



C++

Notes for Professionals

<h2>Chapter 12: File I/O</h2> <p>C++ file I/O is done via streams. The key abstractions are:</p> <ul style="list-style-type: none">std::istream for reading text.std::ostream for writing text.std::iostream for reading or writing characters.Formatted input uses operator >>.Formatted output uses operator <<. <p>Streams use std::locale, e.g., for details of the formatting and for translation between e.g. internal encoding.</p> <p>More on streams: <iostream> library</p>	<h2>Section 12.1: Writing to a file</h2> <p>There are several ways to write to a file. The easiest way is to use an output file stream stream insertion operator (<<).</p> <pre>std::ofstream out("foo.txt"); out << "Hello World!"</pre> <p>Instead of <<, you can also use the output file stream's member function write().</p> <pre>std::ofstream out("foo.txt"); out.open(); char data[3]; // Write 3 characters from data to >> "foo". out.write(data, 3);</pre> <p>After writing to a stream, you should always check if error state flag <code>bfail</code> is set or not. This can be done by calling the output file stream's member function <code>fail()</code>.</p> <pre>if (out.fail()) // Failed to write!</pre> <h2>Section 12.2: Opening a file</h2> <p>Opening a file is done in the same way for all 3 file streams (ifstream, ofstream, ifstream).</p> <p>You can open the file directly in the constructor:</p> <pre>std::ifstream ifs("foo.txt"); // ifstream: Open file "foo.txt" std::ofstream ofs("foo.txt"); // ofstream: Open file "foo.txt"</pre> <p>C++ Notes for Professionals</p>
---	--

Chapter 48: std::array

Parameter
class T **Definition**
std::size_t N **Specifies the data type of array members**
std::size_t n **Specifies the number of members in the array**

Section 48.1: Initializing an std::array

If T is a scalar type, std::array can be initialized in the following ways:

- // 1) Using aggregate initialization:
`std::array<T, N> a{ 0, 1, 2, 3 };`
 or equivalently
`std::array<T, n> a{ 0, 1, 2, 3 };`
- // 2) Using the copy constructor:
`std::array<T, N> a({ 0, 1, 2, 3 });`
 or equivalently
`std::array<T, n> a({ 0, 1, 2, 3 });`
- // 3) Using the move constructor:
`std::array<T, N> a = std::move({ 0, 1, 2, 3 });`

If T is a non-scalar type std::array can be initialized in the following ways:

Allocate a `T` int values[3]; // An aggregate type

- // It works only if `T` is an aggregate type:
`std::array<T, N> a{ 0, 1, 2, 3, 4, 5 };`
 or equivalently
`std::array<T, n> a{ { 0, 1, 2, 3, 4, 5 } };`
- // 2) Using aggregate initialization with brace initialization of sub-elements:
`std::array<T, N> a{ { A(0, 1, 2), A(3, 4, 5) } };`
 or equivalently
`std::array<T, n> a{ { A(0, 1, 2), A(3, 4, 5) } };`
- // 3)
`std::array<T, N> a{ { 0, 1, 2, 3, 4, 5 } };`
 or equivalently
`std::array<T, n> a{ { 0, 1, 2, 3, 4, 5 } };`
- // 4) Using the copy constructor:
`std::array<T, N> a({ 0, 1, 2, 3, 4, 5 });`
 or equivalently
`std::array<T, n> a({ 0, 1, 2, 3, 4, 5 });`
- // 5) Using the move constructor:
`std::array<T, N> a = std::move({ 0, 1, 2, 3, 4, 5 });`

C++ Notes for Professionals

Chapter 12: File I/O

C++ file I/O is done via streams. The key abstractions are:

- `std::istream` for reading text.
- `std::ostream` for writing text.
- `std::iostream` for reading or writing characters.
- Formatted input uses operator `<<`.
- Formatted output uses operator `>>`.

Streams use `std::locale`, e.g., for details of the formatting and for translation between internal encoding.

More on streams: <streams> Library

Section 12.1: Writing to a file

There are several ways to write to a file. The easiest way is to use an output stream, insertion operator (`<<`):

```
std::ofstream out("foo.txt");
out << "Hello World!";
```

Instead of `<<`, you can also use the output file stream's member function `write()`:

```
std::ofstream out("foo.txt");
out.write((char*)"Hello", 5);
// Writes 3 characters from data to "foo".
out.write(data, 3);
```

After writing to a stream, you should always check if error state flag `badbit` is operation failed or not. This can be done by calling the output file stream's `fail()` operation or `!ok()`. It can be done by calling the output file stream's `fail()` operation or `!ok()`.

Section 12.2: Opening a file

Opening a file is done in the same way for all 3 file streams (`ifstream`, `ofstream`, `fstream`). You can open the file directly in the constructor:

```
ifstream ifs("foo.txt"); // ifstream: Opens file "foo.txt" for reading only.
ofstream ofs("foo.txt"); // ofstream: Opens file "foo.txt" for writing only.
```

C++ Notes for Professionals

<p>vs the delimiter to be</p> <p>some implementations do</p> <p>concurrently (this may not be thread-safe) not be used on const at-risk or to operate on a to be copied, then the copy</p> <p>ers, but if the cheapest a hand-spun solution.</p> <p>The input std::string is etting its difficulties, also and by adding support</p> <p>(xx), (1):</p> <p>vides for a more</p>	<h1>Chapter 48: std::array</h1> <p>Parameter Definition</p> <p>class T Specifies the data type of array members</p> <p>std::size_t n Specifies the number of members in the array</p> <h2>Section 48.1: Initializing an std::array</h2> <p>Initializing std::array<T, N>, where T is a scalar type</p> <p>If T is a scalar type, std::array can be initialized in the following ways:</p> <ul style="list-style-type: none"> // 1) Using aggregate initialization: <code>std::array<T, N> arr{0, 1, 2, 3};</code> or equivalently: <code>std::array<T, N> arr = {0, 1, 2, 3};</code> // 2) Using the copy constructor: <code>std::array<T, N> arr{std::array{0, 1, 2, 3}};</code> or equivalently: <code>std::array<T, N> arr = std::array{0, 1, 2, 3};</code> // 3) Using the move constructor: <code>std::array<T, N> arr{std::array{0, 1, 2, 3}};</code> <p>Initializing std::array<T, N>, where T is a non-scalar type</p> <p>If T is a non-scalar type std::array can be initialized in the following ways:</p> <ul style="list-style-type: none"> struct A { int values[3]; }; // An aggregate type // 1) Using aggregate initialization with brace elision: It works only if T is an aggregate type! <code>std::array<A, 2> arr{{0, 1, 2}, {3, 4, 5}};</code> or equivalently: <code>std::array<A, 2> arr = {{0, 1, 2}, {3, 4, 5}};</code> // 2) Using aggregate initialization with brace initialization of sub-elements: <code>std::array<A, 2> arr{{A{0, 1, 2}, A{3, 4, 5}}};</code> or equivalently: <code>std::array<A, 2> arr = {{A{0, 1, 2}, A{3, 4, 5}}};</code> // 3) Using aggregate initialization with brace initialization of sub-elements: <code>std::array<A, 2> arr{{0, 1, 2, 3, 4, 5}};</code> or equivalently: <code>std::array<A, 2> arr = {{0, 1, 2, 3, 4, 5}};</code> // 4) Using the copy constructor: <code>std::array<A, 2> arr{std::array{0, 1, 2, 3}};</code> or equivalently: <code>std::array<A, 2> arr = std::array{0, 1, 2, 3};</code> // 5) Using the move constructor: <code>std::array<A, 2> arr{std::array{0, 1, 2, 3, 4, 5}};</code>
--	--

600多页 专业提示和技巧

600+ pages
of professional hints and tricks

目录

关于	1
第1章：C++入门	2
第1.1节：你好，世界	2
第1.2节：注释	3
第1.3节：标准C++编译过程	5
第1.4节：函数	5
第1.5节：函数原型和声明的可见性	8
第1.6节：预处理器	9
第2章：字面量	11
第2.1节：this	11
第2.2节：整数字面量	11
第2.3节：true	12
第2.4节：false	13
第2.5节：nullptr	13
第3章：运算符优先级	14
第3.1节：逻辑与 (&&) 和逻辑或 () 运算符：短路	14
第3.2节：一元运算符	15
第3.3节：算术运算符	15
第3.4节：逻辑与或或运算符	16
第4章：浮点运算	17
第4.1节：浮点数很奇怪	17
第5章：位运算符	18
第5.1节： - 按位或	18
第5.2节：^ - 按位异或（排他或）	18
第5.3节：& - 按位与	20
第5.4节：<< - 左移	20
第5.5节：>> - 右移	21
第6章：位操作	23
第6.1节：移除最右边的置位	23
第6.2节：设置所有位	23
第6.3节：切换位	23
第6.4节：检查位	23
第6.5节：计算已设置的位数	24
第6.6节：检查整数是否为2的幂	25
第6.7节：设置位	25
第6.8节：清除位	25
第6.9节：将第n位更改为x	25
第6.10节：位操作应用：小写字母转大写字母	26
第7章：位域	27
第7.1节：声明与使用	27
第8章：数组	28
第8.1节：数组初始化	28
第8.2节：固定大小的原始数组矩阵（即二维原始数组）	29
第8.3节：动态大小的原始数组	29
第8.4节：数组大小：编译时类型安全	30
第8.5节：使用std::vector扩展动态大小数组	31

Contents

About	1
Chapter 1: Getting started with C++	2
Section 1.1: Hello World	2
Section 1.2: Comments	3
Section 1.3: The standard C++ compilation process	5
Section 1.4: Function	5
Section 1.5: Visibility of function prototypes and declarations	8
Section 1.6: Preprocessor	9
Chapter 2: Literals	11
Section 2.1: this	11
Section 2.2: Integer literal	11
Section 2.3: true	12
Section 2.4: false	13
Section 2.5: nullptr	13
Chapter 3: operator precedence	14
Section 3.1: Logical && and operators: short-circuit	14
Section 3.2: Unary Operators	15
Section 3.3: Arithmetic operators	15
Section 3.4: Logical AND and OR operators	16
Chapter 4: Floating Point Arithmetic	17
Section 4.1: Floating Point Numbers are Weird	17
Chapter 5: Bit Operators	18
Section 5.1: - bitwise OR	18
Section 5.2: ^ - bitwise XOR (exclusive OR)	18
Section 5.3: & - bitwise AND	20
Section 5.4: << - left shift	20
Section 5.5: >> - right shift	21
Chapter 6: Bit Manipulation	23
Section 6.1: Remove rightmost set bit	23
Section 6.2: Set all bits	23
Section 6.3: Toggling a bit	23
Section 6.4: Checking a bit	23
Section 6.5: Counting bits set	24
Section 6.6: Check if an integer is a power of 2	25
Section 6.7: Setting a bit	25
Section 6.8: Clearing a bit	25
Section 6.9: Changing the nth bit to x	25
Section 6.10: Bit Manipulation Application: Small to Capital Letter	26
Chapter 7: Bit fields	27
Section 7.1: Declaration and Usage	27
Chapter 8: Arrays	28
Section 8.1: Array initialization	28
Section 8.2: A fixed size raw array matrix (that is, a 2D raw array)	29
Section 8.3: Dynamically sized raw array	29
Section 8.4: Array size: type safe at compile time	30
Section 8.5: Expanding dynamic size array by using std::vector	31

第8.6节：使用std::vector存储的动态大小矩阵	32
第9章：迭代器	35
第9.1节：概述	35
第9.2节：向量迭代器	38
第9.3节：映射迭代器	38
第9.4节：反向迭代器	39
第9.5节：流迭代器	40
第9.6节：C语言迭代器（指针）	40
第9.7节：编写你自己的基于生成器的迭代器	41
第10章：C++中的基本输入/输出	43
第10.1节：用户输入和标准输出	43
第11章：循环	44
第11.1节：基于范围的For循环	44
第11.2节：For循环	46
第11.3节：While循环	48
第11.4节：Do-while循环	49
第11.5节：循环控制语句：Break和Continue	50
第11.6节：条件中的变量声明	51
第11.7节：子范围上的范围for循环	52
第12章：文件输入输出	54
第12.1节：写入文件	54
第12.2节：打开文件	54
第12.3节：从文件读取	55
第12.4节：打开模式	57
第12.5节：将ASCII文件读取到std::string中	58
第12.6节：使用非标准区域设置写入文件	59
第12.7节：在循环条件中检查文件结尾，是否是不良做法？	60
第12.8节：刷新流	61
第12.9节：将文件读入容器	61
第12.10节：复制文件	62
第12.11节：关闭文件	62
第12.12节：从格式化文本文件读取`struct`	63
第13章：C++流	65
第13.1节：字符串流	65
第13.2节：使用iostream打印集合	66
第14章：流操纵器	68
第14.1节：流操作器	68
第14.2节：输出流操作器	73
第14.3节：输入流操作器	75
第15章：流程控制	77
第15.1节：case语句	77
第15.2节：switch	77
第15.3节：catch	77
第15.4节：throw	78
第15.5节：default	79
第15.6节：try	79
第15.7节：if	79
第15.8节：else	80
第15.9节：条件结构：if, if..else	80

Section 8.6: A dynamic size matrix using std::vector for storage	32
Chapter 9: Iterators	35
Section 9.1: Overview	35
Section 9.2: Vector Iterator	38
Section 9.3: Map Iterator	38
Section 9.4: Reverse Iterators	39
Section 9.5: Stream Iterators	40
Section 9.6: C Iterators (Pointers)	40
Section 9.7: Write your own generator-backed iterator	41
Chapter 10: Basic input/output in c++	43
Section 10.1: user input and standard output	43
Chapter 11: Loops	44
Section 11.1: Range-Based For	44
Section 11.2: For loop	46
Section 11.3: While loop	48
Section 11.4: Do-while loop	49
Section 11.5: Loop Control statements : Break and Continue	50
Section 11.6: Declaration of variables in conditions	51
Section 11.7: Range-for over a sub-range	52
Chapter 12: File I/O	54
Section 12.1: Writing to a file	54
Section 12.2: Opening a file	54
Section 12.3: Reading from a file	55
Section 12.4: Opening modes	57
Section 12.5: Reading an ASCII file into a std::string	58
Section 12.6: Writing files with non-standard locale settings	59
Section 12.7: Checking end of file inside a loop condition, bad practice?	60
Section 12.8: Flushing a stream	61
Section 12.9: Reading a file into a container	61
Section 12.10: Copying a file	62
Section 12.11: Closing a file	62
Section 12.12: Reading a `struct` from a formatted text file	63
Chapter 13: C++ Streams	65
Section 13.1: String streams	65
Section 13.2: Printing collections with iostream	66
Chapter 14: Stream manipulators	68
Section 14.1: Stream manipulators	68
Section 14.2: Output stream manipulators	73
Section 14.3: Input stream manipulators	75
Chapter 15: Flow Control	77
Section 15.1: case	77
Section 15.2: switch	77
Section 15.3: catch	77
Section 15.4: throw	78
Section 15.5: default	79
Section 15.6: try	79
Section 15.7: if	79
Section 15.8: else	80
Section 15.9: Conditional Structures: if, if..else	80

第15.10节 : goto	81	Section 15.10: goto	81
第15.11节 : 跳转语句 : break, continue, goto, exit	81	Section 15.11: Jump statements : break, continue, goto, exit	81
第15.12节 : return	84	Section 15.12: return	84
第16章 : 元编程	86	Chapter 16: Metaprogramming	86
第16.1节 : 计算阶乘	86	Section 16.1: Calculating Factorials	86
第16.2节 : 遍历参数包	88	Section 16.2: Iterating over a parameter pack	88
第16.3节 : 使用std::integer_sequence遍历	89	Section 16.3: Iterating with std::integer_sequence	89
第16.4节 : 标签分发	90	Section 16.4: Tag Dispatching	90
第16.5节 : 检测表达式是否有效	90	Section 16.5: Detect Whether Expression is Valid	90
第16.6节 : 条件语句 (if-then-else)	92	Section 16.6: If-then-else	92
第16.7节 : 给定任意类型T时的手动类型区分	92	Section 16.7: Manual distinction of types when given any type T	92
第16.8节 : 使用C++11 (及更高版本) 计算幂	93	Section 16.8: Calculating power with C++11 (and higher)	93
第16.9节 : 支持可变参数数量的通用最小值/最大值	94	Section 16.9: Generic Min/Max with variable argument count	94
第17章 : const关键字	95	Chapter 17: const keyword	95
第17.1节 : 避免在const和非const的getter方法中重复代码	95	Section 17.1: Avoiding duplication of code in const and non-const getter methods	95
第17.2节 : const成员函数	96	Section 17.2: Const member functions	96
第17.3节 : const局部变量	97	Section 17.3: Const local variables	97
第17.4节 : const指针	97	Section 17.4: Const pointers	97
第18章 : mutable关键字	99	Chapter 18: mutable keyword	99
第18.1节 : 可变lambda表达式	99	Section 18.1: mutable lambdas	99
第18.2节 : 非静态类成员修饰符	99	Section 18.2: non-static class member modifier	99
第19章 : 友元关键字	101	Chapter 19: Friend keyword	101
第19.1节 : 友元函数	101	Section 19.1: Friend function	101
第19.2节 : 友元方法	102	Section 19.2: Friend method	102
第19.3节 : 友元类	102	Section 19.3: Friend class	102
第20章 : 类型关键字	104	Chapter 20: Type Keywords	104
第20.1节 : 类	104	Section 20.1: class	104
第20.2节 : 枚举	105	Section 20.2: enum	105
第20.3节 : 结构体	106	Section 20.3: struct	106
第20.4节 : 联合体	106	Section 20.4: union	106
第21章 : 基本类型关键字	108	Chapter 21: Basic Type Keywords	108
第21.1节 : char	108	Section 21.1: char	108
第21.2节 : char16_t	108	Section 21.2: char16_t	108
第21.3节 : char32_t	108	Section 21.3: char32_t	108
第21.4节 : int	108	Section 21.4: int	108
第21.5节 : void	108	Section 21.5: void	108
第21.6节 : wchar_t	109	Section 21.6: wchar_t	109
第21.7节 : float	109	Section 21.7: float	109
第21.8节 : double	109	Section 21.8: double	109
第21.9节 : long	109	Section 21.9: long	109
第21.10节 : 简短	110	Section 21.10: short	110
第21.11节 : bool	110	Section 21.11: bool	110
第22章 : 变量声明关键字	111	Chapter 22: Variable Declaration Keywords	111
第22.1节 : decltype	111	Section 22.1: decltype	111
第22.2节 : const	111	Section 22.2: const	111
第22.3节 : volatile	112	Section 22.3: volatile	112
第22.4节 : signed	112	Section 22.4: signed	112
第22.5节 : unsigned	112	Section 22.5: unsigned	112
第23章 : 关键字	114	Chapter 23: Keywords	114

第23.1节 : asm	114
第23.2节 : 不同的关键字	114
第23.3节 : typename	118
第23.4节 : explicit	119
第23.5节 : sizeof	119
第23.6节 : noexcept	120
第24章 : 从函数返回多个值	122
第24.1节 : 使用 std::tuple	122
第24.2节 : 结构化绑定	123
第24.3节 : 使用结构体	124
第24.4节 : 使用输出参数	125
第24.5节 : 使用函数对象消费者	126
第24.6节 : 使用std::pair	127
第24.7节 : 使用std::array	127
第24.8节 : 使用输出迭代器	127
第24.9节 : 使用std::vector	128
第25章 : 多态	129
第25.1节 : 定义多态类	129
第25.2节 : 安全向下转换	130
第25.3节 : 多态与析构函数	131
第26章 : 引用	133
第26.1节 : 定义引用	133
第27章 : 值语义与引用语义	134
第27.1节 : 定义	134
第27.2节 : 深拷贝与移动支持	134
第28章 : C++函数“值传递”与“引用传递”	138
第28.1节 : 值传递	138
第29章 : 拷贝与赋值	140
第29.1节 : 赋值运算符	140
第29.2节 : 拷贝构造函数	140
第29.3节 : 拷贝构造函数与赋值构造函数的比较	141
第30章 : 指针	143
第30.1节 : 指针操作	143
第30.2节 : 指针基础	143
第30.3节 : 指针运算	145
第31章 : 指向成员的指针	147
第31.1节 : 指向静态成员函数的指针	147
第31.2节 : 指向成员函数的指针	147
第31.3节 : 指向成员变量的指针	148
第31.4节 : 指向静态成员变量的指针	148
第32章 : this指针	150
第32.1节 : this指针	150
第32.2节 : 使用this指针访问成员数据	152
第32.3节 : 使用this指针区分成员数据和参数	152
第32.4节 : this指针的CV限定符	153
第32.5节 : this指针的引用限定符	156
第33章 : 智能指针	158
第33.1节 : 唯一所有权 (std::unique_ptr)	158
第33.2节 : 共享所有权 (std::shared_ptr)	159

Section 23.1: asm	114
Section 23.2: Different keywords	114
Section 23.3: typename	118
Section 23.4: explicit	119
Section 23.5: sizeof	119
Section 23.6: noexcept	120
Chapter 24: Returning several values from a function	122
Section 24.1: Using std::tuple	122
Section 24.2: Structured Bindings	123
Section 24.3: Using struct	124
Section 24.4: Using Output Parameters	125
Section 24.5: Using a Function Object Consumer	126
Section 24.6: Using std::pair	127
Section 24.7: Using std::array	127
Section 24.8: Using Output Iterator	127
Section 24.9: Using std::vector	128
Chapter 25: Polymorphism	129
Section 25.1: Define polymorphic classes	129
Section 25.2: Safe downcasting	130
Section 25.3: Polymorphism & Destructors	131
Chapter 26: References	133
Section 26.1: Defining a reference	133
Chapter 27: Value and Reference Semantics	134
Section 27.1: Definitions	134
Section 27.2: Deep copying and move support	134
Chapter 28: C++ function "call by value" vs. "call by reference"	138
Section 28.1: Call by value	138
Chapter 29: Copying vs Assignment	140
Section 29.1: Assignment Operator	140
Section 29.2: Copy Constructor	140
Section 29.3: Copy Constructor Vs Assignment Constructor	141
Chapter 30: Pointers	143
Section 30.1: Pointer Operations	143
Section 30.2: Pointer basics	143
Section 30.3: Pointer Arithmetic	145
Chapter 31: Pointers to members	147
Section 31.1: Pointers to static member functions	147
Section 31.2: Pointers to member functions	147
Section 31.3: Pointers to member variables	148
Section 31.4: Pointers to static member variables	148
Chapter 32: The This Pointer	150
Section 32.1: this Pointer	150
Section 32.2: Using the this Pointer to Access Member Data	152
Section 32.3: Using the this Pointer to Differentiate Between Member Data and Parameters	152
Section 32.4: this Pointer CV-Qualifiers	153
Section 32.5: this Pointer Ref-Qualifiers	156
Chapter 33: Smart Pointers	158
Section 33.1: Unique ownership (std::unique_ptr)	158
Section 33.2: Sharing ownership (std::shared_ptr)	159

第33.3节：带临时所有权的共享 (std::weak_ptr)	161
第33.4节：使用自定义删除器创建C接口的包装器	163
第33.5节：无移动语义的唯一所有权 (auto_ptr)	164
第33.6节：转换std::shared_ptr指针	166
第33.7节：编写智能指针：value_ptr	166
第33.8节：获取指向本对象的shared_ptr	168
第34章：类/结构体	170
第34.1节：类基础	170
第34.2节：最终类和结构体	170
第34.3节：访问说明符	171
第34.4节：继承	172
第34.5节：友元关系	174
第34.6节：虚继承	175
第34.7节：私有继承：限制基类接口	176
第34.8节：访问类成员	177
第34.9节：成员类型和别名	178
第34.10节：嵌套类/结构体	182
第34.11节：无名结构体/类	186
第34.12节：静态类成员	187
第34.13节：多重继承	191
第34.14节：非静态成员函数	192
第35章：函数重载	195
第35.1节：什么是函数重载？	195
第35.2节：函数重载中的返回类型	196
第35.3节：成员函数的cv限定符重载	196
第36章：运算符重载	199
第36.1节：算术运算符	199
第36.2节：数组下标运算符	200
第36.3节：转换运算符	201
第36.4节：复数重访	202
第36.5节：命名运算符	206
第36.6节：一元运算符	208
第36.7节：比较运算符	209
第36.8节：赋值运算符	210
第36.9节：函数调用运算符	211
第36.10节：按位非运算符	211
第36.11节：输入输出的位移运算符	212
第37章：函数模板重载	213
第37.1节：什么是有效的函数模板重载？	213
第38章：虚成员函数	214
第38.1节：最终虚函数	214
第38.2节：在C++11及以后版本中使用override和virtual	214
第38.3节：虚函数与非虚函数成员	215
第38.4节：构造函数和析构函数中虚函数的行为	216
第38.5节：纯虚函数	217
第39章：内联函数	220
第39.1节：非成员内联函数定义	220
第39.2节：成员内联函数	220
第39.3节：什么是函数内联？	220
第39.4节：非成员内联函数声明	221

Section 33.3: Sharing with temporary ownership (std::weak_ptr)	161
Section 33.4: Using custom deleters to create a wrapper to a C interface	163
Section 33.5: Unique ownership without move semantics (auto_ptr)	164
Section 33.6: Casting std::shared_ptr pointers	166
Section 33.7: Writing a smart pointer: value_ptr	166
Section 33.8: Getting a shared_ptr referring to this	168
Chapter 34: Classes/Structures	170
Section 34.1: Class basics	170
Section 34.2: Final classes and structs	170
Section 34.3: Access specifiers	171
Section 34.4: Inheritance	172
Section 34.5: Friendship	174
Section 34.6: Virtual Inheritance	175
Section 34.7: Private inheritance: restricting base class interface	176
Section 34.8: Accessing class members	177
Section 34.9: Member Types and Aliases	178
Section 34.10: Nested Classes/Structures	182
Section 34.11: Unnamed struct/class	186
Section 34.12: Static class members	187
Section 34.13: Multiple Inheritance	191
Section 34.14: Non-static member functions	192
Chapter 35: Function Overloading	195
Section 35.1: What is Function Overloading?	195
Section 35.2: Return Type in Function Overloading	196
Section 35.3: Member Function cv-qualifier Overloading	196
Chapter 36: Operator Overloading	199
Section 36.1: Arithmetic operators	199
Section 36.2: Array subscript operator	200
Section 36.3: Conversion operators	201
Section 36.4: Complex Numbers Revisited	202
Section 36.5: Named operators	206
Section 36.6: Unary operators	208
Section 36.7: Comparison operators	209
Section 36.8: Assignment operator	210
Section 36.9: Function call operator	211
Section 36.10: Bitwise NOT operator	211
Section 36.11: Bit shift operators for I/O	212
Chapter 37: Function Template Overloading	213
Section 37.1: What is a valid function template overloading?	213
Chapter 38: Virtual Member Functions	214
Section 38.1: Final virtual functions	214
Section 38.2: Using override with virtual in C++11 and later	214
Section 38.3: Virtual vs non-virtual member functions	215
Section 38.4: Behaviour of virtual functions in constructors and destructors	216
Section 38.5: Pure virtual functions	217
Chapter 39: Inline functions	220
Section 39.1: Non-member inline function definition	220
Section 39.2: Member inline functions	220
Section 39.3: What is function inlining?	220
Section 39.4: Non-member inline function declaration	221

第40章：特殊成员函数	222
第40.1节：默认构造函数	222
第40.2节：析构函数	224
第40.3节：拷贝与交换	225
第40.4节：隐式移动与拷贝	227
第41章：非静态成员函数	228
第41.1节：非静态成员函数	228
第41.2节：封装	229
第41.3节：名称隐藏与导入	229
第41.4节：虚成员函数	231
第41.5节：常量正确性	233
第42章：常量类成员函数	235
第42.1节：常量成员函数	235
第43章：C++容器	236
第43.1节：C++容器流程图	236
第44章：命名空间	237
第44.1节：什么是命名空间？	237
第44.2节：基于参数的查找	238
第44.3节：扩展命名空间	239
第44.4节：使用指令	239
第44.5节：创建命名空间	240
第44.6节：无名/匿名命名空间	241
第44.7节：紧凑嵌套命名空间	241
第44.8节：命名空间别名	241
第44.9节：内联命名空间	242
第44.10节：长命名空间的别名	244
第44.11节：别名声明的作用域	244
第45章：头文件	246
第45.1节：基本示例	246
第45.2节：头文件中的模板	247
第46章：使用声明	248
第46.1节：从命名空间单独导入名称	248
第46.2节：重新声明基类成员以避免名称隐藏	248
第46.3节：继承构造函数	248
第47章：std::string	250
第47.1节：分词	250
第47.2节：转换为 (const) char*	251
第47.3节：使用std::string_view类	251
第47.4节：转换为std::wstring	252
第47.5节：字典序比较	253
第47.6节：去除开头/结尾字符	254
第47.7节：字符串替换	255
第47.8节：转换为std::string	256
第47.9节：拆分	257
第47.10节：访问字符	258
第47.11节：检查字符串是否为另一个字符串的前缀	258
第47.12节：遍历每个字符	259
第47.13节：转换为整数/浮点类型	259
第47.14节：连接操作	260

Chapter 40: Special Member Functions	222
Section 40.1: Default Constructor	222
Section 40.2: Destructor	224
Section 40.3: Copy and swap	225
Section 40.4: Implicit Move and Copy	227
Chapter 41: Non-Static Member Functions	228
Section 41.1: Non-static Member Functions	228
Section 41.2: Encapsulation	229
Section 41.3: Name Hiding & Importing	229
Section 41.4: Virtual Member Functions	231
Section 41.5: Const Correctness	233
Chapter 42: Constant class member functions	235
Section 42.1: constant member function	235
Chapter 43: C++ Containers	236
Section 43.1: C++ Containers Flowchart	236
Chapter 44: Namespaces	237
Section 44.1: What are namespaces?	237
Section 44.2: Argument Dependent Lookup	238
Section 44.3: Extending namespaces	239
Section 44.4: Using directive	239
Section 44.5: Making namespaces	240
Section 44.6: Unnamed/anonymous namespaces	241
Section 44.7: Compact nested namespaces	241
Section 44.8: Namespace alias	241
Section 44.9: Inline namespace	242
Section 44.10: Aliasing a long namespace	244
Section 44.11: Alias Declaration scope	244
Chapter 45: Header Files	246
Section 45.1: Basic Example	246
Section 45.2: Templates in Header Files	247
Chapter 46: Using declaration	248
Section 46.1: Importing names individually from a namespace	248
Section 46.2: Redeclaring members from a base class to avoid name hiding	248
Section 46.3: Inheriting constructors	248
Chapter 47: std::string	250
Section 47.1: Tokenize	250
Section 47.2: Conversion to (const) char*	251
Section 47.3: Using the std::string_view class	251
Section 47.4: Conversion to std::wstring	252
Section 47.5: Lexicographical comparison	253
Section 47.6: Trimming characters at start/end	254
Section 47.7: String replacement	255
Section 47.8: Converting to std::string	256
Section 47.9: Splitting	257
Section 47.10: Accessing a character	258
Section 47.11: Checking if a string is a prefix of another	258
Section 47.12: Looping through each character	259
Section 47.13: Conversion to integers/floating point types	259
Section 47.14: Concatenation	260

第47.15节：字符编码之间的转换	261
第47.16节：在字符串中查找字符	262
第48章：std::array	263
第48.1节：初始化std::array	263
第48.2节：元素访问	264
第48.3节：遍历数组	266
第48.4节：检查数组大小	266
第48.5节：一次性更改所有数组元素	266
第49章：std::vector	267
第49.1节：访问元素	267
第49.2节：初始化std::vector	269
第49.3节：删除元素	270
第49.4节：遍历 std::vector	272
第49.5节：vector<bool>：众多规则中的例外	274
第49.6节：插入元素	275
第49.7节：将std::vector用作C数组	276
第49.8节：在std::vector中查找元素	277
第49.9节：向量的连接	278
第49.10节：使用向量的矩阵	279
第49.11节：使用排序向量进行快速元素查找	280
第49.12节：减少向量的容量	281
第49.13节：向量的大小和容量	281
第49.14节：迭代器/指针失效	283
第49.15节：查找向量中的最大和最小元素及其对应索引	284
第49.16节：将数组转换为std::vector	284
第49.17节：返回大型向量的函数	285
第50章：std::map	287
第50.1节：访问元素	287
第50.2节：插入元素	288
第50.3节：在std::map或std::multimap中搜索	289
第50.4节：初始化std::map或std::multimap	290
第50.5节：检查元素数量	291
第50.6节：地图类型	291
第50.7节：删除元素	292
第50.8节：遍历std::map或std::multimap	293
第50.9节：使用用户定义类型作为键创建std::map	293
第51章：std::optional	295
第51.1节：使用可选值表示值的缺失	295
第51.2节：可选值作为返回值	295
第51.3节：value_or	296
第51.4节：介绍	296
第51.5节：使用可选值表示函数的失败	297
第52章：std::function：封装任何可调用元素	299
第52.1节：简单用法	299
第52.2节：std::function 与 std::bind 的使用	299
第52.3节：将 std::function 绑定到不同的可调用类型	300
第52.4节：在 std::tuple 中存储函数参数	302
第52.5节：std::function 与 lambda 及 std::bind	303
第52.6节：`function` 的开销	304
第53章：std::forward_list	305

Section 47.15: Converting between character encodings	261
Section 47.16: Finding character(s) in a string	262
Chapter 48: std::array	263
Section 48.1: Initializing an std::array	263
Section 48.2: Element access	264
Section 48.3: Iterating through the Array	266
Section 48.4: Checking size of the Array	266
Section 48.5: Changing all array elements at once	266
Chapter 49: std::vector	267
Section 49.1: Accessing Elements	267
Section 49.2: Initializing a std::vector	269
Section 49.3: Deleting Elements	270
Section 49.4: Iterating Over std::vector	272
Section 49.5: vector<bool>: The Exception To So Many, So Many Rules	274
Section 49.6: Inserting Elements	275
Section 49.7: Using std::vector as a C array	276
Section 49.8: Finding an Element in std::vector	277
Section 49.9: Concatenating Vectors	278
Section 49.10: Matrices Using Vectors	279
Section 49.11: Using a Sorted Vector for Fast Element Lookup	280
Section 49.12: Reducing the Capacity of a Vector	281
Section 49.13: Vector size and capacity	281
Section 49.14: Iterator/Pointer Invalidation	283
Section 49.15: Find max and min Element and Respective Index in a Vector	284
Section 49.16: Converting an array to std::vector	284
Section 49.17: Functions Returning Large Vectors	285
Chapter 50: std::map	287
Section 50.1: Accessing elements	287
Section 50.2: Inserting elements	288
Section 50.3: Searching in std::map or in std::multimap	289
Section 50.4: Initializing a std::map or std::multimap	290
Section 50.5: Checking number of elements	291
Section 50.6: Types of Maps	291
Section 50.7: Deleting elements	292
Section 50.8: Iterating over std::map or std::multimap	293
Section 50.9: Creating std::map with user-defined types as key	293
Chapter 51: std::optional	295
Section 51.1: Using optionals to represent the absence of a value	295
Section 51.2: optional as return value	295
Section 51.3: value_or	296
Section 51.4: Introduction	296
Section 51.5: Using optionals to represent the failure of a function	297
Chapter 52: std::function: To wrap any element that is callable	299
Section 52.1: Simple usage	299
Section 52.2: std::function used with std::bind	299
Section 52.3: Binding std::function to a different callable types	300
Section 52.4: Storing function arguments in std::tuple	302
Section 52.5: std::function with lambda and std::bind	303
Section 52.6: `function` overhead	304
Chapter 53: std::forward_list	305

第53.1节：示例	305
第53.2节：方法	305
第54章：std::pair	307
第54.1节：比较运算符	307
第54.2节：创建Pair及访问元素	307
第55章：std::atomic	309
第55.1节：原子类型	309
第56章：std::variant	311
第56.1节：创建伪方法指针	311
第56.2节：std::variant的基本使用	312
第56.3节：构造`std::variant`	313
第57章：std::iomanip	314
第57.1节：std::setprecision	314
第57.2节：std::setfill	314
第57.3节：std::setiosflags	314
第57.4节：std::setw	316
第58章：std::any	317
第58.1节：基本用法	317
第59章：std::set 和 std::multiset	318
第59.1节：更改集合的默认排序	318
第59.2节：从集合中删除值	320
第59.3节：向集合中插入值	321
第59.4节：向多重集合中插入值	323
第59.5节：在集合和多重集合中搜索值	323
第60章：std::integer_sequence	325
第60.1节：将std::tuple<T...>转换为函数参数	325
第60.2节：创建由整数组成的参数包	326
第60.3节：将索引序列转换为元素的副本	326
第61章：使用std::unordered_map	328
第61.1节：声明与使用	328
第61.2节：一些基本函数	328
第62章：标准库算法	329
第62.1节：std::next_permutation	329
第62.2节：std::for_each	329
第62.3节：std::accumulate	330
第62.4节：std::find	331
第62.5节：std::min_element	333
第62.6节：std::find_if	334
第62.7节：使用 std::nth_element 查找中位数（或其他分位数）	335
第62.8节：std::count	336
第62.9节：std::count_if	337
第63章：ISO C++ 标准	339
第63.1节：当前工作草案	339
第63.2节：C++17	339
第63.3节：C++11	340
第63.4节：C++14	341
第63.5节：C++98	342
第63.6节：C++03	342
第63.7节：C++20	343

Section 53.1: Example	305
Section 53.2: Methods	305
Chapter 54: std::pair	307
Section 54.1: Compare operators	307
Section 54.2: Creating a Pair and accessing the elements	307
Chapter 55: std::atomic	309
Section 55.1: atomic types	309
Chapter 56: std::variant	311
Section 56.1: Create pseudo-method pointers	311
Section 56.2: Basic std::variant use	312
Section 56.3: Constructing a `std::variant`	313
Chapter 57: std::iomanip	314
Section 57.1: std::setprecision	314
Section 57.2: std::setfill	314
Section 57.3: std::setiosflags	314
Section 57.4: std::setw	316
Chapter 58: std::any	317
Section 58.1: Basic usage	317
Chapter 59: std::set and std::multiset	318
Section 59.1: Changing the default sort of a set	318
Section 59.2: Deleting values from a set	320
Section 59.3: Inserting values in a set	321
Section 59.4: Inserting values in a multiset	323
Section 59.5: Searching values in set and multiset	323
Chapter 60: std::integer_sequence	325
Section 60.1: Turn a std::tuple<T...> into function parameters	325
Section 60.2: Create a parameter pack consisting of integers	326
Section 60.3: Turn a sequence of indices into copies of an element	326
Chapter 61: Using std::unordered_map	328
Section 61.1: Declaration and Usage	328
Section 61.2: Some Basic Functions	328
Chapter 62: Standard Library Algorithms	329
Section 62.1: std::next_permutation	329
Section 62.2: std::for_each	329
Section 62.3: std::accumulate	330
Section 62.4: std::find	331
Section 62.5: std::min_element	333
Section 62.6: std::find_if	334
Section 62.7: Using std::nth_element To Find The Median (Or Other Quantiles)	335
Section 62.8: std::count	336
Section 62.9: std::count_if	337
Chapter 63: The ISO C++ Standard	339
Section 63.1: Current Working Drafts	339
Section 63.2: C++17	339
Section 63.3: C++11	340
Section 63.4: C++14	341
Section 63.5: C++98	342
Section 63.6: C++03	342
Section 63.7: C++20	343

第64章：内联变量	344
第64.1节：在类定义中定义静态数据成员	344
第65章：随机数生成	345
第65.1节：真正的随机值生成器	345
第65.2节：生成伪随机数	345
第65.3节：使用生成器进行多重分布	346
第66章：使用<chrono>头文件的日期和时间	347
第66.1节：使用<chrono>测量时间	347
第66.2节：计算两个日期之间的天数	347
第67章：排序	349
第67.1节：排序与序列容器	349
第67.2节：使用std::map排序（升序和降序）	349
第67.3节：通过重载的less运算符排序序列容器	351
第67.4节：使用比较函数对序列容器进行排序	352
第67.5节：使用lambda表达式（C++11）对序列容器进行排序	353
第67.6节：排序内置数组	354
第67.7节：使用指定顺序对序列容器进行排序	354
第68章：枚举	355
第68.1节：枚举的迭代	355
第68.2节：作用域枚举	356
第68.3节：C++11中的枚举前向声明	357
第68.4节：基本枚举声明	357
第68.5节：switch语句中的枚举	358
第69章：迭代	359
第69.1节：break	359
第69.2节：continue	359
第69.3节：do	359
第69.4节：while	359
第69.5节：基于范围的for循环	360
第69.6节：for	360
第70章：正则表达式	361
第70.1节：基本的regex_match和regex_search示例	361
第70.2节：regex_iterator示例	361
第70.3节：锚点	362
第70.4节：regex_replace示例	363
第70.5节：regex_token_iterator示例	363
第70.6节：量词	363
第70.7节：字符串拆分	365
第71章：实现定义的行为	366
第71.1节：整型大小	366
第71.2节：字符可能是无符号或有符号	368
第71.3节：数值类型的范围	368
第71.4节：浮点类型的值表示	369
第71.5节：从整数转换为有符号整数时的溢出	369
第71.6节：枚举的底层类型（因此大小）	370
第71.7节：指针的数值	370
第71.8节：字节中的位数	371
第72章：例外情况	372
第72.1节：捕获异常	372

