* ptr = new Resource();

    int x;
    std::cout << "Enter an integer: ";
    std::cin >> x;

    if (x == 0)
        return; // 函数提前返回，而 ptr 将不被删除!

    // 这里对 ptr 进行一些操作

    delete ptr;
}
```

或是通过一个抛出异常:

```
#include <iostream>

void someFunction()
{
    Resource* ptr = new Resource();

    int x;
    std::cout << "Enter an integer: ";
    std::cin >> x;

    if (x == 0)
        throw 0; // 函数提前退出，而 ptr 将不被删除!
```

```
13
14      // 这里对 ptr 进行一些操作
15
16      delete ptr;
17  }
```

智能指针来拯救

对于类而言其最好的一个地方就在于所包含的析构函数可以自动在该类对象离开作用域时自动执行。因此如果在构造函数中分配（或请求）内存，那么可以在析构函数中释放，并且可以保证在类的对象销毁时（无论是离开作用域还是显式删除，等等）内存能被释放。
那么这可以帮助我们管理以及清理指针吗？是的，可以！

```
1   #include <iostream>
2
3   template <typename T>
4   class Auto_ptr1
5   {
6     T* m_ptr;
7   public:
8     // 通过构造函数传入一个指针
9     Auto_ptr1(T* ptr=nullptr)
10      :m_ptr(ptr)
11    {
12    }
13
14    // 析构函数确保 m_ptr 被释放
15    ~Auto_ptr1()
16    {
17      delete m_ptr;
18    }
19
20    // 重载解引用与操作符->，使得 Auto_ptr1 类似于 m_ptr
21    T& operator*() const { return *m_ptr; }
22    T* operator->() const { return m_ptr; }
23  };
24
25  // 简易例子用于证明上述模板代码有效
26  class Resource
27  {
28  public:
29    Resource() { std::cout << "Resource acquired\n"; }
30    ~Resource() { std::cout << "Resource destroyed\n"; }
31  };
32
33  int main()
34  {
```

```
35   Auto_ptr1<Resource> res(new Resource()); // 注意这里分配内存
36
37   // ... 不需要显式的 delete
38
39   // 同样注意 Resource 在尖括号中不需要 * 符号，因为其功能已被模板所支持
40
41   return 0;
42 } // res 在这里离开作用域，销毁 Resource 并释放内存
```

打印:

```
1 Resource acquired
2 Resource destroyed
```

首先动态创建了一个 Resource，将其作为参数传递给模版化的 Auto_ptr1 类。从这个时候开始，Auto_ptr1 变量 res 拥有了 Resource 对象（Auto_ptr1 与 m_ptr 拥有组合关系）。由于 res 声明为一个本地变量且处于作用域中，当作用域介绍时即离开作用域，随之被销毁（不需要担心忘记释放它）。正因其为一个类，当被销毁时，Auto_ptr1 析构函数被调用，并保证其所持有的 Resource 指针被删除!

只要 Auto_ptr1 被定义为本地变量，那么 Resource 将会被保证在作用域结束后被销毁，无论函数是如何终止的（即便是提前终止）。

这样的类被称为智能指针。**智能指针**是一种组合类设计用来管理动态分配内存以及确保当其对象离开作用域时内存被删除。

现在回到上面的 `someFunction()` 例子，展示智能指针是如何解决的:

```
1  #include <iostream>
2
3  template <typename T>
4  class Auto_ptr1
5  {
6    T* m_ptr;
7  public:
8    Auto_ptr1(T* ptr=nullptr)
9      :m_ptr(ptr)
10   {
11   }
12
13   ~Auto_ptr1()
14   {
15     delete m_ptr;
16   }
17
18   T& operator*() const { return *m_ptr; }
19   T* operator->() const { return m_ptr; }
20 };
21
22 class Resource
```

```
{
public:
  Resource() { std::cout << "Resource acquired\n"; }
  ~Resource() { std::cout << "Resource destroyed\n"; }
  void sayHi() { std::cout << "Hi!\n"; }
};

void someFunction()
{
  Auto_ptr1<Resource> ptr(new Resource()); // ptr 现在拥有 Resource

  int x;
  std::cout << "Enter an integer: ";
  std::cin >> x;

  if (x == 0)
      return; // 函数提前返回

  // ptr 的一些操作
  ptr->sayHi();
}

int main()
{
  someFunction();

  return 0;
}
```

如果用户输入非零整数，打印：

```
Resource acquired
Hi!
Resource destroyed
```

如果用户输入零，打印：

```
Resource acquired
Resource destroyed
```

一个致命缺陷

Auto_ptr1 类拥有一个致命缺陷潜伏在一些自动生成的代码之后。

```
#include <iostream>

// 与之前相同
template <typename T>
class Auto_ptr1
{
```

```
  T* m_ptr;
public:
  Auto_ptr1(T* ptr=nullptr)
    :m_ptr(ptr)
  {
  }

  ~Auto_ptr1()
  {
    delete m_ptr;
  }

  T& operator*() const { return *m_ptr; }
  T* operator->() const { return m_ptr; }
};

class Resource
{
public:
  Resource() { std::cout << "Resource acquired\n"; }
  ~Resource() { std::cout << "Resource destroyed\n"; }
};

int main()
{
  Auto_ptr1<Resource> res1(new Resource());
  Auto_ptr1<Resource> res2(res1); // 作为选择，不初始化 res2 接着赋值 res2 = res1;

  return 0;
}
```

打印:

```
Resource acquired
Resource destroyed
Resource destroyed
```

很有可能程序在这点会崩溃。可以看出问题了吗?因为用户并没有提供一个拷贝构造函数或是赋值操作符，C++ 则帮忙提供了浅拷贝的函数。因此当通过 res1 初始化 res2 时，两个 Auto_ptr1 变量都指向了同一个 Resource。当 res2 离开作用域，资源被删除，使得 res1 成为了悬垂指针。当 res1 要删除（已被删除了）她的 Resource 时，程序崩溃!

运行以下函数也会发生类似的问题:

```
void passByValue(Auto_ptr1<Resource> res)
{
}

int main()
{
```

```
  Auto_ptr1<Resource> res1(new Resource());
  passByValue(res1);

  return 0;
}
```

这个程序中，res1 会被值拷贝进入 passByValue 的参数 res 中，导致了重复的 Resource 指针。崩溃!

那么这并不好，应该怎么进行修复呢?

一种方法是显式定义并删除拷贝构造函数以及赋值操作符，有此来第一时间阻止任何拷贝的发生。

但是该如何从一个函数中返回 Auto_ptr1 给调用者呢?

```
??? generateResource()
{
  Resource* r{ new Resource() };
  return Auto_ptr1(r);
}
```

不可以返回 Auto_ptr1 的引用，因为本地 Auto_ptr1 会在函数结束后销毁，调用者将会得到一个悬垂引用。可以返回一个指针 r 像是 `Resource*` ，但是之后有可能忘记删除它，这也是使用智能指针最好的地方。值返回 Auto_ptr1 作为一种选择是有意义的 – 但是会得到浅拷贝，重复指针，以及崩溃。

另一种选项就是重写拷贝构造函数以及赋值操作符来进行深拷贝。这种方式下至少可以避免了相同对象的重复指针。但是拷贝是昂贵的（可能并不值得或甚至不可能），因此不希望做不必要的拷贝。另外赋值或者初始化一个裸指针并不能拷贝其指向的对象，所以为什么期望智能指针可以有所不同呢?

移动语义

那么如果不是拷贝构造函数以及赋值操作符拷贝指针（拷贝语义），而是从原始对象转移/移动指针的所有权到目标对象呢? 这就是移动语音背后的核心观点。**移动语义** move semantics 意味着类会转移对象的所有权而不是进行拷贝。

```
#include <iostream>

template <typename T>
class Auto_ptr2
{
  T* m_ptr;
public:
  Auto_ptr2(T* ptr=nullptr)
    :m_ptr(ptr)
  {
```

```
  }

  ~Auto_ptr2()
  {
    delete m_ptr;
  }

  // 拷贝构造函数实现移动语义
  Auto_ptr2(Auto_ptr2& a) // 注意：非 const
  {
    m_ptr = a.m_ptr;    // 转移裸指针，从来源到本地对象
    a.m_ptr = nullptr;  // 确保来源不再拥有该指针
  }

  // 赋值操作符实现移动语义
  Auto_ptr2& operator=(Auto_ptr2& a) // 注意：非 const
  {
    if (&a == this)
      return *this;

    delete m_ptr;       // 确保首先释放已有的任何指针
    m_ptr = a.m_ptr;    // 接着转移裸指针，从来源到本地对象
    a.m_ptr = nullptr;  // 确保来源不再拥有该指针
    return *this;
  }

  T& operator*() const { return *m_ptr; }
  T* operator->() const { return m_ptr; }
  bool isNull() const { return m_ptr == nullptr; }
};

class Resource
{
public:
  Resource() { std::cout << "Resource acquired\n"; }
  ~Resource() { std::cout << "Resource destroyed\n"; }
};

int main()
{
  Auto_ptr2<Resource> res1(new Resource());
  Auto_ptr2<Resource> res2; // 开始于 nullptr

  std::cout << "res1 is " << (res1.isNull() ? "null\n" : "not null\n");
  std::cout << "res2 is " << (res2.isNull() ? "null\n" : "not null\n");

  res2 = res1; // res2 假定拥有所有权，res1 设置为 null

  std::cout << "Ownership transferred\n";
```

```
60
61   std::cout << "res1 is " << (res1.isNull() ? "null\n" : "not null\n");
62   std::cout << "res2 is " << (res2.isNull() ? "null\n" : "not null\n");
63
64   return 0;
65 }
```

打印:

```
Resource acquired
res1 is not null
res2 is null
Ownership transferred
res1 is null
res2 is not null
Resource destroyed
```

注意重载的操作符 = 转移了 m_ptr 的所有权，从 res1 到 res2! 因此不再会有重复拷贝的指针，所有东西都变得非常的整洁。

15.2 R-value references

第 9 章中介绍了值分类的概念，即表达式的属性帮助判定一个表达式解析为值，函数，还是对象。同样也介绍了左值与右值。
C++11 添加了被称为右值引用的一种新的引用，其被设计用来初始化右值（仅）。左值引用的创建时通过单个 & 符号，右值引用的创建则是使用双 & 符号:

```
int x{ 5 };
int &lref{ x };   // 左值引用初始化使用左值 x
int &&rref{ 5 };  // 右值引用初始化使用右值 5
```

右值引用不可以通过左值进行初始化。
| R-value reference | Can be initialized with | Can modify | | ———————— | ———————— | ———- | | Modifiable l-values | No | No | | Non-modifiable l-values | No | No | | R-values | Yes | Yes |
| R-value reference to const | Can be initialized with | Can modify | | ———————— | ———————— | ———- | | Modifiable l-values | No | No | | Non-modifiable l-values | No | No | | R-values | Yes | No |
右值引用有两个属性非常有用。首先右值引用延长了对象的寿命，从初始化的寿命延长到了右值引用（左值引用给 const 对象也可以做到）。其次，非 const 右值引用允许用户修改右值!

```
#include <iostream>

class Fraction
{
```

```
private:
  int m_numerator;
  int m_denominator;

public:
  Fraction(int numerator = 0, int denominator = 1) :
    m_numerator{ numerator }, m_denominator{ denominator }
  {
  }

  friend std::ostream& operator<<(std::ostream& out, const Fraction &f1)
  {
    out << f1.m_numerator << '/' << f1.m_denominator;
    return out;
  }
};

int main()
{
  auto &&rref{ Fraction{ 3, 5 } }; // 临时 Fraction 的右值引用

  // f1 的操作符<< 绑定临时值，没有进行拷贝
  std::cout << rref << '\n';

  return 0;
} // rref（以及临时 Fraction）离开作用域
```

打印：

```
3/5
```

作为一个匿名对象，`Fraction{ 3, 5 }` 通常会在其定义的表达式完成时离开作用域。然而因为通过了右值引用来初始化它，其生命周期被延长到了作用域的结束。因此可以使用右值引用来打印 Fraction 的值。

现在来看一个不那么直觉的例子：

```
#include <iostream>

int main()
{
  int &&rref{ 5 }; // 因为对字面值进行右值引用初始化，临时值 5 在这里被创建
  rref = 10;
  std::cout << rref << '\n';

  return 0;
}
```

打印：

```
10
```

虽然通过右值引用来初始化一个字面值接着在修改该值的整个过程看起来很奇怪，但是当右值引用初始化字面值时，一个临时对象从字面值上被构造，因此引用是对临时对象的引用，而不是字面值。
右值引用通常不会像上述展示的方法那样使用。

右值引用作为函数入参

右值引用常用与函数入参。这对于函数重载时，希望左值与右值入参产生不同的行为而言非常的有用。

```
#include <iostream>

void fun(const int &lref) // 左值入参选择该函数
{
  std::cout << "l-value reference to const\n";
}

void fun(int &&rref)      // 右值入参选择该函数
{
  std::cout << "r-value reference\n";
}

int main()
{
  int x{ 5 };
  fun(x); // 左值入参调用左值版本的函数
  fun(5); // 右值入参调用右值版本的函数

  return 0;
}
```

打印:

```
l-value reference to const
r-value reference
```

为什么需要这样呢? 下一节将会进行详细讲解。无需多言，它是移动语义中的一个重要部分。
一个有趣的点:

```
int &&ref{ 5 };
fun(ref);
```

实际上调用的是左值版本的函数! 尽管变量 ref 的类型是*整数的右值引用*，它自身实际上是一个左值（因为是命名的变量）。迷惑的来源来自于使用了右值在两个不同的上下文中。可以这么理解: 命名对象是左值; 匿名对象是右值。那么对于命名对象或是匿名对象的类型而言是独立于其是否为左值或者右值的。或者用另一种方式说，即右值引用被任何东西调用了，这个迷惑变不存在了。

返回一个右值引用

用户应该永远不要返回一个右值引用，同样的理由也永远不要返回一个左值引用。大多数情况下，当引用对象离开函数作用域时，返回得到的总会是悬垂引用。

15.3 Move constructors and move assignment

首先让我们花点时间回顾一下拷贝语义。
拷贝构造函数通过拷贝对象进行类的初始化。拷贝赋值用于拷贝一个类对象至另一个已存在的类对象。默认情况下，如果没有显式提供拷贝构造函数以及拷贝赋值操作符，C++ 将会提供它们。这些编译器提供的函数做浅拷贝，在类进行分配动态内存时可能会导致问题。因此处理动态内存的类需要重写这俩函数成为深拷贝。
回到本章最开始 Auto_ptr 智能指针的例子：

```
#include <iostream>

template<typename T>
class Auto_ptr3
{
  T* m_ptr;
public:
  Auto_ptr3(T* ptr = nullptr)
    :m_ptr(ptr)
  {
  }

  ~Auto_ptr3()
  {
    delete m_ptr;
  }

  // 拷贝构造函数
  // 深拷贝 a.m_ptr 至 m_ptr
  Auto_ptr3(const Auto_ptr3& a)
  {
    m_ptr = new T;
    *m_ptr = *a.m_ptr;
  }

  // 拷贝赋值
  // 深拷贝 a.m_ptr 至 m_ptr
  Auto_ptr3& operator=(const Auto_ptr3& a)
  {
    // 自赋值检测
    if (&a == this)
      return *this;
```

```
33
34      // 释放任何持有的资源
35      delete m_ptr;
36
37      // 拷贝资源
38      m_ptr = new T;
39      *m_ptr = *a.m_ptr;
40
41      return *this;
42    }
43
44    T& operator*() const { return *m_ptr; }
45    T* operator->() const { return m_ptr; }
46    bool isNull() const { return m_ptr == nullptr; }
47  };
48
49  class Resource
50  {
51  public:
52    Resource() { std::cout << "Resource acquired\n"; }
53    ~Resource() { std::cout << "Resource destroyed\n"; }
54  };
55
56  Auto_ptr3<Resource> generateResource()
57  {
58    Auto_ptr3<Resource> res{new Resource};
59    return res; // 这个返回值将唤起拷贝构造函数
60  }
61
62  int main()
63  {
64    Auto_ptr3<Resource> mainres;
65    mainres = generateResource(); // 这个赋值将唤起拷贝赋值
66
67    return 0;
68  }
```

上述代码中，使用了名为 `generateResource()` 的函数创建智能指针封装资源，接着传递给函数 `main()`，接着赋值给已存在的 `Auto_ptr3` 对象。打印：

```
1  Resource acquired
2  Resource acquired
3  Resource destroyed
4  Resource acquired
5  Resource destroyed
6  Resource destroyed
```

很多资源的创建和销毁就是为了一个简单的程序！那么这里发生了什么？
细看一下，有六个重要的步骤发生在该程序中（每个都打印了信息）：

1. 在 `generateResource()` 中本地变量 `res` 被创建并初始化了一个动态分配的 `Resource`，即第一次“Resource acquired”发生。
2. `res` 值返回给 `main()`，因为 `res` 是本地变量 – 由于在 `generateResource()` 结束后它将被销毁，因此不可以返回其地址或者引用。所以 `res` 被拷贝构造进了一个临时对象。因为拷贝构造函数是深拷贝，一个新的 `Resource` 在这里被分配，即第二次“Resource acquired”发生。
3. `res` 离开作用域，销毁原有创造的 `Resource`，即发生第一次“Resource destroyed”。
4. 通过拷贝赋值，临时对象被赋值给了 `mainres`。由于拷贝赋值同样也是一个深拷贝，新的 `Resource` 被分配，即发生第三次“Resource acquired”。
5. 赋值表达式结束，临时对象离开表达式作用域并被销毁，即第二次“Resource destroyed”。
6. 在 `main()` 结束时，`mainres` 离开作用域，即最后一次“Resource destroyed”。

简单来说，因为调用了拷贝构造函数一次对 `res` 进行拷贝构造至临时对象，调用拷贝赋值一次进行拷贝临时对象至 `mainres`，最终就是总共分配与释放了三个独立的对象。
低效，但是至少没有崩溃!
然而，使用移动语义，我们可以做的更好。

移动构造函数与移动赋值

C++11 定义了两个新函数服务于移动语义：移动构造函数，移动赋值操作符。鉴于拷贝构造函数与拷贝赋值的目的是将一个对象拷贝至另一个，移动构造函数与移动赋值则是从一个对象移动资源的所有权至另一个（即通常而言对比拷贝要廉价很多）。
定义一个移动构造函数和移动赋值近似于拷贝语义。不同的是拷贝获取的是 const 左值引用参数，移动获取的是非 const 右值引用参数。

```
#include <iostream>

template<typename T>
class Auto_ptr4
{
  T* m_ptr;
public:
  Auto_ptr4(T* ptr = nullptr)
    :m_ptr(ptr)
  {
  }

  ~Auto_ptr4()
  {
```

```
15    delete m_ptr;
16  }
17
18  // 拷贝构造函数
19  // 深拷贝 a.m_ptr 至 m_ptr
20  Auto_ptr4(const Auto_ptr4& a)
21  {
22    m_ptr = new T;
23    *m_ptr = *a.m_ptr;
24  }
25
26  // 移动构造函数
27  // 转移所有权 a.m_ptr 至 m_ptr
28  Auto_ptr4(Auto_ptr4&& a) noexcept
29    : m_ptr(a.m_ptr)
30  {
31    a.m_ptr = nullptr; // 接下来会讨论这行
32  }
33
34  // 拷贝赋值
35  // 深拷贝 a.m_ptr 至 m_ptr
36  Auto_ptr4& operator=(const Auto_ptr4& a)
37  {
38    // 自赋值检测
39    if (&a == this)
40      return *this;
41
42    // 释放任何持有的资源
43    delete m_ptr;
44
45    // 拷贝资源
46    m_ptr = new T;
47    *m_ptr = *a.m_ptr;
48
49    return *this;
50  }
51
52  // 移动赋值
53  // 转移所有权 a.m_ptr 至 m_ptr
54  Auto_ptr4& operator=(Auto_ptr4&& a) noexcept
55  {
56    // 自赋值检测
57    if (&a == this)
58      return *this;
59
60    // 释放任何持有的资源
61    delete m_ptr;
62
63    // 转移所有权 a.m_ptr 至 m_ptr
```

```
64      m_ptr = a.m_ptr;
65      a.m_ptr = nullptr; // 接下来会讨论这行
66
67      return *this;
68    }
69
70    T& operator*() const { return *m_ptr; }
71    T* operator->() const { return m_ptr; }
72    bool isNull() const { return m_ptr == nullptr; }
73  };
74
75  class Resource
76  {
77  public:
78    Resource() { std::cout << "Resource acquired\n"; }
79    ~Resource() { std::cout << "Resource destroyed\n"; }
80  };
81
82  Auto_ptr4<Resource> generateResource()
83  {
84    Auto_ptr4<Resource> res{new Resource};
85    return res; // 该返回值将唤起移动构造函数
86  }
87
88  int main()
89  {
90    Auto_ptr4<Resource> mainres;
91    mainres = generateResource(); // 该赋值将唤起移动赋值
92
93    return 0;
94  }
```

移动构造函数与移动赋值操作符很简单。相比于深拷贝原始对象，仅需移动原始对象的资源。这里涉及到了浅拷贝原始指针至隐式对象，接着设置原始指针为空。

打印：

```
1  Resource acquired
2  Resource destroyed
```

这好多了！

程序的工作流与之前完全相同。不同点在于不是调用拷贝构造函数与拷贝赋值操作符，这个程序调用的是移动构造函数与移动赋值操作符。深入了解细节：

1. 在 `generateResource()` 中本地变量 `res` 被创建并初始化了一个动态分配的 `Resource`，即“Resource acquired”发生。
2. `res` 值返回给 `main()`，其由移动构造至临时对象，转移存储于 `res` 中动态创建的对象至临时对象。接下来会讲解为何会发生。

3. `res` 离开作用域，由于其不再管理指针（因为被移动至临时对象了），这里没有特别需要注意的地方。
4. 临时对象移动赋值至 `mainres` 。这里转移存储于临时对象中动态创建对象至 `mainres` 。
5. 赋值表达式结束，临时对象离开表达式作用域并被销毁。然而由于临时对象不再管理一个指针（因为被移动至 `mainers` ），这里也没有特别需要注意的 df。
6. 在 `main()` 结束时，`mainres` 离开作用域，即“Resource destroyed”发生。

何时调用移动构造函数与移动赋值

移动构造函数与移动赋值被定义后，只要构造函数的入参或赋值的是右值，它们就会被调用。最典型的右值就是字面值或临时值。
大多数情况下，移动构造函数与移动赋值操作符不会由默认提供，除非是类没有提供任何的拷贝构造函数，拷贝赋值，移动赋值或析构函数。

移动语义背后的重点

如果参数是左值的情况下构造一个对象或者是赋值，能做的唯一事情就只有拷贝左值。我们不能假设改动左值是安全的，因为它有可能在之后的程序中继续被使用。如果有一个表达式 `a = b` ，我们不能合理的预期 b 在任何情况下被改变。
然而如果参数是右值的情况下构造一个对象或者是赋值，可以知道右值是一种临时的对象。不同于拷贝它（即可以是很昂贵），可以简单的转移它的资源（廉价）给正在构造或者赋值的对象。这是安全的因为临时值会在表达式结束时销毁，因此可以知道它永远不会被使用!
C++11 通过右值引用，在参数是右值时，提供了不同的行为，赋予了用户更聪明和更有效的决定对象行为决策的能力。

移动函数应该总是让两个对象处于有效状态

上述例子中，移动构造函数和移动赋值函数都设置了 `a.m_ptr` 为空指针。这似乎没有关联 – 毕竟如果 `a` 是一个临时的右值，`a` 总是会被销毁的，那为什么还需要清理呢?
答案非常的简单：当 `a` 离开作用域，它的析构函数会被调用，`a.m_ptr` 会被删除。如果这个时候 `a.m_ptr` 仍然指向了与 `m_ptr` 相同的对象时，`m_ptr` 将会变成悬垂指针。当对象包含 `m_ptr` 最终被使用（或者销毁），则带来未定义行为。
当实现移动语义时，确保被移动的对象处于有效状态是非常重要的，因此才能合理的析构（不产生未定义行为）。

自动左值的值返回有可能是移动而不是拷贝

在 Auto_ptr4 例子的 generateResource() 函数中，当变量 res 被值返回，它是被移动的而不是拷贝，即使 res 是一个左值。C++ 规范中有一个特殊的规则就是从一个函数里值返回的自动对象可以被移动，即使它们是左值。这很合理，因为 res 怎样都会在函数结算时被销毁！同样也可以窃取其资源而不是使用昂贵且不必要的拷贝。
尽管编译器可以移动左值返回值，一些情况下可能做的更好，即简单的省略所有的拷贝（也就是避免了所有的拷贝或移动）。这种情况下，拷贝构造函数或移动构造函数都不会被调用。

禁用拷贝

在 Auto_ptr4 类中，留下了拷贝构造函数与赋值操作符作为比较的目的。但是在可移动的类中，有时删除它们来确保不会有拷贝更为合适。

```
#include <iostream>

template<typename T>
class Auto_ptr5
{
  T* m_ptr;
public:
  Auto_ptr5(T* ptr = nullptr)
    :m_ptr(ptr)
  {
  }

  ~Auto_ptr5()
  {
    delete m_ptr;
  }

  // 拷贝构造函数 -- 不允许拷贝！
  Auto_ptr5(const Auto_ptr5& a) = delete;

  // 移动构造函数
  // 转移所有权 a.m_ptr 至 m_ptr
  Auto_ptr5(Auto_ptr5&& a) noexcept
    : m_ptr(a.m_ptr)
  {
    a.m_ptr = nullptr;
  }

  // 拷贝赋值 -- 不允许拷贝！
  Auto_ptr5& operator=(const Auto_ptr5& a) = delete;

  // 移动赋值
  // 转移所有权 a.m_ptr 至 m_ptr
```

```
34   Auto_ptr5& operator=(Auto_ptr5&& a) noexcept
35   {
36     // 自赋值检测
37     if (&a == this)
38       return *this;
39
40     // 释放任何持有的资源
41     delete m_ptr;
42
43     // 转移所有权 a.m_ptr 至 m_ptr
44     m_ptr = a.m_ptr;
45     a.m_ptr = nullptr;
46
47     return *this;
48   }
49
50   T& operator*() const { return *m_ptr; }
51   T* operator->() const { return m_ptr; }
52   bool isNull() const { return m_ptr == nullptr; }
53 };
```

`Auto_ptr5` （最终）是一个非常好的智能指针。实际上标准库包含了一个类似的类，其名为 `std::unique_ptr` （也是用户该使用的，下一节将会覆盖）。

另一个例子

现在来看另一个使用动态内存的例子：一个简单的动态模板数组。该类包含了一个深拷贝构造函数与拷贝赋值操作符。

```
1  #include <iostream>
2
3  template <typename T>
4  class DynamicArray
5  {
6  private:
7    T* m_array;
8    int m_length;
9
10 public:
11   DynamicArray(int length)
12     : m_array(new T[length]), m_length(length)
13   {
14   }
15
16   ~DynamicArray()
17   {
18     delete[] m_array;
19   }
```

```
20
21    // Copy constructor
22    DynamicArray(const DynamicArray &arr)
23      : m_length(arr.m_length)
24    {
25      m_array = new T[m_length];
26      for (int i = 0; i < m_length; ++i)
27        m_array[i] = arr.m_array[i];
28    }
29
30    // Copy assignment
31    DynamicArray& operator=(const DynamicArray &arr)
32    {
33      if (&arr == this)
34        return *this;
35
36      delete[] m_array;
37
38      m_length = arr.m_length;
39      m_array = new T[m_length];
40
41      for (int i = 0; i < m_length; ++i)
42        m_array[i] = arr.m_array[i];
43
44      return *this;
45    }
46
47    int getLength() const { return m_length; }
48    T& operator[](int index) { return m_array[index]; }
49    const T& operator[](int index) const { return m_array[index]; }
50
51  };
```

接下来在程序中使用该类，展示该类在堆上分配一百万整型的数据时候的性能。最终可以看到拷贝与移动之间性能的差异。

```
1  #include <iostream>
2  #include <chrono> // std::chrono 函数
3
4  // 使用上面的 DynamicArray 类
5
6  class Timer
7  {
8  private:
9    // 类型别名
10   using Clock = std::chrono::high_resolution_clock;
11   using Second = std::chrono::duration<double, std::ratio<1> >;
12
13   std::chrono::time_point<Clock> m_beg { Clock::now() };
```

```
14
15 public:
16   void reset()
17   {
18     m_beg = Clock::now();
19   }
20
21   double elapsed() const
22   {
23     return std::chrono::duration_cast<Second>(Clock::now() - m_beg).count();
24   }
25 };
26
27 // 返回 arr 的拷贝，其中所有元素乘以二
28 DynamicArray<int> cloneArrayAndDouble(const DynamicArray<int> &arr)
29 {
30   DynamicArray<int> dbl(arr.getLength());
31   for (int i = 0; i < arr.getLength(); ++i)
32     dbl[i] = arr[i] * 2;
33
34   return dbl;
35 }
36
37 int main()
38 {
39   Timer t;
40
41   DynamicArray<int> arr(1000000);
42
43   for (int i = 0; i < arr.getLength(); i++)
44     arr[i] = i;
45
46   arr = cloneArrayAndDouble(arr);
47
48   std::cout << t.elapsed();
49 }
```

作者的电脑花费 0.00825559 秒。

现在来看一下使用移动构造函数与移动赋值版本的例子。

```
1 template <typename T>
2 class DynamicArray
3 {
4 private:
5   T* m_array;
6   int m_length;
7
8 public:
9   DynamicArray(int length)
```

```
    : m_array(new T[length]), m_length(length)
  {
  }

  ~DynamicArray()
  {
    delete[] m_array;
  }

  // Copy constructor
  DynamicArray(const DynamicArray &arr) = delete;

  // Copy assignment
  DynamicArray& operator=(const DynamicArray &arr) = delete;

  // Move constructor
  DynamicArray(DynamicArray &&arr) noexcept
    :  m_array(arr.m_array), m_length(arr.m_length)
  {
    arr.m_length = 0;
    arr.m_array = nullptr;
  }

  // Move assignment
  DynamicArray& operator=(DynamicArray &&arr) noexcept
  {
    if (&arr == this)
      return *this;

    delete[] m_array;

    m_length = arr.m_length;
    m_array = arr.m_array;
    arr.m_length = 0;
    arr.m_array = nullptr;

    return *this;
  }

  int getLength() const { return m_length; }
  T& operator[](int index) { return m_array[index]; }
  const T& operator[](int index) const { return m_array[index]; }

};

#include <iostream>
#include <chrono> // std::chrono 函数

class Timer
```

```
{
private:
  using Clock = std::chrono::high_resolution_clock;
  using Second = std::chrono::duration<double, std::ratio<1> >;

  std::chrono::time_point<Clock> m_beg { Clock::now() };

public:
  void reset()
  {
    m_beg = Clock::now();
  }

  double elapsed() const
  {
    return std::chrono::duration_cast<Second>(Clock::now() - m_beg).count();
  }
};

DynamicArray<int> cloneArrayAndDouble(const DynamicArray<int> &arr)
{
  DynamicArray<int> dbl(arr.getLength());
  for (int i = 0; i < arr.getLength(); ++i)
    dbl[i] = arr[i] * 2;

  return dbl;
}

int main()
{
  Timer t;

  DynamicArray<int> arr(1000000);

  for (int i = 0; i < arr.getLength(); i++)
    arr[i] = i;

  arr = cloneArrayAndDouble(arr);

  std::cout << t.elapsed();
}
```

作者的电脑花费 0.0056 秒。也就是说移动语义版本快了 47.4% 之多！

不要通过 std::swap 来实现移动语义

由于移动语义的目标是从一个原始对象移动资源到目标对象上，有人就会想着用 `std::swap` 来实现移动构造函数与移动赋值操作符。然而这是一个坏主意，因为 `std::swap` 调用了双方

的移动构造函数与移动赋值在可移动的对象上，这便会造成无限递归。以下例子：

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

class Name
{
private:
  std::string m_name; // std::string 是可移动的

public:
  Name(std::string name) : m_name{ name }
  {
  }

  Name(Name& name) = delete;
  Name& operator=(Name& name) = delete;

  Name(Name&& name)
  {
      std::cout << "Move ctor\n";

      std::swap(*this, name); // 坏!
  }

  Name& operator=(Name&& name)
  {
      std::cout << "Move assign\n";

      std::swap(*this, name); // 坏!

      return *this;
  }

  const std::string& get() { return m_name; }
};

int main()
{
  Name n1{ "Alex" };
  n1 = Name{"Joe"}; // 唤起移动赋值

  std::cout << n1.get() << '\n';

  return 0;
}
```

打印：

```
Move assign
```

```
Move ctor
Move ctor
Move ctor
Move ctor
...
```

无限循环直到栈溢出。

可以使用自定义的交换函数，只要自定义的交换成员函数不去调用移动构造函数或移动赋值。例如：

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

class Name
{
private:
  std::string m_name;

public:
  Name(std::string name) : m_name{ name }
  {
  }

  Name(Name& name) = delete;
  Name& operator=(Name& name) = delete;

  // 创建自定义的友元 swap 函数用于交换成员 Name
  friend void swap(Name& a, Name& b) noexcept
  {
      // 避免在 std::string 成员上递归调用 std::swap
      std::swap(a.m_name, b.m_name);
  }

  Name(Name&& name)
  {
      std::cout << "Move ctor\n";

      swap(*this, name); // 调用自身的 swap，而不是 std::swap
  }

  Name& operator=(Name&& name)
  {
      std::cout << "Move assign\n";

      swap(*this, name); // 调用自身的 swap，而不是 std::swap

      return *this;
  }
};
```

```
40
41 int main()
42 {
43   Name n1{ "Alex" };
44   n1 = Name{"Joe"}; // 唤起移动赋值
45
46   std::cout << n1.get() << '\n';
47
48   return 0;
49 }
```

与预期一致，打印:

```
1 Move assign
2 Joe
```

15.4 std::move

一旦开始普遍的使用移动语义，会发现有时希望唤起移动语义，但是处理的对象却是左值而不是右值。考虑下面代码:

```
1  #include <iostream>
2  #include <string>
3
4  template<class T>
5  void myswapCopy(T& a, T& b)
6  {
7    T tmp { a };  // 唤起拷贝构造函数
8    a = b;        // 唤起拷贝赋值
9    b = tmp;      // 唤起拷贝赋值
10 }
11
12 int main()
13 {
14   std::string x{ "abc" };
15   std::string y{ "de" };
16
17   std::cout << "x: " << x << '\n';
18   std::cout << "y: " << y << '\n';
19
20   myswapCopy(x, y);
21
22   std::cout << "x: " << x << '\n';
23   std::cout << "y: " << y << '\n';
24
25   return 0;
26 }
```

打印:

```
x: abc
y: de
x: de
y: abc
```

上一节讲到过拷贝性能低，而这个版本的 swap 做了三次拷贝。

C++11 中 `std::move` 作为一个标准库函数转型（使用 `static_cast`）它的参数成为右值引用，因此移动语义可以被唤起。所以可以使用 `std::move` 来转换左值成为一个想要被移动而不是拷贝的类型，`std::move` 定义于 `<utility>` 头文件。

以下是使用了 `std::move` 转换左值成为右值并唤起移动语义的例子：

```cpp
#include <iostream>
#include <string>
#include <utility> // std::move

template<class T>
void myswapMove(T& a, T& b)
{
  T tmp { std::move(a) }; // 唤起移动构造函数
  a = std::move(b);       // 唤起移动赋值
  b = std::move(tmp);     // 唤起移动赋值
}

int main()
{
  std::string x{ "abc" };
  std::string y{ "de" };

  std::cout << "x: " << x << '\n';
  std::cout << "y: " << y << '\n';

  myswapMove(x, y);

  std::cout << "x: " << x << '\n';
  std::cout << "y: " << y << '\n';

  return 0;
}
```

打印：

```
x: abc
y: de
x: de
y: abc
```

另一个例子

在填充容器的元素时，可以使用 `std::move` ，例如通过左值填充 `std::vector` 。
下列的例子中，首先通过拷贝语义添加一个元素给向量。接着通过移动语义再添加一个元素。

```cpp
#include <iostream>
#include <string>
#include <utility> // std::move
#include <vector>

int main()
{
  std::vector<std::string> v;

  // 这里使用 std::string，因为它是可移动的（std::string_view 不可移动）
  std::string str { "Knock" };

  std::cout << "Copying str\n";
  v.push_back(str); // 调用左值版本的 push_back，即拷贝 str 至数组元素

  std::cout << "str: " << str << '\n';
  std::cout << "vector: " << v[0] << '\n';

  std::cout << "\nMoving str\n";

  v.push_back(std::move(str)); // 调用右值版本的 push_back，即移动 str 至数组元素

  std::cout << "str: " << str << '\n'; // 这个结果是难以确定的
  std::cout << "vector:" << v[0] << ' ' << v[1] << '\n';

  return 0;
}
```

可能打印：

```
Copying str
str: Knock
vector: Knock

Moving str
str:
vector: Knock Knock
```

第一种情况中，传递给 `push_back()` 的是一个左值，因此它使用了拷贝语义来添加元素至向量。也正是这个原因，str 仍然还在。
第二种情况中，传递给 `push_back()` 的是一个右值（事实上是左值通过 `std::move` 转换的），因此它使用了移动语义来添加元素至向量。这更高效，因为向量元素可以窃取字符串的值而不是拷贝值。

被移动后的对象将变为无效，不过也可能处于不确定的状态

当从一个临时对象中移动值，无需在意被移动的对象会是什么样子，因为临时对象会被立刻销毁。但是如果是使用了 `std::move` 之后的左值对象呢?
这里有两派的学说。第一种学说认为被移动后的对象应该被重置成为默认/零值状态，该对象再也不拥有任何资源了；另一种学说认为应该怎么简单怎么来，在不便利的情况下，不需要限制清除移动后的对象。
那么标准库是怎么处理的呢? 关于这个问题 C++ 标准说"除非另外指定，被移动后的对象（其类型是定义于 C++ 标准库中的）应该放置在一个有效但是未指明的状态"。
本章的例子中，在调用 `std::move` 之后再打印 `str` 的值，得出的是空字符串。然而这不是必须得，它可以打印任何有效的字符串，包括空字符串、原始字符串、或者其他有效字符串。因此，用户应该避免使用被移动过后的对象，因为其结果是根据实现制定的。
关键点: `std::move()` 提示编译器程序员不再需要改值的对象了。仅对持久对象使用 `std::move()`，并且不要对移动后对象的值做任何的假设。当值被移动后，该对象再次被赋值时可以的（使用操作符 =）。

std::move() 还在那里有用?

当排列数组元素时，`std::move()` 同样也非常的有用。很多排序算法（例如选择排序和冒泡排序）都是由元素之前进行交换完成的。
同样有用的地方在于从一个智能指针中移动内容至另一个智能指针。

15.5 std::move_if_noexcept

20.9 异常规范与 noexcept 中覆盖了 `noexcept` 异常说明符与操作符，即本篇的基础。
同样也覆盖了 `strong exception guarantee`，其用于保证一个函数若是被异常打断，不会有内存泄漏同时程序状态也不会改变。特别是所有的构造函数都应该是强异常保证的，因此程序的状态不会在对象的构造函数失败后有所改变。

移动构造函数异常问题

在拷贝对象的情况下，拷贝如果由于某些原因（例如机器内存不足）失败了，被拷贝的对象不会有任何的危害，因为原始对象在创建拷贝时并不需要被改变。这样程序可以弃置失败的拷贝并继续执行。`strong exception guarantee` 是被支持的。
现在考虑移动对象的情况。一个移动的操作转移了给定对象的所有权至目标对象。如果移动操作在转移所有权之后被异常打断，那么原始对象会处于一个被修改状态。这对于原始对象是临时对象而言并不是问题，因为在其被移动之后总是会被抛弃 – 但是对于非临时对象而言，这就

破坏了原始对象。为了遵从 strong exception guarantee，我们需要移动资源回去给原始对象，但是如果第一次移动就失败了，就不能保证移动回去还能成功。
那么如何能给予移动构造函数 strong exception guarantee 呢？在移动构造函数中避免抛出异常很容易，但是其自身有可能唤起其他的构造函数是 potentially throwing 的。以 std::pair 作为移动构造函数为例，需要移动每个字对象进入新的 pair 对象。

```
// Example move constructor definition for std::pair
// Take in an 'old' pair, and then move construct the new pair's 'first' and 'second'
    subobjects from the 'old' ones
template <typename T1, typename T2>
pair<T1,T2>::pair(pair&& old)
  : first(std::move(old.first)),
    second(std::move(old.second))
{}
```

那么现在使用两个类，MoveClass 与 CopyClass，将它们 pair 在一起来证明移动构造函数的 strong exception guarantee 问题：

```
#include <iostream>
#include <utility> // std::pair, std::make_pair, std::move, std::move_if_noexcept
#include <stdexcept> // std::runtime_error

class MoveClass
{
private:
  int* m_resource{};

public:
  MoveClass() = default;

  MoveClass(int resource)
    : m_resource{ new int{ resource } }
  {}

  // 拷贝构造函数
  MoveClass(const MoveClass& that)
  {
    // 深拷贝
    if (that.m_resource != nullptr)
    {
      m_resource = new int{ *that.m_resource };
    }
  }

  // 移动构造函数
  MoveClass(MoveClass&& that) noexcept
    : m_resource{ that.m_resource }
  {
```

```
    that.m_resource = nullptr;
  }

  ~MoveClass()
  {
    std::cout << "destroying " << *this << '\n';

    delete m_resource;
  }

  friend std::ostream& operator<<(std::ostream& out, const MoveClass& moveClass)
  {
    out << "MoveClass(";

    if (moveClass.m_resource == nullptr)
    {
      out << "empty";
    }
    else
    {
      out << *moveClass.m_resource;
    }

    out << ')';

    return out;
  }
};


class CopyClass
{
public:
  bool m_throw{};

  CopyClass() = default;

  // 拷贝构造函数在 m_throw 为 'true' 时抛出异常
  CopyClass(const CopyClass& that)
    : m_throw{ that.m_throw }
  {
    if (m_throw)
    {
      throw std::runtime_error{ "abort!" };
    }
  }
};

int main()
```

```
{
  // 创建一个没有任何问题的 std::pair
  std::pair my_pair{ MoveClass{ 13 }, CopyClass{} };

  std::cout << "my_pair.first: " << my_pair.first << '\n';

  // 但是当移动 pair 至另一个 pair 时，问题出现了
  try
  {
    my_pair.second.m_throw = true; // 触发拷贝构造函数的异常

    // 下一行将抛出异常
    std::pair moved_pair{ std::move(my_pair) }; // 之后将注释掉这一行
    // std::pair moved_pair{ std::move_if_noexcept(my_pair) }; // 之后将反注释这一行

    std::cout << "moved pair exists\n"; // 永不会打印
  }
  catch (const std::exception& ex)
  {
      std::cerr << "Error found: " << ex.what() << '\n';
  }

  std::cout << "my_pair.first: " << my_pair.first << '\n';

  return 0;
}
```

打印:

```
destroying MoveClass(empty)
my_pair.first: MoveClass(13)
destroying MoveClass(13)
Error found: abort!
my_pair.first: MoveClass(empty)
destroying MoveClass(empty)
```

现在来探讨一下具体发生了些什么。

第一个打印出的行展示了用于初始化 `my_pair` 的临时 `MoveClass` 对象，其在 `my_pair` 实例化声明被执行之后立刻被销毁。它为 `empty` 是因为在 `my_pair` 中的 `MoveClass` 子对象是由移动构造的，在下一行的打印中证明了 `my_pair.first` 包含了 `MoveClass` 对象的值 `13` 。

有趣的事情发生在第三行。通过拷贝构造函数创建 `moved_pair` 的 `CopyClass` 子对象（它没有移动构造函数），但是拷贝构造函数抛出了异常，因为修改了 Boolean 标记。`moved_pair` 的构造由于异常的原因失败了，同时其早前构造的成员函数被删除。这个情况下 `MoveClass` 成员被销毁，打印 `destroying MoveClass(13)` ，接着就看到了 `Error found: abort!` 信息被打印出来。

当尝试再次打印 `my_pair.first` 时，显示出 `MoveClass` 成员为空。因为 `moved_pair` 的初始化使用过 `std::move` 完成的，而 `MoveClass` 成员（拥有移动构造函数）被移动构造了且 `my_pair.first` 变为空值。
最后 `my_pair` 在 `main()` 结束前销毁。

std::move_if_noexcept 来拯救

注意上述的问题本来是可以避免的，如果 `std::pair` 尝试的是拷贝而不是移动，这种情况下 `moved_pair` 将会构造失败，但是 `my_pair` 并不会被修改。
但是拷贝有性能成本，因此不想为所有对象都付出这种代价 – 如果安全的话，理想的情况是使用移动，其次才是拷贝。
幸运的是 C++ 有两种机制，可以组合在一起使用。
首先 `noexcept` 函数是 no-throw/no-fail 的，它们是隐式符合 `strong exception guarantee` 标准。因此 `noexcept` 移动构造函数是被保证可以成功的。
其次是使用标准库的函数 `std::move_if_noexcept` 在移动构造函数中（或者对象时仅可移动且没有拷贝构造函数时）不会抛出异常，那么 `std::move_if_noexcept` 将与 `std::move` 完全相同（且返回被转换成右值的对象）。否则 `std::move_if_noexcept` 将返回对象的一个普通的左值引用。
关键点：如果对象拥有一个 noexcept 移动构造函数，那么 `std::move_if_noexcept` 将返回一个可移动的右值，否则将返回可拷贝的左值。可以使用 `noexcept` 说明符作为连接 `std::move_if_noexcept` ，在当强异常保证存在时，使用移动语义（否则使用拷贝语义）。
现在更新之前例子中的代码：

```
//std::pair moved_pair{std::move(my_pair)}; // 现在注释掉这一行
std::pair moved_pair{std::move_if_noexcept(my_pair)}; // 并反注释这一行
```

现在运行程序再打印：

```
destroying MoveClass(empty)
my_pair.first: MoveClass(13)
destroying MoveClass(13)
Error found: abort!
my_pair.first: MoveClass(13)
destroying MoveClass(13)
```

可以看到在异常抛出之后，子对象 `my_pair.first` 仍然指向值 `13` 。
`std::pair` 的移动构造函数不是 `noexcept` ，因此 `std::move_if_noexcept` 返回 `my_pair` 为一个左值引用。这导致 `moved_pair` 通过拷贝构造函数（而不是移动构造函数）进行创建。拷贝构造函数可以安全的抛出异常，因为它不会修改原始对象。
标准库进出使用 `std::move_if_noexcept` 来优化 `noexcept` 函数。例如 `std::vector::resize` 将使用移动语义如果元素的类型是 `noexcept` 移动构造函数，否则使用拷贝语义。这

就意味着 `std::vector` 通常对于拥有 `noexcept` 移动构造函数的对象来说，其操作会更快。警告：如果一个类型同时是潜在抛出异常的移动语义，以及删除了拷贝语义（拷贝构造函数和拷贝赋值操作符不可用），那么 `std::move_if_noexcept` 则会放弃强保证同时唤起移动语义。这种条件性的放弃强保证是普遍存在于标准库的容器类中的，因为它们时常使用 `std::move_if_noexcept` 。

15.6 std::unique_ptr

本章开始时讨论了指针在某些情况下是如何导致 bugs 与内存泄漏的。例如当一个函数提前退出或抛出异常，都会使得指针不能被正确的删除。

```
#include <iostream>

void someFunction()
{
    auto* ptr{ new Resource() };

    int x{};
    std::cout << "Enter an integer: ";
    std::cin >> x;

    if (x == 0)
        throw 0; // 函数提前返回，ptr 将不被删除

    // 这里是 ptr 的一些操作

    delete ptr;
}
```

现在我们已经覆盖了移动语义的基础，可以回到智能指针类的主题了。作为提醒，一个智能指针是一个管理动态分配对象的类。尽管智能指针可以提供一些额外功能，其定义的特性就是管理动态分配的资源，并确保动态分配的对象可以在合适的时间（通常是智能指针离开作用域时）被正确的清理掉。

正因如此，智能指针应该永远不动态分配它们自身（否则就要承担无法正确释放的风险，意味着其所持的对象不能被释放，从而导致内存泄漏）。总是将分配智能指针在栈上（作为本地变量或者类的组合成员），当函数或者包含其的对象结束时，保证智能指针可以正确的离开作用域，以确保其所拥有的对象能被正确的释放。

C++11 的标准库提供了 4 种智能指针类：`std::auto_ptr` （在 C++17 中移除）， `std::unique_ptr` ， `std::shared_ptr` 以及 `std::weak_ptr` 。其中 `std::unique_ptr` 是现今使用的最广泛的智能指针类，因此我们首先学习它。之后的章节中再覆盖 `std::shared_ptr` 以及 `std::weak_ptr` 。

`std::unique_ptr` 是 C++11 用来替代 `std::auto_ptr` 的。它用于管理任何动态分配的且

不被多个对象共享的对象。也就是说 `std::unique_ptr` 应该完全管理其持有的对象，而不是与其他类共享所有权。`std::unique_ptr` 定义在 `<memory>` 头文件。

```
#include <iostream>
#include <memory> // std::unique_ptr

class Resource
{
public:
  Resource() { std::cout << "Resource acquired\n"; }
  ~Resource() { std::cout << "Resource destroyed\n"; }
};

int main()
{
  // 分配 Resource 对象，并让其被 std::unique_ptr 所有
  std::unique_ptr<Resource> res{ new Resource() };

  return 0;
} // res 离开作用域，被分配的 Resource 被销毁
```

因为 `std::unique_ptr` 分配在栈上，它可以确保最终离开作用域，当其离开时，它将删除其管理的 Resource。

不同于 `std::auto_ptr` ，`std::unique_ptr` 正确的实现了移动语义。

```
#include <iostream>
#include <memory> // std::unique_ptr
#include <utility> // std::move

class Resource
{
public:
  Resource() { std::cout << "Resource acquired\n"; }
  ~Resource() { std::cout << "Resource destroyed\n"; }
};

int main()
{
  std::unique_ptr<Resource> res1{ new Resource{} }; // Resource 在这里被创建
  std::unique_ptr<Resource> res2{}; // 开始为空指针

  std::cout << "res1 is " << (res1 ? "not null\n" : "null\n");
  std::cout << "res2 is " << (res2 ? "not null\n" : "null\n");

  // res2 = res1; // 不能编译：拷贝赋值被禁用了
  res2 = std::move(res1); // res2 行使所有权，res1 被设为空值

  std::cout << "Ownership transferred\n";

```

```
  std::cout << "res1 is " << (res1 ? "not null\n" : "null\n");
  std::cout << "res2 is " << (res2 ? "not null\n" : "null\n");

  return 0;
} // 当 res2 离开作用域，Resource 在此处被销毁
```

打印：

```
Resource acquired
res1 is not null
res2 is null
Ownership transferred
res1 is null
res2 is not null
Resource destroyed
```

访问被管理的对象

`std::unique_ptr` 拥有一个重载的操作符 * 与操作符->，可用于返回被管理的资源。操作符 * 返回被管理资源的引用，操作符-> 返回一个指针。

牢记 `std::unique_ptr` 不总是管理一个对象 – 有可能是因为它在被创建的时候就是空的（使用默认构造函数或者传递空指针作为参数），也有可能是其管理的资源被移动到了另一个 `std::unique_ptr` 。所以在使用这些操作符之前，应该检查 `std::unique_ptr` 是否真正拥有资源。幸运的是，`std::unique_ptr` 有一个隐式转型成 bool 值的方法，即其在管理资源时返回 true。

```
#include <iostream>
#include <memory> // std::unique_ptr

class Resource
{
public:
  Resource() { std::cout << "Resource acquired\n"; }
  ~Resource() { std::cout << "Resource destroyed\n"; }
  friend std::ostream& operator<<(std::ostream& out, const Resource &res)
  {
  out << "I am a resource";
  return out;
  }
};

int main()
{
  std::unique_ptr<Resource> res{ new Resource{} };

  if (res) // 使用隐式转型成为 bool 来确保 res 包含一个 Resource
    std::cout << *res << '\n'; // 打印 res 所有的 Resource
```

```
22
23   return 0;
24 }
```

```
1 Resource acquired
2 I am a resource
3 Resource destroyed
```

std::unique_ptr 与数组

不同于 `std::auto_ptr`，`std::unique_ptr` 聪明的知道是否使用标量删除还是数组删除，因此 `std::unique_ptr` 可以同时使用标量对象与数组。
然而相比于使用 `std::unique_ptr` 加上固定的数组，动态数组或是 C-style 字符串，使用 `std::array` 或 `std::vector`（或 `std::string`）会是一个更优的选项。
最佳实践：推荐直接使用 `std::array`，`std::vector` 或 `std::string`，而不是 `std::unique_ptr` 加固定数组，动态数组或 C-style 字符串。

std::make_unique

C++14 增加了一个额外名为 `std::make_unique` 的函数。该模板函数构造一个模板类型的对象并通过传入函数的参数初始化该对象。

```
1  #include <memory> // std::unique_ptr 与 std::make_unique
2  #include <iostream>
3
4  class Fraction
5  {
6  private:
7    int m_numerator{ 0 };
8    int m_denominator{ 1 };
9
10 public:
11   Fraction(int numerator = 0, int denominator = 1) :
12     m_numerator{ numerator }, m_denominator{ denominator }
13   {
14   }
15
16   friend std::ostream& operator<<(std::ostream& out, const Fraction &f1)
17   {
18     out << f1.m_numerator << '/' << f1.m_denominator;
19     return out;
20   }
21 };
22
23
```

```
int main()
{
  // 创建单个动态分配的 Fraction，可以使用自动类型推导
  auto f1{ std::make_unique<Fraction>(3, 5) };
  std::cout << *f1 << '\n';

  // 创建一个动态分配数组的 Fraction
  auto f2{ std::make_unique<Fraction[]>(4) };
  std::cout << f2[0] << '\n';

  return 0;
}
```

打印:

```
3/5
0/1
```

`std::make_unique` 的使用是可选的，但是比起直接创建 `std::unique_ptr` 更推荐使用它。这是因为 `std::make_unique` 的使用更加简单，同样只需要更少的类型（使用自动类型推导）。更重要的是，它解析了异常安全的问题，即 C++ 函数入参的顺序问题。
最佳实践：使用 `std::make_unique` 而不是直接通过 new 来创建 `std::unique_ptr` 。

异常安全问题的更多细节

对于上述的“异常安全问题”，这里是一个问题的描述。
考虑以下这个表达式:

```
some_function(std::unique_ptr<T>(new T), function_that_can_throw_exception());
```

编译器给予了很大的灵活性，按照如何处理这个调用。它可以创建一个新的 `T` ，接着调用 `function_that_can_throw_exception()` ，然后创建 `std::unique_ptr` 管理动态分配的 `T` 。如果 `function_that_can_throw_exception()` 抛出了异常，那么之前分配的 `T` 将不会被释放，因为作用释放的智能指针并没有被创建。这就导致了内存泄漏。
`std::make_unique` 并不会收到上述问题影响，因为对象 `T` 的创建以及 `std::unique_ptr` 的创建，发生在 `std::make_unique` 函数内部，这就没有了执行顺序的问题。

从一个函数中返回 std::unique_ptr

`std::make_unique` 可以安全的被一个函数返回值:

```
#include <memory> // std::unique_ptr

std::unique_ptr<Resource> createResource()
{
    return std::make_unique<Resource>();
```

```
}

int main()
{
    auto ptr{ createResource() };

    // 做些事

    return 0;
}
```

上述代码中，`createResource()` 值返回一个 `std::unique_ptr` 。如果该值没有被赋予给任何变量，那么临时返回值将离开作用域，之后 `Resource` 将被清理。如果被赋值了（如上的 `main()` 中），C++14 或更早前，移动语义将会转移返回值的 `Resource` 到被赋值的变量（上述例子为 `ptr` ），而 C++17 或之后，返回会被省略。这就使得通过 `std::unique_ptr` 返回一个资源比起返回裸指针更为安全!
通常而言，用户不应该返回指针（也永远不要）或者引用（除非有特殊的编译原因）的 `std::unique_ptr` 。

传递 std::unique_ptr 至函数

如果希望函数获取指针内容的所有权，值传递 `std::unique_ptr` 。注意因为拷贝语义被禁用了，需要使用 `std::move` 来传递值。

```
#include <iostream>
#include <memory> // std::unique_ptr
#include <utility> // std::move

class Resource
{
public:
  Resource() { std::cout << "Resource acquired\n"; }
  ~Resource() { std::cout << "Resource destroyed\n"; }
  friend std::ostream& operator<<(std::ostream& out, const Resource &res)
  {
    out << "I am a resource";
    return out;
  }
};

void takeOwnership(std::unique_ptr<Resource> res)
{
  if (res)
    std::cout << *res << '\n';
} // Resource 在此销毁

```

```
int main()
{
  auto ptr{ std::make_unique<Resource>() };

  // takeOwnership(ptr); // 无效，需要使用移动语义
  takeOwnership(std::move(ptr)); // 有效：使用了移动语义

  std::cout << "Ending program\n";

  return 0;
}
```

打印：

```
Resource acquired
I am a resource
Resource destroyed
Ending program
```

注意这个例子中，Resource 的所有权被 takeOwnership() 转移了，因此 Resource 在 takeOwnership() 的结尾被销毁，而不是在 main() 的结尾销毁。

然而大多数时候并不希望函数获取资源的所有权。尽管可以传递 std::unique_ptr 的引用（即允许函数使用对象而不行使所有权），但是应该仅在调用函数可能会修改被管理的对象时才这样使用。

相对而言，传递资源自身（通过指针或引用，取决于 null 是否为有效参数）更好一些。这允许函数保留对调用者如何管理资源的不可知性。从 std::unique_ptr 获取一个裸指针，可以使用 get() 成员函数：

```
#include <memory> // std::unique_ptr
#include <iostream>

class Resource
{
public:
  Resource() { std::cout << "Resource acquired\n"; }
  ~Resource() { std::cout << "Resource destroyed\n"; }

  friend std::ostream& operator<<(std::ostream& out, const Resource &res)
  {
    out << "I am a resource";
    return out;
  }
};

// 函数仅使用 resource，所以接受 resource 的指针，而不是整个 std::unique_ptr<Resource> 的引用
void useResource(Resource* res)
{
  if (res)
```

```
    std::cout << *res << '\n';
  else
    std::cout << "No resource\n";
}

int main()
{
  auto ptr{ std::make_unique<Resource>() };

  useResource(ptr.get()); // 注意：这里使用了 get() 来获取 Resource 的指针

  std::cout << "Ending program\n";

  return 0;
} // Resource 在此销毁
```

打印：

```
Resource acquired
I am a resource
Ending program
Resource destroyed
```

std::unique_ptr 与类

当然用户也可以使用 `std::unique_ptr` 作为类组合的成员。这样便不再需要担心确保类的析构函数删除动态内存，因为 `std::unique_ptr` 将在类对象被销毁时一同销毁。
然而，如果类对象并没有正确的销毁（例如其动态分配没有正确的释放），那么 `std::unique_ptr` 成员也不会被销毁，那么 `std::unique_ptr` 所管理的对象也不会被释放。

错误使用 std::unique_ptr

有两种误用 `std::unique_ptr` 的方式，它们都可以被简单的避免。首先不要让多个类管理同一个资源。例如：

```
Resource* res{ new Resource() };
std::unique_ptr<Resource> res1{ res };
std::unique_ptr<Resource> res2{ res };
```

虽然这在语法上合法，但是最后就是 `res1` 和 `res2` 都会尝试删除 `Resource`，这就导致了未定义行为。
其次不要手动删除 `std::unique_ptr` 所管理的资源：

```
Resource* res{ new Resource() };
std::unique_ptr<Resource> res1{ res };
delete res;
```

如果这么做了，之后的 `std::unique_ptr` 将会删除已被删除的资源，同样导致未定义行为。
注意，`std::make_unique()` 同时避免了上述两种误用。

15.7 std::shared_ptr

不同于 `std::unique_ptr` 设计用来各自拥有并管理资源，`std::shared_ptr` 意为解决需要若干智能指针共同拥有同一个资源。
这就意味着可以有若干个 `std::shared_ptr` 指向同一个资源。在其内部持续追踪总共有多少个 `std::shared_ptr` 正在共享资源。只要至少有一个 `std::shared_ptr` 指向资源，那么该资源就不会被释放。一旦最后一个 `std::shared_ptr` 离开了作用域（或者重新赋值指向其它资源），那么资源则被释放。
与 `std::unique_ptr` 一样，`std::shared_ptr` 定义于 `<memory>` 头文件。

```
#include <iostream>
#include <memory> // std::shared_ptr

class Resource
{
public:
  Resource() { std::cout << "Resource acquired\n"; }
  ~Resource() { std::cout << "Resource destroyed\n"; }
};

int main()
{
  // 分配 Resource 对象，且使其被 std::shared_ptr 所有
  Resource* res { new Resource };
  std::shared_ptr<Resource> ptr1{ res };
  {
    std::shared_ptr<Resource> ptr2 { ptr1 }; // 使另一个 std::shared_ptr 指向同样的东西

    std::cout << "Killing one shared pointer\n";
  } // ptr2 离开作用域，但是无事发生

  std::cout << "Killing another shared pointer\n";

  return 0;
} // ptr1 在此离开作用域，被分配的 Resource 被销毁
```

打印：

```
Resource acquired
Killing one shared pointer
Killing another shared pointer
Resource destroyed
```

上述代码中创建了一个动态 Resource 对象，并设置一个名为 ptr1 的 std::shared_ptr 来管理它。在嵌套的代码块中使用了拷贝构造函数来创建第二个 std::shared_ptr （ptr2 ）指向同样的 Resource 。当 ptr2 离开作用域，Resource 没有被释放，因为 ptr1 仍然指向 Resource 。当 ptr1 离开作用域，ptr1 发现没有任何 std::shared_ptr 在管理 Resource ，所以 Resource 被释放了。

注意第二个共享指针是由第一个共享指针创建而来的。这很重要。考虑下面类似的代码：

```
#include <iostream>
#include <memory> // std::shared_ptr

class Resource
{
public:
  Resource() { std::cout << "Resource acquired\n"; }
  ~Resource() { std::cout << "Resource destroyed\n"; }
};

int main()
{
  Resource* res { new Resource };
  std::shared_ptr<Resource> ptr1 { res };
  {
    std::shared_ptr<Resource> ptr2 { res }; // 直接从 res 创建 ptr2（而不是从 ptr1）

    std::cout << "Killing one shared pointer\n";
  } // ptr2 离开嘞作用域，被分配的 Resource 被销毁

  std::cout << "Killing another shared pointer\n";

  return 0;
} // ptr1 在此离开作用域，被分配的 Resource 再次被销毁
```

打印：

```
Resource acquired
Killing one shared pointer
Resource destroyed
Killing another shared pointer
Resource destroyed
```

接着程序崩溃。

上述两份代码唯一的区别在于第二份代码中创建了两个独立的 std::shared_ptr 。结果就是尽管它们都指向了 Resource ，它们并不知晓对方。当 ptr2 离开作用域，它认为它是 Resource 的唯一拥有者，接着被释放。当 ptr1 离开作用域时考虑同样的事情，尝试再次删除 Resource 。

最佳实践：如果需要多个 std::shared_ptr 指向给定资源，拷贝现有的 std::shared_ptr

即可。

std::make_shared

类似于在 C++14 中使用 `std::make_unique()` 方法来创建 `std::unique_ptr`，`std::make_shared()` 可以用于创建 `std::shared_ptr`（C++11 可用）。

```cpp
#include <iostream>
#include <memory> // std::shared_ptr

class Resource
{
public:
  Resource() { std::cout << "Resource acquired\n"; }
  ~Resource() { std::cout << "Resource destroyed\n"; }
};

int main()
{
  // 分配一个 Resource 对象，且使其被 std::shared_ptr 所有
  auto ptr1 { std::make_shared<Resource>() };
  {
    auto ptr2 { ptr1 }; // 使用拷贝 ptr1 创建 ptr2

    std::cout << "Killing one shared pointer\n";
  } // ptr2 在此离开作用域，无事发生

  std::cout << "Killing another shared pointer\n";

  return 0;
} // ptr1 离开作用域，分配的 Resource 被销毁
```

深挖 std::shared_ptr

不同于 `std::unique_ptr` 内部使用单个指针，`std::shared_ptr` 内部使用两个指针。一个指向了其所管理的资源，另一个指向一个“控制块”，即一个动态分配的对象用于追踪一些东西，包含了有多少个 `std::shared_ptr` 指向了资源。当一个通过 `std::shared_ptr` 构造函数创建的 `std::shared_ptr`，被管理对象（通常为传入的）的内存和控制块（由构造函数创造的）是分开的内存分配。然而当使用 `std::make_shared()`，这可以被优化成一个单独的内存分配，这带来了更好的性能。

这同时解释了为什么独立创建两个 `std::shared_ptr` 指向同一个资源会带来麻烦。每个 `std::shared_ptr` 将拥有一个指针指向资源。然而每个 `std::shared_ptr` 将独立分配其自己的控制块，即表示它是唯一拥有该资源的指针。因此当 `std::shared_ptr` 离开作用域，释放资源，并不会察觉还有其他的 `std::shared_ptr` 同样也尝试管理该资源。

然而当 `std::shared_ptr` 是由拷贝赋值克隆，数据在控制块中可以被恰当的更新来表明现在有额外一个 `std::shared_ptr` 协同管理该资源。

std::shared_ptr 的事故

`std::shared_ptr` 有着与 `std::unique_ptr` 相同的挑战 – 如果 `std::shared_ptr` 没有被正确的销毁（要么是因为它是动态分配的且永远没有删除，要么它是某个动态分配且永远没有删除的对象中的一部分），那么其管理的资源也不会被释放。`std::unique_ptr` 仅需要担心一个智能指针是否被正确的销毁；而 `std::shared_ptr` 需要担心所有的智能指针。如果有任何一个管理资源的 `std::shared_ptr` 没有被正确的销毁，那么资源将不能被正确的释放。

std::shared_ptr 与数组

C++17 或更早，`std::shared_ptr` 并不能正确的管理数组，同时也不允许用来管理 C-style 数组。C++20 开始 `std::shared_ptr` 支持数组。

15.8 Circular dependency issues with std::shared_ptr and std::weak_ptr

上一节讲述了 `std::shared_ptr` 允许用户拥有若干智能指针协同拥有相同的资源。然而在某些情况下，这会变得有问题的。考虑以下案例，当共享指针指向两个独立对象，其中又各自指向对方：

```
#include <iostream>
#include <memory> // std::shared_ptr
#include <string>

class Person
{
  std::string m_name;
  std::shared_ptr<Person> m_partner; // 初始创建为空

public:
  Person(const std::string &name): m_name(name)
  {
    std::cout << m_name << " created\n";
  }
  ~Person()
  {
    std::cout << m_name << " destroyed\n";
  }

  friend bool partnerUp(std::shared_ptr<Person> &p1, std::shared_ptr<Person> &p2)
  {
    if (!p1 || !p2)
```

```
      return false;

    p1->m_partner = p2;
    p2->m_partner = p1;

    std::cout << p1->m_name << " is now partnered with " << p2->m_name << '\n';

    return true;
  }
};

int main()
{
  auto lucy { std::make_shared<Person>("Lucy") }; // 创建一个名为 "Lucy" 的 Person
  auto ricky { std::make_shared<Person>("Ricky") }; // 创建一个名为 "Ricky" 的 Person

  partnerUp(lucy, ricky); // 使 "Lucy" 指向 "Ricky", 反之亦然

  return 0;
}
```

上述例子中动态分配两个 Person，“Lucy”与“Ricky”使用 `make_shared()` （确保 `lucy` 和 `rick` 在 `main()` 结束时销毁）。接着使他们 partner，即在“Lucy”内设置 `std::shared_ptr` 指向“Ricky”，在“Ricky”内指向“Lucy”。共享指针意味着被共享，因此 lucy 共享指针和 rick 的 m_partner 共享指针共同指向“Lucy”（反之亦然）。

然而这个程序并不能像预期那样执行：

```
Lucy created
Ricky created
Lucy is now partnered with Ricky
```

在 `partnerUp()` 被调用，有两个共享指针指向“Ricky”（ricky 以及 Lucy 的 m_partner）以及两个共享指针指向“Lucy”（lucy 以及 Ricky 的 m_partner）。

在 `main()` 结束时，ricky 共享指针首先离开作用域。当这个发生时，ricky 检查是否有任何共享指针协调所有”Ricky“。有（Lucy 的 m_partner），所以”Ricky“没有被释放（如果释放了，Lucy 的 m_partner 会变成悬垂指针）。这个时候，有一个功效指针指向”Ricky“（Lucy 的 m_partner）以及两个共享指针指向”Lucy“（lucy 以及 Ricky 的 m_partner）。

接着 lucy 的共享指针离开作用域，同样的事发生了。lucy 的共享指针检查是否有其他任何共享指针协同所有“Lucy”。有（Ricky 的 m_partner），所以“Lucy”没有被释放。这个时候，仍然有一个共享指针指向“Lucy”（Ricky 的 m_partner）以及一个共享指针指向“Ricky”（Lucy 的 m_partner）。

接着程序结束 – ”Lucy“或”Ricky“没有任何一个被释放！本质上，”Lucy“维系了”Ricky“不被销毁，”Ricky“维系了”Lucy“不被销毁。

事实证明这可以发生的任何事件共享指针形成一个循环引用。

循环引用

循环引用 circular reference（同样被称为 **cyclical reference** 或 **cycle**）是一系列的引用，其中每个对象引用下一个，最后一个对象引用回第一个，最终导致了引用环。引用不需要是真实的 C++ 引用 – 它们可以是指针，唯一 IDs，或者任何其他用于身份识别的对象。
在共享指针的语境中，引用为指针。
这完全就是上述例子中的：”Lucy“指向”Ricky“，同时”Ricky“指向”Lucy“。对于三指针而言，A 指向 B，B 指向 C，C 又指向 A 也会得到相同的结论。共享指针形成的环所造成的实际效果是每个对象都指向下一个对象 – 最后一个对象指向第一个对象。因此，这种情况下没有对象会被释放，因为它们都会认为其它对象仍然需要自身！

一个还原案例

实际上这种循环引用问题甚至会发生在单个 `std::shared_ptr` 上 – `std::shared_ptr` 引用的对象包含自身仍然是一个循环（仅为还原）。尽管它完全不像会在现实中发生，这里是一个额外的例子作为理解：

```
#include <iostream>
#include <memory> // std::shared_ptr

class Resource
{
public:
  std::shared_ptr<Resource> m_ptr {}; // 初始化为空

  Resource() { std::cout << "Resource acquired\n"; }
  ~Resource() { std::cout << "Resource destroyed\n"; }
};

int main()
{
  auto ptr1 { std::make_shared<Resource>() };

  ptr1->m_ptr = ptr1; // m_ptr 现在共享了 Resource，包含了自身

  return 0;
}
```

上述例子中，当 `ptr1` 离开作用域，`Resource` 不会被释放，因为 `Resource` 的 `m_ptr` 共享 `Resource` 。这个时候 `Resource` 能被释放的唯一办法就是设置 `m_ptr` 指向其它（这样就没有任何贡献 `Resource` 的了）。但是不能访问 `m_ptr` 因为 `ptr` 离开了最重要，所以不再可能做到了。`Resouce` 变为了内存泄漏。
所以程序打印：

```
Resource acquired
```

这样就没了。

那么 std::weak_ptr 是什么呢?

`std::weak_ptr` 被设计用来解决上述描述的”循环所有权“问题。一个 `std::weak_ptr` 是一个观测者 – 它可以观测并像 `std::shared_ptr` （或者其他 `std::weak_ptr` ）那样访问同一个对象，但是不会被认为成一个所有者。牢记当一个 `std::shared` 指针离开作用域，它仅会考虑是否有其他的 `std::shared_ptr` 协同所有对象，而 `std::weak_ptr` 则不会考虑!
现在通过 `std::weak_ptr` 来解决之前的问题:

```
#include <iostream>
#include <memory> // std::shared_ptr 与 std::weak_ptr
#include <string>

class Person
{
  std::string m_name;
  std::weak_ptr<Person> m_partner; // 注意: 这里现在是一个 std::weak_ptr

public:
  Person(const std::string &name): m_name(name)
  {
    std::cout << m_name << " created\n";
  }
  ~Person()
  {
    std::cout << m_name << " destroyed\n";
  }

  friend bool partnerUp(std::shared_ptr<Person> &p1, std::shared_ptr<Person> &p2)
  {
    if (!p1 || !p2)
      return false;

    p1->m_partner = p2;
    p2->m_partner = p1;

    std::cout << p1->m_name << " is now partnered with " << p2->m_name << '\n';

    return true;
  }
};

int main()
{
  auto lucy { std::make_shared<Person>("Lucy") };
  auto ricky { std::make_shared<Person>("Ricky") };
```

```
38
39    partnerUp(lucy, ricky);
40
41    return 0;
42  }
```

现在代码行为正常了：

```
1  Lucy created
2  Ricky created
3  Lucy is now partnered with Ricky
4  Ricky destroyed
5  Lucy destroyed
```

使用 std::weak_ptr

`std::weak_ptr` 的缺陷是它不能直接使用（它们并没有操作符->）。为了使用一个 `std::weak_ptr`，必须先转换其成为一个 `std::shared_ptr`。可以使用 `lock()` 成员函数进行转换。

```
1  #include <iostream>
2  #include <memory> // std::shared_ptr 与 std::weak_ptr
3  #include <string>
4
5  class Person
6  {
7    std::string m_name;
8    std::weak_ptr<Person> m_partner; // 注意：这里现在是一个 std::weak_ptr
9
10 public:
11
12   Person(const std::string &name) : m_name(name)
13   {
14     std::cout << m_name << " created\n";
15   }
16   ~Person()
17   {
18     std::cout << m_name << " destroyed\n";
19   }
20
21   friend bool partnerUp(std::shared_ptr<Person> &p1, std::shared_ptr<Person> &p2)
22   {
23     if (!p1 || !p2)
24       return false;
25
26     p1->m_partner = p2;
27     p2->m_partner = p1;
28
```

```
    std::cout << p1->m_name << " is now partnered with " << p2->m_name << '\n';

    return true;
  }

  const std::shared_ptr<Person> getPartner() const { return m_partner.lock(); } // 使用 lock() 转换 weak_ptr 成为 shared_ptr
  const std::string& getName() const { return m_name; }
};

int main()
{
  auto lucy { std::make_shared<Person>("Lucy") };
  auto ricky { std::make_shared<Person>("Ricky") };

  partnerUp(lucy, ricky);

  auto partner = ricky->getPartner(); // 获取 shared_ptr 为 Ricky 的 partner
  std::cout << ricky->getName() << "'s partner is: " << partner->getName() << '\n';

  return 0;
}
```

打印:

```
Lucy created
Ricky created
Lucy is now partnered with Ricky
Ricky's partner is: Lucy
Ricky destroyed
Lucy destroyed
```

不需要担心 `std::shared_ptr` 变量”partner“循环依赖，因为它仅为函数内部的一个本地变量。它最终会在函数结束时离开作用域，同时引用计数减少了 1。

std::weak_ptr 的悬垂指针

因为 `std::weak_ptr` 并不会维持其所持的资源存活，那么对于一个 `std::weak_ptr` 而言，留下它指向一个已被 `std::shared_ptr` 释放的对象也是有可能的。这样的 `std::weak_ptr` 变成了悬垂，使用它会导致未定义行为。

```
#include <iostream>
#include <memory>

class Resource
{
public:
  Resource() { std::cerr << "Resource acquired\n"; }
  ~Resource() { std::cerr << "Resource destroyed\n"; }
```

```
};

auto getWeakPtr()
{
  auto ptr{ std::make_shared<Resource>() }; // Resource 获取

  return std::weak_ptr{ ptr };
} // ptr 离开作用域， Resource 被销毁

int main()
{
  std::cerr << "Getting weak_ptr...\n";

  auto ptr{ getWeakPtr() }; // 悬垂

  std::cerr << "Done.\n";
}
```

上述例子中，在 `getWeakPtr()` 中使用 `std::make_shared()` 来创建一个名为 `ptr` 并所有 `Resource` 对象的 `std::shared_ptr` 变量。函数返回一个 `std::weak_ptr` 给调用者，其不增加引用计数。由于 `ptr` 是本地变量，它在函数结束时离开作用域，即减少引用计数至 0，同时释放 `Resource` 对象。返回的 `std::weak_ptr` 变成了悬垂指针，其指向的 `Resource` 被释放过了。

16 An Introduction to Object Relationships

16.2 Composition

对象组合

复杂对象通常是由小而简单的对象构成的，这个构建过程被称为**对象组合** object composition。
广义上而言，对象组合构造了两个对象之间的“has-a”关系模型。复杂对象有时被称为整体，或者父；而简单对象被称为部分，子，或者组件。
结构体与类有时被称为**组合类型** composite types。
对象组合在 C++ 中很有用，因为它允许用户通过合并简单而又更好管理的不符来创建复杂的类，并减少了复杂性。

对象组合的类型

对象组合又两种基础的子类型：组合 composition 与聚合 aggregation。本章讲解前者，下一章讲解后者。

组合

被视为**组合**，一个对象以及其部分必须拥有以下关系：
- 部分（成员）是对象（类）的一部分 - 部分（成员）同一时间仅属于一个对象（类）- 部分（成员）的存在是由对象（类）所管理的 - 部分（成员）不知道对象（类）的存在
组合关系是一种“部分-整体”的关系，其中“部分”必须是“整体”对象的组成部分。
在组合关系中，对象对“部分”的存在负责。通常而言，这意味着部分是由对象创建时而被创造，由对象销毁时而被销毁。更宽泛的来说，意味着对象管理“部分”的生命周期，而使用对象的用户不需要参与进来。
最后是“部分”不知道“整体”的存在。这也被称为**单向** unidirectional 关系。

实现组合

在 C++ 中，组合是一种实现起来最简单的关系类型。它们通常被创建为结构体或者带有普通数据成员的类。因为这些数据成员是作为结构体/类的一部分而直接存在的，它们的生命周期与类的实例化对象的生命周期绑定在一起。
需要动态分配或者释放的组合可能由指针数据成员实现。这种情况下，组合类应该对所有需要内存管理的部分负责（而不是类的使用者）。
通常而言，如果可以使用组合来设计一个类，则应该这么做。使用组合的类是直接，灵活，且稳固的。

更多例子

Point2D.h:

```
#if !defined(POINT2D_H)
#define POINT2D_H

#include <iostream>

class Point2D
{
  int m_x;
  int m_y;

  public:
  // 默认构造函数
  Point2D() : m_x{0}, m_y{0} {}

  // 指定构造函数
  Point2D(int x, int y) : m_x{x}, m_y{y} {}

  // 重载输出操作符
  friend std::ostream& operator<<(std::ostream& out, const Point2D& point)
  {
    out << '(' << point.m_x << ", " << point.m_y << ')';
    return out;
  }

  void setPoint(int x, int y)
  {
    m_x = x;
    m_y = y;
  }
};

#endif // POINT2D_H
```

Creature.h:

```
#if !defined(CREATURE_H)
#define CREATURE_H

#include "Point2D.h"
#include <iostream>
#include <string>

class Creature
{
  private:
  /* data */
```

```
  std::string m_name;
  Point2D m_location;

  public:
  Creature(const std::string& name, const Point2D& location)
      : m_name{name}, m_location{location} {}

  friend std::ostream& operator<<(std::ostream& out, const Creature& creature)
  {
    out << creature.m_name << " is at " << creature.m_location;
    return out;
  }

  void moveTo(int x, int y) { m_location.setPoint(x, y); }
};

#endif // CREATURE_H
```

main.cpp:

```
#include "Creature.h"
#include "Point2D.h"
#include <iostream>
#include <string>

int main()
{

  std::cout << "Enter a name for your creature: ";
  std::string name;
  std::cin >> name;
  Creature creature{name, {4, 7}};

  while (true)
  {
    std::cout << creature << std::endl;

    std::cout << "Enter new X location for create (-1 to quit): ";
    int x{0};
    std::cin >> x;
    if (x == -1)
      break;

    std::cout << "Enter new Y location for create (-1 to quit): ";
    int y{0};
    std::cin >> y;
    if (y == -1)
      break;

    creature.moveTo(x, y);
```

```
31   }
32
33   return 0;
34 }
```

组合的变体

尽管多数组合在被创建与销毁时，会直接创建与销毁它们的“部分，但是又几种组合的变体与该规则有些差异。
例如：

- 组合可能会延迟一些部分的创建直到它们被需要时。例如，字符串类不会创建一个字符的动态数组直到用户赋予其字符串数据用于存储。
- 组合可能会选择使用一部分被给与的输入，而不是直接由自身来创建这一部分。
- 组合可能会委托其它对象来销毁自身的一些部分。

这里的关键在于组合应该管理其自身的部分而不需要使用该组合的用户来管理。

16.3 Aggregation

聚合

被视为**聚合** aggregation，一个对象以及其部分必须符合以下关系：

- 部分（成员）是对象（类）的一部分
- 部分（成员）同一时间可以属于不止一个对象（类）
- 部分（成员）的存在没有被对象（类）管理
- 部分（成员）不知道对象（类）的存在

与组合类似，聚合仍然是“部分-整体”的关系，“部分”包含在“整体”内，同时也是单向关系。而与组合不同的是，“部分”在同一时间内可以属于多个对象，并且对象对其“部分”的存活并不负责。当一个聚合被创建时，没有创建其部分的责任；当聚合被销毁时，也没有销毁其部分的责任。
可以说聚合构建了“has-a”模型关系。
类似于组合，聚合的“部分”可以是单个的或者多个的。

实现聚合

由于聚合与组合类似都是“部分-整体”的关系，它们的实现几乎一致，其中的差异更多在于语义上。在组合内，通常使用普通成员变量（或者是由组合类自身管理内存分配与释放的指针）来添加“部分”。在聚合内，同样添加成员变量。然而这些成员变量通常是用于指向在类外部创建的对象的引用或者指针。结果而言，一个聚合通常要么是使用指向的对象作为构造函数参数，要么是为空并在之后通过函数或者操作符添加子对象。

由于这些“部分”是存在于类的外部作用域，当类被销毁时，指针或者引用成员变量也会被销毁（但是不删除）。结果而言，“部分”的本体仍然存在。

```
#include <functional> // std::reference_wrapper
#include <iostream>
#include <string>
#include <vector>

class Teacher
{
  private:
  std::string m_name{};

  public:
  Teacher(const std::string& name)
      : m_name{name}
  {
  }

  const std::string& getName() const { return m_name; }
};

using Teachers = std::vector<std::reference_wrapper<Teacher>>;

class Department
{
  private:
  Teachers m_teachers{};

  public:
  Department(const Teachers teachers)
      : m_teachers{teachers}
  {
  }

  void add(Teacher& teacher) { m_teachers.push_back(teacher); }
};

int main(int argc, char const* argv[])
{
```

```
38   Teacher jacob{"Jacob"};
39   Teacher april{"April"};
40
41   Teachers teachers{};
42
43   {
44     Department department{teachers};
45
46     department.add(jacob);
47     department.add(april);
48   }
49
50   std::cout << jacob.getName() << " still exists!\n";
51   std::cout << april.getName() << " still exists!\n";
52
53   return 0;
54 }
```

建模时选择正确的关系

最佳实践：实现最简单的关系类型用于满足项目需求，而不是现实世界中看起来对的那个。

16.4 Association

关联

被视为**关联**，一个对象与另一个对象之间必须符合以下关系：

- 关联对象（成员）除了关联关系以外，与对象（类）无其它关系
- 关联对象（成员）同一时间可以属于不止一个对象（类）
- 关联对象（成员）的存在没有被对象（类）管理
- 关联对象（成员）有可能知道对象（类）的存在

不同于组合与聚合，”部分“是整个对象的一部分，在关联关系中，被关联的对象与对象没有其他关系。类似于聚合，被关联对象可以同时属于若干对象，且不由这些对象关联。然而不同于聚合，即总是单向关系的，关联关系中，可以使单向或者双向的（即两个对象感知对方）。

实现关联

doctor_patient.h:

```cpp
#include <functional>
#include <iostream>
#include <string>
#include <vector>

class Patient;

class Doctor
{
    private:
    std::string m_name{};
    std::vector<std::reference_wrapper<const Patient>> m_patient{};

    public:
    Doctor(const std::string& name)
        : m_name{name}
    {
    }

    void addPatient(Patient& patient);

    friend std::ostream& operator<<(std::ostream& out, const Doctor& doctor);

    const std::string& getName() const { return m_name; }
};

class Patient
{
    private:
    std::string m_name{};
    std::vector<std::reference_wrapper<const Doctor>> m_doctor{};

    void addDoctor(const Doctor& doctor) { m_doctor.push_back(doctor); }

    public:
    Patient(const std::string& name)
        : m_name{name}
    {
    }

    friend std::ostream& operator<<(std::ostream& out, const Patient& patient);

    const std::string& getName() const { return m_name; }

    friend void Doctor::addPatient(Patient& patient);
};

void Doctor::addPatient(Patient& patient)
{
```

```
    m_patient.push_back(patient);

    patient.addDoctor(*this);
}

std::ostream& operator<<(std::ostream& out, const Doctor& doctor)
{
    if (doctor.m_patient.empty())
    {
    out << doctor.m_name << " has no patients right now";
    return out;
    }

    out << doctor.m_name << " is seeing patients: ";
    for (const auto& patient : doctor.m_patient)
    out << patient.get().getName() << ' ';

    return out;
}

std::ostream& operator<<(std::ostream& out, const Patient& patient)
{
    if (patient.m_doctor.empty())
    {
    out << patient.getName() << " has no doctors right now";
    return out;
    }

    out << patient.m_name << " is seeing doctors: ";
    for (const auto& doctor : patient.m_doctor)
    out << doctor.get().getName() << ' ';

    return out;
}
```

自身关联

有时候对象可能与同样类型的其他对象拥有关系。这被称为**自身关联** reflexive association。一个比较好的例子就是大学课程与其前置课程（同样也是大学课程）之间的关系。

```
#include <string>
class Course
{
private:
    std::string m_name;
    const Course* m_prerequisite;

public:
```

```
    Course(const std::string& name, const Course* prerequisite = nullptr):
        m_name{ name }, m_prerequisite{ prerequisite }
    {
    }