Chapter 64: Inline variables	344
Section 64.1: Defining a static data member in the class definition	344
Chapter 65: Random number generation	345
Section 65.1: True random value generator	345
Section 65.2: Generating a pseudo-random number	345
Section 65.3: Using the generator for multiple distributions	346
Chapter 66: Date and time using <chrono> header	347
Section 66.1: Measuring time using <chrono>	347
Section 66.2: Find number of days between two dates	347
Chapter 67: Sorting	349
Section 67.1: Sorting and sequence containers	349
Section 67.2: sorting with std::map (ascending and descending)	349
Section 67.3: Sorting sequence containers by overloaded less operator	351
Section 67.4: Sorting sequence containers using compare function	352
Section 67.5: Sorting sequence containers using lambda expressions (C++11)	353
Section 67.6: Sorting built-in arrays	354
Section 67.7: Sorting sequence containers with specified ordering	354
Chapter 68: Enumeration	355
Section 68.1: Iteration over an enum	355
Section 68.2: Scoped enums	356
Section 68.3: Enum forward declaration in C++11	357
Section 68.4: Basic Enumeration Declaration	357
Section 68.5: Enumeration in switch statements	358
Chapter 69: Iteration	359
Section 69.1: break	359
Section 69.2: continue	359
Section 69.3: do	359
Section 69.4: while	359
Section 69.5: range-based for loop	360
Section 69.6: for	360
Chapter 70: Regular expressions	361
Section 70.1: Basic regex_match and regex_search Examples	361
Section 70.2: regex_iterator Example	361
Section 70.3: Anchors	362
Section 70.4: regex_replace Example	363
Section 70.5: regex_token_iterator Example	363
Section 70.6: Quantifiers	363
Section 70.7: Splitting a string	365
Chapter 71: Implementation-defined behavior	366
Section 71.1: Size of integral types	366
Section 71.2: Char might be unsigned or signed	368
Section 71.3: Ranges of numeric types	368
Section 71.4: Value representation of floating point types	369
Section 71.5: Overflow when converting from integer to signed integer	369
Section 71.6: Underlying type (and hence size) of an enum	370
Section 71.7: Numeric value of a pointer	370
Section 71.8: Number of bits in a byte	371
Chapter 72: Exceptions	372
Section 72.1: Catching exceptions	372

第72.2节：重新抛出（传播）异常	373
第72.3节：最佳实践：按值抛出，按常量引用捕获	374
第72.4节：自定义异常	375
第72.5节：std::uncaught_exceptions	377
第72.6节：常规函数的函数尝试块	378
第72.7节：嵌套异常	378
第72.8节：构造函数中的函数尝试块	380
第72.9节：析构函数中的函数尝试块	381
第73章：Lambda表达式	382
第73.1节：什么是lambda表达式？	382
第73.2节：指定返回类型	384
第73.3节：按值捕获	385
第73.4节：递归lambda表达式	386
第73.5节：默认捕获	388
第73.6节：类lambda表达式与this的捕获	388
第73.7节：按引用捕获	390
第73.8节：通用Lambda表达式	390
第73.9节：使用Lambda表达式进行内联参数包展开	391
第73.10节：广义捕获	393
第73.11节：转换为函数指针	394
第73.12节：使用函数对象将Lambda函数移植到C++03	394
第74章：值类别	396
第74.1节：值类别含义	396
第74.2节：右值 (rvalue)	396
第74.3节：x值 (xvalue)	397
第74.4节：纯右值 (prvalue)	397
第74.5节：左值 (lvalue)	398
第74.6节：通用左值 (glvalue)	398
第75章：预处理器	399
第75.1节：包含保护	399
第75.2节：条件逻辑与跨平台处理	400
第75.3节：X宏	401
第75.4节：宏	403
第75.5节：预定义宏	406
第75.6节：预处理器运算符	408
第75.7节：#pragma once	408
第75.8节：预处理器错误信息	409
第76章：C++中的数据结构	410
第76.1节：C++中的链表实现	410
第77章：模板	413
第77.1节：基本类模板	413
第77.2节：函数模板	413
第77.3节：可变参数模板数据结构	415
第77.4节：参数转发	417
第77.5节：部分模板特化	418
第77.6节：模板特化	420
第77.7节：别名模板	420
第77.8节：显式实例化	420
第77.9节：非类型模板参数	421
第77.10节：使用auto声明非类型模板参数	422

Section 72.2: Rethrow (propagate) exception	373
Section 72.3: Best practice: throw by value, catch by const reference	374
Section 72.4: Custom exception	375
Section 72.5: std::uncaught_exceptions	377
Section 72.6: Function Try Block for regular function	378
Section 72.7: Nested exception	378
Section 72.8: Function Try Blocks In constructor	380
Section 72.9: Function Try Blocks In destructor	381
Chapter 73: Lambdas	382
Section 73.1: What is a lambda expression?	382
Section 73.2: Specifying the return type	384
Section 73.3: Capture by value	385
Section 73.4: Recursive lambdas	386
Section 73.5: Default capture	388
Section 73.6: Class lambdas and capture of this	388
Section 73.7: Capture by reference	390
Section 73.8: Generic lambdas	390
Section 73.9: Using lambdas for inline parameter pack unpacking	391
Section 73.10: Generalized capture	393
Section 73.11: Conversion to function pointer	394
Section 73.12: Porting lambda functions to C++03 using functors	394
Chapter 74: Value Categories	396
Section 74.1: Value Category Meanings	396
Section 74.2: rvalue	396
Section 74.3: xvalue	397
Section 74.4: prvalue	397
Section 74.5: lvalue	398
Section 74.6: glvalue	398
Chapter 75: Preprocessor	399
Section 75.1: Include Guards	399
Section 75.2: Conditional logic and cross-platform handling	400
Section 75.3: X-macros	401
Section 75.4: Macros	403
Section 75.5: Predefined macros	406
Section 75.6: Preprocessor Operators	408
Section 75.7: #pragma once	408
Section 75.8: Preprocessor error messages	409
Chapter 76: Data Structures in C++	410
Section 76.1: Linked List implementation in C++	410
Chapter 77: Templates	413
Section 77.1: Basic Class Template	413
Section 77.2: Function Templates	413
Section 77.3: Variadic template data structures	415
Section 77.4: Argument forwarding	417
Section 77.5: Partial template specialization	418
Section 77.6: Template Specialization	420
Section 77.7: Alias template	420
Section 77.8: Explicit instantiation	420
Section 77.9: Non-type template parameter	421
Section 77.10: Declaring non-type template arguments with auto	422

第77.11节：模板模板参数	423
第77.12节：模板参数的默认值	424
第78章：表达式模板	425
第78.1节：表达式模板的基本示例	425
第79章：奇异递归模板模式（CRTP）	429
第79.1节：奇异递归模板模式（CRTP）	429
第79.2节：使用CRTP避免代码重复	430
第80章：线程	432
第80.1节：创建std::thread	432
第80.2节：向线程传递引用	434
第80.3节：使用std::async代替std::thread	434
第80.4节：基本同步	435
第80.5节：创建一个简单的线程池	435
第80.6节：确保线程总是被join	437
第80.7节：当前线程的操作	438
第80.8节：使用条件变量	439
第80.9节：线程操作	441
第80.10节：线程局部存储	441
第80.11节：重新分配线程对象	442
第81章：线程同步结构	443
第81.1节：std::condition_variable any, std::cv_status	443
第81.2节：std::shared_lock	443
第81.3节：std::call_once, std::once_flag	443
第81.4节：对象锁定以实现高效访问	444
第82章：三法则、五法则与零法则	446
第82.1节：零法则	446
第82.2节：五法则	447
第82.3节：三法则	448
第82.4节：自我赋值保护	449
第83章：RAII：资源获取即初始化	451
第83.1节：锁定	451
第83.2节：ScopeSuccess (C++17)	452
第83.3节：ScopeFail (C++17)	453
第83.4节：Finally/ScopeExit	454
第84章：RTTI：运行时类型信息	455
第84.1节：dynamic_cast	455
第84.2节：typeid关键字	455
第84.3节：类型名称	456
第84.4节：在C++中何时使用哪种类型转换	456
第85章：互斥量	457
第85.1节：互斥量类型	457
第85.2节：std::lock	457
第85.3节：std::unique_lock, std::shared_lock, std::lock_guard	457
第85.4节：锁类别策略：std::try_to_lock, std::adopt_lock, std::defer_lock	458
第85.5节：std::mutex	459
第85.6节：std::scoped_lock (C++17)	459
第86章：递归互斥锁	460
第86.1节：std::recursive_mutex	460
第87章：信号量	461

Section 77.11: Template template parameters	423
Section 77.12: Default template parameter value	424
Chapter 78: Expression templates	425
Section 78.1: A basic example illustrating expression templates	425
Chapter 79: Curiously Recurring Template Pattern (CRTP)	429
Section 79.1: The Curiously Recurring Template Pattern (CRTP)	429
Section 79.2: CRTP to avoid code duplication	430
Chapter 80: Threading	432
Section 80.1: Creating a std::thread	432
Section 80.2: Passing a reference to a thread	434
Section 80.3: Using std::async instead of std::thread	434
Section 80.4: Basic Synchronization	435
Section 80.5: Create a simple thread pool	435
Section 80.6: Ensuring a thread is always joined	437
Section 80.7: Operations on the current thread	438
Section 80.8: Using Condition Variables	439
Section 80.9: Thread operations	441
Section 80.10: Thread-local storage	441
Section 80.11: Reassigning thread objects	442
Chapter 81: Thread synchronization structures	443
Section 81.1: std::condition_variable any, std::cv_status	443
Section 81.2: std::shared_lock	443
Section 81.3: std::call_once, std::once_flag	443
Section 81.4: Object locking for efficient access	444
Chapter 82: The Rule of Three, Five, And Zero	446
Section 82.1: Rule of Zero	446
Section 82.2: Rule of Five	447
Section 82.3: Rule of Three	448
Section 82.4: Self-assignment Protection	449
Chapter 83: RAII: Resource Acquisition Is Initialization	451
Section 83.1: Locking	451
Section 83.2: ScopeSuccess (c++17)	452
Section 83.3: ScopeFail (c++17)	453
Section 83.4: Finally/ScopeExit	454
Chapter 84: RTTI: Run-Time Type Information	455
Section 84.1: dynamic_cast	455
Section 84.2: The typeid keyword	455
Section 84.3: Name of a type	456
Section 84.4: When to use which cast in c++	456
Chapter 85: Mutexes	457
Section 85.1: Mutex Types	457
Section 85.2: std::lock	457
Section 85.3: std::unique_lock, std::shared_lock, std::lock_guard	457
Section 85.4: Strategies for lock classes: std::try_to_lock, std::adopt_lock, std::defer_lock	458
Section 85.5: std::mutex	459
Section 85.6: std::scoped_lock (C++ 17)	459
Chapter 86: Recursive Mutex	460
Section 86.1: std::recursive_mutex	460
Chapter 87: Semaphore	461

第87.1节：信号量 C++ 11	461	Section 87.1: Semaphore C++ 11	461
第87.2节：信号量类的应用	461	Section 87.2: Semaphore class in action	461
第88章：期货和承诺	463	Chapter 88: Futures and Promises	463
第88.1节：异步操作类	463	Section 88.1: Async operation classes	463
第88.2节：std::future 和 std::promise	463	Section 88.2: std::future and std::promise	463
第88.3节：延迟异步示例	463	Section 88.3: Deferred async example	463
第88.4节：std::packaged_task 和 std::future	464	Section 88.4: std::packaged_task and std::future	464
第88.5节：std::future_error 和 std::future_errc	464	Section 88.5: std::future_error and std::future_errc	464
第88.6节：std::future 和 std::async	465	Section 88.6: std::future and std::async	465
第89章：原子类型	468	Chapter 89: Atomic Types	468
第89.1节：多线程访问	468	Section 89.1: Multi-threaded Access	468
第90章：类型擦除	470	Chapter 90: Type Erasure	470
第90.1节：仅可移动的 `std::function`	470	Section 90.1: A move-only `std::function`	470
第90.2节：使用手动虚表擦除为常规类型	472	Section 90.2: Erasing down to a Regular type with manual vtable	472
第90.3节：基本机制	475	Section 90.3: Basic mechanism	475
第90.4节：擦除到连续的T缓冲区	476	Section 90.4: Erasing down to a contiguous buffer of T	476
第90.5节：使用std::any进行类型擦除	477	Section 90.5: Type erasing type erasure with std::any	477
第91章：显式类型转换	482	Chapter 91: Explicit type conversions	482
第91.1节：C风格转换	482	Section 91.1: C-style casting	482
第91.2节：去除const属性的转换	482	Section 91.2: Casting away constness	482
第91.3节：基类到派生类的转换	482	Section 91.3: Base to derived conversion	482
第91.4节：指针与整数之间的转换	483	Section 91.4: Conversion between pointer and integer	483
第91.5节：通过显式构造函数或显式转换函数进行转换	484	Section 91.5: Conversion by explicit constructor or explicit conversion function	484
第91.6节：隐式转换	484	Section 91.6: Implicit conversion	484
第91.7节：枚举转换	484	Section 91.7: Enum conversions	484
第91.8节：指向成员的指针的派生类到基类转换	486	Section 91.8: Derived to base conversion for pointers to members	486
第91.9节：void* 转换为 T*	486	Section 91.9: void* to T*	486
第91.10节：类型混用转换	487	Section 91.10: Type punning conversion	487
第92章：无名类型	488	Chapter 92: Unnamed types	488
第92.1节：无名类	488	Section 92.1: Unnamed classes	488
第92.2节：作为类型别名	488	Section 92.2: As a type alias	488
第92.3节：匿名成员	488	Section 92.3: Anonymous members	488
第92.4节：匿名联合体	489	Section 92.4: Anonymous Union	489
第93章：类型特征	490	Chapter 93: Type Traits	490
第93.1节：类型属性	490	Section 93.1: Type Properties	490
第93.2节：标准类型特征	491	Section 93.2: Standard type traits	491
第93.3节：使用 std::is_same<T, T> 的类型关系	492	Section 93.3: Type relations with std::is_same<T, T>	492
第93.4节：基本类型特征	493	Section 93.4: Fundamental type traits	493
第94章：返回类型协变	495	Chapter 94: Return Type Covariance	495
第94.1节：基例的协变结果版本，静态类型检查	495	Section 94.1: Covariant result version of the base example, static type checking	495
第94.2节：协变智能指针结果（自动清理）	495	Section 94.2: Covariant smart pointer result (automated cleanup)	495
第95章：对象类型的布局	497	Chapter 95: Layout of object types	497
第95.1节：类类型	497	Section 95.1: Class types	497
第95.2节：算术类型	499	Section 95.2: Arithmetic types	499
第95.3节：数组	500	Section 95.3: Arrays	500
第96章：类型推断	501	Chapter 96: Type Inference	501
第96.1节：数据类型：自动类型	501	Section 96.1: Data Type: Auto	501
第96.2节：Lambda自动推导	501	Section 96.2: Lambda auto	501
第96.3节：循环与自动推导	501	Section 96.3: Loops and auto	501

第97章：typedef和类型别名	503
第97.1节：基本的typedef语法	503
第97.2节：typedef的更复杂用法	503
第97.3节：使用typedef声明多种类型	504
第97.4节：使用“using”声明别名	504
第98章：类型推导	505
第98.1节：构造函数的模板参数推导	505
第98.2节：自动类型推导	505
第98.3节：模板类型推导	506
第99章：尾返回类型	508
第99.1节：避免限定嵌套类型名	508
第99.2节：Lambda表达式	508
第100章：对齐	509
第100.1节：控制对齐	509
第100.2节：查询类型的对齐	509
第101章：完美转发	511
第101.1节：工厂函数	511
第102章：decltype	512
第102.1节：基本示例	512
第102.2节：另一个示例	512
第103章：SFINAE（替换失败不是错误）	513
第103.1节：什么是SFINAE	513
第103.2节：void_t	513
第103.3节：enable_if	515
第103.4节：is_detected	516
第103.5节：具有大量选项的重载解析	518
第103.6节：函数模板中的尾随decltype	519
第103.7节：enable_if_all / enable_if_any	520
第104章：未定义行为	522
第104.1节：通过空指针读取或写入	522
第104.2节：使用未初始化的局部变量	522
第104.3节：访问越界索引	523
第104.4节：通过指向没有虚析构函数的基类的指针删除派生对象	523
第104.5节：扩展`std`或`posix`命名空间	523
第104.6节：无效的指针运算	524
第104.7节：非void返回类型函数缺少返回语句	525
第104.8节：访问悬空引用	525
第104.9节：整数除以零	526
第104.10节：按无效的位数进行移位	526
第104.11节：内存分配和释放的不正确配对	526
第104.12节：有符号整数溢出	527
第104.13节：多个不相同的定义（唯一定义规则）	527
第104.14节：修改const对象	528
第104.15节：从[[noreturn]]函数返回	529
第104.16节：无限模板递归	529
第104.17节：转换为或从浮点类型时的溢出	530
第104.18节：修改字符串字面量	530
第104.19节：以错误类型访问对象	530

Chapter 97: Typedef and type aliases	503
Section 97.1: Basic typedef syntax	503
Section 97.2: More complex uses of typedef	503
Section 97.3: Declaring multiple types with typedef	504
Section 97.4: Alias declaration with "using"	504
Chapter 98: type deduction	505
Section 98.1: Template parameter deduction for constructors	505
Section 98.2: Auto Type Deduction	505
Section 98.3: Template Type Deduction	506
Chapter 99: Trailing return type	508
Section 99.1: Avoid qualifying a nested type name	508
Section 99.2: Lambda expressions	508
Chapter 100: Alignment	509
Section 100.1: Controlling alignment	509
Section 100.2: Querying the alignment of a type	509
Chapter 101: Perfect Forwarding	511
Section 101.1: Factory functions	511
Chapter 102: decltype	512
Section 102.1: Basic Example	512
Section 102.2: Another example	512
Chapter 103: SFINAE (Substitution Failure Is Not An Error)	513
Section 103.1: What is SFINAE	513
Section 103.2: void_t	513
Section 103.3: enable_if	515
Section 103.4: is_detected	516
Section 103.5: Overload resolution with a large number of options	518
Section 103.6: trailing decltype in function templates	519
Section 103.7: enable_if_all / enable_if_any	520
Chapter 104: Undefined Behavior	522
Section 104.1: Reading or writing through a null pointer	522
Section 104.2: Using an uninitialized local variable	522
Section 104.3: Accessing an out-of-bounds index	523
Section 104.4: Deleting a derived object via a pointer to a base class that doesn't have a virtual destructor	523
Section 104.5: Extending the `std` or `posix` Namespace	523
Section 104.6: Invalid pointer arithmetic	524
Section 104.7: No return statement for a function with a non-void return type	525
Section 104.8: Accessing a dangling reference	525
Section 104.9: Integer division by zero	526
Section 104.10: Shifting by an invalid number of positions	526
Section 104.11: Incorrect pairing of memory allocation and deallocation	526
Section 104.12: Signed Integer Overflow	527
Section 104.13: Multiple non-identical definitions (the One Definition Rule)	527
Section 104.14: Modifying a const object	528
Section 104.15: Returning from a [[noreturn]] function	529
Section 104.16: Infinite template recursion	529
Section 104.17: Overflow during conversion to or from floating point type	530
Section 104.18: Modifying a string literal	530
Section 104.19: Accessing an object as the wrong type	530

第104.20节：指向成员的指针的无效派生到基类转换	531
第104.21节：销毁已被销毁的对象	531
第104.22节：通过指向成员的指针访问不存在的成员	532
第104.23节：无效的基类到派生类的静态转换	532
第104.24节：浮点数溢出	532
第104.25节：从构造函数或析构函数调用（纯）虚函数成员	532
第104.26节：通过不匹配的函数指针类型调用函数	533
第105章：重载解析	534
第105.1节：参数到形参代价的分类	534
第105.2节：算术提升和转换	534
第105.3节：基于转发引用的重载	535
第105.4节：精确匹配	536
第105.5节：基于const性和易变性的重载	536
第105.6节：名称查找与访问检查	537
第105.7节：类层次结构内的重载	538
第105.8节：重载解析步骤	539
第106章：移动语义	541
第106.1节：移动语义	541
第106.2节：使用 <code>std::move</code> 将复杂度从 $O(n^2)$ 降低到 $O(n)$	541
第106.3节：移动构造函数	544
第106.4节：重新使用已移动的对象	546
第106.5节：移动赋值	546
第106.6节：在容器上使用移动语义	547
第107章：Pimpl惯用法	549
第107.1节：基本的Pimpl惯用法	549
第108章：auto	551
第108.1节：基本的auto示例	551
第108.2节：泛型lambda（C++14）	551
第108.3节：auto与代理对象	552
第108.4节：auto与表达式模板	552
第108.5节：auto、const与引用	553
第108.6节：尾随返回类型	553
第109章：拷贝省略	555
第109.1节：拷贝省略的目的	555
第109.2节：保证的拷贝省略	556
第109.3节：参数省略	557
第109.4节：返回值省略	557
第109.5节：命名返回值省略	557
第109.6节：复制初始化省略	558
第110章：折叠表达式	559
第110.1节：一元折叠	559
第110.2节：二元折叠	559
第110.3节：逗号折叠	560
第111章：联合体	561
第111.1节：未定义行为	561
第111.2节：基本联合特性	561
第111.3节：典型用法	561
第112章：C++中的设计模式实现	563
第112.1节：适配器模式	563

Section 104.20: Invalid derived-to-base conversion for pointers to members	531
Section 104.21: Destroying an object that has already been destroyed	531
Section 104.22: Access to nonexistent member through pointer to member	532
Section 104.23: Invalid base-to-derived static cast	532
Section 104.24: Floating point overflow	532
Section 104.25: Calling (Pure) Virtual Members From Constructor Or Destructor	532
Section 104.26: Function call through mismatched function pointer type	533
Chapter 105: Overload resolution	534
Section 105.1: Categorization of argument to parameter cost	534
Section 105.2: Arithmetic promotions and conversions	534
Section 105.3: Overloading on Forwarding Reference	535
Section 105.4: Exact match	536
Section 105.5: Overloading on constness and volatility	536
Section 105.6: Name lookup and access checking	537
Section 105.7: Overloading within a class hierarchy	538
Section 105.8: Steps of Overload Resolution	539
Chapter 106: Move Semantics	541
Section 106.1: Move semantics	541
Section 106.2: Using <code>std::move</code> to reduce complexity from $O(n^2)$ to $O(n)$	541
Section 106.3: Move constructor	544
Section 106.4: Re-use a moved object	546
Section 106.5: Move assignment	546
Section 106.6: Using move semantics on containers	547
Chapter 107: Pimpl Idiom	549
Section 107.1: Basic Pimpl idiom	549
Chapter 108: auto	551
Section 108.1: Basic auto sample	551
Section 108.2: Generic lambda (C++14)	551
Section 108.3: auto and proxy objects	552
Section 108.4: auto and Expression Templates	552
Section 108.5: auto, const, and references	553
Section 108.6: Trailing return type	553
Chapter 109: Copy Elision	555
Section 109.1: Purpose of copy elision	555
Section 109.2: Guaranteed copy elision	556
Section 109.3: Parameter elision	557
Section 109.4: Return value elision	557
Section 109.5: Named return value elision	557
Section 109.6: Copy initialization elision	558
Chapter 110: Fold Expressions	559
Section 110.1: Unary Folds	559
Section 110.2: Binary Folds	559
Section 110.3: Folding over a comma	560
Chapter 111: Unions	561
Section 111.1: Undefined Behavior	561
Section 111.2: Basic Union Features	561
Section 111.3: Typical Use	561
Chapter 112: Design pattern implementation in C++	563
Section 112.1: Adapter Pattern	563

第112.2节：观察者模式	565
第112.3节：工厂模式	568
第112.4节：带有流畅接口的建造者模式	568
第113章：单例设计模式	572
第113.1节：延迟初始化	572
第113.2节：静态析构安全单例	573
第113.3节：线程安全单例	573
第113.4节：子类	573
第114章：用户自定义字面量	575
第114.1节：自制二进制用户自定义字面量	575
第114.2节：用于持续时间的标准用户自定义字面量	575
第114.3节：带有长双精度值的用户自定义字面量	576
第114.4节：用于字符串的标准用户自定义字面量	576
第114.5节：用于复数的标准用户自定义字面量	577
第115章：内存管理	578
第115.1节：自由存储（堆，动态分配...）	578
第115.2节：定位 new	579
第115.3节：堆栈	580
第116章：C++11内存模型	581
第116.1节：内存模型的必要性	582
第116.2节：栅栏示例	584
第117章：作用域	585
第117.1节：全局变量	585
第117.2节：简单块作用域	585
第118章：static assert	587
第118.1节：static assert	587
第119章：constexpr	588
第119.1节：constexpr变量	588
第119.2节：静态if语句	589
第119.3节：constexpr函数	590
第120章：一义性规则（ODR）	592
第120.1节：通过重载解析违反ODR	592
第120.2节：多重定义函数	592
第120.3节：内联函数	593
第121章：未指定行为	595
第121.1节：超出范围的枚举值	595
第121.2节：函数参数的求值顺序	595
第121.3节：某些reinterpret_cast转换的结果	596
第121.4节：引用所占空间	597
第121.5节：大多数标准库类的移动后状态	597
第121.6节：某些指针比较的结果	598
第121.7节：从无效void*值的静态转换	598
第121.8节：跨翻译单元的全局变量初始化顺序	598
第122章：依赖参数的名称查找	600
第122.1节：包含哪些函数	600
第123章：属性	601
第123.1节：[[fallthrough]]	601
第123.2节：[[nodiscard]]	601
第123.3节：[[deprecated]] 和 [[deprecated("reason")]]	602

Section 112.2: Observer pattern	565
Section 112.3: Factory Pattern	568
Section 112.4: Builder Pattern with Fluent API	568
Chapter 113: Singleton Design Pattern	572
Section 113.1: Lazy Initialization	572
Section 113.2: Static deinitialization-safe singleton	573
Section 113.3: Thread-safe Singeton	573
Section 113.4: Subclasses	573
Chapter 114: User-Defined Literals	575
Section 114.1: Self-made user-defined literal for binary	575
Section 114.2: Standard user-defined literals for duration	575
Section 114.3: User-defined literals with long double values	576
Section 114.4: Standard user-defined literals for strings	576
Section 114.5: Standard user-defined literals for complex	577
Chapter 115: Memory management	578
Section 115.1: Free Storage (Heap, Dynamic Allocation ...)	578
Section 115.2: Placement new	579
Section 115.3: Stack	580
Chapter 116: C++11 Memory Model	581
Section 116.1: Need for Memory Model	582
Section 116.2: Fence example	584
Chapter 117: Scopes	585
Section 117.1: Global variables	585
Section 117.2: Simple block scope	585
Chapter 118: static assert	587
Section 118.1: static assert	587
Chapter 119: constexpr	588
Section 119.1: constexpr variables	588
Section 119.2: Static if statement	589
Section 119.3: constexpr functions	590
Chapter 120: One Definition Rule (ODR)	592
Section 120.1: ODR violation via overload resolution	592
Section 120.2: Multiply defined function	592
Section 120.3: Inline functions	593
Chapter 121: Unspecified behavior	595
Section 121.1: Value of an out-of-range enum	595
Section 121.2: Evaluation order of function arguments	595
Section 121.3: Result of some reinterpret_cast conversions	596
Section 121.4: Space occupied by a reference	597
Section 121.5: Moved-from state of most standard library classes	597
Section 121.6: Result of some pointer comparisons	598
Section 121.7: Static cast from bogus void* value	598
Section 121.8: Order of initialization of globals across TU	598
Chapter 122: Argument Dependent Name Lookup	600
Section 122.1: What functions are found	600
Chapter 123: Attributes	601
Section 123.1: [[fallthrough]]	601
Section 123.2: [[nodiscard]]	601
Section 123.3: [[deprecated]] and [[deprecated("reason")]]	602

第123.4节：[[maybe_unused]]	602
第123.5节：[[noreturn]]	603
第124章：C++中的递归	605
第124.1节：使用尾递归和斐波那契式递归解决斐波那契数列	605
第124.2节：带备忘录的递归	605
第125章：算术元编程	607
第125.1节：以O(log n)计算幂	607
第126章：可调用对象	609
第126.1节：函数指针	609
第126.2节：带有operator()的类（函数对象）	609
第127章：客户端服务器示例	611
第127.1节：Hello TCP客户端	611
第127.2节：Hello TCP服务器	612
第128章：常量正确性	616
第128.1节：基础知识	616
第128.2节：常量正确的类设计	616
第128.3节：常量正确的函数参数	618
第128.4节：作为文档的常量正确性	620
第129章：参数包	624
第129.1节：带参数包的模板	624
第129.2节：参数包的展开	624
第130章：构建系统	625
第130.1节：使用CMake生成构建环境	625
第130.2节：使用GNU make编译	626
第130.3节：使用SCons构建	628
第130.4节：Autotools (GNU)	628
第130.5节：Ninja	629
第130.6节：NMAKE（微软程序维护工具）	629
第131章：使用OpenMP进行并发	630
第131.1节：OpenMP：并行区段	630
第131.2节：OpenMP：并行区段	630
第131.3节：OpenMP：并行For循环	631
第131.4节：OpenMP：并行收集/归约	631
第132章：资源管理	633
第132.1节：资源获取即初始化	633
第132.2节：互斥锁与线程安全	634
第133章：存储类说明符	636
第133.1节：extern	636
第133.2节：register	637
第133.3节：static	637
第133.4节：auto	638
第133.5节：mutable	638
第134章：链接说明	640
第134.1节：类Unix操作系统的信号处理程序	640
第134.2节：使C库头文件兼容C++	640
第135章：数字分隔符	642
第135.1节：数字分隔符	642
第136章：C语言不兼容性	643

Section 123.4: [[maybe_unused]]	602
Section 123.5: [[noreturn]]	603
Chapter 124: Recursion in C++	605
Section 124.1: Using tail recursion and Fibonacci-style recursion to solve the Fibonacci sequence	605
Section 124.2: Recursion with memoization	605
Chapter 125: Arithmetic Metaprogramming	607
Section 125.1: Calculating power in O(log n)	607
Chapter 126: Callable Objects	609
Section 126.1: Function Pointers	609
Section 126.2: Classes with operator() (Functors)	609
Chapter 127: Client server examples	611
Section 127.1: Hello TCP Client	611
Section 127.2: Hello TCP Server	612
Chapter 128: Const Correctness	616
Section 128.1: The Basics	616
Section 128.2: Const Correct Class Design	616
Section 128.3: Const Correct Function Parameters	618
Section 128.4: Const Correctness as Documentation	620
Chapter 129: Parameter packs	624
Section 129.1: A template with a parameter pack	624
Section 129.2: Expansion of a parameter pack	624
Chapter 130: Build Systems	625
Section 130.1: Generating Build Environment with CMake	625
Section 130.2: Compiling with GNU make	626
Section 130.3: Building with SCons	628
Section 130.4: Autotools (GNU)	628
Section 130.5: Ninja	629
Section 130.6: NMAKE (Microsoft Program Maintenance Utility)	629
Chapter 131: Concurrency With OpenMP	630
Section 131.1: OpenMP: Parallel Sections	630
Section 131.2: OpenMP: Parallel Sections	630
Section 131.3: OpenMP: Parallel For Loop	631
Section 131.4: OpenMP: Parallel Gathering / Reduction	631
Chapter 132: Resource Management	633
Section 132.1: Resource Acquisition Is Initialization	633
Section 132.2: Mutexes & Thread Safety	634
Chapter 133: Storage class specifiers	636
Section 133.1: extern	636
Section 133.2: register	637
Section 133.3: static	637
Section 133.4: auto	638
Section 133.5: mutable	638
Chapter 134: Linkage specifications	640
Section 134.1: Signal handler for Unix-like operating system	640
Section 134.2: Making a C library header compatible with C++	640
Chapter 135: Digit separators	642
Section 135.1: Digit Separator	642
Chapter 136: C incompatibilities	643

第136.1节：保留关键字	643
第136.2节：弱类型指针	643
第136.3节：goto或switch	643
第137章：经典C++示例的并排比较——使用C++与C++11	
与C++14与C++17的比较	644
第137.1节：遍历容器	644
第138章：编译与构建	645
第138.1节：使用GCC编译	645
第138.2节：使用Visual Studio (图形界面) 编译——Hello World	646
第138.3节：在线编译器	651
第138.4节：使用Visual C++编译 (命令行)	653
第138.5节：使用Clang编译	656
第138.6节：C++编译过程	656
第138.7节：使用Code::Blocks编译 (图形界面)	658
第139章：常见的编译/链接错误 (GCC)	661
第139.1节：未定义的对`***`的引用	661
第139.2节：错误：`***`未在此作用域中声明	661
第139.3节：致命错误：`**`：没有此类文件或目录	663
第140章：C++中的更多未定义行为	664
第140.1节：在初始化列表中引用非静态成员	664
第141章：C++中的单元测试	665
第141.1节：Google测试	665
第141.2节：Catch	665
第142章：C++调试及防错工具与技术	667
第142.1节：静态分析	667
第142.2节：使用GDB进行段错误分析	668
第142.3节：整洁代码	669
第143章：C++中的优化	671
第143.1节：性能介绍	671
第143.2节：空基类优化	671
第143.3节：通过执行更少代码进行优化	672
第143.4节：使用高效容器	673
第143.5节：小对象优化	674
第144章：优化	676
第144.1节：内联展开/内联	676
第144.2节：空基优化	676
第145章：性能分析	678
第145.1节：使用gcc和gprof进行性能分析	678
第145.2节：使用gperf2dot生成调用图	678
第145.3节：使用gcc和Google性能工具进行CPU使用率分析	679
第146章：重构技术	681
第146.1节：Goto语句清理	681
学分	682
你可能也喜欢	690

Section 136.1: Reserved Keywords	643
Section 136.2: Weakly typed pointers	643
Section 136.3: goto or switch	643
Chapter 137: Side by Side Comparisons of classic C++ examples solved via C++ vs C++11	
vs C++14 vs C++17	644
Section 137.1: Looping through a container	644
Chapter 138: Compiling and Building	645
Section 138.1: Compiling with GCC	645
Section 138.2: Compiling with Visual Studio (Graphical Interface) - Hello World	646
Section 138.3: Online Compilers	651
Section 138.4: Compiling with Visual C++ (Command Line)	653
Section 138.5: Compiling with Clang	656
Section 138.6: The C++ compilation process	656
Section 138.7: Compiling with Code::Blocks (Graphical interface)	658
Chapter 139: Common compile/linker errors (GCC)	661
Section 139.1: undefined reference to `***'	661
Section 139.2: error: `***' was not declared in this scope	661
Section 139.3: fatal error: `**': No such file or directory	663
Chapter 140: More undefined behaviors in C++	664
Section 140.1: Referring to non-static members in initializer lists	664
Chapter 141: Unit Testing in C++	665
Section 141.1: Google Test	665
Section 141.2: Catch	665
Chapter 142: C++ Debugging and Debug-prevention Tools & Techniques	667
Section 142.1: Static analysis	667
Section 142.2: Segfault analysis with GDB	668
Section 142.3: Clean code	669
Chapter 143: Optimization in C++	671
Section 143.1: Introduction to performance	671
Section 143.2: Empty Base Class Optimization	671
Section 143.3: Optimizing by executing less code	672
Section 143.4: Using efficient containers	673
Section 143.5: Small Object Optimization	674
Chapter 144: Optimization	676
Section 144.1: Inline Expansion/Inlining	676
Section 144.2: Empty base optimization	676
Chapter 145: Profiling	678
Section 145.1: Profiling with gcc and gprof	678
Section 145.2: Generating callgraph diagrams with gperf2dot	678
Section 145.3: Profiling CPU Usage with gcc and Google Perf Tools	679
Chapter 146: Refactoring Techniques	681
Section 146.1: Goto Cleanup	681
Credits	682
You may also like	690

请随意免费分享此PDF，
本书的最新版本可从以下网址下载：
<https://goalkicker.com/CPlusPlusBook>

这本专业人士的C++笔记是从Stack Overflow
文档汇编而成，内容由Stack Overflow的优秀人士撰写。
文本内容采用知识共享署名-相同方式共享许可协议发布，详见本书末尾对各章节贡
献者的致谢。图片版权归各自所有者所有，除非另有说明

这是一本非官方的免费书籍，旨在教育用途，与官方C++组织或公司以及Stac
k Overflow无关。所有商标和注册商标均为其各自公司所有者的财产

本书中提供的信息不保证正确或准确，使用风险自负

请将反馈和更正发送至 web@petercv.com

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
<https://goalkicker.com/CPlusPlusBook>

This C++ Notes for Professionals book is compiled from [Stack Overflow Documentation](#), the content is written by the beautiful people at Stack Overflow.
Text content is released under Creative Commons BY-SA, see credits at the end
of this book whom contributed to the various chapters. Images may be copyright
of their respective owners unless otherwise specified

This is an unofficial free book created for educational purposes and is not
affiliated with official C++ group(s) or company(s) nor Stack Overflow. All
trademarks and registered trademarks are the property of their respective
company owners

The information presented in this book is not guaranteed to be correct nor
accurate, use at your own risk

Please send feedback and corrections to web@petercv.com

第1章：C++入门

版本	标准	发布日期
C++98	ISO/IEC 14882:1998	1998-09-01
C++03	ISO/IEC 14882:2003	2003-10-16
C++11	ISO/IEC 14882:2011	2011-09-01
C++14	ISO/IEC 14882:2014	2014-12-15
C++17	待定	2017-01-01
C++20	待定	2020-01-01

第1.1节：你好，世界

该程序将Hello World!打印到标准输出流：

```
#include <iostream>

int main()
{
    std::cout << "Hello World!" << std::endl;
}
```

在Coliru上实时查看。

分析

让我们详细检查这段代码的每个部分：

- `#include <iostream>` 是一个**预处理指令**，用于包含标准C++头文件 `iostream` 的内容。

`iostream` 是一个**标准库头文件**，包含标准输入和输出流的定义。这些定义包含在下面解释的 `std` 命名空间中。

标准输入/输出（I/O）流提供程序从外部系统——通常是终端——获取输入和输出的方式。

- `int main() { ... }` 定义了一个名为 `main` 的新函数。按照惯例，程序执行时会调用 `main` 函数。C++ 程序中必须且只能有一个 `main` 函数，并且它必须始终返回一个 `int` 类型的数字。

这里，`int` 被称为函数的返回类型。由 `main` 函数返回的值是一个退出代码（exit code）。

按照惯例，程序退出代码为 0 或 `EXIT_SUCCESS` 时，执行该程序的系统会将其解释为成功。任何其他返回代码都表示错误。

如果没有 `return` 语句，`main` 函数（因此程序本身）默认返回 0。在此示例中，我们不需要显式写出 `return 0;`

除返回 `void` 类型的函数外，所有其他函数必须根据其返回类型显式返回一个值，或者根本不返回。

Chapter 1: Getting started with C++

Version	Standard	Release Date
C++98	ISO/IEC 14882:1998	1998-09-01
C++03	ISO/IEC 14882:2003	2003-10-16
C++11	ISO/IEC 14882:2011	2011-09-01
C++14	ISO/IEC 14882:2014	2014-12-15
C++17	TBD	2017-01-01
C++20	TBD	2020-01-01

Section 1.1: Hello World

This program prints Hello World! to the standard output stream:

```
#include <iostream>

int main()
{
    std::cout << "Hello World!" << std::endl;
}
```

See it [live on Coliru](#).

Analysis

Let's examine each part of this code in detail:

- `#include <iostream>` is a **preprocessor directive** that includes the content of the standard C++ header file `iostream`.

`iostream` is a **standard library header file** that contains definitions of the standard input and output streams. These definitions are included in the `std` namespace, explained below.

The **standard input/output (I/O) streams** provide ways for programs to get input from and output to an external system -- usually the terminal.

- `int main() { ... }` defines a new function named `main`. By convention, the `main` function is called upon execution of the program. There must be only one `main` function in a C++ program, and it must always return a number of the `int` type.

Here, the `int` is what is called the function's return type. The value returned by the `main` function is an **exit code**.

By convention, a program exit code of 0 or `EXIT_SUCCESS` is interpreted as success by a system that executes the program. Any other return code is associated with an error.

If no `return` statement is present, the `main` function (and thus, the program itself) returns 0 by default. In this example, we don't need to explicitly write `return 0;`.

All other functions, except those that return the `void` type, must explicitly return a value according to their return type, or else must not return at all.

- `std::cout << "Hello World!" << std::endl;` 将“Hello World!”打印到标准输出流：

- `std`是一个命名空间，`::`是作用域解析运算符（scope resolution operator），允许在命名空间内按名称查找对象。

命名空间有很多。这里，我们使用`::`表示我们想使用`std`命名空间中的`cout`。
更多信息请参阅[作用域解析运算符 - Microsoft文档](#)。

- `std::cout` 是定义在 `iostream` 中的 标准输出流 对象，用于打印到标准输出 (`stdout`)。
- `<<` 是在此上下文中，流插入运算符，之所以称为此名，是因为它将一个对象插入到流对象中。

标准库定义了`<<`运算符，用于将某些数据类型的数据插入到输出流中。`stream << content` 将`content`插入流中并返回同一个但已更新的流。这允许流插入操作链式调用：`std::cout << "Foo" << "Bar";` 会在控制台打印“`FooBar`”。

- “`Hello World!`” 是一个字符字符串字面量，或称为“文本字面量”。字符字符串字面量的流插入运算符定义在文件 `iostream` 中。
- `std::endl` 是一个特殊的输入/输出流操纵器对象，也定义在文件 `iostream` 中。将操纵器插入流中会改变流的状态。

流操纵器 `std::endl` 做两件事：首先插入换行符，然后刷新流缓冲区，强制文本显示在控制台上。这确保插入到流中的数据实际出现在你的控制台上。（流数据通常存储在缓冲区中，然后批量“刷新”，除非你立即强制刷新。）

避免刷新的一种替代方法是：

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

其中 `\n` 是换行符的字符转义序列。

- 分号`(;)`通知编译器语句已结束。所有C++语句和类定义都需要以分号结尾。

- `std::cout << "Hello World!" << std::endl;` prints "Hello World!" to the standard output stream:

- `std` is a namespace, and `::` is the **scope resolution operator** that allows look-ups for objects by name within a namespace.

There are many namespaces. Here, we use `::` to show we want to use `cout` from the `std` namespace.
For more information refer to [Scope Resolution Operator - Microsoft Documentation](#).

- `std::cout` is the **standard output stream** object, defined in `iostream`, and it prints to the standard output (`stdout`).
- `<<` is, *in this context*, the **stream insertion operator**, so called because it *inserts* an object into the *stream* object.

The standard library defines the `<<` operator to perform data insertion for certain data types into output streams. `stream << content` inserts content into the stream and returns the same, but updated stream. This allows stream insertions to be chained: `std::cout << "Foo" << "Bar";` prints "FooBar" to the console.

- “`Hello World!`” is a **character string literal**, or a “text literal.” The stream insertion operator for character string literals is defined in file `iostream`.
- `std::endl` is a special **I/O stream manipulator** object, also defined in file `iostream`. Inserting a manipulator into a stream changes the state of the stream.

The stream manipulator `std::endl` does two things: first it inserts the end-of-line character and then it flushes the stream buffer to force the text to show up on the console. This ensures that the data inserted into the stream actually appear on your console. (Stream data is usually stored in a buffer and then “flushed” in batches unless you force a flush immediately.)

An alternate method that avoids the flush is:

```
std::cout << "Hello World!\n";
```

where `\n` is the **character escape sequence** for the newline character.

- The semicolon`(;)`notifies the compiler that a statement has ended. All C++ statements and class definitions require an ending/terminating semicolon.

第1.2节：注释

注释是一种在源代码中插入任意文本的方法，C++ 编译器不会将其解释为任何功能性内容。注释用于提供程序设计或方法的见解。

C++ 中有两种类型的注释：

单行注释

双斜杠序列`//`会将直到换行符的所有文本标记为注释：

```
int main()
{
```

Section 1.2: Comments

A **comment** is a way to put arbitrary text inside source code without having the C++ compiler interpret it with any functional meaning. Comments are used to give insight into the design or method of a program.

There are two types of comments in C++:

Single-Line Comments

The double forward-slash sequence `//` will mark all text until a newline as a comment:

```
int main()
{
```

```
// 这是一个单行注释。  
int a; // 这也是一个单行注释  
int i; // 这是另一个单行注释  
}
```

C 风格/块注释

序列/*用于声明注释块的开始，序列*/用于声明注释的结束。开始和结束序列之间的所有文本都被解释为注释，即使文本本身是有效的 C++ 语法。这些有时被称为“C 风格”注释，因为这种注释语法继承自 C++ 的前身语言 C：

```
int main()  
{  
    /*  
     * 这是一个块注释。  
     */  
    int a;  
}
```

在任何块注释中，你可以写任何你想写的内容。当编译器遇到符号 */ 时，它会终止块注释：

```
int main()  
{  
    /* 一个包含符号 /* 的块注释  
       注意编译器不会被第二个 /* 影响  
       但是，一旦遇到结束块注释符号，  
       注释就结束了。  
    */  
    int a;  
}
```

上述示例是有效的 C++ (和 C) 代码。然而，在块注释中包含额外的 /* 可能会导致某些编译器发出警告。

块注释也可以在单行内开始和结束。例如：

```
void SomeFunction(/* 参数 1 */ int a, /* 参数 2 */ int b);
```

注释的重要性

与所有编程语言一样，注释提供了多种好处：

- 对代码进行明确的文档说明，使其更易于阅读和维护
- 解释代码的目的和功能
- 关于代码的历史或设计原因的详细信息
- 将版权/许可证、项目备注、特别感谢、贡献者名单等直接放置在源代码中。

然而，注释也有其缺点：

- 必须维护注释以反映代码的任何更改
- 过多的注释往往会使代码不易读

通过编写清晰、自我说明的代码，可以减少对注释的需求。一个简单的例子是为变量、函数和类型使用说明性名称。将逻辑相关的任务拆分成独立的函数与此密切相关。

```
// This is a single-line comment.  
int a; // this also is a single-line comment  
int i; // this is another single-line comment  
}
```

C-Style/Block Comments

The sequence /* is used to declare the start of the comment block and the sequence */ is used to declare the end of comment. All text between the start and end sequences is interpreted as a comment, even if the text is otherwise valid C++ syntax. These are sometimes called "C-style" comments, as this comment syntax is inherited from C++'s predecessor language, C:

```
int main()  
{  
    /*  
     * This is a block comment.  
     */  
    int a;  
}
```

In any block comment, you can write anything you want. When the compiler encounters the symbol */, it terminates the block comment:

```
int main()  
{  
    /* A block comment with the symbol */  
    Note that the compiler is not affected by the second /*  
    however, once the end-block-comment symbol is reached,  
    the comment ends.  
    */  
    int a;  
}
```

The above example is valid C++ (and C) code. However, having additional /* inside a block comment might result in a warning on some compilers.

Block comments can also start and end *within* a single line. For example:

```
void SomeFunction(/* argument 1 */ int a, /* argument 2 */ int b);
```

Importance of Comments

As with all programming languages, comments provide several benefits:

- Explicit documentation of code to make it easier to read/maintain
- Explanation of the purpose and functionality of code
- Details on the history or reasoning behind the code
- Placement of copyright/licenses, project notes, special thanks, contributor credits, etc. directly in the source code.

However, comments also have their downsides:

- They must be maintained to reflect any changes in the code
- Excessive comments tend to make the code *less* readable

The need for comments can be reduced by writing clear, self-documenting code. A simple example is the use of explanatory names for variables, functions, and types. Factoring out logically related tasks into discrete functions goes hand-in-hand with this.

用于禁用代码的注释标记

在开发过程中，注释也可以用来快速禁用部分代码而不删除它。这通常对测试或调试很有用，但除了临时修改外，这种做法并不符合良好风格。这通常被称为“注释掉”。

同样，将旧版本的代码保留在注释中以供参考是不被提倡的，因为这会使文件变得杂乱，而相比通过版本控制系统查看代码历史，这种做法价值不大。

第1.3节：标准C++编译过程

可执行的C++程序代码通常由编译器生成。

编译器**compiler**是一种将编程语言代码翻译成另一种形式（更）直接可供计算机执行的程序。使用编译器翻译代码称为编译**compilation**。

C++继承了其“父”语言C的编译过程形式。以下是C++编译的四个主要步骤列表：

1.C++预处理器将任何包含的头文件内容复制到源代码文件中，生成宏代码，并用#define定义的符号常量的值替换它们。

2.C++预处理器生成的扩展源代码文件被编译成适合平台的汇编语言。

3.编译器生成的汇编代码被组装成适合平台的目标代码。

4.汇编器生成的目标代码文件与任何使用的库函数的目标代码文件链接在一起，生成可执行文件。

- 注意：有些编译代码会被链接，但不是为了创建最终程序。通常，这种“链接”的代码也可以打包成其他程序可用的格式。这种“打包的、可用的代码集合”就是C++程序员所说的库library。

许多C++编译器也可能为了方便或额外分析而合并或拆分编译过程的某些部分。许多C++程序员会使用不同的工具，但所有工具在参与程序生成时通常都会遵循这个通用过程。

下面的链接扩展了这一讨论，并提供了一个很好的图示帮助理解。[1]:

<http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html>

第1.4节：函数

函数（function）是一段表示语句序列的代码单元。

函数可以接受参数或值，并返回一个单一的值（也可以不返回）。要使用函数，需要对参数值进行函数调用，函数调用本身会被其返回值替代。

每个函数都有一个类型签名——即其参数类型和返回类型。

函数的概念来源于过程和数学函数。

- 注意：C++函数本质上是过程，并不完全遵循数学函数的定义或规则。

函数通常用于执行特定任务，并且可以从程序的其他部分调用。函数必须在程序中被调用之前声明和定义。

Comment markers used to disable code

During development, comments can also be used to quickly disable portions of code without deleting it. This is often useful for testing or debugging purposes, but is not good style for anything other than temporary edits. This is often referred to as “commenting out”.

Similarly, keeping old versions of a piece of code in a comment for reference purposes is frowned upon, as it clutters files while offering little value compared to exploring the code's history via a versioning system.

Section 1.3: The standard C++ compilation process

Executable C++ program code is usually produced by a compiler.

A **compiler** is a program that translates code from a programming language into another form which is (more) directly executable for a computer. Using a compiler to translate code is called **compilation**.

C++ inherits the form of its compilation process from its "parent" language, C. Below is a list showing the four major steps of compilation in C++:

1. The C++ preprocessor copies the contents of any included header files into the source code file, generates macro code, and replaces symbolic constants defined using #define with their values.
 2. The expanded source code file produced by the C++ preprocessor is compiled into assembly language appropriate for the platform.
 3. The assembler code generated by the compiler is assembled into appropriate object code for the platform.
 4. The object code file generated by the assembler is linked together with the object code files for any library functions used to produce an executable file.
- Note: some compiled code is linked together, but not to create a final program. Usually, this "linked" code can also be packaged into a format that can be used by other programs. This "bundle of packaged, usable code" is what C++ programmers refer to as a **library**.

Many C++ compilers may also merge or un-merge certain parts of the compilation process for ease or for additional analysis. Many C++ programmers will use different tools, but all of the tools will generally follow this generalized process when they are involved in the production of a program.

The link below extends this discussion and provides a nice graphic to help. [1]:

<http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html>

Section 1.4: Function

A **function** is a unit of code that represents a sequence of statements.

Functions can accept **arguments** or values and **return** a single value (or not). To use a function, a **function call** is used on argument values and the use of the function call itself is replaced with its return value.

Every function has a **type signature** -- the types of its arguments and the type of its return type.

Functions are inspired by the concepts of the procedure and the mathematical function.

- Note: C++ functions are essentially procedures and do not follow the exact definition or rules of mathematical functions.

Functions are often meant to perform a specific task. and can be called from other parts of a program. A function must be declared and defined before it is called elsewhere in a program.

- 注意：常见的函数定义可能隐藏在其他包含的文件中（通常为了方便和在多个文件间重用）。这也是头文件的常见用途。

函数声明

函数声明声明了一个函数的存在，包括其名称和类型签名，供编译器使用。

语法如下：

```
int add2(int i); // 该函数的类型为 (int) -> (int)
```

在上面的例子中，`int add2(int i)` 函数向编译器声明了以下内容：

- 返回类型是int。
- 函数名是add2。
- 函数的参数个数是1个：
 - 第一个参数的类型是int。
 - 第一个参数在函数体内将以名称 i 引用。

参数名是可选的；函数的声明也可以是以下形式：

```
int add2(int); // 省略函数参数名也是允许的。
```

根据一一定义规则，具有某种类型签名的函数在整个对C++编译器可见的C++代码库中只能声明或定义一次。换句话说，具有特定类型签名的函数不能被重新定义——它们只能被定义一次。因此，以下代码不是有效的C++：

```
int add2(int i); // 编译器会注意到add2是一个函数，类型为(int) -> int
int add2(int j); // 由于add2已经有了(int) -> int的定义，编译器
                 // 会将此视为错误。
```

如果函数没有返回值，其返回类型写为void。如果函数不接受参数，参数列表应为空。

```
void do_something(); // 该函数不接受参数，也不返回任何值。
                     // 注意它仍然可以影响它能访问的变量。
```

函数调用

函数在声明后可以被调用。例如，下面的程序在main函数中调用了add2，传入的值是2：

```
#include <iostream>

int add2(int i); // add2的声明

// 注意：add2仍然缺少定义。
// 尽管它没有直接出现在代码中，
// add2 的定义可能是从另一个目标文件链接进来的。

int main()
{
    std::cout << add2(2) << "";
    // add2(2) 会在此处被计算，// 并打印结果。
}

return 0;
}
```

这里，`add2(2)` 是函数调用的语法。

- Note: popular function definitions may be hidden in other included files (often for convenience and reuse across many files). This is a common use of header files.

Function Declaration

A **function declaration** is declares the existence of a function with its name and type signature to the compiler. The syntax is as the following:

```
int add2(int i); // The function is of the type (int) -> (int)
```

In the example above, the `int add2(int i)` function declares the following to the compiler:

- The **return type** is `int`.
- The **name** of the function is `add2`.
- The **number of arguments** to the function is 1:
 - The first argument is of the type `int`.
 - The first argument will be referred to in the function's contents by the name `i`.

The argument name is optional; the declaration for the function could also be the following:

```
int add2(int); // Omitting the function arguments' name is also permitted.
```

Per the **one-definition rule**, a function with a certain type signature can only be declared or defined once in an entire C++ code base visible to the C++ compiler. In other words, functions with a specific type signature cannot be re-defined -- they must only be defined once. Thus, the following is not valid C++:

```
int add2(int i); // The compiler will note that add2 is a function (int) -> int
int add2(int j); // As add2 already has a definition of (int) -> int, the compiler
                 // will regard this as an error.
```

If a function returns nothing, its return type is written as `void`. If it takes no parameters, the parameter list should be empty.

```
void do_something(); // The function takes no parameters, and does not return anything.
                     // Note that it can still affect variables it has access to.
```

Function Call

A function can be called after it has been declared. For example, the following program calls add2 with the value of 2 within the function of main:

```
#include <iostream>

int add2(int i); // Declaration of add2

// Note: add2 is still missing a DEFINITION.
// Even though it doesn't appear directly in code,
// add2's definition may be LINKED in from another object file.

int main()
{
    std::cout << add2(2) << "\n";
    // add2(2) will be evaluated at this point,
    // and the result is printed.
}

return 0;
}
```

Here, `add2(2)` is the syntax for a function call.

函数定义

函数定义* 类似于声明，但它还包含了函数被调用时在函数体内执行的代码。

add2的函数定义示例可能是：

```
int add2(int i)      // 传入的数据 (int i) 在函数的花括号或“作用域”内将被称为 i
{
    int j = i + 2;  // 定义变量 j，其值为 i+2。
    return j;        // 返回，或者本质上是用 j 替代对函数 add2 的调用。
}
```

函数重载

你可以创建多个同名但参数不同的函数。

```
int add2(int i)      // 当调用带有一个参数的 add2() 时，将执行此定义中的代码。
{
    int j = i + 2;
    return j;
}

int add2(int i, int j) // 但是，当调用带有两个参数的 add2() 时，
{
    // 初始声明中的代码将被重载，
    int k = i + j + 2; // 并且将执行此声明中的代码
    return k;          // 代替。
}
```

这两个函数都使用相同的名称add2调用，但实际调用哪个函数直接取决于调用中参数的数量和类型。在大多数情况下，C++ 编译器可以计算出调用哪个函数。
在某些情况下，必须显式指定类型。

默认参数

函数参数的默认值只能在函数声明中指定。

```
int multiply(int a, int b = 7); // b 的默认值为 7。
int multiply(int a, int b)
{
    return a * b;              // 如果 multiply() 只传入一个参数，
}                                // 该值将与默认值 7 相乘。
```

在此示例中，multiply() 可以传入一个或两个参数。如果只给出一个参数，b 将具有默认值 7。默认参数必须放在函数的后面参数中。例如：

```
int multiply(int a = 10, int b = 20); // 这是合法的
int multiply(int a = 10, int b);     // 这是非法的，因为 int a 是前面的参数
```

特殊函数调用 - 运算符

C++ 中存在一些特殊的函数调用，其语法不同于 name_of_function(value1, value2, value3)。最常见的例子是运算符。

某些特殊字符序列会被编译器转换为函数调用，例如 !、+、-、*、% 以及 << 等等。这些特殊字符通常与非编程用途相关，或用于

Function Definition

A *function definition** is similar to a declaration, except it also contains the code that is executed when the function is called within its body.

An example of a function definition for add2 might be:

```
int add2(int i)      // Data that is passed into (int i) will be referred to by the name i
{
    int j = i + 2;  // Definition of a variable j as the value of i+2.
    return j;        // Returning or, in essence, substitution of j for a function call to
                    // add2.
}
```

Function Overloading

You can create multiple functions with the same name but different parameters.

```
int add2(int i)      // Code contained in this definition will be evaluated
{
    int j = i + 2;
    return j;
}

int add2(int i, int j) // However, when add2() is called with two parameters, the
{
    // code from the initial declaration will be overloaded,
    int k = i + j + 2; // and the code in this declaration will be evaluated
    return k;          // instead.
}
```

Both functions are called by the same name add2, but the actual function that is called depends directly on the amount and type of the parameters in the call. In most cases, the C++ compiler can compute which function to call. In some cases, the type must be explicitly stated.

Default Parameters

Default values for function parameters can only be specified in function declarations.

```
int multiply(int a, int b = 7); // b has default value of 7.
int multiply(int a, int b)
{
    return a * b;              // If multiply() is called with one parameter, the
}                                // value will be multiplied by the default, 7.
```

In this example, multiply() can be called with one or two parameters. If only one parameter is given, b will have default value of 7. Default arguments must be placed in the latter arguments of the function. For example:

```
int multiply(int a = 10, int b = 20); // This is legal
int multiply(int a = 10, int b);     // This is illegal since int a is in the former
```

Special Function Calls - Operators

There exist special function calls in C++ which have different syntax than name_of_function(value1, value2, value3)。The most common example is that of operators.

Certain special character sequences that will be reduced to function calls by the compiler, such as !, +, -, *, %, and << and many more. These special characters are normally associated with non-programming usage or are used for

美学（例如，+ 字符在 C++ 编程以及初等数学中通常被识别为加号符号）。

C++ 使用特殊语法处理这些字符序列；但本质上，每个运算符的出现都被简化为函数调用。例如，以下 C++ 表达式：

3+3

等价于以下函数调用：

operator+(3, 3)

所有运算符函数名都以 operator 开头。

虽然在 C++ 的直接前身 C 语言中，运算符函数名不能通过提供不同类型签名的额外定义赋予不同含义，但在 C++ 中这是有效的。在 C++ 中，将多个函数定义“隐藏”在一个唯一函数名下被称为 运算符重载，这是一种相对常见但非普遍的惯例。

第 1.5 节：函数原型和声明的可见性

在 C++ 中，代码必须在使用前声明或定义。例如，以下代码会产生编译时错误：

```
int main()
{
    foo(2); // 错误：调用了 foo，但尚未声明
}

void foo(int x) // 这个后面的定义在 main 中未知
{}
```

有两种方法可以解决这个问题：将foo()的定义或声明放在main()中使用之前。

这里有一个例子：

```
void foo(int x) {} // 先声明并定义foo函数体

int main()
{
    foo(2); // 可以：foo已完全定义，因此可以在这里调用。
}
```

但是也可以通过在使用之前只放置一个“原型”声明来“前向声明”函数，然后再定义函数体：

```
void foo(int); // foo的原型声明，main可见
                // 必须指定返回类型、名称和参数列表类型
int main()
{
    foo(2); // 可以：foo已知，即使函数体尚未定义也可以调用
}

void foo(int x) // 必须与原型匹配
{
    // 在这里定义foo的函数体
}
```

aesthetics (e.g. the + character is commonly recognized as the addition symbol both within C++ programming as well as in elementary math).

C++ handles these character sequences with a special syntax; but, in essence, each occurrence of an operator is reduced to a function call. For example, the following C++ expression:

3+3

is equivalent to the following function call:

operator+(3, 3)

All operator function names start with operator.

While in C++'s immediate predecessor, C, operator function names cannot be assigned different meanings by providing additional definitions with different type signatures, in C++, this is valid. "Hiding" additional function definitions under one unique function name is referred to as **operator overloading** in C++, and is a relatively common, but not universal, convention in C++.

Section 1.5: Visibility of function prototypes and declarations

In C++, code must be declared or defined before usage. For example, the following produces a compile time error:

```
int main()
{
    foo(2); // error: foo is called, but has not yet been declared
}

void foo(int x) // this later definition is not known in main
{}
```

There are two ways to resolve this: putting either the definition or declaration of foo() before its usage in main(). Here is one example:

```
void foo(int x) {} //Declare the foo function and body first

int main()
{
    foo(2); // OK: foo is completely defined beforehand, so it can be called here.
}
```

However it is also possible to "forward-declare" the function by putting only a "prototype" declaration before its usage and then defining the function body later:

```
void foo(int); // Prototype declaration of foo, seen by main
                // Must specify return type, name, and argument list types
int main()
{
    foo(2); // OK: foo is known, called even though its body is not yet defined
}

void foo(int x) //Must match the prototype
{
    // Define body of foo here
}
```

原型必须指定返回类型（void）、函数名（foo）和参数列表变量类型（int），但参数名称不是必需的。

将此集成到源文件组织中的一种常见方法是制作一个包含所有函数原型声明的头文件：

```
// foo.h  
void foo(int); // 函数原型声明
```

然后在其他地方提供完整的定义：

```
// foo.cpp --> foo.o  
#include "foo.h" // foo 的函数原型声明“隐藏”在这里  
void foo(int x) { } // foo 的函数体定义
```

然后，编译后，将对应的目标文件foo.o链接到使用它的编译目标文件中，在链接阶段，main.o：

```
// main.cpp --> main.o  
#include "foo.h" // foo 的函数原型声明“隐藏”在这里  
int main() { foo(2); } // 因为之前有函数原型声明，调用 foo 是有效的。  
// foo 的函数原型和函数体定义通过目标文件进行链接
```

当函数原型和调用存在，但函数体未定义时，会出现“未解析的外部符号”错误。这类错误更难解决，因为编译器直到最终链接阶段才会报告错误，且不知道代码中跳转到哪一行以显示错误。

第1.6节：预处理器

预处理器是编译器的重要组成部分。

它编辑源代码，剪切部分内容，修改其他内容，并添加新的内容。

在源文件中，我们可以包含预处理指令。这些指令告诉预处理器执行特定的操作。指令以#开头，且位于新的一行。例如：

```
#define ZERO 0
```

你可能首先会遇到的预处理指令是

```
#include <something>
```

指令。它的作用是将所有的something内容插入到指令所在的位置。Hello World程序以这一行开始

```
#include <iostream>
```

这一行添加了允许你使用标准输入和输出的函数和对象。

C语言也使用预处理器，但其头文件没有C++语言那么多，不过在C++中你可以使用所有的C头文件。

下一个重要的指令可能是

The prototype must specify the return type (`void`), the name of the function (`foo`), and the argument list variable types (`int`), but the names of the arguments are NOT required.

One common way to integrate this into the organization of source files is to make a header file containing all of the prototype declarations:

```
// foo.h  
void foo(int); // prototype declaration
```

and then provide the full definition elsewhere:

```
// foo.cpp --> foo.o  
#include "foo.h" // foo's prototype declaration is "hidden" in here  
void foo(int x) { } // foo's body definition
```

and then, once compiled, link the corresponding object file `foo.o` into the compiled object file where it is used in the linking phase, `main.o`:

```
// main.cpp --> main.o  
#include "foo.h" // foo's prototype declaration is "hidden" in here  
int main() { foo(2); } // foo is valid to call because its prototype declaration was beforehand.  
// the prototype and body definitions of foo are linked through the object files
```

An “unresolved external symbol” error occurs when the function *prototype* and *call* exist, but the function *body* is not defined. These can be trickier to resolve as the compiler won't report the error until the final linking stage, and it doesn't know which line to jump to in the code to show the error.

Section 1.6: Preprocessor

The preprocessor is an important part of the compiler.

It edits the source code, cutting some bits out, changing others, and adding other things.

In source files, we can include preprocessor directives. These directives tell the preprocessor to perform specific actions. A directive starts with a # on a new line. Example:

```
#define ZERO 0
```

The first preprocessor directive you will meet is probably the

```
#include <something>
```

directive. What it does is takes all of something and inserts it in your file where the directive was. The hello world program starts with the line

```
#include <iostream>
```

This line adds the functions and objects that let you use the standard input and output.

The C language, which also uses the preprocessor, does not have as many header files as the C++ language, but in C++ you can use all the C header files.

The next important directive is probably the

```
#define something something_else
```

指令。它告诉预处理器，在处理文件时，应将每个出现的something替换为something_else。它也可以实现类似函数的功能，但这可能属于高级C++范畴。

something_else不是必须的，但如果你将something定义为空，那么在预处理器指令之外，所有出现的something都会消失。

这实际上很有用，因为有#if、#else和#endif指令。这些指令的格式如下：

```
#if something==true  
//代码  
#else  
//更多代码  
#endif  
  
#ifdef thing_that_you_want_to_know_if_is_defined  
//code  
#endif
```

这些指令插入位于真位的代码，并删除假位的代码。这可以用于仅在某些操作系统上包含特定代码段，而无需重写整个代码。

```
#define something something_else
```

directive. This tells the preprocessor that as it goes along the file, it should replace every occurrence of something with something_else. It can also make things similar to functions, but that probably counts as advanced C++.

The something_else is not needed, but if you define something as nothing, then outside preprocessor directives, all occurrences of something will vanish.

This actually is useful, because of the #if,#else and #ifdef directives. The format for these would be the following:

```
#if something==true  
//code  
#else  
//more code  
#endif  
  
#ifdef thing_that_you_want_to_know_if_is_defined  
//code  
#endif
```

These directives insert the code that is in the true bit, and deletes the false bits. this can be used to have bits of code that are only included on certain operating systems, without having to rewrite the whole code.

第2章：字面量

传统上，字面量是表示常量的表达式，其类型和值从其拼写中显而易见。例如，42 是一个字面量，而 x 不是，因为必须查看其声明才能知道其类型，并且需要阅读之前的代码行才能知道其值。

然而，C++11 还添加了用户自定义字面量，它们在传统意义上不是字面量，但可以用作函数调用的简写。

第2.1节：this

在类的成员函数中，关键字 this 是指向调用该函数的类实例的指针。静态成员函数中不能使用 this。

```
结构体 S {  
    int x;  
S& operator=(const S& other) {  
    x = other.x;  
    // 返回被赋值对象的引用  
    return *this;  
}  
};
```

this的类型取决于成员函数的cv限定符：如果X::f是const，那么f中this的类型就是const X*，因此this不能用于修改const成员函数中的非静态数据成员。同样，this会继承其所在函数的volatile限定符。

版本 ≥ C++11

this也可以用于非静态数据成员的brace-or-equal-initializer中。

```
struct S;  
struct T {  
    T(const S* s);  
    // ...  
};  
struct S {  
    // ...  
    T t{this};  
};
```

this是一个右值，因此不能被赋值。

第2.2节：整数字面量

整数字面量是一种形式为的基本表达式

- 十进制字面量

它是一个非零十进制数字（1、2、3、4、5、6、7、8、9），后跟零个或多个十进制数字（0、1、2、3、4、5、6、7、8、9）

```
int d = 42;
```

- 八进制字面量

它是数字零（0），后跟零个或多个八进制数字（0、1、2、3、4、5、6、7）

Chapter 2: Literals

Traditionally, a literal is an expression denoting a constant whose type and value are evident from its spelling. For example, 42 is a literal, while x is not since one must see its declaration to know its type and read previous lines of code to know its value.

However, C++11 also added user-defined literals, which are not literals in the traditional sense but can be used as a shorthand for function calls.

Section 2.1: this

Within a member function of a class, the keyword this is a pointer to the instance of the class on which the function was called. this cannot be used in a static member function.

```
struct S {  
    int x;  
S& operator=(const S& other) {  
    x = other.x;  
    // return a reference to the object being assigned to  
    return *this;  
}  
};
```

The type of this depends on the cv-qualification of the member function: if X::f is const, then the type of this within f is const X*, so this cannot be used to modify non-static data members from within a const member function. Likewise, this inherits volatile qualification from the function it appears in.

Version ≥ C++11

this can also be used in a brace-or-equal-initializer for a non-static data member.

```
struct S;  
struct T {  
    T(const S* s);  
    // ...  
};  
struct S {  
    // ...  
    T t{this};  
};
```

this is an rvalue, so it cannot be assigned to.

Section 2.2: Integer literal

An integer literal is a primary expression of the form

- decimal-literal

It is a non-zero decimal digit (1, 2, 3, 4, 5, 6, 7, 8, 9), followed by zero or more decimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

```
int d = 42;
```

- octal-literal

It is the digit zero (0) followed by zero or more octal digits (0, 1, 2, 3, 4, 5, 6, 7)

```
int o = 052
```

- 十六进制字面量

它是字符序列0x或字符序列0X，后跟一个或多个十六进制数字（0、1、2、3、4、5、6、7、8、9、a、A、b、B、c、C、d、D、e、E、f、F）

```
int x = 0x2a; int X = 0X2A;
```

- 二进制字面量（自C++14起）

它是字符序列0b或字符序列0B，后跟一个或多个二进制数字（0, 1）

```
int b = 0b101010; // C++14
```

如果提供了整数后缀，可能包含以下一种或两种（如果两者都提供，顺序可以任意）：

- 无符号后缀（字符u或字符U）

```
unsigned int u_1 = 42u;
```

- 长整型后缀（字符l或字符L）或长长整型后缀（字符序列ll或字符序列LL）（自C++11起）

以下变量也被初始化为相同的值：

```
unsigned long long l1 = 18446744073709550592ull; // C++11
unsigned long long l2 = 18'446'744'073'709'550'5921lu; // C++14
unsigned long long l3 = 1844'6744'0737'0955'0592uLL; // C++14
unsigned long long l4 = 184467'440737'0'95505'92LLU; // C++14
```

注释

整数字面量中的字母不区分大小写：0xDEADBABEU和0XdeadBABEU表示相同的数字
(唯一例外是长长整型后缀，它是ll或LL，绝不使用lL或Ll)

没有负整数字面量。诸如 -1 之类的表达式是对字面量所表示的值应用一元负号运算符，这可能涉及隐式类型转换。

在 C99 之前的 C 语言（但不包括 C++）中，不带后缀且不能放入 long int 的十进制值允许具有类型 unsigned long int。

当用于 #if 或 #elif 的控制表达式中时，所有有符号整数常量都表现得像具有类型 std::intmax_t，所有无符号整数常量都表现得像具有类型 std::uintmax_t。

第 2.3 节：true

表示类型 bool 两个可能值之一的关键字。

```
bool ok = true;
if (!f()) {
    ok = false;
    goto end;
}
```

```
int o = 052
```

- hex-literal

It is the character sequence 0x or the character sequence 0X followed by one or more hexadecimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, F)

```
int x = 0x2a; int X = 0X2A;
```

- binary-literal (since C++14)

It is the character sequence 0b or the character sequence 0B followed by one or more binary digits (0, 1)

```
int b = 0b101010; // C++14
```

Integer-suffix, if provided, may contain one or both of the following (if both are provided, they may appear in any order):

- unsigned-suffix (the character u or the character U)

```
unsigned int u_1 = 42u;
```

- long-suffix (the character l or the character L) or the long-long-suffix (the character sequence ll or the character sequence LL) (since C++11)

The following variables are also initialized to the same value:

```
unsigned long long l1 = 18446744073709550592ull; // C++11
unsigned long long l2 = 18'446'744'073'709'550'5921lu; // C++14
unsigned long long l3 = 1844'6744'0737'0955'0592uLL; // C++14
unsigned long long l4 = 184467'440737'0'95505'92LLU; // C++14
```

Notes

Letters in the integer literals are case-insensitive: 0xDEADBABEU and 0XdeadBABEU represent the same number (one exception is the long-long-suffix, which is either ll or LL, never lL or Ll)

There are no negative integer literals. Expressions such as -1 apply the unary minus operator to the value represented by the literal, which may involve implicit type conversions.

In C prior to C99 (but not in C++), unsuffixed decimal values that do not fit in long int are allowed to have the type unsigned long int.

When used in a controlling expression of #if or #elif, all signed integer constants act as if they have type std::intmax_t and all unsigned integer constants act as if they have type std::uintmax_t.

Section 2.3: true

A keyword denoting one of the two possible values of type bool.

```
bool ok = true;
if (!f()) {
    ok = false;
    goto end;
}
```

第 2.4 节 : false

表示类型 `bool` 两个可能值之一的关键字。

```
bool ok = true;
if (!f()) {
    ok = false;
    goto end;
}
```

第 2.5 节 : nullptr

版本 \geq C++11

表示空指针常量的关键字。它可以转换为任何指针或指向成员的指针类型，产生该类型的空指针。

```
Widget* p = new Widget();
delete p;
p = nullptr; // 删除后将指针设为 null
```

注意，`nullptr` 本身不是指针。`nullptr` 的类型是一种基本类型，称为 `std::nullptr_t`。

```
void f(int* p);

template <class T>
void g(T* p);

void h(std::nullptr_t p);

int main() {
    f(nullptr); // 正确
    g(nullptr); // 错误
    h(nullptr); // 正确
}
```

Section 2.4: false

A keyword denoting one of the two possible values of type `bool`.

```
bool ok = true;
if (!f()) {
    ok = false;
    goto end;
}
```

Section 2.5: nullptr

Version \geq C++11

A keyword denoting a null pointer constant. It can be converted to any pointer or pointer-to-member type, yielding a null pointer of the resulting type.

```
Widget* p = new Widget();
delete p;
p = nullptr; // set the pointer to null after deletion
```

Note that `nullptr` is not itself a pointer. The type of `nullptr` is a fundamental type known as `std::nullptr_t`.

```
void f(int* p);

template <class T>
void g(T* p);

void h(std::nullptr_t p);

int main() {
    f(nullptr); // ok
    g(nullptr); // error
    h(nullptr); // ok
}
```

第3章：运算符优先级

第3.1节：逻辑 && 和 || 运算符：短路

&& 的优先级高于 ||，这意味着括号用于评估那些应当一起计算的部分。

C++ 在 && 和 || 中使用短路求值，以避免不必要的执行。

如果 || 左侧返回 true，则右侧不再需要计算。

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

using namespace std;

bool True(string id){
    cout << "True" << id << endl;
    return true;
}

bool False(string id){
    cout << "False" << id << endl;
    return false;
}

int main(){
    bool result;
    //让我们用 || 和 && 来评估三个布尔值，以说明运算符优先级
    //优先级并不意味着 && 会先被计算，而是指
    //括号会被添加的位置
    //示例 1
    result =
        False("A") || False("B") && False("C");
        // 等价于 False("A") || (False("B") && False("C"))
    //FalseA
    //FalseB
    //短路求值跳过 C"
    //A 是 false，所以必须计算 || 右边的表达式，
    //B 是 false，因此不必计算 C 就能知道结果是 false

    result =
        True("A") || False("B") && False("C");
        // 等价于 True("A") || (False("B") && False("C"))
    cout << result << " =====" << endl;
    //TrueA
    //短路求值跳过 B"
    //短路求值跳过 C"
    //A 是 true，所以不必计算
    //    || 右边的表达式就能知道结果是 true
    //如果 || 的优先级高于 &&，等价的计算将是：
    // (True("A") || False("B")) && False("C")
    //会打印什么
    //TrueA
    //短路求值跳过 B"
    //FalseC
    //因为括号的位置不同
    //被求值的部分也不同
    //这使得最终结果在此情况下为False，因为C为假
```

Chapter 3: operator precedence

Section 3.1: Logical && and || operators: short-circuit

&& has precedence over ||, this means that parentheses are placed to evaluate what would be evaluated together.

c++ uses short-circuit evaluation in && and || to not do unnecessary executions.

If the left hand side of || returns true the right hand side does not need to be evaluated anymore.

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

using namespace std;

bool True(string id){
    cout << "True" << id << endl;
    return true;
}

bool False(string id){
    cout << "False" << id << endl;
    return false;
}

int main(){
    bool result;
    //let's evaluate 3 booleans with || and && to illustrate operator precedence
    //precedence does not mean that && will be evaluated first but rather where
    //parentheses would be added
    //example 1
    result =
        False("A") || False("B") && False("C");
        // eq. False("A") || (False("B") && False("C"))
    //FalseA
    //FalseB
    //Short-circuit evaluation skip of C"
    //A is false so we have to evaluate the right of ||
    //B being false we do not have to evaluate C to know that the result is false

    result =
        True("A") || False("B") && False("C");
        // eq. True("A") || (False("B") && False("C"))
    cout << result << " =====" << endl;
    //TrueA
    //Short-circuit evaluation skip of B"
    //Short-circuit evaluation skip of C"
    //A is true so we do not have to evaluate
    //    the right of || to know that the result is true
    //If || had precedence over && the equivalent evaluation would be:
    // (True("A") || False("B")) && False("C")
    //What would print
    //TrueA
    //Short-circuit evaluation skip of B"
    //FalseC
    //Because the parentheses are placed differently
    //the parts that get evaluated are differently
    //which makes that the end result in this case would be False because C is false
```

}

第3.2节：一元运算符

一元运算符作用于调用它们的对象，且优先级较高。（参见备注）当以后缀形式使用时，操作仅在整个表达式

求值后发生，导致一些有趣的算术运算：

```
int a = 1;
++a;           // 结果: 2
a--;           // 结果: 1
int minusa=-a; // 结果: -1

bool b = true;
!b; // 结果: true

a=4;
int c = a++/2;    // 等同于: (a==4) 4 / 2  结果: 2 ('a' 后缀递增)
cout << a << endl; // 打印5!
int d = ++a/2;    // 等于: (a+1) == 6 / 2 结果: 3

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

int *ptr1 = &arr[0];    // 指向 arr[0], 值为 1
int *ptr2 = ptr1++;    // ptr2 指向 arr[0], 值仍为 1; ptr1 自增
std::cout << *ptr1++ << std::endl; // 输出 2

int e = arr[0]++;      // 接收 arr[0] 自增前的值
std::cout << e << std::endl; // 输出 1
std::cout << *ptr2 << std::endl; // 输出 arr[0], 现在为 2
```

}

Section 3.2: Unary Operators

Unary operators act on the object upon which they are called and have high precedence. (See Remarks)

When used postfix, the action occurs only after the entire operation is evaluated, leading to some interesting arithmetics:

```
int a = 1;
++a;           // result: 2
a--;           // result: 1
int minusa=-a; // result: -1

bool b = true;
!b; // result: true

a=4;
int c = a++/2;    // equal to: (a==4) 4 / 2  result: 2 ('a' incremented postfix)
cout << a << endl; // prints 5!
int d = ++a/2;    // equal to: (a+1) == 6 / 2 result: 3

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

int *ptr1 = &arr[0];    // points to arr[0] which is 1
int *ptr2 = ptr1++;    // ptr2 points to arr[0] which is still 1; ptr1 incremented
std::cout << *ptr1++ << std::endl; // prints 2

int e = arr[0]++;      // receives the value of arr[0] before it is incremented
std::cout << e << std::endl; // prints 1
std::cout << *ptr2 << std::endl; // prints arr[0] which is now 2
```

第3.3节：算术运算符

C++中的算术运算符与数学中的优先级相同：

乘法和除法具有左结合性（意味着它们从左到右计算），且优先级高于加法和减法，加法和减法也具有左结合性。

我们也可以使用括号()强制表达式的优先级，就像在普通数学中那样。

```
// 球壳体积 = 4 pi R^3 - 4 pi r^3
double vol = 4.0*pi*R*R*R/3.0 - 4.0*pi*r*r*r/3.0;

// 加法：

int a = 2+4/2;          // 等价于: 2+(4/2)    结果: 4
int b = (3+3)/2;        // 等价于: (3+3)/2   结果: 3

// 乘法

int c = 3+4/2*6;        // 等价于: 3+((4/2)*6)  结果: 15
int d = 3*(3+6)/9;      // 等价于: (3*(3+6))/9  结果: 3

// 除法和取模

int g = 3-3%1;          // 等同于: 3 % 1 = 0  3 - 0 = 3
int h = 3-(3%1);         // 等同于: 3 % 1 = 0  3 - 0 = 3
```

Section 3.3: Arithmetic operators

Arithmetic operators in C++ have the same precedence as they do in mathematics:

Multiplication and division have left associativity(meaning that they will be evaluated from left to right) and they have higher precedence than addition and subtraction, which also have left associativity.