};
```

这样就可以实现关联链（一个课程有前置课程，前置课程也有其前置课程等等）。

关联可以使非直接的

之前所有的案例中，使用的要么是指针要么是引用来直接连接对象。然而在关联关系中，这并不是严格需要的。任何种类的数据允许连个对象链接在一起就足够了。

car_driver.h:

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

class Car
{
  private:
  std::string m_name;
  int m_id;

  public:
  Car(const std::string& name, int id)
      : m_name{name}, m_id{id}
  {
  }

  const std::string& getName() const { return m_name; }
  int getId() const { return m_id; }
};

// CarLot 本质上是一个 static 的 Car 数组，以及一个用于获取它们的查询函数。
// 因为是 static 的，不需要为其分配一个 CarLot 类型的对象
class CarLot
{
  private:
  static Car s_carLot[4];

  public:
  CarLot(/* args */) = delete; // 确保不会创建一个 CarLot

  static Car* getCar(int id)
  {
    for (int count{0}; count < 4; ++count)
    {
```

```
      if (s_carLot[count].getId() == id)
      {
        return &(s_carLot[count]);
      }
    }

    return nullptr;
  }
};

Car CarLot::s_carLot[4]{
    {"Prius", 4}, {"Corolla", 17}, {"Accord", 84}, {"Matrix", 62}};

class Driver
{
  private:
  std::string m_name;
  int m_carId; // 可以通过 ID 而不是指针来关联 Car

  public:
  Driver(const std::string& name, int carId)
      : m_name{name}, m_carId{carId}
  {
  }

  const std::string& getName() const { return m_name; }
  int getCarId() const { return m_carId; }
};
```

组合 vs 聚合 vs 关联

Property	Composition	Aggregation	Association
Relationship type	Whole/part	Whole/part	Otherwise unrelated
Members can belong to multiple classes	No	Yes	Yes
Members'existence managed by class	Yes	No	No
Directionality	Unidirectional	Unidirectional	Unidirectional or bidirectional
Relationship verb	Part-of	Has-a	Uses-a

16.5 Dependencies

当一个对象的构建需要另一个对象的功能才能完成时，**依赖** dependency 则会出现。这种关系比关联还要弱，不过任何对被依赖的对象的修改都有可能破坏依赖者的功能。依赖永远是单向关系。

```
#include <iostream>

class Point
{
```

```
private:
    double m_x{};
    double m_y{};
    double m_z{};

public:
    Point(double x=0.0, double y=0.0, double z=0.0): m_x{x}, m_y{y}, m_z{z}
    {
    }

    friend std::ostream& operator<<(std::ostream& out, const Point& point); // Point 依赖于 std
    ::ostream
};

std::ostream& operator<< (std::ostream& out, const Point& point)
{
    // 因为 operator<< 是 Point 类的友元，可以直接访问 Point 的成员
    out << "Point(" << point.m_x << ", " << point.m_y << ", " << point.m_z << ')';

    return out;
}

int main()
{
    Point point1 { 2.0, 3.0, 4.0 };

    std::cout << point1; // 项目依赖 std::cout

    return 0;
}
```

依赖 vs 关联

关于依赖与关联的差异，通常会有一些疑惑。
在 C++ 中，关联是在于两个类之间的类等级关系。也就是说一个类与其关联的类作为成员，保持一个直接或间接的”连接“。
依赖通常来说不在类等级上表现 – 也就是说被依赖的对象并不作为成员来进行连接。而是，被依赖的对象通常在实例化时被需要，或者是作为一个参数传递进一个函数。

16.6 Container classes

容器类 container class 是一种被设计用于存储并组织若干其他类型实例（别的类，或是基础类型）的类。有不同类型的容器类，它们每个都有各种各样的优势，劣势以及限制。迄今为止看到最多的容器就是数组。尽管 C++ 拥有内建的数组功能，程序员通常还是会使用数组容器类

(‘std::array‘ 或 ‘std::vector‘)，因为它们提供了更多的便利。不同于内建数组，数组容器类通常提供动态扩缩（当元素被添加或删除），当传递给函数时记下它们的大小，并且进行边界检查。这不仅使数组容器类比起普通数组更加的便利，同时也更加的安全。
容器类通常实现了最小标准的一系列功能。大多数设计优秀的容器都会包含以下功能：

- 创建一个空容器（通过构造函数）
- 插入新对象进入容器
- 从容器中移除一个对象
- 报告容器中现有的对象数
- 清除容器
- 提供访问存储对象的能力
- 元素排序（可选）

有时候一些特定的容器类会省略上述的一些功能。例如数组容器类通常省略了插入以及移除功能，因为它们很慢并且类的设计者并不鼓励这么使用。
容器类实现了 member-of 关系。例如，一个数组中的元素是 members-of（属于）这个数组的。

容器的类型

容器类通常有两种不同的种类。**值容器** value containers 作为组合 compositions 用于存储它们所持有的对象的副本（因此负责创建与销毁这些副本）。**引用容器** reference containers 作为聚合 aggregations 用于存储其他对象的指针或引用（因此不负责创建或销毁这些对象）。
不同于现实生活的容器可以持有任何类型的对象，C++ 中的容器通常只能持有一种类型的数据。尽管使用上有限制，容器还是非常的有用，同时它们也使得编程编的更加简单，安全，与快速。

16.7 std::initializer_list

考虑一个固定的整型数组：

```
1 int array[5];
```

如果使用值来初始化这个数组，可以直接通过初始化列表语法：

```
1 #include <iostream>
2
3 int main()
4 {
```

```
  int array[] { 5, 4, 3, 2, 1 }; // 初始化列表
  for (auto i : array)
    std::cout << i << ' ';

  return 0;
}
```

同样也可以对动态分配的数组生效：

```
#include <iostream>

int main()
{
  auto* array{ new int[5]{ 5, 4, 3, 2, 1 } }; // 初始化列表
  for (int count{ 0 }; count < 5; ++count)
    std::cout << array[count] << ' ';
  delete[] array;

  return 0;
}
```

上一章中介绍了容器类的概念，并且展示了 IntArray 类持有整型数组的例子：

```
#include <cassert> // assert()
#include <iostream>

class IntArray
{
private:
    int m_length{};
    int* m_data{};

public:
    IntArray() = default;

    IntArray(int length)
        : m_length{ length }
        , m_data{ new int[length]{} }
    {
    }

    ~IntArray()
    {
        delete[] m_data;
        // 这里不需要设置 m_data 为 null 或者 m_length 为 0，因为在这个函数完成之后，对象会立刻被销毁。
    }

    int& operator[](int index)
    {
```

```
        assert(index >= 0 && index < m_length);
        return m_data[index];
    }

    int getLength() const { return m_length; }
};

int main()
{
  // 如果尝试使用容器类进行初始化列表会发生什么呢?
  IntArray array { 5, 4, 3, 2, 1 }; // 这一行不能编译
  for (int count{ 0 }; count < 5; ++count)
    std::cout << array[count] << ' ';

  return 0;
}
```

代码不能被编译的原因是 `IntArray` 不具有一个构造函数用于初始化列表。结果还是只能通过单独的初始化每个元素:

```
int main()
{
  IntArray array(5);
  array[0] = 5;
  array[1] = 4;
  array[2] = 3;
  array[3] = 2;
  array[4] = 1;

  for (int count{ 0 }; count < 5; ++count)
    std::cout << array[count] << ' ';

  return 0;
}
```

这很不好。

使用 std::initializer_list 进行类的初始化

当编译器看到初始化列表，则会自动转换它成为一个 `std::initializer_list` 类型的对象。因此，如果创建一个接受 `std::initializer_list` 参数的构造函数，就可以使用初始化列表来创建对象了。

`std::initializer_list` 位于 `<initializer_list>` 头文件。

关于 `std::initializer_list` 有几点需要知道的。

类似于 `std::array` 或 `std::vector` ，需要通过尖括号告诉 `std::initializer_list` 数据的类型，除非是直接对 `std::initializer_list` 进行初始化。因此几乎见不到一个

纯 `std::initializer_list`，而是类似于 `std::initializer_list<int>` 或者 `std::initializer_list<std::string>`。

其次，`std::initializer_list` 拥有一个 `size()` 函数（取名不当），其返回列表中元素的数量。这对于需要知道传递的列表长度时非常有用。

现在看一下添加了入参为 `std::initializer_list` 构造函数，更新后的代码：

```
#include <cassert> // assert()
#include <initializer_list> // std::initializer_list
#include <iostream>

class IntArray
{
private:
  int m_length {};
  int* m_data {};

public:
  IntArray() = default;

  IntArray(int length)
    : m_length{ length }
    , m_data{ new int[length]{} }
  {

  }

  IntArray(std::initializer_list<int> list) // 允许 IntArray 列表初始化
    : IntArray(static_cast<int>(list.size())) // 使用委派构造函数设置初始数组
  {
    // 现在从 list 开始初始化数组
    int count{ 0 };
    for (auto element : list)
    {
      m_data[count] = element;
      ++count;
    }
  }

  ~IntArray()
  {
    delete[] m_data;
  }

  IntArray(const IntArray&) = delete; // 为了避免浅拷贝
  IntArray& operator=(const IntArray& list) = delete; // 为了避免浅拷贝

  int& operator[](int index)
  {
```

```
    assert(index >= 0 && index < m_length);
    return m_data[index];
  }

  int getLength() const { return m_length; }
};

int main()
{
  IntArray array{ 5, 4, 3, 2, 1 }; // 初始化列表
  for (int count{ 0 }; count < array.getLength(); ++count)
    std::cout << array[count] << ' ';

  return 0;
}
```

现在来看看细节：

```
IntArray(std::initializer_list<int> list)
  : IntArray(static_cast<int>(list.size()))
{
  int count{ 0 };
  for (int element : list)
  {
    m_data[count] = element;
    ++count;
  }
}
```

第一行：根据之前的描述使用尖括号表示列表中元素的类型。本例中因为是 `IntArray` 因此为 int。注意这里没有传递 const 引用的 list。类似于 `std::string_view` ,`std::initializer_list` 是非常轻量的，同时其拷贝是相对廉价的。

第二行：通过委派构造函数（减少冗余代码）委派分配内存给 `IntArray` 至另一个构造函数。另一个构造函数需要知道数组的长度，因此这里传递了 `list.size()` ，即列表中的元素数量。注意 `list.size()` 返回的是 `size_t` （即无符号的），因此还需要转型为有符号的 int。这里使用的是直接初始化而不是花括号初始化，因为花括号初始化倾向于使用列表构造函数。尽管构造函数可以被正确解析，使用直接初始化来初始化类更加的安全。

构造函数的函数体保留用于从列表中拷贝元素至 `IntArray` 类中。由于一些难以说明的原因，`std::initializer_list` 并没有提供通过下标（操作符 []）来访问元素的功能。

然而缺乏下标功能可以被绕过。这里最简单的方法就是使用 for-each 循环。通过 ranged-base 的 for 循环，可以遍历每个初始化列表中的元素，手动的拷贝这些元素进入内部的数组。

一个警告：初始化列表总是喜欢一个匹配的 `initializer_list` 构造函数胜过于其他的潜在匹配构造函数。因此，如下的变量定义：

```
IntArray array { 5 };
```

将匹配 `IntArray(std::initializer_list<int>)` 而不是 `IntArray(int)`。在列表构造函数被定义后，如果还希望匹配到 `IntArray(int)`，那么则需要拷贝初始化或者直接初始化。这也与 `std::vector` 以及其他容器类相似，它们都有列表构造函数以及同类单个入参的构造器：

```
std::vector<int> array(5); // 调用 std::vector::vector(std::vector::size_type), 5 value-initialized elements: 0 0 0 0 0
std::vector<int> array{ 5 }; // 调用 std::vector::vector(std::initializer_list<int>), 1 element : 5
```

使用 std::initializer_list 进行类的赋值

同样也可以使用 `std::initializer_list` 复制操作符来获取 `std::initializer_list` 参数给一个类进行赋值。这与上述例子类似。
注意如果要实现一个以 `std::initializer_list` 为入参的构造函数，就必须要确保满足以下至少一点：

1. 提供重载列表赋值操作符
2. 提供合适的深拷贝的拷贝复制操作符
3. 删除拷贝复制操作符

原因是：考虑以下代码中的类（即没有满足上述三点中的任何一点），与列表赋值声明一起：

```
#include <cassert> // assert()
#include <initializer_list> // std::initializer_list
#include <iostream>

class IntArray
{
private:
  int m_length{};
  int* m_data{};

public:
  IntArray() = default;

  IntArray(int length)
    : m_length{ length }
    , m_data{ new int[length] {} }
  {

  }

  IntArray(std::initializer_list<int> list)
    : IntArray(static_cast<int>(list.size()))
```

```
23   {
24     int count{ 0 };
25     for (auto element : list)
26     {
27       m_data[count] = element;
28       ++count;
29     }
30   }
31
32   ~IntArray()
33   {
34     delete[] m_data;
35   }
36
37   // IntArray(const IntArray&) = delete; // 为了避免浅拷贝
38   // IntArray& operator=(const IntArray& list) = delete; // 为了避免浅拷贝
39
40   int& operator[](int index)
41   {
42     assert(index >= 0 && index < m_length);
43     return m_data[index];
44   }
45
46   int getLength() const { return m_length; }
47 };
48
49 int main()
50 {
51   IntArray array{};
52   array = { 1, 3, 5, 7, 9, 11 }; // 这里是列表赋值声明
53
54   for (int count{ 0 }; count < array.getLength(); ++count)
55     std::cout << array[count] << ' ';
56
57   return 0;
58 }
```

首先，编译器会注意到接受 `std::initializer_list` 的赋值函数并不存在。接着会查看其他可以使用的赋值函数，然后发现隐式提供的拷贝赋值操作符。然而，这个函数仅可用于转换初始化列表成为 `IntArray` 。因为 `{ 1, 3, 5, 7, 9, 11 }` 是一个 `std::initializer_list` ，编译器将会使用列表构造函数来转换一个初始化列表成为一个临时的 `IntArray` 。接着讲调用隐式赋值操作符，即浅拷贝临时的 `IntArray` 给数组对象。

这个时候，无论是临时的 `IntArray` 的 `m_data` 与 `array->m_data` 指向了相同的地址（因为浅拷贝的原因）。已经可以看到后面会发生什么了。

在赋值声明的最后，临时的 `IntArray` 被摧毁。析构函数的调用删除了临时 `IntArray` 中的 `m_data` 。也就留下 `array->m_data` 变为悬垂指针。当尝试使用 `array->m_data` 时则会导

致未定义行为。
最佳实践：如果提供了列表构造函数，再提供一个列表赋值是一个好主意。

17 Inheritance

17.2 Basic inheritance in C++

C++ 中的继承发生在类之间。在一个继承（is-a）的关系中，被继承的类被称为**父类** parent class，**基类** base class 或者**超类** superclass，而继承的类被称为**子类** child class，**派生类** derived class 或是**子类** subclass。

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

class Person
{
public:
    std::string m_name{};
    int m_age{};

    Person(const std::string& name = "", int age = 0)
        : m_name{name}, m_age{age}
    {
    }

    const std::string& getName() const { return m_name; }
    int getAge() const { return m_age; }

};

// BaseballPlayer publicly inheriting Person
class BaseballPlayer : public Person
{
public:
    double m_battingAverage{};
    int m_homeRuns{};

    BaseballPlayer(double battingAverage = 0.0, int homeRuns = 0)
       : m_battingAverage{battingAverage}, m_homeRuns{homeRuns}
    {
    }
};

// Employee publicly inherits from Person
class Employee: public Person
{
public:
    double m_hourlySalary{};
    long m_employeeID{};

    Employee(double hourlySalary = 0.0, long employeeID = 0)
```

```
        : m_hourlySalary{hourlySalary}, m_employeeID{employeeID}
    {
    }

    void printNameAndSalary() const
    {
        std::cout << m_name << ": " << m_hourlySalary << '\n';
    }
};

int main()
{
    // Create a new BaseballPlayer object
    BaseballPlayer joe{};
    // Assign it a name (we can do this directly because m_name is public)
    joe.m_name = "Joe";
    // Print out the name
    std::cout << joe.getName() << '\n'; // use the getName() function we've acquired from the
     Person base class

    Employee frank{20.25, 12345};
    frank.m_name = "Frank"; // we can do this because m_name is public

    frank.printNameAndSalary();

    return 0;
}
```

17.3 Order of construction of derived classes

```
class Base
{
public:
    int m_id {};

    Base(int id=0)
        : m_id { id }
    {
    }

    int getId() const { return m_id; }
};

class Derived: public Base
{
public:
    double m_cost {};

```

```
19      Derived(double cost=0.0)
20          : m_cost { cost }
21      {
22      }
23
24      double getCost() const { return m_cost; }
25  };
```

这个例子中 `Derived` 由 `Base` 中派生出来。

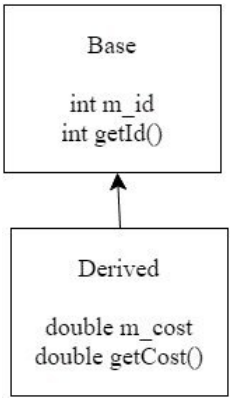

图 3: DerivedBase

因为 `Derived` 从 `Base` 中继承了函数与变量，也许可能会认为 `Base` 的成员被拷贝到了 `Derived` 。然而这并不正确。相反的是 `Derived` 被视为两部分的类：一部分是 `Derived` ，另一部分是 `Base` 。

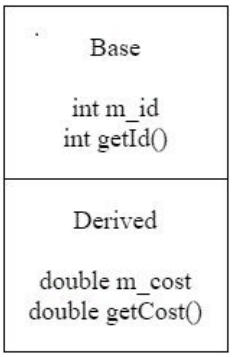

图 4: DerivedBaseCombined

当 C++ 构造派生对象时：首先最开始的类（即继承树最顶层）被构造，接着每个子类按照顺序依次构造，直到最底层的类（继承树的最底层）被构造。

```cpp
#include <iostream>

class Base
{
public:
    int m_id {};

    Base(int id=0)
        : m_id { id }
    {
        std::cout << "Base\n";
    }

    int getId() const { return m_id; }
};

class Derived: public Base
{
public:
    double m_cost {};

    Derived(double cost=0.0)
        : m_cost { cost }
    {
        std::cout << "Derived\n";
    }

    double getCost() const { return m_cost; }
};

int main()
{
    std::cout << "Instantiating Base\n";
    Base base;

    std::cout << "Instantiating Derived\n";
    Derived derived;

    return 0;
}
```

打印:

```
Instantiating Base
Base
Instantiating Derived
Base
Derived
```

继承链的构造顺序

```cpp
#include <iostream>

class A
{
public:
    A()
    {
        std::cout << "A\n";
    }
};

class B: public A
{
public:
    B()
    {
        std::cout << "B\n";
    }
};

class C: public B
{
public:
    C()
    {
        std::cout << "C\n";
    }
};

class D: public C
{
public:
    D()
    {
        std::cout << "D\n";
    }
};

int main()
{
    std::cout << "Constructing A: \n";
    A a;

    std::cout << "Constructing B: \n";
    B b;

    std::cout << "Constructing C: \n";
```

```
48      C c;
49
50      std::cout << "Constructing D: \n";
51      D d;
52  }
```

记住 C++ 总是最先构建”最初“或者”最基础“的类。接着通过继承树的顺序来构造每个派生类。打印:

```
1  Constructing A:
2  A
3  Constructing B:
4  A
5  B
6  Constructing C:
7  A
8  B
9  C
10 Constructing D:
11 A
12 B
13 C
14 D
```

17.4 Constructors and initialization of derived classes

继续使用上一章的例子。

```
1  class Base
2  {
3  public:
4      int m_id {};
5
6      Base(int id=0)
7          : m_id{ id }
8      {
9      }
10
11     int getId() const { return m_id; }
12 };
13
14 class Derived: public Base
15 {
16 public:
17     double m_cost {};
18
19     Derived(double cost=0.0)
20         : m_cost{ cost }
21     {
```

```
    }

    double getCost() const { return m_cost; }
};
```

不使用派生类时，构造函数只需要考虑自身的成员：

```
int main()
{
    Base base{ 5 }; // 使用 Base(int) 构造函数

    return 0;
}
```

实例化时的真实步骤：

1. base 的内存被分配
2. 合适的 Base 构造函数被调用
3. 成员初始化列表初始化变量
4. 构造函数的函数体被执行
5. 调用者拿回控制权

对于派生类的实例化过程而言，也是非常直接的：

```
int main()
{
    Derived derived{ 1.3 }; // 使用 Derived(double) 构造函数

    return 0;
}
```

1. 派生类的内存被分配（足够同时用于 Base 与 Derived 部分）
2. 合适的 Derived 构造函数被调用
3. Base 对象首先使用合适的 Base 构造函数被构建。如果没有合适的，则默认构造函数被调用
4. 成员初始化列表初始化变量
5. 构造函数的函数体被执行
6. 调用者拿回控制权

初始化基类成员

```
class Derived: public Base
{
public:
    double m_cost {};

    Derived(double cost=0.0, int id=0)
        : Base{ id } // 调用 Base(int) 构造函数并传入 id 值!
        , m_cost{ cost }
    {
    }

    double getCost() const { return m_cost; }
};
```

另一个例子

优化之前章节的例子:

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

class Person
{
private:
    std::string m_name;
    int m_age {};

public:
    Person(const std::string_view name = "", int age = 0)
        : m_name{ name }, m_age{ age }
    {
    }

    const std::string& getName() const { return m_name; }
    int getAge() const { return m_age; }

};
// BaseballPlayer 公开继承 Person
class BaseballPlayer : public Person
{
private:
    double m_battingAverage {};
    int m_homeRuns {};

public:
```

```
    BaseballPlayer(const std::string_view name = "", int age = 0,
        double battingAverage = 0.0, int homeRuns = 0)
        : Person{ name, age } // 调用 Person(const std::string_view, int) 来初始化这些字段
        , m_battingAverage{ battingAverage }, m_homeRuns{ homeRuns }
    {
    }

    double getBattingAverage() const { return m_battingAverage; }
    int getHomeRuns() const { return m_homeRuns; }
};

int main()
{
    BaseballPlayer pedro{ "Pedro Cerrano", 32, 0.342, 42 };

    std::cout << pedro.getName() << '\n';
    std::cout << pedro.getAge() << '\n';
    std::cout << pedro.getBattingAverage() << '\n';
    std::cout << pedro.getHomeRuns() << '\n';

    return 0;
}
```

打印:

```
Pedro Cerrano
32
0.342
42
```

继承链

```
#include <iostream>

class A
{
public:
    A(int a)
    {
        std::cout << "A: " << a << '\n';
    }
};

class B: public A
{
public:
    B(int a, double b)
    : A{ a }
    {
```

```
        std::cout << "B: " << b << '\n';
    }
};

class C: public B
{
public:
    C(int a, double b, char c)
    : B{ a, b }
    {
        std::cout << "C: " << c << '\n';
    }
};

int main()
{
    C c{ 5, 4.3, 'R' };

    return 0;
}
```

打印:

```
A: 5
B: 4.3
C: R
```

值得注意的是构造函数仅可以调用它们的父类构造函数。也就是说 C 的构造函数不能直接调用或者传递参数给 A 的构造函数。

析构函数

当一个派生类被销毁，每个析构函数也会与构造函数相反的顺序被调用。上述例子中 c 被销毁，C 的析构函数先被调用，然后是 B 的析构函数，最后是 A 的析构函数。

17.5 Inheritance and access specifiers

protected 访问限定符

C++ 有第三种暂未被提及的访问限定符，因为其仅在继承的上下文中有用。**protected** 访问限定符运允许类成员属于，友元，以及派生类访问成员，而 protected 成员不能被类的外部访问。

```
class Base
{
public:
    int m_public {};    // 可以被任何人访问
```

```
protected:
    int m_protected {}; // 可以被 Base 成员，友元，以及派生类访问
private:
    int m_private {};   // 仅可被 Base 成员以及友元（派生类访问不了）访问
};

class Derived: public Base
{
public:
    Derived()
    {
        m_public = 1;     // 允许
        m_protected = 2;  // 允许
        m_private = 3;    // 不允许
    }
};

int main()
{
    Base base;
    base.m_public = 1;    // 允许
    base.m_protected = 2; // 不允许
    base.m_private = 3;   // 不允许

    return 0;
}
```

何时使用 protected 访问限定符

基类带有 protected 属性的成员可以在派生类中直接访问。这就意味着之后任何关于该成员的修改（类型，值的含义，等等）需要同时去修改基类以及派生类。
隐藏使用 protected 访问限定符在开发者（或组）编写自定义类与派生类时很有用。这样的话修改基类的实现，并更新作为结果的派生类时，可以自主的进行更新（不会消耗大量时间，毕竟派生类的数量是有限的）。
将成员设为私有则意味着公共与派生类都不能直接访问基类。这么做隔离了公共与派生类进行修改，并确保了变体能够正确的被维护。然而这也意味着可能需要大量 public（或 protected）结构来支持所有用于公共或派生类操作的函数，它们都有各自的构建，测试以及维护成本。
通常来说，尽可能的让成员私有，仅当计划需要有派生类的构建与维护接口，而私有成员的成本又太高时，才使用 protected。
最佳实践：更加推荐私有成员而不是 protected 成员。

不同类型的继承，以及它们在访问时的影响

```cpp
// Inherit from Base publicly
class Pub: public Base
{
};

// Inherit from Base protectedly
class Pro: protected Base
{
};

// Inherit from Base privately
class Pri: private Base
{
};

class Def: Base // Defaults to private inheritance
{
};
```

如果不选择继承类型，C++ 默认使用 private 继承（正如成员也是默认为 private 访问）。这就带来了 9 种组合：3 种成员访问限定符（public，private，protected），以及 3 种继承类型（public，private，protected）。

Public 继承

Public 继承是继承中最为广泛使用的。实际上，通常很难看到其他类型的继承。

Access specifier in base class	Access specifier when inherited publicly
Public	Public
Protected	Protected
Private	Inaccessible

```cpp
class Base
{
public:
    int m_public {};
protected:
    int m_protected {};
private:
    int m_private {};
};

class Pub: public Base // note: public inheritance
{
    // Public inheritance means:
    // Public inherited members stay public (so m_public is treated as public)
    // Protected inherited members stay protected (so m_protected is treated as protected)
    // Private inherited members stay inaccessible (so m_private is inaccessible)
public:
```

```
    Pub()
    {
        m_public = 1; // okay: m_public was inherited as public
        m_protected = 2; // okay: m_protected was inherited as protected
        m_private = 3; // not okay: m_private is inaccessible from derived class
    }
};

int main()
{
    // Outside access uses the access specifiers of the class being accessed.
    Base base;
    base.m_public = 1; // okay: m_public is public in Base
    base.m_protected = 2; // not okay: m_protected is protected in Base
    base.m_private = 3; // not okay: m_private is private in Base

    Pub pub;
    pub.m_public = 1; // okay: m_public is public in Pub
    pub.m_protected = 2; // not okay: m_protected is protected in Pub
    pub.m_private = 3; // not okay: m_private is inaccessible in Pub

    return 0;
}
```

最佳实践：使用 public 继承除非有非常特殊的原因。

Protected 继承

Protected 继承是最少见的一种继承方式。可以说是几乎不怎么使用它，除非是特别的案例。通过 protected 继承，public 与 protected 成员都变为 protected，同时 private 保持不可访问。

Access specifier in base class	Access specifier when inherited protectedly
Public	Protected
Protected	Protected
Private	Inaccessible

Private 继承

通过 private 继承，所有基类的成员都被继承为私有。
注意着并不影响派生类访问基类成员！这仅仅影响的是派生来外部想要访问这些基类成员的代码。

```
class Base
{
public:
    int m_public {};
protected:
    int m_protected {};
private:
```

```
    int m_private {};
};

class Pri: private Base // note: private inheritance
{
    // Private inheritance means:
    // Public inherited members become private (so m_public is treated as private)
    // Protected inherited members become private (so m_protected is treated as private)
    // Private inherited members stay inaccessible (so m_private is inaccessible)
public:
    Pri()
    {
        m_public = 1; // okay: m_public is now private in Pri
        m_protected = 2; // okay: m_protected is now private in Pri
        m_private = 3; // not okay: derived classes can't access private members in the base
     class
    }
};

int main()
{
    // Outside access uses the access specifiers of the class being accessed.
    // In this case, the access specifiers of base.
    Base base;
    base.m_public = 1; // okay: m_public is public in Base
    base.m_protected = 2; // not okay: m_protected is protected in Base
    base.m_private = 3; // not okay: m_private is private in Base

    Pri pri;
    pri.m_public = 1; // not okay: m_public is now private in Pri
    pri.m_protected = 2; // not okay: m_protected is now private in Pri
    pri.m_private = 3; // not okay: m_private is inaccessible in Pri

    return 0;
}
```

Access specifier in base class	Access specifier when inherited privately
Public	Private
Protected	Private
Private	Inaccessible

总结

Access specifier in base class	Access specifier when inherited publicly	Access specifier when inherited privately	Access specifier when inherited protectedly
Public	Public	Private	Protected
Protected	Protected	Private	Protected
Private	Inaccessible	Inaccessible	Inaccessible

17.6 Adding new functionality to a derived class

之前的章节里讲过使用派生类最大的好处就是其拥有复用代码的能力。可以从基类中继承功能，接着添加新的功能，修改现有功能，或者是隐藏不需要的功能。从这一章节开始，开始演示上述能力。

一个简单的基类：

```
#include <iostream>

class Base
{
protected:
    int m_value {};

public:
    Base(int value)
        : m_value { value }
    {
    }

    void identify() const { std::cout << "I am a Base\n"; }
};
```

添加新功能至派生类

```
class Derived: public Base
{
public:
    Derived(int value)
        : Base { value }
    {
    }

    int getValue() const { return m_value; }
};
```

17.7 Calling inherited functions and overriding behavior

调用基类方法

```
#include <iostream>

class Base
{
protected:
    int m_value {};
```

```

public:
    Base(int value)
        : m_value { value }
    {
    }

    void identify() const { std::cout << "I am a Base\n"; }
};

class Derived: public Base
{
public:
    Derived(int value)
        : Base { value }
    {
    }
};
int main()
{
    Base base { 5 };
    base.identify();

    Derived derived { 7 };
    derived.identify();

    return 0;
}
```

重新定义行为

```
#include <iostream>

class Derived: public Base
{
public:
    Derived(int value)
        : Base { value }
    {
    }

    int getValue() const { return m_value; }

    // 这里修改了函数
    void identify() const { std::cout << "I am a Derived\n"; }
};
```

对已有功能新增

```cpp
#include <iostream>

class Derived: public Base
{
public:
    Derived(int value)
        : Base { value }
    {
    }

    int getValue() const  { return m_value; }

    void identify() const
    {
        Base::identify(); // 先调用 Base::identify()
        std::cout << "I am a Derived\n"; // 再定义自己的 identify
    }
};
```

或者:

```cpp
#include <iostream>

class Base
{
private:
  int m_value {};

public:
  Base(int value)
    : m_value{ value }
  {
  }

  friend std::ostream& operator<< (std::ostream& out, const Base& b)
  {
    out << "In Base\n";
    out << b.m_value << '\n';
    return out;
  }
};

class Derived : public Base
{
public:
  Derived(int value)
    : Base{ value }
  {
```

```
}

  friend std::ostream& operator<< (std::ostream& out, const Derived& d)
  {
    out << "In Derived\n";
    // static_cast 转换 Derived 为 Base 对象
    out << static_cast<const Base&>(d);
    return out;
  }
};

int main()
{
  Derived derived { 7 };

  std::cout << derived << '\n';

  return 0;
}
```

17.8 Hiding inherited functionality

修改继承成员的访问等级

C++ 允许程序员在派生类中修改继承成员的访问等级。这是通过在不同访问（域）中，对需要修改属性的成员使用 *using* 声明所达成的。

```
#include <iostream>

class Base
{
private:
    int m_value {};

public:
    Base(int value)
        : m_value { value }
    {
    }

protected:
    void printValue() const { std::cout << m_value; }
};

class Derived: public Base
{
public:
    Derived(int value)
```

```
        : Base { value }
    {
    }

    // Base::printValue 本身是 protected 继承的，一次不能被公共访问
    // 但是这里通过 using 声明修改成为了公共访问
    using Base::printValue; // 注意：这里没有圆括号
};
```

隐藏功能

在 C++ 中，除了修改源代码，移除或者限制基类的功能是不可能的。然而在派生类中可以隐藏基类的已有功能，这样可以不再被派生类访问。简单的修改相关的访问限定符即可达成。

```
#include <iostream>
class Base
{
public:
  int m_value {};
};

class Derived : public Base
{
private:
  using Base::m_value;

public:
  Derived(int value)
  // 不可以初始化 m_value，因为它是 Base 成员（Base 必须初始化它）
  {
    // 但是可以对它进行赋值
    m_value = value;
  }
};

int main()
{
  Derived derived { 7 };

  // 以下不能工作是因为 m_value 已经被重新定义为私有
  std::cout << derived.m_value;

  return 0;
}
```

同样也可以在派生类中，标记成员函数为 deleted，来确保不会通过派生类来调用它：

```
#include <iostream>
class Base
```

```
{
private:
  int m_value {};

public:
  Base(int value)
    : m_value { value }
  {
  }

  int getValue() const { return m_value; }
  };

class Derived : public Base
  {
public:
  Derived(int value)
    : Base { value }
  {
  }


  int getValue() = delete; // mark this function as inaccessible
  };

int main()
  {
  Derived derived { 7 };

  // The following won't work because getValue() has been deleted!
  std::cout << derived.getValue();

  return 0;
}
```

17.9 Multiple inheritance

C++ 提供了多重继承的能力。**多重继承** multiple inheritance 让派生类从若干父类中继承成员。

```
#include <string>
#include <string_view>

class Person
{
  private:
  std::string m_name;
  int m_age{};

```

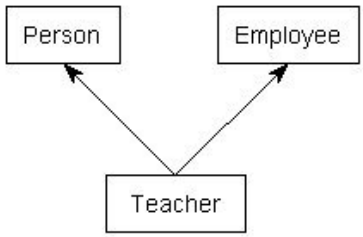

图 5: PersonTeacher

```
  public:
  Person(std::string_view name, int age)
      : m_name{name}, m_age{age}
  {
  }

  const std::string& getName() const { return m_name; }
  int getAge() const { return m_age; }
};

class Employee
{
  private:
  std::string m_employer;
  double m_wage{};

  public:
  Employee(std::string_view employer, double wage)
      : m_employer{employer}, m_wage{wage}
  {
  }

  const std::string& getEmployer() const { return m_employer; }
  double getWage() const { return m_wage; }
};

// Teacher 公共继承 Person 与 Employee
class Teacher : public Person, public Employee
{
  private:
  int m_teachesGrade{};

  public:
  Teacher(std::string_view name, int age, std::string_view employer,
          double wage, int teachesGrade)
      : Person{name, age}, Employee{employer, wage}, m_teachesGrade{teachesGrade}
  {
```

```
  }
};
```

Mixins

mixin 是一种为了给类添加属性的，且用作于继承的小型类。mixin 这个名字意味着类的作用是混合进其它类，而不自己进行实例化。
下面代码中，`Box` 与 `Label` 类作为 mixins 用于继承并创建新的 `Button` 类。

```
#include <string>

struct Point2D
{
  int x;
  int y;
};

class Box
{
  private:
  Point2D m_topLeft{};
  Point2D m_bottomRight{};

  public:
  void setTopLeft(Point2D point) { m_topLeft = point; }
  void setBottomRight(Point2D point) { m_bottomRight = point; }
};

class Label
{
  private:
  std::string m_text{};
  int m_fontSize{};

  public:
  void setText(const std::string_view str) { m_text = str; }
  void setFontSize(int fontSize) { m_fontSize = fontSize; }
};

class Button : public Box, public Label
{
};
```

进阶

因为 mixins 是设计来给派生类添加功能，而不是提供接口，mixins 通常不使用虚函数（下一章节覆盖）。相反的是，如果 mixin 类需要自定义特俗用途，则通常使用模版。正因如此，mixin

类多为模版化的。
使用派生类为模版类型参数，传递给自己的基类。这样的继承被称为 **Curiously Recurring Template Pattern**（简称 CRTP），类似如下：

```
// The Curiously Recurring Template Pattern (CRTP)

template <class T>
class Mixin
{
    // Mixin<T> can use template type parameter T to access members of Derived
    // via (static_cast<T*>(this))
};

class Derived : public Mixin<Derived>
{
};
```

多重继承的问题

由于多重继承看起来像是单个继承的拓展，多重继承引入了很多问题可以显著的增加程序的复杂性，同时使得它们的维护变成噩梦。现在来看一些这样的情况。
首先，可能会出现模棱两可的情况，当若干基类包含同名函数。

```
#include <iostream>

class USBDevice
{
private:
    long m_id {};

public:
    USBDevice(long id)
        : m_id { id }
    {
    }

    long getID() const { return m_id; }
};

class NetworkDevice
{
private:
    long m_id {};

public:
    NetworkDevice(long id)
        : m_id { id }
```

```
    {
    }

    long getID() const { return m_id; }
};

class WirelessAdapter: public USBDevice, public NetworkDevice
{
public:
    WirelessAdapter(long usbId, long networkId)
        : USBDevice { usbId }, NetworkDevice { networkId }
    {
    }
};

int main()
{
    WirelessAdapter c54G { 5442, 181742 };
    std::cout << c54G.getID(); // 哪个 getID() 被调用了?

    return 0;
}
```

不过有一种方法可以绕过这个问题：显式指定调用的版本：

```
int main()
{
    WirelessAdapter c54G { 5442, 181742 };
    std::cout << c54G.USBDevice::getID();

    return 0;
}
```

其次，一个更严肃的问题是菱形问题。

```
class PoweredDevice
{
};

class Scanner: public PoweredDevice
{
};

class Printer: public PoweredDevice
{
};

class Copier: public Scanner, public Printer
{
};
```

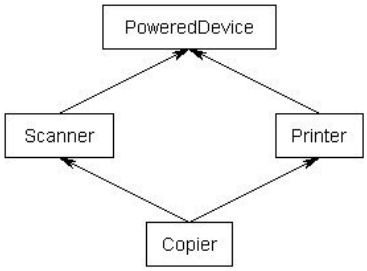

图 6: PoweredDevice

下一章节会讲解如何解决这个问题。

多重继承是否值得?

事实证明，大多数可以使用多重继承来解决的问题也可以被单独继承解决。很多面向对象语言甚至于不支持多重继承。很多相关的现代语言如 Java 和 C# 限制类为单独继承普通类，但是允许多重继承接口类（之后会提到）。这些语言中的禁止多重继承概念反而使得语言变得更复杂，最终导致更多问题。
最佳实践：避免多重继承，除非不这么做会导致更大的复杂性。

18 Virtual Functions

18.1 Pointers and references to the base class of derived objects

在之前的章节中学习了如何使用继承从现有类派生出新的类。这一章将会集中学习最重要也是最强大的继承 – 虚函数。

```cpp
#include <string_view>

class Base
{
protected:
    int m_value {};

public:
    Base(int value)
        : m_value{ value }
    {
    }

    std::string_view getName() const { return "Base"; }
    int getValue() const { return m_value; }
};

class Derived: public Base
{
public:
    Derived(int value)
        : Base{ value }
    {
    }

    std::string_view getName() const { return "Derived"; }
    int getValueDoubled() const { return m_value * 2; }
};
```

当创建一个派生对象，它包含了 `Base` 部分（首先被构造出来），以及 `Derived` 部分（其次被构造）。注意继承表示两个对象之间是 is-a 关系。因为 `Derived` is-a `Base`，那么 `Derived` 包含了 `Base` 部分是合理的。

指针，引用，以及派生类

对 `Derived` 对象设置指针和引用应该是符合直觉的。

```cpp
#include <iostream>

int main()
{
```

```
    Derived derived{ 5 };
    std::cout << "derived is a " << derived.getName() << " and has value " << derived.getValue
    () << '\n';

    Derived& rDerived{ derived };
    std::cout << "rDerived is a " << rDerived.getName() << " and has value " << rDerived.
    getValue() << '\n';

    Derived* pDerived{ &derived };
    std::cout << "pDerived is a " << pDerived->getName() << " and has value " << pDerived->
    getValue() << '\n';

    return 0;
}
```

打印:

```
derived is a Derived and has value 5
rDerived is a Derived and has value 5
pDerived is a Derived and has value 5
```

然而因为 `Derived` 拥有 `Base` 部分，一个更加有趣的问题就是 C++ 是否让用户设置 `Base` 指针或者引用给 `Derived` 对象。实际上这是可以的!

```
#include <iostream>

int main()
{
    Derived derived{ 5 };

    // 它们都是合法的!
    Base& rBase{ derived };
    Base* pBase{ &derived };

    std::cout << "derived is a " << derived.getName() << " and has value " << derived.getValue
    () << '\n';
    std::cout << "rBase is a " << rBase.getName() << " and has value " << rBase.getValue() << '
    \n';
    std::cout << "pBase is a " << pBase->getName() << " and has value " << pBase->getValue() <<
     '\n';

    return 0;
}
```

打印:

```
derived is a Derived and has value 5
rBase is a Base and has value 5
pBase is a Base and has value 5
```

这个结果可能并不像一开始预期那样!