We can also force the precedence of expression using parentheses (). Just the same way as you would do that in normal mathematics.

```
// volume of a spherical shell = 4 pi R^3 - 4 pi r^3
double vol = 4.0*pi*R*R*R/3.0 - 4.0*pi*r*r*r/3.0;

//Addition:

int a = 2+4/2;          // equal to: 2+(4/2)      result: 4
int b = (3+3)/2;        // equal to: (3+3)/2     result: 3

//With Multiplication

int c = 3+4/2*6;        // equal to: 3+((4/2)*6)  result: 15
int d = 3*(3+6)/9;      // equal to: (3*(3+6))/9  result: 3

//Division and Modulo

int g = 3-3%1;          // equal to: 3 % 1 = 0  3 - 0 = 3
int h = 3-(3%1);         // equal to: 3 % 1 = 0  3 - 0 = 3
```

```
int i = 3-3/1%3; // 等同于: 3 / 1 = 3 3 % 3 = 0 3 - 0 = 3  
int l = 3-(3/1)%3; // 等同于: 3 / 1 = 3 3 % 3 = 0 3 - 0 = 3  
int m = 3-(3/(1%3)); // 等同于: 1 % 3 = 1 3 / 1 = 3 3 - 3 = 0
```

第3.4节：逻辑与（AND）和或（OR）运算符

这些运算符在C++中具有通常的优先级：与（AND）优先于或（OR）。

```
// 你可以用外国驾照驾驶最多60天  
bool can_drive = has_domestic_license || has_foreign_license && num_days <= 60;
```

这段代码等价于以下代码：

```
// 你可以用外国驾照驾驶最多60天  
bool can_drive = has_domestic_license || (has_foreign_license && num_days <= 60);
```

添加括号不会改变行为，但确实使代码更易读。通过添加这些括号，消除了对作者意图的任何疑惑。

```
int i = 3-3/1%3; // equal to: 3 / 1 = 3 3 % 3 = 0 3 - 0 = 3  
int l = 3-(3/1)%3; // equal to: 3 / 1 = 3 3 % 3 = 0 3 - 0 = 3  
int m = 3-(3/(1%3)); // equal to: 1 % 3 = 1 3 / 1 = 3 3 - 3 = 0
```

Section 3.4: Logical AND and OR operators

These operators have the usual precedence in C++: AND before OR.

```
// You can drive with a foreign license for up to 60 days  
bool can_drive = has_domestic_license || has_foreign_license && num_days <= 60;
```

This code is equivalent to the following:

```
// You can drive with a foreign license for up to 60 days  
bool can_drive = has_domestic_license || (has_foreign_license && num_days <= 60);
```

Adding the parenthesis does not change the behavior, though, it does make it easier to read. By adding these parentheses, no confusion exist about the intent of the writer.

第4章：浮点数运算

第4.1节：浮点数很奇怪

几乎每个程序员都会犯的第一个错误是认为这段代码会按预期工作：

```
float total = 0;  
for(float a = 0; a != 2; a += 0.01f) {  
    total += a;  
}
```

初学者程序员认为这将会把范围内的每一个数字`0`、`0.01`、`0.02`、`0.03`、`...`、`1.97`、`1.98`、`1.99`相加，得到结果`199`——数学上正确的答案。

有两件事使得这个假设不成立：

- 程序按写法永远不会结束。 a 永远不会等于 2，循环永远不会终止。
 - 如果我们将循环逻辑改写为检查 $a < 2$ ，循环会终止，但总和最终会是与 199 不同的值。在符合 IEEE754 标准的机器上，它通常会加到大约 201。

发生这种情况的原因是 浮点数表示的是其赋值的近似值。

经典示例是以下计算：

```
double a = 0.1;
double b = 0.2;
double c = 0.3;
if(a + b == c)
    //在符合IEEE754标准的机器上，这行代码永远不会打印
    std::cout << "这台计算机是魔法！" << std::endl;
else
    std::cout << "考虑到所有情况，这台计算机相当正常。" << std::endl;
```

虽然我们程序员看到的是用十进制写的三个数字，但编译器（以及底层硬件）看到的是二进制数字。因为0.1、0.2和0.3需要被10完美除尽——这在十进制系统中很容易，但在二进制系统中是不可能的——这些数字必须以不精确的格式存储，类似于数字 $1/3$ 在十进制中必须以不精确的形式0.333333333333...存储。

Chapter 4: Floating Point Arithmetic

Section 4.1: Floating Point Numbers are Weird

The first mistake that nearly every single programmer makes is presuming that this code will work as intended:

```
float total = 0;  
for(float a = 0; a != 2; a += 0.01f) {  
    total += a;  
}
```

The novice programmer assumes that this will sum up every single number in the range $0, 0.01, 0.02, 0.03, \dots, 1.97, 1.98, 1.99$, to yield the result 199—the mathematically correct answer.

Two things happen that make this untrue:

1. The program as written never concludes. a never becomes equal to 2, and the loop never terminates.
 2. If we rewrite the loop logic to check $a < 2$ instead, the loop terminates, but the total ends up being something different from 199. On IEEE754-compliant machines, it will often sum up to about 201 instead.

The reason that this happens is that **Floating Point Numbers represent Approximations of their assigned values.**

The classical example is the following computation:

```
double a = 0.1;
double b = 0.2;
double c = 0.3;
if(a + b == c)
    //This never prints on IEEE754-compliant machines
    std::cout << "This Computer is Magic!" << std::endl;
else
    std::cout << "This Computer is pretty normal, all things considered." << std::endl;
```

Though what we the programmer see is three numbers written in base10, what the compiler (and the underlying hardware) see are binary numbers. Because 0.1 , 0.2 , and 0.3 require perfect division by 10—which is quite easy in a base-10 system, but impossible in a base-2 system—these numbers have to be stored in imprecise formats, similar to how the number $1/3$ has to be stored in the imprecise form $0.333333333333333\dots$ in base-10.

第5章：位运算符

第5.1节：| - 按位或

```
int a = 5;      // 0101b (0x05)
int b = 12;     // 1100b (0x0C)
int c = a | b; // 1101b (0x0D)

std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
```

输出

```
a = 5, b = 12, c = 13
```

为什么

按位或在位级别上操作，使用以下布尔真值表：

true或true=true
true或false=true
false或false=false

当 a (0101) 的二进制值和 b (1100) 的二进制值进行或操作时，我们得到的二进制值是 1101：

```
int a = 0 1 0 1
int b = 1 1 0 0 |
-----
int c = 1 1 0 1
```

按位或不会改变原始值，除非使用按位赋值复合运算符 |= 明确赋值：

```
int a = 5; // 0101b (0x05)
a |= 12; // a = 0101b | 1101b
```

第5.2节：^ - 按位异或（异或）

```
int a = 5;      // 0101b (0x05)
int b = 9;      // 1001b (0x09)
int c = a ^ b; // 1100b (0x0C)

std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
```

输出

```
a = 5, b = 9, c = 12
```

为什么

按位XOR（异或）在位级别上操作，使用以下布尔真值表：

true OR true = false
true OR false = true
false OR false = false

Chapter 5: Bit Operators

Section 5.1: | - bitwise OR

```
int a = 5;      // 0101b (0x05)
int b = 12;     // 1100b (0x0C)
int c = a | b; // 1101b (0x0D)

std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
```

Output

```
a = 5, b = 12, c = 13
```

Why

A bit wise OR operates on the bit level and uses the following Boolean truth table:

true OR true = true
true OR false = true
false OR false = false

When the binary value for a (0101) and the binary value for b (1100) are OR'ed together we get the binary value of 1101:

```
int a = 0 1 0 1
int b = 1 1 0 0 |
-----
int c = 1 1 0 1
```

The bit wise OR does not change the value of the original values unless specifically assigned to using the bit wise assignment compound operator |=:

```
int a = 5; // 0101b (0x05)
a |= 12; // a = 0101b | 1101b
```

Section 5.2: ^ - bitwise XOR (exclusive OR)

```
int a = 5;      // 0101b (0x05)
int b = 9;      // 1001b (0x09)
int c = a ^ b; // 1100b (0x0C)

std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
```

Output

```
a = 5, b = 9, c = 12
```

Why

A bit wise XOR (exclusive or) operates on the bit level and uses the following Boolean truth table:

true OR true = false
true OR false = true
false OR false = false

注意，使用XOR操作时，`true OR true = false`，而使用操作`true AND/OR true = true`，因此体现了XOR操作的排他性。

利用这一点，当 a (0101) 的二进制值和 b (1001) 的二进制值进行XOR操作时，我们得到二进制值1100：

```
int a = 0 1 0 1  
int b = 1 0 0 1 ^  
-----  
int c = 1 1 0 0
```

按位XOR不会改变原始值，除非使用按位赋值复合运算符`^=`进行赋值：

```
int a = 5; // 0101b (0x05)  
a ^= 9; // a = 0101b ^ 1001b
```

按位XOR可以用于多种方式，常用于加密和压缩的位掩码操作中。

注意：以下示例常被用作一个巧妙技巧的例子。但不应在生产代码中使用（有更好的方法 `std::swap()` 来实现相同的结果）。

你也可以利用异或（XOR）操作在不使用临时变量的情况下交换两个变量：

```
int a = 42;  
int b = 64;  
  
// 异或交换  
a ^= b;  
b ^= a;  
a ^= b;  
  
std::cout << "a = " << a << ", b = " << b << "";
```

要将此方法用于生产环境，需要添加检查以确保其可用。

```
void doXORSwap(int& a, int& b)  
{  
    // 需要添加一个检查，确保不会将同一个变量与自身交换。否则会将值置零。  
    if (&a != &b)  
    {  
        // 异或交换  
        b ^= a;  
        a ^= b;  
        }  
}
```

所以虽然它看起来像一个很巧妙的技巧，但在实际代码中并不实用。异或不是一种基本的逻辑运算，而是其他运算的组合： $a \oplus c = \sim(a \& c) \& (a \mid c)$

此外，在2015年及以后版本的编译器中，变量可以以二进制形式赋值：

```
int cn=0b0111;
```

Notice that with an XOR operation `true OR true = false` where as with operations `true AND/OR true = true`, hence the exclusive nature of the XOR operation.

Using this, when the binary value for a (0101) and the binary value for b (1001) are XOR'ed together we get the binary value of 1100:

```
int a = 0 1 0 1  
int b = 1 0 0 1 ^  
-----  
int c = 1 1 0 0
```

The bit wise XOR does not change the value of the original values unless specifically assigned to using the bit wise assignment compound operator `^=`:

```
int a = 5; // 0101b (0x05)  
a ^= 9; // a = 0101b ^ 1001b
```

The bit wise XOR can be utilized in many ways and is often utilized in bit mask operations for encryption and compression.

Note: The following example is often shown as an example of a nice trick. But should not be used in production code (there are better ways `std::swap()` to achieve the same result).

You can also utilize an XOR operation to swap two variables without a temporary:

```
int a = 42;  
int b = 64;  
  
// XOR swap  
a ^= b;  
b ^= a;  
a ^= b;  
  
std::cout << "a = " << a << ", b = " << b << "\n";
```

To productionalize this you need to add a check to make sure it can be used.

```
void doXORSwap(int& a, int& b)  
{  
    // Need to add a check to make sure you are not swapping the same  
    // variable with itself. Otherwise it will zero the value.  
    if (&a != &b)  
    {  
        // XOR swap  
        a ^= b;  
        b ^= a;  
        a ^= b;  
    }  
}
```

So though it looks like a nice trick in isolation it is not useful in real code. xor is not a base logical operation, but a combination of others: $a \oplus c = \sim(a \& c) \& (a \mid c)$

also in 2015+ compilers variables may be assigned as binary:

```
int cn=0b0111;
```

第5.3节：& - 按位与

```
int a = 6;      // 0110b (0x06)
int b = 10;     // 1010b (0x0A)
int c = a & b; // 0010b (0x02)

std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
```

输出

```
a = 6, b = 10, c = 2
```

为什么

按位与运算在位级别上操作，使用以下布尔真值表：

真 AND 真 = 真
真 AND 假 = 假
假 AND 假 = 假

当变量 a (0110) 的二进制值与变量 b (1010) 的二进制值进行与运算时，得到的二进制值是 0010：

```
int a = 0 1 1 0
int b = 1 0 1 0 &
-----
int c = 0 0 1 0
```

按位与运算不会改变原始值，除非使用按位赋值复合运算符&=进行赋值：

```
int a = 5; // 0101b (0x05)
a &= 10; // a = 0101b & 1010b
```

第5.4节：<< - 左移

```
int a = 1;      // 0001b
int b = a << 1; // 0010b

std::cout << "a = " << a << ", b = " << b << std::endl;
```

输出

```
a = 1, b = 2
```

为什么

左移位操作会将左侧值 (a) 的位向左移动右侧指定的次数 (1)，本质上是用0填充最低有效位，因此将值5 (二进制 0000 0101) 向左移动4次 (例如 $5 \ll 4$) 将得到值80 (二进制0101 0000)。你可能注意到，将值左移1次也相当于将该值乘以2，例如：

```
int a = 7;
while (a < 200) {
    std::cout << "a = " << a << std::endl;
    a <<= 1;
```

Section 5.3: & - bitwise AND

```
int a = 6;      // 0110b (0x06)
int b = 10;     // 1010b (0x0A)
int c = a & b; // 0010b (0x02)

std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
```

Output

```
a = 6, b = 10, c = 2
```

Why

A bit wise AND operates on the bit level and uses the following Boolean truth table:

TRUE AND TRUE = TRUE
TRUE AND FALSE = FALSE
FALSE AND FALSE = FALSE

When the binary value for a (0110) and the binary value for b (1010) are AND'ed together we get the binary value of 0010:

```
int a = 0 1 1 0
int b = 1 0 1 0 &
-----
int c = 0 0 1 0
```

The bit wise AND does not change the value of the original values unless specifically assigned to using the bit wise assignment compound operator &=:

```
int a = 5; // 0101b (0x05)
a &= 10; // a = 0101b & 1010b
```

Section 5.4: << - left shift

```
int a = 1;      // 0001b
int b = a << 1; // 0010b

std::cout << "a = " << a << ", b = " << b << std::endl;
```

Output

```
a = 1, b = 2
```

Why

The left bit wise shift will shift the bits of the left hand value (a) the number specified on the right (1), essentially padding the least significant bits with 0's, so shifting the value of 5 (binary 0000 0101) to the left 4 times (e.g. $5 \ll 4$) will yield the value of 80 (binary 0101 0000). You might note that shifting a value to the left 1 time is also the same as multiplying the value by 2, example:

```
int a = 7;
while (a < 200) {
    std::cout << "a = " << a << std::endl;
    a <<= 1;
```

```
}
```

```
a = 7;  
while (a < 200) {  
    std::cout << "a = " << a << std::endl;  
    a *= 2;  
}
```

但需要注意的是，左移操作会将所有位向左移动，包括符号位，例如：

```
int a = 2147483647; // 0111 1111 1111 1111 1111 1111 1111 1111  
int b = a << 1; // 1111 1111 1111 1111 1111 1111 1111 1110  
  
std::cout << "a = " << a << ", b = " << b << std::endl;
```

可能的输出：a = 2147483647, b = -2

虽然有些编译器会产生看似预期的结果，但应注意，如果对有符号数进行左移操作导致符号位被影响，结果是未定义的。如果你想左移的位数是负数或大于左侧类型所能容纳的位数，结果也是未定义的，例如：

```
int a = 1;  
int b = a << -1; // 未定义行为  
char c = a << 20; // 未定义行为
```

位运算左移不会改变原始值，除非使用位运算赋值复合运算符<<=明确赋值：

```
int a = 5; // 0101b  
a <<= 1; // a = a << 1;
```

第5.5节：>> - 右移

```
int a = 2; // 0010b  
int b = a >> 1; // 0001b  
  
std::cout << "a = " << a << ", b = " << b << std::endl;
```

输出

a = 2, b = 1

为什么

右位移操作会将左侧值 (a) 的位向右移动右侧指定的位数 (1)；需要注意的是，虽然右移操作是标准的，但对于有符号的负数右移时，位的处理是实现定义的，因此不能保证具有可移植性，例如：

```
int a = -2;  
int b = a >> 1; // b 的值将取决于编译器
```

如果你想要右移的位数是负数，行为也是未定义的，例如：

```
int a = 1;  
int b = a >> -1; // 未定义行为
```

```
}
```

```
a = 7;  
while (a < 200) {  
    std::cout << "a = " << a << std::endl;  
    a *= 2;  
}
```

But it should be noted that the left shift operation will shift *all* bits to the left, including the sign bit, example:

```
int a = 2147483647; // 0111 1111 1111 1111 1111 1111 1111 1111  
int b = a << 1; // 1111 1111 1111 1111 1111 1111 1111 1110  
  
std::cout << "a = " << a << ", b = " << b << std::endl;
```

Possible output: a = 2147483647, b = -2

While some compilers will yield results that seem expected, it should be noted that if you left shift a signed number so that the sign bit is affected, the result is **undefined**. It is also **undefined** if the number of bits you wish to shift by is a negative number or is larger than the number of bits the type on the left can hold, example:

```
int a = 1;  
int b = a << -1; // undefined behavior  
char c = a << 20; // undefined behavior
```

The bit wise left shift does not change the value of the original values unless specifically assigned to using the bit wise assignment compound operator <<=:

```
int a = 5; // 0101b  
a <<= 1; // a = a << 1;
```

Section 5.5: >> - right shift

```
int a = 2; // 0010b  
int b = a >> 1; // 0001b  
  
std::cout << "a = " << a << ", b = " << b << std::endl;
```

Output

a = 2, b = 1

Why

The right bit wise shift will shift the bits of the left hand value (a) the number specified on the right (1); it should be noted that while the operation of a right shift is standard, what happens to the bits of a right shift on a *signed* negative number is *implementation defined* and thus cannot be guaranteed to be portable, example:

```
int a = -2;  
int b = a >> 1; // the value of b will be depend on the compiler
```

It is also undefined if the number of bits you wish to shift by is a negative number, example:

```
int a = 1;  
int b = a >> -1; // undefined behavior
```

按位右移不会改变原始值，除非使用按位赋值复合运算符 `>>=` 进行赋值：

```
int a = 2; // 0010b  
a >>= 1; // a = a >> 1;
```

The bit wise right shift does not change the value of the original values unless specifically assigned to using the bit wise assignment compound operator `>>=`:

```
int a = 2; // 0010b  
a >>= 1; // a = a >> 1;
```

第6章：位操作

第6.1节：移除最右边的置位

C风格的位操作

```
template <typename T>
T rightmostSetBitRemoved(T n)
{
    // static_assert(std::is_integral<T>::value && !std::is_signed<T>::value, "type should be
    // unsigned"); // 对于 C++11 及更高版本
    return n & (n - 1);
}
```

解释

- 如果 n 是零，我们有 $0 \& 0xFF..FF$ ，结果是零
- 否则 n 可以写成 $0bxxxxxx10..00$ ，且 $n - 1$ 是 $0bxxxxxx011..11$ ，所以 $n \& (n - 1)$ 是 $0bxxxxxx000..00$ 。

第6.2节：设置所有位

C风格的位操作

```
x = -1; // -1 == 1111 1111 ... 1111b
```

(参见[此处](#)，了解为什么这有效且实际上是最佳方法。)

使用 std::bitset

```
std::bitset<10> x;
x.set(); // 将所有位设置为 '1'
```

第6.3节：切换位

C风格的位操作

可以使用异或运算符 (\wedge) 切换一个位。

```
// 位x将变为当前值的相反值
number ^= 1LL << x;
```

使用 std::bitset

```
std::bitset<4> num(std::string("0100"));
num.flip(2); // 现在num是0000
num.flip(0); // 现在num是0001
num.flip(); // 现在num是1110 (翻转所有位)
```

第6.4节：检查位

C风格的位操作

可以通过将数字右移 x 位，然后对其执行按位与运算

(a) 未获取该位的值：

```
(number >> x) & 1LL; // 如果'number'的第'x'位被设置，则为1，否则为0
```

右移操作可以实现为算术（有符号）右移或逻辑（无符号）右移。如果

Chapter 6: Bit Manipulation

Section 6.1: Remove rightmost set bit

C-style bit-manipulation

```
template <typename T>
T rightmostSetBitRemoved(T n)
{
    // static_assert(std::is_integral<T>::value && !std::is_signed<T>::value, "type should be
    // unsigned"); // For C++11 and later
    return n & (n - 1);
}
```

Explanation

- if n is zero, we have $0 \& 0xFF..FF$ which is zero
- else n can be written $0bxxxxxx10..00$ and $n - 1$ is $0bxxxxxx011..11$, so $n \& (n - 1)$ is $0bxxxxxx000..00$.

Section 6.2: Set all bits

C-style bit-manipulation

```
x = -1; // -1 == 1111 1111 ... 1111b
```

(See [here](#) for an explanation of why this works and is actually the best approach.)

Using std::bitset

```
std::bitset<10> x;
x.set(); // Sets all bits to '1'
```

Section 6.3: Toggling a bit

C-style bit-manipulation

A bit can be toggled using the XOR operator (\wedge).

```
// Bit x will be the opposite value of what it is currently
number ^= 1LL << x;
```

Using std::bitset

```
std::bitset<4> num(std::string("0100"));
num.flip(2); // num is now 0000
num.flip(0); // num is now 0001
num.flip(); // num is now 1110 (flips all bits)
```

Section 6.4: Checking a bit

C-style bit-manipulation

The value of the bit can be obtained by shifting the number to the right x times and then performing bitwise AND (&) on it:

```
(number >> x) & 1LL; // 1 if the 'x'th bit of 'number' is set, 0 otherwise
```

The right-shift operation may be implemented as either an arithmetic (signed) shift or a logical (unsigned) shift. If

表达式中的number (number >> x) 是有符号类型且为负值，则结果值由实现决定。

如果我们需要直接就地获取该位的值，可以改为左移掩码：

```
(number & (1LL << x)); // 如果 'number' 的第 'x' 位被设置，则为 (1 << x)，否则为 0
```

两者都可以用作条件判断，因为所有非零值都被视为真。

使用 std::bitset

```
std::bitset<4> num(std::string("0010"));
bool bit_val = num.test(1); // bit_val 的值被设置为 true；
```

第6.5节：计数已设置的位数

位串的计数在密码学和其他应用中经常需要，这个问题已经被广泛研究。

最简单的方法是对每一位进行一次迭代：

```
unsigned value = 1234;
unsigned bits = 0; // 累积 `n` 中已设置的位数

for (bits = 0; value; value >>= 1)
    bits += value & 1;
```

一个很好的技巧（基于移除最右边的置位）是：

```
unsigned bits = 0; // 累加 `n` 中被设置的位数总和

for (; value; ++bits)
    value &= value - 1;
```

它的迭代次数等于被设置的位数，因此当value预期只有少数非零位时效果很好。

该方法最早由彼得·韦格纳 (Peter Wegner) 提出 (发表于 CACM 3 / 322 - 1960)，并因出现在C Programming Language (《C程序设计语言》) 一书中而广为人知，作者是布赖恩·W·柯尼汉和丹尼斯·M·里奇。

这需要12次算术运算，其中一次是乘法：

```
unsigned popcount(std::uint64_t x)
{
    const std::uint64_t m1 = 0x5555555555555555; // 二进制：0101...
    const std::uint64_t m2 = 0x3333333333333333; // 二进制：00110011..
    const std::uint64_t m4 = 0x0f0f0f0f0f0f0f0f; // 二进制：0000111100001111

    x -= (x >> 1) & m1; // 将每2位的计数放入这2位中
    x = (x & m2) + ((x >> 2) & m2); // 将每4位的计数放入这4位中
    x = (x + (x >> 4)) & m4; // 将每8位的计数放入这8位中
    return (x * h01) >> 56; // x + (x<<8) + (x<<16) + (x<<24) + ... 的左8位
}
```

这种实现方式具有最佳的最坏情况表现（详见汉明重量）。

许多CPU有特定指令（如x86的popcnt），编译器也可能提供特定的（非标准）

number in the expression number >> x has a signed type and a negative value, the resulting value is implementation-defined.

If we need the value of that bit directly in-place, we could instead left shift the mask:

```
(number & (1LL << x)); // (1 << x) if the 'x'th bit of 'number' is set, 0 otherwise
```

Either can be used as a conditional, since all non-zero values are considered true.

Using std::bitset

```
std::bitset<4> num(std::string("0010"));
bool bit_val = num.test(1); // bit_val value is set to true;
```

Section 6.5: Counting bits set

The population count of a bitstring is often needed in cryptography and other applications and the problem has been widely studied.

The naive way requires one iteration per bit:

```
unsigned value = 1234;
unsigned bits = 0; // accumulates the total number of bits set in `n`

for (bits = 0; value; value >>= 1)
    bits += value & 1;
```

A nice trick (based on Remove rightmost set bit) is:

```
unsigned bits = 0; // accumulates the total number of bits set in `n`

for (; value; ++bits)
    value &= value - 1;
```

It goes through as many iterations as there are set bits, so it's good when value is expected to have few nonzero bits.

The method was first proposed by Peter Wegner (in CACM 3 / 322 - 1960) and it's well known since it appears in C Programming Language by Brian W. Kernighan and Dennis M. Ritchie.

This requires 12 arithmetic operations, one of which is a multiplication:

```
unsigned popcount(std::uint64_t x)
{
    const std::uint64_t m1 = 0x5555555555555555; // binary: 0101...
    const std::uint64_t m2 = 0x3333333333333333; // binary: 00110011..
    const std::uint64_t m4 = 0x0f0f0f0f0f0f0f0f; // binary: 0000111100001111

    x -= (x >> 1) & m1; // put count of each 2 bits into those 2 bits
    x = (x & m2) + ((x >> 2) & m2); // put count of each 4 bits into those 4 bits
    x = (x + (x >> 4)) & m4; // put count of each 8 bits into those 8 bits
    return (x * h01) >> 56; // left 8 bits of x + (x<<8) + (x<<16) + (x<<24) + ...
}
```

This kind of implementation has the best worst-case behavior (see Hamming weight for further details).

Many CPUs have a specific instruction (like x86's popcnt) and the compiler could offer a specific (**non standard**)

内置函数。例如，g++中有：

```
int __builtin_popcount (unsigned x);
```

第6.6节：检查整数是否为2的幂

$n \& (n - 1)$ 技巧（见移除最右边的1位）也可用于判断整数是否为2的幂：

```
bool power_of_2 = n && !(n & (n - 1));
```

注意，如果没有检查的第一部分 ($n \&&$)，0会被错误地认为是2的幂。

第6.7节：设置位

C风格的位操作

可以使用按位或运算符 ($|$) 来设置位。

```
// 位 x 将被设置  
number |= 1LL << x;
```

使用 std::bitset

set(x) 或 set(x,true) - 将位置 x 的位设置为 1。

```
std::bitset<5> num(std::string("01100"));  
num.set(0); // 现在 num 是 01101  
num.set(2); // num 仍然是 01101  
num.set(4,true); // 现在 num 是 11110
```

第6.8节：清除位

C风格的位操作

可以使用按位与运算符 ($&$) 来清除位。

```
// 位 x 将被清除  
number &= ~(1LL << x);
```

使用 std::bitset

reset(x) 或 set(x,false) - 清除位置 x 的位。

```
std::bitset<5> num(std::string("01100"));  
num.reset(2); // 现在 num 是 01000  
num.reset(0); // num 仍然是 01000  
num.set(3,false); // 现在 num 是 00000
```

第6.9节：将第 n 位更改为 x

C风格的位操作

```
// 如果 x 为 1，则设置第 n 位；如果 x 为 0，则清除第 n 位。  
number ^= (~x ^ number) & (1LL << n);
```

使用 std::bitset

set(n,val) - 将第 n 位设置为值 val。

built in function. E.g. with g++ there is:

```
int __builtin_popcount (unsigned x);
```

Section 6.6: Check if an integer is a power of 2

The $n \& (n - 1)$ trick (see Remove rightmost set bit) is also useful to determine if an integer is a power of 2:

```
bool power_of_2 = n && !(n & (n - 1));
```

Note that without the first part of the check ($n \&&$), 0 is incorrectly considered a power of 2.

Section 6.7: Setting a bit

C-style bit manipulation

A bit can be set using the bitwise OR operator ($|$).

```
// Bit x will be set  
number |= 1LL << x;
```

Using std::bitset

set(x) or set(x,true) - sets bit at position x to 1.

```
std::bitset<5> num(std::string("01100"));  
num.set(0); // num is now 01101  
num.set(2); // num is still 01101  
num.set(4,true); // num is now 11110
```

Section 6.8: Clearing a bit

C-style bit-manipulation

A bit can be cleared using the bitwise AND operator ($&$).

```
// Bit x will be cleared  
number &= ~(1LL << x);
```

Using std::bitset

reset(x) or set(x,false) - clears the bit at position x.

```
std::bitset<5> num(std::string("01100"));  
num.reset(2); // num is now 01000  
num.reset(0); // num is still 01000  
num.set(3,false); // num is now 00000
```

Section 6.9: Changing the nth bit to x

C-style bit-manipulation

```
// Bit n will be set if x is 1 and cleared if x is 0.  
number ^= (~x ^ number) & (1LL << n);
```

Using std::bitset

set(n,val) - sets bit n to the value val.

```
std::bitset<5> num(std::string("00100"));
num.set(0,true); // 现在 num 是 00101
num.set(2,false); // 现在 num 是 00001
```

第 6.10 节：位操作应用：小写字母转大写字母

位操作的一个应用是通过选择一个掩码（mask）和合适的位操作，将字母从小写转换为大写或反之。例如，字母 a 的二进制表示是 01(1)00001，而其大写对应字母的表示是 01(0)00001。它们仅在括号中的位不同。在这种情况下，将字母 a 从小写转换为大写基本上就是将括号中的位设置为 1。为此，我们执行以下操作：

```
*****
将小写字母转换为大写字母。
=====
a: 01100001
mask: 11011111 <- (0xDF) 11(0)11111
-----
a&mask: 01000001 <- 大写字母 A
*****/
```

将字母转换为大写字母的代码是

```
#include <cstdio>

int main()
{
    char op1 = 'a'; // 字母 "a" (即小写字母)
    int mask = 0xDF; // 选择合适的掩码

    printf("a (AND) mask = A");printf(
        "%c & 0xDF = %c", op1, op1 & mask);return 0;
}
```

结果是

```
$ g++ main.cpp -o test1
$ ./test1
a (AND) mask = A
a & 0xDF = A
```

```
std::bitset<5> num(std::string("00100"));
num.set(0,true); // num is now 00101
num.set(2,false); // num is now 00001
```

Section 6.10: Bit Manipulation Application: Small to Capital Letter

One of several applications of bit manipulation is converting a letter from small to capital or vice versa by choosing a **mask** and a proper **bit operation**. For example, the **a** letter has this binary representation **01(1)00001** while its capital counterpart has **01(0)00001**. They differ solely in the bit in parenthesis. In this case, converting the **a** letter from small to capital is basically setting the bit in parenthesis to one. To do so, we do the following:

```
*****
convert small letter to captial letter.
=====
a: 01100001
mask: 11011111 <- (0xDF) 11(0)11111
-----
a&mask: 01000001 <- A letter
*****/
```

The code for converting a letter to A letter is

```
#include <cstdio>

int main()
{
    char op1 = 'a'; // "a" letter (i.e. small case)
    int mask = 0xDF; // choosing a proper mask

    printf("a (AND) mask = A\n");
    printf("%c & 0xDF = %c\n", op1, op1 & mask);

    return 0;
}
```

The result is

```
$ g++ main.cpp -o test1
$ ./test1
a (AND) mask = A
a & 0xDF = A
```

第7章：位域

位域紧凑地打包C和C++结构以减少大小。这看起来很简单：指定成员的位数，编译器负责混合位。限制是无法获取位域成员的地址，因为它是混合存储的。`sizeof()` 也不允许使用。

位域的代价是访问速度较慢，因为必须检索内存并应用按位操作来提取或修改成员值。这些操作也会增加可执行文件的大小。

第7.1节：声明和使用

结构体 文件属性

```
{  
    无符号整数 只读: 1;  
    无符号整数 隐藏: 1;  
};
```

这里，这两个字段中的每一个将在内存中占用1位。通过变量名后面的`: 1`表达式指定。

位域的基本类型可以是任何整型（8位整型到64位整型）。建议使用`unsigned`类型，否则可能会出现意外情况。

如果需要更多位数，将“1”替换为所需的位数。例如：

结构体 日期

```
{  
    无符号整数 年份 : 13; // 2^13 = 8192, 足够表示"年份"很长时间  
    无符号整数 月份: 4; // 2^4 = 16, 足够表示1-12月的值。  
    无符号整数 日期: 5; // 32  
};
```

整个结构体只使用了22位，且在普通编译器设置下，`sizeof`该结构体将是4字节。

用法非常简单。只需声明变量，并像使用普通结构体一样使用它。

日期 d;

```
d.Year = 2016;  
d.Month = 7;  
d.Day = 22;  
  
std::cout << "Year:" << d.Year << std::endl <<  
    "Month:" << d.Month << std::endl <<  
    "Day:" << d.Day << std::endl;
```

Chapter 7: Bit fields

Bit fields tightly pack C and C++ structures to reduce size. This appears painless: specify the number of bits for members, and compiler does the work of co-mingling bits. The restriction is inability to take the address of a bit field member, since it is stored co-mingled. `sizeof()` is also disallowed.

The cost of bit fields is slower access, as memory must be retrieved and bitwise operations applied to extract or modify member values. These operations also add to executable size.

Section 7.1: Declaration and Usage

```
struct FileAttributes  
{  
    unsigned int ReadOnly: 1;  
    unsigned int Hidden: 1;  
};
```

Here, each of these two fields will occupy 1 bit in memory. It is specified by `: 1` expression after the variable names. Base type of bit field could be any integral type (8-bit int to 64-bit int). Using `unsigned` type is recommended, otherwise surprises may come.

If more bits are required, replace "1" with number of bits required. For example:

```
struct Date  
{  
    unsigned int Year : 13; // 2^13 = 8192, enough for "year" representation for long time  
    unsigned int Month: 4; // 2^4 = 16, enough to represent 1-12 month values.  
    unsigned int Day: 5; // 32  
};
```

The whole structure is using just 22 bits, and with normal compiler settings, `sizeof` this structure would be 4 bytes.

Usage is pretty simple. Just declare the variable, and use it like ordinary structure.

```
Date d;  
  
d.Year = 2016;  
d.Month = 7;  
d.Day = 22;  
  
std::cout << "Year:" << d.Year << std::endl <<  
    "Month:" << d.Month << std::endl <<  
    "Day:" << d.Day << std::endl;
```

第8章：数组

数组是放置在相邻内存位置的同类型元素。元素可以通过唯一标识符加上索引单独引用。

这允许你声明特定类型的多个变量值，并且可以单独访问它们，而无需为每个值声明一个变量。

第8.1节：数组初始化

数组只是特定类型变量的一块连续内存位置。数组的分配方式与普通变量相同，但在名称后附加方括号[]，里面包含数组内存可容纳的元素数量。

下面的数组示例使用类型int，变量名arrayOfInts，以及数组可容纳的元素数量[5]：

```
int arrayOfInts[5];
```

数组可以像这样同时声明和初始化

```
int arrayOfInts[5] = {10, 20, 30, 40, 50};
```

当通过列出所有成员来初始化数组时，方括号内不必包含元素数量。编译器会自动计算。在下面的例子中，数量是5：

```
int arrayOfInts[] = {10, 20, 30, 40, 50};
```

也可以只初始化前几个元素，同时分配更多空间。在这种情况下，必须在方括号中定义长度。下面的代码将分配一个长度为5的数组并部分初始化，编译器会将所有剩余元素初始化为该元素类型的默认值，这里是零。

```
int arrayOfInts[5] = {10, 20}; // 表示 10, 20, 0, 0, 0
```

其他基本数据类型的数组也可以用同样的方式初始化。

```
char arrayOfChars[5]; // 声明数组并分配内存，但不初始化
```

```
char arrayOfChars[5] = { 'a', 'b', 'c', 'd', 'e' } ; // 声明并初始化
```

```
double arrayOfDoubles[5] = {1.14159, 2.14159, 3.14159, 4.14159, 5.14159};
```

```
string arrayOfStrings[5] = { "C++", "is", "super", "duper", "great!" };
```

还需要注意的是，访问数组元素时，数组元素的索引（或位置）从0开始。

```
int array[5] = { 10/*元素编号0*/, 20/*元素编号1*/, 30, 40, 50/*元素编号4*/ };
std::cout << array[4]; //输出50
std::cout << array[0]; //输出10
```

Chapter 8: Arrays

Arrays are elements of the same type placed in adjoining memory locations. The elements can be individually referenced by a unique identifier with an added index.

This allows you to declare multiple variable values of a specific type and access them individually without needing to declare a variable for each value.

Section 8.1: Array initialization

An array is just a block of sequential memory locations for a specific type of variable. Arrays are allocated the same way as normal variables, but with square brackets appended to its name [] that contain the number of elements that fit into the array memory.

The following example of an array uses the typ int, the variable name arrayOfInts, and the number of elements [5] that the array has space for:

```
int arrayOfInts[5];
```

An array can be declared and initialized at the same time like this

```
int arrayOfInts[5] = {10, 20, 30, 40, 50};
```

When initializing an array by listing all of its members, it is not necessary to include the number of elements inside the square brackets. It will be automatically calculated by the compiler. In the following example, it's 5:

```
int arrayOfInts[] = {10, 20, 30, 40, 50};
```

It is also possible to initialize only the first elements while allocating more space. In this case, defining the length in brackets is mandatory. The following will allocate an array of length 5 with partial initialization, the compiler initializes all remaining elements with the standard value of the element type, in this case zero.

```
int arrayOfInts[5] = {10, 20}; // means 10, 20, 0, 0, 0
```

Arrays of other basic data types may be initialized in the same way.

```
char arrayOfChars[5]; // declare the array and allocate the memory, don't initialize
```

```
char arrayOfChars[5] = { 'a', 'b', 'c', 'd', 'e' } ; //declare and initialize
```

```
double arrayOfDoubles[5] = {1.14159, 2.14159, 3.14159, 4.14159, 5.14159};
```

```
string arrayOfStrings[5] = { "C++", "is", "super", "duper", "great!" };
```

It is also important to take note that when accessing array elements, the array's element index(or position) starts from 0.

```
int array[5] = { 10/*Element no.0*/, 20/*Element no.1*/, 30, 40, 50/*Element no.4*/ };
std::cout << array[4]; //outputs 50
std::cout << array[0]; //outputs 10
```

第8.2节：固定大小的原始数组矩阵（即二维原始数组）

```
// 固定大小的原始数组矩阵（即二维原始数组）。
#include <iostream>
#include <iomanip>
using namespace std;

auto main() -> int
{
    int const n_rows = 3;
    int const n_cols = 7;
    int const m[n_rows][n_cols] =           // 一个原始数组矩阵。
    {
        { 1, 2, 3, 4, 5, 6, 7 },
        { 8, 9, 10, 11, 12, 13, 14 },
        { 15, 16, 17, 18, 19, 20, 21 }
    };

    for( int y = 0; y < n_rows; ++y )
    {
        for( int x = 0; x < n_cols; ++x )
        {
            cout << setw( 4 ) << m[y][x];      // 注意：不要使用 m[y,x] !
        }
        cout << "\n";
    }
}
```

输出：

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
```

C++ 不支持多维数组的特殊索引语法。相反，这样的数组被视为一个数组的数组（可能还有数组的数组，依此类推），每一级都使用普通的单索引符号 [i]。在上面的例子中，m[y] 指的是 m 的第 y 行，其中 y 是从零开始的索引。然后可以对该行进行索引，例如 m[y][x]，表示第 y 行的第 x 个元素——即第 x 列。

也就是说，最后一个索引变化最快，在声明中该索引的范围（这里是每行的列数）是最后指定的、最“内层”的大小。

由于 C++ 除了动态分配外，没有内置对动态大小数组的支持，动态大小矩阵通常实现为一个类。这样，原始数组矩阵的索引符号 m[y][x] 会有一定代价，要么暴露实现细节（例如转置矩阵的视图几乎不可能实现），要么通过从 operator[] 返回代理对象来增加一些开销和轻微的不便。因此，这种抽象的索引符号通常会不同，无论是外观还是索引顺序，例如 m(x,y) 或 m.at(x,y) 或 m.item(x,y)。

第 8.3 节：动态大小的原始数组

```
// 原始动态大小数组示例。通常建议使用 std::vector。
#include <algorithm>           // std::sort
#include <iostream>
using namespace std;

auto int_from( istream& in ) -> int { int x; in >> x; return x; }
```

Section 8.2: A fixed size raw array matrix (that is, a 2D raw array)

```
// A fixed size raw array matrix (that is, a 2D raw array).
#include <iostream>
#include <iomanip>
using namespace std;

auto main() -> int
{
    int const n_rows = 3;
    int const n_cols = 7;
    int const m[n_rows][n_cols] =           // A raw array matrix.
    {
        { 1, 2, 3, 4, 5, 6, 7 },
        { 8, 9, 10, 11, 12, 13, 14 },
        { 15, 16, 17, 18, 19, 20, 21 }
    };

    for( int y = 0; y < n_rows; ++y )
    {
        for( int x = 0; x < n_cols; ++x )
        {
            cout << setw( 4 ) << m[y][x];      // Note: do NOT use m[y,x]!
        }
        cout << '\n';
    }
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
```

C++ doesn't support special syntax for indexing a multi-dimensional array. Instead such an array is viewed as an array of arrays (possibly of arrays, and so on), and the ordinary single index notation [i] is used for each level. In the example above m[y] refers to row y of m, where y is a zero-based index. Then this row can be indexed in turn, e.g. m[y][x], which refers to the xth item – or column – of row y.

I.e. the last index varies fastest, and in the declaration the range of this index, which here is the number of columns per row, is the last and “innermost” size specified.

Since C++ doesn't provide built-in support for dynamic size arrays, other than dynamic allocation, a dynamic size matrix is often implemented as a class. Then the raw array matrix indexing notation m[y][x] has some cost, either by exposing the implementation (so that e.g. a view of a transposed matrix becomes practically impossible) or by adding some overhead and slight inconvenience when it's done by returning a proxy object from operator[]. And so the indexing notation for such an abstraction can and will usually be different, both in look-and-feel and in the order of indices, e.g. m(x,y) or m.at(x,y) or m.item(x,y).

Section 8.3: Dynamically sized raw array

```
// Example of raw dynamic size array. It's generally better to use std::vector.
#include <algorithm>           // std::sort
#include <iostream>
using namespace std;

auto int_from( istream& in ) -> int { int x; in >> x; return x; }
```

```

auto main()
-> int
{
    cout << "对您提供的 n 个整数进行排序。\\n";
    cout << "n? ";
    int const n = int_from( cin );
    int* a = new int[n]; // ← 分配 n 个元素的数组。

    for( int i = 1; i <= n; ++i )
    {
        cout << "第 #" << i << " 个数字, 请输入: ";
        a[i-1] = int_from( cin );
    }

    sort( a, a + n );
    for( int i = 0; i < n; ++i ) { cout << a[i] << ' '; }
    cout << '\\n';

    delete[] a;
}

```

一个声明数组T a[n];的程序，其中n是在运行时确定的，可以使用支持C99可变长度数组（VLA）作为语言扩展的某些编译器进行编译。但标准C++不支持VLA。

此示例展示了如何通过new[]表达式手动分配动态大小的数组，

```
int* a = new int[n]; // ← 分配 n 个元素的数组。
```

...然后使用它，最后通过delete[]表达式释放它：

```
delete[] a;
```

这里分配的数组具有不确定的值，但只需添加一个空括号()，就可以将其初始化为零，方法如下：new int[n]()。更一般地，对于任意项类型，这执行的是值初始化。

作为调用层次结构中某个函数的一部分，这段代码不是异常安全的，因为在发生异常之前
delete[] 表达式（以及 new[] 之后）会导致内存泄漏。解决该问题的一种方法是通过例如 std::unique_ptr 智能指针来自动清理。但通常更好的方法是直接使用 std::vector：这正是 std::vector 存在的目的。

第8.4节：数组大小：编译时类型安全

```

#include // size_t, ptrdiff_t

//----- 机械部分：

using Size = ptrdiff_t;

template< class Item, size_t n >
constexpr auto n_items( Item (&)[n] ) noexcept
-> Size
{ return n; }

//----- 用法：

#include
using namespace std;
auto main()

```

```

auto main()
-> int
{
    cout << "Sorting n integers provided by you.\\n";
    cout << "n? ";
    int const n = int_from( cin );
    int* a = new int[n]; // ← Allocation of array of n items.

    for( int i = 1; i <= n; ++i )
    {
        cout << "The #" << i << " number, please: ";
        a[i-1] = int_from( cin );
    }

    sort( a, a + n );
    for( int i = 0; i < n; ++i ) { cout << a[i] << ' '; }
    cout << '\\n';

    delete[] a;
}

```

A program that declares an array T a[n] ; where n is determined at run-time, can compile with certain compilers that support C99 *variadic length arrays* (VLAs) as a language extension. But VLAs are not supported by standard C++. This example shows how to manually allocate a dynamic size array via a new[]-expression,

```
int* a = new int[n]; // ← Allocation of array of n items.
```

... then use it, and finally deallocate it via a delete[]-expression:

```
delete[] a;
```

The array allocated here has indeterminate values, but it can be zero-initialized by just adding an empty parenthesis () , like this: new int[n](). More generally, for arbitrary item type, this performs a *value-initialization*.

As part of a function down in a call hierarchy this code would not be exception safe, since an exception before the delete[] expression (and after the new[]) would cause a memory leak. One way to address that issue is to automate the cleanup via e.g. a std::unique_ptr smart pointer. But a generally better way to address it is to just use a std::vector: that's what std::vector is there for.

Section 8.4: Array size: type safe at compile time

```

#include // size_t, ptrdiff_t

//----- Machinery:

using Size = ptrdiff_t;

template< class Item, size_t n >
constexpr auto n_items( Item (&)[n] ) noexcept
-> Size
{ return n; }

//----- Usage:

#include
using namespace std;
auto main()

```

```

-> int
{
int const a[] = {3, 1, 4, 1, 5, 9, 2, 6, 5, 4};
Size const n = n_items(a);
int b[n]; // 与 a 大小相同的数组。

(void) b;
cout <>

```

用于获取数组大小的C语言习惯用法, `sizeof(a)/sizeof(a[0])`, 会接受指针作为参数, 通常会产生错误的结果。

对于C++11

使用C++11你可以这样做:

```
std::extent<decltype(MyArray)>::value;
```

示例:

```

char MyArray[] = { 'X','o','c','e' };const auto n
= std::extent<decltype(MyArray)>::value;std::cout << n << ""; //
输出4

```

直到C++17 (本文撰写时即将发布), C++没有内置的核心语言或标准库工具来获取数组大小, 但可以通过将数组按引用传递给函数模板来实现, 如上所示。重要但细节的点是: 模板的大小参数是一个`size_t`, 这与有符号的

`Size`函数返回类型不一致, 这是为了兼容g++编译器, 该编译器有时坚持使用 `size_t`进行模板匹配。

在C++17及以后版本, 可以改用`std::size`, 该函数对数组进行了特化。

第8.5节 : 使用

`std::vector`扩展动态大小数组

```

// std::vector 作为可扩展动态大小数组的示例。
#include <algorithm> // std::sort
#include <iostream>
#include <vector> // std::vector
using namespace std;

int int_from( std::istream& in ) { int x = 0; in >> x; return x; }

int main()
{
    cout << "正在对您提供的整数进行排序。";cout << "您可以
通过 Windows 中的 F6 或 Unix 系统中的 Ctrl+D 来表示文件结束符 (EOF) 。"; vector<int>
a; // ← 默认大小为零。

    while( cin )
    {
        cout << "请输入一个数字, 或表示文件结束符 : ";
        int const x = int_from( cin );
        if( !cin.fail() ) { a.push_back( x ); } // 根据需要扩展。
    }

    sort( a.begin(), a.end() );
}

```

```

-> int
{
int const a[] = {3, 1, 4, 1, 5, 9, 2, 6, 5, 4};
Size const n = n_items(a);
int b[n]; // An array of the same size as a.

(void) b;
cout <>

```

The C idiom for array size, `sizeof(a)/sizeof(a[0])`, will accept a pointer as argument and will then generally yield an incorrect result.

For C++11

using C++11 you can do:

```
std::extent<decltype(MyArray)>::value;
```

Example:

```

char MyArray[] = { 'X','o','c','e' };
const auto n = std::extent<decltype(MyArray)>::value;
std::cout << n << "\n"; // Prints 4

```

Up till C++17 (forthcoming as of this writing) C++ had no built-in core language or standard library utility to obtain the size of an array, but this can be implemented by passing the array *by reference* to a function template, as shown above. Fine but important point: the template size parameter is a `size_t`, somewhat inconsistent with the signed `Size` function result type, in order to accommodate the g++ compiler which sometimes insists on `size_t` for template matching.

With C++17 and later one may instead use `std::size`, which is specialized for arrays.

Section 8.5: Expanding dynamic size array by using `std::vector`

```

// Example of std::vector as an expanding dynamic size array.
#include <algorithm> // std::sort
#include <iostream>
#include <vector> // std::vector
using namespace std;

int int_from( std::istream& in ) { int x = 0; in >> x; return x; }

int main()
{
    cout << "Sorting integers provided by you.\n";
    cout << "You can indicate EOF via F6 in Windows or Ctrl+D in Unix-land.\n";
    vector<int> a; // ← Zero size by default.

    while( cin )
    {
        cout << "One number, please, or indicate EOF: ";
        int const x = int_from( cin );
        if( !cin.fail() ) { a.push_back( x ); } // Expands as necessary.
    }

    sort( a.begin(), a.end() );
}

```

```

int const n = a.size();
for( int i = 0; i < n; ++i ) { cout << a[i] << ' '; }cout << ";
}

```

`std::vector` 是一个标准库类模板，提供了可变大小数组的概念。它负责所有内存管理，且缓冲区是连续的，因此可以将指向缓冲区的指针（例如 `&v[0]` 或 `v.data()`）传递给需要原始数组的 API 函数。通过例如 `push_back` 成员函数追加元素，`vector` 甚至可以在运行时扩展。

包含向量扩展时涉及的复制或移动操作的 n `push_back` 操作序列的复杂度是摊销 $O(n)$ 。“摊销”：平均而言。

内部通常通过向量在需要更大缓冲区时将其缓冲区大小（容量）翻倍来实现。例如，对于初始大小为1的缓冲区，并在需要时反复翻倍，对于 $n=17$ `push_back` 调用，这涉及 $1 + 2 + 4 + 8 + 16 = 31$ 次复制操作，少于 $2 \times n = 34$ 。更一般地，这个序列的总和不会超过 $2 \times n$ 。

与动态大小的原始数组示例相比，这种基于 `vector` 的代码不要求用户预先提供（和知道）元素数量。相反，向量会根据用户指定的每个新元素值按需扩展。

第8.6节：使用 `std::vector` 作为存储的动态大小矩阵

遗憾的是，截至 C++14，C++ 标准库中没有动态大小矩阵类。然而，支持动态大小的矩阵类可以从多个第三方库获得，包括 Boost 矩阵库（Boost 库中的子库）。

如果你不想依赖 Boost 或其他库，那么 C++ 中一个简易的动态大小矩阵实现就是

```
vector<vector<int>> m( 3, vector<int>( 7 ) );
```

... 其中 `vector` 是 `std::vector`。这里通过复制一个行向量 n 次来创建矩阵， n 是行数，这里为3。它的优点是提供了与固定大小原始数组矩阵相同的 `m[y][x]` 索引表示法，但效率较低，因为每行都涉及动态分配，而且有点不安全，因为可能会无意中调整某一行的大小。

更安全且高效的方法是使用单个向量作为矩阵的 `storage`，并将客户端代码的 (x, y) 映射到该向量中的相应索引：

```

// 使用 std::vector 作为存储的动态大小矩阵。

//----- 机械：
#include      // std::copy
#include      // assert
#include // std::initializer_list
#include      // std::vector
#include      // ptrdiff_t

namespace my {
using Size = ptrdiff_t;
using std::initializer_list;
using std::vector;

```

```

int const n = a.size();
for( int i = 0; i < n; ++i ) { cout << a[i] << ' '; }
cout << '\n';
}

```

`std::vector` 是一个标准库类模板，提供了可变大小数组的概念。它负责所有内存管理，且缓冲区是连续的，因此可以将指向缓冲区的指针（例如 `&v[0]` 或 `v.data()`）传递给需要原始数组的 API 函数。通过例如 `push_back` 成员函数追加元素，`vector` 甚至可以在运行时扩展。

The complexity of the sequence of n `push_back` operations, including the copying or moving involved in the vector expansions, is amortized $O(n)$ 。“Amortized”：on average.

Internally this is usually achieved by the vector *doubling* its buffer size, its capacity, when a larger buffer is needed. E.g. for a buffer starting out as size 1, and being repeatedly doubled as needed for $n=17$ `push_back` calls, this involves $1 + 2 + 4 + 8 + 16 = 31$ copy operations, which is less than $2 \times n = 34$. And more generally the sum of this sequence can't exceed $2 \times n$.

Compared to the dynamic size raw array example, this vector-based code does not require the user to supply (and know) the number of items up front. Instead the vector is just expanded as necessary, for each new item value specified by the user.

Section 8.6: A dynamic size matrix using `std::vector` for storage

Unfortunately as of C++14 there's no dynamic size matrix class in the C++ standard library. Matrix classes that support dynamic size are however available from a number of 3rd party libraries, including the Boost Matrix library (a sub-library within the Boost library).

If you don't want a dependency on Boost or some other library, then one poor man's dynamic size matrix in C++ is just like

```
vector<vector<int>> m( 3, vector<int>( 7 ) );
```

... where `vector` is `std::vector`. The matrix is here created by copying a row vector n times where n is the number of rows, here 3. It has the advantage of providing the same `m[y][x]` indexing notation as for a fixed size raw array matrix, but it's a bit inefficient because it involves a dynamic allocation for each row, and it's a bit unsafe because it's possible to inadvertently resize a row.

A more safe and efficient approach is to use a single vector as `storage` for the matrix, and map the client code's (x, y) to a corresponding index in that vector:

```

// A dynamic size matrix using std::vector for storage.

//----- Machinery:
#include      // std::copy
#include      // assert
#include // std::initializer_list
#include      // std::vector
#include      // ptrdiff_t

namespace my {
using Size = ptrdiff_t;
using std::initializer_list;
using std::vector;

```

```

template< class Item >
class Matrix
{
private:
vector< Item > items_;
Size n_cols_;

auto index_for( Size const x, Size const y ) const
-> Size
{ return y*n_cols_ + x; }

public:
auto n_rows() const -> Size { return items_.size()/n_cols_; }
auto n_cols() const -> Size { return n_cols_; }

auto item( Size const x, Size const y )
-> Item&
{ return items_[index_for(x, y)]; }

auto item( Size const x, Size const y ) const
-> Item const&
{ return items_[index_for(x, y)]; }

Matrix(): n_cols_( 0 ) {}

Matrix( Size const n_cols, Size const n_rows )
: items_( n_cols*n_rows )
, n_cols_( n_cols )
{}

Matrix( initializer_list< initializer_list > const& values )
: items_()
, n_cols_( values.size() == 0? 0 : values.begin()->size() )
{
for( auto const& row : values )
{
assert( Size( row.size() ) == n_cols_ );
items_.insert( items_.end(), row.begin(), row.end() );
}
}
};

} // 命名空间 my

//----- 用法：
using my::Matrix;

auto some_matrix()
-> Matrix
{
return
{
{ 1, 2, 3, 4, 5, 6, 7 },
{ 8, 9, 10, 11, 12, 13, 14 },
{ 15, 16, 17, 18, 19, 20, 21 }
};

#include
#include
using namespace std;
auto main() -> int
{
Matrix const m = some_matrix();
assert( m.n_cols() == 7 );
}

```

```

template< class Item >
class Matrix
{
private:
vector< Item > items_;
Size n_cols_;

auto index_for( Size const x, Size const y ) const
-> Size
{ return y*n_cols_ + x; }

public:
auto n_rows() const -> Size { return items_.size()/n_cols_; }
auto n_cols() const -> Size { return n_cols_; }

auto item( Size const x, Size const y )
-> Item&
{ return items_[index_for(x, y)]; }

auto item( Size const x, Size const y ) const
-> Item const&
{ return items_[index_for(x, y)]; }

Matrix(): n_cols_( 0 ) {}

Matrix( Size const n_cols, Size const n_rows )
: items_( n_cols*n_rows )
, n_cols_( n_cols )
{}

Matrix( initializer_list< initializer_list > const& values )
: items_()
, n_cols_( values.size() == 0? 0 : values.begin()->size() )
{
for( auto const& row : values )
{
assert( Size( row.size() ) == n_cols_ );
items_.insert( items_.end(), row.begin(), row.end() );
}
}
};

} // namespace my

//----- Usage:
using my::Matrix;

auto some_matrix()
-> Matrix
{
return
{
{ 1, 2, 3, 4, 5, 6, 7 },
{ 8, 9, 10, 11, 12, 13, 14 },
{ 15, 16, 17, 18, 19, 20, 21 }
};

#include
#include
using namespace std;
auto main() -> int
{
Matrix const m = some_matrix();
assert( m.n_cols() == 7 );
}

```

```
assert( m.n_rows() == 3 );
for( int y = 0, y_end = m.n_rows(); y < y_end; ++y )
{
for( int x = 0, x_end = m.n_cols(); x < x_end; ++x )
{
cout << 注意：不是 `m[y][x]` !
}
cout <}
```

输出：

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
```

上述代码不是工业级的：它旨在展示基本原理，并满足学习C++的学生的需求。

例如，可以定义operator()重载来简化索引表示法。

```
assert( m.n_rows() == 3 );
for( int y = 0, y_end = m.n_rows(); y < y_end; ++y )
{
for( int x = 0, x_end = m.n_cols(); x < x_end; ++x )
{
cout << Note: not `m[y][x]` !
}
cout <}
```

Output:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
```

The above code is not industrial grade: it's designed to show the basic principles, and serve the needs of students learning C++.

For example, one may define operator() overloads to simplify the indexing notation.

第9章：迭代器

第9.1节：概述

迭代器是位置

迭代器是一种在元素序列中导航和操作的手段，是指针的广义扩展。概念上重要的是要记住，迭代器是位置，而不是元素。例如，考虑以下序列：

A B C

该序列包含三个元素和四个位置

```
+---+---+---+
| A | B | C |   |
+---+---+---+
```

元素是序列中的事物。位置是序列中可以进行有意义操作的地方。例如，插入操作是插入到一个位置，在元素A之前或之后，而不是插入到元素中。即使是删除一个元素（擦除（A））也是先找到它的位置，然后再删除它。

从迭代器到值

要从位置转换为值，需要对迭代器进行解引用：

```
auto my_iterator = my_vector.begin(); // 位置
auto my_value = *my_iterator; // 值
```

可以将迭代器看作是对序列中它所指向的值进行解引用。这对于理解为什么绝不应该对序列中的end()迭代器进行解引用特别有用：

```
+---+---+---+
| A | B | C |   |
+---+---+---+
↑           ↑
|           +- 此处的迭代器没有值。不要解引用它！
+----- 此处的迭代器解引用得到值A。
```

在C++标准库中所有的序列和容器中，begin()将返回指向第一个位置的迭代器，end()将返回指向最后一个位置之后的迭代器（不是最后一个位置！）。因此，这些迭代器在算法中的名称通常标记为first和last：

```
+---+---+---+
| A | B | C |   |
+---+---+---+
↑           ↑
|           |
+- 首个     +- 最后
```

也可以获得对任意序列的迭代器，因为即使是空序列也至少包含一个

Chapter 9: Iterators

Section 9.1: Overview

Iterators are Positions

Iterators are a means of navigating and operating on a sequence of elements and are a generalized extension of pointers. Conceptually it is important to remember that iterators are positions, not elements. For example, take the following sequence:

A B C

The sequence contains three elements and four positions

```
+---+---+---+
| A | B | C |   |
+---+---+---+
```

Elements are things within a sequence. Positions are places where meaningful operations can happen to the sequence. For example, one inserts into a position, *before* or *after* element A, not into an element. Even deletion of an element (*erase(A)*) is done by first finding its position, then deleting it.

From Iterators to Values

To convert from a position to a value, an iterator is *dereferenced*:

```
auto my_iterator = my_vector.begin(); // position
auto my_value = *my_iterator; // value
```

One can think of an iterator as dereferencing to the value it refers to in the sequence. This is especially useful in understanding why you should never dereference the end() iterator in a sequence:

```
+---+---+---+
| A | B | C |   |
+---+---+---+
↑           ↑
|           +- An iterator here has no value. Do not dereference it!
+----- An iterator here dereferences to the value A.
```

In all the sequences and containers found in the C++ standard library, begin() will return an iterator to the first position, and end() will return an iterator to one past the last position (*not* the last position!). Consequently, the names of these iterators in algorithms are oftentimes labelled first and last:

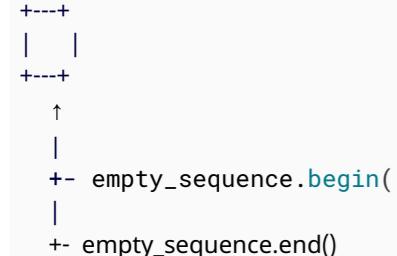
```
+---+---+---+
| A | B | C |   |
+---+---+---+
↑           ↑
|           |
+- first    +- last
```

It is also possible to obtain an iterator to *any sequence*, because even an empty sequence contains at least one

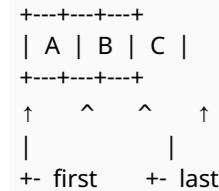
位置：



在空序列中，begin()和end()将是相同的位置，且两者都不能被解引用：



迭代器的另一种可视化方式是它们标记元素之间的位置：



解引用迭代器返回的是迭代器后面的元素的引用。在以下一些情况下，这种视角特别有用：

- insert 操作会在迭代器指示的位置插入元素，
- erase 操作会返回一个迭代器，该迭代器对应于传入的迭代器相同的位置，
- 一个迭代器及其对应的反向迭代器位于元素之间的同一位置

无效迭代器

如果（例如，在某个操作过程中）迭代器的位置不再是序列的一部分，则该迭代器会变为无效。

无效的迭代器在重新赋值为有效位置之前不能被解引用。例如：

```
std::vector<int>::iterator first;  
{  
    std::vector<int> foo;  
    first = foo.begin(); // first 现在是有效的  
} // foo 超出作用域并被销毁  
// 此时 first 变为无效
```

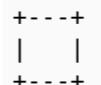
C++标准库中的许多算法和序列成员函数都有关于迭代器何时失效的规则。每个算法在处理（以及使迭代器失效）迭代器的方式上都不同。

使用迭代器进行导航

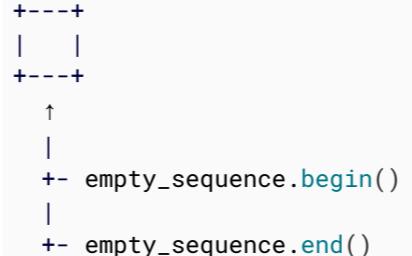
正如我们所知，迭代器用于导航序列。为了实现这一点，迭代器必须在序列中移动其位置。迭代器可以向序列前方移动，有些还可以向后移动：

```
auto first = my_vector.begin();  
++first; // 将迭代器前进1个位置
```

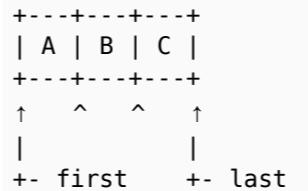
position:



In an empty sequence, begin() and end() will be the same position, and neither can be dereferenced:



The alternative visualization of iterators is that they mark the positions *between* elements:



and dereferencing an iterator returns a reference to the element coming after the iterator. Some situations where this view is particularly useful are:

- insert operations will insert elements into the position indicated by the iterator,
- erase operations will return an iterator corresponding to the same position as the one passed in,
- an iterator and its corresponding reverse iterator are located in the same .position between elements

Invalid Iterators

An iterator becomes *invalidated* if (say, in the course of an operation) its position is no longer a part of a sequence. An invalidated iterator cannot be dereferenced until it has been reassigned to a valid position. For example:

```
std::vector<int>::iterator first;  
{  
    std::vector<int> foo;  
    first = foo.begin(); // first is now valid  
} // foo falls out of scope and is destroyed  
// At this point first is now invalid
```

The many algorithms and sequence member functions in the C++ standard library have rules governing when iterators are invalidated. Each algorithm is different in the way they treat (and invalidate) iterators.

Navigating with Iterators

As we know, iterators are for navigating sequences. In order to do that an iterator must migrate its position throughout the sequence. Iterators can advance forward in the sequence and some can advance backwards:

```
auto first = my_vector.begin();  
++first; // advance the iterator 1 position
```

```

std::advance(first, 1);           // 将迭代器前进1个位置
first = std::next(first);        // 返回指向下一个元素的迭代器
std::advance(first, -1);         // 将迭代器后退1个位置
first = std::next(first, 20);    // 返回指向第20个位置元素的迭代器
forward
first = std::prev(first, 5);     // 返回指向第5个位置元素的迭代器
backward
auto dist = std::distance(my_vector.begin(), first); // 返回两个迭代器之间的距离。

```

注意，`std::distance`的第二个参数应当可以从第一个参数到达（换句话说，`first` 应小于或等于 `second`）。

尽管你可以对迭代器执行算术运算符，但并非所有操作都适用于所有类型的迭代器。`a = b + 3;` 对于随机访问迭代器来说是可行的，但对于前向迭代器或双向迭代器则不可行，后者仍然可以通过类似 `b = a; ++b; ++b; ++b;` 的方式前进3个位置。因此，如果不确定迭代器类型（例如，在接受迭代器的模板函数中），建议使用专门的函数。

迭代器概念

C++标准描述了几种不同的迭代器概念。这些概念根据它们在所指序列中的行为进行分组。如果你知道一个迭代器所模拟（表现为）的概念，你就可以确定该迭代器的行为无论它属于哪个序列。它们通常按从最严格到最宽松的顺序描述（因为下一个迭代器概念比前一个更进一步）：

- 输入迭代器：每个位置只能被解引用一次。只能向前移动，并且一次只能移动一个位置。
- 前向迭代器：一种可以任意次数解引用的输入迭代器。
- 双向迭代器：一种前向迭代器，也可以一次向后移动一个位置。
- 随机访问迭代器：一种双向迭代器，可以一次向前或向后移动任意多个位置。
- 连续迭代器（自C++17起）：一种随机访问迭代器，保证底层数据在内存中是连续的。

算法可能会根据所给迭代器所模拟的概念而有所不同。例如，虽然 `random_shuffle` 可以针对前向迭代器实现，但也可以提供一种更高效的变体，该变体需要随机访问迭代器。

迭代器特性

迭代器特性提供了对迭代器属性的统一接口。它们允许你获取值类型、差值类型、指针类型、引用类型以及迭代器类别：

```

template<class 迭代器>
迭代器 find(迭代器 first, 迭代器 last, typename std::iterator_traits<迭代器>::value_type val) {
    while (first != last) {
        if (*first == val)
            return first;
        ++first;
    }
    return last;
}

```

迭代器类别可以用来对算法进行特化：

```
template<class 双向迭代器>
```

```

std::advance(first, 1);           // advance the iterator 1 position
first = std::next(first);        // returns iterator to the next element
std::advance(first, -1);         // advance the iterator 1 position backwards
first = std::next(first, 20);    // returns iterator to the element 20 position
forward
first = std::prev(first, 5);      // returns iterator to the element 5 position
backward
auto dist = std::distance(my_vector.begin(), first); // returns distance between two iterators.

```

Note, second argument of `std::distance` should be reachable from the first one(or, in other words `first` should be less or equal than `second`).

Even though you can perform arithmetic operators with iterators, not all operations are defined for all types of iterators. `a = b + 3;` would work for Random Access Iterators, but wouldn't work for Forward or Bidirectional Iterators, which still can be advanced by 3 position with something like `b = a; ++b; ++b; ++b;`. So it is recommended to use special functions in case you are not sure what is iterator type (for example, in a template function accepting iterator).

Iterator Concepts

The C++ standard describes several different iterator concepts. These are grouped according to how they behave in the sequences they refer to. If you know the concept an iterator *models* (behaves like), you can be assured of the behavior of that iterator *regardless of the sequence to which it belongs*. They are often described in order from the most to least restrictive (because the next iterator concept is a step better than its predecessor):

- Input Iterators : Can be dereferenced *only once* per position. Can only advance, and only one position at a time.
- Forward Iterators : An input iterator that can be dereferenced any number of times.
- Bidirectional Iterators : A forward iterator that can also advance *backwards* one position at a time.
- Random Access Iterators : A bidirectional iterator that can advance forwards or backwards any number of positions at a time.
- Contiguous Iterators (since C++17) : A random access iterator that guarantees that underlying data is contiguous in memory.

Algorithms can vary depending on the concept modeled by the iterators they are given. For example, although `random_shuffle` can be implemented for forward iterators, a more efficient variant that requires random access iterators could be provided.

Iterator traits

Iterator traits provide uniform interface to the properties of iterators. They allow you to retrieve value, difference, pointer, reference types and also category of iterator:

```

template<class Iter>
Iter find(Iter first, Iter last, typename std::iterator_traits<Iter>::value_type val) {
    while (first != last) {
        if (*first == val)
            return first;
        ++first;
    }
    return last;
}

```

Category of iterator can be used to specialize algorithms:

```
template<class BidirIt>
```

```

void test(双向迭代器 a, std::bidirectional_iterator_tag) {
    std::cout << "使用了双向迭代器" << std::endl;
}

template<class 前向迭代器>
void test(前向迭代器 a, std::forward_iterator_tag) {
    std::cout << "使用了前向迭代器" << std::endl;
}

template<class 迭代器>
void test(Iter a) {
    test(a, typename std::iterator_traits<Iter>::iterator_category());
}

```

迭代器的类别基本上是迭代器的概念，除了连续迭代器（Contiguous Iterators）没有自己的标签，因为发现这样会破坏代码。

第9.2节：向量迭代器

`begin` 返回序列容器中第一个元素的迭代器。

`end` 返回第一个超出末尾元素的迭代器。

如果向量对象是`const`, `begin`和`end`都返回`const_iterator`。如果你希望即使向量不是`const`也返回`const_iterator`, 可以使用`cbegin`和`cend`。

示例：

```

#include <vector>
#include <iostream>

int main() {
    std::vector<int> v = { 1, 2, 3, 4, 5 }; // 使用初始化列表初始化向量

    for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
        std::cout << *it << " ";
    }

    return 0;
}

```

输出：

1 2 3 4 5

第9.3节：映射迭代器

指向容器中第一个元素的迭代器。

如果映射对象是常量限定的，函数返回一个`const_iterator`。否则，返回一个`iterator`。

```

// 创建一个映射并插入一些值
std::map<char,int> mymap;
mymap['b'] = 100;
mymap['a'] = 200;
mymap['c'] = 300;

```

```

void test(BidirIt a, std::bidirectional_iterator_tag) {
    std::cout << "Bidirectional iterator is used" << std::endl;
}

template<class ForwIt>
void test(ForwIt a, std::forward_iterator_tag) {
    std::cout << "Forward iterator is used" << std::endl;
}

template<class Iter>
void test(Iter a) {
    test(a, typename std::iterator_traits<Iter>::iterator_category());
}

```

Categories of iterators are basically iterators concepts, except Contiguous Iterators don't have their own tag, since it was found to break code.

Section 9.2: Vector Iterator

`begin` returns an `iterator` to the first element in the sequence container.

`end` returns an `iterator` to the first element past the end.

If the vector object is `const`, both `begin` and `end` return a `const_iterator`. If you want a `const_iterator` to be returned even if your vector is not `const`, you can use `cbegin` and `cend`.

Example:

```

#include <vector>
#include <iostream>

int main() {
    std::vector<int> v = { 1, 2, 3, 4, 5 }; // initialize vector using an initializer_list

    for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
        std::cout << *it << " ";
    }

    return 0;
}

```

Output:

1 2 3 4 5

Section 9.3: Map Iterator

An iterator to the first element in the container.

If a map object is const-qualified, the function returns a `const_iterator`. Otherwise, it returns an `iterator`.

```

// Create a map and insert some values
std::map<char,int> mymap;
mymap['b'] = 100;
mymap['a'] = 200;
mymap['c'] = 300;

```

```
// 遍历所有元组
for (std::map<char,int>::iterator it = mymap.begin(); it != mymap.end(); ++it)    std::cout << it->first << " => " << it->second << ";
```

输出：

```
a => 200
b => 100
c => 300
```

第9.4节：反向迭代器

如果我们想要反向遍历一个列表或向量，可以使用reverse_iterator。反向迭代器是由双向迭代器或随机访问迭代器构成的，它将该迭代器作为成员保存，可以通过base()访问。

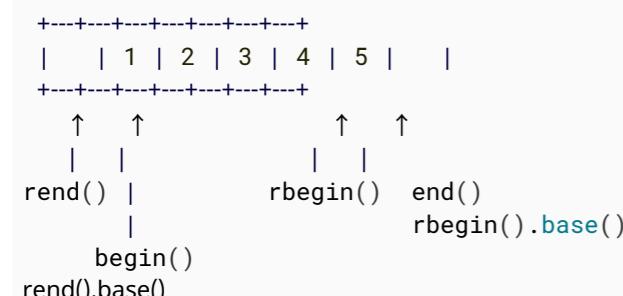
要反向迭代，使用begin()和end()作为集合的结束和开始的迭代器。

例如，要反向迭代，使用：

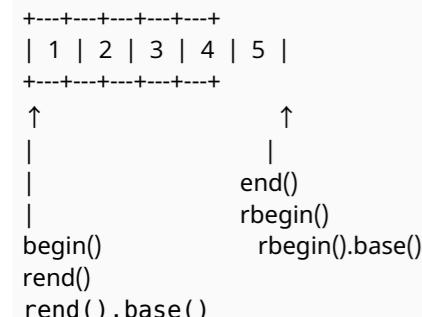
```
std::vector<int> v{1, 2, 3, 4, 5};
for (std::vector<int>::reverse_iterator it = v.rbegin(); it != v.rend(); ++it)
{
    cout << *it;
} // 输出 54321
```

反向迭代器可以通过base()成员函数转换为正向迭代器。其关系是反向迭代器引用的是base()迭代器之后的一个元素：

```
std::vector<int>::reverse_iterator r = v.rbegin();
std::vector<int>::iterator i = r.base();
assert(&r == &*(i-1)); // 如果 r 和 (i-1) 可解引用且不是代理迭代器，则总为真
```



在迭代器标记元素之间位置的可视化中，关系更简单：



```
// Iterate over all tuples
for (std::map<char,int>::iterator it = mymap.begin(); it != mymap.end(); ++it)
    std::cout << it->first << " => " << it->second << '\n';
```

Output:

```
a => 200
b => 100
c => 300
```

Section 9.4: Reverse Iterators

If we want to iterate backwards through a list or vector we can use a `reverse_iterator`. A reverse iterator is made from a bidirectional, or random access iterator which it keeps as a member which can be accessed through `base()`.

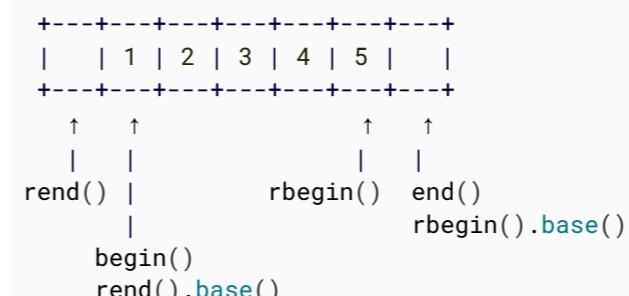
To iterate backwards use `rbegin()` and `rend()` as the iterators for the end of the collection, and the start of the collection respectively.

For instance, to iterate backwards use:

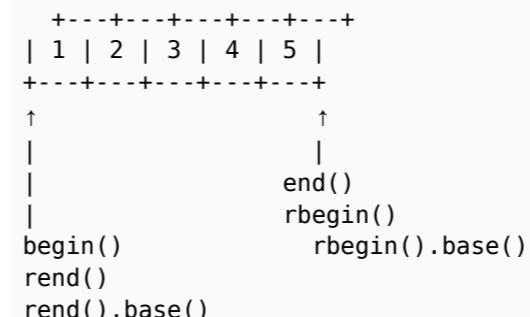
```
std::vector<int> v{1, 2, 3, 4, 5};
for (std::vector<int>::reverse_iterator it = v.rbegin(); it != v.rend(); ++it)
{
    cout << *it;
} // prints 54321
```

A reverse iterator can be converted to a forward iterator via the `base()` member function. The relationship is that the reverse iterator references one element past the `base()` iterator:

```
std::vector<int>::reverse_iterator r = v.rbegin();
std::vector<int>::iterator i = r.base();
assert(&r == &*(i-1)); // always true if r, (i-1) are dereferenceable
// and are not proxy iterators
```



In the visualization where iterators mark positions between elements, the relationship is simpler:



第9.5节：流迭代器

当我们需要从容器中读取序列或打印格式化数据时，流迭代器非常有用：

```
// 数据流。任意数量的各种空白字符都可以。  
std::istringstream istr("1 2    3 4");  
std::vector<int> v;  
  
// 构造流迭代器并将数据从流复制到向量中。  
std::copy(  
    // 迭代器，将以整数形式读取流数据。  
    std::istream_iterator<int>(istr),  
    // 默认构造函数生成流末尾迭代器。  
    std::istream_iterator<int>(),  
    std::back_inserter(v));  
  
// 打印向量内容。  
std::copy(v.begin(), v.end(),  
    // 将以整数形式打印值到标准输出，值之间用 " -- " 分隔。  
    std::ostream_iterator<int>(std::cout, " -- "));
```

示例程序将打印 1 -- 2 -- 3 -- 4 -- 到标准输出。

第9.6节：C语言迭代器（指针）

```
// 这将创建一个包含5个值的数组。  
const int array[] = { 1, 2, 3, 4, 5 };  
  
#ifdef BEFORE_CPP11  
  
// 你可以使用 `sizeof` 来确定数组中有多少元素。  
const int* first = array;  
const int* afterLast = first + sizeof(array) / sizeof(array[0]);  
  
// 然后你可以通过递增指针来遍历数组，直到指针超过数组末尾。  
  
for (const int* i = first; i < afterLast; ++i) {  
    std::cout << *i << std::endl;  
}  
  
#else  
  
// 使用 C++11，你可以让 STL 计算开始和结束的迭代器：  
for (auto i = std::begin(array); i != std::end(array); ++i) {  
    std::cout << *i << std::endl;  
}  
  
#endif
```

这段代码会输出数字 1 到 5，每个数字占一行，如下所示：



Section 9.5: Stream Iterators

Stream iterators are useful when we need to read a sequence or print formatted data from a container:

```
// Data stream. Any number of various whitespace characters will be OK.  
std::istringstream istr("1\t2    3 4");  
std::vector<int> v;  
  
// Constructing stream iterators and copying data from stream into vector.  
std::copy(  
    // Iterator which will read stream data as integers.  
    std::istream_iterator<int>(istr),  
    // Default constructor produces end-of-stream iterator.  
    std::istream_iterator<int>(),  
    std::back_inserter(v));  
  
// Print vector contents.  
std::copy(v.begin(), v.end(),  
    // Will print values to standard output as integers delimited by " -- ".  
    std::ostream_iterator<int>(std::cout, " -- "));
```

The example program will print 1 -- 2 -- 3 -- 4 -- to standard output.

Section 9.6: C Iterators (Pointers)

```
// This creates an array with 5 values.  
const int array[] = { 1, 2, 3, 4, 5 };  
  
#ifdef BEFORE_CPP11  
  
// You can use `sizeof` to determine how many elements are in an array.  
const int* first = array;  
const int* afterLast = first + sizeof(array) / sizeof(array[0]);  
  
// Then you can iterate over the array by incrementing a pointer until  
// it reaches past the end of our array.  
for (const int* i = first; i < afterLast; ++i) {  
    std::cout << *i << std::endl;  
}  
  
#else  
  
// With C++11, you can let the STL compute the start and end iterators:  
for (auto i = std::begin(array); i != std::end(array); ++i) {  
    std::cout << *i << std::endl;  
}  
  
#endif
```

This code would output the numbers 1 through 5, one on each line like this:

```
1  
2  
3  
4
```

分解说明

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

这一行创建了一个包含5个值的新整数数组。C语言中的数组只是指向内存的指针，每个值都存储在一个连续的内存块中。

```
const int* first = array;
const int* afterLast = first + sizeof(array) / sizeof(array[0]);
```

这些行创建了两个指针。第一个指针被赋值为数组指针，即数组第一个元素的地址。对C数组使用 `sizeof` 操作符时，会返回数组占用的字节数。

除以一个元素的大小后，就得到了数组中元素的数量。我们可以用它来找到数组后面内存块的地址。

```
for (const int* i = first; i < afterLast; ++i) {
```

这里我们创建了一个指针，用作迭代器。它被初始化为我们想要迭代的第一个元素的地址，只要 `i` 小于 `afterLast`，我们就继续迭代，这意味着 `i` 指向的地址仍在 `array` 内。

```
std::cout << *i << std::endl;
```

最后，在循环中我们可以通过解引用迭代器 `i` 来访问它指向的值。这里的解引用操作符 `*` 返回 `i` 中地址对应的值。

第9.7节：编写你自己的基于生成器的迭代器

其他语言中常见的模式是有一个函数产生对象的“流”，并且能够使用循环代码对其进行遍历。

我们可以用C++来建模这个

```
template<class T>
struct generator_iterator {
    using difference_type=std::ptrdiff_t;
    using value_type=T;
    using pointer=T*;
    using reference=T;
    using iterator_category=std::input_iterator_tag;
    std::optional<T> state;
    std::function< std::optional<T>() > operation;
    // 如果有当前元素，我们将其存储在"state"中：
    T operator*() const {
        return *state;
    }
    // 要前进，我们调用操作函数。如果它返回nullopt
    // 表示已经到达末尾：
    generator_iterator& operator++() {
        state = operation();
        return *this;
    }
    generator_iterator 操作符++(int) {
        auto r = *this;
```

Breaking It Down

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

This line creates a new integer array with 5 values. C arrays are just pointers to memory where each value is stored together in a contiguous block.