这是因为 rBase 与 pBase 是 Base 的引用与指针，它们只能看到 Base 的成员（或者是任何 Base 所继承的类的成员）。因此即使 Derived::getName() 覆盖 shadows 了 Derived 对象的 Base::getName()，Base 指针/引用并不能看到 Derived::getName()。结果而言，它们调用的是 Base::getName()。

注意这也同样意味着不能使用 rBase 或 pBase 来调用 Derived::getValueDoubled()。它们不能看到任何 Derived 的内容。

下面是一个将在下一章构建的更为复杂的例子：

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

class Animal
{
protected:
    std::string m_name;

    // 使该构造函数为 protected，因为不希望任何人可以直接创建 Animal 对象，
    // 但是仍然希望派生类可以使用它。
    Animal(std::string_view name)
        : m_name{ name }
    {
    }

    // 用于阻止切片（稍后讲解）
    Animal(const Animal&) = default;
    Animal& operator=(const Animal&) = default;

public:
    std::string_view getName() const { return m_name; }
    std::string_view speak() const { return "???"; }
};

class Cat: public Animal
{
public:
    Cat(std::string_view name)
        : Animal{ name }
    {
    }

    std::string_view speak() const { return "Meow"; }
};

class Dog: public Animal
{
public:
```

```
    Dog(std::string_view name)
        : Animal{ name }
    {
    }

    std::string_view speak() const { return "Woof"; }
};

int main()
{
    const Cat cat{ "Fred" };
    std::cout << "cat is named " << cat.getName() << ", and it says " << cat.speak() << '\n';

    const Dog dog{ "Garbo" };
    std::cout << "dog is named " << dog.getName() << ", and it says " << dog.speak() << '\n';

    const Animal* pAnimal{ &cat };
    std::cout << "pAnimal is named " << pAnimal->getName() << ", and it says " << pAnimal->
    speak() << '\n';

    pAnimal = &dog;
    std::cout << "pAnimal is named " << pAnimal->getName() << ", and it says " << pAnimal->
    speak() << '\n';

    return 0;
}
```

打印:

```
cat is named Fred, and it says Meow
dog is named Garbo, and it says Woof
pAnimal is named Fred, and it says ???
pAnimal is named Garbo, and it says ???
```

这里也有同样的问题。因为 `pAnimal` 是 `Animal` 指针，它只可以看见作为 `Animal` 类的部分。因此，`pAnimal->speak()` 调用了 `Animal::speak()` 而不是 `Dog::speak()` 或 `Cat::speak()` 函数。

对基类使用指针与引用

这个时候有人可能会说“上述的例子看起来有点蠢。为什么不直接设置派生对象的指针或者引用?”这有几个很好的原因。

首先，假设要编写一个函数打印动物名以及声音。不使用基类指针的话则需要使用重载函数，例如:

```
void report(const Cat& cat)
{
    std::cout << cat.getName() << " says " << cat.speak() << '\n';
```

```
}

void report(const Dog& dog)
{
    std::cout << dog.getName() << " says " << dog.speak() << '\n';
}
```

这么看起来并不难，但是如果有 30 种不同的动物类型呢？需要编写 30 个几乎相同的函数！另外每次新增一个动物类型都需要为其编写一个新函数。
然而因为 Cat 和 Dog 都是 Animal 的派生类，Cat 和 Dog 都拥有 Animal 部分。因此以下的做法是符合常理的：

```
void report(const Animal& rAnimal)
{
    std::cout << rAnimal.getName() << " says " << rAnimal.speak() << '\n';
}
```

当然了，这样的问题还是在于引用还是调用基类的函数而不是派生类的函数。
其次，假设存在 3 个 cat 与 3 个 dog，并希望使它们在同一个数组中便于访问。因为数组仅可以保存一种类型的对象，在不使用基类的指针或者引用的情况下，需要为每个派生类型都去创建各自的数组。例如：

```
#include <array>
#include <iostream>

// Cat and Dog from the example above

int main()
{
    const auto& cats{ std::to_array<Cat>({{ "Fred" }, { "Misty" }, { "Zeke" }}) };
    const auto& dogs{ std::to_array<Dog>({{ "Garbo" }, { "Pooky" }, { "Truffle" }}) };

    // Before C++20
    // const std::array<Cat, 3> cats{{ { "Fred" }, { "Misty" }, { "Zeke" } }};
    // const std::array<Dog, 3> dogs{{ { "Garbo" }, { "Pooky" }, { "Truffle" } }};

    for (const auto& cat : cats)
    {
        std::cout << cat.getName() << " says " << cat.speak() << '\n';
    }

    for (const auto& dog : dogs)
    {
        std::cout << dog.getName() << " says " << dog.speak() << '\n';
    }

    return 0;
}
```

那么如果是 30 种不同类型的动物呢？需要 30 个数组！
然而因为 Cat 和 Dog 都是 Animal 的派生类，以下的做法也是符合常理的：

```
#include <array>
#include <iostream>

// Cat and Dog from the example above

int main()
{
    const Cat fred{ "Fred" };
    const Cat misty{ "Misty" };
    const Cat zeke{ "Zeke" };

    const Dog garbo{ "Garbo" };
    const Dog pooky{ "Pooky" };
    const Dog truffle{ "Truffle" };

    // Set up an array of pointers to animals, and set those pointers to our Cat and Dog objects
    // Note: to_array requires C++20 support (and at the time of writing, Visual Studio 2022 still doesn't support it correctly)
    const auto animals{ std::to_array<const Animal*>({&fred, &garbo, &misty, &pooky, &truffle, &zeke }) };

    // Before C++20, with the array size being explicitly specified
    // const std::array<const Animal*, 6> animals{ &fred, &garbo, &misty, &pooky, &truffle, &zeke };

    for (const auto animal : animals)
    {
        std::cout << animal->getName() << " says " << animal->speak() << '\n';
    }

    return 0;
}
```

然而这里遇到的问题还是与上述案例一样。现在可以猜一猜虚函数是做什么用的呢？

18.2 Virtual functions and polymorphism

虚函数是一种特殊类型的函数，当被调用时解析基类与派生类之间最-派生版本的函数。这个能力也被称为**多态** polymorphism。如果派生函数拥有相同签名（名称，入参类型，以及是否为 const）以及相同的返回类型，则被视为匹配。这样的函数被称为**重写** overrides。
要是函数成为虚函数，仅需要简单的在函数声明前加上 virtual 关键字即可。

```
#include <iostream>
#include <string_view>
```

```
3
4  class Base
5  {
6  public:
7      virtual std::string_view getName() const { return "Base"; } // 注意额外的 virtual 关键字
8  };
9
10 class Derived: public Base
11 {
12 public:
13     virtual std::string_view getName() const { return "Derived"; }
14 };
15
16 int main()
17 {
18     Derived derived;
19     Base& rBase{ derived };
20     std::cout << "rBase is a " << rBase.getName() << '\n';
21
22     return 0;
23 }
```

打印:

```
1 rBase is a Derived
```

因为 rBase 是 Derived 对象中 Base 部分的引用，当 rBase.getName() 被计算时，通常会是 Base::getName() 。然而由于 Base::getName() 是虚函数，这就告诉程序去查看在 Base 与 Derived 之前是否有任何更-派生版本的函数。这个情况下即 Derived::getName() !

下面是一个稍微复杂一点的例子:

```
1  #include <iostream>
2  #include <string_view>
3
4  class A
5  {
6  public:
7      virtual std::string_view getName() const { return "A"; }
8  };
9
10 class B: public A
11 {
12 public:
13     virtual std::string_view getName() const { return "B"; }
14 };
15
16 class C: public B
17 {
```

```
public:
    virtual std::string_view getName() const { return "C"; }
};

class D: public C
{
public:
    virtual std::string_view getName() const { return "D"; }
};

int main()
{
    C c;
    A& rBase{ c };
    std::cout << "rBase is a " << rBase.getName() << '\n';

    return 0;
}
```

打印:

```
rBase is a C
```

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

class Animal
{
protected:
    std::string m_name;

    // 使该构造函数为 protected，因为不希望任何人可以直接创建 Animal 对象，
    // 但是仍然希望派生类可以使用它。
    Animal(const std::string& name)
        : m_name{ name }
    {
    }

public:
    const std::string& getName() const { return m_name; }
    virtual std::string_view speak() const { return "???"; }
};

class Cat: public Animal
{
public:
    Cat(const std::string& name)
        : Animal{ name }
    {
```

```
    }

    virtual std::string_view speak() const { return "Meow"; }
};

class Dog: public Animal
{
public:
    Dog(const std::string& name)
        : Animal{ name }
    {
    }

    virtual std::string_view speak() const { return "Woof"; }
};

void report(const Animal& animal)
{
    std::cout << animal.getName() << " says " << animal.speak() << '\n';
}

int main()
{
    Cat cat{ "Fred" };
    Dog dog{ "Garbo" };

    report(cat);
    report(dog);

    return 0;
}
```

打印:

```
Fred says Meow
Garbo says Woof
```

同样的:

```
Cat fred{ "Fred" };
Cat misty{ "Misty" };
Cat zeke{ "Zeke" };

Dog garbo{ "Garbo" };
Dog pooky{ "Pooky" };
Dog truffle{ "Truffle" };

// 设置一个 animals 指针的数组, 同时放入 Cat 与 Dog 对象的指针
Animal* animals[]{ &fred, &garbo, &misty, &pooky, &truffle, &zeke };

```

```
for (const auto* animal : animals)
    std::cout << animal->getName() << " says " << animal->speak() << '\n';
```

打印:

```
Fred says Meow
Garbo says Woof
Misty says Meow
Pooky says Woof
Truffle says Woof
Zeke says Meow
```

虚函数的返回类型

在普通的情况下，虚函数的返回类型以及其的重写必须匹配。考虑以下例子:

```
class Base
{
public:
    virtual int getValue() const { return 5; }
};

class Derived: public Base
{
public:
    virtual double getValue() const { return 6.78; }
};
```

这种情况下，`Derived::getValue()` 不会被视为 `Base::getValue()` 的重写而导致编译错误。

构造函数与析构函数不要调用虚函数

当派生类被创建时，基类部分会首先被创建。如果调用了基类构造函数中的虚函数，而派生部分并未被创建，这就会无法调用派生版本的函数。在 C++ 中，则还是会调用基类版本的函数。析构函数同理，如果在基类析构函数中调用虚函数，还是只会使用基类版本的函数，因为派生部分已经被销毁了。
最佳实践：永远不要在构造函数与析构函数中调用虚函数。

虚函数的缺陷

由于大部分时间都希望函数是虚函数，那么为什么不让所有函数都称为虚函数呢？答案是因为低性能 – 虚函数的解析要长于正常函数。除此以外，编译器还需要为每个拥有一个或多个虚函数的类对象分配额外的指针。

18.3 The override and final specifiers, and covariant return types

在处理一些继承的常见挑战时，有两个特殊的标识符：override 与 final。注意它们不被视为关键字 – 它们是在特定上下文中有特殊意义的普通标识符。
尽管 final 并没有那么常用，override 却是一个需要经常使用的好东西。这一节中将讲解它们，并讲解虚函数重写返回类型必须匹配的一种例外规则。

override 限定符

上一节提到过，派生类的虚函数被视为重写仅会考虑其签名与返回类型完全匹配。这就会导致疏忽错误，当一个函数意图重写但实际上并没有。
考虑以下例子：

```
#include <iostream>
#include <string_view>

class A
{
public:
  virtual std::string_view getName1(int x) { return "A"; }
  virtual std::string_view getName2(int x) { return "A"; }
};

class B : public A
{
public:
  virtual std::string_view getName1(short int x) { return "B"; } // 注意：参数是 short int
  virtual std::string_view getName2(int x) const { return "B"; } // 注意：函数是 const
};

int main()
{
  B b{};
  A& rBase{ b };
  std::cout << rBase.getName1(1) << '\n';
  std::cout << rBase.getName2(2) << '\n';

  return 0;
}
```

为了解决想要重写的函数却没有成功的问题，override 限定符可以作用于任何虚函数，仅需要在 const 处添加即可。如果函数没有重写基类函数（或是应用到了非虚函数），编译器则会报错。

```
#include <string_view>

class A
{
```

```
public:
  virtual std::string_view getName1(int x) { return "A"; }
  virtual std::string_view getName2(int x) { return "A"; }
  virtual std::string_view getName3(int x) { return "A"; }
};

class B : public A
{
public:
  std::string_view getName1(short int x) override { return "B"; } // 编译错误，函数并没有重写
  std::string_view getName2(int x) const override { return "B"; } // 编译错误，函数并没有重写
  std::string_view getName3(int x) override { return "B"; } // 可以，函数重写了 A::getName3(int
    )

};

int main()
{
  return 0;
}
```

最佳实践：使用 virtual 关键字在基类函数的虚函数上。使用 override 限定符（而不是 virtual 关键字）在派生类中需要重写的函数上。

final 限定符

可能也有一种情况就是不希望任何人去重写一个虚函数，或是从类中继承。那么 final 限定符就可以用来强制告诉编译器不让这么做。如果用户尝试对标记 final 限定符上重写函数或者继承类，编译器则会报错。

final 限定符处于 override 限定符同样的位置。

```
#include <string_view>

class A
{
public:
  virtual std::string_view getName() { return "A"; }
};

class B : public A
{
public:
  // 注意 final 限定符在下一行 -- 使该函数不再可以被重写
  std::string_view getName() override final { return "B"; } // 可以，重写 A::getName()
};

class C : public B
```

```
{
public:
  std::string_view getName() override { return "C"; } // 编译错误：重写 B::getName()，其为
    final
};
```

组织类被继承也可以在类名前使用 final 限定符：

```
#include <string_view>

class A
{
public:
  virtual std::string_view getName() { return "A"; }
};

class B final : public A // 注意这里使用了 final 限定符
{
public:
  std::string_view getName() override { return "B"; }
};

class C : public B // 编译错误：不可以继承 final 类
{
public:
  std::string_view getName() override { return "C"; }
};
```

协变返回类型

还有一个特殊情况就是派生类的虚函数重写可以拥有与基类不同类型的返回类型，并且还是被视为匹配的重写。如果虚函数的返回类型是某个类的指针或者引用，重写函数可以返回派生类的指针或者引用。这被称为**协变返回类型** covariant return types。

```
#include <iostream>
#include <string_view>

class Base
{
public:
  // 这个版本的 getThis() 返回一个指向 Base 类的指针
  virtual Base* getThis() { std::cout << "called Base::getThis()\n"; return this; }
  void printType() { std::cout << "returned a Base\n"; }
};

class Derived : public Base
{
public:
```

```
  // 通常的重写函数必须返回与基类函数相同类型的对象
  // 然而，因为 Derived 是从 Base 派生而来，因此返回 Derived* 而不是 Base*
  Derived* getThis() override { std::cout << "called Derived::getThis()\n";  return this; }
  void printType() { std::cout << "returned a Derived\n"; }
};

int main()
{
  Derived d{};
  Base* b{ &d };
  d.getThis()->printType();   // 调用 Derived::getThis()，返回 Derived*，调用 Derived::
    printType
  b->getThis()->printType();  // 调用 Derived::getThis()，返回 Base*，调用 Base::printType

  return 0;
}
```

打印:

```
called Derived::getThis()
returned a Derived
called Derived::getThis()
returned a Base
```

一个有趣的地方是关于协变返回类型: C++ 不可以动态选择类型，因此用户总是会获得匹配真实版本的被调用函数的类型。

上述例子中，首先调用 `d.getThis()` 。因为 d 是一个 Derived，调用 `Derived::getThis()` ，返回的是 `*Derived` 。该 `*Derived` 接着调用非虚函数 `Derived::printType()` 。

接着是有趣的部分了。调用 `b->getThis()` 。变量 b 是指向 Derived 对象的 Base 指针。`Base::getThis()` 是一个虚函数，所以调用 `Derived::getThis()` 。尽管其返回的是 `*Derived` ，但是因为 Base 版本的函数返回的是 `Base*` ，返回的 `Derived*` 被转换成 `Base*` 。因为 `Base::printType()` 是非虚函数，其被调用。

换句话说，上述例子中，如果在派生类的对象上调用 `getThis()` 只会得到一个 `Derived*` 。注意如果 `printType()` 是虚函数的话，`b->getThis()` （即 `Base*` 类型的对象）的返回值将会通过虚函数解析，接着 `Derived::printType()` 则会被调用。

协变返回类型通常用于虚函数成员函数返回一个类的指针或者引用，该类包含了成员函数（例如 `Base::getThis()` 返回一个 `Base*` ，同时 `Derived::getThis()` 返回一个 `Derived*` ）。然而这并不是严格必要的。协变返回类型可以用于任何这样的场景：重写成员函数的返回值类型是从基类虚函数成员中派生出来的。

18.4 Virtual destructors, virtual assignment, and overriding virtualization

虚析构函数

尽管 C++ 为所有类都提供了默认的析构函数，用户有时也需要自定义的析构函数（特别是如果类需要释放内存）。那么如果在处理继承的时候，析构函数应该**总是**虚函数。

```cpp
#include <iostream>
class Base
{
public:
    ~Base() // 注意：非虚函数
    {
        std::cout << "Calling ~Base()\n";
    }
};

class Derived: public Base
{
private:
    int* m_array;

public:
    Derived(int length)
      : m_array{ new int[length] }
    {
    }

    ~Derived() // 注意：非虚函数（编译器有可能会警告）
    {
        std::cout << "Calling ~Derived()\n";
        delete[] m_array;
    }
};

int main()
{
    Derived* derived { new Derived(5) };
    Base* base { derived };

    delete base;

    return 0;
}
```

注意：如果编译上述代码，编译器可能会警告析构函数为非虚函数（这个例子中是故意的）。由于 `base` 是 `Base` 的指针，当 `base` 被删除时，程序会检测 `Base` 的析构函数是否为虚

函数。如果不是，则会假设仅需调用 `Base` 析构函数。可以看到上述例子会打印：

```
Calling ~Base()
```

然而这里事非常希望 delete 函数调用 Derived 的析构函数（也会依次调用 Base 的析构函数），否则 m_array 不会被删除。那么这里可以使 Base 的析构函数变为虚函数：

```
#include <iostream>
class Base
{
public:
    virtual ~Base() // 注意：virtual 关键字
    {
        std::cout << "Calling ~Base()\n";
    }
};

class Derived: public Base
{
private:
    int* m_array;

public:
    Derived(int length)
      : m_array{ new int[length] }
    {
    }

    virtual ~Derived() // 注意：virtual 关键字
    {
        std::cout << "Calling ~Derived()\n";
        delete[] m_array;
    }
};

int main()
{
    Derived* derived { new Derived(5) };
    Base* base { derived };

    delete base;

    return 0;
}
```

现在打印：

```
Calling ~Derived()
Calling ~Base()
```

规则：当处理继承时，应该使任何析构函数都显式的成为虚函数。

正如普通的虚函数成员那样，如果基类函数是虚函数，所有派生的重写都被视为虚函数，无论它们是否有指定。派生类中创建一个空的析构函数并标记为虚函数并不是必须的。
注意如果希望基类拥有一个虚函数的析构函数并且是为空，可以这样进行定义：

```
virtual ~Base() = default; // 生成一个 默认的虚化析构函数
```

虚化赋值

赋值操作符也可以被虚化。然而不同于析构函数使用虚化总是一个好主意，虚化赋值操作符会带来很多 bug。结果而言，还是推荐赋值为非虚化函数，也是为了简化的目的。

忽略虚化

罕见的情况下希望忽略函数的虚化。例如：

```
class Base
{
public:
    virtual ~Base() = default;
    virtual const char* getName() const { return "Base"; }
};

class Derived: public Base
{
public:
    virtual const char* getName() const { return "Derived"; }
};
```

有些情况下希望指向 `Derived` 对象的 `Base` 类型指针调用的是 `Base::getName()` 而不是 `Derived::getName()` 。仅需使用作用域解析操作符就可以做到：

```
#include <iostream>
int main()
{
    Derived derived;
    const Base& base { derived };
    // 调用 Base::getName() 而不是虚化的 Derived::getName()
    std::cout << base.Base::getName() << '\n';

    return 0;
}
```

可能并不需要经常这么做，不过这是一个可以了解到的可行方案。

应该让所有的析构函数虚化吗?

这是一个新手程序员经常会问到的问题。正如最开始的例子中展示，如果基类析构函数并没有被标记为 virtual，那么如果程序员在之后删除嘞基类类型指向派生类对象的指针时，程序则会有泄漏内存的风险。其中一种避免问题发生的方式就是标记所有析构函数为 virtual。但是这应该吗?
简单来说是应该的，因为之后这样使用任何类当做基类 – 但是这会带来性能惩罚（虚化指针被添加到类的所有实例中）。因此用户需要权衡成本。
最终建议:

- 如果希望自定义类被继承，请确保析构函数为虚函数。
- 如果不希望自定义类被继承，请标记 final。这样就可以阻止被其他类继承，而不遵守类本身的限制。

18.5 Early binding and late binding

这一章节与下一章节将会学习到虚函数是如何实现的。虽然这个信息不是使用虚函数严格需求的，它很有趣。可以认为这两个章节为可选阅读。
当 C++ 程序被执行时，它会顺序执行，开始于 main() 函数。当遇到函数调用时，执行点跳转到被调用函数的起始处。那么 CPU 是怎么知道的呢?
当一个程序被编译，编译器转换每个 C++ 程序中的声明成为一行或多行的机器语言。每一行的机器语言都会被给予唯一的序列地址。函数也是同样的 – 当遇到一个函数，它被转换为机器语言并给与下一个可用的地址。因此，每个函数都会带有一个唯一的地址。
绑定 Binding 指的是用于转换标识符（例如变量和函数名）成为地址的过程。尽管绑定是用于变量和函数的，这一节中着重谈论函数绑定。

Early binding

编译期遇到的大多数函数调用都是直接函数调用。直接的函数调用是一个声明:

```
#include <iostream>

void printValue(int value)
{
    std::cout << value;
}

int main()
{
    printValue(5); // 直接的函数调用
    return 0;
```

```
}
```

直接函数调用可以使用 early binding 的过程进行解析。**Early binding**（也被称为 static binding）意为编译器（或 linker）可以通过机器地址直接关联标识符名称（例如函数或变量名）。鉴于所有函数都有唯一的地址。因此当编译器（或 linker）遇到了函数调用，替换函数调用为机器语言指令，并告诉 CPU 跳转的函数的地址。

```
#include <iostream>

int add(int x, int y)
{
    return x + y;
}

int subtract(int x, int y)
{
    return x - y;
}

int multiply(int x, int y)
{
    return x * y;
}

int main()
{
    int x{};
    std::cout << "Enter a number: ";
    std::cin >> x;

    int y{};
    std::cout << "Enter another number: ";
    std::cin >> y;

    int op{};
    do
    {
        std::cout << "Enter an operation (0=add, 1=subtract, 2=multiply): ";
        std::cin >> op;
    } while (op < 0 || op > 2);

    int result {};
    switch (op)
    {
        // 直接使用 early binging 来调用目标函数
        case 0: result = add(x, y); break;
        case 1: result = subtract(x, y); break;
        case 2: result = multiply(x, y); break;
    }
```

```
43
44      std::cout << "The answer is: " << result << '\n';
45
46      return 0;
47  }
```

Late binding

在有些程序中，函数调用只能在运行时才可以解析。在 C++ 中，有时被称为 **late binding**（或者使用虚函数方案 **dynamic binding**）。
注：通常的编程术语中，“late binding”的概念意为函数被调用时在运行时查找函数名称，而 C++ 并不支持这么做。C++ 中的”late binding“通常用于函数被调用时，不被编译期或 linker 所知。相反，函数调用（在运行时）在此时被决定。
C++ 中，使用 late binding 的一种方式是通过函数指针。
通过函数指针调用函数同样也被认作是间接函数调用。下面使用函数指针改造上面的代码：

```
1   #include <iostream>
2
3   int add(int x, int y)
4   {
5       return x + y;
6   }
7
8   int subtract(int x, int y)
9   {
10      return x - y;
11  }
12
13  int multiply(int x, int y)
14  {
15      return x * y;
16  }
17
18  int main()
19  {
20      int x{};
21      std::cout << "Enter a number: ";
22      std::cin >> x;
23
24      int y{};
25      std::cout << "Enter another number: ";
26      std::cin >> y;
27
28      int op{};
29      do
30      {
```

```
31          std::cout << "Enter an operation (0=add, 1=subtract, 2=multiply): ";
32          std::cin >> op;
33      } while (op < 0 || op > 2);
34
35      // 创建一个名为 pFcn 的函数指针（语法很丑陋）
36      int (*pFcn)(int, int) { nullptr };
37
38      // 设置 pFcn 指向用户所选的函数
39      switch (op)
40      {
41          case 0: pFcn = add; break;
42          case 1: pFcn = subtract; break;
43          case 2: pFcn = multiply; break;
44      }
45
46      // 调用 pFcn 指向的函数，x 与 y 作为入参，这里使用了 late binding
47      std::cout << "The answer is: " << pFcn(x, y) << '\n';
48
49      return 0;
50  }
```

Late binding 的性能稍微差一些这是因为它涉及到了额外一层的间接行为。使用 early binding，CPU 可以直接跳转到函数地址；使用 late binding，程序需要读取指针中的地址，接着再跳转去改地址。这里就牵扯到了额外的一步，因此稍微变慢了。然而 late binding 的优势是拥有了更多的灵活性，因为决定函数调用可以在运行时做。

18.6 The virtual table

为了实现虚函数，C++ 使用了一种被称为虚表的特别形式的 late binding。**虚表** virtual table 是一个检索函数的表，用于在动态/late binding 情况下解析函数调用。虚表有时会有别的名字，例如“vtable”, “virtual function table”, “virtual method table”，或者“dispatch table”。

虚表实际上非常简单，尽管用文字表达起来会有些复杂。首先，每个使用了虚函数的类（或是使用虚函数的派生类）都会被给予自身的虚表。这个表是一个由编译器在编译时设置好的静态数组。虚表包含了每个可以被类对象所调用的虚函数的入口。每个入口就是一个函数指针，其指向的是类可以访问到的最-派生的函数。

其次，编译器添加一个身为基类成员的隐藏指针，其可以称为 `*__vptr` 。当类对象被创建时，`*__vptr` 被自动设置，因此它可以指向该类的虚表。不同于 `*this` 指针，即实际上是一个函数参数用作于编译器解析自引用，`*__vptr` 是一个真实指针。结果而言，它使得每个类对象分配的大小都会大于一个指针。这同样也意味着 `*__vptr` 是会继承给派生类，这点非常重要。

现在来看一个例子：

```
1  class Base
2  {
```

```
public:
    virtual void function1() {};
    virtual void function2() {};
};

class D1: public Base
{
public:
    void function1() override {};
};

class D2: public Base
{
public:
    void function2() override {};
};
```

上述有 3 个类，编译器会设置 3 个虚表：`Base` ，`D1` 以及 `D2` 。
编译器同时添加了隐藏指针成员给使用虚函数的基类。尽管编译器自动化了这个步骤，在下一个例子中展示它添加在哪里：

```
class Base
{
public:
    VirtualTable* __vptr;
    virtual void function1() {};
    virtual void function2() {};
};

class D1: public Base
{
public:
    void function1() override {};
};

class D2: public Base
{
public:
    void function2() override {};
};
```

当一个类对象被创建，`*__vptr` 被设置指向该类的虚表。例如，当一个 Base 类型的对象被创建，`*__vptr` 被设置指向 `Base` 的虚表。当 `D1` 或 `D2` 类型的对象被构建，`*__vptr` 则分别被设置指向它们各自的虚表。
那么现在来看一下这些虚表是如何被填充的。因为这里只有两个虚函数，每个虚表都会有两个入口（一个是 `function1()` 另一个是 `function2()` ）。当这些虚表被填充，每个入口都会填充该类对象所能调用的最-派生的函数。

Base 对象的虚表很简单。Base 类型的对象仅可以访问 Base 的成员。Base 不能访问 D1 或 D2 的函数。结果而言，function1 的入口指向 Base::function1() 同时 function2 的入口指向 Base::function2() 。
D1 的虚表会稍微复杂一些。D1 类型的对象可以访问 D1 与 Base 的成员。然而，D1 拥有重写函数 function1() ，使得 D1::function1() 比 Base::function1() 更-派生一些。结果，function1 的入口指向了 D1::function1() 。D1 没有重写 function2() ，因此 function2 的入口指向 Base::function2() 。
D2 与 D1 类似，不再赘述。

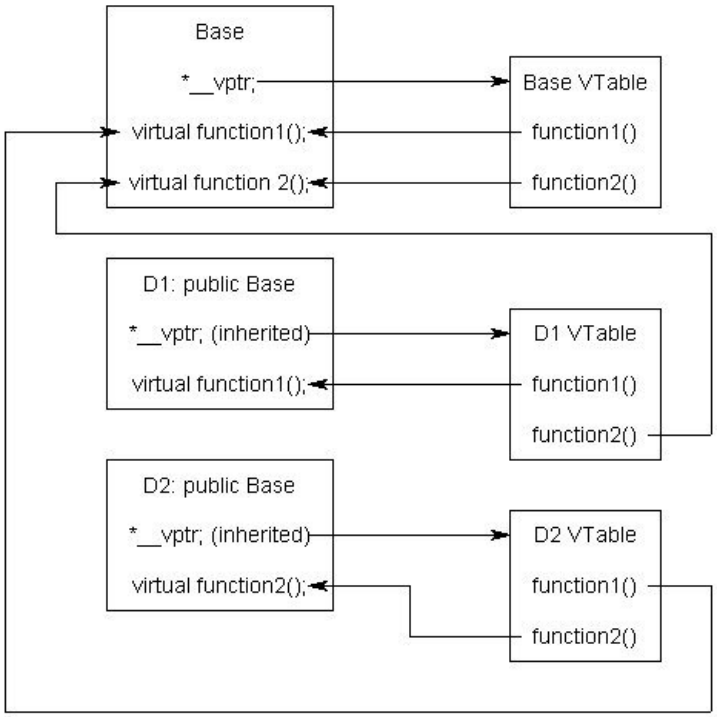

图 7: VTable

经过上图看起来很复杂，但是实际上非常简单：每个类中的 *__vptr 指向该类自身的虚表。虚表中的入口指向了该类对象可以调用的最-派生版本的函数。
考虑一下创建了 D1 类型对象会发生什么：

```
1 int main()
2 {
3     D1 d1;
4 }
```

由于 d1 是 D1 对象，其拥有的 *__vptr 设置到 D1 虚表。

现在设置一个 D1 的基类指针:

```
int main()
{
    D1 d1;
    Base* dPtr = &d1;

    return 0;
}
```

注意因为 dPtr 是一个基类指针，它仅指向了 d1 的 Base 部分。然而同样注意 __vptr 也在 Base 部分，因此 dPtr 可以访问到这个指针。最后就是注意 dPtr->__vptr 指向 D1 虚表！结果，即使 dPtr 是 Base 类型，它仍然能访问 D1 虚表（通过 __vptr ）。
那么当尝试调用 dPtr->function1() 会发生什么呢?

```
int main()
{
    D1 d1;
    Base* dPtr = &d1;
    dPtr->function1();

    return 0;
}
```

首先程序会察觉 function1() 是一个虚函数。其次，程序使用 dPtr->__vptr 来获取 D1 的虚表。第三，查看 D1 中需要调用哪个版本的 function1() 。这里是 D1::function1() 。因此 dPtr->function1() 解析成为 D1::function1() 。
但是有人可能会说“如果 dPtr 指向的是 Base 对象而不是 D1 对象，还会调用 D1::function1() 吗?”。答案是不。

```
int main()
{
    Base b;
    Base* bPtr = &b;
    bPtr->function1();

    return 0;
}
```

这个情况中，当 b 被创建，__vptr 指向的是 Base 的虚表，而不是 D1 的虚表。结果就是 bPtr->__vptr 同样指向 Base 的虚表，其 function1() 的入口指向 Base::function1() 。因此，bPtr->function1() 解析为 Base::function1() ，即对于一个 Base 对象而言，可以调用的最-派生的 function1() 版本。

18.7 Pure virtual functions, abstract base classes, and interface classes

迄今为止所有的虚函数都有函数体（定义）。然而 C++ 允许创建一种名为**纯虚函数** pure virtual function（或**抽象函数** abstract function）的没有函数体的特殊虚函数！纯虚函数类似于占位符一样，意为在派生类中被重新定义。

创建一个纯虚函数仅需要对函数赋值为 0。

```
class Base
{
public:
    const char* sayHi() const { return "Hi"; } // 普通的非虚函数

    virtual const char* getName() const { return "Base"; } // 普通的虚函数

    virtual int getValue() const = 0; // 纯虚函数

    int doSomething() = 0; // 编译错误：不能设置非虚函数为 0
};
```

在类中添加一个纯虚函数，等同于是在说，“由派生类来实现这个函数”。

使用纯虚函数有两个主要的结果：首先，任何带有纯虚函数的类会变成**抽象基类** abstract base class，这就意味着它不能被实例化！考虑以下代码：

```
int main()
{
    Base base; // 不能实例化一个抽象基类，但是为了演示，假设这样可以
    base.getValue(); // 那么这会是什么呢？

    return 0;
}
```

其次，任何派生类必须为此函数定义，否则该派生类同样也会被视为抽象基类。

纯虚函数案例

```
#include <string>

class Animal
{
protected:
    std::string m_name;

    // 使该构造函数为 protected，因为不希望其他人直接创建 Animal 对象，
    // 但是派生类还是可以使用它。
    Animal(const std::string& name)
        : m_name{ name }
    {
    }
```

```
14
15 public:
16     std::string getName() const { return m_name; }
17     virtual const char* speak() const { return "???"; }
18
19     virtual ~Animal() = default;
20 };
21
22 class Cat: public Animal
23 {
24 public:
25     Cat(const std::string& name)
26         : Animal{ name }
27     {
28     }
29
30     const char* speak() const override { return "Meow"; }
31 };
32
33 class Dog: public Animal
34 {
35 public:
36     Dog(const std::string& name)
37         : Animal{ name }
38     {
39     }
40
41     const char* speak() const override { return "Woof"; }
42 };
```

上述代码通过 protected 构造函数，阻止了 Animal 对象被直接创建。然而，还是可以创建派生类可以不用重新定义函数 `speak()` 。

例如：

```
1  #include <iostream>
2  #include <string>
3
4  class Cow : public Animal
5  {
6  public:
7      Cow(const std::string& name)
8          : Animal{ name }
9      {
10     }
11
12     // 这里忘记了去重新定义 speak
13 };
14
15 int main()
```

```
{
    Cow cow{"Betsy"};
    std::cout << cow.getName() << " says " << cow.speak() << '\n';

    return 0;
}
```

该问题的一个更好的解决方案是使用纯虚函数：

```
#include <string>

class Animal // 该 Animal 是一个抽象基类
{
protected:
    std::string m_name;

public:
    Animal(const std::string& name)
        : m_name{ name }
    {
    }

    const std::string& getName() const { return m_name; }
    virtual const char* speak() const = 0; // 注意 speak 现在是一个纯虚函数

    virtual ~Animal() = default;
};
```

这里有几点需要注意的。首先，`speak()` 现在是一个纯虚函数。这就意味着 `Animal` 现在是一个抽象基类，并不可再被实例化。结果就是不再需要让构造函数 protected（尽管也没别的影响）。其次，因为 `Cow` 类是由 `Animal` 派生而来，但是并没有定义 `Cow::speak()`，`Cow` 仍然是一个抽象基类。如果尝试编译代码：

```
#include <iostream>

class Cow: public Animal
{
public:
    Cow(const std::string& name)
        : Animal{ name }
    {
    }

    // 忘记重新定义 speak
};

int main()
{
    Cow cow{ "Betsy" };
```

```
    std::cout << cow.getName() << " says " << cow.speak() << '\n';

    return 0;
}
```

编译器会报错，因为 `Cow` 是一个抽象基类，不能直接实例化它。也就是告诉我们除非提供了 `speak()` 函数体给 `Cow`，否则不能实例化它。

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

class Animal
{
protected:
    std::string m_name;

public:
    Animal(const std::string& name)
        : m_name{ name }
    {
    }

    const std::string& getName() const { return m_name; }
    virtual const char* speak() const = 0;

    virtual ~Animal() = default;
};

class Cow: public Animal
{
public:
    Cow(const std::string& name)
        : Animal(name)
    {
    }

    const char* speak() const override { return "Moo"; }
};

int main()
{
    Cow cow{ "Betsy" };
    std::cout << cow.getName() << " says " << cow.speak() << '\n';

    return 0;
}
```

现在打印：

```
Betsy says Moo
```

带有定义的纯虚函数

结果就是可以创建带有定义的纯虚函数：

```
#include <string>

class Animal
{
protected:
    std::string m_name;

public:
    Animal(const std::string& name)
        : m_name{ name }
    {
    }

    std::string getName() { return m_name; }
    virtual const char* speak() const = 0;

    virtual ~Animal() = default;
};

const char* Animal::speak() const  // 尽管它有一个定义
{
    return "buzz";
}
```

上述例子中，`speak()` 仍然被视为纯虚函数因为“=0”（尽管被给与了一个定义），同时 `Animal` 仍然被视为抽象基类（因此还是不能被实例化）。任何继承了 `Animal` 的类需要提供自身的 `speak()` 定义，否则被视为抽象基类。

当为纯虚函数提供了定义，该定义必须是分开定义的（而不是内联的方式）。

这个范式非常的有用：当用户需要为函数提供一个默认实现，但是仍然可以强制派生类提供自身的实现。同时如果派生类愿意使用有基类提供的默认实现，可以简单的直接调用基类的实现。

例如：

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

class Animal // 抽象基类
{
protected:
    std::string m_name;

public:
    Animal(const std::string& name)
        : m_name(name)
    {
```

```
    }

    const std::string& getName() const { return m_name; }
    virtual const char* speak() const = 0; // 注意 speak 是一个纯虚函数

    virtual ~Animal() = default;
};

const char* Animal::speak() const
{
    return "buzz"; // 默认实现
}

class Dragonfly: public Animal
{

public:
    Dragonfly(const std::string& name)
        : Animal{name}
    {
    }

    const char* speak() const override// 该类不再是抽象基类，因为提供了函数定义
    {
        return Animal::speak(); // 使用 Animal 的默认实现
    }
};

int main()
{
    Dragonfly dfly{"Sally"};
    std::cout << dfly.getName() << " says " << dfly.speak() << '\n';

    return 0;
}
```

打印:

```
Sally says buzz
```

析构函数也可以是纯虚函数，但是必须给予定义，这样可以在派生对象销毁时调用它。

接口类

接口类 interface class 是一种没有成员变量的类，同时 * 所有 * 函数都是纯虚函数！换言之，这种类是纯粹的定义，并没有实际的实现。希望在派生类中必须定义所有功能时，并且将所有实现的细节完全交托给派生类时，接口非常的有用。

```
class IErrorLog
```

```
{
public:
    virtual bool openLog(const char* filename) = 0;

    virtual bool closeLog() = 0;

    virtual bool writeError(const char* errorMessage) = 0;

    virtual ~IErrorLog() {} // 使析构函数为虚函数，以防删除了 IErrorLog 指针
};
```

任何继承 IErrorLog 的类必须提供上述的三个函数实现才可以进行实例化。可以派生一个 FileErrorLog 的类，其中的 openLog() 打开一个文件，closeLog() 关闭文件，writeError() 向文件写入信息。也可以派生一个 ScreenErrorLog 类，其中的 openLog() 与 closeLog() 不做任何事，writeError() 向显示器打印信息。

假设需要使用日志，如果直接引入 FileErrorLog 或是 ScreenErrorLog，那么很容易就被卡住。例如入参的选择：

```
#include <cmath> // for sqrt()

double mySqrt(double value, FileErrorLog& log)
{
    if (value < 0.0)
    {
        log.writeError("Tried to take square root of value less than 0");
        return 0.0;
    }
    else
    {
        return std::sqrt(value);
    }
}
```

一个更好的做法是使用 IErrorLog：

```
#include <cmath> // for sqrt()
double mySqrt(double value, IErrorLog& log)
{
    if (value < 0.0)
    {
        log.writeError("Tried to take square root of value less than 0");
        return 0.0;
    }
    else
    {
        return std::sqrt(value);
    }
}
```

现在调用者可以传递任何符合 `IErrorLog` 接口的类。
不要忘记为接口类包含一个虚化的析构函数，这样的话如果一个该接口的指针被删除时，正确的析构函数才能够被调用。
接口类变得非常的流行，因为它们的使用简单，扩展方便，以及易于维护。实际上一些现代语言，例如 Java 和 C# 都添加了“interface”关键字，便于程序员直接定义接口类而不需要标记所有成员函数为抽象。
此外，尽管 Java（版本 8 之前）以及 C# 不让普通类使用若干继承，它们允许使用若干的接口。因为接口没有数据，也没有函数本体，它们避免了很多由多重继承带来的传统问题的同时提供了灵活性。

纯虚函数和虚表

抽象类仍然有虚表，它们仍然可以被使用，如果抽象类拥有一个指针或者引用。对于一个类而言虚表的入口是纯虚函数的情况下，会生成空指针或者指向普通打印错误的函数（有时这个函数名为 `__purecall` ）。

18.8 Virtual base classes

17.9 章节中提到了继承中的“菱形问题”，这章节将恢复这个问题的讨论。

```
#include <iostream>

class PoweredDevice
{
public:
    PoweredDevice(int power)
    {
    std::cout << "PoweredDevice: " << power << '\n';
    }
};

class Scanner: public PoweredDevice
{
public:
    Scanner(int scanner, int power)
        : PoweredDevice{ power }
    {
    std::cout << "Scanner: " << scanner << '\n';
    }
};

class Printer: public PoweredDevice
{
public:
```

```
    Printer(int printer, int power)
        : PoweredDevice{ power }
    {
    std::cout << "Printer: " << printer << '\n';
    }
};

class Copier: public Scanner, public Printer
{
public:
    Copier(int scanner, int printer, int power)
        : Scanner{ scanner, power }, Printer{ printer, power }
    {
    }
};
```

尽管预期继承的图像是这样:

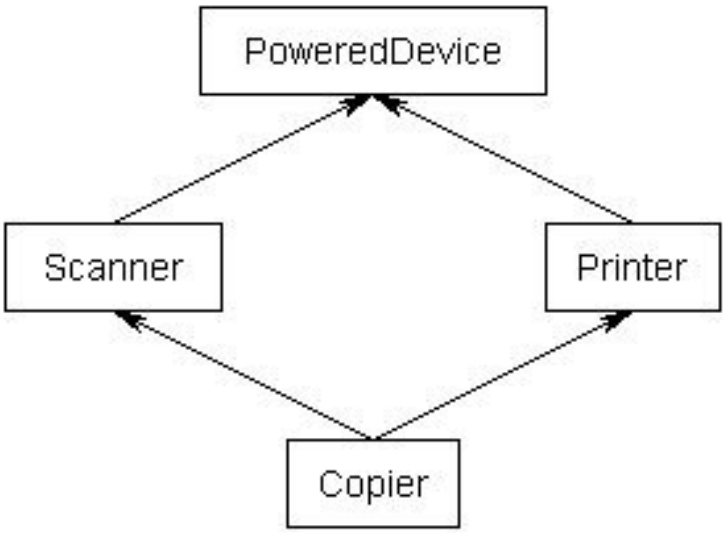

图 8: PoweredDevice recap

如果创建了一个 `Copier` 类的对象，默认会得到两份 `PoweredDevice` 类 – 一个给 `Printer` ，另一个给 `Scanner` 。

```
int main()
{
    Copier copier{ 1, 2, 3 };

    return 0;
}
```

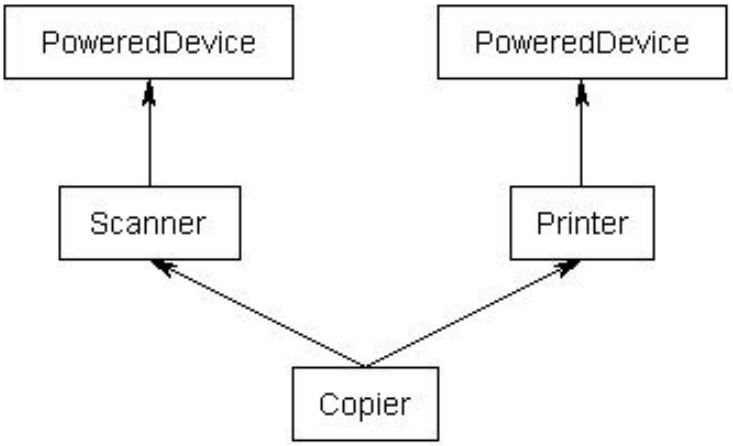

图 9: PoweredDevice2

打印:

```
PoweredDevice: 3
Scanner: 1
PoweredDevice: 3
Printer: 2
```

可以看到，`PoweredDevice` 被构造了两次。
虽然以上是经常需要的，但是其它时候可以只有一个副本的 `PoweredDevice` 被 `Printer` 与 `Scanner` 共享。

虚基类

为了共享一个基类，只需要简单的在派生类的继承列表前加上“virtual”关键字即可。这样创建的被称为**虚基类** virtual base class，意味着仅会有一个基对象。基对象被所有基础树种的对象共享，并且只会被构造一次。

```
class PoweredDevice
{
};

class Scanner: virtual public PoweredDevice
{
};

class Printer: virtual public PoweredDevice
{
};