```
const int* first = array;
const int* afterLast = first + sizeof(array) / sizeof(array[0]);
```

These lines create two pointers. The first pointer is given the value of the array pointer, which is the address of the first element in the array. The `sizeof` operator when used on a C array returns the size of the array in bytes. Divided by the size of an element this gives the number of elements in the array. We can use this to find the address of the block *after* the array.

```
for (const int* i = first; i < afterLast; ++i) {
```

Here we create a pointer which we will use as an iterator. It is initialized with the address of the first element we want to iterate over, and we'll continue to iterate as long as `i` is less than `afterLast`, which means as long as `i` is pointing to an address within `array`.

```
    std::cout << *i << std::endl;
```

Finally, within the loop we can access the value our iterator `i` is pointing to by dereferencing it. Here the dereference operator `*` returns the value at the address in `i`.

Section 9.7: Write your own generator-backed iterator

A common pattern in other languages is having a function that produces a "stream" of objects, and being able to use loop-code to loop over it.

We can model this in C++ as

```
template<class T>
struct generator_iterator {
    using difference_type=std::ptrdiff_t;
    using value_type=T;
    using pointer=T*;
    using reference=T;
    using iterator_category=std::input_iterator_tag;
    std::optional<T> state;
    std::function< std::optional<T>() > operation;
    // we store the current element in "state" if we have one:
    T operator*() const {
        return *state;
    }
    // to advance, we invoke our operation. If it returns a nullopt
    // we have reached the end:
    generator_iterator& operator++() {
        state = operation();
        return *this;
    }
    generator_iterator operator++(int) {
        auto r = *this;
```

```

++(*this);
return r;
}
// generator 迭代器只有在两者都处于“结束”状态时才相等：
friend bool operator==( generator_iterator const& lhs, generator_iterator const& rhs ) {
if (!lhs.state && !rhs.state) return true;
return false;
}
friend bool operator!=( generator_iterator const& lhs, generator_iterator const& rhs ) {
return !(lhs==rhs);
}
// 我们隐式地从具有正确签名的 std::function 构造：
generator_iterator( std::function< std::optional<T>() > f ):operation(std::move(f))
{
    if (operation)
        state = operation();
}
// 默认所有特殊成员函数：
generator_iterator( generator_iterator && ) =default;
generator_iterator( generator_iterator const& ) =default;
generator_iterator& operator=( generator_iterator && ) =default;
generator_iterator& operator=( generator_iterator const& ) =default;
generator_iterator() =default;
};

```

实时示例。

我们提前存储生成的元素，这样可以更容易地检测是否已经到达末尾。

由于结束生成器迭代器的函数从未被使用，我们可以通过只复制一次 `std::function` 来创建一系列生成器迭代器。默认构造的生成器迭代器与自身以及所有其他结束生成器迭代器比较时相等。

```

++(*this);
return r;
}
// generator iterators are only equal if they are both in the "end" state:
friend bool operator==( generator_iterator const& lhs, generator_iterator const& rhs ) {
if (!lhs.state && !rhs.state) return true;
return false;
}
friend bool operator!=( generator_iterator const& lhs, generator_iterator const& rhs ) {
return !(lhs==rhs);
}
// We implicitly construct from a std::function with the right signature:
generator_iterator( std::function< std::optional<T>() > f ):operation(std::move(f))
{
    if (operation)
        state = operation();
}
// default all special member functions:
generator_iterator( generator_iterator && ) =default;
generator_iterator( generator_iterator const& ) =default;
generator_iterator& operator=( generator_iterator && ) =default;
generator_iterator& operator=( generator_iterator const& ) =default;
generator_iterator() =default;
};

```

live example.

We store the generated element early so we can more easily detect if we are already at the end.

As the function of an end generator iterator is never used, we can create a range of generator iterators by only copying the `std::function` once. A default constructed generator iterator compares equal to itself, and to all other end-generator-iterators.

第10章：C++中的基本输入/输出

第10.1节：用户输入和标准输出

```
#include <iostream>

int main()
{
    int value;
    std::cout << "请输入一个值: " << std::endl;
    std::cin >> value;
    std::cout << "输入值的平方是: " << value * value << std::endl;
    return 0;
}
```

Chapter 10: Basic input/output in c++

Section 10.1: user input and standard output

```
#include <iostream>

int main()
{
    int value;
    std::cout << "Enter a value: " << std::endl;
    std::cin >> value;
    std::cout << "The square of entered value is: " << value * value << std::endl;
    return 0;
}
```

第11章：循环

循环语句重复执行一组语句，直到满足某个条件。C++中有3种基本循环类型：for、while和do...while。

第11.1节：基于范围的for循环

版本 ≥ C++11

for循环可用于遍历基于迭代器的范围内的元素，无需使用数字索引或直接访问迭代器：

```
vector<float> v = {0.4f, 12.5f, 16.234f};

for(auto val: v)
{
    std::cout << val << " ";
}

std::cout << std::endl;
```

这将遍历v中的每个元素，val获取当前元素的值。以下语句：

```
for (for-range-declaration : for-range-initializer ) statement
```

等价于：

```
{
    auto&& __range = for-range-initializer;
    auto __begin = begin-expr, __end = end-expr;
    for (; __begin != __end; ++__begin) {
        for-范围-声明 = *__begin;
        语句
    }
}
```

版本 ≥ C++17

```
{
    auto&& __range = for-范围-初始化器;
    auto __begin = begin-表达式;
    auto __end = end-表达式; // 在 C++17 中, end 允许与 begin 类型不同
    for (; __begin != __end; ++__begin) {
        for-范围-声明 = *__begin;
        语句
    }
}
```

此更改是为计划支持 C++20 中的 Ranges TS 引入的。

在这种情况下，我们的循环等价于：

```
{
    auto&& __range = v;
    auto __begin = v.begin(), __end = v.end();
    for (; __begin != __end; ++__begin) {
        auto val = *__begin;
        std::cout << val << " ";
    }
}
```

Chapter 11: Loops

A loop statement executes a group of statements repeatedly until a condition is met. There are 3 types of primitive loops in C++: for, while, and do...while.

Section 11.1: Range-Based For

Version ≥ C++11

for loops can be used to iterate over the elements of a iterator-based range, without using a numeric index or directly accessing the iterators:

```
vector<float> v = {0.4f, 12.5f, 16.234f};

for(auto val: v)
{
    std::cout << val << " ";
}

std::cout << std::endl;
```

This will iterate over every element in v, with val getting the value of the current element. The following statement:

```
for (for-range-declaration : for-range-initializer ) statement
```

is equivalent to:

```
{
    auto&& __range = for-range-initializer;
    auto __begin = begin-expr, __end = end-expr;
    for (; __begin != __end; ++__begin) {
        for-range-declaration = *__begin;
        statement
    }
}
```

Version ≥ C++17

```
{
    auto&& __range = for-range-initializer;
    auto __begin = begin-expr;
    auto __end = end-expr; // end is allowed to be a different type than begin in C++17
    for (; __begin != __end; ++__begin) {
        for-range-declaration = *__begin;
        statement
    }
}
```

This change was introduced for the planned support of Ranges TS in C++20.

In this case, our loop is equivalent to:

```
{
    auto&& __range = v;
    auto __begin = v.begin(), __end = v.end();
    for (; __begin != __end; ++__begin) {
        auto val = *__begin;
        std::cout << val << " ";
    }
}
```

```
}
```

请注意，`auto val` 声明了一个值类型，它将是存储在范围内的值的副本（我们在遍历时从迭代器复制初始化它）。如果范围内存储的值复制代价较高，您可能想使用`const auto &val`。您也不必非用`auto`；只要它能隐式转换自范围的值类型，您可以使用合适的类型名。

如果您需要访问迭代器，基于范围的`for`循环无法帮到您（至少不轻松）。

如果您想引用它，可以这样做：

```
vector<float> v = {0.4f, 12.5f, 16.234f};

for(float &val: v)
{
    std::cout << val << " ";
}
```

如果您有`const`容器，可以对`const`引用进行迭代：

```
const vector<float> v = {0.4f, 12.5f, 16.234f};

for(const float &val: v)
{
    std::cout << val << " ";
}
```

当序列迭代器返回代理对象且您需要以非`const`方式操作该对象时，会使用转发引用。注意：这很可能会让阅读您代码的人感到困惑。

```
vector<bool> v(10);

for(auto&& val: v)
{
    val = true;
}
```

提供给基于范围的`for`的“范围”类型可以是以下之一：

- 语言数组：

```
float arr[] = {0.4f, 12.5f, 16.234f};

for(auto val: arr)
{
    std::cout << val << " ";
}
```

注意，分配动态数组不算数：

```
float *arr = new float[3]{0.4f, 12.5f, 16.234f};

for(auto val: arr) //编译错误。
{
    std::cout << val << " ";
}
```

```
}
```

```
}
```

Note that `auto val` declares a value type, which will be a copy of a value stored in the range (we are copy-initializing it from the iterator as we go). If the values stored in the range are expensive to copy, you may want to use `const auto &val`. You are also not required to use `auto`; you can use an appropriate typename, so long as it is implicitly convertible from the range's value type.

If you need access to the iterator, range-based `for` cannot help you (not without some effort, at least).

If you wish to reference it, you may do so:

```
vector<float> v = {0.4f, 12.5f, 16.234f};

for(float &val: v)
{
    std::cout << val << " ";
}
```

You could iterate on `const` reference if you have `const` container:

```
const vector<float> v = {0.4f, 12.5f, 16.234f};

for(const float &val: v)
{
    std::cout << val << " ";
}
```

One would use forwarding references when the sequence iterator returns a proxy object and you need to operate on that object in a non-`const` way. Note: it will most likely confuse readers of your code.

```
vector<bool> v(10);

for(auto&& val: v)
{
    val = true;
}
```

The “range” type provided to range-based `for` can be one of the following:

- Language arrays:

```
float arr[] = {0.4f, 12.5f, 16.234f};

for(auto val: arr)
{
    std::cout << val << " ";
}
```

Note that allocating a dynamic array does not count:

```
float *arr = new float[3]{0.4f, 12.5f, 16.234f};

for(auto val: arr) //Compile error.
{
    std::cout << val << " ";
}
```

- 任何具有成员函数begin()和end()，且返回该类型元素的迭代器的类型。
标准库容器符合条件，但用户自定义类型也可以使用：

```
结构体 Rng
{
    float arr[3];

    // 指针即迭代器
    const float* begin() const {return &arr[0];}
    const float* end() const {return &arr[3];}
    float* begin() {return &arr[0];}
    float* end() {return &arr[3];}
};

int main()
{
    Rng rng = {{0.4f, 12.5f, 16.234f}};

    for(auto val: rng)
    {
        std::cout << val << " ";
    }
}
```

- 任何具有非成员函数begin(type)和end(type)，且这些函数可以通过基于 type 的参数依赖查找找到的类型。
这对于创建范围类型而无需修改类类型本身非常有用：

```
namespace Mine
{
    struct Rng {float arr[3];};

    // 指针即迭代器
    const float* begin(const Rng &rng) {return &rng.arr[0];}
    const float* end(const Rng &rng) {return &rng.arr[3];}
    float* begin(Rng &rng) {return &rng.arr[0];}
    float* end(Rng &rng) {return &rng.arr[3];}
}

int main()
{
    Mine::Rng rng = {{0.4f, 12.5f, 16.234f}};

    for(auto val: rng)
    {
        std::cout << val << " ";
    }
}
```

第11.2节：for循环

for循环在循环条件为真时执行循环体中的语句。循环初始化语句在循环开始时执行一次。每次循环后，执行迭代语句。

for循环定义如下：

```
for /*初始化语句*/ ; /*条件*/ ; /*迭代语句*/
```

- Any type which has member functions begin() and end(), which return iterators to the elements of the type.
The standard library containers qualify, but user-defined types can be used as well:

```
struct Rng
{
    float arr[3];

    // pointers are iterators
    const float* begin() const {return &arr[0];}
    const float* end() const {return &arr[3];}
    float* begin() {return &arr[0];}
    float* end() {return &arr[3];}
};

int main()
{
    Rng rng = {{0.4f, 12.5f, 16.234f}};

    for(auto val: rng)
    {
        std::cout << val << " ";
    }
}
```

- Any type which has non-member begin(type) and end(type) functions which can be found via argument dependent lookup, based on type. This is useful for creating a range type without having to modify class type itself:

```
namespace Mine
{
    struct Rng {float arr[3];};

    // pointers are iterators
    const float* begin(const Rng &rng) {return &rng.arr[0];}
    const float* end(const Rng &rng) {return &rng.arr[3];}
    float* begin(Rng &rng) {return &rng.arr[0];}
    float* end(Rng &rng) {return &rng.arr[3];}
}

int main()
{
    Mine::Rng rng = {{0.4f, 12.5f, 16.234f}};

    for(auto val: rng)
    {
        std::cout << val << " ";
    }
}
```

Section 11.2: For loop

A **for** loop executes statements in the **loop body**, while the **loop condition** is true. Before the **loop initialization** statement is executed exactly once. After each cycle, the **iteration execution** part is executed.

A **for** loop is defined as follows:

```
for /*initialization statement*/ ; /*condition*/ ; /*iteration execution*/
```

```
{  
    // 循环体  
}
```

占位语句说明：

- 初始化语句：该语句仅在for循环开始时执行一次。你可以声明同一类型的多个变量，例如int i = 0, a = 2, b = 3。这些变量仅在循环作用域内有效。循环外定义的同名变量在循环执行期间被隐藏。
- 条件：该语句在每次执行循环体之前进行判断，如果结果为false，则终止循环。
- 迭代语句：该语句在循环体执行后、下一次条件判断之前执行，除非循环体内通过break、goto、return 或抛出异常等方式终止了for循环。
您可以在迭代执行部分输入多个语句，例如a++、b+=10、c=b+a。

将for循环大致等价地重写为while循环的形式是：

```
/*初始化*/  
while /*条件*/  
{  
    // 循环体；使用 'continue' 会跳过下面的迭代执行部分  
    /*迭代执行*/  
}
```

使用for循环最常见的情况是执行特定次数的语句。例如，考虑以下代码：

```
for(int i = 0; i < 10; i++) {  
    std::cout << i << std::endl;  
}
```

一个有效的循环也可以是：

```
for(int a = 0, b = 10, c = 20; (a+b+c < 100); c--, b++, a+=c) {  
    std::cout << a << " " << b << " " << c << std::endl;  
}
```

在循环前隐藏声明变量的一个例子是：

```
int i = 99; //i = 99  
for(int i = 0; i < 10; i++) { //我们声明了一个新的变量 i  
    //一些操作，循环执行期间 i 的值从 0 到 9 变化  
}  
//循环执行完后，我们可以访问值为 99 的 i
```

但是如果你想使用已经声明的变量而不隐藏它，则省略声明部分：

```
int i = 99; //i = 99  
for(i = 0; i < 10; i++) { //我们使用已经声明的变量 i  
    //一些操作，循环执行期间 i 的值从 0 到 9 变化  
}  
//循环执行完后，我们可以访问值为 10 的 i
```

注意：

```
{  
    // body of the loop  
}
```

Explanation of the placeholder statements:

- initialization statement: This statement gets executed only once, at the beginning of the `for` loop. You can enter a declaration of multiple variables of one type, such as `int i = 0, a = 2, b = 3`. These variables are only valid in the scope of the loop. Variables defined before the loop with the same name are hidden during execution of the loop.
- condition: This statement gets evaluated ahead of each *loop body* execution, and aborts the loop if it evaluates to `false`.
- iteration execution: This statement gets executed after the *loop body*, ahead of the next *condition* evaluation, unless the `for` loop is aborted in the *body* (by `break`, `goto`, `return` or an exception being thrown). You can enter multiple statements in the *iteration execution* part, such as `a++, b+=10, c=b+a`.

The rough equivalent of a `for` loop, rewritten as a `while` loop is:

```
/*initialization*/  
while /*condition*/  
{  
    // body of the loop; using 'continue' will skip to increment part below  
    /*iteration execution*/  
}
```

The most common case for using a `for` loop is to execute statements a specific number of times. For example, consider the following:

```
for(int i = 0; i < 10; i++) {  
    std::cout << i << std::endl;  
}
```

A valid loop is also:

```
for(int a = 0, b = 10, c = 20; (a+b+c < 100); c--, b++, a+=c) {  
    std::cout << a << " " << b << " " << c << std::endl;  
}
```

An example of hiding declared variables before a loop is:

```
int i = 99; //i = 99  
for(int i = 0; i < 10; i++) { //we declare a new variable i  
    //some operations, the value of i ranges from 0 to 9 during loop execution  
}  
//after the loop is executed, we can access i with value of 99
```

But if you want to use the already declared variable and not hide it, then omit the declaration part:

```
int i = 99; //i = 99  
for(i = 0; i < 10; i++) { //we are using already declared variable i  
    //some operations, the value of i ranges from 0 to 9 during loop execution  
}  
//after the loop is executed, we can access i with value of 10
```

Notes:

- 初始化语句和递增语句可以执行与条件语句无关的操作，或者什么都不做——如果你愿意的话。但为了可读性，最佳实践是只执行与循环直接相关的操作。
- 在初始化语句中声明的变量仅在for循环的作用域内可见，循环结束后该变量被释放。
- 别忘了，在初始化语句中声明的变量可以在循环中被修改，条件语句中检查的变量也可以被修改。

从0计数到10的循环示例：

```
for (int 计数器 = 0; 计数器 <= 10; ++计数器)
{
    std::cout << 计数器 << "\n";
}

// 计数器在这里不可访问 (结束时值为11)
```

代码片段说明：

- int 计数器 = 0 初始化变量 计数器 为0。（该变量只能在 for 循环内部使用。）计数器 <= 10 是一个布尔条件，用于检查 计数器 是否小于或等于10。如果为 true，循环执行。如果为 false，循环结束。
- ++计数器 是一个自增操作，在下一次条件检查之前将 计数器 的值加1。

通过将所有语句留空，可以创建一个无限循环：

```
// 无限循环
for (;;)
    std::cout << "永无止境！";
```

上述代码的 while 循环等价写法是：

```
// 无限循环
while (true)
    std::cout << "永无止境！";
```

然而，仍然可以通过使用语句 break、goto 或 return，或者抛出异常来留下无限循环。

下一个常见的例子是在不使用

```
<algorithm> 头文件的情况下，通过 STL 容器（例如 vector）中所有元素的代码是：

std::vector<std::string> names = {"阿尔伯特·爱因斯坦", "斯蒂芬·霍金", "迈克尔·埃利斯"};
for(std::vector<std::string>::iterator it = names.begin(); it != names.end(); ++it) {
    std::cout << *it << std::endl;
}
```

第11.3节：while 循环

while 循环会重复执行语句，直到给定条件计算结果为 false。该控制语句用于事先不知道代码块需要执行多少次的情况。

例如，要打印从0到9的所有数字，可以使用以下代码：

```
int i = 0;
```

- The initialization and increment statements can perform operations unrelated to the condition statement, or nothing at all - if you wish to do so. But for readability reasons, it is best practice to only perform operations directly relevant to the loop.
- A variable declared in the initialization statement is visible only inside the scope of the `for` loop and is released upon termination of the loop.
- Don't forget that the variable which was declared in the `initialization statement` can be modified during the loop, as well as the variable checked in the condition.

Example of a loop which counts from 0 to 10:

```
for (int counter = 0; counter <= 10; ++counter)
{
    std::cout << counter << '\n';
}
// counter is not accessible here (had value 11 at the end)
```

Explanation of the code fragments:

- `int counter = 0` initializes the variable `counter` to 0. (This variable can only be used inside of the `for` loop.)
- `counter <= 10` is a Boolean condition that checks whether `counter` is less than or equal to 10. If it is `true`, the loop executes. If it is `false`, the loop ends.
- `++counter` is an increment operation that increments the value of `counter` by 1 ahead of the next condition check.

By leaving all statements empty, you can create an infinite loop:

```
// infinite loop
for (;;)
    std::cout << "Never ending!\n";
```

The `while` loop equivalent of the above is:

```
// infinite loop
while (true)
    std::cout << "Never ending!\n";
```

However, an infinite loop can still be left by using the statements `break`, `goto`, or `return` or by throwing an exception.

The next common example of iterating over all elements from an STL collection (e.g., a `vector`) without using the `<algorithm>` header is:

```
std::vector<std::string> names = {"Albert Einstein", "Stephen Hawking", "Michael Ellis"};
for(std::vector<std::string>::iterator it = names.begin(); it != names.end(); ++it) {
    std::cout << *it << std::endl;
}
```

Section 11.3: While loop

A `while` loop executes statements repeatedly until the given condition evaluates to `false`. This control statement is used when it is not known, in advance, how many times a block of code is to be executed.

For example, to print all the numbers from 0 up to 9, the following code can be used:

```
int i = 0;
```

```

while (i < 10)
{
    std::cout << i << " ";
    ++i; // 计数器递增
}
std::cout << std::endl; // 行结束；控制台打印“0 1 2 3 4 5 6 7 8 9”

```

版本 ≥ C++17

注意，自C++17起，前两条语句可以合并

```

while (int i = 0; i < 10)
//... 其余部分相同

```

要创建一个无限循环，可以使用以下结构：

```

while (true)
{
    // 永远执行某些操作（不过，可以通过调用'break'退出循环）
}

```

还有另一种while循环的变体，即do...while结构。更多信息请参见do-while循环示例。

第11.4节：Do-while循环

do-while循环与while循环非常相似，不同之处在于条件是在每个循环结束时检查，而不是开始时。因此，该循环至少会执行一次。

以下代码将打印0，因为条件在第一次迭代结束时会计算为false：

```

int i = 0;
do
{
    std::cout << i;
    ++i; // 计数器递增
}
while (i < 0);
std::cout << std::endl; // 行尾；0被打印到控制台

```

注意：不要忘记在while(condition);末尾的分号，这是do-while结构所必需的。

与do-while循环相反，以下代码不会打印任何内容，因为条件在第一次迭代开始时计算为false：

```

int i = 0;
while (i < 0)
{
    std::cout << i;
    ++i; // 计数器递增
}
std::cout << std::endl; // 行尾；控制台不打印任何内容

```

注意：可以使用break、goto或return语句退出while循环，而无需条件变为假。

```

int i = 0;
do
{

```

```

while (i < 10)
{
    std::cout << i << " ";
    ++i; // Increment counter
}
std::cout << std::endl; // End of line; "0 1 2 3 4 5 6 7 8 9" is printed to the console

```

Version ≥ C++17

Note that since C++17, the first 2 statements can be combined

```

while (int i = 0; i < 10)
//... The rest is the same

```

To create an infinite loop, the following construct can be used:

```

while (true)
{
    // Do something forever (however, you can exit the loop by calling 'break'
}

```

There is another variant of `while` loops, namely the `do...while` construct. See the do-while loop example for more information.

Section 11.4: Do-while loop

A `do-while` loop is very similar to a `while` loop, except that the condition is checked at the end of each cycle, not at the start. The loop is therefore guaranteed to execute at least once.

The following code will print 0, as the condition will evaluate to `false` at the end of the first iteration:

```

int i = 0;
do
{
    std::cout << i;
    ++i; // Increment counter
}
while (i < 0);
std::cout << std::endl; // End of line; 0 is printed to the console

```

Note: Do not forget the semicolon at the end of `while(condition);`, which is needed in the `do-while` construct.

In contrast to the `do-while` loop, the following will not print anything, because the condition evaluates to `false` at the beginning of the first iteration:

```

int i = 0;
while (i < 0)
{
    std::cout << i;
    ++i; // Increment counter
}
std::cout << std::endl; // End of line; nothing is printed to the console

```

Note: A `while` loop can be exited without the condition becoming false by using a `break`, `goto`, or `return` statement.

```

int i = 0;
do
{

```

```

std::cout << i;
++i; // 计数器递增
if (i > 5)
{
    break;
}
}
while (true);
std::cout << std::endl; // 行结束；控制台打印 0 1 2 3 4 5

```

一个简单的do-while循环有时也用于编写需要自己作用域的宏（此时宏定义中省略尾部分号，要求用户提供）：

```

#define BAD_MACRO(x) f1(x); f2(x); f3(x);

// 这里只有对 f1 的调用受条件保护
if (cond) BAD_MACRO(var);

#define GOOD_MACRO(x) do { f1(x); f2(x); f3(x); } while(0)

// 这里所有调用都受保护
if (cond) GOOD_MACRO(var);

```

第11.5节：循环控制语句：break和continue

循环控制语句用于改变执行流程的正常顺序。当执行离开一个作用域时，该作用域中创建的所有自动对象都会被销毁。`break`和`continue`是循环控制语句。

`break`语句会立即终止循环，不做任何进一步处理。

```

for (int i = 0; i < 10; i++)
{
    if (i == 4)
        break; // 这将立即退出我们的循环    std::cout << i <<
";"

```

上述代码将输出：

```

(int i = 0; i < 6; i++)
{
    if (i % 2 == 0) // 当i为偶数时条件为真
        continue; // 这将立即回到循环开始处
    /* 只有当上述"continue"语句未执行时，下一行才会被执行 */
    std::cout << i << " 是一个奇数";
}

```

上述代码将输出：

```

std::cout << i;
++i; // Increment counter
if (i > 5)
{
    break;
}
}
while (true);
std::cout << std::endl; // End of line; 0 1 2 3 4 5 is printed to the console

```

A trivial *do-while* loop is also occasionally used to write macros that require their own scope (in which case the trailing semicolon is omitted from the macro definition and required to be provided by the user):

```

#define BAD_MACRO(x) f1(x); f2(x); f3(x);

// Only the call to f1 is protected by the condition here
if (cond) BAD_MACRO(var);

#define GOOD_MACRO(x) do { f1(x); f2(x); f3(x); } while(0)

// All calls are protected here
if (cond) GOOD_MACRO(var);

```

Section 11.5: Loop Control statements : Break and Continue

Loop control statements are used to change the flow of execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. The `break` and `continue` are loop control statements.

The `break` statement terminates a loop without any further consideration.

```

for (int i = 0; i < 10; i++)
{
    if (i == 4)
        break; // this will immediately exit our loop
    std::cout << i << '\n';
}

```

The above code will print out:

```

1
2
3

```

The `continue` statement does not immediately exit the loop, but rather skips the rest of the loop body and goes to the top of the loop (including checking the condition).

```

for (int i = 0; i < 6; i++)
{
    if (i % 2 == 0) // evaluates to true if i is even
        continue; // this will immediately go back to the start of the loop
    /* the next line will only be reached if the above "continue" statement
       does not execute */
    std::cout << i << " is an odd number\n";
}

```

The above code will print out:

```
1是一个奇数  
3是一个奇数  
5是一个奇数
```

因为这种控制流的变化有时人们不容易理解，`break` 和 `continue` 通常会被谨慎使用。更直接的实现通常更容易阅读和理解。例如，上面带有 `break` 的第一个 `for` 循环可以重写为：

```
for (int i = 0; i < 4; i++)  
{  
    std::cout << i << ":";
```

带有 `continue` 的第二个示例可以重写为：

```
for (int i = 0; i < 6; i++)  
{  
    if (i % 2 != 0) {  
        std::cout << i << " 是一个奇数";  
    }  
}
```

第11.6节：条件中变量的声明

在 `for` 和 `while` 循环的条件下，也允许声明一个对象。该对象将被视为在循环结束之前处于作用域内，并且在循环的每次迭代中持续存在：

```
for (int i = 0; i < 5; ++i) {  
    do_something(i);  
}  
// i 不再处于作用域内。  
  
for (auto& a : some_container) {  
    a.do_something();  
}  
// a 不再处于作用域内。  
  
while(std::shared_ptr<Object> p = get_object()) {  
    p->do_something();  
}  
// p 不再处于作用域内。
```

但是，不允许在 `do...while` 循环中这样做；相反，应在循环之前声明变量，并且（可选地）如果希望变量在循环结束后退出作用域，可以将变量和循环都包含在局部作用域内：

```
//这段代码无法编译  
do {  
    s = do_something();  
} while (short s > 0);  
  
// 好的  
short s;  
do {  
    s = do_something();  
} while (s > 0);
```

```
1 is an odd number  
3 is an odd number  
5 is an odd number
```

Because such control flow changes are sometimes difficult for humans to easily understand, `break` and `continue` are used sparingly. More straightforward implementation are usually easier to read and understand. For example, the first `for` loop with the `break` above might be rewritten as:

```
for (int i = 0; i < 4; i++)  
{  
    std::cout << i << '\n';  
}
```

The second example with `continue` might be rewritten as:

```
for (int i = 0; i < 6; i++)  
{  
    if (i % 2 != 0) {  
        std::cout << i << " is an odd number\n";  
    }  
}
```

Section 11.6: Declaration of variables in conditions

In the condition of the `for` and `while` loops, it's also permitted to declare an object. This object will be considered to be in scope until the end of the loop, and will persist through each iteration of the loop:

```
for (int i = 0; i < 5; ++i) {  
    do_something(i);  
}  
// i is no longer in scope.  
  
for (auto& a : some_container) {  
    a.do_something();  
}  
// a is no longer in scope.  
  
while(std::shared_ptr<Object> p = get_object()) {  
    p->do_something();  
}  
// p is no longer in scope.
```

However, it is not permitted to do the same with a `do...while` loop; instead, declare the variable before the loop, and (optionally) enclose both the variable and the loop within a local scope if you want the variable to go out of scope after the loop ends:

```
//This doesn't compile  
do {  
    s = do_something();  
} while (short s > 0);  
  
// Good  
short s;  
do {  
    s = do_something();  
} while (s > 0);
```

这是因为 do...while 循环的 statement 部分（循环体）在达到 expression部分（即 while）之前就会被执行，因此，expression 中的任何声明在循环的第一次迭代中都不可见。

第11.7节：对子范围的范围for循环

使用基于范围的循环，可以通过生成一个符合基于范围的for循环要求的代理对象，遍历给定容器或其他范围的子部分。

```
template<class 迭代器, class 哨兵=迭代器>
struct range_t {
    迭代器 b;
    哨兵 e;
    迭代器 begin() const { return b; }
    哨兵 end() const { return e; }
    bool empty() const { return begin()==end(); }
    range_t without_front( std::size_t count=1 ) const {
        if (std::is_same< std::random_access_iterator_tag, typename
std::iterator_traits<迭代器>::iterator_category >{} ) {
            count = (std::min)(std::size_t(std::distance(b,e)), count);
        }
        return {std::next(b, count), e};
    }
    range_t without_back( std::size_t count=1 ) const {
        if (std::is_same< std::random_access_iterator_tag, typename
std::iterator_traits<Iterator>::iterator_category >{} ) {
            count = (std::min)(std::size_t(std::distance(b,e)), count);
        }
        return {b, std::prev(e, count)};
    }
};

template<class Iterator, class Sentinel>
range_t<Iterator, Sentinel> range( Iterator b, Sentinel e ) {
    return {b,e};
}
template<class Iterable>
auto range( Iterable& r ) {
    using std::begin; using std::end;
    return range(begin(r),end(r));
}

template<class C>
auto except_first( C& c ) {
    auto r = range(c);
    if (r.empty()) return r;
    return r.without_front();
}
```

现在我们可以这样做：

```
std::vector<int> v = {1,2,3,4};for (auto
    i : except_first(v)) std::cout << i <
    < ";
```

并打印输出

This is because the *statement* portion of a `do...while` loop (the loop's body) is evaluated before the *expression* portion (the `while`) is reached, and thus, any declaration in the *expression* will not be visible during the first iteration of the loop.

Section 11.7: Range-for over a sub-range

Using range-base loops, you can loop over a sub-part of a given container or other range by generating a proxy object that qualifies for range-based loops.

```
template<class Iterator, class Sentinel=Iterator>
struct range_t {
    Iterator b;
    Sentinel e;
    Iterator begin() const { return b; }
    Sentinel end() const { return e; }
    bool empty() const { return begin()==end(); }
    range_t without_front( std::size_t count=1 ) const {
        if (std::is_same< std::random_access_iterator_tag, typename
std::iterator_traits<Iterator>::iterator_category >{} ) {
            count = (std::min)(std::size_t(std::distance(b,e)), count);
        }
        return {std::next(b, count), e};
    }
    range_t without_back( std::size_t count=1 ) const {
        if (std::is_same< std::random_access_iterator_tag, typename
std::iterator_traits<Iterator>::iterator_category >{} ) {
            count = (std::min)(std::size_t(std::distance(b,e)), count);
        }
        return {b, std::prev(e, count)};
    }
};

template<class Iterator, class Sentinel>
range_t<Iterator, Sentinel> range( Iterator b, Sentinel e ) {
    return {b,e};
}
template<class Iterable>
auto range( Iterable& r ) {
    using std::begin; using std::end;
    return range(begin(r),end(r));
}

template<class C>
auto except_first( C& c ) {
    auto r = range(c);
    if (r.empty()) return r;
    return r.without_front();
}
```

now we can do:

```
std::vector<int> v = {1,2,3,4};

for (auto i : except_first(v))
    std::cout << i << '\n';
```

and print out

for(:range_expression)部分生成的中间对象将在for循环
开始时已经失效。

Be aware that intermediate objects generated in the `for(:range_expression)` part of the `for` loop will have expired by the time the `for` loop starts.

第12章：文件输入输出

C++文件输入输出是通过streams完成的。关键抽象包括：

std::istream用于读取文本。

std::ostream用于写入文本。

std::streambuf用于读取或写入字符。

格式化输入使用operator>>。

格式化输出使用operator<<。

流使用std::locale，例如，用于格式化的细节以及外部编码与内部编码之间的转换。

关于流的更多内容：`<iostream>` 库

第12.1节：写入文件

写入文件有多种方法。最简单的方法是使用输出文件流（ofstream）配合流插入运算符（<<）：

```
std::ofstream os("foo.txt");
if(os.is_open()){
    os << "Hello World!";
}
```

除了 <<，你也可以使用输出文件流的成员函数 write()：

```
std::ofstream os("foo.txt");
if(os.is_open()){
    char data[] = "Foo";

    // 从 data 写入3个字符 -> "Foo".
    os.write(data, 3);
}
```

写入流后，应始终检查错误状态标志 badbit 是否被设置，因为它表示操作是否失败。可以通过调用输出文件流的成员函数 bad() 来完成此检查：

```
os << "Hello Badbit!"; // 此操作可能因任何原因失败.
if (os.bad())
    // 写入失败！
```

第12.2节：打开文件

打开文件对于所有三种文件流（ifstream、ofstream 和 fstream）来说方法相同。

你可以直接在构造函数中打开文件：

```
std::ifstream ifs("foo.txt"); // ifstream：以只读方式打开文件 "foo.txt".

std::ofstream ofs("foo.txt"); // ofstream：以只写方式打开文件 "foo.txt".
```

Chapter 12: File I/O

C++ file I/O is done via streams. The key abstractions are:

std::istream for reading text.

std::ostream for writing text.

std::streambuf for reading or writing characters.

Formatted input uses operator>>.

Formatted output uses operator<<.

Streams use std::locale, e.g., for details of the formatting and for translation between external encodings and the internal encoding.

More on streams: `<iostream>` Library

Section 12.1: Writing to a file

There are several ways to write to a file. The easiest way is to use an output file stream (ofstream) together with the stream insertion operator (<<):

```
std::ofstream os("foo.txt");
if(os.is_open()){
    os << "Hello World!";
}
```

Instead of <<，你也可以使用输出文件流的成员函数 write()：

```
std::ofstream os("foo.txt");
if(os.is_open()){
    char data[] = "Foo";

    // Writes 3 characters from data -> "Foo".
    os.write(data, 3);
}
```

After writing to a stream, you should always check if error state flag badbit has been set, as it indicates whether the operation failed or not. This can be done by calling the output file stream's member function bad():

```
os << "Hello Badbit!"; // This operation might fail for any reason.
if (os.bad())
    // Failed to write!
```

Section 12.2: Opening a file

Opening a file is done in the same way for all 3 file streams (ifstream, ofstream, and fstream).

You can open the file directly in the constructor:

```
std::ifstream ifs("foo.txt"); // ifstream: Opens file "foo.txt" for reading only.

std::ofstream ofs("foo.txt"); // ofstream: Opens file "foo.txt" for writing only.
```

```
std::fstream iofs("foo.txt"); // fstream : 以读写方式打开文件 "foo.txt"。
```

或者，你可以使用文件流的成员函数 `open()` :

```
std::ifstream ifs; // ifstream : 以只读方式打开文件 "bar.txt"。
ifs.open("bar.txt");

std::ofstream ofs; // ofstream : 仅打开文件 "bar.txt" 以进行写入。
ofs.open("bar.txt");

std::fstream iofs; // fstream : 打开文件 "bar.txt" 以进行读写。
iofs.open("bar.txt");
```

你应该始终检查文件是否成功打开（即使是写入时）。失败原因可能包括：文件不存在、文件没有正确的访问权限、文件已被占用、发生磁盘错误、驱动器断开连接.....

检查可以按如下方式进行：

```
// 尝试读取文件 'foo.txt'。
std::ifstream ifs("foo.txt"); // 注意拼写错误；文件无法打开。

// 检查文件是否成功打开。
if (!ifs.is_open()) {
    // 文件尚未打开；请在此处采取适当的操作。
    throw CustomException(ifs, "文件无法打开");
}
```

当文件路径包含反斜杠（例如，在Windows系统上）时，应正确转义它们：

```
// 在Windows上打开文件 'c:\\\\folder\\\\foo.txt'。
std::ifstream ifs("c:\\\\folder\\\\foo.txt"); // 使用转义的反斜杠
版本 ≥ C++11
```

或者使用原始字符串字面量：

```
// 在Windows上打开文件 'c:\\\\folder\\\\foo.txt'。
std::ifstream ifs(R"(c:\\\\folder\\\\foo.txt)"); // 使用原始字符串字面量
```

或者改用正斜杠：

```
// 在Windows上打开文件 'c:\\\\folder\\\\foo.txt'。
std::ifstream ifs("c:/folder/foo.txt");
版本 ≥ C++11
```

如果你想在Windows上打开路径中包含非ASCII字符的文件，目前可以使用非标准的宽字符路径参数：

```
// 在Windows上打开文件 'пример\\\\foo.txt'。
std::ifstream ifs(LR"(пример\\\\foo.txt)"); // 使用带有原始字面量的宽字符
```

第12.3节：从文件读取

有多种方法可以从文件中读取数据。

如果你知道数据的格式，可以使用流提取运算符 (`>>`)。假设你有一个名为 `foo.txt` 的文件，里面包含以下数据：

```
std::fstream iofs("foo.txt"); // fstream: Opens file "foo.txt" for reading and writing.
```

Alternatively, you can use the file stream's member function `open()`:

```
std::ifstream ifs; // ifstream: Opens file "bar.txt" for reading only.
ifs.open("bar.txt");

std::ofstream ofs; // ofstream: Opens file "bar.txt" for writing only.
ofs.open("bar.txt");

std::fstream iofs; // fstream: Opens file "bar.txt" for reading and writing.
iofs.open("bar.txt");
```

You should **always** check if a file has been opened successfully (even when writing). Failures can include: the file doesn't exist, file hasn't the right access rights, file is already in use, disk errors occurred, drive disconnected ... Checking can be done as follows:

```
// Try to read the file 'foo.txt'.
std::ifstream ifs("foo.txt"); // Note the typo; the file can't be opened.