```

```
class Copier: public Scanner, public Printer
{
};
```

现在可以创建一个 Copier 类对象，并且只会拥有一个被 Printer 与 Scanner 共享的 PoweredDevice 副本。

然而这会带来另一个问题：如果 Scanner 和 Printer 共享一个 PoweredDevice 基类，那么谁去负责构建它？答案是 Copier，其构造函数负责创建 PoweredDevice。因此，这是 Copier 被允许直接调用非直接父类的构造函数的时刻：

```
#include <iostream>

class PoweredDevice
{
public:
    PoweredDevice(int power)
    {
        std::cout << "PoweredDevice: " << power << '\n';
    }
};

class Scanner: virtual public PoweredDevice // 注意：PoweredDevice 现在是虚基类
{
public:
    Scanner(int scanner, int power)
        : PoweredDevice{ power } // 该行需要用于 Scanner 对象的创建，但是本例忽略
    {
        std::cout << "Scanner: " << scanner << '\n';
    }
};

class Printer: virtual public PoweredDevice // 注意：PoweredDevice 现在是虚基类
{
public:
    Printer(int printer, int power)
        : PoweredDevice{ power } // 该行需要用于 Printer 对象的创建，但是本例忽略
    {
        std::cout << "Printer: " << printer << '\n';
    }
};

class Copier: public Scanner, public Printer
{
public:
    Copier(int scanner, int printer, int power)
        : PoweredDevice{ power }, // PoweredDevice 在这里构造
        Scanner{ scanner, power }, Printer{ printer, power }
    {
```

```
39      }
40  };
41
42  int main()
43  {
44      Copier copier{ 1, 2, 3 };
45
46      return 0;
47  }
```

打印:

```
1  PoweredDevice: 3
2  Scanner: 1
3  Printer: 2
```

可以看到 PoweredDevice 仅被构建了一次。

有几个细节需要注意。

首先，虚基类总是创建在非虚基类之前，这样可以确保所有基类在派生类创建之前被创建。

其次，注意 Printer 与 Scanner 构造函数仍需要调用 PoweredDevice 构造函数。当创建 Copier 的实例时,这些构造函数会被简单的忽略掉,因为是 Copier 负责创建 PoweredDevice 而不是 Printer 或 Scanner 。然而，如果创建的是 Printer 或 Scanner 实例，这些构造函数还是会被调用，应用普通的继承规则。

第三，如果一个类继承了一个或多个带有虚化父类的类，那么最派生的类是负责构造虚基类的。本例中，Copier 继承了 Printer 与 Scanner ，它们都有一个 PoweredDevice 的虚基类。Copier 作为最派生的类，负责构造 PoweredDevice 。注意就算是单个继承的情况下，这也是成立的：如果 Copier 只继承了 Printer ，其也是继承了 PoweredDevice ，那么 Copier 仍然负责创建 PoweredDevice 。

第四，所有类继承了虚基类的都会拥有一个虚表，即使通常来说是不需要的，因此该类的实例的大小会大过一个指针。

因为 Printer 与 Scanner 都是虚继承了 PoweredDevice ，Copier 仅为 PoweredDevice 的一个子对象。Printer 与 Scanner 都需要知道如何寻找单个 PoweredDevice 的子对象，所以它们访问自身成员（因为毕竟它们是派生而来的）。这通常是靠一些虚表魔法来完成的（即本质上存储了每个子类相较于 PoweredDevice 子对象的偏移）。

18.9 Object slicing

现在来看之前的案例:

```
1  #include <iostream>
2
3  class Base
4  {
```

```
protected:
    int m_value{};

public:
    Base(int value)
        : m_value{ value }
    {
    }

    virtual const char* getName() const { return "Base"; }
    int getValue() const { return m_value; }
};

class Derived: public Base
{
public:
    Derived(int value)
        : Base{ value }
    {
    }

    const char* getName() const override { return "Derived"; }
};

int main()
{
    Derived derived{ 5 };
    std::cout << "derived is a " << derived.getName() << " and has value " << derived.getValue
    () << '\n';

    Base& ref{ derived };
    std::cout << "ref is a " << ref.getName() << " and has value " << ref.getValue() << '\n';

    Base* ptr{ &derived };
    std::cout << "ptr is a " << ptr->getName() << " and has value " << ptr->getValue() << '\n';

    return 0;
}
```

上述例子中，derived 的 ref 引用与 ptr 指针，拥有一个 Base 部分，以及一个 Derived 部分。因为 ref 与 ptr 的类型都是 Base，它们只能看到 derived 的 Base 部分 – 而 Derived 部分仍然存在，就是不能简单的通过 ref 与 ptr 来看见。然而通过虚函数的使用，我们可以访问最-派生的函数版本。因此，上述程序打印:

```
derived is a Derived and has value 5
ref is a Derived and has value 5
ptr is a Derived and has value 5
```

那么如果不设置 Base 引用或者指针给 Derived 对象，而是简单的 * 赋值 * 一个 Derived

对象给 Base 对象呢?

```
int main()
{
    Derived derived{ 5 };
    Base base{ derived }; // 这里会发生什么?
    std::cout << "base is a " << base.getName() << " and has value " << base.getValue() << '\n'
    ;

    return 0;
}
```

注意 derived 拥有 Base 部分以及 Derived 部分。当赋值一个 Derived 对象给 Base 对象时，只有 Base 部分会被拷贝，而 Derived 部分并没有被拷贝。上述例子中，base 获取了 derived 的 Base 部分的拷贝。Derived 部分实际上是被“切掉了 sliced off”。因此，将 Derived 类对象赋值给 Base 类对象被称为**对象切片** object slicing（或是简称切片）。
因为变量 base 没有 Derived 部分，因此 base.getName() 被解析为 Base::getName() 。
上述代码打印:

```
base is a Base and has value 5
```

然而不正确的使用切片可以导致若干不同方向的预想不到的结果。下面来看一下这些情况。

切片与函数

可能会认为上述的例子很蠢。毕竟为什么会有人赋值 derived 给 base 呢? 然而切片的发生更可能出现在函数上。
考虑以下函数:

```
void printName(const Base base) // 注意: base 是值传递而不是引用传递
{
    std::cout << "I am a " << base.getName() << '\n';
}
```

这个函数非常的简单，const 的 Base 对象作为入参被值传递。如果像以下这样调用函数:

```
int main()
{
    Derived d{ 5 };
    printName(d); // 啊，并没有察觉这是值传递

    return 0;
}
```

编写程序的时候很有可能会没有注意 base 是一个值入参而不是引用。因此，当调用 printName (d) 时，虽然预期的是 base.getName() 调用虚函数最终打印出“I am a Derived”，但是实际上并没有。

当然避免该问题的发生也很简单只需要使入参变为引用即可（这也是另一个传递类的引用而不是值的原因）。

```
void printName(const Base& base) // 注意: base 现在是引用传递
{
    std::cout << "I am a " << base.getName() << '\n';
}

int main()
{
    Derived d{ 5 };
    printName(d);

    return 0;
}
```

向量切片

另一个新手程序员可能经常会犯的错误是尝试在 `std::vector` 上实现多态。

```
#include <vector>

int main()
{
  std::vector<Base> v{};
  v.push_back(Base{ 5 }); // 添加 Base 对象至向量
  v.push_back(Derived{ 6 }); // 添加 Derived 对象至向量

  // 打印所有向量中的元素
  for (const auto& element : v)
    std::cout << "I am a " << element.getName() << " with value " << element.getValue() << '\n'
    ;

  return 0;
}
```

打印:

```
I am a Base with value 5
I am a Base with value 6
```

与之前例子相似，因为 std::vector 声明的类型是 `Base` ，当 `Derived(6)` 添加至向量时，其被切片了。
修复这个问题可能有点困难。很多新手程序员会尝试创建一个引用对象的向量，类似于:

```
std::vector<Base&> v{};
```

然而这并不能编译。向量中的元素必须是可赋值的，而引用并不能被赋值（仅可初始化）。
一种处理的方法是使用指针:

```cpp
#include <iostream>
#include <vector>

int main()
{
  std::vector<Base*> v{};

  Base b{ 5 }; // b 与 d 不能是匿名对象
  Derived d{ 6 };

  v.push_back(&b); // 添加 Base 对象至向量
  v.push_back(&d); // 添加 Derived 对象至向量

  // 打印所有向量中的元素
  for (const auto* element : v)
    std::cout << "I am a " << element->getName() << " with value " << element->getValue() << '\
    n';

  return 0;
}
```

打印:

```
I am a Base with value 5
I am a Derived with value 6
```

这就能工作了！其中有一些注意项。首先，**nullptr** 现在是一个合法的选项，这有可能并不是期望的。其次，需要处理指针语义，这会变得笨拙。不过这样的好处是，允许了动态内存分配的可能性，这对于有些对象可能会离开作用域来说很有用。

另一个选项是使用 **std::reference_wrapper** ，即一个模仿一个可重新赋值的引用的类:

```cpp
#include <functional> // std::reference_wrapper
#include <iostream>
#include <vector>

class Base
{
protected:
  int m_value{};

public:
  Base(int value)
    : m_value{ value }
  {
  }

  virtual const char* getName() const { return "Base"; }
  int getValue() const { return m_value; }
};
```

```
19
20 class Derived : public Base
21 {
22 public:
23   Derived(int value)
24     : Base{ value }
25   {
26   }
27
28   const char* getName() const override { return "Derived"; }
29 };
30
31 int main()
32 {
33   std::vector<std::reference_wrapper<Base>> v{}; // 可重新赋值的 Base 引用的向量
34
35   Base b{ 5 }; // b 与 d 不能是匿名对象
36   Derived d{ 6 };
37
38   v.push_back(b); // 添加 Base 对象至向量
39   v.push_back(d); // 添加 Derived 对象至向量
40
41   // 打印向量中的所有元素
42   // 使用 .get() 从 std::reference_wrapper 中获取元素
43   for (const auto& element : v) // 元素类型为 const std::reference_wrapper<Base>&
44     std::cout << "I am a " << element.get().getName() << " with value " << element.get().
      getValue() << '\n';
45
46 return 0;
47 }
```

Frankenobject

上述例子中讲述的都是由于派生类被切片了所带来的错误结果。现在来看一下另一个仍然在派生类中存在的危险案例!

```
1 int main()
2 {
3     Derived d1{ 5 };
4     Derived d2{ 6 };
5     Base& b{ d2 };
6
7     b = d1; // 这一行是有问题的
8
9     return 0;
10 }
```

前三行非常的直接：创建两个 `Derived` 对象，对第二个对象设置 `Base` 引用。

第四行出了问题。因为 `b` 指向 `d2`，又赋值了 `d1` 给 `b`，有人可能会认为 `d1` 将会给 `d2` 一个拷贝 – 如果 `b` 是 `Derived` 的情况并不是这样。如果 `b` 是 `Base`，C++ 中的操作符 `=` 默认不会提供类的虚化。因此，`d1` 只用 `Base` 部分被拷贝进了 `d2`。
结果就是 `d2` 有了 `d1` 的 `Base` 部分以及 `d2` 的 `Derived` 部分。在这个特定的例子中，这并没有问题（因为 `Derived` 类自身没有数据），但是大多数情况下，用户创建了一个 Frankenobject – 由若干对象的不同部分组成的对象。更坏的是，这没有简单的办法来阻止这样的事情发生。

总结

尽管 C++ 支持通过对象切片的方式将派生对象赋值给基类对象，通常而言，这并不能带来什么好处而都是麻烦，因此应该尽量避免切片。请确保所有函数入参是引用（或者指针），同时在处理派生类时尽量避免任何值传递类型。

18.10 Dynamic casting

8.5 章节讲到了显式类型转换（casting）以及 static_cast，本章节继续测试另一种类型的 cast：dynamic_cast。

dynamic_cast 的需求

在处理多态时，经常会遇到指针指向的是基类，但是又希望能访问派生类中的信息。
考虑以下代码（稍微有些做作）：

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

class Base
{
protected:
  int m_value{};

public:
  Base(int value)
    : m_value{value}
  {
  }

  virtual ~Base() = default;
};

class Derived : public Base
{
```

```
protected:
  std::string m_name{};

public:
  Derived(int value, const std::string& name)
    : Base{value}, m_name{name}
  {
  }

  const std::string& getName() const { return m_name; }
};

Base* getObject(bool returnDerived)
{
  if (returnDerived)
    return new Derived{1, "Apple"};
  else
    return new Base{2};
}

int main()
{
  Base* b{ getObject(true) };

  // 在仅有基类指针的情况下，如何打印派生类对象的名字？

  delete b;

  return 0;
}
```

这个程序中，函数 `getObject()` 总是返回一个 `Base` 指针，但是该指针很有可能本身是指向 `Base` 或 `Derived` 的对象的。这个例子中的指针指向的是 `Derived` 的对象，那么该如何调用 `Derived::getName()` 呢？

一种方法是添加名为 `getName()` 的虚函数至 `Base` 中（这样可以通过 `Base` 的指针/引用来进行调用，同时使其动态解析成 `Derived::getName()`）。但是如果通过 `Base` 指针/引用而实际上其指向的是 `Base` 对象时，调用这个函数返回的是什么呢？这个时候其实没有任何值具有意义。此外，有些应该只在派生类中考虑的东西将会污染 `Base` 类。

众所周知 C++ 将隐式转换 `Derived` 指针称为 `Base` 指针。这个过程有时被称为**向上转型** upcasting。然而是否有一种方法使得 `Base` 指针转换回 `Derived` 指针呢？这样就可以使用该指针直接调用 `Derived::getName()`，且不需要担心虚函数的解析了。

dynamic_cast

C++ 提供了一种名为 **dynamic_cast** 的转型操作符以完成上述目的。尽管动态转型只有很少的功能，现在位置最常用的地方是转化基类指针称为派生类指针。这个过程被称为**向下转型** downcasting。

dynamic_cast 的使用与 static_cast 类似。

```
int main()
{
  Base* b{ getObject(true) };

  Derived* d{ dynamic_cast<Derived*>(b) }; // 使用动态转型转换 Base 指针成为 Derived 指针

  std::cout << "The name of the Derived is: " << d->getName() << '\n';

  delete b;

  return 0;
}
```

打印:

```
The name of the Derived is: Apple
```

dynamic_cast 失败

上述例子可以工作是因为 `b` 实际上指向的是 `Derived` 对象，因此转换 `b` 成为 `Derived` 指针可以成功。

然而我们这里下定了一个危险的假设：`b` 指向的是 `Derived` 对象，那么如果 `b` 指向的不是 `Derived` 对象呢？通过改变 `getObject()` 的入参从 true 变为 false 可以很容易的测试出来。这种情况下，`getObject()` 将返回一个指向 `Base` 对象的 `Base` 指针。这时尝试 dynamic_cast 转为 `Derived` 将会失败，因为转换是不可能的。

如果 dynamic_cast 失败了，装换的结果会是一个空指针。

因为没有对结果进行空指针检查，访问 `d->getName()` 时将会对空指针进行解引用，从而导致未定义行为（可能是程序崩溃）。

为了让程序安全，可以确保 dynamic_cast 的结果是真的成功:

```
int main()
{
  Base* b{ getObject(true) };

  Derived* d{ dynamic_cast<Derived*>(b) }; // 使用动态转型转换 Base 指针成为 Derived 指针

  if (d) // 确保 d 非空
    std::cout << "The name of the Derived is: " << d->getName() << '\n';
```

```
9
10    delete b;
11
12    return 0;
13  }
```

规则：总是确保动态转型之后进行空指针检查。
注意因为 dynamic_cast 在运行时做了一些连贯性检测（用来确保可以进行转换），使用 dynamic_cast 会带来性能下降的惩罚。
同时注意有以下几种情况使用 dynamic_cast 进行向下转型不能工作：

1. 带有 protected 或 private 的继承
2. 没有声明或者继承任何虚函数的类（因为没有虚表）
3. 一些特定的情况涉及了虚基类（详见这里）

通过 static_cast 进行向下转型

事实上向下转型同样也可以通过 static_cast 完成。这里主要的区别是 static_cast 没有运行时检查来确保转型是否有意义。这就让 static_cast 更快了，但是也更危险了。如果转换一个 `Base*` 成为 `Derived*`，可以“成功”即使 `Base` 指针并没有指向 `Derived` 对象。当尝试访问 `Derived` 指针时将会导致未定义行为（因为实际上指向的是 `Base` 对象）。
如果对于向下转型的指针完全有把握成功，那么使用 static_cast 是可以的。其中一种保证指向的对象类型的方式，是使用虚函数。这里是一个方法（并不是那么好）：

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

// Class 标识符
enum class ClassID
{
  base,
  derived
  // 其它可以之后在这里添加
};

class Base
{
protected:
int m_value{};

public:
  Base(int value)
    : m_value{value}
  {
```

```cpp
  }

  virtual ~Base() = default;
  virtual ClassID getClassID() const { return ClassID::base; }
};

class Derived : public Base
{
protected:
  std::string m_name{};

public:
  Derived(int value, const std::string& name)
    : Base{value}, m_name{name}
  {
  }

  const std::string& getName() const { return m_name; }
  virtual ClassID getClassID() const { return ClassID::derived; }
};

Base* getObject(bool bReturnDerived)
{
  if (bReturnDerived)
    return new Derived{1, "Apple"};
  else
    return new Base{2};
}

int main()
{
  Base* b{ getObject(true) };

  if (b->getClassID() == ClassID::derived)
  {
    // 已经证明了 b 是指向 Derived 对象的，因此这里总是会成功
    Derived* d{ static_cast<Derived*>(b) };
    std::cout << "The name of the Derived is: " << d->getName() << '\n';
  }

  delete b;

  return 0;
}
```

但是如果通过这些麻烦的步骤来实现（花费调用虚函数的成本，以及处理整个过程），或许直接使用 dynamic_cast 更好。

dynamic_cast 与引用

尽管上述所有的例子展示的都是指针的动态转型（也是最常见的），dynamic_cast 同样也可以作用于引用。

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

class Base
{
protected:
  int m_value;

public:
  Base(int value)
    : m_value{value}
  {
  }

  virtual ~Base() = default;
};

class Derived : public Base
{
protected:
  std::string m_name;

public:
  Derived(int value, const std::string& name)
    : Base{value}, m_name{name}
  {
  }

  const std::string& getName() const { return m_name; }
};

int main()
{
  Derived apple{1, "Apple"};
  Base& b{ apple };
  Derived& d{ dynamic_cast<Derived&>(b) }; // 动态转型，使用引用而不是指针

  std::cout << "The name of the Derived is: " << d.getName() << '\n'; // 可以通过 d 来访问
    Derived::getName

  return 0;
}
```

由于 C++ 并没有一个“空引用”，失败的时候 dynamic_cast 并不能返回一个空引用。反而如果

dynamic_cast 对于引用失败时，`std::bad_cast` 的异常会被抛出。

dynamic_cast vs static_cast

新手程序员会疑惑何时使用 dynamic_cast 或是 static_cast。答案非常简单：使用 static_cast 除非使用向下转型，这种情况下 dynamic_cast 会是更好的选择。然而同样也需要考虑避免使用转型，而是去使用虚函数。

向下转型 vs 虚函数

有一些开发者认为 dynamic_cast 是邪恶的，并且是坏的类设计。相反这些程序员会建议使用虚函数。
通常来说，使用虚函数 * 应该 * 是比向下转型更为推荐的。然而有时候向下转型却是更好的选择：

- 当不可以修改基类添加一个虚函数时（例如因为基类是标准库的一部分）
- 当需要访问一些只有派生类的东西（例如访问仅存在于派生类中的函数）
- 当添加虚函数给基类没有任何意义时（例如没有合适的值作为基类的返回）。这里使用存虚函数可能是一个选项，只要不需要实例化基类。

dynamic_cast 与 RTTI 的警告

运行时类型信息（Run-time type information (RTTI)）是 C++ 的一个特征，用来在运行时暴露对象的数据类型。这个能力是由 dynamic_cast 完成的。因为 RTTI 拥有很大的性能成本，一些编译器允许用户关闭 RTTI 来进行优化。无需多言，如果这么做，dynamic_cast 不能正确工作。

18.11 Printing inherited classes using operator«

考虑下面使用虚函数的程序：

```
1  #include <iostream>
2
3  class Base
4  {
5  public:
6    virtual void print() const { std::cout << "Base";  }
7  };
8
9  class Derived : public Base
10 {
```

```
public:
  void print() const override { std::cout << "Derived"; }
};

int main()
{
  Derived d{};
  Base& b{ d };
  b.print(); // 将调用 Derived::print()

  return 0;
}
```

虽然像是上面这样调用成员函数是没问题的，但是这种风格的函数并不能与 `std::cout` 混用：

```
#include <iostream>

int main()
{
  Derived d{};
  Base& b{ d };

  std::cout << "b is a ";
  b.print(); // 凌乱，调用该函数打破了打印声明
  std::cout << '\n';

  return 0;
}
```

本章将会详解如何使用继承来为类进行重写操作符 `<<`，使得可以使用 `<<` 如：

```
std::cout << "b is a " << b << '\n'; // 更好了
```

操作符 « 的挑战

用典型的方法来重载操作符 «：

```
#include <iostream>

class Base
{
public:
  virtual void print() const { std::cout << "Base"; }

  friend std::ostream& operator<<(std::ostream& out, const Base& b)
  {
    out << "Base";
    return out;
  }
};
```

```
14
15 class Derived : public Base
16 {
17 public:
18   void print() const override { std::cout << "Derived"; }
19
20   friend std::ostream& operator<<(std::ostream& out, const Derived& d)
21   {
22     out << "Derived";
23     return out;
24   }
25 };
26
27 int main()
28 {
29   Base b{};
30   std::cout << b << '\n';
31
32   Derived d{};
33   std::cout << d << '\n';
34
35   return 0;
36 }
```

这里不需要虚函数解析，程序正如预期，打印：

```
1 Base
2 Derived
```

现在来看看以下的 `main()` 函数：

```
1 int main()
2 {
3     Derived d{};
4     Base& bref{ d };
5     std::cout << bref << '\n';
6
7     return 0;
8 }
```

打印：

```
1 Base
```

这并不是预期的结果，这是因为操作符 `<<` 处理的 `Base` 对象并不是虚化的，因此 `std::cout << bref` 调用的是 `Base` 对象的 `<<` 而不是 `Derived` 对象的。

能让操作符 `<<` 虚化吗?

答案是不。这里有一系列的原因。

首先，仅有成员函数可以被虚化 – 这很合理，因为只有类可以继承其他的类，并且没有办法在类外部去重写一个函数（可以重载非成员函数，而不是重写它们）。由于通常实现的操作符 « 是作为友元，友元并不被视为成员函数，那么友元的操作符 « 并不没有被虚化的资格。
其次，就算是可以虚化操作符 «，问题在于 `Base::operator<<` 的函数入参以及 `Derived::operator<<` 是不同的（`Base` 的入参是 `Base`，`Derived` 的则是 `Derived`）。因此 `Derived` 并不能视为重写了 `Base`，因此也不没有虚化函数解析的资格。

解决方案

答案其实惊人的简单。
首先，与通常一样设置操作符 « 为基类的友元。但是不让操作符 « 打印其自身，而是委派这个责任给与一个普通的成员函数，即可以被虚化！

```
#include <iostream>

class Base
{
public:
  // 此处为重载操作符<<
  friend std::ostream& operator<<(std::ostream& out, const Base& b)
  {
    // 委派打印功能给函数 print()
    return b.print(out);
  }

  // 依赖成员函数 print() 来完成真正的打印工作。
  // 因为 print() 是一个普通的成员函数，它可以被虚化
  virtual std::ostream& print(std::ostream& out) const
  {
    out << "Base";
    return out;
  }
};

class Derived : public Base
{
public:
  // 这里是 print() 函数的重写，用来处理 Derived 的情况
  std::ostream& print(std::ostream& out) const override
  {
    out << "Derived";
    return out;
  }
};

int main()
```

```
{
  Base b{};
  std::cout << b << '\n';

  Derived d{};
  std::cout << d << '\n'; // 注意这可以工作，即使没有操作符<< 显示的处理 Derived 对象

  Base& bref{ d };
  std::cout << bref << '\n';

  return 0;
}
```

上述程序打印:

```
Base
Derived
Derived
```

现在来看一下细节。

首先，在 Base 里，调用操作符 « 后调用虚函数 print() 。因为引用入参指向的是 Base 对象，b.print() 解析为 Base::print() ，即可以打印。这里没有特别的。

在 Derived 里，编译器首先检查操作符 « 是否获得一个 Derived 对象。而并没有这样的重载，因为并没有被定义。接着编译器检查操作符 « 是否获得一个 Base 对象。有，因此编译器完成嘞一个隐式的向上转型，使得 Derived 对象成为一个 Base& ，并调用函数（这里可以人工进行向上转型，但是编译器可以完成）。该函数调用虚函数 print() ，并被解析为 Derived::print() 。

注意这里不需要在每个派生类中定义操作符 «! Base 类中定义的完全可以胜任任何 Base 对象以及其派生的类!

在第三个打印过程中混合了前两种。首先，编译器通过操作符 « 匹配变量 bref 从未获取 Base 。其调用了虚函数 print() 。因为 Base 引用实际上指向的是 Derived 对象，因此被解析为 Derived::print() ，正如预期。

19 Templates and Classes

19.1 Template classes

模版与容器类

16.6 章节中学习到了如何使用组合 composition 来实现可以包含若干其他类实例的容器类。这里是一个简化版的代码:

```
#ifndef INTARRAY_H
#define INTARRAY_H

#include <cassert>

class IntArray
{
private:
    int m_length{};
    int* m_data{};

public:

    IntArray(int length)
    {
        assert(length > 0);
        m_data = new int[length]{};
        m_length = length;
    }

    // 不希望 IntArray 的拷贝允许被创建
    IntArray(const IntArray&) = delete;
    IntArray& operator=(const IntArray&) = delete;

    ~IntArray()
    {
        delete[] m_data;
    }

    void erase()
    {
        delete[] m_data;
        // 这里需要确保设置 m_data 为空指针，否则将会仍然指向已释放的内存!
        m_data = nullptr;
        m_length = 0;
    }

    int& operator[](int index)
    {
        assert(index >= 0 && index < m_length);
```

```
        return m_data[index];
    }

    int getLength() const { return m_length; }
};

#endif
```

那么如果是想要创建一个 double 的数组呢？与 `IntArray` 相比，仅有实质性不同的在于数据类型不同。创建一个模板类类似于创建模板函数。

```
#ifndef ARRAY_H
#define ARRAY_H

#include <cassert>

template <typename T> // 新增
class Array
{
private:
    int m_length{};
    T* m_data{}; // 修改类型为 T

public:

    Array(int length)
    {
        assert(length > 0);
        m_data = new T[length]{}; // 分配类型为 T 的对象的数组
        m_length = length;
    }

    Array(const Array&) = delete;
    Array& operator=(const Array&) = delete;

    ~Array()
    {
        delete[] m_data;
    }

    void erase()
    {
        delete[] m_data;
        m_data = nullptr;
        m_length = 0;
    }

    T& operator[](int index) // 现在返回的是 T&
    {
```

```
        assert(index >= 0 && index < m_length);
        return m_data[index];
    }

    // 模板化的 getLength() 函数在下方定义
    int getLength() const;
};

// 成员函数定义在类的外部，需要它们自己的模版声明
template <typename T>
int Array<T>::getLength() const // 注意类名为 Array<T>，而不是 Array
{
  return m_length;
}

#endif
```

可以看到上述代码几乎与 `IntArray` 相同，除了添加了模版声明以外，以及使用 T 更换了数据类型。

注意除此之外还定义了 `getLength()` 函数在类声明之外。这并不是必须的，但是新手程序员通常会卡在这里的语法上，因此这个案例具有指导意义。每个模板成员函数定义在类声明之外就需要其自身的模版声明。同时，注意模版数组的名称现在是 `Array<T>`，而不是 `Array`。

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

int main()
{
  Array<int> intArray { 12 };
  Array<double> doubleArray { 12 };

  for (int count{ 0 }; count < intArray.getLength(); ++count)
  {
    intArray[count] = count;
    doubleArray[count] = count + 0.5;
  }

  for (int count{ intArray.getLength() - 1 }; count >= 0; --count)
    std::cout << intArray[count] << '\t' << doubleArray[count] << '\n';

  return 0;
}
```

打印:

```
11      11.5
10      10.5
9       9.5
8       8.5
```

```
7       7.5
6       6.5
5       5.5
4       4.5
3       3.5
2       2.5
1       1.5
0       0.5
```

模板类的实例化与模板函数一样 – 编译器拓印一份所需的拷贝，将模板参数替换为用户所需的真实数据类型，接着编译该副本。如果没有使用模板类，编译器甚至不会编译它。
模板类用来实现容器类是理想的，因为它高度期望容器可以对不同类型的数据有效，同样的模板也避免了大量重复代码。尽管语法很挫，同时报错信息可能会很隐晦，然而模板类是 C++ 中真正最好的最有用的特性。

标准库中的模板类

现在已经覆盖了模板类，那么理解 `std::vector<int>` 的真正意义就不难了 – 它实际上就是一个模板类，二 `int` 则是模板的类型参数！标准库中存在着大量的预定义的模板类可供使用。之后的章节会覆盖到它们。

拆分模板类

一个模板既不是类也不是函数 – 它是拓印用于创建类或者函数的。因此，它的工作方式与普通的类或者函数不一样。大多数情况下，这并不是什么问题。然而有一个地方是开发者通常容易出错的。
非模板类时，通常的过程是放置类定义与头文件中，其成员函数定义放在同名的源文件中。这样的方式，类的源代码会以分隔项目文件的方式被编译。然而使用模板时，这样并不能工作。考虑下面的代码：
Array.h:

```
#ifndef ARRAY_H
#define ARRAY_H

#include <cassert>

template <typename T>
class Array
{
private:
    int m_length{};
    T* m_data{};

public:
```

```

    Array(int length)
    {
        assert(length > 0);
        m_data = new T[length]{};
        m_length = length;
    }

    Array(const Array&) = delete;
    Array& operator=(const Array&) = delete;

    ~Array()
    {
        delete[] m_data;
    }

    void erase()
    {
        delete[] m_data;

        m_data = nullptr;
        m_length = 0;
    }

    T& operator[](int index)
    {
        assert(index >= 0 && index < m_length);
        return m_data[index];
    }

    int getLength() const;
};

# endif
```

Array.cpp:

```
#include "Array.h"

template <typename T>
int Array<T>::getLength() const // 注意类名称为 Array<T>, 而不是 Array
{
  return m_length;
}
```

main.cpp:

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

```

```
int main()
{
  Array<int> intArray(12);
  Array<double> doubleArray(12);

  for (int count{ 0 }; count < intArray.getLength(); ++count)
  {
    intArray[count] = count;
    doubleArray[count] = count + 0.5;
  }

  for (int count{ intArray.getLength() - 1 }; count >= 0; --count)
    std::cout << intArray[count] << '\t' << doubleArray[count] << '\n';

  return 0;
}
```

上述程序可以被编译但是会有一个 linker 错误：

```
unresolved external symbol "public: int __thiscall Array<int>::getLength(void)" (?GetLength@?\
    $Array@H@@QAEHXZ)
```

为了让编译器使用模板，编译器必须同时看到模板定义（不仅只是声明）以及用于实例化模板的模板类型。同时不要忘记 C++ 是单独的编译文件。当 `Array.h` 头文件被 main 引入，模板类的定义被拷贝进 `main.cpp` 。当编译器如果看到两个模板实例，`Array<int>` 与 `Array<double>` 时，编译器会实例化它们，并将它们编译成 `main.cpp` 中的一部分。然而当编译器分开编译 `Array.cpp` 时，它并不知道需要的是 `Array<int>` 与 `Array<double>` ，因此模板函数永远不会被实例化。因此获得了 linker 错误，这是因为编译器没有发现 `Array<int>::getLength()` 与 `Array<double>::getLength()` 的定义。

有一些方法可以绕过这个问题。

最简单的方法就是把所有模板类的代码放到头文件中（这个案例中就是将 `Array.cpp` 的内容放进 `Array.h` 中位于类的下方）。这样当 #include 头文件时，所有模板代码都会在同一个地方。这么做的好处就是简单。这么做的坏处是如果模板类在很多地方被使用的时候，将会得到大量模板类的本地副本，这增加了编译时间以及 link 次数（linker 应该会移除重复的定义，因此并不会使可执行文件膨胀起来）。这是推荐的解决方案，直到编译器或 link 次数开始成为一个问题时。

如果将 `Array.cpp` 代码放进 `Array.h` 头文件使得头文件非常大且凌乱，另一种做法是移动 `Array.cpp` 至新的名为 `Array.inl` 文件中（.inl 意为内联 inline），接着将 `Array.inl` 包含在 `Array.h` 头文件的尾部（头文件保护符之内）。这将返回与将所有代码放入头文件中，产生同样的结果，并给代码带来了更好的管理。

小贴士：如果使用的是.inl 方法并触发关于重复定义的编译错误时，编译器大概率是将.inl 文件视为了代码文件而成为了项目的一部分。这个结果是.inl 被编译了两次：第一次是编译了.inl，

还有一次是当.cpp 文件引用.inl 文件时被编译。如果这种情况发生了，那么则需要从 build 中排除.inl 文件。这通常是可以由编辑器中对该文件点击右键并选择其属性来完成的。
另一种方法是使用三文件处理法。模板类定义放在头文件，模板类成员函数放在代码文件，接着添加第三个文件，包含所有所需的实例化的类：
templates.cpp:

```
// 确保所有 Array 模板定义都可以被看见
#include "Array.h"
#include "Array.cpp" // 这里破坏了最佳实践，但是仅在此一处

// 在这里 #include 其他所有需要的 .h 与 .cpp 模板定义

template class Array<int>;    // 显式实例化模板 Array<int>
template class Array<double>; // 显式实例化模板 Array<double>

// 这里实例化其他模板
```

这个方法有可能更高效（取决于编译器和 linker 是如何处理模板与重复定义的），但是需要为每个项目都维护一个 `templates.cpp` 文件。

19.2 Template non-type parameters

前一节学到了如何使用模板类型变量来创建类型独立的函数与类。模板类型参数是一个占位类型用于替换传入的类型参数。
然而模板类型参数并不是唯一可用的模板参数。模板类与函数可以使用其他类型的模板参数，被称为非类型参数。

非类型参数

一个模板非类型参数是一种参数类型是预定义的并且由 constexpr 值传递给入参的模板参数。
一个非类型参数可以是以下任意一个类型：

- 整型
- 枚举类型
- 一个类对象的指针或引用
- 一个函数的指针或引用
- 一个类成员函数的指针或引用
- sts::nullptr_t

- 浮点类型（C++20 起）

下面的例子中，创建了一个非动态（static）数组类，同时用到了类型参数与非类型参数。类型参数控制静态数组的数据类型，而整型的非类型参数控制静态数组的大小。

```cpp
#include <iostream>

template <typename T, int size> // size 是一个整型的非类型参数
class StaticArray
{
private:
    // 非类型参数控制数组大小
    T m_array[size] {};

public:
    T* getArray();

    T& operator[](int index)
    {
        return m_array[index];
    }
};

// 展示带有非类型参数的类成员函数是如何定义在类定义外部的
template <typename T, int size>
T* StaticArray<T, size>::getArray()
{
    return m_array;
}

int main()
{
    // 声明整型数组，大小为 12
    StaticArray<int, 12> intArray;

    // 顺序填数，倒序打印
    for (int count { 0 }; count < 12; ++count)
        intArray[count] = count;

    for (int count { 11 }; count >= 0; --count)
        std::cout << intArray[count] << ' ';
    std::cout << '\n';

    // 声明浮点型数组，大小为 4
    StaticArray<double, 4> doubleArray;

    for (int count { 0 }; count < 4; ++count)
        doubleArray[count] = 4.4 + 0.1 * count;

```

```
    for (int count { 0 }; count < 4; ++count)
        std::cout << doubleArray[count] << ' ';

    return 0;
}
```

上述代码中一个值得注意的点就是没有动态分配 `m_array` 成员变量！这是因为任何给定的 `StaticArray` 类，`size` 必须是 constexpr。例如，如果实例化一个 `StaticArray<int, 12>`，编译器替换 `size` 为 12。因此 `m_array` 的类型是 `int[12]`，即可被静态分配。

19.3 Function template specialization

当用给定类型实例化一个函数模板时，编译器拓印模板函数的副本并用真实类型替换掉模板类型参数。这意味着对于每个实例化类型（使用不同的类型）特定的函数将拥有同样的实现细节。大多数时候这正是用户预期的，而在偶尔一些情况下，根据特定数据类型实现一个差异化的模板函数却非常有用。

模板特化就是一种手段。

现在来看一个非常简单的模板类：

```
#include <iostream>

template <typename T>
class Storage
{
private:
    T m_value {};
public:
    Storage(T value)
      : m_value { value }
    {
    }

    void print()
    {
        std::cout << m_value << '\n';
    }
};
```

上述代码可以服务于很多种数据类型：

```
int main()
{
    // 定义一些存储单元
    Storage<int> nValue { 5 };
    Storage<double> dValue { 6.7 };

    // 打印一些值
```

```
    nValue.print();
    dValue.print();
}
```

现在希望浮点值（且仅浮点值）输出的是科学计数法，这个时候可以使用**函数模板特化** function template specialization （有时被称为完全或显式函数模板特化）来为浮点类型创建一个特化版本的 `print()` 函数。做法相当的简单：仅需定义一个特化函数（如果函数是成员函数，在类定义外同样可以定义）替换模板类型为特定希望重新定义的类型。这里是一个浮点特化的 `print()` 函数：

```
template <>
void Storage<double>::print()
{
    std::cout << std::scientific << m_value << '\n';
}
```

当编译器实例化 `Storage<double>::print()` 时，会看到已经显式定义的函数，并使用它而不是拓印通用模板类。

模板 `<>` 告诉编译器这是一个模板函数，但是没有模板变量（这个例子中已经显式的指定了类型）。

另一个例子

现在来看另一个模板特化的案例。如果尝试使用 `const char*` 作为模板 `Storage` 类的类型。

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

template <typename T>
class Storage
{
private:
    T m_value {};
public:
    Storage(T value)
      : m_value { value }
    {
    }

    void print()
    {
        std::cout << m_value << '\n';
    }
};

int main()
{
```

```
    // 动态分配临时字符串
    std::string s;

    // 询问名称
    std::cout << "Enter your name: ";
    std::cin >> s;

    // 存储名称
    Storage<char*> storage(s.data());

    storage.print(); // 打印名称

    s.clear(); // 清除 std::string

    storage.print(); // 打印空
}
```

第二次的 `storage.print()` 什么都没有打印！那这是为什么呢?
当 `Storage` 为 `char*` 类型实例化时，`Storage<char*>` 的构造函数类似于:

```
template <>
Storage<char*>::Storage(char* value)
      : m_value { value }
{
}
```

换言之，这里完成了一个指针赋值（浅拷贝）！结果 `m_value` 指向了与 `string` 相同的地址。当 `main()` 中删除 `string` 时，也是删除了 `m_value` 指向的值！因此尝试打印值的时候得到的是垃圾。
幸运的是，可以通过模板特化来解决这个问题。相比于使用一个指针拷贝，更加希望的是构造函数获取的是输入字符串的拷贝。

```
template <>
Storage<char*>::Storage(char* const value)
{
    if (!value)
        return;

    // 弄清楚字符串的长度
    int length { 0 };
    while (value[length] != '\0')
        ++length;
    ++length; // +1 用于解释 null 终止符

    // 分配内存用于存储值字符串
    m_value = new char[length];

    // 拷贝实际值字符串给 m_value 刚分配好的内存
```

```
    for (int count=0; count < length; ++count)
        m_value[count] = value[count];
}
```

现在当分配 Storage<char*> 类型变量时，这个构造函数将使用非默认版本的。接着 m_value 将获取字符串的拷贝。因此当删除字符串时，m_value 并不会受到影响。
然而这个类现在对于 char* 类型会产生内存泄露，因为 m_value 在 Storage 离开作用域时并不会被删除。如果想着可以由指定的 Storage<char*> 析构函数来解决：

```
template <>
Storage<char*>::~Storage()
{
    delete[] m_value;
}
```

这样的话当 Storage<char*> 类型的变量离开作用域时，由构造函数产生的内存分配会被特化的析构函数删除。
然而，可能出乎意料，上述特化的析构函数并不会通过编译。这是因为特化函数必须特化一个显式函数（而不是编译器提供的默认函数）。由于并没有为 Storage<T> 定义一个析构函数，编译器提供了一个默认析构函数，因此不能对其进行特化。为了解决这个问题，就必须为 Storage<T> 定义一个结构函数。

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

template <typename T>
class Storage
{
private:
    T m_value{};
public:
    Storage(T value)
        : m_value{ value }
    {
    }
    ~Storage() {}; // 为了特化，需要一个显式定义的析构函数

    void print()
    {
        std::cout << m_value << '\n';
    }
};

template <>
Storage<char*>::Storage(char* const value)
{
    if (!value)
```

```
26          return;
27
28      // 弄清楚字符串的长度
29      int length{ 0 };
30      while (value[length] != '\0')
31          ++length;
32      ++length; // +1 用于解释 null 终止符
33
34      // 分配内存用于存储值字符串
35      m_value = new char[length];
36
37      // 拷贝实际值字符串给 m_value 刚分配好的内存
38      for (int count = 0; count < length; ++count)
39          m_value[count] = value[count];
40  }
41
42  template <>
43  Storage<char*>::~Storage()
44  {
45      delete[] m_value;
46  }
47
48  int main()
49  {
50      // 动态分配一个临时的字符串
51      std::string s;
52
53      // 询问名称
54      std::cout << "Enter your name: ";
55      std::cin >> s;
56
57      // 保存名称
58      Storage<char*> storage(s.data());
59
60      storage.print(); // 打印名称
61
62      s.clear(); // 清除 std::string
63
64      storage.print(); // 打印名称
65  }
```

尽管上述代码中包含的都是成员函数，不过还是可以用同样的方式特化非成员函数。

19.4 Class template specialization

上一节讲述了如何特化函数来为特定的数据类型提供不同的功能。事实证明，不仅可以特化函数，还可以特化整个类！
这里有一个简化的类：

```
template <typename T>
class Storage8
{
private:
    T m_array[8];

public:
    void set(int index, const T& value)
    {
        m_array[index] = value;
    }

    const T& get(int index) const
    {
        return m_array[index];
    }
};
```

因为这是一个模板类，它可以对任意给定类型生效：

```
#include <iostream>

int main()
{
    // Define a Storage8 for integers
    Storage8<int> intStorage;

    for (int count{ 0 }; count < 8; ++count)
        intStorage.set(count, count);

    for (int count{ 0 }; count < 8; ++count)
        std::cout << intStorage.get(count) << '\n';

    // Define a Storage8 for bool
    Storage8<bool> boolStorage;
    for (int count{ 0 }; count < 8; ++count)
        boolStorage.set(count, count & 3);

    std::cout << std::boolalpha;

    for (int count{ 0 }; count < 8; ++count)
    {
        std::cout << boolStorage.get(count) << '\n';
    }

    return 0;
}
```

尽管这个类完全有效，但是 `Storage8<bool>` 的实现非常的低效。因为所有变量都有地址，同时 CPU 无法处理任何小于一字节的东西，所有变量都必须是至少一个字节的大小。因此，布

尔型的值使用了整个字节，即使技术上而言它只需要一个 bit 来存储 true 与 false 值！也就是说 1 bit 是有用的信息而其他 7 bit 浪费了空间。
为了消除浪费的空间，需要对类进行修改以满足布尔型，替换 8 个布尔值的数组为单个字节大小。虽然可以创建一个完全新的类来完成目标，但是却会有一个最大的缺点：需要给其一个不同的名字。那么程序员就必须记住 `Storage8<T>` 是用于非布尔类型的，而 `Storage8Bool`（或者其他名字）是用于布尔类型的。幸运的是，C++ 提供了一个更好的方法：类模板特化。

类模板特化

类模板特化允许用户为了特定数据类型（若干模板参数时同样成立）特化一个模板类。这个例子中使用类模板特化编写一个自定义版本的 `Storage8<bool>` 优先级高于通用的 `Storage8<T>`。其工作方式类似于特化函数的优先级高于通用的模板函数。
类模板特化被视为完全独立的类，即使它们的内存分配方式与模板类相同。这就意味这可以修改特化类中的所有的东西，包括其实现，甚至于公有化函数，正如一个独立的类一般。

```
// 需要上面 Storage8 的类型定义

template <> // 以下是一个不带有模板参数的模板类
class Storage8<bool> // 开始为 bool 特化 Storage8
{
// 依照标准的类实现细节
private:
    unsigned char m_data{};

public:
    void set(int index, bool value)
    {
        // 弄清楚那个 bit 是 setting/unsetting
        // 放置 1 进 bit 用于 on/off
        auto mask{ 1 << index };

        if (value)  // 如果是 setting bit
            m_data |= mask;   // 使用 bitwise-or 使 bit on
        else  // 如果是 bit off
            m_data &= ~mask;  // 使用 bitwise-and 以及反转 mask 使 bit off
    }

    bool get(int index)
    {
        // 弄清楚获取的是哪个 bit
        auto mask{ 1 << index };
        // bitwise-and 获取 bit 值，接着隐式转换为布尔值
        return (m_data & mask);
    }
};
```

首先，注意由 `template<>` 开始。template 关键字告诉编译器下面都是模板，同时空的尖括号意味着没有任何模板参数。本例中没有任何模板参数是因为替换了仅有的参数（T）为指定的类型（bool）。
接着，添加 `<bool>` 给类名来表示正在特化一个 bool 版本的 `Storage8` 类。
所有其他的修改都是类的实现细节。现在暂时不需要理解 bit 逻辑是如何工作的。
注意特化类使用了一个 `unsigned char`（1 bit）而不是 8 个 bool 的数组（8 字节）。
现在当声明一个 `Storage8<T>` 而 `T` 不为布尔使，获取的是拓印通用模板 `Storage8<T>` 类，而当声明类型为 `Storage8<bool>` 时，获取的是刚创建好的特化版本。注意在两种模板中都用的是共有暴露接口 – 而 C++ 给与了全力来添加，移除，或者修改 `Storage8<bool>`，保持一致的接口意味着程序员可以使用拥有完全相同行为的类。

19.5 Partial template specialization

19.2 模板非类型参数章节中学习了表达式参数是如何使用在参数化模板类的。现在再来看一下之前写过的 `StaticArray` 例子：

```
template <typename T, int size> // size 是表达式参数
class StaticArray
{
private:
    // 表达式参数控制了数组的长度
    T m_array[size]{};

public:
    T* getArray() { return m_array; }

    T& operator[](int index)
    {
        return m_array[index];
    }
};
```

这个类拥有两个模板参数，一个类型参数，以及一个表达式参数。
现在希望编写一个函数来打印整个数组。尽管可以以成员函数的方式实现它，不过现在这里选择使用一个非成员函数可以让例子更容易向主题展开。
使用模板，可以想到以下方案：

```
template <typename T, int size>
void print(StaticArray<T, size>& array)
{
    for (int count{ 0 }; count < size; ++count)
        std::cout << array[count] << ' ';
}
```

整合一下：

```cpp
#include <iostream>
#include <cstring>

template <typename T, int size>
class StaticArray
{
private:
  T m_array[size]{};

public:
  T* getArray() { return m_array; }

  T& operator[](int index)
  {
    return m_array[index];
  }
  };

template <typename T, int size>
void print(StaticArray<T, size>& array)
{
  for (int count{ 0 }; count < size; ++count)
    std::cout << array[count] << ' ';
}

int main()
{
  // 声明一个 int 数组
  StaticArray<int, 4> int4{};
  int4[0] = 0;
  int4[1] = 1;
  int4[2] = 2;
  int4[3] = 3;

  // 打印数组
  print(int4);

  return 0;
}
```

打印:

```
0 1 2 3
```

尽管这可以工作，但是它有一个设计缺陷。考虑以下代码:

```cpp
int main()
{
    // 声明一个 char 数组
    StaticArray<char, 14> char14{};
```

```
5
6       std::strcpy(char14.getArray(), "Hello, world!");
7
8       // 打印数组
9       print(char14);
10
11      return 0;
12  }
```

该程序可以被编译，执行，并打印以下值：

```
H e l l o ,   w o r l d !
```

对于非字符类型而言，在每个元素之间添加空格是有意义的，这样它们不会跑到一起去。然而对于字符类型而言，没有空格更有意义。那么该如何解决这个问题呢？

模板特化来拯救吗？

很多人可能第一个想到的会是使用模板特化。

```
#include <iostream>
#include <cstring>

template <typename T, int size> // size is the expression parameter
class StaticArray
{
private:
  // The expression parameter controls the size of the array
  T m_array[size]{};

public:
  T* getArray() { return m_array; }

  T& operator[](int index)
  {
    return m_array[index];
  }
};

template <typename T, int size>
void print(StaticArray<T, size>& array)
{
  for (int count{ 0 }; count < size; ++count)
    std::cout << array[count] << ' ';
}

// Override print() for fully specialized StaticArray<char, 14>
template <>
void print(StaticArray<char, 14>& array)
```

```
{
  for (int count{ 0 }; count < 14; ++count)
    std::cout << array[count];
}

int main()
{
  // declare a char array
  StaticArray<char, 14> char14{};

  std::strcpy(char14.getArray(), "Hello, world!");

  // Print the array
  print(char14);

  return 0;
}
```

很明显假设要调用 `StaticArray<char, 12>` 的时候还需要再拷贝一份 `print` 函数。

模板偏特化

模板片特化 partial template specialization 允许特化类（但不是单独函数!）的一部分，而不是全部。上述问题的最理想的解决方案是重载 `print` 函数，但是留下长度的表达式参数模板化以便应对不同的需求。

这里是重载 `print` 函数并接受一个片特化的 `StaticArray`：

```
// 重载 print() 函数，为了偏特化的 StaticArray<char, size>
template <int size> // size 仍然是一个模板表达式参数
void print(StaticArray<char, size>& array) // 这里显式定义字符类型
{
  for (int count{ 0 }; count < size; ++count)
    std::cout << array[count];
}
```

可以看到显示声明的这个函数仅对于 `StaticArray` 的字符类型有效，size 仍然是一个模板表达式参数，所以它将对任何 size 的字符数组有效。

```
#include <iostream>
#include <cstring>

template <typename T, int size>
class StaticArray
{
private:
  T m_array[size]{};

public:
```

```
  T* getArray() { return m_array; }

  T& operator[](int index)
  {
    return m_array[index];
  }
};

template <typename T, int size>
void print(StaticArray<T, size>& array)
{
  for (int count{ 0 }; count < size; ++count)
    std::cout << array[count] << ' ';
}

template <int size>
void print(StaticArray<char, size>& array)
{
  for (int count{ 0 }; count < size; ++count)
    std::cout << array[count];
}

int main()
{
  // 声明 size 14 的数组
  StaticArray<char, 14> char14{};

  std::strcpy(char14.getArray(), "Hello, world!");

  print(char14);

  std::cout << ' ';

  // 声明 size 12 的数组
  StaticArray<char, 12> char12{};

  std::strcpy(char12.getArray(), "Hello, mom!");

  print(char12);

  return 0;
}
```

偏特化模板仅可以在类上面使用，而不能在模板函数使用（函数必须是全特化的）。

对成员函数的模板偏特化

由于函数的偏特化限制导致了在处理成员函数的时候会出现一些挑战。例如，如果是以下这样定义 `StaticArray` 的呢?

```
template <typename T, int size>
class StaticArray
{
private:
    T m_array[size]{};

public:
    T* getArray() { return m_array; }

    T& operator[](int index)
    {
        return m_array[index];
    }

    void print()
    {
        for (int i{ 0 }; i < size; ++i)
            std::cout << m_array[i] << ' ';
        std::cout << '\n';
    }
};
```

`print()` 现在是类的成员函数，那么这么尝试:

```
// 不起作用
template <int size>
void StaticArray<double, size>::print()
{
  for (int i{ 0 }; i < size; ++i)
    std::cout << std::scientific << m_array[i] << ' ';
  std::cout << '\n';
}
```

不幸的是这并不起作用，因为尝试偏特化一个函数，这是不允许的。
那么该如何绕过这个问题呢? 一种显而易见的方式就是对整个类特化:

```
#include <iostream>

template <typename T, int size>
class StaticArray
{
private:
  T m_array[size]{};

public:
```

```
  T* getArray() { return m_array; }

  T& operator[](int index)
  {
    return m_array[index];
  }
  void print()
  {
    for (int i{ 0 }; i < size; ++i)
      std::cout << m_array[i] << ' ';
    std::cout << '\n';
  }
};

template <int size>
class StaticArray<double, size>
{
private:
  double m_array[size]{};

public:
  double* getArray() { return m_array; }

  double& operator[](int index)
  {
    return m_array[index];
  }
  void print()
  {
    for (int i{ 0 }; i < size; ++i)
      std::cout << std::scientific << m_array[i] << ' ';
    std::cout << '\n';
  }
};

int main()
{
  StaticArray<int, 6> intArray{};

  for (int count{ 0 }; count < 6; ++count)
    intArray[count] = count;

  intArray.print();

  StaticArray<double, 4> doubleArray{};

  for (int count{ 0 }; count < 4; ++count)
    doubleArray[count] = (4.0 + 0.1 * count);

```

```
  doubleArray.print();

  return 0;
}
```

然而这并不是一个好的解决方案，因为重复了很多 `StaticArray<T, size>` 的代码。
幸运的是，使用一个公共的基类可以绕过这个问题：

```
#include <iostream>

template <typename T, int size>
class StaticArray_Base
{
protected:
  T m_array[size]{};

public:
  T* getArray() { return m_array; }

  T& operator[](int index)
  {
    return m_array[index];
  }

  void print()
  {
    for (int i{ 0 }; i < size; ++i)
      std::cout << m_array[i] << ' ';
    std::cout << '\n';
  }

  virtual ~StaticArray_Base() = default;
};

template <typename T, int size>
class StaticArray: public StaticArray_Base<T, size>
{
};

template <int size>
class StaticArray<double, size>: public StaticArray_Base<double, size>
{
public:

  void print()
  {
    for (int i{ 0 }; i < size; ++i)
      std::cout << std::scientific << this->m_array[i] << ' ';
  // 注意：上面一行的 this-> prefix 是必须的。
```

```
  // 查看 https://stackoverflow.com/a/6592617 或 https://isocpp.org/wiki/faq/templates#
     nondependent-name-lookup-members 可以了解更多的细节。
    std::cout << '\n';
  }
};

int main()
{
  StaticArray<int, 6> intArray{};

  for (int count{ 0 }; count < 6; ++count)
    intArray[count] = count;

  intArray.print();

  StaticArray<double, 4> doubleArray{};

  for (int count{ 0 }; count < 4; ++count)
    doubleArray[count] = (4.0 + 0.1 * count);

  doubleArray.print();

  return 0;
}
```

这可以打印相同的内容，并且大大的减少了重复代码。

19.6 Partial template specialization for pointers

19.3 函数模板特化的章节中提到的 `Storage` 类：

```
#include <iostream>

template <typename T>
class Storage
{
private:
    T m_value;
public:
    Storage(T value)
        : m_value { value }
    {
    }

    ~Storage()
    {
    }

    void print() const
```

```
    {
        std::cout << m_value << '\n';
    }
};
```

之前展示了在 T 为 char* 类型的时候遇到的问题，因为浅拷贝/指针赋值会发生在构造函数中。在之前的章节中，用到全模板特化来为 char* 类型创建一个特殊版本的构造函数，使其可以分配内存并对 m_value 进行真正的深拷贝。作为参考，以下是全特化的 char* Storage 的构造函数与析构函数：

```
// 需要 include Storage<T> 类

template <>
Storage<char*>::Storage(char* value)
{
  int length { 0 };

  while (value[length] != '\0')
    ++length;
  ++length;

  m_value = new char[length];

  for (int count=0; count < length; ++count)
    m_value[count] = value[count];
}

template<>
Storage<char*>::~Storage()
{
  delete[] m_value;
}
```

对于 Storage<char*> 而言效果非常的好，那么其他的指针类型（例如 int* ）该怎么办呢？很容易看得出来如果 T 是任何指针类型，之前讨论过的问题仍然都会存在。

由于全模板特化强制全解析了模板类型，为了解决这个问题，所有的指针类型都必须定义一个新的特化构造函数（以及析构函数）！这又带来了很多重复的代码，那么现在可以用偏特化模板来避免这个问题。

```
#include <iostream>

template <typename T>
class Storage<T*> // 这是一个偏特化的 Storage 仅作用于指针类型
{
private:
    T* m_value;
public:
  Storage(T* value) // 指针类型的 T
```

```
    : m_value { new T { *value } } // 拷贝单个值，而不是数组
  {
  }

  ~Storage()
  {
    delete m_value; // 因此这里用的是标量的删除，而不是数组删除
  }

  void print() const
  {
    std::cout << *m_value << '\n';
  }
};
```

测试：

```
int main()
{
  Storage<int> myint { 5 };
  myint.print();

  int x { 7 };
  Storage<int*> myintptr { &x };

  // 展示 myintptr 与 x 无关。
  // 如果修改了 x，myintptr 不应该改变。
  x = 9;
  myintptr.print();

  return 0;
}
```

打印：

```
5
7
```

当 myintptr 的定义带有 int* 模板参数时，编译器看到定义得到偏特化模板类可用作于任何指针类型，接着通过模板实例化一个 Storage，其构造函数会对参数 x 进行深拷贝，当修改 x 成 9 时，myintptr.m_value 并不收到影响，因为其指向的是自身独立的值。

值得注意的是因为偏特化的 Storage 类仅分配单独一个值，对于 C-style 字符串而言，仅会拷贝第一个字符。如果需要拷贝整个字符串，则需要为 char* 类型全特化一个构造函数（以及析构函数）。全特化的版本优先级高于偏特化的版本。以下是一个为指针使用的偏特化，以及为 char* 全特化的例子：

```
#include <iostream>
#include <cstring>

```

```
// 非指针的 Storage 模板类
template <typename T>
class Storage
{
private:
  T m_value;
public:
  Storage(T value)
    : m_value { value }
  {
  }

  ~Storage()
  {
  }

  void print() const
  {
    std::cout << m_value << '\n';
  }
};

// 指针的 Storage 偏特化模板类
template <typename T>
class Storage<T*>
{
private:
  T* m_value;
public:
  Storage(T* value)
    : m_value { new T { *value } } // 这里拷贝单独一个值，而不是一个数组
  {
  }

  ~Storage()
  {
    delete m_value;
  }

  void print() const
  {
    std::cout << *m_value << '\n';
  }
};

// char* 类型的全特化的构造函数
template <>
Storage<char*>::Storage(char* value)
{
```

```
53    // 弄清楚 string 的长度
54    int length { 0 };
55    while (value[length] != '\0')
56      ++length;
57    ++length; // +1 用于解释 null 终止符
58
59    // 分配内存用于存储 string 值
60    m_value = new char[length];
61
62    // 拷贝实际的 string 值到刚分配的 m_value 内存
63    for (int count = 0; count < length; ++count)
64      m_value[count] = value[count];
65  }
66
67  // char* 类型的全特化的析构函数
68  template<>
69  Storage<char*>::~Storage()
70  {
71    delete[] m_value;
72  }
73
74  // char* 类型的全特化的打印函数
75  // 没有这个函数打印 Storage<char*> 将会调用 Storage<T*>::print()，则只会打印第一个字符
76  template<>
77  void Storage<char*>::print() const
78  {
79    std::cout << m_value;
80  }
81
82  int main()
83  {
84    // 声明一个非指针 Storage 来证明可以工作
85    Storage<int> myint { 5 };
86    myint.print();
87
88    // 声明一个指针 Storage 来证明可以工作
89    int x { 7 };
90    Storage<int*> myintptr { &x };
91
92    // 如果 myintptr 对 x 实施的是指针赋值，那么修改 x 也会修改 myintptr
93    x = 9;
94    myintptr.print();
95
96    // 动态分配一个临时 string
97    char* name { new char[40]{ "Alex" } };
98
99    // 存储名称
100   Storage<char*> myname { name };
101
```

```
102   // 删除临时 string
103   delete[] name;
104
105   // 打印名称来证明构造时进行了拷贝
106   myname.print();
107
108   return 0;
109 }
```

打印:

```
1 5
2 7
3 Alex
```

使用偏函数模板类特化来分别创建指针与非指针的实现非常的有用，尤其是希望一个类能够分别处理这两种情况，但是某种层度上对端点使用者是完全透明的。

20 Exceptions

20.1 The need for exceptions

在之前的处理错误的章节中地道过使用 `assert()`，`std::cerr`，以及 `exit()` 来处理异常。然而，有一个更重要的方式将在本章覆盖：异常。

当返回码为失败时

当编写可复用的代码时，错误处理事必须得。一个最常见的方式来处理潜在的错误就是通过返回码。例如：

```
#include <string_view>

int findFirstChar(std::string_view string, char ch)
{
    // 遍历字符串中每个字符
    for (std::size_t index{ 0 }; index < string.length(); ++index)
        // 如果匹配 ch，返回其索引
        if (string[index] == ch)
            return index;

    // 如果不匹配，返回 -1
    return -1;
}
```

这个函数返回字符串中第一个匹配 `ch` 字符的索引。如果没有找到则返回 -1 作为错误指示。这种方法的主要优点是，它非常的简单。然而在重要场合下使用返回码的若干缺陷会快速暴露出来：

首先，返回值的含义是模糊的 – 如果一个函数返回 -1，那么它到底是代表错误，还是一个真实有效的值?

其次，函数仅可以返回一个值，那么如果希望既返回一个函数结果又返回一个错误码呢?

再者，一系列的代码中有很多地方可以出错，错误码需要经常被检查。考虑下面的代码切片，涉及到传递一个文本文件：

```
std::ifstream setupIni { "setup.ini" }; // 打开 setup.ini 用于读
// 如果文件不能被打开（例如文件缺失）返回一些某错误枚举
if (!setupIni)
    return ERROR_OPENING_FILE;

// 开始从文件中读取内容
if (!readIntegerFromFile(setupIni, m_firstParameter)) // 尝试从文件中读取整型值
    return ERROR_READING_VALUE; // 返回代表不能读取值的枚举值

if (!readDoubleFromFile(setupIni, m_secondParameter)) // 尝试从文件中读取双精度值
```

```
11     return ERROR_READING_VALUE;
12
13 if (!readFloatFromFile(setupIni, m_thirdParameter))   // 尝试从文件中读取浮点值
14     return ERROR_READING_VALUE;
```

暂时还未覆盖文件访问，无须担心上述代码如何工作的 – 仅需要注意每个调用都需要错误检查并返回给调用者。现在试想一下有二十种类型的参数 – 那么按照上面的做法要检查二十次并返回 ERROR_READING_VALUE 二十次!

第四，错误码并不与构造函数契合。如果创建一个对象时构造函数内发生错误应该怎么办? 构造函数没有返回类型传递回状态指示，通过引用参数传递回的话又麻烦又必须要显示检查。另外，即使做了这些，对象仍然会被创建，那么就需要处理它或者销毁它。

最后，当错误码返回给调用者时，调用者并不能总是处理错误。如果调用者不想处理错误，那么要么是忽略它（这种情况下就永远失去了），要么是再向上返回错误码。这相当的胡乱同时也带来了之前提到过的那些问题。

异常

异常处理提供了一种机制用来从通常代码的控制流中解耦错误或者其它特殊情况。它使处理错误在合适以及如何处理上，拥有更多的自由，消除了大部分（如果不是全部的）的错误码带来的混乱。

20.2 Basic exception handling

C++ 中的异常处理是由三个关键字关联工作的：*throw*，*try* 以及 *catch*。

抛出异常

在 C++ 中，一个 *throw* 声明用于发出异常或者错误情况出现了的信号。一个异常的出现同样被称为 *raising* 一个异常。例如：

```
1 throw -1; // throw a literal integer value
2 throw ENUM_INVALID_INDEX; // throw an enum value
3 throw "Can not take square root of negative number"; // throw a literal C-style (const char*)
      string
4 throw dX; // throw a double variable that was previously defined
5 throw MyException("Fatal Error"); // Throw an object of class MyException
```

每个声明作为一个信号代表某种类型的问题出现了且需要被解决。

查看异常

抛出异常仅是异常处理过程的一部分。C++ 中，使用 *try* 关键字来定义一块声明（成为 *try block*），检查其中任何代码块中可能抛出异常的地方。

```
try
{
    // Statements that may throw exceptions you want to handle go here
    throw -1; // here's a trivial throw statement
}
```

注意 try block 并没有定义如何去处理异常。

处理异常

真实处理异常的地方是在 catch block(s)。*catch* 关键字用于定义单个数据类型的异常处理。

```
catch (int x)
{
    // Handle an exception of type int here
    std::cerr << "We caught an int exception with value" << x << '\n';
}
```

如果参数不会在 catch block 中用到，则可以被省略：

```
catch (double) // note: no variable name since we don't use it in the catch block below
{
    // Handle exception of type double here
    std::cerr << "We caught an exception of type double" << '\n';
}
```

throw，try 以及 catch

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

int main()
{
    try
    {
        // Statements that may throw exceptions you want to handle go here
        throw -1; // here's a trivial example
    }
    catch (int x)
    {
        // Any exceptions of type int thrown within the above try block get sent here
        std::cerr << "We caught an int exception with value: " << x << '\n';
    }
    catch (double) // no variable name since we don't use the exception itself in the catch
     block below
    {
        // Any exceptions of type double thrown within the above try block get sent here
        std::cerr << "We caught an exception of type double" << '\n';
```

```
    }
    catch (const std::string&) // catch classes by const reference
    {
        // Any exceptions of type std::string thrown within the above try block get sent here
        std::cerr << "We caught an exception of type std::string" << '\n';
    }

    std::cout << "Continuing on our merry way\n";

    return 0;
}
```

catch block 通常用来做什么

如果一个异常被导航到了一个 catch block，它被视为“已处理”即使该 catch block 是空的。然而，通常希望的是 catch block 能做一些有用的事情。有三种通常的做法：
首先，可以打印错误（输出到终端或是日志文件）。
其次，可以返回一个值或者错误码给调用者。
第三，可以抛出另一个异常。因为 catch block 是在 try block 外部，新的抛出异常并没有被 try block 处理 – 它会被另一个包围其的 try block 处理。

20.3 Exceptions, functions, and stack unwinding

被调用的函数中抛出异常

重点：try blocks 不仅可以从 try block 中的声明中捕获异常，同样也可以捕获 try block 中被调用函数的异常。

```
#include <cmath> // sqrt() 函数
#include <iostream>

// 一个组合式的开根号函数
double mySqrt(double x)
{
    // 如果用户输入的是负值，这是一个错误条件
    if (x < 0.0)
        throw "Can not take sqrt of negative number"; // 抛出 const char* 类型的异常

    return std::sqrt(x);
}

int main()
{
    std::cout << "Enter a number: ";
    double x {};
```

```
    std::cin >> x;

    try // 查看在 try block 中出现的异常，并导航到附属的 catch block(s)
    {
        double d = mySqrt(x);
        std::cout << "The sqrt of " << x << " is " << d << '\n';
    }
    catch (const char* exception) // 捕获异常类型 const char*
    {
        std::cerr << "Error: " << exception << std::endl;
    }

    return 0;
}
```

异常处理与栈展开

当异常被抛出时，程序首先查看异常是否能立刻被当前函数中处理（意为在当前函数的 try block 中抛出异常，且有相关的关联 catch block）。如果可以处理那么便处理。
如果不能被处理，程序会检查函数调用者（下一个在栈上的函数）是否能处理异常。为了使函数的调用者处理异常，调用当前函数必须时在一个 try block 中，并与一个匹配的 catch block 关联。如果没有匹配的捕获，调用者的调用者再去进行检查，以此类推。
如果一个匹配的异常被捕获，则执行跳转到与之匹配的 catch block。这就需要展开栈（在调用栈中移除当前函数）若干次直到处理异常的函数在调用栈的顶部。
如果没有匹配被发现，栈有可能会被展开。这会在下一章中详细讲解。

另一个栈展开的例子

```
#include <iostream>

void D() // 由 C() 调用
{
    std::cout << "Start D\n";
    std::cout << "D throwing int exception\n";

    throw - 1;

    std::cout << "End D\n"; // 跳过
}

void C() // 由 B() 调用
{
    std::cout << "Start C\n";
    D();
```

```
17      std::cout << "End C\n";
18  }
19
20  void B() // 由 A() 调用
21  {
22      std::cout << "Start B\n";
23
24      try
25      {
26          C();
27      }
28      catch (double) // 不捕获：异常类型不匹配
29      {
30          std::cerr << "B caught double exception\n";
31      }
32
33      try
34      {
35      }
36      catch (int) // 不捕获：异常类型不匹配
37      {
38          std::cerr << "B caught int exception\n";
39      }
40
41      std::cout << "End B\n";
42  }
43
44  void A() // 由 main() 调用
45  {
46      std::cout << "Start A\n";
47
48      try
49      {
50          B();
51      }
52      catch (int) // 这里捕获异常并处理
53      {
54          std::cerr << "A caught int exception\n";
55      }
56
57      std::cout << "End A\n";
58  }
59
60  int main()
61  {
62      std::cout << "Start main\n";
63
64      try
65      {
```

```
66          A();
67      }
68      catch (int) // 没有被调用，因为异常在 A 中被处理了
69      {
70          std::cerr << "main caught int exception\n";
71      }
72      std::cout << "End main\n";
73
74      return 0;
75  }
```

打印:

```
Start main
Start A
Start B
Start C
Start D
D throwing int exception
A caught int exception
End A
End main
```

20.4 Uncaught exceptions and catch-all handlers

未捕获异常

当一个函数抛出异常且并没有被自身处理，那么是在假设函数的调用栈上将会处理该异常。

```
#include <iostream>
#include <cmath> // sqrt()

double mySqrt(double x)
{
    if (x < 0.0)
        throw "Can not take sqrt of negative number";

    return std::sqrt(x);
}

int main()
{
    std::cout << "Enter a number: ";
    double x;
    std::cin >> x;

    // 这里没有异常处理!
    std::cout << "The sqrt of " << x << " is " << mySqrt(x) << '\n';

```

```
    return 0;
}
```

当一个函数不能找到异常处理时，`std::terminate()` 会被调用，整个程序会被终止。这样的情况，调用栈有可能被展开！如果没有被展开，本地变量则不会被销毁，那么任何预期的析构函数并不会出现！
警告：如果一个异常未被捕获，调用栈有可能不被展开。

catch-all 处理

那么现在的难题是：

- 函数可能潜在抛出任何类型异常（包括程序定义的数据类型），意味着异常会有无限种可能。
- 如果异常没有被捕获，程序则会立马停止（同时栈有可能不被展开，因此程序有可能不能正确的清除自身）。
- 为每一种可能得类型添加一个显式的捕获处理太过冗长，特别是一些预期就是仅有在特殊情况下才能达到！

幸运的是 C++ 同样也提供了一个异常全捕获的机制。也就是熟知的 **catch-all handler**。

```
#include <iostream>

int main()
{
  try
  {
    throw 5; // throw an int exception
  }
  catch (double x)
  {
    std::cout << "We caught an exception of type double: " << x << '\n';
  }
  catch (...) // catch-all handler
  {
    std::cout << "We caught an exception of an undetermined type\n";
  }
}
```

使用 catch-all handler 来包裹 main()

```cpp
#include <iostream>

int main()
{

    try
    {
        runGame();
    }
    catch(...)
    {
        std::cerr << "Abnormal termination\n";
    }

    saveState();
    return 1;
}
```

最佳实践：如果程序使用了异常，考虑在 main 中使用 catch-all handler，这样可以避免未捕获异常出现时而导致的极端情况发生。同样在 debug 构建中不使用 catch-all handler，这样可以更简单的定位到未捕获异常是如何发生的。

20.5 Exceptions, classes, and inheritance

异常与成员函数

在成员函数中，异常同样非常有用，甚至在重载操作符中更有用。考虑下列的重载 `[]` 操作符作为简易整型数组类：

```cpp
int& IntArray::operator[](const int index)
{
    return m_data[index];
}
```

尽管这个函数在有效范围能工作的很好，但是它急缺一个好的错误检查。可以通过断言声明来确保其有效性：

```cpp
int& IntArray::operator[](const int index)
{
    assert (index >= 0 && index < getLength());
    return m_data[index];
}
```

现在如果用户传入了无效索引，项目会导致断言错误。不幸的是因为重载操作符有其特定的需求的数量和类型参数用于返回，通过向调用者返回错误码或者布尔值的做法非常的不灵活。然而，由于异常并不会修改函数签名，它们在此的作用意义重大。

```
int& IntArray::operator[](const int index)
{
    if (index < 0 || index >= getLength())
        throw index;

    return m_data[index];
}
```

当构造函数失败时

构造函数是另一个使用异常非常有用的地方。如果一个构造函数因为某些原因（例如用户无效输入）必须失败时，简单的抛出一个异常就可以指示创建对象失败了。这种情况下对象的构造流产了，所有的类成员（已被创建的，以及之前在构造函数体内初始化的）会被照常被析构。然而类的析构函数永远不会被调用（因为对象从未完成构造函数）。因为析构函数从未执行，因此不可依赖析构函数来清理已被分配的内存。

那么问题就来了，应该怎么做当构造函数中分配了内存之后且构造函数结束之前异常出现了。如何确保已被分配的资源被正确的清理？一个方法是使用 try block 包裹任何可能出现失败的代码，使用关联的 catch block 来捕获异常并做出必要的清理，接着重新抛出异常。然而这添加了一堆杂乱的东西，且非常容易出错，特别是当类分配了若干资源时。

幸运的是，还有一个更好的办法。利用了类成员即使在构建函数失败时也会被析构的优势，如果资源分配处于类成员之中（而不是处于构造函数之中），那么这些成员可以在被析构时清理自身。

```
#include <iostream>

class Member
{
public:
  Member()
  {
    std::cerr << "Member allocated some resources\n";
  }

  ~Member()
  {
    std::cerr << "Member cleaned up\n";
  }
};

class A
{
private:
  int m_x {};
  Member m_member;
```

```

public:
  A(int x) : m_x{x}
  {
    if (x <= 0)
      throw 1;
  }

  ~A()
  {
    std::cerr << "~A\n"; // 不应该被调用到
  }
};


int main()
{
  try
  {
    A a{0};
  }
  catch (int)
  {
    std::cerr << "Oops\n";
  }

  return 0;
}
```

打印:

```
Member allocated some resources
Member cleaned up
Oops
```

上述代码中，当类 A 抛出一个异常，所有 A 的成员都被析构，`m_member` 的析构函数被调用，提供了清理其分配的任何资源的机会。

这也是 RAII 被高度提倡的部分原因 – 即使在异常环境中，实现 RAII 了的类有能力清理自身。然而像是创建一个 Member 这样的自定义类来管理资源分配并不高效。幸运的是，C++ 标准库带来的符合 RAII 类来管理通常的资源类型，例如文件（`std::fstream` 23.6 中覆盖）以及动态内存（`std::unique_ptr` 以及其他类型的智能指针 M.1 中覆盖）。

例如，与其这么做:

```
class Foo
private:
    int* ptr; // Foo 将处理 分配/释放
```

不如这么做:

```
class Foo
private:
    std::unique_ptr<int> ptr; // std::unique_ptr 将处理 分配/释放
```

前一种情况中，如果 Foo 构造函数在 ptr 分配动态内存后失败了，Foo 则需要负责清理，这会带来挑战。后一种情况中，Foo 构造函数在 ptr 分配动态内存后失败了，ptr 的析构函数会执行并将内存返回给系统。Foo 不需要做任何的显示清理，因为资源处理被委派给了符合 RAII 的成员！

异常类

使用基础数据类型做异常类型的一个最大问题就是他们是本身的意义是模糊的。一个更大的问题是当多个声明或函数调用存在一个 try block 时会引起歧义。
一个解决该问题的方法是使用异常类。一个**异常类**就是一个被设计成抛出异常的普通的类。

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

class ArrayException
{
private:
  std::string m_error;

public:
  ArrayException(std::string_view error)
    : m_error{ error }
  {
  }

  const std::string& getError() const { return m_error; }
};

class IntArray
{
private:
  int m_data[3]{};

public:
  IntArray() {}

  int getLength() const { return 3; }

  int& operator[](const int index)
  {
    if (index < 0 || index >= getLength())
      throw ArrayException{ "Invalid index" };
```

```
33
34      return m_data[index];
35    }
36  };
37
38  int main()
39  {
40    IntArray array;
41
42    try
43    {
44      int value{ array[5] };
45    }
46    catch (const ArrayException& exception)
47    {
48      std::cerr << "An array exception occurred (" << exception.getError() << ")\n";
49    }
50  }
```

使用这样的一个类就可以返回带有问题描述信息的异常，因此报错时可以提供异常信息。因为 `ArrayException` 拥有其自身类型，可以通过其类型指定捕获异常。

异常与继承

由于抛出类作为异常是可行的，类又是可以由其他类派生而来，因此需要考虑清楚当使用继承的类作为异常时回发生什么。事实上，异常处理将不仅仅匹配指定类型，它们也会匹配从指定类型中派生而来的类！

```
1  #include <iostream>
2
3  class Base
4  {
5  public:
6      Base() {}
7  };
8
9  class Derived: public Base
10 {
11 public:
12     Derived() {}
13 };
14
15 int main()
16 {
17     try
18     {
19         throw Derived();
20     }
```

```
    catch (const Base& base)
    {
        std::cerr << "caught Base";
    }
    catch (const Derived& derived)
    {
        std::cerr << "caught Derived";
    }

    return 0;
}
```

上述的例子抛出的是 `Derived` 类型的异常，然而输出的却是：

```
caught Base
```

为什么会这样呢？

首先，派生类会被当成基类被捕获。因为 `Derived` 由 `Base` 派生而来，`Derived` 是一个 `Base` 。其次当 C++ 尝试寻找一个被抛出异常的处理句柄，它是顺序进行的。因此，C++ 做的第一件事就是检查异常处理是否匹配 `Base` ，如果匹配，则执行 `Base` 的 catch block！而 `Derived` 的 catch block 永远不会被执行。

为了使其符合预期，需要调转一下 catch blocks 的顺序：

```
#include <iostream>

class Base
{
public:
    Base() {}
};

class Derived: public Base
{
public:
    Derived() {}
};

int main()
{
    try
    {
        throw Derived();
    }
    catch (const Derived& derived)
    {
        std::cerr << "caught Derived";
    }
    catch (const Base& base)
```

```
    {
        std::cerr << "caught Base";
    }

    return 0;
}
```

规则：处理派生类的异常句柄需要被排列在基类之前。

std::exception

在标准库中又很多类与操作符在失败时会抛出异常。例如，操作符 `new` 可以在不能分配足够内存时抛出 `std::bad_alloc` 。一个失败的 `dynamic_cast` 将抛出 `std::bad_cast` 。到 C++20 时已经有 28 中不同的异常类可以被抛出，更多的异常类会添加在随后的语言标准。

好消息是所有的这些异常类都是从单个名为 **std::exception** （定义在 `<exception>` 头文件中）派生而来的。`std::exception` 是一个小型接口类，设计用于服务所有被 C++ 标准库抛出异常的基类。

大部分时间，当一个由标准库抛出的异常，用户不需要关心其是否为一个不好的内存分配，一个不好的转型或其他。仅需要关注是什么灾难性的错误导致程序崩溃。感谢 `std::exception` ，用户可以设置其错误处理，并一次性捕获其所有的派生异常！

```
#include <cstddef> // for std::size_t
#include <exception> // for std::exception
#include <iostream>
#include <limits>
#include <string> // for this example

int main()
{
    try
    {
        // Your code using standard library goes here
        // We'll trigger one of these exceptions intentionally for the sake of the example
        std::string s;
        s.resize(std::numeric_limits<std::size_t>::max()); // will trigger a std::length_error
    or allocation exception
    }
    // This handler will catch std::exception and all the derived exceptions too
    catch (const std::exception& exception)
    {
        std::cerr << "Standard exception: " << exception.what() << '\n';
    }

    return 0;
}
```

有时候会需要某个特定类型的异常，这种情况可以添加该指定类型。

```
try
{
     // code using standard library goes here
}
// This handler will catch std::length_error (and any exceptions derived from it) here
catch (const std::length_error& exception)
{
    std::cerr << "You ran out of memory!" << '\n';
}
// This handler will catch std::exception (and any exception derived from it) that fall
// through here
catch (const std::exception& exception)
{
    std::cerr << "Standard exception: " << exception.what() << '\n';
}
```

直接使用标准异常

没有任何直接抛出 `std::exception` 的情况，用户也不可以。然而如果符合需求，则可以随便抛出标准库中其他类型的异常类。可以在 cppreference 中找到所有的标准异常。

`std::runtime_error`（被包含在 `stdexcept` 头文件中）是一个好的选择，因为它拥有一个泛型名称，同时其构造函数可以接收自定义信息：

```
#include <exception> // for std::exception
#include <iostream>
#include <stdexcept> // for std::runtime_error

int main()
{
  try
  {
    throw std::runtime_error("Bad things happened");
  }
  // This handler will catch std::exception and all the derived exceptions too
  catch (const std::exception& exception)
  {
    std::cerr << "Standard exception: " << exception.what() << '\n';
  }

  return 0;
}
```

从 std::exception 或 std::runtime_error 中派生自定义的类

用户当然也可以从 `std::exception` 中派生自定义的类，并重写虚函数 `what()` const 成员函数。

```
#include <exception> // std::exception
#include <iostream>
#include <string>
#include <string_view>

class ArrayException : public std::exception
{
private:
  std::string m_error{}; // 处理自定义字符串

public:
  ArrayException(std::string_view error)
    : m_error{error}
  {
  }

  // std::exception::what() 返回一个 const char*
  const char* what() const noexcept override { return m_error.c_str(); }
};

class IntArray
{
private:
  int m_data[3] {};

public:
  IntArray() {}

  int getLength() const { return 3; }

  int& operator[](const int index)
  {
    if (index < 0 || index >= getLength())
      throw ArrayException("Invalid index");

    return m_data[index];
  }
};

int main()
{
  IntArray array;

  try
```

```
  {
    int value{ array[5] };
  }
  catch (const ArrayException& exception) // 派生的 catch blocks 先行
  {
    std::cerr << "An array exception occurred (" << exception.what() << ")\n";
  }
  catch (const std::exception& exception)
  {
    std::cerr << "Some other std::exception occurred (" << exception.what() << ")\n";
  }
}
```

注意虚函数 `what()` 有一个限定符 `noexcept` （意为函数保证其自身不会抛出任何异常）。因此重写应该也带上该限定符。

因为 `std::runtime_error` 已经拥有了字符串处理能力，它同样也是派生异常的基类。

```
#include <exception> // std::exception
#include <iostream>
#include <stdexcept> // std::runtime_error
#include <string>

class ArrayException : public std::runtime_error
{
public:
  // std::runtime_error 接收 const char* 空终止字符串
  // std::string_view 可能不是空终止字符串，因此在这里并不是一个好的选择
  // ArrayException 将获取一个 const std::string&，因为其保证是一个空终止字符串，
  // 同时可以被转换成一个 const char*
  ArrayException(const std::string& error)
    : std::runtime_error{ error.c_str() } // std::runtime_error 将处理字符串
  {
  }

  // 不再需要重写 what() 因为可以使用 std::runtime_error::what()
};

class IntArray
{
private:
  int m_data[3]{};

public:
  IntArray() {}

  int getLength() const { return 3; }

  int& operator[](const int index)
  {
```

```
33      if (index < 0 || index >= getLength())
34        throw ArrayException("Invalid index");
35
36      return m_data[index];
37    }
38  };
39
40  int main()
41  {
42    IntArray array;
43
44    try
45    {
46      int value{ array[5] };
47    }
48    catch (const ArrayException& exception) // 派生的 catch blocks 先行
49    {
50      std::cerr << "An array exception occurred (" << exception.what() << ")\n";
51    }
52    catch (const std::exception& exception)
53    {
54      std::cerr << "Some other std::exception occurred (" << exception.what() << ")\n";
55    }
56  }
```

用户是否要创建自定义的异常类完全根据自身选择，无论是使用标准异常类，亦或是从 `std::exception` 或 `std::runtime_error` 派生自定义异常类。所有有效的解决方案取决于目的。

20.6 Rethrowing exceptions

有时候用户可能会遇到捕获了一个异常，但是并不想（或没有能力）完全的在该时间点进行异常处理。这常见于当用户希望输出错误日志时，还需传递异常给真正的处理者。
试想如下场景：

```
1  Database* createDatabase(std::string filename)
2  {
3      try
4      {
5          Database* d = new Database(filename);
6          d->open(); // 假设这里抛出了一个 int 异常
7          return d;
8      }
9      catch (int exception)
10     {
11         // 数据库创建失败
12         delete d;
```

```
        // 将错误写入日志
        g_log.logError("Creation of Database failed");
    }

    return nullptr;
}
```

考虑下面的函数：

```
int getIntValueFromDatabase(Database* d, std::string table, std::string key)
{
    assert(d);

    try
    {
        return d->getIntValue(table, key); // 失败时抛出 int 异常
    }
    catch (int exception)
    {
        // 将错误写入日志
        g_log.logError("getIntValueFromDatabase failed");

        // 然而并没有实际处理这个错误，这里应该怎么做？
    }
}
```

抛出一个新的异常

一个显而易见的方案就是抛出一个新的异常。

```
int getIntValueFromDatabase(Database* d, std::string table, std::string key)
{
    assert(d);

    try
    {
        return d->getIntValue(table, key); // 失败时抛出 int 异常
    }
    catch (int exception)
    {
        // 将错误写入日志
        g_log.logError("getIntValueFromDatabase failed");

        throw 'q'; // 抛出字符异常 'q' 给栈，给 getIntValueFromDatabase() 的调用者去处理
    }
}
```

重新抛出一个异常（错误方式）

```cpp
int getIntValueFromDatabase(Database* d, std::string table, std::string key)
{
    assert(d);

    try
    {
        return d->getIntValue(table, key); // 失败时抛出 int 异常
    }
    catch (int exception)
    {
        // 将错误写入日志
        g_log.logError("getIntValueFromDatabase failed");

        throw exception;
    }
}
```

尽管这可以工作，这个方法存在一些缺陷。首先，并不会抛出与捕获到的异常完全相同 – 而是抛出拷贝初始化的异常变量的副本。尽管编译器可以免除拷贝，但是也有可能不免除，从而导致性能下降。

但是最重要的是考虑下面的例子：