// Check if the file has been opened successfully.
if (!ifs.is_open()) {
    // The file hasn't been opened; take appropriate actions here.
    throw CustomException(ifs, "File could not be opened");
}
```

When file path contains backslashes (for example, on Windows system) you should properly escape them:

```
// Open the file 'c:\\\\folder\\\\foo.txt' on Windows.
std::ifstream ifs("c:\\\\folder\\\\foo.txt"); // using escaped backslashes
Version ≥ C++11
```

or use raw literal:

```
// Open the file 'c:\\\\folder\\\\foo.txt' on Windows.
std::ifstream ifs(R"(c:\\\\folder\\\\foo.txt)"); // using raw literal
```

or use forward slashes instead:

```
// Open the file 'c:\\\\folder\\\\foo.txt' on Windows.
std::ifstream ifs("c:/folder/foo.txt");
Version ≥ C++11
```

If you want to open file with non-ASCII characters in path on Windows currently you can use **non-standard** wide character path argument:

```
// Open the file 'пример\\\\foo.txt' on Windows.
std::ifstream ifs(LR"(пример\\\\foo.txt)"); // using wide characters with raw literal
```

Section 12.3: Reading from a file

There are several ways to read data from a file.

If you know how the data is formatted, you can use the stream extraction operator (`>>`). Let's assume you have a file named `foo.txt` which contains the following data:

约翰·多 25 4 6 1987
简·多 15 5 24 1976

那么你可以使用以下代码从文件中读取这些数据：

```
// 定义变量。  
std::ifstream is("foo.txt");  
std::string firstname, lastname;  
int age, bmonth, bday, byear;  
  
// 提取名字、姓氏、年龄、出生月份、出生日和出生年份，按此顺序。  
// 注意：'>>' 如果到达文件末尾 (EOF) 或输入数据与输入变量类型不匹配（例如，字符串 "foo" 不能提取到 'int' 变量中），则返回 false。  
  
while (is >> firstname >> lastname >> age >> bmonth >> bday >> byear)  
    // 处理已读取的数据。
```

流提取运算符 `>>` 会提取每个字符，并在遇到无法存储的字符或特殊字符时停止：

- 对于字符串类型，运算符在遇到空白字符（空格）或换行符（`\n`）时停止。
- 对于数字，运算符在遇到非数字字符时停止。

这意味着以下版本的文件 `foo.txt` 也能被前面的代码成功读取：

John
Doe 25
4 6 1987

Jane
Doe
15 5
24
1976

) 运算符只要流没有错误就会返回 `true`。如果流进入

错误状态（例如，因为无法再提取更多数据），则 `bool()` 运算符将返回 `false`。

因此，前面代码中的 `while` 循环将在输入文件读取到末尾后退出。

如果你想将整个文件读取为字符串，可以使用以下代码：

```
// 打开 'foo.txt'。  
std::ifstream is("foo.txt");  
std::string whole_file;  
  
// 将位置设置到文件末尾。  
is.seekg(0, std::ios::end);  
  
// 为文件预留内存。  
whole_file.reserve(is.tellg());  
  
// 设置位置到文件开头。  
is.seekg(0, std::ios::beg);
```

John Doe 25 4 6 1987
Jane Doe 15 5 24 1976

Then you can use the following code to read that data from the file:

```
// Define variables.  
std::ifstream is("foo.txt");  
std::string firstname, lastname;  
int age, bmonth, bday, byear;  
  
// Extract firstname, lastname, age, bday month, bday day, and bday year in that order.  
// Note: '>>' returns false if it reached EOF (end of file) or if the input data doesn't  
// correspond to the type of the input variable (for example, the string "foo" can't be  
// extracted into an 'int' variable).  
while (is >> firstname >> lastname >> age >> bmonth >> bday >> byear)  
    // Process the data that has been read.
```

The stream extraction operator `>>` extracts every character and stops if it finds a character that can't be stored or if it is a special character:

- For string types, the operator stops at a whitespace () or at a newline (`\n`).
- For numbers, the operator stops at a non-number character.

This means that the following version of the file `foo.txt` will also be successfully read by the previous code:

John
Doe 25
4 6 1987

Jane
Doe
15 5
24
1976

The stream extraction operator `>>` always returns the stream given to it. Therefore, multiple operators can be chained together in order to read data consecutively. However, a stream can also be used as a Boolean expression (as shown in the `while` loop in the previous code). This is because the stream classes have a conversion operator for the type `bool`. This `bool()` operator will return `true` as long as the stream has no errors. If a stream goes into an error state (for example, because no more data can be extracted), then the `bool()` operator will return `false`. Therefore, the `while` loop in the previous code will be exited after the input file has been read to its end.

If you wish to read an entire file as a string, you may use the following code:

```
// Opens 'foo.txt'.  
std::ifstream is("foo.txt");  
std::string whole_file;  
  
// Sets position to the end of the file.  
is.seekg(0, std::ios::end);  
  
// Reserves memory for the file.  
whole_file.reserve(is.tellg());  
  
// Sets position to the start of the file.  
is.seekg(0, std::ios::beg);
```

```
// 将 'whole_file' 的内容设置为文件中的所有字符。  
whole_file.assign(std::istreambuf_iterator<char>(is),  
    std::istreambuf_iterator<char>());
```

这段代码为string预留空间，以减少不必要的内存分配。

如果你想逐行读取文件，可以使用函数getline()：

```
std::ifstream is("foo.txt");  
  
// 如果没有更多行，getline函数返回false。  
for (std::string str; std::getline(is, str);) {  
    // 处理已读取的行。  
}
```

如果你想读取固定数量的字符，可以使用流的成员函数read()：

```
std::ifstream is("foo.txt");  
char str[4];  
  
// 从文件中读取4个字符。  
is.read(str, 4);
```

执行读取命令后，您应始终检查错误状态标志failbit是否被设置，因为它表示操作是否失败。可以通过调用文件流的成员函数fail()来完成此操作：

```
is.read(str, 4); // 此操作可能因任何原因失败。  
  
if (is.fail())  
    // 读取失败！
```

第12.4节：打开模式

创建文件流时，可以指定打开模式。打开模式基本上是一个设置，用于控制流如何打开文件。

(所有模式都可以在std::ios命名空间中找到。)

打开模式可以作为文件流构造函数的第二个参数，或其open()成员函数的参数提供：

```
std::ofstream os("foo.txt", std::ios::out | std::ios::trunc);  
  
std::ifstream is;  
is.open("foo.txt", std::ios::in | std::ios::binary);
```

需要注意的是，如果你想设置其他标志，必须显式设置ios::in或ios::out，因为虽然iostream成员有正确的默认值，但它们不会隐式设置这些标志。

如果你没有指定打开模式，则使用以下默认模式：

- ifstream - in
- ofstream - out
- fstream - in 和 out

你可以设计指定的文件打开模式有：

```
// Sets contents of 'whole_file' to all characters in the file.  
whole_file.assign(std::istreambuf_iterator<char>(is),  
    std::istreambuf_iterator<char>());
```

This code reserves space for the string in order to cut down on unneeded memory allocations.

If you want to read a file line by line, you can use the function [getline\(\)](#):

```
std::ifstream is("foo.txt");  
  
// The function getline returns false if there are no more lines.  
for (std::string str; std::getline(is, str);) {  
    // Process the line that has been read.  
}
```

If you want to read a fixed number of characters, you can use the stream's member function [read\(\)](#):

```
std::ifstream is("foo.txt");  
char str[4];  
  
// Read 4 characters from the file.  
is.read(str, 4);
```

After executing a read command, you should always check if the error state flag failbit has been set, as it indicates whether the operation failed or not. This can be done by calling the file stream's member function [fail\(\)](#):

```
is.read(str, 4); // This operation might fail for any reason.  
  
if (is.fail())  
    // Failed to read!
```

Section 12.4: Opening modes

When creating a file stream, you can specify an opening mode. An opening mode is basically a setting to control how the stream opens the file.

(All modes can be found in the std::ios namespace.)

An opening mode can be provided as second parameter to the constructor of a file stream or to its open() member function:

```
std::ofstream os("foo.txt", std::ios::out | std::ios::trunc);  
  
std::ifstream is;  
is.open("foo.txt", std::ios::in | std::ios::binary);
```

It is to be noted that you have to set ios::in or ios::out if you want to set other flags as they are not implicitly set by the iostream members although they have a correct default value.

If you don't specify an opening mode, then the following default modes are used:

- ifstream - in
- ofstream - out
- fstream - in and out

The file opening modes that you may specify by design are:

模式	含义	用于	描述
app	append	Output	在文件末尾追加数据。
binary	binary	输入/输出	输入和输出以二进制进行。
in	输入	输入	打开文件以进行读取。
out	输出	输出	打开文件以进行写入。
trunc	truncate	输入/输出	打开文件时清除文件内容。
ate	文件末尾	输入	打开时跳转到文件末尾。

注意：设置二进制模式可以让数据被精确地读写；不设置则允许将换行符'\\n'字符转换为/从特定平台的行尾序列。

例如，在Windows上行尾序列是CRLF ("\\r\\n")。

写入："\\n" => "\\r"读取

：“\\r” => “”

第12.5节：将ASCII文件读入std::string

```
std::ifstream f("file.txt");

if (f)
{
    std::stringstream buffer;
    buffer << f.rdbuf();
    f.close();

    // "file.txt"的内容已存于字符串`buffer.str()`中
}
```

rdbuf()方法返回一个指向 streambuf 的指针，该指针可以通过 stringstream::operator<< 成员函数推入缓冲区 buffer 中。

另一种方法（由 Scott Meyers 在《Effective STL》中推广）是：

```
std::ifstream f("file.txt");

if (f)
{
    std::string str((std::istreambuf_iterator<char>(f),
                    std::istreambuf_iterator<char>()));

    // 对 `str` 进行操作...
}
```

这种方法的优点是代码量少（并且允许直接将文件读取到任何 STL 容器中，而不仅仅是字符串），但对于大文件来说可能较慢。

注意：传递给字符串构造函数的第一个参数周围的额外括号是必需的，以防止“最令人困惑的解析”问题。

最后但同样重要的是：

```
std::ifstream f("file.txt");

if (f)
{
    f.seekg(0, std::ios::end);
```

Mode	Meaning	For	Description
app	append	Output	Appends data at the end of the file.
binary	binary	Input/Output	Input and output is done in binary.
in	input	Input	Opens the file for reading.
out	output	Output	Opens the file for writing.
trunc	truncate	Input/Output	Removes contents of the file when opening.
ate	at end	Input	Goes to the end of the file when opening.

Note: Setting the binary mode lets the data be read/written exactly as-is; not setting it enables the translation of the newline '\\n' character to/from a platform specific end of line sequence.

For example on Windows the end of line sequence is CRLF ("\\r\\n").

Write: "\\n" => "\\r\\n"

Read: "\\r\\n" => "\\n"

Section 12.5: Reading an ASCII file into a std::string

```
std::ifstream f("file.txt");

if (f)
{
    std::stringstream buffer;
    buffer << f.rdbuf();
    f.close();

    // The content of "file.txt" is available in the string `buffer.str()`
}
```

The rdbuf() method returns a pointer to a streambuf that can be pushed into buffer via the stringstream::operator<< member function.

Another possibility (popularized in Effective STL by Scott Meyers) is:

```
std::ifstream f("file.txt");

if (f)
{
    std::string str((std::istreambuf_iterator<char>(f),
                     std::istreambuf_iterator<char>()));

    // Operations on `str`...
}
```

This is nice because requires little code (and allows reading a file directly into any STL container, not only strings) but can be slow for big files.

NOTE: the extra parentheses around the first argument to the string constructor are essential to prevent the most vexing parse problem.

Last but not least:

```
std::ifstream f("file.txt");

if (f)
{
    f.seekg(0, std::ios::end);
```

```

const auto size = f.tellg();

std::string str(size, ' ');
f.seekg(0);
f.read(&str[0], size);
f.close();

// 对 `str` 进行操作...
}

```

这可能是三种提议中最快的选项。

第12.6节：使用非标准区域设置写文件

如果需要使用不同于默认的区域设置写文件，可以使用 `std::locale` 和 `std::basic_ios::imbue()` 来针对特定文件流进行设置：

使用指导：

- 应始终在打开文件之前将区域设置应用到流上。
- 一旦流被赋予区域设置，就不应更改该区域设置。

限制原因：如果当前区域设置不是状态独立的或未指向文件开头，则给文件流赋予区域设置会导致未定义行为。

UTF-8流（及其他）不是状态独立的。此外，带有UTF-8区域设置的文件流在打开时可能会尝试读取文件中的BOM标记；因此，仅打开文件就可能从文件中读取字符，导致流不在文件开头。

```

#include <iostream>
#include <fstream>
#include <locale>

int main()
{
    std::cout << "用户首选的区域设置是 "
        << std::locale("").name().c_str() << std::endl;

    // 使用用户首选的区域设置写入浮点数值。
    std::ofstream ofs1;
    ofs1.imbue(std::locale(""));
    ofs1.open("file1.txt");
    ofs1 << 78123.456 << std::endl;

    // 使用特定的区域设置（名称依赖于系统）
    std::ofstream ofs2;
    ofs2.imbue(std::locale("en_US.UTF-8"));
    ofs2.open("file2.txt");
    ofs2 << 78123.456 << std::endl;

    // 切换到经典的 "C" 区域设置
    std::ofstream ofs3;
    ofs3.imbue(std::locale::classic());
    ofs3.open("file3.txt");
    ofs3 << 78123.456 << std::endl;
}

```

如果程序使用了不同的默认区域设置，显式切换到经典的 "C" 区域设置是有用的，且你想要

```

const auto size = f.tellg();

std::string str(size, ' ');
f.seekg(0);
f.read(&str[0], size);
f.close();

// Operations on `str`...
}

```

which is probably the fastest option (among the three proposed).

Section 12.6: Writing files with non-standard locale settings

If you need to write a file using different locale settings to the default, you can use `std::locale` and `std::basic_ios::imbue()` to do that for a specific file stream:

Guidance for use:

- You should always apply a local to a stream before opening the file.
- Once the stream has been imbued you should not change the locale.

Reasons for Restrictions: Imbuing a file stream with a locale has undefined behavior if the current locale is not state independent or not pointing at the beginning of the file.

UTF-8 streams (and others) are not state independent. Also a file stream with a UTF-8 locale may try and read the BOM marker from the file when it is opened; so just opening the file may read characters from the file and it will not be at the beginning.

```

#include <iostream>
#include <fstream>
#include <locale>

int main()
{
    std::cout << "User-preferred locale setting is "
        << std::locale("").name().c_str() << std::endl;

    // Write a floating-point value using the user's preferred locale.
    std::ofstream ofs1;
    ofs1.imbue(std::locale(""));
    ofs1.open("file1.txt");
    ofs1 << 78123.456 << std::endl;

    // Use a specific locale (names are system-dependent)
    std::ofstream ofs2;
    ofs2.imbue(std::locale("en_US.UTF-8"));
    ofs2.open("file2.txt");
    ofs2 << 78123.456 << std::endl;

    // Switch to the classic "C" locale
    std::ofstream ofs3;
    ofs3.imbue(std::locale::classic());
    ofs3.open("file3.txt");
    ofs3 << 78123.456 << std::endl;
}

```

Explicitly switching to the classic "C" locale is useful if your program uses a different default locale and you want to

确保读取和写入文件的固定标准。使用首选区域设置为“C”，示例写入

```
78,123.456  
78,123.456  
78123.456
```

例如，如果首选语言环境是德语，因此使用不同的数字格式，示例写道

```
78 123,456  
78,123.456  
78123.456
```

(注意第一行中的小数逗号)。

第12.7节：在循环条件中检查文件结尾，是不良做法吗？

`eof` 仅在读取到文件末尾后返回`true`。它不表示下一次读取将是流的结尾。

```
while (!f.eof())  
{  
    // 一切正常  
  
    f >> buffer;  
  
    // 如果只有现在才设置了 eof / fail 位怎么办?  
  
    /* 使用 `buffer` */  
}
```

你可以正确地写成：

```
while (!f.eof())  
{  
    f >> buffer >> std::ws;  
  
    if (f.fail())  
        break;  
  
    /* 使用 `buffer` */  
}
```

但是

```
while (f >> buffer)  
{  
    /* 使用 `buffer` */  
}
```

更简单且不易出错。

更多参考资料：

- [std::ws](#): 从输入流中丢弃前导空白字符
- [std::basic_ios::fail](#): 如果关联的流发生错误，则返回`true`

ensure a fixed standard for reading and writing files. With a "C" preferred locale, the example writes

```
78,123.456  
78,123.456  
78123.456
```

If, for example, the preferred locale is German and hence uses a different number format, the example writes

```
78 123,456  
78,123.456  
78123.456
```

(note the decimal comma in the first line).

Section 12.7: Checking end of file inside a loop condition, bad practice?

`eof` returns `true` only **after** reading the end of file. It does NOT indicate that the next read will be the end of stream.

```
while (!f.eof())  
{  
    // Everything is OK  
  
    f >> buffer;  
  
    // What if *only* now the eof / fail bit is set?  
  
    /* Use `buffer` */  
}
```

You could correctly write:

```
while (!f.eof())  
{  
    f >> buffer >> std::ws;  
  
    if (f.fail())  
        break;  
  
    /* Use `buffer` */  
}
```

but

```
while (f >> buffer)  
{  
    /* Use `buffer` */  
}
```

is simpler and less error prone.

Further references:

- [std::ws](#): discards leading whitespace from an input stream
- [std::basic_ios::fail](#): returns `true` if an error has occurred on the associated stream

第12.8节：刷新流

文件流默认是缓冲的，许多其他类型的流也是如此。这意味着对流的写入可能不会立即导致底层文件发生变化。为了强制所有缓冲的写入立即生效，你可以刷新流。你可以直接调用flush()方法，也可以通过std::flush流操纵器来实现：

```
std::ofstream os("foo.txt");
os << "Hello World!" << std::flush;

char data[3] = "Foo";
os.write(data, 3);
os.flush();
```

有一个流操纵器std::endl，它将写入换行符和刷新流结合在一起：

```
// 以下两行代码效果相同
os << "Hello World!" << std::flush;
os << "Hello world!" << std::endl;
```

缓冲可以提高写入流的性能。因此，频繁写入的应用程序应避免不必要的刷新。相反，如果I/O操作不频繁，应用程序应考虑刷新，以避免数据滞留在流对象中。

第12.9节：将文件读入容器

下面的示例中，我们使用std::string和operator>>从文件中读取项目。

```
std::ifstream file("file3.txt");

std::vector<std::string> v;

std::string s;
while(file >> s) // 一直读取直到文件结束
{
    v.push_back(s);
}
```

在上面的例子中，我们只是简单地通过operator>>一次读取一个“项”来遍历文件。这个效果也可以通过std::istream_iterator实现，它是一个输入迭代器，可以一次从流中读取一个“项”。此外，大多数容器都可以通过两个迭代器构造，因此我们可以将上述代码简化为：

```
std::ifstream file("file3.txt");

std::vector<std::string> v(std::istream_iterator<std::string>(file),
                         std::istream_iterator<std::string>{});
```

我们可以通过简单地指定想要读取的对象类型作为std::istream_iterator的模板参数，来扩展读取任何对象类型。因此，我们可以简单地扩展上述代码来读取行（而不是单词），如下所示：

```
// 不幸的是，没有内置类型可以使用 >> 读取整行// 所以这里我们构建了一个简单的辅助
// 类来实现这一点。当在字符串上下文中使用时，它会转换回字符串。
struct Line
```

Section 12.8: Flushing a stream

File streams are buffered by default, as are many other types of streams. This means that writes to the stream may not cause the underlying file to change immediately. In order to force all buffered writes to take place immediately, you can *flush* the stream. You can do this either directly by invoking the `flush()` method or through the `std::flush` stream manipulator:

```
std::ofstream os("foo.txt");
os << "Hello World!" << std::flush;

char data[3] = "Foo";
os.write(data, 3);
os.flush();
```

There is a stream manipulator `std::endl` that combines writing a newline with flushing the stream:

```
// Both following lines do the same thing
os << "Hello World!\n" << std::flush;
os << "Hello world!" << std::endl;
```

Buffering can improve the performance of writing to a stream. Therefore, applications that do a lot of writing should avoid flushing unnecessarily. Contrary, if I/O is done infrequently, applications should consider flushing frequently in order to avoid data getting stuck in the stream object.

Section 12.9: Reading a file into a container

In the example below we use `std::string` and `operator>>` to read items from the file.

```
std::ifstream file("file3.txt");

std::vector<std::string> v;

std::string s;
while(file >> s) // keep reading until we run out
{
    v.push_back(s);
}
```

In the above example we are simply iterating through the file reading one "item" at a time using `operator>>`. This same effect can be achieved using the `std::istream_iterator` which is an input iterator that reads one "item" at a time from the stream. Also most containers can be constructed using two iterators so we can simplify the above code to:

```
std::ifstream file("file3.txt");

std::vector<std::string> v(std::istream_iterator<std::string>(file),
                         std::istream_iterator<std::string>{});
```

We can extend this to read any object types we like by simply specifying the object we want to read as the template parameter to the `std::istream_iterator`. Thus we can simply extend the above to read lines (rather than words) like this:

```
// Unfortunately there is no built in type that reads line using >>
// So here we build a simple helper class to do it. That will convert
// back to a string when used in string context.
struct Line
```

```

{
    // 在这里存储数据
    std::string data;
    // 将对象转换为字符串
    operator std::string const&() const {return data;}
    // 从流中读取一行。
    friend std::istream& operator>>(std::istream& stream, Line& line)
    {
        return std::getline(stream, line.data);
    }
};

std::ifstream file("file3.txt");

// 将文件的行读取到容器中。
std::vector<std::string> v(std::istream_iterator<Line>{file},
                           std::istream_iterator<Line>{});

```

第12.10节：复制文件

```

std::ifstream src("source_filename", std::ios::binary);
std::ofstream dst("dest_filename", std::ios::binary);
dst << src.rdbuf();

```

版本 ≥ C++17

在C++17中，复制文件的标准方式是包含`<filesystem>`头文件并使用`copy_file`：

```
std::filesystem::copy_file("source_filename", "dest_filename");
```

`filesystem`库最初作为`boost.filesystem`开发，最终在C++17中合并进ISO C++标准。

第12.11节：关闭文件

在C++中，显式关闭文件很少是必要的，因为文件流会在其析构函数中自动关闭关联的文件。然而，你应该尽量限制文件流对象的生命周期，以避免文件句柄被保持打开时间过长。例如，可以通过将所有文件操作放入一个独立的作用域（{}）来实现：

```

std::string const prepared_data = prepare_data();
{
    // 打开一个文件用于写入。
    std::ofstream output("foo.txt");

    // 写入数据。
    output << prepared_data;
} // ofstream将在此处超出作用域。
// 其析构函数将负责正确关闭文件。

```

只有当你想重复使用同一个`fstream`对象，但又不想在两次使用之间保持文件打开时，才需要显式调用`close()`：

```

// 第一次打开文件 "foo.txt"。
std::ofstream output("foo.txt");

// 从某处获取一些要写入的数据。
std::string const prepared_data = prepare_data();

```

```

{
    // Store data here
    std::string data;
    // Convert object to string
    operator std::string const&() const {return data;}
    // Read a line from a stream.
    friend std::istream& operator>>(std::istream& stream, Line& line)
    {
        return std::getline(stream, line.data);
    }
};

std::ifstream file("file3.txt");

// Read the lines of a file into a container.
std::vector<std::string> v(std::istream_iterator<Line>{file},
                           std::istream_iterator<Line>{});

```

Section 12.10: Copying a file

```

std::ifstream src("source_filename", std::ios::binary);
std::ofstream dst("dest_filename", std::ios::binary);
dst << src.rdbuf();

```

Version ≥ C++17

With C++17 the standard way to copy a file is including the `<filesystem>` header and using `copy_file`:

```
std::filesystem::copy_file("source_filename", "dest_filename");
```

The `filesystem` library was originally developed as `boost.filesystem` and finally merged to ISO C++ as of C++17.

Section 12.11: Closing a file

Explicitly closing a file is rarely necessary in C++, as a file stream will automatically close its associated file in its destructor. However, you should try to limit the lifetime of a file stream object, so that it does not keep the file handle open longer than necessary. For example, this can be done by putting all file operations into an own scope {}:

```

std::string const prepared_data = prepare_data();
{
    // Open a file for writing.
    std::ofstream output("foo.txt");

    // Write data.
    output << prepared_data;
} // The ofstream will go out of scope here.
// Its destructor will take care of closing the file properly.

```

Calling `close()` explicitly is only necessary if you want to reuse the same `fstream` object later, but don't want to keep the file open in between:

```

// Open the file "foo.txt" for the first time.
std::ofstream output("foo.txt");

// Get some data to write from somewhere.
std::string const prepared_data = prepare_data();

```

```

// 将数据写入文件 "foo.txt".
output << prepared_data;

// 关闭文件 "foo.txt".
output.close();

// 准备数据可能需要很长时间。因此，在我们真正能够写入数据之前，  

// 不会打开输出文件流。
std::string const more_prepared_data = prepare_complex_data();

// 一旦准备好写入，第二次打开文件 "foo.txt".
output.open("foo.txt");

// 将数据写入文件 "foo.txt".
output << more_prepared_data;

// 再次关闭文件 "foo.txt".
output.close();

```

第12.12节：从格式化文本文件读取`struct`

版本 ≥ C++11

```

struct info_type
{
    std::string name;
    int age;
    float height;

    // 我们定义了一个operator>>的重载作为友元函数，  

    // 使其可以特权访问私有数据成员
    friend std::istream& operator>>(std::istream& is, info_type& info)
    {
        // 跳过空白字符
        is >> std::ws;
        std::getline(is, info.name);
        is >> info.age;
        is >> info.height;
        return is;
    }
};

void func4()
{
    auto file = std::ifstream("file4.txt");

    std::vector<info_type> v;

    for(info_type info; file >> info) // 持续读取直到文件结束
    {
        // 我们只有在读取成功时才会到这里
        v.push_back(info);
    }

    for(auto const& info: v)
    {
        std::cout << " name: " << info.name << ";      std::cout <<
        " age: " << info.age << " years" << ";      std::cout << "height: " << inf
        o.height << "lbs" << ";      std::cout << ";
    }
}

```

```

// Write data to the file "foo.txt".
output << prepared_data;

// Close the file "foo.txt".
output.close();

// Preparing data might take a long time. Therefore, we don't open the output file stream
// before we actually can write some data to it.
std::string const more_prepared_data = prepare_complex_data();

// Open the file "foo.txt" for the second time once we are ready for writing.
output.open("foo.txt");

// Write the data to the file "foo.txt".
output << more_prepared_data;

// Close the file "foo.txt" once again.
output.close();

```

Section 12.12: Reading a `struct` from a formatted text file

```

Version ≥ C++11
struct info_type
{
    std::string name;
    int age;
    float height;

    // we define an overload of operator>> as a friend function which
    // gives in privileged access to private data members
    friend std::istream& operator>>(std::istream& is, info_type& info)
    {
        // skip whitespace
        is >> std::ws;
        std::getline(is, info.name);
        is >> info.age;
        is >> info.height;
        return is;
    }
};

void func4()
{
    auto file = std::ifstream("file4.txt");

    std::vector<info_type> v;

    for(info_type info; file >> info) // keep reading until we run out
    {
        // we only get here if the read succeeded
        v.push_back(info);
    }

    for(auto const& info: v)
    {
        std::cout << " name: " << info.name << '\n';
        std::cout << " age: " << info.age << " years" << '\n';
        std::cout << "height: " << info.height << "lbs" << '\n';
        std::cout << '\n';
    }
}

```

file4.txt

```
沃格尔·瓦比特  
2  
6.2  
比尔博·巴金斯  
111  
81.3  
玛丽·波平斯  
29  
154.8
```

输出：

```
姓名：沃格·瓦比特  
年龄：2岁  
体重：6.2磅
```

```
姓名：比尔博·巴金斯  
年龄：111岁  
体重：81.3磅
```

```
姓名：玛丽·波平斯  
年龄：29岁  
体重：154.8磅
```

file4.txt

```
Wogger Wabbit  
2  
6.2  
Bilbo Baggins  
111  
81.3  
Mary Poppins  
29  
154.8
```

Output:

```
name: Wogger Wabbit  
age: 2 years  
height: 6.2lbs
```

```
name: Bilbo Baggins  
age: 111 years  
height: 81.3lbs
```

```
name: Mary Poppins  
age: 29 years  
height: 154.8lbs
```

第13章：C++流

第13.1节：字符串流

`std::ostringstream` 是一个类，其对象看起来像输出流（即可以通过 `operator<<` 写入它们），但实际上存储写入结果，并以流的形式提供这些结果。

考虑以下简短代码：

```
#include <sstream>
#include <string>

using namespace std;

int main()
{
    ostringstream ss;
    ss << "万物的答案是 " << 42;
}
```

这一行

```
ostringstream ss;
```

创建了这样一个对象。这个对象首先像普通流一样被操作：

```
ss << "万物的答案是 " << 42;
```

之后，可以通过如下方式获取生成的流内容：

```
const string result = ss.str();
```

(字符串result将等于"万物的答案是 42")。

这主要在我们有一个定义了流序列化的类，并且想要它的字符串形式时非常有用。例如，假设我们有一个类

```
class foo
{
    // 这里是各种内容。
};

ostream &operator<<(ostream &os, const foo &f);
```

要获取foo对象的字符串表示，

```
foo f;
```

我们可以使用

```
ostringstream ss;
ss << f;
const string result = ss.str();
```

Chapter 13: C++ Streams

Section 13.1: String streams

`std::ostringstream` is a class whose objects look like an output stream (that is, you can write to them via `operator<<`), but actually store the writing results, and provide them in the form of a stream.

Consider the following short code:

```
#include <sstream>
#include <string>

using namespace std;

int main()
{
    ostringstream ss;
    ss << "the answer to everything is " << 42;
    const string result = ss.str();
}
```

The line

```
ostringstream ss;
```

creates such an object. This object is first manipulated like a regular stream:

```
ss << "the answer to everything is " << 42;
```

Following that, though, the resulting stream can be obtained like this:

```
const string result = ss.str();
```

(the string result will be equal to "the answer to everything is 42").

This is mainly useful when we have a class for which stream serialization has been defined, and for which we want a string form. For example, suppose we have some class

```
class foo
{
    // All sort of stuff here.
};

ostream &operator<<(ostream &os, const foo &f);
```

To get the string representation of a foo object,

```
foo f;
```

we could use

```
ostringstream ss;
ss << f;
const string result = ss.str();
```

然后result包含了foo对象的字符串表示。

第13.2节：使用iostream打印集合

基本打印

`std::ostream_iterator` 允许将 STL 容器的内容打印到任何输出流，而无需显式循环。`std::ostream_iterator` 构造函数的第二个参数设置分隔符。例如，以下代码：

```
std::vector<int> v = {1,2,3,4};  
std::copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " ! "));
```

将打印

```
1 ! 2 ! 3 ! 4 !
```

隐式类型转换

`std::ostream_iterator` 允许隐式转换容器内容的类型。例如，我们调整 `std::cout` 以打印小数点后三位的浮点数：

```
std::cout << std::setprecision(3);  
std::fixed(std::cout);
```

并实例化`std::ostream_iterator`为`float`类型，而容器中的值仍然是`int`类型：

```
std::vector<int> v = {1,2,3,4};  
std::copy(v.begin(), v.end(), std::ostream_iterator<float>(std::cout, " ! "));
```

因此，上述代码输出结果为

```
1.000 ! 2.000 ! 3.000 ! 4.000 !
```

尽管`std::vector`中保存的是`int`类型。

生成与转换

`std::generate`、`std::generate_n`和`std::transform`函数提供了一个非常强大的工具，用于即时数据操作。例如，假设有一个向量：

```
std::vector<int> v = {1,2,3,4,8,16};
```

我们可以轻松地打印每个元素“x是否为偶数”的布尔值：

```
std::boolalpha(std::cout); // 以字母形式打印布尔值  
std::transform(v.begin(), v.end(), std::ostream_iterator<bool>(std::cout, " "),  
[](int val) {  
    return (val % 2) == 0;  
});
```

或打印平方元素：

```
std::transform(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " "),  
[](int val) {  
    return val * val;  
});
```

Then result contains the string representation of the foo object.

Section 13.2: Printing collections with iostream

Basic printing

`std::ostream_iterator` allows to print contents of an STL container to any output stream without explicit loops. The second argument of `std::ostream_iterator` constructor sets the delimiter. For example, the following code:

```
std::vector<int> v = {1,2,3,4};  
std::copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " ! "));
```

will print

```
1 ! 2 ! 3 ! 4 !
```

Implicit type cast

`std::ostream_iterator` allows to cast container's content type implicitly. For example, let's tune `std::cout` to print floating-point values with 3 digits after decimal point:

```
std::cout << std::setprecision(3);  
std::fixed(std::cout);
```

and instantiate `std::ostream_iterator` with `float`, while the contained values remain `int`:

```
std::vector<int> v = {1,2,3,4};  
std::copy(v.begin(), v.end(), std::ostream_iterator<float>(std::cout, " ! "));
```

so the code above yields

```
1.000 ! 2.000 ! 3.000 ! 4.000 !
```

despite `std::vector` holds `ints`.

Generation and transformation

`std::generate`, `std::generate_n` and `std::transform` functions provide a very powerful tool for on-the-fly data manipulation. For example, having a vector:

```
std::vector<int> v = {1,2,3,4,8,16};
```

we can easily print boolean value of "x is even" statement for each element:

```
std::boolalpha(std::cout); // print booleans alphabetically  
std::transform(v.begin(), v.end(), std::ostream_iterator<bool>(std::cout, " "),  
[](int val) {  
    return (val % 2) == 0;  
});
```

or print the squared element:

```
std::transform(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " "),  
[](int val) {  
    return val * val;  
});
```

});

打印 N 个以空格分隔的随机数：

```
const int N = 10;
std::generate_n(std::ostream_iterator<int>(std::cout, " "), N, std::rand);
```

数组

正如读取文本文件部分所述，几乎所有这些考虑都可以应用于原生数组。例如，让我们打印一个原生数组中的平方值：

```
int v[] = {1,2,3,4,8,16};
std::transform(v, std::end(v), std::ostream_iterator<int>(std::cout, " "),
[](int val) {
    return val * val;
});
```

};

Printing N space-delimited random numbers:

```
const int N = 10;
std::generate_n(std::ostream_iterator<int>(std::cout, " "), N, std::rand);
```

Arrays

As in the section about reading text files, almost all these considerations may be applied to native arrays. For example, let's print squared values from a native array:

```
int v[] = {1,2,3,4,8,16};
std::transform(v, std::end(v), std::ostream_iterator<int>(std::cout, " "),
[](int val) {
    return val * val;
});
```

第14章：流操纵器

操纵器是特殊的辅助函数，帮助使用operator `>>`或operator `<<`来控制输入输出流。

它们都可以通过`#include <iomanip>`来包含。

第14.1节：流操纵器

`std::boolalpha`和`std::noboolalpha`——在布尔值的文本表示和数字表示之间切换。

```
std::cout << std::boolalpha << 1;
// 输出: true

std::cout << std::noboolalpha << false;
// 输出: 0

bool boolValue;
std::cin >> std::boolalpha >> boolValue;
std::cout << "值 \" " << std::boolalpha << boolValue
      << "\" 被解析为 " << std::noboolalpha << boolValue;
// 输入: true
// 输出: 值 "true" 被解析为 0
```

`std::showbase` 和 `std::noshowbase` - 控制是否使用表示数字进制的前缀。

`std::dec` (十进制)、`std::hex` (十六进制) 和 `std::oct` (八进制) - 用于改变整数的进制。

```
#include <sstream>

std::cout << std::dec << 29 << ' - '
      << std::hex << 29 << ' - '
      << std::showbase << std::oct << 29 << ' - ' << std::n
      oshowbase << 29 " ;
int number;
std::istringstream("3B") >> std::hex >> number;
std::cout << std::dec << 10;
// 输出: 22 - 1D - 35 - 035
// 59
```

默认值是 `std::ios_base::noshowbase` 和 `std::ios_base::dec`。

如果你想了解更多关于`std::istringstream`的内容，请查看`<sstream>`头文件。

`std::uppercase`和`std::nouppercase`——控制浮点数和十六进制整数输出时是否使用大写字符。对输入流无影响。

```
std::cout << std::hex << std::showbase
      << "无大写的0x2a: " << std::nouppercase << 0x2a << " << "带大写的1e-10: " <<
      std::uppercase << 1e-10 << " }

// 输出: 无大写的0x2a: 0x2a
// 带大写的1e-10: 1E-10
```

Chapter 14: Stream manipulators

Manipulators are special helper functions that help controlling input and output streams using operator `>>` or operator `<<`.

They all can be included by `#include <iomanip>`.

Section 14.1: Stream manipulators

`std::boolalpha` and `std::noboolalpha` - switch between textual and numeric representation of booleans.

```
std::cout << std::boolalpha << 1;
// Output: true

std::cout << std::noboolalpha << false;
// Output: 0

bool boolValue;
std::cin >> std::boolalpha >> boolValue;
std::cout << "Value \" " << std::boolalpha << boolValue
      << "\" was parsed as " << std::noboolalpha << boolValue;
// Input: true
// Output: Value "true" was parsed as 0
```

`std::showbase` and `std::noshowbase` - control whether prefix indicating numeric base is used.

```
std::dec (decimal), std::hex (hexadecimal) and std::oct (octal) - are used for changing base for integers.

#include <sstream>

std::cout << std::dec << 29 << ' - '
      << std::hex << 29 << ' - '
      << std::showbase << std::oct << 29 << ' - '
      << std::noshowbase << 29 " \n";
int number;
std::istringstream("3B") >> std::hex >> number;
std::cout << std::dec << 10;
// Output: 22 - 1D - 35 - 035
// 59
```

Default values are `std::ios_base::noshowbase` and `std::ios_base::dec`.

If you want to see more about `std::istringstream` check out the `<sstream>` header.

`std::uppercase` and `std::nouppercase` - control whether uppercase characters are used in floating-point and hexadecimal integer output. Have no effect on input streams.

```
std::cout << std::hex << std::showbase
      << "0x2a with nouppercase: " << std::nouppercase << 0x2a << '\n'
      << "1e-10 with uppercase: " << std::uppercase << 1e-10 << '\n'
}
// Output: 0x2a with nouppercase: 0x2a
// 1e-10 with uppercase: 1E-10
```

默认是std::nouppercase。

std::setw(n)——将下一个输入/输出字段的宽度设置为恰好为 n。

当调用某些函数时，宽度属性 n 会被重置为 0（完整列表见此处）。

```
std::cout << "无 setw:" << 51 << "<< \"setw(7)"  
    << std::setw(7) << 51 << "<< \"setw(7), 更多输出: "  
    << 13  
    << std::setw(7) << std::setfill('*') << 67 << ' ' << 94 << ";"  
  
char* input = "Hello, world!";  
char arr[10];  
std::cin >> std::setw(6) >> arr;  
std::cout << "Input from \"Hello, world!\" with setw(6) gave \" " << arr << "\"// 输出: 51  
  
// setw(7):      51  
// setw(7), more output: 13*****67 94  
  
// 输入: Hello, world!  
// 输出: Input from "Hello, world!" with setw(6) gave "Hello"
```

默认值是 std::setw(0)。

std::left, std::right 和 std::internal - 通过设置

std::ios_base::adjustfield 为 std::ios_base::left, std::ios_base::right 和 std::ios_base::internal 分别修改填充字符的默认位置。 std::left 和 std::right 适用于任何输出, std::internal 适用于整数、浮点数和货币输出。对输入流无影响。

```
#include <iostream>  
...  
  
std::cout.imbue(std::locale("en_US.utf8"));std::cout <<  
  
std::left << std::showbase << std::setfill('*') << "flt: " << std::setw(15) << -  
    9.87 << "<< \"hex: " << std::setw(15) << 41 << "<< "  
    "$: " << std::setw(15) << std::put_money(367, false)  
    << "<< \"usd: " << std::setw(15) << std::put_money(367, true) << "<< \"usd: "  
    << std::setw(15)  
    << std::setfill(' ') << std::put_money(367, false) << ";// 输出:  
  
// 浮点数: -9.87*****  
// 十六进制: 41*****  
// $: $3.67*****  
// 美元: USD *3.67*****  
// 美元: $3.67  
  
std::cout << std::internal << std::showbase << std::setfill('*') << "浮点数: " << std::setw(15) << -9.87 << "  
    << "十六进制: " << std::setw(15) << 41 << "<< "  
    "$: " << std::setw(15) << std::put_money(367, false) << "<< \"美元: " << std::setw(15)  
    << std::put_money(367, true) << "<< \"美元: " << std::setw(15)  
    << std::setfill(' ') << std::put_money(367, true) << ";
```

Default is std::nouppercase.

std::setw(n) - changes the width of the next input/output field to exactly n.

The width property n is resetting to 0 when some functions are called (full list is [here](#)).

```
std::cout << "no setw:" << 51 << '\n'  
    << "setw(7): " << std::setw(7) << 51 << '\n'  
    << "setw(7), more output: " << 13  
    << std::setw(7) << std::setfill('*') << 67 << ' ' << 94 << '\n';  
  
char* input = "Hello, world!";  
char arr[10];  
std::cin >> std::setw(6) >> arr;  
std::cout << "Input from \"Hello, world!\" with setw(6) gave \" " << arr << "\"\\n";  
  
// Output: 51  
// setw(7):      51  
// setw(7), more output: 13*****67 94  
  
// Input: Hello, world!  
// Output: Input from "Hello, world!" with setw(6) gave "Hello"
```

Default is std::setw(0)。

std::left, std::right 和 std::internal - modify the default position of the fill characters by setting std::ios_base::adjustfield to std::ios_base::left, std::ios_base::right and std::ios_base::internal correspondingly. std::left and std::right apply to any output, std::internal - for integer, floating-point and monetary output. Have no effect on input streams.