```cpp
int getIntValueFromDatabase(Database* d, std::string table, std::string key)
{
    assert(d);

    try
    {
        return d->getIntValue(table, key); // 失败时抛出 int 异常
    }
    catch (Base &exception)
    {
        // 将错误写入日志
        g_log.logError("getIntValueFromDatabase failed");

        throw exception; // 危险：抛出一个 Base 对象，而不是 Derived 对象
    }
}
```

重新抛出一个异常（正确方式）

幸运的是 C++ 提供一种重新抛出完全一致异常的方法。仅需要使用 throw 关键字在 catch block 中（且不带关联变量），像是这样：

```cpp
#include <iostream>
class Base
{
public:
```

```
    Base() {}
    virtual void print() { std::cout << "Base"; }
};

class Derived: public Base
{
public:
    Derived() {}
    void print() override { std::cout << "Derived"; }
};

int main()
{
    try
    {
        try
        {
            throw Derived{};
        }
        catch (Base& b)
        {
            std::cout << "Caught Base b, which is actually a ";
            b.print();
            std::cout << '\n';
            throw; // 注意：这里重新抛出异常对象
        }
    }
    catch (Base& b)
    {
        std::cout << "Caught Base b, which is actually a ";
        b.print();
        std::cout << '\n';
    }

    return 0;
}
```

规则：当需要抛出同样的异常时，单个使用 throw 关键字。

20.7 Function try blocks

try 以及 catch blocks 可以应付大多数情况，但是有一种特殊的情况不适用。考虑下面例子：

```
#include <iostream>

class A
{
private:
  int m_x;
```

```
public:
  A(int x) : m_x{x}
  {
    if (x <= 0)
      throw 1; // 这里抛出异常
  }
};

class B : public A
{
public:
  B(int x) : A{x}
  {
    // 如果 A 的创建失败了会发生什么，以及该如何应对？
  }
};

int main()
{
  try
  {
    B b{0};
  }
  catch (int)
  {
    std::cout << "Oops\n";
  }
}
```

函数 try blocks

函数 try blocks 是设计用来对整个函数体建立异常而不是一个代码块，其语法稍微有点难以描述，且看下面例子：

```
#include <iostream>

class A
{
private:
  int m_x;
public:
  A(int x) : m_x{x}
  {
    if (x <= 0)
      throw 1; // 这里抛出异常
  }
};

```

```
class B : public A
{
public:
  B(int x) try : A{x} // 注意这里额外的 try 关键字
  {
  }
  catch (...) // 注意这与函数体本身处于同级
  {
    // 成员初始化列表的异常或是构造函数体在这里被捕获

    std::cerr << "Exception caught\n";

    throw; // 重新抛出现有异常
  }
};

int main()
{
  try
  {
    B b{0};
  }
  catch (int)
  {
    std::cout << "Oops\n";
  }
}
```

打印:

```
Exception caught
Oops
```

首先，注意额外的“try”关键字位于成员初始化列表之前。这代表着其所有的后续（直到函数结束）应该被视为在 try block 之内。

其次，注意关联的 catch block 是位于整个函数相同缩进的级别。任何在 try 关键字与函数结尾中抛出的异常都应该在这里被捕获。

当上面的程序运行时，变量 `b` 开始构造，调用 `B` 的构造函数（使用了函数 try）。`B` 的构造函数调用 `A` 的构造函数，接着抛出一个异常。因为 `A` 的构造函数并未处理这个异常，异常传递给栈上的 `B` 构造函数，其被 `B` 构造函数同等级所捕获。catch block 打印“Exception caught”，接着重新抛出当前异常给栈，即被 `main()` 中的 catch block 捕获，并打印“Oops”。

函数 catch blocks 的限制

对于常规的 catch block（函数内部）而言，有三种选项：可以抛出一个新的异常，重新抛出当前异常，或者是解决该异常（通过一个返回声明，或者使控制触及 catch block 底部）。

对于一个构造函数而言，函数等级的 catch block 必须抛出一个新异常或者抛出已有异常 – 也就是不允许解决异常！返回声明同样也不被运行，同时触及 catch block 底部将隐式的重新抛出异常。
对于一个结构函数而言，函数等级的 catch block 可以抛出异常，重新抛出异常，或者是通过一个返回声明解决当前异常。触及 catch block 的底部将会隐式的重新抛出异常。
对于其他函数而言，函数等级的 catch block 可以抛出异常，重新抛出异常，或者是通过一个返回声明解决当前异常。触及 catch block 的底部将会隐式的解析非值（`void` ）返回的函数，以及对值返回的函数产生未定义行为！
下面的表格总结了限制与函数等级 catch blocks 的行为：

Function type	Can resolve exceptions via return statement	Behavior at end of catch block
Constructor	No, must throw or rethrow	Implicit rethrow
Destructor	Yes	Implicit rethrow
Non-value returning function	Yes	Resolve exception
Value-returning function	Yes	Undefined behavior

由于在 catch block 结尾的不同行为依赖于函数类型（包括值返回函数的未定义行为），因此推荐永远不要让控制触及 catch block 的结尾，并总是显示抛出异常，重新抛出异常或者是返回。
最佳实践：应当避免函数等级的 catch block 触及结尾，而是显示的抛出异常，重新抛出异常或者是返回。

函数 try blocks 可以捕获基类以及当前类异常

上面的例子中，如果 `A` 或 `B` 构造函数抛出异常，则会被 `B` 的构造函数的 try block 所捕获。下面代码从 `B` 类中抛出异常，而不是从 `A` 。

```
#include <iostream>

class A
{
private:
  int m_x;
public:
  A(int x) : m_x{x}
  {
  }
};

class B : public A
{
public:
  B(int x) try : A{x} // 注意这里额外的 try 关键字
  {
    if (x <= 0) // 从 A 中移动到 B
      throw 1;  // 以及移动改行
  }
```

```
  catch (...)
  {
    std::cerr << "Exception caught\n";

    // 如果异常没有在这里被显式的抛出，当前异常将会被隐式重新抛出
  }
};

int main()
{
  try
  {
    B b{0};
  }
  catch (int)
  {
    std::cout << "Oops\n";
  }
}
```

可以获得相同的输出：

```
Exception caught
Oops
```

不要使用函数尝试清理资源

当一个对象的构造函数失败嘞，该类的析构函数并不会被调用。因此用户可能倾向使用一个函数 try block 在失败之前清理之前分配过的资源。然而指向失败对象的成员被视为未定义行为，因为对象在 catch block 之前就已经“死了”。这就意味着不可以在一个类结束后还使用函数 try 来进行清理。如果希望在类结束后清理，遵循标准规则来清理类，且抛出异常。
函数 try 主要用于日志失败之前通过传递异常给栈，或者改变抛出异常的类型。

20.8 Exception dangers and downsides

异常有很多的好处，但同时也存在一些潜在的缺陷！

清理资源

新手程序员在使用异常时遇到最大的问题就是清理资源。考虑下面代码：

```
#include <iostream>

try
{
    openFile(filename);
```

```
    writeFile(filename, data);
    closeFile(filename);
}
catch (const FileException& exception)
{
    std::cerr << "Failed to write to file: " << exception.what() << '\n';
}
```

如果 `WriteFile()` 失败并抛出一个 `FileException` 会发生什么? 这个时间点已经打开了文件，现在的控制流调到了 `FileException` 句柄，即打印错误并退出。注意文件永远没有被关闭！这个例子应该重写成:

```
#include <iostream>

try
{
    openFile(filename);
    writeFile(filename, data);
}
catch (const FileException& exception)
{
    std::cerr << "Failed to write to file: " << exception.what() << '\n';
}

// 确保文件被关闭
closeFile(filename);
```

该类型的错误通常以另一种形式出现，即当处理动态分配内存时:

```
#include <iostream>

try
{
    auto* john { new Person{ "John", 18, PERSON_MALE } };
    processPerson(john);
    delete john;
}
catch (const PersonException& exception)
{
    std::cerr << "Failed to process person: " << exception.what() << '\n';
}
```

如果 `processPerson()` 抛出异常，控制流调转到捕获句柄。结果 john 永远没有被释放！这个例子比上一个例子更为棘手 – 因为 john 是属于 try block 本地的，它会在 try block 退出时离开作用域。这就意味着异常处理完全不能访问 john（它已经被销毁了），因此没有办法释放其内存。

然而有两种相关的简单方法来修复这个问题。首先，在 try block 外声明 john，因此其不会在 try block 退出时离开作用域:

```cpp
#include <iostream>

Person* john{ nullptr };

try
{
    john = new Person("John", 18, PERSON_MALE);
    processPerson(john);
}
catch (const PersonException& exception)
{
    std::cerr << "Failed to process person: " << exception.what() << '\n';
}

delete john;
```

第二种方法是使用一个本地的类变量，其明白在离开作用域时改如何清理自身（通常称为智能指针）。标准库为了这个目的提供了一个名为 `std::unique_ptr` 的类。**std::unique_ptr** 是一个模板类其存储一个指针，在离开作用域时释放该指针。

```cpp
#include <iostream>
#include <memory> // std::unique_ptr

try
{
    auto* john { new Person("John", 18, PERSON_MALE) };
    std::unique_ptr<Person> upJohn { john }; // upJohn 现在拥有 john

    ProcessPerson(john);

    // 当 upJohn 离开作用域，它会删除 john
}
catch (const PersonException& exception)
{
    std::cerr << "Failed to process person: " << exception.what() << '\n';
}
```

下一章节将会详细讲解智能指针。

异常与析构函数

不同于构造函数，抛出异常在表示对象创建没有成功时非常的有用，但是异常应该永远不在析构函数中抛出。
问题出现在当一个异常从析构函数中抛出时的栈展开过程中。编译器会进入一种状态不知道是继续栈展开过程还是处理新的异常。最终的结果就是程序立刻被终结。
规则：析构函数不应该抛出异常。

性能考虑

异常确实具有小的性能成本。它们增加了可执行的大小，同时由于需要执行额外的检查，这也可能会造成运行变慢。然而异常所带来的主要性能惩罚适当其真实被抛出的时候。这种情况下，栈必须被展开同时查找合适的异常处理，也就是一个相对昂贵的操作。
作为记录，一些现代计算机架构支持被称为零成本异常的异常模型。如果支持的话，就没有额外的运行时成本在非错误情况（也是最关心性能的地方）。然而，它们在发现异常时带来了更大的惩罚。

何时使用异常

异常处理的最佳使用是当以下条件皆满足时：

- 低频率的出错
- 错误很严峻，且不处理就没法继续
- 错误不能在其出现处被处理
- 没有其他的方案来返回错误码给调用者

20.9 Exception specifications and noexcept

C++ 中的所有函数都被归类为**非抛出** non-throwing（不抛出异常）或者是**潜在抛出** potentially throwing（有可能抛出异常）。
考虑以下函数声明：

```
1 int doSomething(); // 这个函数抛出异常还是不会?
```

观察一个通常的函数定义不可能判断的了其是否会抛出异常。虽然注释可能有用，但是文档并不可以并且编译器也不能执行注释。
异常说明 exception specifications 是一种最初设计用来记录何种类型的异常可能会被函数抛出的语言机制。虽然大部分的异常说明已经被废弃或者移除了，一个有用的异常说明添加进来作为代替，即本章节覆盖的内容。

noexcept 说明符

noexcept 说明符定义一个函数为 non-throwing。

```
1 void doSomething() noexcept; // 该函数为 non-throwing
```

注意 `noexcept` 并不实际阻止函数抛出异常或者调用其它潜在抛出异常的函数。相反，当一个异常被抛出，如果异常退出 noexcept 函数，`std::terminate` 则会被调用。同样注意如果

`std::terminate` 在 noexcept 函数中被调用，栈展开有可能不出现（根据实现与优化），这就意味着对象有可能不能被正确销毁。
就像函数因为不同的返回值不能重新，函数的异常说明不同也不能被重写。

带有布尔参数的 noexcept 说明符

`noexcept(true)` 等同于 `noexcept` ，意为函数为 non-throwing。`noexcept(false)` 意为函数为潜在 throwing。这些参数通常只会用在模板函数中，所以模版函数可以通过一些参数值动态的被创建为 non-throwing 或者潜在 throwing。

哪个函数是 non-throwing 以及 potentially-throwing 的

函数是隐式 non-throwing 的：

- 析构函数

函数默认为 non-throwing 的隐式声明或默认函数：

- 构造函数：默认，拷贝，移动
- 赋值：拷贝，移动
- 比较操作符（从 C++20 开始）

然而，如果这些函数调用（无论显式或隐式）了其他函数为潜在 throwing 的，那么它们也会被视为潜在 throwing。例如，如果一个类的数据成员拥有潜在 throwing 的构造函数，那么该类的构造函数也会被视为潜在 throwing。另一个例子，如果拷贝赋值操作符调用了前者 throwing 的赋值操作符，那么拷贝赋值也将会变成潜在 throwing。
函数为潜在 throwing（如果不是隐式声明或默认的）：

- 普通函数
- 用户定义的构造函数
- 用户定义的操作符

noexcept 操作符

noexcept 操作符同样也可以在函数中使用。其获取一个表达式作为参数，根据编译器判断是否会抛出异常来返回 `true` 或 `false` 。noexcept 操作符会在编译期进行静态检查，且不会实际的计算输入的表达式。

```cpp
void foo() {throw -1;}
void boo() {};
void goo() noexcept {};
struct S{};

constexpr bool b1{ noexcept(5 + 3) }; // true; ints 为 non-throwing
constexpr bool b2{ noexcept(foo()) }; // false; foo() 抛出异常
constexpr bool b3{ noexcept(boo()) }; // false; boo() 为隐式 noexcept(false)
constexpr bool b4{ noexcept(goo()) }; // true; goo() 为显式 noexcept(true)
constexpr bool b5{ noexcept(S{}) };   // true; 机构提默认构造函数是 noexcept
```

noexcept 操作符可以用于根据是否为潜在 throwing 有条件的执行代码。这是需要满足某些**异常安全保证**。

异常安全保证

异常安全保证是一种合规指南用于保证函数或类如何在异常出现时做出的行为。有四种等级的异常安全：

- 无保证 – 在异常抛出时没有保证（例如一个类处于不可用状态）
- 基础保证 – 如果异常抛出，没有内存泄漏同时对象仍然不可用，但是程序有可能处于修改状态
- 强保证 – 如果异常抛出，没有内存泄漏同时程序状态不会被改变。这意味着函数必须是完全成功或者说失败后没有副作用。也就是说失败发生之前任何被修改的都可以回滚到之前的状态。
- 无抛出/无失败 – 函数总是成功（no-fail）或者失败也不抛出异常（no-throw）

现在来看 no-throw/no-fail 保证的细节：
no-throw 保证：如果函数失败，那么不会抛出异常。而是返回错误码或者忽略问题。在栈展开期间，当一个异常已经被处理了，则需要 no-throw 保证；例如，所有的析构函数应该都有 no-throw 保证（其内部调用的任何函数也是一样）。
例如以下应该为 no-throw：

- 析构函数与内存释放/清除函数
- 调用函数更高一层的函数需要 no-throw 时

no-fail 保证：函数总能成功（因此拥有不需要抛出异常，因此 no-fail 稍微比 no-throw 的形式更强）。
例如以下应该为 no-fail：

- 移动构造函数以及移动赋值（移动语义，在 15 章中覆盖）
- swap 函数
- 容器的 clear/erase/reset 函数
- 对 `std::unique_ptr` 的操作（同样在 15 章覆盖）
- 调用函数更高一层的函数需要 no-fail 时

何时使用 noexcept

即使代码不显式的抛出异常，并不意味着应该在代码中加上 `noexcept` 。默认情况下，大多数函数都是潜在 throwing 的，因此如果函数调用其他函数，它们本身就是潜在 throwing 的，因此符合自身潜在 throwing。
标记函数为 non-throwing 有一些不错的原因：

- non-throwing 函数可以被安全的调用在非异常安全的函数中，例如析构函数
- noexcept 的函数可以允许编译期提供更好的优化。因为一个 noexcept 函数不能在其外部抛出异常，编译器不需要担心维护运行时的栈与展开状态，这样使得代码变得更快。
- 有几种情况下知道函数为 noexcept 时可以生产高性能实现的代码：标准库容器（例如 `std::vector` ）就是 noexcept 的，并且使用 noexcept 操作符判定使用 `move semantics`（更快）还是 `copy semantics` （更慢）。

标准库使用 `noexcept` 的方针仅存在于*必须不能*抛出异常或失败。函数是潜在 throwing 的但是实际并没有抛出异常（因为实现的原因）通常不被标记为 `noexcept` 。
对于用户的代码，有两个地方使用 `noexcept` 有意义：

- 构造函数以及重载的赋值操作符为 no-throw（获取性能优化）
- 函数希望表达 no-throw 或 no-fail 保证（记录它们可以被安全的用在析构函数或者其他 noexcept 函数）

最佳实践：当允许时，为构造函数与重载赋值操作符加上 `noexcept` 。其它希望表达 no-fail 或 no-throw 保证的函数也加上 `noexcept` 。
最佳实践：如果不能确定函数是否带有 no-fail 或 no-throw 的保证，那么谨慎起见不要标记其 `noexcept` 。若是移除原本的 noexcept 则会违反一个接口向用户承诺的行为功能，也可能会破坏已有代码。为了更强的保证，在之后给非 noexcept 函数添加 noexcept 是被视为安全的。

21 The Standard Template Library

21.1 The Standard Library

标准库包含了一系列模板化容器，算法，以及遍历器的类的集合。好的一面是用户可以利用这些类而无须自己编写以及调试，标准库提供合理高效版本的类可以胜任这些工作。坏的一面是标准库很复杂，并且由于所有东西都是模板化的看起来就很吓人。
幸运的是我们可以一点一点的学习标准库，按需学习。

21.2 STL containers overview

现如今 STL 库中使用的最普遍的功能就是 STL 容器类。STL 包含了很多不同的容器类可供用于不同的场景。通常来说，容器类有三种分类：序列容器，关联容器，以及容器适配器。

序列容器

序列容器是一种用于在容器中维护元素顺序的容器类。序列容器定义的特征是可以根据位置插入元素。最常见的例子就是数组。
从 C++11 开始，STL 包含 6 种序列容器：`std::vector` ，`std::deque` ，`std::array` ，`std::list` ，`std::forward_list` ，以及 `std::basic_string` 。
- **vector** 类允许通过操作符 [] 进行随机访问其中的元素，插入与移除 vector 末端的元素通常很快。

```
#include <vector>
#include <iostream>

int main()
{

    std::vector<int> vect;
    for (int count=0; count < 6; ++count)
        vect.push_back(10 - count); // insert at end of array

    for (int index=0; index < vect.size(); ++index)
        std::cout << vect[index] << ' ';

    std::cout << '\n';
}

```

- **deque**（发音“deck”）是一个双端队列类，实现为动态数组可以从两端扩展

```
#include <iostream>
#include <deque>

```

```
4  int main()
5  {
6      std::deque<int> deq;
7      for (int count=0; count < 3; ++count)
8      {
9          deq.push_back(count); // insert at end of array
10         deq.push_front(10 - count); // insert at front of array
11     }
12
13     for (int index=0; index < deq.size(); ++index)
14         std::cout << deq[index] << ' ';
15
16     std::cout << '\n';
17 }
18
```

- **list** 是一种特殊类型的序列容器，被称为双向链表，即容器中的每个元素包含了指向上一个元素以及下一个元素的指针。链表仅提供访问两端的功能 – 没有提供随机访问。如果想要找到中间的值，则需要从某一段开始遍历直至达到希望找到的元素。链表的优势是在知道位置时，插入元素非常的快。之后的章节将会详细讲解。
- **string** 可以被视为字符类型（或 wchar）的向量。

关联容器

关联容器会自动对输入进其的元素进行排序。默认情况下，关联容器通过操作符 < 进行元素比较。
- **set** 是一种用于存储唯一元素的容器。根据容器中元素的值进行的排列。
- **multiset** 是允许重复元素的 set。
- **map**（也被称为关联数组）是一个 set，其中每个元素都是一对，被称为键/值。键用于排序和索引，必须唯一；值就是数据。
- **multimap**（也被称为字典）是一个允许重复键的 map。

容器适配器

容器适配器是特殊的预定义的容器，用于适配指定的用法。这其中有趣的是用户可以选择哪种序列容器被使用。
- **stack** 是一种容器，其元素操作为 LIFO（后进先出），元素在容器的尾部被插入（pushed）与移除（popped）。Stack 使用 deque 作为其默认序列容器（看起来很奇怪，因为 vector 看上去更符合直觉），不过也可以使用 vector 或 list。
- **queue** 是一种容器，其元素操作为 FIFO（先进先出），元素在容器尾部插入（pushed）并在头部移除（popped）。Queue 使用 deque 作为其默认实现，不过也可以使用 list。

- **priority queue** 是一种元素维持在排序（通过操作符 <）状态下的 queue。当元素被插入，该元素会在 queue 中被排序。移除头部元素返回的则是最高优先级的项。

21.3 STL iterators overview

迭代器 iterator 是一种对象可以穿过（遍历）一个容器类而不需要用户知道容器是如何实现的。很多类（特别是 list 和关联类），遍历器是主要的元素访问方式。
迭代器最好的可视化是作为一个指针指向给定容器中的元素，并带有一系列重载的操作符提供了良好的功能：

- **操作符 *** – 解引用迭代器返回迭代器当前指向的元素
- **操作符 ++** – 移动迭代器到容器中的下一个元素。大多数迭代器同样也提供操作符–用于移动到上一个元素。
- **操作符 == 与操作符!=** – 基础比较符用于判断两个迭代器是否指向同一个元素。为了比较两个值，首先需要解引用迭代器，接着使用比较操作符。
- **操作符 =** – 指派迭代器至新的位置（通常是容器元素的开始或结尾）。为了指派，首先需要解引用迭代器，接着使用指派操作符。

每个容器包含了四种基础成员函数用于使用操作符 =：

- **begin()** 返回一个迭代器表示容器的起始元素。
- **end()** 返回一个迭代器表示容器的末尾元素之后的位置。
- **cbegin()** 与 `begin()` 相同，只读。
- **cend()** 与 `end()` 相同，只读。

`end()` 不指向 list 的最后一个元素看起来有点奇怪，但是这主要是让循环更简单：遍历所有可继续的元素直到触及 `end()`，接着就知道完成了。
最后，所有容器提供（至少）两种类型的迭代器：

- **container::iterator** 提供读/写迭代器
- **container::const_iterator** 提供只读迭代器

遍历一个 vector：

```cpp
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> vect;
    for (int count=0; count < 6; ++count)
        vect.push_back(count);

    std::vector<int>::const_iterator it; // 声明一个只读迭代器
    it = vect.cbegin(); // 赋值 it 来开始 vector
    while (it != vect.cend()) // while it 直到触及结尾
        {
        std::cout << *it << ' '; // 打印所指向的元素
        ++it; // 迭代器指向下一个元素
        }

    std::cout << '\n';
}
```

遍历一个 list：

```cpp
#include <iostream>
#include <list>

int main()
{

    std::list<int> li;
    for (int count=0; count < 6; ++count)
        li.push_back(count);

    std::list<int>::const_iterator it; // 声明一个迭代器
    it = li.cbegin(); // 赋值 it 来开始 list
    while (it != li.cend()) // while it 直到触及结尾
    {
        std::cout << *it << ' '; // 打印所指向的元素
        ++it; // 迭代器指向下一个元素
    }

    std::cout << '\n';
}
```

遍历一个 set：

```cpp
#include <iostream>
#include <set>

int main()
{
```

```
    std::set<int> myset;
    myset.insert(7);
    myset.insert(2);
    myset.insert(-6);
    myset.insert(8);
    myset.insert(1);
    myset.insert(-4);

    std::set<int>::const_iterator it; // 声明一个迭代器
    it = myset.cbegin(); // 赋值 it 来开始 set
    while (it != myset.cend()) // while it 直到触及结尾
    {
        std::cout << *it << ' '; // 打印所指向的元素
        ++it; // 迭代器指向下一个元素
    }

    std::cout << '\n';
}
```

遍历 map 会有一点棘手，maps 和 multimaps 获取的是一对元素（定义为 `std::pair`），可以使用 `make_pair()` 帮助函数来创建对，`std::pair` 允许通过 `first` 与 `second` 成员函数来访问元素。

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

int main()
{
    std::map<int, std::string> mymap;
    mymap.insert(std::make_pair(4, "apple"));
    mymap.insert(std::make_pair(2, "orange"));
    mymap.insert(std::make_pair(1, "banana"));
    mymap.insert(std::make_pair(3, "grapes"));
    mymap.insert(std::make_pair(6, "mango"));
    mymap.insert(std::make_pair(5, "peach"));

    auto it{ mymap.cbegin() }; // 声明一个 const 迭代器并赋值 it 来开始 vector
    while (it != mymap.cend()) // while it 直到触及结尾
    {
        std::cout << it->first << '=' << it->second << ' '; // 打印所指向的元素
        ++it; // 迭代器指向下一个元素
    }

    std::cout << '\n';
}
```

21.4 STL algorithms overview

除了容器类与迭代器，STL 也提供了很多泛型算法用于处理容器类中的元素。这些算法提供了比如查询，排序，插入，重排，移除，以及拷贝容器类中的元素。
注意算法被实现为用来操作迭代器的函数。这就意味着每个算法仅需要被实现一次，它便会自动的对所有提供了迭代器的容器生效（包括自定义容器类）。虽然这是非常强大的且带来了快速编写复杂代码的能力，但是它同样也有黑暗面：一些算法的组合与容器类型可能是无效的，可能导致无限循环，或者是有效但性能极差。因此使用它们需要承担风险。
STL 提供不少的算法 – 这里只介绍一些常用的，其它的将留在新的章节讲述（暂未实现）。
要使用 STL 算法仅需引入 `<algorithm>` 头文件。

min_element 与 max_element

`std::min_element` 与 `std::max_element` 算法查询容器类中最小和最大的元素。`std::iota` 生成一个连续序列的值。

```
#include <algorithm> // std::min_element 与 std::max_element
#include <iostream>
#include <list>
#include <numeric> // std::iota

int main()
{
    std::list<int> li(6);
    // 从 0 开始填充 li
    std::iota(li.begin(), li.end(), 0);

    std::cout << *std::min_element(li.begin(), li.end()) << ' '
              << *std::max_element(li.begin(), li.end()) << '\n';

    return 0;
}
```

find (list::insert)

下面的例子中使用 `std::find()` 算法来查找 list 类中的值，接着使用 `list::insert()` 函数在该值的位置添加新值。

```
#include <algorithm>
#include <iostream>
#include <list>
#include <numeric>

int main()
{
```

```
    std::list<int> li(6);
    std::iota(li.begin(), li.end(), 0);

    // 查找 list 中的值 3
    auto it{ std::find(li.begin(), li.end(), 3) };

    // 在 3 的位置之前插入 8
    li.insert(it, 8);

    for (int i : li) // 迭代器的 for 循环
        std::cout << i << ' ';

    std::cout << '\n';

    return 0;
}
```

打印:

```
0 1 2 8 3 4 5
```

排序与翻转

```
#include <iostream>
#include <vector>
#include <algorithm>

int main()
{
    std::vector<int> vect{ 7, -3, 6, 2, -5, 0, 4 };

    // vector 排序
    std::sort(vect.begin(), vect.end());

    for (int i : vect)
    {
        std::cout << i << ' ';
    }

    std::cout << '\n';

    // vector 翻转
    std::reverse(vect.begin(), vect.end());

    for (int i : vect)
    {
        std::cout << i << ' ';
    }

```

```
27      std::cout << '\n';
28
29      return 0;
30  }
```

打印:

```
1  -5 -3 0 2 4 6 7
2  7 6 4 2 0 -3 -5
```

另外，用户可以提供自定义的比较函数在 `std::sort` 的第三个参数的位置。`functional` 头文件中拥有若干比较函数可供使用。例如可以传递 `std::greater` 给 `std::sort` 接着移除 `std::reverse` 。vector 也会从高到底进行排序。

注意 `std::sort` 对于 list 容器类不起作用 – list 类提供了自身的 `sort()` 成员函数，其比通用的版本更加高效。

22 std::string

22.1 std::string and std::wstring

标准库包含了很多有用的类 – 但是可能最有用的就是 `std::string`（与 `std::wstring`）了，其身为一个字符串类提供了很多操作用于赋值、比较、以及修改字符串。本章开始深入讲解字符串类。
注意：C-style 字符串会被归类为“C-style strings”，而 `std::string`（与 `std::wstring`）被归为“strings”。

字符串类的动机

之前的章节中覆盖了 C-style strings，即使用字符数组来存储字符串的字符。如果尝试了它，将会很快的发现一个结论，那就是它相当的难用，容易变得混乱，并且难以调试。
C-style 字符串拥有很多短处，最主要都是围绕在必须手动管理内存这个问题上。例如赋值字符串“hello!”给缓存，那么首先需要动态分配正确长度的缓存：

```
1 char* strHello { new char[7] };
```

不要忘了计算额外的字符用作于空字符终结符！
接着需要拷贝值：

```
1 strcpy(strHello, "hello!");
```

更重要的是缓存必须足够大，不然就会溢出！
当然了因为字符串是动态分配的，当使用完成后记住必须要正确的释放掉它。

```
1 delete[] strHello;
```

记住是数组删除而不是普通删除！
此外很多 C 语言提供的操作符比如赋值与比较，并不在 C-style 字符串上生效。有时看起来生效了但是实际上产生的是错误结果 – 例如使用 `==` 来比较两个 C-style 字符串，实际上却是在做指针比较而不是字符串比较。使用操作符 = 赋值另一个 C-style 字符串看起来第一次生效了，但是实际上做的是指针拷贝（浅拷贝），并不如预期的那样。这些都会导致程序崩溃，而难以找到错误并进行调试！

String 概述

标准库中所有字符串的功能都位于 `<string>` 头文件中：

```
1 # include <string>
```

其中包含了 3 中不同的字符类。首先是一个名为 `basic_string` 的模板基类：

```
namespace std
{
    template<class charT, class traits = char_traits<charT>, class Allocator = allocator<charT>
     >
       class basic_string;
}
```

用户不需要直接使用该类，暂时无需担心 traits 或 Allocator。默认值猪狗应对大多数情况。
其中标准库中有两个推荐的 `basic_string`：

```
namespace std
{
    typedef basic_string<char> string;
    typedef basic_string<wchar_t> wstring;
}
```

这两个类才是用户真实使用的。`std::string` 用于标准 ascii 以及 utf-8 字符串；`std::wstring` 用于宽字符/unicode（utf-16）字符串。
尽管用户直接使用 `std::string` 与 `std::wstring`，然而所有的字符串功能都是实现在 `basic_string` 类中。`std::string` 与 `std::wstring` 可以直接通过虚函数来访问这些功能，因此所有展现在 `basic_string` 上的函数都可以在它们身上生效。然而因为 `basic_string` 是一个模板类，这也意味着当用户使用 string 或 wstring 时的语法错误，编译器会生成恐怖的模板错误。不要害怕这些错误，它们只是看上去可怕而已！
以下是字符串类的所有函数。大多数函数都有若干处理不同类型输入的功能，这些将会在下一节深入。

函数	效果
Creation and destruction	
(constructor)	Create or copy a string
(destructor)	Destroy a string
Size and capacity	
capacity()	Returns the number of characters that can be held without reallocation
empty()	Returns a boolean indicating whether the string is empty
length(), size()	Returns the number of characters in string
max_size()	Returns the maximum string size that can be allocated
reserve()	Expand or shrink the capacity of the string
Element access	
[], at()	Accesses the character at a particular index
Modification	
=, assign()	Assigns a new value to the string
+=, append(), push_back()	Concatenates characters to end of the string
insert()	Inserts characters at an arbitrary index in string
clear()	Delete all characters in the string
erase()	Erase characters at an arbitrary index in string
replace()	Replace characters at an arbitrary index with other characters
resize()	Expand or shrink the string (truncates or adds characters at end of string)
swap()	Swaps the value of two strings
Input and Output	
», getline()	Reads values from the input stream into the string
«	Writes string value to the output stream
c_str()	Returns the contents of the string as a NULL-terminated C-style string
copy()	Copies contents (not NULL-terminated) to a character array
data()	Same as c_str(). The non-const overload allows writing to the returned string.
String comparison	
==, !=	Compares whether two strings are equal/unequal (returns bool)
<, <=, > >=	Compares whether two strings are less than / greater than each other (returns bool)
compare()	Compares whether two strings are equal/unequal (returns -1, 0, or 1)
Substrings and concatenation	
+	Concatenates two strings
substr()	Returns a substring
Searching	
find()	Find index of first character/substring
find_first_of()	Find index of first character from a set of characters
find_first_not_of()	Find index of first character not from a set of characters
find_last_of()	Find index of last character from a set of characters
find_last_not_of()	Find index of last character not from a set of characters
rfind()	Find index of last character/substring
Iterator and allocator support	
begin(), end()	Forward-direction iterator support for beginning/end of string
get_allocator()	Returns the allocator
rbegin(), rend()	Reverse-direction iterator support for beginning/end of string

虽然标准库的字符串类提供了非常多的功能，但是有一些被省略了的功能却值得注意:

- 正则表达式的支持
- 从数字创建字符串的构造函数
- 首字母大写/大写/小写转换函数
- 大小写不敏感的比较
- 词语切分/分割字符成数组

- 获取字符串左侧或右侧部分的简单函数
- 空格修剪
- 标准化字符串 sprintf 风格
- utf-8 与 utf-16 间的互相转换

对于上述这些需求，需要用户自己编写函数，或者是转换字符串为 C-style 字符串（使用 `c_str()`）并使用 C 所提供的这些功能的函数。

22.2 std::string construction and destruction

本节将会讲述如何构造 `std::string` 对象，以及如何由数字创建 `std::string`，反之亦然。

字符串构造函数

字符串类拥有很多构造函数，让我们一个一个来看看。
注意：`string::size_type` 解析为 `size_t`，由 `sizeof` 操作符返回的结果等同于无符号整数类型。`size_t` 的真实大小取决于其环境。本教程中将其视为无符号整型。

string::string()

- 此为默认构造函数，其创建空字符串。

```
std::string sSource;
std::cout << sSource;

```

输出：

```


```

string::string(const string& strString)

- 此为拷贝构造函数，由 strString 拷贝创建一个新字符串。

```
std::string sSource{ "my string" };
std::string sOutput{ sSource };
std::cout << sOutput;

```

输出：

```
my string

```

string::string(const string& strString, size_type unIndex)

string::string(const string& strString, size_type unIndex, size_type unLength)

- 创建新字符串包含至多从 strString 而来的 unLength 大小的字符，由索引 unIndex 开始。如果遇到 NULL，字符串的拷贝则结束，即使没有触及 unLength。
- 如果没有提供 unLength，所有字符由 unIndex 开始使用。
- 如果 unIndex 大于字符串的长度，那么 out_of_range 异常将被抛出

```
std::string sSource{ "my string" };
std::string sOutput{ sSource, 3 };
std::cout << sOutput<< '\n';
std::string sOutput2(sSource, 3, 4);
std::cout << sOutput2 << '\n';

```

输出：

```
string
stri

```

string::string(const char* szCString)

- 从 C-style 字符串 szCString 创建新字符串，但不包括空终止符。
- 如果返回的大小超过了最大字符串长度，length_error 异常将被抛出。
- 警告：szCSting 必须不为空。

```
const char* szSource{ "my string" };
std::string sOutput{ szSource };
std::cout << sOutput << '\n';

```

输出：

```
my string

```

string::string(const char* szCString, size_type unLength)

- 从 C-style 字符串 szCString 的首个 unLength 字符创建字符串。
- 如果返回的大小超过了最大字符串长度，length_error 异常将被抛出。

- 警告：仅在这个函数，空值不被视为 szCString 中的终止字符串！这就意味着如果 unLength 足够大，可以读取字符串到最后！注意不要溢出字符串的缓存！

```
const char* szSource{ "my string" };
std::string sOutput(szSource, 4);
std::cout << sOutput << '\n';

```

输出：

```
my s

```

string::string(size_type nNum, char chChar)

- 通过字符 chChar 出现的 nNum 次数创建字符串。
- 如果返回的大小超过了最大字符串长度，length_error 异常将被抛出。

```
std::string sOutput(4, 'Q');
std::cout << sOutput << '\n';

```

输出：

```
QQQQ

```

template string::string(InputIterator itBeg, InputIterator itEnd)

- 由初始化范围字符 [itBeg, itEnd] 创建新字符串。
- 如果返回的大小超过了最大字符串长度，length_error 异常将被抛出。

没有示例代码，因为它的晦涩使得用户基本不会使用它。

string:: string()
字符串析构函数

- 析构函数。销毁字符串并释放内存。

没有示例代码，因为不会显式调用析构函数。

数值的构造函数

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

template <typename T>
inline std::string ToString(T tX)
{
    std::ostringstream oStream;
    oStream << tX;
    return oStream.str();
}

int main()
{
    std::string sFour{ ToString(4) };
    std::string sSixPointSeven{ ToString(6.7) };
    std::string sA{ ToString('A') };
    std::cout << sFour << '\n';
    std::cout << sSixPointSeven << '\n';
    std::cout << sA << '\n';
}
```

打印:

```
4
6.7
A
```

注意这个方法省略了任何错误检查。在 tX 插入 oStream 可能失败。合适的响应应该是在转换失败时抛出异常。

字符串转换成数值

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

template <typename T>
inline bool FromString(const std::string& sString, T& tX)
{
    std::istringstream iStream(sString);
    return !(iStream >> tX).fail(); // extract value into tX, return success or not
}

int main()
{
    double dX;
```

```
    if (FromString("3.4", dX))
        std::cout << dX << '\n';
    if (FromString("ABC", dX))
        std::cout << dX << '\n';
}
```

打印:

```
3.4
```

注意第二个转换失败了并返回 false。

22.3 std::string length and capacity

一旦创建了字符串，知道其长度是很有用的。这里就是长度和容量操作符的作用。我们也将讨论不同的芳芳来转换 `std::string` 回 C-style 字符串，这样可以使用它们通过期望 char* 类型的字符串函数。

字符串长度

size_type string::length() const
size_type string::size() const

- 这俩函数返回的都是当前字符串中的字符数量，不包含空值终止符。

```
std::string s { "012345678" };
std::cout << s.length() << '\n';

```

输出:

```
9

```

bool string::empty() const

- 如果字符串中无字符则返回 true，反之 false。

```
std::string string1 { "Not Empty" };
std::cout << (string1.empty() ? "true" : "false") << '\n';
std::string string2; // empty
std::cout << (string2.empty() ? "true" : "false")  << '\n';

```

输出:

```
false
true

```

字符串容量

字符串容量反应了其被分配了多少的内存用于存储其内容。该值基于字符串字符为单位测量，不包含空值终止符。

size_type string::capacity() const

- 返回字符串不需要重新分配内存时的可用于存储字符的数量。

```
std::string s { "01234567" };
std::cout << "Length: " << s.length() << '\n';
std::cout << "Capacity: " << s.capacity() << '\n';

```

输出：

```
Length: 8
Capacity: 15

```

实际上重新分配是不好的原因如下：
首先，重新分配字符串相对而言比较昂贵。新的内存需要被分配，每个字符需要被拷贝到新的内存。如果字符串巨大，则会消耗大量的时间，最后旧内存需要被释放。如果重新分配很多次，这个过程将显著的减慢程序的运行。
其次，任何时候当一个字符串被重新分配，字符串的内容改成了新的内存地址。这意味着该字符串的所有引用，指针，以及迭代器都变为无效！
void string::reserve()
void string::reserve(size_type unSize)

- 第二种函数为字符串设置了至少 unSize 容量。注意这可能需要重新分配。
- 如果第一种函数被调用，或者第二种函数调用的 unSize 小于当前容量，函数讲尝试缩容来匹配长度。缩容可能会被无视，取决于其实现。

```
std::string s { "01234567" };
std::cout << "Length: " << s.length() << '\n';
std::cout << "Capacity: " << s.capacity() << '\n';

s.reserve(200);
std::cout << "Length: " << s.length() << '\n';
std::cout << "Capacity: " << s.capacity() << '\n';

s.reserve();
std::cout << "Length: " << s.length() << '\n';
std::cout << "Capacity: " << s.capacity() << '\n';
```

打印:

```
Length: 8
Capacity: 15
Length: 8
Capacity: 207
Length: 8
Capacity: 207
```

22.4 std::string character access and conversion to C-style arrays

字符访问

字符串访问字符有两个几乎相同的方法。最简单快捷的使用方式是重载操作符 []:

char& string::operator[](
size_type nIndex)
const char& string::operator[](size_type nIndex) const

- 这俩函数返回索引 nIndex 的字符
- 传入无效索引将导致未定义行为
- 因为 char& 是返回类型，用户可以使用它来编辑数字中的字符

```
std::string sSource{ "abcdefg" };
std::cout << sSource[5] << '\n';
sSource[5] = 'X';
std::cout << sSource << '\n';

```

输出:

```
f
abcdeXg

```

转换成 C-style 数组

很多函数（包括所有的 C 函数）预期字符串被格式化为 C-style 字符串而不是 `std::string` 。因为这个原因，`std::string` 提供了 3 种不同的方法来进行转换:

const char* string::c_str() const

- 返回包含了字符串内容的 const C-style 字符串
- 空值终止符被附加

- 由 `std::string` 所有的 C-style 字符串不应该被删除

```
#include <cstring>

std::string sSource{ "abcdefg" };
std::cout << std::strlen(sSource.c_str());

```

输出：

```
7

```

const char* string::data() const

- 返回包含了字符串内容的 const C-style 字符串
- 空值终止符被附加。本函数执行的行为等同于 `c_str()`
- 由 `std::string` 所有的 C-style 字符串不应该被删除

```
#include <cstring>

std::string sSource{ "abcdefg" };
const char* szString{ "abcdefg" };
// memcmp compares the first n characters of two C-style strings and returns 0 if
  they are equal
if (std::memcmp(sSource.data(), szString, sSource.length()) == 0)
    std::cout << "The strings are equal";
else
    std::cout << "The strings are not equal";

```

输出：

```
The strings are equal

```

size_type string::copy(char* szBuf, size_type nLength, size_type nIndex = 0) const

- 拷贝至多 nLength 字符的 szBuf 字符串，开始于 nIndex 的字符
- 返回被拷贝字符的数量
- 非空不被附加。由调用者确保 szBuf 初始化了 NULL 或者通过使用返回长度来确定字符串
- 调用者负责 szBuf 不溢出

```
std::string sSource{ "sphinx of black quartz, judge my vow" };

char szBuf[20];
int nLength{ static_cast<int>(sSource.copy(szBuf, 5, 10)) };
szBuf[nLength] = '\0';  // Make sure we terminate the string in the buffer

std::cout << szBuf << '\n';

```

输出：

```
black

```

应该避免使用该函数，因为相对来说它更危险（因为由调用者来提供空值终止符以及避免溢出）。

22.5 std::string assignment and swapping

字符串赋值

给字符串赋值的最简单方法是使用重载操作符 = 函数。同样也有一个成员函数 `assign()` 复制了一些功能。

string& string::operator=(const string& str)
string& string::assign(const string& str)
string& string::operator=(const char* str)
string& string::assign(const char* str)
string& string::operator=(char c)

- 这些函数指派不同类型的值给字符串
- 这些函数返回 *this 以便它们可以被串联
- 注意没有单个字符的 assign() 函数

```
std::string sString;

// Assign a string value
sString = std::string("One");
std::cout << sString << '\n';

const std::string sTwo("Two");
sString.assign(sTwo);
std::cout << sString << '\n';

// Assign a C-style string
sString = "Three";
```

```
13  std::cout << sString << '\n';
14
15  sString.assign("Four");
16  std::cout << sString << '\n';
17
18  // Assign a char
19  sString = '5';
20  std::cout << sString << '\n';
21
22  // Chain assignment
23  std::string sOther;
24  sString = sOther = "Six";
25  std::cout << sString << ' ' << sOther << '\n';
26
```

输出：

```
1  One
2  Two
3  Three
4  Four
5  5
6  Six Six
7
```

`assign()` 成员函数还有一些其他的样式：

string& string::assign(const string& str, size_type index, size_type len)

- 赋值 str 的子字符串，从 index 开始，长度为 len
- 如果索引超出边界，抛出 out_of_range 异常
- 返回 *this 以便可以被串联

```
1  const std::string sSource("abcdefg");
2  std::string sDest;
3
4  sDest.assign(sSource, 2, 4); // assign a substring of source from index 2 of
     length 4
5  std::cout << sDest << '\n';
6
```

输出：

```
1  cdef
2
```

string& string::assign(const char* chars, size_type len)

- 从 C-style 字符数组中赋值 len 长的字符
- 如果结果超出字符最大数值，抛出 length_error 异常
- 返回 *this 以便可以被串联

```
std::string sDest;

sDest.assign("abcdefg", 4);
std::cout << sDest << '\n';

```

输出：

```
abcd

```

该函数有潜在风险，不建议使用

string& string::assign(size_type len, char c)

- 赋值 len 长度的字符 c
- 如果结果超出字符最大数值，抛出 length_error 异常
- 返回 *this 以便可以被串联

```
std::string sDest;

sDest.assign(4, 'g');
std::cout << sDest << '\n';

```

输出：

```
gggg

```

Swapping

如果拥有两个字符串并想让它们值互换，有两个名为 `swap()` 的函数可供使用：

void string::swap(string& str)
void swap(string& str1, string& str2)

- 两个函数可以互换字符串的值。成员函数 swaps 的是 *this 与 str，全局函数 swaps 的是 str1 与 str2

- 这俩函数效率高，互换字符的场景下不要使用赋值，而是这俩函数

```
std::string sStr1("red");
std::string sStr2("blue");

std::cout << sStr1 << ' ' << sStr2 << '\n';
swap(sStr1, sStr2);
std::cout << sStr1 << ' ' << sStr2 << '\n';
sStr1.swap(sStr2);
std::cout << sStr1 << ' ' << sStr2 << '\n';

```

输出：

```
red blue
blue red
red blue

```

22.6 std::string appending

在字符串末尾 appending 字符串可以简单的由操作符 +=，`append()` 或者 `push_back()` 来实现。

string& string::operator+=(const string& str)
string& string::append(const string& str)

- 俩函数可在字符串末尾添加 str 字符串
- 俩函数返回的都是 *this 以便被串联
- 如果返回超过了字符串的最大数量则抛出 length_error 异常

```
std::string sString{"one"};

sString += std::string{" two"};

std::string sThree{" three"};
sString.append(sThree);

std::cout << sString << '\n';

```

输出：

```
one two three

```

还有一种风格的 `append()` 可以 append 子字符串：

string& string::append(const string& str, size_type index, size_type num)

- 函数 append 起始于 index 共 num 长的字符给字符串
- 俩函数返回的都是 *this 以便被串联
- 如果 index 超出边界抛出 out_of_range 异常
- 如果返回超过了字符串的最大数量则抛出 length_error 异常

```
std::string sString{"one "};

const std::string sTemp{"twothreefour"};
sString.append(sTemp, 3, 5); // append substring of sTemp starting at index 3 of
  length 5
std::cout << sString << '\n';

```

输出：

```
one three

```

操作符 += 与 `append()` 也有 C-style 字符串的版本：

string& string::operator+= (const char* str)
string& string::append (const char* str)

- 俩函数可在字符串末尾添加 str 字符串
- 俩函数返回的都是 *this 以便被串联
- 如果返回超过了字符串的最大数量则抛出 length_error 异常
- str 不能为 NULL

```
std::string sString{"one"};

sString += " two";
sString.append(" three");
std::cout << sString << '\n';

```

输出：

```
one two three

```

同样风格的 `append()` 也对 C-style 字符串起作用：

string& string::append(const char* str, size_type len)

- append 起始 len 个字符至字符串
- 俩函数返回的都是 *this 以便被串联
- 如果返回超过了字符串的最大数量则抛出 length_error 异常
- 忽略特殊字符（包括”）

```
std::string sString{"one "};

sString.append("threefour", 5);
std::cout << sString << '\n';

```

输出：

```
one three

```

该函数很危险，并不推荐使用。

同样还有一系列的函数用于 append 字符。注意用于 append 字符的非操作符函数的名称为 `push_back()`，而不是 `append()`！

string& string::operator+=(char c)
void string::push_back(char c)

- 俩函数 append 字符 c 至字符串
- 操作符 += 返回 *this，以便被串联
- 如果返回超过了字符串的最大数量则俩函数都会抛出 length_error 异常

```
std::string sString{"one"};

sString += ' ';
sString.push_back('2');
std::cout << sString << '\n';

```

输出：

```
one 2

```

现在或许会疑惑为什么函数是 `push_back()` 而不是 `append()`。
这是因为栈的命名习惯，`push_back()` 是添加单个项至栈的函数。如果预想一个字符串是一个字符的栈，使用 `push_back()` 在字符串末尾添加一个字符是有道理的。然而缺少 `append()` 函数确是与习俗不一致的！
实际上对于字符的 `append()` 函数，看上去像是这样：

string& string::append(size_type num, char c)

- 添加字符 c 出现的 size_type 次数
- 俩函数返回的都是 *this 以便被串联
- 如果返回超过了字符串的最大数量则抛出 length_error 异常

```
std::string sString{"aaa"};

sString.append(4, 'b');
std::cout << sString << '\n';

```

输出：

```
aaabbbb

```

最后一种风格的 `append()` 用作于迭代器：

string& string::append(InputIterator start, InputIterator end)

- 从 [start, end) 范围中 append 所有的字符
- 俩函数返回的都是 *this 以便被串联
- 如果返回超过了字符串的最大数量则抛出 length_error 异常

22.7 std::string inserting

插入字符至已有的字符串可以通过 `insert()` 函数实现：

string& string::insert(size_type index, const string& str)
string& string::insert(size_type index, const char* str)

- 俩函数插入字符 str 在字符串的 index 位置
- 俩函数都返回 *this 以便被串联
- 如果 index 无效，俩函数都会抛出 out_of_range

- 如果结果超过了字符串最大数量，俩函数都会抛出 length_error 异常
- C-style 版本中 str 不可为 NULL

```
string sString("aaaa");
cout << sString << endl;

sString.insert(2, string("bbbb"));
cout << sString << endl;

sString.insert(4, "cccc");
cout << sString << endl;

```

输出：

```
aaaa
aabbbbaa
aabbccccbbaa

```

以下版本的 `insert()` 允许用户在任意位置插入子字符串：

string& string::insert(size_type index, const string& str, size_type startindex, size_type num)

- 该函数插入 num 字符的 str，始于 startindex，插入进 string 的 index 位置
- 返回 *this 以便被串联
- 如果 index 或 startindex 超出边界则抛出 out_of_range
- 如果结果超出字符串的最大长度则抛出 length_error 异常

```
string sString("aaaa");

const string sInsert("01234567");
sString.insert(2, sInsert, 3, 4); // insert substring of sInsert from index [3,7)
  into sString at index 2
cout << sString << endl;

```

输出：

```
aa3456aa

```

另外还有 `insert()` 多次插入同样的字符：

string& string::insert(size_type index, size_type num, char c)

- 插入 num 数量的 c 字符至 index 位置
- 返回 *this 以便被串联
- 如果 index 无效，则抛出 out_of_range 异常
- 如果结果超出字符串的最大长度则抛出 length_error 异常

```
string sString("aaaa");

sString.insert(2, 4, 'c');
cout << sString << endl;

```

输出：

```
aaccccaa

```

最后还有使用迭代器的三个不同版本的 `insert()`：

void insert(iterator it, size_type num, char c)
iterator insert(iterator it, char c)
void string::insert(iterator it, InputIterator begin, InputIterator end)

- 第一个函数在迭代之前，插入 num 数量的 c 字符
- 第二个函数在迭代之前，插入单个字符 c，并返回插入位置的迭代器
- 第三个函数在迭代之前，在 $[begin, end)$ 之间插入所有字符
- 如果结果超出字符串最大长度，所有函数抛出 length_error 异常

23 Input and Output (I/O)

23.1 Input and output (IO) streams

输入与输出的功能并没有定义在 C++ 语言的核心库中，而是通过 C++ 标准库来提供（并因此存在于 std 命名空间中）。之前的章节中已经引入了很多次 iostream 库头文件并使用了 cin 与 cout 对象来进行简单的 I/O。这一节中我们将更为深入的了解其中的细节。

iostream 库

当用户引入了 iostream 头文件时可以访问整个提供了 I/O 功能所有的类（包括一个已经命名为 iostream 的类）。
这些类的等级体系是由若干继承构成的（也是之前讲述过尽可能需要避免的），而 iostream 库是被精心设计并得到了极致的测试，就是为了避免任何典型的多重继承问题，因此用户可以放心的使用它们。

Streams

抽象的角度，**流** stream 是一系列的字节可用作于序列式的访问。随着时间流逝，一个流可能会潜在的生成或消费无限数量的数据。
通常而言会处理两种不同类型的刘。**输入流** input streams 用于存储数据生产者的数据输入，例如键盘，文件，或者网络。例如用户可能在键盘上按下一个键而程序当前并未预期有任何的输入。相比于忽视用户的输入，数据则是被放入输入流中，直到程序准备好使用该数据。
相反，**输出流** output streams 用作于存储特定数据消费者的输出，例如一个监控器，文件，或者是打印机。当将数据写入输出设备时，设备有可能并未对数据接收就绪 – 例如在程序写入数据只输出流时，打印机可能仍然在启动。数据将会留存在输出流中，直到打印机开始消费它。
某些设备，例如文件或者网络，能够同时作为输入与输出源。
对于程序员而言，刘的好处是只需要学习如何与流交互，便可以与不同类型设备进行数据读写。而流与真实设备是如何挂钩的则交给运行环境或者操作系统来操心。

C++ 中的输入/输出

尽管 ios 类通常是从 ios_base 派生而来，而 ios 是用户直接使用的最基础的类。ios 类定义了一大堆东西是通用于输入与输出流的。之后的章节将会详细进行了解。
istream 类是处理输入流的主要类。通过输入流，**提取操作符（»）** extraction operator» 用于从流中移除值。这有道理：当用户在键盘上按下按键，键代码被放置进输入流中。接着程序从流中提取该值，使其可被使用。

ostream 类是处理输出流的主要类。通过输出流，**插入操作符（«）** insertion operator« 用于放置值值流中。这同样有道理：用户插入值至流中，接着数据消费者（例如显示器）使用它们。**iostream** 类可以同时处理输入与输出，允许双向 I/O。

C++ 中的标准 streams

标准流是一个预连接的流提供给计算机程序其环境。C++ 提供了四种标准流对象。

1. cin – 一个 istream 对象与标准输入关联（通常而言是键盘）
2. cout – 一个 ostream 对象与标准输出关联（通常而言是显示器）
3. cerr – 一个 ostream 对象与标准错误关联（通常而言是显示器），提供了非缓存的输出
4. clog – 一个 ostream 对象与标准错误关联（通常而言是显示器），提供了缓存输出

非缓存输出通常是立刻处理的，而缓存输出通常是存储起来并以块的方式输出。由于 clog 没有被经常使用，在标准流的列表中经常被省略。

23.2 Input with istream

iostream 库相当的复杂 – 因此我们不能在此教程中做到完全覆盖。然而我们会展示最常见的功能。本节将深入不同角度的输入类（istream）。

提取操作符

在之前的很多章节已经见到过了，我们可以使用提取操作符（»）来入去输入流中的信息。C++ 已经为所有内建数据类型预定义了提取操作符，其中我们也学习了如何为用户自定义的类进行重载提取操作符。
当读取字符串时，提取操作符中的一个最常见的问题就是如何避免输入超出缓存。例如：

```
1 char buf[10];
2 std::cin >> buf;
```

如果用户输入了 18 个字符呢？缓存会溢出，坏事发生。通常来说，对于用户输入多少字符做任何的预期都是一个坏的主意。
其中一个解决该问题的方案是使用调制器。**调制器** manipulator 是一个当应用了提取符（»）或插入（`setw` 位于 iomanip 头文件中）的流时，用作修改流的对象，其可用于限制字符读取进流的数量。使用 `setw()` 只需要简单的提供一个最大字符数作为读取的参数，并插入至输入声明中。

```
#include <iomanip>
char buf[10];
std::cin >> std::setw(10) >> buf;
```

该程序现在只会从流中读取前 9 个字符（为结束符留位置）。任何剩余的字符将被留在流中，直到下一次提取。

提取与空白字符

作为提醒，提取操作符跳过空白字符（空格，tabs，以及换行）。

```
int main()
{
    char ch;
    while (std::cin >> ch)
        std::cout << ch;

    return 0;
}
```

当用户进行以下输入：

```
Hello my name is Alex
```

提取操作符跳过空格与换行，输出：

```
HellomynameisAlex
```

大多数时候并不想抛弃空格，为此 istream 类提供了一些函数来帮助我们。其中一个最常用的是 **get()** 函数：

```
int main()
{
    char ch;
    while (std::cin.get(ch))
        std::cout << ch;

    return 0;
}
```

现在再输入：

```
Hello my name is Alex
```

那么输出则是：

```
Hello my name is Alex
```

`std::get()` 同样还有一个字符串版本，用作带有上限的字符读取：

```cpp
int main()
{
    char strBuf[11];
    std::cin.get(strBuf, 11);
    std::cout << strBuf << '\n';

    return 0;
}
```

现在输入：

```
Hello my name is Alex
```

其输出则是：

```
Hello my n
```

注意只读取了前 10 个字符（我们需要留一个字符给终止符），而剩下的字符仍然留在输入流中。`get()` 函数中有一点需要注意的是其不会读取换行字符！这会导致一些非预期的结果：

```cpp
int main()
{
    char strBuf[11];
    // Read up to 10 characters
    std::cin.get(strBuf, 11);
    std::cout << strBuf << '\n';

    // Read up to 10 more characters
    std::cin.get(strBuf, 11);
    std::cout << strBuf << '\n';
    return 0;
}
```

如果用户输入：

```
Hello!
```

程序输出：

```
Hello!
```

接着结束！为什么它不要求另外十个字符呢？这是因为第一个 `get()` 读取到换行接着停止了。第二个 `get()` 看到 cin 流中仍然有输入并尝试读取。但是第一个字符是换行，因此立刻停止。因此有另一个称为 **getline()** 的函数，其功能与 `get()` 一致，但是会读取换行。

```cpp
int main()
{
    char strBuf[11];
    // Read up to 10 characters
    std::cin.getline(strBuf, 11);
    std::cout << strBuf << '\n';

```

```
    // Read up to 10 more characters
    std::cin.getline(strBuf, 11);
    std::cout << strBuf << '\n';
    return 0;
}
```

这个代码正如预期，即使用户输入了带有换行的字符串。
如果需要知道上一次调用 `getline()` 提取了多少字符，使用 **gcount()**：

```
int main()
{
    char strBuf[100];
    std::cin.getline(strBuf, 100);
    std::cout << strBuf << '\n';
    std::cout << std::cin.gcount() << " characters were read" << '\n';

    return 0;
}
```

std::string 的一个特殊版本的 getline()

还有一个特殊版本的 `getline()` 位于 istream 类之外，用于读取 std::string 变量中的值。这个特殊的版本并不是 ostream 或是 istream 中的成员，它是位于 string 头文件中。

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

int main()
{
    std::string strBuf;
    std::getline(std::cin, strBuf);
    std::cout << strBuf << '\n';

    return 0;
}
```

一些额外的有用的 istream 函数

还有一些有用的输入函数用户可能需要会用到：

- **ignore()** 丢弃流中的第一个字符
- **ignore(int nCount)** 丢弃流中前 nCount 的字符
- **peek()** 允许从流中读取字符而不移除它
- **unget()** 返回最后一个字符并不消耗它，这样在下次调用时可以再次读取

- **putback(char ch)** 允许将一个字符放入流中，下次调用时读取

23.3 Output with ostream and ios

这一节将学习 iostream 输出类（ostream）的各个方面。

插入操作符

插入操作符（«）用于输入信息进入输出流中。C++ 为所有内建的数据类型预定义的插入操作，我们已经见到了是如何为自定义的类重载插入操作符的。
在 streams 小节中我们见到了 istream 与 ostream 被一个称为 ios 的类派生。ios（以及 ios_base）其中的一个职能就是控制输出的格式选项。

格式化

有两种方法用于修改格式化选项：flags 以及 manipulators。可以将 **flags** 视为布尔变量，即可开关；**manipulators** 则是对象用于放置在流中，影响输入与输出。
开启 flag 可以使用 **setf()** 函数，以及正确的 flag 作为参数。例如 C++ 默认不会打印 + 号在正数前。然而使用 `std::ios::showops` flag 可以修改这个行为：

```
std::cout.setf(std::ios::showpos); // turn on the std::ios::showpos flag
std::cout << 27 << '\n';
```

输出：

```
+27
```

使用 Bitwise OR（|）操作符也可以同时开启若干 ios flags：

```
std::cout.setf(std::ios::showpos | std::ios::uppercase); // turn on the std::ios::showpos and
    std::ios::uppercase flag
std::cout << 1234567.89f << '\n';
```

输出：

```
+1.23457E+06
```

关闭 flag 可以使用 **unsetf()** 函数：

```
std::cout.setf(std::ios::showpos); // turn on the std::ios::showpos flag
std::cout << 27 << '\n';
std::cout.unsetf(std::ios::showpos); // turn off the std::ios::showpos flag
std::cout << 28 << '\n';
```

输出：

```
+27
28
```

在使用 `setf()` 时有一个棘手地方需要注意。很多 flags 属于的组被称为格式组。**格式组** format group 是一组 flags 用于执行相似（有时为互斥）的格式选项。例如一个名为“basefield”的格式组包含了“oct”, “dec”以及“hex”用于控制整数值。默认为“dec”flag。因此，如果这样：

```
std::cout.setf(std::ios::hex); // try to turn on hex output
std::cout << 27 << '\n';
```

会得到以下输出：

```
27
```

它并没有生效！原因是因为 `setf()` 仅只是开启了 – 它并没有智能到将互斥的 flags 关闭。因此，当开启 `std::hex` 时，`std::ios::dec` 仍然是开启的，而明显 `std::ios::dec` 的优先级更高。有两种办法来解决这个问题。

首先可以关闭 `std::ios::dec` 使得 `std::hex` 生效。

```
std::cout.unsetf(std::ios::dec); // turn off decimal output
std::cout.setf(std::ios::hex); // turn on hexadecimal output
std::cout << 27 << '\n';
```

得到预期的输出：

```
1b
```

第二种方法就是使用另一种形式的 `setf()` 接受两个参数：第一个参数是用于设置的 flag，第二个是其所属的格式组。当使用这种形式的 `setf()`，所有属于该组的 flag 都会关闭，并仅留下传递进的 flag 开启。

```
// Turn on std::ios::hex as the only std::ios::basefield flag
std::cout.setf(std::ios::hex, std::ios::basefield);
std::cout << 27 << '\n';
```

这也可以得到预期的输出：

```
1b
```

使用 `setf()` 与 `unsetf()` 往往很尴尬，因此 C++ 提供了另一种方法来修改格式选项：manipulators。manipulators 的好处是它们足够的聪明来开启与关闭合适的 flags。

```
std::cout << std::hex << 27 << '\n'; // print 27 in hex
std::cout << 28 << '\n'; // we're still in hex
std::cout << std::dec << 29 << '\n'; // back to decimal
```

输出：

```
1b
1c
29
```

通常来说，使用 manipulators 比起设置 flags 而言简单了很多。很多选项都可以通过 flags 与 manipulators 设置（例如修改 base），然而其他选项仅能通过 flags 或是 manipulators，因此如何使用它们两者是很重要的。

有用的格式化

以下列举了一些更有用的 flags 与 manipulators 以及成员函数。flags 存在于 `std::ios` 类中，manipulators 存在于 `std` 命名空间中，成员函数存在于 `std::ostream` 类中。

Group	Flag	Meaning
	std::ios::boolalpha	If set, booleans print "true"or "false". If not set, booleans print 0 or 1

Manipulator	Meaning
std::boolalpha	Booleans print "true"or "false"
std::noboolalpha	Booleans print 0 or 1 *default*

例子:

```
std::cout << true << ' ' << false << '\n';

std::cout.setf(std::ios::boolalpha);
std::cout << true << ' ' << false << '\n';

std::cout << std::noboolalpha << true << ' ' << false << '\n';

std::cout << std::boolalpha << true << ' ' << false << '\n';
```

结果:

```
1 0
true false
1 0
true false
```

Group	Flag	Meaning
	std::ios::showpos	If set, prefix positive numbers with a +

Manipulator	Meaning
std::showpos	Prefixes positive numbers with a +
std::noshowpos	Doesn't prefix positive numbers with a +

例子:

```
std::cout << 5 << '\n';

std::cout.setf(std::ios::showpos);
std::cout << 5 << '\n';

std::cout << std::noshowpos << 5 << '\n';

std::cout << std::showpos << 5 << '\n';
```

输出:

```
5
+5
5
+5
```

Group	Flag	Meaning
	std::ios::uppercase	If set, uses upper case letters

Manipulator	Meaning
std::uppercase	Uses upper case letters
std::nouppercase	Uses lower case letters

例子:

```
std::cout << 12345678.9 << '\n';

std::cout.setf(std::ios::uppercase);
std::cout << 12345678.9 << '\n';

std::cout << std::nouppercase << 12345678.9 << '\n';

std::cout << std::uppercase << 12345678.9 << '\n';
```

结果:

```
1.23457e+007
1.23457E+007
1.23457e+007
1.23457E+007
```

Group	Flag	Meaning
std::ios::basefield	std::ios::dec	Prints values in decimal (default)
std::ios::basefield	std::ios::hex	Prints values in hexadecimal
std::ios::basefield	std::ios::oct	Prints values in octal
std::ios::basefield	(none)	Prints values according to leading characters of value

Manipulator	Meaning
std::dec	Prints values in decimal
std::hex	Prints values in hexadecimal
std::oct	Prints values in octal

例子:

```
std::cout << 27 << '\n';

std::cout.setf(std::ios::dec, std::ios::basefield);
std::cout << 27 << '\n';

std::cout.setf(std::ios::oct, std::ios::basefield);
std::cout << 27 << '\n';

std::cout.setf(std::ios::hex, std::ios::basefield);
std::cout << 27 << '\n';

std::cout << std::dec << 27 << '\n';
std::cout << std::oct << 27 << '\n';
std::cout << std::hex << 27 << '\n';
```

结果:

```
27
27
33
1b
27
33
1b
```

现在我们能够看到通过 flag 与通过 manipulators 设置之间的关系了。之后的例子中我们将使用 manipulators，除非它们不可用。

精度，注释，以及小数点

使用 manipulators（或 flags），能够修改精度以及浮点数展示的格式。有若干格式选线被结合成复杂的方式，让我们深入了解一下。

Group	Flag	Meaning
std::ios::floatfield	std::ios::fixed	Uses decimal notation for floating-point numbers
std::ios::floatfield	std::ios::scientific	Uses scientific notation for floating-point numbers
std::ios::floatfield	(none)	Uses fixed for numbers with few digits, scientific otherwise
std::ios::floatfield	std::ios::showpoint	Always show a decimal point and trailing 0's for floating-point values

Manipulator	Meaning
std::fixed	Use decimal notation for values
std::scientific	Use scientific notation for values
std::showpoint	Show a decimal point and trailing 0's for floating-point values
std::noshowpoint	Don't show a decimal point and trailing 0's for floating-point values
std::setprecision	(int) Sets the precision of floating-point numbers (defined in the iomanip header)

Member function	Meaning
std::ios_base::precision()	Returns the current precision of floating-point numbers
std::ios_base::precision(int)	Sets the precision of floating-point numbers and returns old precision

如果使用了固定的或者科学计数，精度决定小数部分应该展示多少位。注意如果精度小于有效数字的数量，数值将被四舍五入。

```
std::cout << std::fixed << '\n';
std::cout << std::setprecision(3) << 123.456 << '\n';
std::cout << std::setprecision(4) << 123.456 << '\n';
std::cout << std::setprecision(5) << 123.456 << '\n';
std::cout << std::setprecision(6) << 123.456 << '\n';
std::cout << std::setprecision(7) << 123.456 << '\n';

std::cout << std::scientific << '\n';
std::cout << std::setprecision(3) << 123.456 << '\n';
std::cout << std::setprecision(4) << 123.456 << '\n';
std::cout << std::setprecision(5) << 123.456 << '\n';
std::cout << std::setprecision(6) << 123.456 << '\n';
std::cout << std::setprecision(7) << 123.456 << '\n';
```

输出：

```
123.456
123.4560
123.45600
123.456000
123.4560000

1.235e+002
1.2346e+002
1.23456e+002
1.234560e+002
1.2345600e+002
```

如果使用的不是固定数或者科学计数，精度决定了应该显式多少个有效数字。同样的，如果精度小于有效数字的数量，数值将被四舍五入。

```
std::cout << std::setprecision(3) << 123.456 << '\n';
std::cout << std::setprecision(4) << 123.456 << '\n';
std::cout << std::setprecision(5) << 123.456 << '\n';
std::cout << std::setprecision(6) << 123.456 << '\n';
std::cout << std::setprecision(7) << 123.456 << '\n';
```

输出：

```
123
123.5
123.46
123.456
123.456
```

使用 showpoint manipulator 或者 flag，可以使流写下小数点以及尾随零。

```
std::cout << std::showpoint << '\n';
std::cout << std::setprecision(3) << 123.456 << '\n';
std::cout << std::setprecision(4) << 123.456 << '\n';
std::cout << std::setprecision(5) << 123.456 << '\n';
std::cout << std::setprecision(6) << 123.456 << '\n';
std::cout << std::setprecision(7) << 123.456 << '\n';
```

输出：

```
123.
123.5
123.46
123.456
123.4560
```

以下是一个总结：

Option	Precision	12345.0	0.12345
Normal	3	1.23e+004	0.123
	4	1.235e+004	0.1235
	5	12345	0.12345
	6	12345	0.12345
Showpoint	3	1.23e+004	0.123
	4	1.235e+004	0.1235
	5	12345.	0.12345
	6	12345.0	0.123450
Fixed	3	12345.000	0.123
	4	12345.0000	0.1235
	5	12345.00000	0.12345
	6	12345.000000	0.123450
Scientific	3	1.235e+004	1.235e-001
	4	1.2345e+004	1.2345e-001
	5	1.23450e+004	1.23450e-001
	6	1.234500e+004	1.234500e-001

长度，填充字符，以及校正

通常在打印数值时，数值边上是不会有任何空格的。然而可以在左侧或右侧校正打印。要这么做需要首先定义字段 width，即为即将输出的值定义输出空间。如果实际打印的数值小于字段 width，那么其将被校正居左或居右；如果实际数值大于字段 width，它会被切断 – 它将溢出字段。

Group	Flag	Meaning
std::ios::adjustfield	std::ios::internal	Left-justifies the sign of the number, and right-justifies the value
std::ios::adjustfield	std::ios::left	Left-justifies the sign and value
std::ios::adjustfield	std::ios::right	Right-justifies the sign and value (default)

Manipulator	Meaning
std::internal	Left-justifies the sign of the number, and right-justifies the value
std::left	Left-justifies the sign and value
std::right	Right-justifies the sign and value
std::setfill(char)	Sets the parameter as the fill character (defined in the iomanip header)
std::setw(int)	Sets the field width for input and output to the parameter (defined in the iomanip header)

Member function	Meaning
std::basic_ostream::fill()	Returns the current fill character
std::basic_ostream::fill(char)	Sets the fill character and returns the old fill character
std::ios_base::width()	Returns the current field width
std::ios_base::width(int)	Sets the current field width and returns old field width

为了使用以上这些格式化，首先需要设置字段 width。这可以通过 `width(int)` 成员函数实现，或者 `setw()` manipulator。注意居右是默认的。

```
std::cout << -12345 << '\n'; // print default value with no field width
std::cout << std::setw(10) << -12345 << '\n'; // print default with field width
std::cout << std::setw(10) << std::left << -12345 << '\n'; // print left justified
std::cout << std::setw(10) << std::right << -12345 << '\n'; // print right justified
std::cout << std::setw(10) << std::internal << -12345 << '\n'; // print internally justified
```

结果：

```
-12345
    -12345
-12345
    -12345
-    12345
```

还有一件值得注意的是 `width(int)` 与 `setw()` 只会影响下一次输出声明。它们不像是其它 flags/manipulators 那样持久性质的。

现在来看看填充字符的例子：

```
std::cout.fill('*');
std::cout << -12345 << '\n'; // print default value with no field width
std::cout << std::setw(10) << -12345 << '\n'; // print default with field width
std::cout << std::setw(10) << std::left << -12345 << '\n'; // print left justified
std::cout << std::setw(10) << std::right << -12345 << '\n'; // print right justified
std::cout << std::setw(10) << std::internal << -12345 << '\n'; // print internally justified
```

输出：

```
-12345
****-12345
-12345****
****-12345
-****12345
```

注意所有的空格都被填充上了填充字符。

23.4 Stream classes for strings

迄今为止所接触到的所有 I/O 例子都是 cout 写入或者 cin 读取。然而还有另外一套关于字符串流的类，允许用户对字符串进行插入（«）以及提取（»）操作符。类似于 istream 与 ostream，字符串流提供了一个缓存用于存储数据。然而不同于 cin 与 cout，这些流并没有连接到一个 I/O 管道（例如键盘，显示器灯）。字符串流的一个最主要使用时缓存用于展示的输出，或者处理一行一行的输入。

字符串有六种流类：istringstream（从 istream 派生），ostringstream（从 ostream 派生），以及 stringstream（从 iostream 派生）用于读取与写入普通字符宽度的字符串，wistringstream，wostringstream，以及 wstringstream 用于读取与写入宽字符的字符串。需要 #include `sstream` 头文件，才能使用 stringstreams。

有两种存储 stringstream 数据的方式：

1. 使用插入（«）操作符：

```
        std::stringstream os;
        os << "en garde!\n"; // insert "en garde!" into the stringstream
```

2. 使用 str(string) 函数来设置缓存的值：

```
        std::stringstream os;
        os.str("en garde!"); // set the stringstream buffer to "en garde!"

```

有两种类似的获取 stringstream 数据的方式：

1. 使用 `str()` 函数来获取缓存的数据：

```
        std::stringstream os;
        os << "12345 67.89\n";
        std::cout << os.str();

```

打印：

```
        12345 67.89

```

2. 使用提取（»）操作符：

```
        std::stringstream os;
        os << "12345 67.89"; // insert a string of numbers into the stream

        std::string strValue;
        os >> strValue;

        std::string strValue2;
        os >> strValue2;

        // print the numbers separated by a dash
        std::cout << strValue << " - " << strValue2 << '\n';

```

打印：

```
        12345 - 67.89

```

字符串与数字间的转换

由于插入与提取操作符明白如何与所有基础数据类型工作，我们可以使用它们来转换字符成为数字，反之亦然。
首先看一下数值转字符串：

```
std::stringstream os;

int nValue{ 12345 };
double dValue{ 67.89 };
os << nValue << ' ' << dValue;

std::string strValue1, strValue2;
os >> strValue1 >> strValue2;

std::cout << strValue1 << ' ' << strValue2 << '\n';
```

打印:

```
12345 67.89
```

再是数值字符串转数值:

```
std::stringstream os;
os << "12345 67.89"; // insert a string of numbers into the stream
int nValue;
double dValue;

os >> nValue >> dValue;

std::cout << nValue << ' ' << dValue << '\n';
```

打印:

```
12345 67.89
```

清空一个 stringstream 用作重用

有若干种办法可以清空一个 stringstream 缓存:

1. 使用 `str()` 与空的 C-style 字符串，设置其为空字符串:

```
    std::stringstream os;
    os << "Hello ";

    os.str(""); // erase the buffer

    os << "World!";
    std::cout << os.str();

```

2. 使用 `str()` 与空的 `std::string` 对象，设置其为空字符串:

```
    std::stringstream os;
    os << "Hello ";

```

```
    os.str(std::string{}); // erase the buffer

    os << "World!";
    std::cout << os.str();

```

它们都会打印以下结果：

```
World!
```

当清理一个 stringstream 时，调用 `clear()` 函数通常是一个好主意：

```
std::stringstream os;
os << "Hello ";

os.str(""); // erase the buffer
os.clear(); // reset error flags

os << "World!";
std::cout << os.str();
```

23.5 Stream states and input validation

流状态

ios_base 类包含若干状态 flags 用于标记不同情况下可能的状态：

Flag	Meaning
goodbit	Everything is okay
badbit	Some kind of fatal error occurred (e.g. the program tried to read past the end of a file)
eofbit	The stream has reached the end of a file
failbit	A non-fatal error occurred (e.g. the user entered letters when the program was expecting an integer)

尽管这些 flags 存在于 ios_base 中，由于 ios 是从 ios_base 派生而来的，且 ios 相比于 ios_base 只需少量打字，因此我们通常通过 ios 使用它们（例如 `std::ios::failbit` ）。
ios 同样提供了一系列成员函数用于方便的使用这些状态：

Member	function Meaning
good()	Returns true if the goodbit is set (the stream is ok)
bad()	Returns true if the badbit is set (a fatal error occurred)
eof()	Returns true if the eofbit is set (the stream is at the end of a file)
fail()	Returns true if the failbit is set (a non-fatal error occurred)
clear()	Clears all flags and restores the stream to the goodbit state
clear(state)	Clears all flags and sets the state flag passed in
rdstate()	Returns the currently set flags
setstate(state)	Sets the state flag passed in

处理 bit 最常用的是 failbit，当用户的输入无效时它会被设置。例如，考虑以下程序：

```
std::cout << "Enter your age: ";
int age {};
std::cin >> age;
```

注意该程序预期用户输入一个整数，然而如果用户输入非数值数据，例如“Alex”，cin 将无法提取任何东西到 age，此时 failbit 将被设置。
如果错误出现且一个流被设置了除了 goodbit 以外的任何标记，之后对该流进行的所有流操作将被忽略。这个状态可以通过调用 `clear()` 函数来进行清除。

输入校验

输入校验 input validation 是检查用户输入是否符合某些条件的一个过程。输入校验通常可以拆分为两种类型：字符串与数值。
通过字符串校验，我们接受所有用户输入为字符串，然后根据其是否被正确的格式化来接受或拒绝该字符串。例如，如果要求用户输入一个电话号码，我们希望确保输入的数据有十位。在大多数语言中（特别是脚本语言例如 Perl 以及 PHP），这是通过正则表达来完成的。C++ 标准库同样也拥有一个正则表达库。因为正则表达式相比于手动字符串校验而言要慢很多，它们通常仅在不考虑性能（编译时间与运行时间）或者手动校验太笨重时使用。
通过数值校验，我们通常将用户输入的数值确保在特定的范围内（例如在 0 到 20 之间）。然而不同于字符串校验，用户输入的完全不是数字是有可能的 – 我们还需要处理这种情况。
C++ 提供了一系列函数供用户决定特定字符是数值还是字符。下面这些函数存在于 `cctype` 头文件中：

Function	Meaning
std::isalnum(int)	Returns non-zero if the parameter is a letter or a digit
std::isalpha(int)	Returns non-zero if the parameter is a letter
std::iscntrl(int)	Returns non-zero if the parameter is a control character
std::isdigit(int)	Returns non-zero if the parameter is a digit
std::isgraph(int)	Returns non-zero if the parameter is printable character that is not whitespace
std::isprint(int)	Returns non-zero if the parameter is printable character (including whitespace)
std::ispunct(int)	Returns non-zero if the parameter is neither alphanumeric nor whitespace
std::isspace(int)	Returns non-zero if the parameter is whitespace
std::isxdigit(int)	Returns non-zero if the parameter is a hexadecimal digit (0-9, a-f, A-F)

字符串校验

现在让我们做一个简单的要求用户输入字符串的校验案例。校验的标准是用户仅输入字母字符或空格。如果任何其他类型的输入出现，输入则被拒绝。
当遇到变量长度输入时，最好校验字符串的方式（在不使用一个正则表达式的库的情况下）是遍历每个字符并确保它符合校验标准。以下案例便是这么做的，或者更确切的说是 `std::all_of` 做的。

```
#include <algorithm> // std::all_of
#include <cctype> // std::isalpha, std::isspace
#include <iostream>
#include <ranges>
```

```cpp
#include <string>
#include <string_view>

bool isValidName(std::string_view name)
{
  return std::ranges::all_of(name, [](char ch) {
    return (std::isalpha(ch) || std::isspace(ch));
  });

  // Before C++20, without ranges
  // return std::all_of(name.begin(), name.end(), [](char ch) {
  //    return (std::isalpha(ch) || std::isspace(ch));
  // });
}

int main()
{
  std::string name{};

  do
  {
    std::cout << "Enter your name: ";
    std::getline(std::cin, name); // get the entire line, including spaces
  } while (!isValidName(name));

  std::cout << "Hello " << name << "!\n";
}
```

注意这段代码并不完美：用户可能会输入“asf w jweo s di we ao”或者其它垃圾信息，或者更坏的是一堆空格。我们可以强化校验标准来避免上述问题。

现在让我们看一下另一个例子，用户输入电话号码。不同于用户名，对于每个字符的长度与校验标准都是一致的，电话号码是固定长度的，但是校验标准根据字符所处的位置不同。因此我们需要不同的方法来处理电话号码的输入。编写的函数将会检查用户的输入是否与预定义的模板匹配。该模板的工作原理如下：

- # 将匹配任意用户输入的数值
- @ 将匹配任何用户输入的字母
- _ 将匹配任何空值
- ? 将匹配任意值

除此之外，用户的输入必须完全与模板匹配。

那么如果询问函数匹配模板 `(######)######-########` ，意味着预期用户输入一个 (字符，三个数字，一个) 字符，一个空格，三个数字，一杠，以及四个数字。如果出现任何不匹配，输入将被拒绝。

```cpp
#include <algorithm> // std::equal
#include <cctype> // std::isdigit, std::isspace, std::isalpha
#include <iostream>
#include <map>
#include <ranges>
#include <string>
#include <string_view>

bool inputMatches(std::string_view input, std::string_view pattern)
{
    if (input.length() != pattern.length())
    {
        return false;
    }

    // This table defines all special symbols that can match a range of user input
    // Each symbol is mapped to a function that determines whether the input is valid for that symbol
    static const std::map<char, int (*)(int)> validators{
      { '#', &std::isdigit },
      { '_', &std::isspace },
      { '@', &std::isalpha },
      { '?', [](int) { return 1; } }
    };

    // Before C++20, use
    // return std::equal(input.begin(), input.end(), pattern.begin(), [](char ch, char mask) -> bool {
    // ...

    return std::ranges::equal(input, pattern, [](char ch, char mask) -> bool {
        if (auto found{ validators.find(mask) }; found != validators.end())
        {
            // The pattern's current element was found in the validators. Call the
            // corresponding function.
            return (*found->second)(ch);
        }
        else
        {
            // The pattern's current element was not found in the validators. The
            // characters have to be an exact match.
            return (ch == mask);
        }
        });
}

int main()
{
    std::string phoneNumber{};
```

```
48
49      do
50      {
51          std::cout << "Enter a phone number (###) ###-####: ";
52          std::getline(std::cin, phoneNumber);
53      } while (!inputMatches(phoneNumber, "(###) ###-####"));
54
55      std::cout << "You entered: " << phoneNumber << '\n';
56  }
```

使用该函数可以强制用户匹配特殊的格式。然而，该函数仍然若干约束：如果 #，@，_，以及?是合法的用户字符输入，该函数不能正常工作，因为这些符号被赋予了特殊的含义。另外，有别于正则表达式，缺少一个模板符号用作于表示“一个数值字符可以被输入”。因此，这样一个模板不能被用于保证用户输入两个单词间添加一个空格，因为它不能处理单词变量的长度。正因上述问题，非模板处理通常更为合适。

数值校验

```
1   #include <iostream>
2   #include <limits>
3
4   int main()
5   {
6       int age{};
7
8       while (true)
9       {
10          std::cout << "Enter your age: ";
11          std::cin >> age;
12
13          if (std::cin.fail()) // no extraction took place
14          {
15              std::cin.clear(); // reset the state bits back to goodbit so we can use ignore()
16              std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // clear out
        the bad input from the stream
17              continue; // try again
18          }
19
20          if (age <= 0) // make sure age is positive
21              continue;
22
23          break;
24      }
25
26      std::cout << "You entered: " << age << '\n';
27  }
```

如果不想这样的输入有效，我们需要做一些额外的工作。幸运的是，之前的例子已经给出了一半的解决方案。我们可以使用 `gcount()` 函数来决定多少字符被忽略。如果输出是有效的，`gcount()` 应该返回 1（新行字符已被弃用了）。如果返回的数大于 1，那么用户的输入并不正确，我们需要问他们要新的输入。以下是例子：

```
#include <iostream>
#include <limits>

int main()
{
    int age{};

    while (true)
    {
        std::cout << "Enter your age: ";
        std::cin >> age;

        if (std::cin.fail()) // no extraction took place
        {
            std::cin.clear(); // reset the state bits back to goodbit so we can use ignore()
            std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // clear out the bad input from the stream
            continue; // try again
        }

        std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // clear out any additional input from the stream
        if (std::cin.gcount() > 1) // if we cleared out more than one additional character
        {
            continue; // we'll consider this input to be invalid
        }

        if (age <= 0) // make sure age is positive
        {
            continue;
        }

        break;
    }

    std::cout << "You entered: " << age << '\n';
}
```

数值校验作为一个字符串

上述相当大的例子仅作用于简单的工作。另一种处理数值输入的方式是将它们读取成一个字符串。以下例子就是使用这样的方法论：

```cpp
#include <charconv> // std::from_chars
#include <iostream>
#include <optional>
#include <string>
#include <string_view>

std::optional<int> extractAge(std::string_view age)
{
  int result{};
  auto end{ age.data() + age.length() };

  // Try to parse an int from age
  if (std::from_chars(age.data(), end, result).ptr != end)
  {
    return {};
  }

  if (result <= 0) // make sure age is positive
  {
    return {};
  }

  return result;
}

int main()
{
  int age{};

  while (true)
  {
    std::cout << "Enter your age: ";
    std::string strAge{};
    std::getline(std::cin >> std::ws, strAge);

    if (auto extracted{ extractAge(strAge) })
    {
      age = *extracted;
      break;
    }
  }

  std::cout << "You entered: " << age << '\n';
}
```

这种处理方式的工作量大小取决于验证参数和限制。

如此可见，C++ 中的输入验证需要做大量的工作。幸运的是，很多这样的任务（例如作为一个字符串的数值校验）可以简单的转换成函数，即可以在不同的情况下进行复用。

24 Miscellaneous Subjects