```
#include <iostream>  
...  
  
std::cout.imbue(std::locale("en_US.utf8"));  
  
std::cout << std::left << std::showbase << std::setfill('*')  
    << "flt: " << std::setw(15) << -9.87 << '\n'  
    << "hex: " << std::setw(15) << 41 << '\n'  
    << "$: " << std::setw(15) << std::put_money(367, false) << '\n'  
    << "usd: " << std::setw(15) << std::put_money(367, true) << '\n'  
    << "usd: " << std::setw(15)  
    << std::setfill(' ') << std::put_money(367, false) << '\n';  
  
// Output:  
// flt: -9.87*****  
// hex: 41*****  
// $: $3.67*****  
// usd: USD *3.67*****  
// usd: $3.67  
  
std::cout << std::internal << std::showbase << std::setfill('*')  
    << "flt: " << std::setw(15) << -9.87 << '\n'  
    << "hex: " << std::setw(15) << 41 << '\n'  
    << "$: " << std::setw(15) << std::put_money(367, false) << '\n'  
    << "usd: " << std::setw(15) << std::put_money(367, true) << '\n'  
    << "usd: " << std::setw(15)  
    << std::setfill(' ') << std::put_money(367, true) << '\n';
```

```

// 输出:
// flt: -*****9.87
// hex: *****41
// $: $3.67*****
// usd: USD *****3.67
// usd: USD 3.67

std::cout << std::right << std::showbase << std::setfill('*') << "flt: " << std::
    setw(15) << -9.87 << "
<< "十六进制: " << std::setw(15) << 41 << "
$: " << std::setw(15) << std::put_money(367, false) << "
<< "美元: " << std::setw(15)
(15) << std::put_money(367, true) << "
<< "美元: " << std::setw(15)

<< std::setfill(' ') << std::put_money(367, true) << "; // Output:

// flt: *****-9.87
// hex: *****41
// $: *****$3.67
// usd: *****USD *3.67
// usd: USD 3.67

```

默认是 `std::left`。

[std::fixed, std::scientific, std::hexfloat \[C++11\]](#) 和 [std::defaultfloat \[C++11\]](#) - 改变浮点数输入/输出的格式。

`std::fixed` 设置 `std::ios_base::floatfield` 为 `std::ios_base::fixed`,
`std::scientific` 设置为 `std::ios_base::scientific`,
`std::hexfloat` - 转换为 `std::ios_base::fixed | std::ios_base::scientific` 和
`std::defaultfloat` - 转换为 `std::ios_base::fmtflags(0)`。

fmtflags

```

#include <iostream>
...

std::cout << "
    << "数字 0.07 的固定格式:      " << std::fixed << 0.01 << "
    << "数字 0.07 的科学
计数法格式: " << std::scientific << 0.01 << "
    << "数字 0.07 的十六进制浮点格式:   " << std::hexfloat << 0.01 <<
    "double f;

std::istringstream is("0x1P-1022");
double f = std::strtod(is.str().c_str(), NULL); std::cout <<
将 0x1P-1022 解析为十六进制得到 " << f << "; // 输出:

// 数字 0.01 的固定格式:      0.070000
// 数字 0.01 的科学计数法格式: 7.000000e-02
// 数字 0.01 的十六进制浮点格式: 0x1.1eb851eb851ecp-4
// 数字 0.01 的默认格式:      0.07
// 将 0x1P-1022 解析为十六进制得到 2.22507e-308

```

默认是 `std::ios_base::fmtflags(0)`。

某些编译器上存在一个错误，导致

```

// Output:
// flt: -*****9.87
// hex: *****41
// $: $3.67*****
// usd: USD *****3.67
// usd: USD 3.67

std::cout << std::right << std::showbase << std::setfill('*')
    << "flt: " << std::setw(15) << -9.87 << "
    << "hex: " << std::setw(15) << 41 << "
    << " $: " << std::setw(15) << std::put_money(367, false) << "
    << "usd: " << std::setw(15) << std::put_money(367, true) << "
    << "usd: " << std::setw(15)

    << std::setfill(' ') << std::put_money(367, true) << ";

// Output:
// flt: *****-9.87
// hex: *****41
// $: *****$3.67
// usd: *****USD *3.67
// usd: USD 3.67

```

Default is `std::left`.

[std::fixed, std::scientific, std::hexfloat \[C++11\]](#) and [std::defaultfloat \[C++11\]](#) - change formatting for floating-point input/output.

`std::fixed` sets the `std::ios_base::floatfield` to `std::ios_base::fixed`,
`std::scientific` - to `std::ios_base::scientific`,
`std::hexfloat` - to `std::ios_base::fixed | std::ios_base::scientific` and
`std::defaultfloat` - to `std::ios_base::fmtflags(0)`.

fmtflags

```

#include <iostream>
...

std::cout << "
    << "The number 0.07 in fixed:      " << std::fixed << 0.01 << "
    << "The number 0.07 in scientific: " << std::scientific << 0.01 << "
    << "The number 0.07 in hexfloat:   " << std::hexfloat << 0.01 << "
    << "The number 0.07 in default:   " << std::defaultfloat << 0.01 << ";

double f;
std::istringstream is("0x1P-1022");
double f = std::strtod(is.str().c_str(), NULL);
std::cout << "Parsing 0x1P-1022 as hex gives " << f << "

// Output:
// The number 0.01 in fixed:      0.070000
// The number 0.01 in scientific: 7.000000e-02
// The number 0.01 in hexfloat:   0x1.1eb851eb851ecp-4
// The number 0.01 in default:   0.07
// Parsing 0x1P-1022 as hex gives 2.22507e-308

```

Default is `std::ios_base::fmtflags(0)`.

There is a **bug** on some compilers which causes

```

double f;
std::istringstream("0x1P-1022") >> std::hexfloat >> f;std::cout << "
将 0x1P-1022 解析为十六进制得到 " << f << ";// 输出: 将 0x1P-1022 解析
为十六进制得到 0

```

std::showpoint 和 std::noshowpoint - 控制浮点数表示中是否总是包含小数点。对输入流无影响。

```

std::cout << "显示小数点的 7.0: " << std::showpoint << 7.0 << "不显示小数点的
7.0: " << std::noshowpoint << 7.0 << ";// 输出: 显示小数点的 7.0: 7.0// 不显
示小数点的 7.0: 7

```

默认是 std::showpoint。

std::showpos 和 std::noshowpos - 控制非负数输出时是否显示 + 符号。对输入流无影响。

```

std::cout << "显示正号: " << std::showpos << 0 << ' ' <
< -2.718 << ' ' << 17 << "不显示正号: " << std:
:noshowpos << 0 << ' ' << -2.718 << ' ' << 17 <
< ";// 输出: 显示正号: +0 -2.718 +17// 不显示正号: 0
-2.718 17

```

默认是 std::noshowpos。

std::unitbuf, std::nounitbuf - 控制每次操作后是否刷新输出流。对输入流无影响。std::unitbuf 会导致刷新。

std::setbase(base) - 设置流的数字进制。

std::setbase(8) 等同于设置 std::ios_base::basefield 为 std::ios_base::oct,
std::setbase(16) - 设置为 std::ios_base::hex,
std::setbase(10) - 设置为 std::ios_base::dec。

如果 base 不是 8、10 或 16，则 std::ios_base::basefield 被设置为 std::ios_base::fmtflags(0)。这意味着十进制输出和前缀依赖的输入。

默认情况下 std::ios_base::basefield 是 std::ios_base::dec，因此默认 std::setbase(10)。

std::setprecision(n) - 改变浮点数的精度。

```

#include <cmath>
#include <limits>
...
typedef std::numeric_limits<long double> ld;

```

```

double f;
std::istringstream("0x1P-1022") >> std::hexfloat >> f;
std::cout << "Parsing 0x1P-1022 as hex gives " << f << '\n';
// Output: Parsing 0x1P-1022 as hex gives 0

```

std::showpoint 和 std::noshowpoint - control whether decimal point is always included in floating-point representation. Have no effect on input streams.

```

std::cout << "7.0 with showpoint: " << std::showpoint << 7.0 << '\n'
<< "7.0 with noshowpoint: " << std::noshowpoint << 7.0 << '\n';
// Output: 1.0 with showpoint: 7.00000
// 1.0 with noshowpoint: 7

```

Default is std::showpoint.

std::showpos 和 std::noshowpos - control displaying of the + sign in *non-negative* output. Have no effect on input streams.

```

std::cout << "With showpos: " << std::showpos
<< 0 << ' ' << -2.718 << ' ' << 17 << '\n'
<< "Without showpos: " << std::noshowpos
<< 0 << ' ' << -2.718 << ' ' << 17 << '\n';
// Output: With showpos: +0 -2.718 +17
// Without showpos: 0 -2.718 17

```

Default if std::noshowpos.

std::unitbuf, std::nounitbuf - control flushing output stream after every operation. Have no effect on input stream. std::unitbuf causes flushing.

std::setbase(base) - sets the numeric base of the stream.

std::setbase(8) equals to setting std::ios_base::basefield to std::ios_base::oct,
std::setbase(16) - to std::ios_base::hex,
std::setbase(10) - to std::ios_base::dec.

If base is other than 8, 10 or 16 then std::ios_base::basefield is setting to std::ios_base::fmtflags(0). It means decimal output and prefix-dependent input.

As default std::ios_base::basefield is std::ios_base::dec then by default std::setbase(10).

std::setprecision(n) - changes floating-point precision.

```

#include <cmath>
#include <limits>
...
typedef std::numeric_limits<long double> ld;

```

```

const long double pi = std::acos(-1.L);std::cout
<< "
<< "默认精度 (6) : pi: " << pi << "<<
    10pi: " << 10 * pi << "<< "std::setprecision(4): 10pi: "
<< std::setprecision(4) << 10 * pi << "<< "
    10000pi: " << 10000 * pi <<
<< "std::fixed:      10000pi: " << std::fixed << 10000 * pi << std::defaultfloat <<

<< "std::setprecision(10): pi: " << std::setprecision(10) << pi << "<< "max-1 radix preci
cion: pi: " << std::setprecision(1d::digits - 1) << pi << "<< "max+1 radix precision: pi: " << std::setp
recision(1d::digits + 1) << pi << "<< "significant digits prec: pi: " << std::setprecision(1d::digits10) << pi
<< ";// Output:

// 默认精度 (6) : pi: 3.14159
//                      10pi: 31.4159
// std::setprecision(4): 10pi: 31.42
//                      10000pi: 3.142e+04
// std::fixed:          10000pi: 31415.9265
// std::setprecision(10): pi: 3.141592654
// max-1 基数精度:   pi: 3.14159265358979323851280895940618620443274267017841339111328125
// max+1 基数精度:   pi: 3.14159265358979323851280895940618620443274267017841339111328125
// 有效数字精度:   pi: 3.14159265358979324

```

默认是 `std::setprecision(6)`。

`std::setiosflags(mask)` 和 `std::resetiosflags(mask)` - 设置和清除 `mask` 中指定的标志，
`mask` 类型为 `std::ios_base::fmtflags`。

```

#include <iostream>
...
std::istringstream in("10 010 10 010 10 010");
int num1, num2;

in >> std::oct >> num1 >> num2;
std::cout << "解析 \"10 010\" 使用 std::oct 得到: " << num1 << ' ' << num2 << ";// 输出: 解析 \"10 010"
" 使用 std::oct 得到: 8 8

in >> std::dec >> num1 >> num2;
std::cout << "解析 \"10 010\" 使用 std::dec 得到: " << num1 << ' ' << num2 << ";// 输出: 解析 \"10 010"
" 使用 std::oct 得到: 10 10

in >> std::resetiosflags(std::ios_base::basefield) >> num1 >> num2; std::cout <<
解析 \"10 010\" 使用自动检测得到: " << num1 << ' ' << num2 << ";// 解析 \"10 010\" 使用自动检测得到: 1
0 8

std::cout << std::setiosflags(std::ios_base::hex |
    std::ios_base::uppercase |
    std::ios_base::showbase) << 42 << ";// 输出: 0X2A

```

`std::skipws` 和 `std::noskipws` - 控制格式化输入函数是否跳过前导空白字符。对输出流无影响。

```
#include <iostream>
```

```

const long double pi = std::acos(-1.L);

std::cout << '\n'
<< "default precision (6): pi: " << pi << '\n'
<< "                                10pi: " << 10 * pi << '\n'
<< "std::setprecision(4): 10pi: " << std::setprecision(4) << 10 * pi << '\n'
<< "                                10000pi: " << 10000 * pi << '\n'
<< "std::fixed:      10000pi: " << std::fixed << 10000 * pi << std::defaultfloat <<
'\n'

<< "std::setprecision(10): pi: " << std::setprecision(10) << pi << '\n'
<< "max-1 radix precision: pi: " << std::setprecision(1d::digits - 1) << pi << '\n'
<< "max+1 radix precision: pi: " << std::setprecision(1d::digits + 1) << pi << '\n'
<< "significant digits prec: pi: " << std::setprecision(1d::digits10) << pi
<< ";// Output:

// default precision (6): pi: 3.14159
//                                10pi: 31.4159
// std::setprecision(4): 10pi: 31.42
//                                10000pi: 3.142e+04
// std::fixed:          10000pi: 31415.9265
// std::setprecision(10): pi: 3.141592654
// max-1 radix precision: pi: 3.14159265358979323851280895940618620443274267017841339111328125
// max+1 radix precision: pi: 3.14159265358979323851280895940618620443274267017841339111328125
// significant digits prec: pi: 3.14159265358979324

```

Default is `std::setprecision(6)`.

`std::setiosflags(mask)` 和 `std::resetiosflags(mask)` - set and clear flags specified in `mask` of
`std::ios_base::fmtflags` type.

```

#include <iostream>
...
std::istringstream in("10 010 10 010 10 010");
int num1, num2;

in >> std::oct >> num1 >> num2;
std::cout << "Parsing \"10 010\" with std::oct gives: " << num1 << ' ' << num2 << '\n';
// Output: Parsing "10 010" with std::oct gives: 8 8

in >> std::dec >> num1 >> num2;
std::cout << "Parsing \"10 010\" with std::dec gives: " << num1 << ' ' << num2 << '\n';
// Output: Parsing "10 010" with std::dec gives: 10 10

in >> std::resetiosflags(std::ios_base::basefield) >> num1 >> num2;
std::cout << "Parsing \"10 010\" with autodetect gives: " << num1 << ' ' << num2 << '\n';
// Parsing "10 010" with autodetect gives: 10 8

std::cout << std::setiosflags(std::ios_base::hex |
    std::ios_base::uppercase |
    std::ios_base::showbase) << 42 << '\n';
// Output: 0X2A

```

`std::skipws` 和 `std::noskipws` - control skipping of leading whitespace by the formatted input functions. Have no effect on output streams.

```
#include <iostream>
```

...

```
char c1, c2, c3;
std::istringstream("a b c") >> c1 >> c2 >> c3;
std::cout << "默认行为: c1 = " << c1 << " c2 = " << c2 << " c3 = " << c3 << ";std::istringstream("a b c") >> s
std::noskipws >> c1 >> c2 >> c3;
std::cout << "noskipws 行为: c1 = " << c1 << " c2 = " << c2 << " c3 = " << c3 << "// 输出: 默认行为: c1 =
a c2 = b c3 = c
// noskipws 行为: c1 = a c2 = c3 = b
```

默认是 `std::ios_base::skipws`。

`std::quoted(s[, delim[, escape]])` [C++14] - 插入或提取带有嵌入空格的带引号字符串。

s - 要插入或提取的字符串。

delim - 用作分隔符的字符，默认是 "。

escape - 用作转义字符的字符，默认是 \。

```
#include <sstream>
...
std::stringstream ss;
std::string in = "带空格的字符串, 以及嵌入的 \"引号\" 也有";
std::string out;

ss << std::quoted(in);
std::cout << "读取 [" << in << "]<< "存储为 ["
    << ss.str() << "];ss >> std::quoted(out);

std::cout << "写出 [" << out << "]";// 输出:

// 读取 [带空格的字符串, 以及嵌入的 "引号" 也有]
// 存储为 [带空格的字符串, 以及嵌入的 \"引号\" 也有"]
// 写出 [带空格的字符串, 以及嵌入的 "引号" 也有]
```

更多信息请参见上方链接。

第14.2节：输出流操纵器

`std::ends` - 向输出流插入一个空字符 '\0'。更正式地说，该操纵器的声明如下

```
template <class charT, class traits>
std::basic_ostream<charT, traits>& ends(std::basic_ostream<charT, traits>& os);
```

当在表达式中使用时，该操作器通过调用 `os.put(charT())` 来放置字符
`os << std::ends;`

`std::endl` 和 `std::flush` 都通过调用 `out.flush()` 来刷新输出流 `out`。它会立即产生输出。但 `std::endl` 在刷新之前插入换行符 '\n' 符号。

```
std::cout << "第一行." << std::endl << "第二行. " << std::flush
    << "仍然是第二行。";
```

...

```
char c1, c2, c3;
std::istringstream("a b c") >> c1 >> c2 >> c3;
std::cout << "Default behavior: c1 = " << c1 << " c2 = " << c2 << " c3 = " << c3 << '\n';
std::istringstream("a b c") >> std::noskipws >> c1 >> c2 >> c3;
std::cout << "noskipws behavior: c1 = " << c1 << " c2 = " << c2 << " c3 = " << c3 << '\n';
// Output: Default behavior: c1 = a c2 = b c3 = c
// noskipws behavior: c1 = a c2 = c3 = b
```

Default is `std::ios_base::skipws`.

`std::quoted(s[, delim[, escape]])` [C++14] - inserts or extracts quoted strings with embedded spaces.

s - the string to insert or extract.

delim - the character to use as the delimiter, " by default.

escape - the character to use as the escape character, \ by default.

```
#include <sstream>
...
std::stringstream ss;
std::string in = "String with spaces, and embedded \"quotes\" too";
std::string out;

ss << std::quoted(in);
std::cout << "read in      [ " << in << " ]\n"
    << "stored as   [ " << ss.str() << " ]\n";

ss >> std::quoted(out);
std::cout << "written out [ " << out << " ]\n";
// Output:
// read in      [String with spaces, and embedded "quotes" too]
// stored as   [String with spaces, and embedded \"quotes\" too]
// written out [String with spaces, and embedded "quotes" too]
```

For more information see the link above.

Section 14.2: Output stream manipulators

`std::ends` - inserts a null character '\0' to output stream. More formally this manipulator's declaration looks like

```
template <class charT, class traits>
std::basic_ostream<charT, traits>& ends(std::basic_ostream<charT, traits>& os);
```

and this manipulator places character by calling `os.put(charT())` when used in an expression
`os << std::ends;`

`std::endl` and `std::flush` both flush output stream `out` by calling `out.flush()`. It causes immediately producing output. But `std::endl` inserts end of line '\n' symbol before flushing.

```
std::cout << "First line." << std::endl << "Second line. " << std::flush
    << "Still second line.";
```

```
// 输出：第一行。  
// 第二行。 仍然是第二行。
```

std::setfill(c) - 将填充字符更改为 c。通常与 std::setw 一起使用。

```
std::cout << "默认填充：" << std::setw(10) << 79 << "setfill('#')：" << std::setfill('#')  
<< std::setw(10) << 42 << ";// 输出：
```

```
// 默认填充： 79  
// setfill('#')： #####79
```

std::put_money(mon[, intl]) [C++11]。在表达式 out << std::put_money(mon, intl) 中，将货币值 mon (类型为 long double 或 std::basic_string) 转换为其字符表示，格式由当前赋予 out 的区域设置中的 std::money_put facet 指定。如果 intl 为 true，则使用国际货币字符串，否则使用货币符号。

```
long double money = 123.45;  
// 或者 std::string money = "123.45";std::c  
  
out.imbue(std::locale("en_US.utf8"));std::cout << std::s  
howbase << "en_US：" << std::put_money(money)<< " or " << std::put_money  
(money, true) << ";// 输出: en_US: $1.23 or USD 1.23std::c  
out.imbue(std::locale("ru_RU.utf8"));std::cout  
  
<< "ru_RU：" << std::put_money(money)<< " or " <<  
std::put_money(money, true) << ";// 输出: ru_RU: 1.23 p  
y6 or 1.23 RUBstd::cout.imbue(std::locale("ja_JP.utf8"));std::c  
out << "ja_JP：" << std::put_money(money)<<  
" or " << std::put_money(money, true) << ";// 输出:  
ja_JP: ¥123 or JPY 123
```

std::put_time(tmb, fmt) [C++11] - 根据指定的格式 fmt 格式化并输出日期/时间值到 std::tm。

tmb - 指向日历时间结构体的指针 const std::tm*，通常由 localtime() 或 gmtime() 获得。

fmt - 指向以空字符结尾的字符串的指针 const CharT*，指定转换格式。

```
#include <ctime>  
...  
  
std::time_t t = std::time(nullptr);std::tm tm  
= *std::localtime(&t);std::cout.imbue(std  
::locale("ru_RU.utf8"));std::cout << "ru_RU：" << std::put_time(&tm, "%c %Z") << ";// 可能的输出:  
  
// ru_RU: Вт 04 июл 2017 15:08:35 UTC
```

更多信息请参见上方链接。

```
// Output: First line.  
// Second line. Still second line.
```

std::setfill(c) - changes the fill character to c. Often used with std::setw.

```
std::cout << "\nDefault fill：" << std::setw(10) << 79 << '\n'  
<< "setfill('#')：" << std::setfill('#')  
<< std::setw(10) << 42 << '\n';  
// Output:  
// Default fill: 79  
// setfill('#'): #####79
```

std::put_money(mon[, intl]) [C++11]。In an expression out << std::put_money(mon, intl), converts the monetary value mon (of long double or std::basic_string type) to its character representation as specified by the std::money_put facet of the locale currently imbued in out. Use international currency strings if intl is true, use currency symbols otherwise.

```
long double money = 123.45;  
// or std::string money = "123.45";  
  
std::cout.imbue(std::locale("en_US.utf8"));  
std::cout << std::showbase << "en_US：" << std::put_money(money)  
<< " or " << std::put_money(money, true) << '\n';  
// Output: en_US: $1.23 or USD 1.23  
  
std::cout.imbue(std::locale("ru_RU.utf8"));  
std::cout << "ru_RU：" << std::put_money(money)  
<< " or " << std::put_money(money, true) << '\n';  
// Output: ru_RU: 1.23 py6 or 1.23 RUB  
  
std::cout.imbue(std::locale("ja_JP.utf8"));  
std::cout << "ja_JP：" << std::put_money(money)  
<< " or " << std::put_money(money, true) << '\n';  
// Output: ja_JP: ¥123 or JPY 123
```

std::put_time(tmb, fmt) [C++11] - formats and outputs a date/time value to std::tm according to the specified format fmt.

tmb - pointer to the calendar time structure const std::tm* as obtained from localtime() or gmtime().
fmt - pointer to a null-terminated string const CharT* specifying the format of conversion.

```
#include <ctime>  
...  
  
std::time_t t = std::time(nullptr);  
std::tm tm = *std::localtime(&t);  
  
std::cout.imbue(std::locale("ru_RU.utf8"));  
std::cout << "\nru_RU：" << std::put_time(&tm, "%c %Z") << '\n';  
// Possible output:  
// ru_RU: Вт 04 июл 2017 15:08:35 UTC
```

For more information see the link above.

第14.3节：输入流操纵器

std::ws - 消耗输入流中的前导空白字符。它不同于 std::skipws。

```
#include <sstream>
...
std::string str;
std::istringstream(" \v\r 哇！这里没有空白字符！") >> std::ws >> str;std::cout << str;
// 输出：哇！这里没有空白字符！
```

std::get_money(mon[, intl]) [C++11]。表达式 in >> std::get_money(mon, intl) 将字符输入解析为货币值，按照当前 in 所使用的区域设置中的 std::money_get facet 规范，并将值存储在 mon (类型为 long double 或 std::basic_string)。如果 intl 为 true，操纵器期望必须的国际货币字符串，否则期望可选的货币符号。

```
#include <sstream>
#include <locale>
...
std::istringstream in("$1,234.56 2.22 USD 3.33");
long double v1, v2;
std::string v3;

in.imbue(std::locale("en_US.UTF-8"));
in >> std::get_money(v1) >> std::get_money(v2) >> std::get_money(v3, true);
if (in) {
    std::cout << std::quoted(in.str()) << " 解析为: " << v1 << ", " << v2 <
        <, " << v3 << "}";
}

// 输出：
// "$1,234.56 2.22 USD 3.33" 解析为: 123456, 222, 333
```

std::get_time(tmb, fmt) [C++11] - 解析存储在 tmb 中的指定格式 fmt 的日期/时间值。

tmb - 指向 const std::tm* 对象的有效指针，结果将存储在该对象中。

fmt - 指向以空字符结尾的字符串 const CharT*，指定转换格式。

```
#include <sstream>
#include <locale>
...
std::tm t = {};
std::istringstream ss("2011-Februar-18 23:12:34");

ss.imbue(std::locale("de_DE.utf-8"));
ss >> std::get_time(&t, "%Y-%b-%d %H:%M:%S");
if (ss.fail()) {
    std::cout << "解析失败";
}

else {
    std::cout << std::put_time(&t, "%c") << "}";
}

// 可能的输出：
// 2011年2月18日 星期五 23:12:34
```

Section 14.3: Input stream manipulators

std::ws - consumes leading whitespaces in input stream. It different from std::skipws.

```
#include <sstream>
...
std::string str;
std::istringstream(" \v\n\r\t Wow!There is no whitespaces!") >> std::ws >> str;
std::cout << str;
// Output: Wow!There is no whitespaces!
```

std::get_money(mon[, intl]) [C++11]。In an expression in >> std::get_money(mon, intl) parses the character input as a monetary value, as specified by the std::money_get facet of the locale currently imbued in in, and stores the value in mon (of long double or std::basic_string type). Manipulator expects required international currency strings if intl is true, expects optional currency symbols otherwise.

```
#include <sstream>
#include <locale>
...
std::istringstream in("$1,234.56 2.22 USD 3.33");
long double v1, v2;
std::string v3;

in.imbue(std::locale("en_US.UTF-8"));
in >> std::get_money(v1) >> std::get_money(v2) >> std::get_money(v3, true);
if (in) {
    std::cout << std::quoted(in.str()) << " parsed as: "
        << v1 << ", " << v2 << ", " << v3 << '\n';
}
// Output:
// "$1,234.56 2.22 USD 3.33" parsed as: 123456, 222, 333
```

std::get_time(tmb, fmt) [C++11] - parses a date/time value stored in tmb of specified format fmt.

tmb - valid pointer to the const std::tm* object where the result will be stored.

fmt - pointer to a null-terminated string const CharT* specifying the conversion format.

```
#include <sstream>
#include <locale>
...
std::tm t = {};
std::istringstream ss("2011-Februar-18 23:12:34");

ss.imbue(std::locale("de_DE.utf-8"));
ss >> std::get_time(&t, "%Y-%b-%d %H:%M:%S");
if (ss.fail()) {
    std::cout << "Parse failed\n";
}
else {
    std::cout << std::put_time(&t, "%c") << '\n';
}
// Possible output:
// Sun Feb 18 23:12:34 2011
```

更多信息请参见上方链接。

For more information see the link above.

第15章：流程控制

第15.1节：case

介绍switch语句的case标签。操作数必须是常量表达式，且类型与switch条件匹配。当执行switch语句时，如果有操作数等于条件的case标签，将跳转到该标签。

```
char c = getchar();
bool confirmed;
switch (c) {
    case 'y':
        confirmed = true;
        break;
    case 'n':
        confirmed = false;
        break;
    default:
        std::cout << "invalid response!" ;abort();
}
```

第15.2节：switch语句

根据C++标准，

switch语句根据条件的值将控制转移到若干语句中的一个。

关键字switch后跟一个带括号的条件和一个代码块，该代码块可能包含case标签和一个可选的default标签。当执行switch语句时，控制将转移到与条件值匹配的case标签（如果有），或者转移到default标签（如果有）。

条件必须是一个表达式或声明，其类型为整数类型或枚举类型，或者是具有转换为整数或枚举类型的转换函数的类类型。

```
char c = getchar();
bool confirmed;
switch (c) {
    case 'y':
        confirmed = true;
        break;
    case 'n':
        confirmed = false;
        break;
    default:
        std::cout << "invalid response!" ;abort();
}
```

第15.3节：catch语句

catch关键字引入一个异常处理器，即当抛出兼容类型的异常时，控制将转移到的代码块。catch关键字后跟一个带括号的异常声明，

Chapter 15: Flow Control

Section 15.1: case

Introduces a case label of a switch statement. The operand must be a constant expression and match the switch condition in type. When the switch statement is executed, it will jump to the case label with operand equal to the condition, if any.

```
char c = getchar();
bool confirmed;
switch (c) {
    case 'y':
        confirmed = true;
        break;
    case 'n':
        confirmed = false;
        break;
    default:
        std::cout << "invalid response!\n";
        abort();
}
```

Section 15.2: switch

According to the C++ standard,

The `switch` statement causes control to be transferred to one of several statements depending on the value of a condition.

The keyword `switch` is followed by a parenthesized condition and a block, which may contain `case` labels and an optional `default` label. When the switch statement is executed, control will be transferred either to a `case` label with a value matching that of the condition, if any, or to the `default` label, if any.

The condition must be an expression or a declaration, which has either integer or enumeration type, or a class type with a conversion function to integer or enumeration type.

```
char c = getchar();
bool confirmed;
switch (c) {
    case 'y':
        confirmed = true;
        break;
    case 'n':
        confirmed = false;
        break;
    default:
        std::cout << "invalid response!\n";
        abort();
}
```

Section 15.3: catch

The `catch` keyword introduces an exception handler, that is, a block into which control will be transferred when an exception of compatible type is thrown. The `catch` keyword is followed by a parenthesized `exception declaration`,

其形式类似于函数参数声明：参数名可以省略，并且允许使用省略号...，表示匹配任何类型。异常处理器仅在其声明与异常类型兼容时处理该异常。更多细节请参见异常捕获。

```
try {
std::vector<int> v(N);
    // 做一些操作
} 捕获 (const std::bad_alloc&) {
std::cout << "为向量分配内存失败！" << std::endl;
} 捕获 (const std::runtime_error& e) {
std::cout << "运行时错误: " << e.what() << std::endl;
} 捕获 (...) {
std::cout << "意外异常！" << std::endl;
    抛出;
}
```

第15.4节：throw

- 当 `throw` 出现在带有操作数的表达式中时，其效果是抛出一个异常，该异常是操作数的副本。操作数。

```
void print_asterisks(int count) {
    如果 (count < 0) {
        抛出 std::invalid_argument("count 不能为负数！");
    }
    当 (count--) { putchar('*'); }
}
```

- 当 `throw` 在没有操作数的表达式中出现时，其作用是重新抛出当前异常。如果没有当前异常，则调用 `std::terminate`。

```
try {
    // 有风险的操作
} catch (const std::bad_alloc&) {
    std::cerr << "内存不足" << std::endl;
} catch (...) {
    std::cerr << "意外异常" << std::endl;
    // 希望调用者知道如何处理此异常
    throw;
}
```

- 当 `throw` 出现在函数声明符中时，它引入了动态异常说明，列出了函数允许传播的异常类型。

```
// 该函数可能传播 std::runtime_error,
// 但不会传播例如 std::logic_error
void risky() throw(std::runtime_error);
// 该函数不能传播任何异常
void safe() throw();
```

动态异常规范自C++11起已被弃用。

请注意，上述前两个使用 `throw` 的情况构成的是表达式而非语句。（`throw` 表达式的类型是 `void`。）这使得它们可以嵌套在表达式中，如下所示：

which is similar in form to a function parameter declaration: the parameter name may be omitted, and the ellipsis ... is allowed, which matches any type. The exception handler will only handle the exception if its declaration is compatible with the type of the exception. For more details, see catching exceptions.

```
try {
    std::vector<int> v(N);
    // do something
} catch (const std::bad_alloc&) {
    std::cout << "failed to allocate memory for vector!" << std::endl;
} catch (const std::runtime_error& e) {
    std::cout << "runtime error: " << e.what() << std::endl;
} catch (...) {
    std::cout << "unexpected exception!" << std::endl;
    throw;
}
```

Section 15.4: throw

- When `throw` occurs in an expression with an operand, its effect is to throw an exception, which is a copy of the operand.

```
void print_asterisks(int count) {
    if (count < 0) {
        throw std::invalid_argument("count cannot be negative!");
    }
    while (count--) { putchar('*'); }
}
```

- When `throw` occurs in an expression without an operand, its effect is to rethrow the current exception. If there is no current exception, `std::terminate` is called.

```
try {
    // something risky
} catch (const std::bad_alloc&) {
    std::cerr << "out of memory" << std::endl;
} catch (...) {
    std::cerr << "unexpected exception" << std::endl;
    // hope the caller knows how to handle this exception
    throw;
}
```

- When `throw` occurs in a function declarator, it introduces a dynamic exception specification, which lists the types of exceptions that the function is allowed to propagate.

```
// this function might propagate a std::runtime_error,
// but not, say, a std::logic_error
void risky() throw(std::runtime_error);
// this function can't propagate any exceptions
void safe() throw();
```

Dynamic exception specifications are deprecated as of C++11.

Note that the first two uses of `throw` listed above constitute expressions rather than statements. (The type of a `throw` expression is `void`.) This makes it possible to nest them within expressions, like so:

```
无符号整数 predecessor(无符号整数 x) {
    返回 (x > 0) ? (x - 1) : (throw std::invalid_argument("0 没有前驱"));
}
```

第15.5节：default

在switch语句中，引入一个标签，如果条件的值不等于任何case标签的值，则跳转到该标签。

```
char c = getchar();
bool confirmed;
switch (c) {
    case 'y':
        confirmed = true;
        break;
    case 'n':
        confirmed = false;
        break;
    default:
        std::cout << "invalid response!"; abort();
}
```

版本 ≥ C++11

定义默认构造函数、拷贝构造函数、移动构造函数、析构函数、拷贝赋值运算符或移动赋值运算符，使其具有默认行为。

```
类 Base {
    // ...
    // 我们希望能够通过Base*删除派生类,
    // 但保持Base析构函数的常规行为。
    虚函数 ~Base() = default;
};
```

第15.6节：try

关键字try后面跟着一个代码块，或者先跟一个构造函数初始化列表然后是一个代码块（见此处）。try代码块后面跟着一个或多个catch代码块。如果异常从try代码块中传播出来，try代码块后面的每个对应的catch代码块都有机会处理该异常，前提是类型匹配。

```
std::vector<int> v(N);      // 如果这里抛出异常,
                            // 后面的catch代码块将无法捕获该异常
try {
    std::vector<int> v(N); // 如果这里抛出异常,
                            // 后面的catch代码块将会捕获该异常
    // 对v进行一些操作
} 捕获 (const std::bad_alloc&) {
    // 处理来自try代码块的bad_alloc异常
}
```

第15.7节：if

介绍if语句。关键字if后面必须跟一个带括号的条件，该条件可以是表达式也可以是声明。如果条件为真，则执行条件后面的子语句。

```
int x;
```

```
unsigned int predecessor(unsigned int x) {
    return (x > 0) ? (x - 1) : (throw std::invalid_argument("0 has no predecessor"));
}
```

Section 15.5: default

In a switch statement, introduces a label that will be jumped to if the condition's value is not equal to any of the case labels' values.

```
char c = getchar();
bool confirmed;
switch (c) {
    case 'y':
        confirmed = true;
        break;
    case 'n':
        confirmed = false;
        break;
    default:
        std::cout << "invalid response!\n";
        abort();
}
```

Version ≥ C++11

Defines a default constructor, copy constructor, move constructor, destructor, copy assignment operator, or move assignment operator to have its default behaviour.

```
class Base {
    // ...
    // we want to be able to delete derived classes through Base*,
    // but have the usual behaviour for Base's destructor.
    virtual ~Base() = default;
};
```

Section 15.6: try

The keyword `try` is followed by a block, or by a constructor initializer list and then a block (see here). The try block is followed by one or more catch blocks. If an exception propagates out of the try block, each of the corresponding catch blocks after the try block has the opportunity to handle the exception, if the types match.

```
std::vector<int> v(N);      // if an exception is thrown here,
                            // it will not be caught by the following catch block
try {
    std::vector<int> v(N); // if an exception is thrown here,
                            // it will be caught by the following catch block
    // do something with v
} catch (const std::bad_alloc&) {
    // handle bad_alloc exceptions from the try block
}
```

Section 15.7: if

Introduces an if statement. The keyword `if` must be followed by a parenthesized condition, which can be either an expression or a declaration. If the condition is truthy, the substatement after the condition will be executed.

```
int x;
```

```

std::cout << "请输入一个正数。" << std::endl;
std::cin >> x;
if (x <= 0) {
    std::cout << "你没有输入一个正数！" << std::endl;
    abort();
}

```

第15.8节：else

if语句的第一个子语句后面可以跟随关键字else。当条件为假（即第一个子语句未执行）时，将执行else关键字后的子语句。

```

int x;
std::cin >> x;
if (x%2 == 0) {
    std::cout << "该数字是偶数";} else {

    std::cout << "该数字是奇数";}

```

第15.9节：条件结构：if, if..else

if 和 else :

用于检查给定表达式是否返回真或假，并据此执行操作：

```
if (condition) statement
```

条件可以是任何有效的C++表达式，其返回值可用于判断真/假，例如：

```

if (true) { /* 这里的代码 */ } // 判断true是否为真，若是则执行括号内代码
if (false) { /* 这里的代码 */ } // 始终跳过代码，因为false永远为假

```

条件可以是任何东西，例如函数、变量或比较

```

if(istruer()) { } // 计算函数，如果返回true，则if会执行代码if(isTrue(var)) { } // 计算传入参数var后函数的返回值if(a ==
b) { } // 这将计算表达式(a==b)的返回值，如果相等则为true，不等则为false

```

```
if(a) { } // 如果a是布尔类型，则判断其值；如果是整数，任何非零值都为true，
```

如果我们想检查多个表达式，可以用两种方式：

使用二元运算符：

```

if (a && b) { } // 只有当a和b都为true时才为true (二元运算符在这里不在范围内)
if (a || b) { } // 如果a或b为true则为true

```

使用if/ifelse/else：

对于简单的切换，使用if或else

```
if (a== "test") {
```

```

std::cout << "Please enter a positive number." << std::endl;
std::cin >> x;
if (x <= 0) {
    std::cout << "You didn't enter a positive number!" << std::endl;
    abort();
}

```

Section 15.8: else

The first substatement of an if statement may be followed by the keyword else. The substatement after the else keyword will be executed when the condition is falsey (that is, when the first substatement is not executed).

```

int x;
std::cin >> x;
if (x%2 == 0) {
    std::cout << "The number is even\n";
} else {
    std::cout << "The number is odd\n";
}

```

Section 15.9: Conditional Structures: if, if..else

if and else:

it used to check whether the given expression returns true or false and acts as such:

```
if (condition) statement
```

the condition can be any valid C++ expression that returns something that be checked against truth/falsehood for example:

```

if (true) { /* code here */ } // evaluate that true is true and execute the code in the brackets
if (false) { /* code here */ } // always skip the code since false is always false

```

the condition can be anything, a function, a variable, or a comparison for example

```

if(istruer()) { } // evaluate the function, if it returns true, the if will execute the code
if(isTrue(var)) { } //evaluate the return of the function after passing the argument var
if(a == b) { } // this will evaluate the return of the expression (a==b) which will be true if equal and false if unequal
if(a) { } //if a is a boolean type, it will evaluate for its value, if it's an integer, any non zero value will be true,

```

if we want to check for a multiple expressions we can do it in two ways :

using binary operators :

```

if (a && b) { } // will be true only if both a and b are true (binary operators are outside the scope here)
if (a || b) { } //true if a or b is true

```

using if/ifelse/else:

for a simple switch either if or else

```
if (a== "test") {
```

```
//如果a是字符串"test"则执行
} 否则 {
    // 只有当第一个失败时，才会执行
}
```

多重选择：

```
if (a=='a') {
// 如果 a 是字符 'a' 的值
} else if (a=='b') {
// 如果 a 是字符 'b' 的值
} else if (a=='c') {
// 如果 a 是字符 'c' 的值
} else {
// 如果 a 不是上述任何一个
}
```

但必须注意，如果代码检查的是同一变量的值，应使用 'switch'

第15.10节：goto

跳转到带标签的语句，该语句必须位于当前函数中。

```
bool f(int arg) {
    bool result = false;
    hWidget widget = get_widget(arg);
    if (!g()) {
        // 我们无法继续，但仍必须进行清理
        goto end;
    }
    // ...
result = true;
end:
release_widget(widget);
    return result;
}
```

第15.11节：跳转语句：break、continue、goto、exit

break语句：

使用break我们可以离开循环，即使循环结束的条件未满足。它可以用来结束无限循环，或者强制其在自然结束之前终止

语法是

```
break;
```

示例：我们经常在 `switch` 语句中使用 `break`，即一旦满足某个 `switch` 的 `case` 条件，该条件对应的代码块就会被执行。

```
switch(condition){
case 1: block1;
case 2: block2;
case 3: block3;
default: blockdefault;
}
```

```
//will execute if a is a string "test"
} else {
    // only if the first failed, will execute
}
```

for multiple choices :

```
if (a=='a') {
// if a is a char valued 'a'
} else if (a=='b') {
// if a is a char valued 'b'
} else if (a=='c') {
// if a is a char valued 'c'
} else {
//if a is none of the above
}
```

however it must be noted that you should use '`switch`' instead if your code checks for the same variable's value

Section 15.10: goto

Jumps to a labelled statement, which must be located in the current function.

```
bool f(int arg) {
    bool result = false;
    hWidget widget = get_widget(arg);
    if (!g()) {
        // we can't continue, but must do cleanup still
        goto end;
    }
    // ...
result = true;
end:
release_widget(widget);
    return result;
}
```

Section 15.11: Jump statements : break, continue, goto, exit

The break instruction:

Using `break` we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end

The syntax is

```
break;
```

Example: we often use `break` in `switch` cases, ie once a case in `switch` is satisfied then the code block of that condition is executed .

```
switch(conditon){
case 1: block1;
case 2: block2;
case 3: block3;
default: blockdefault;
}
```

在这种情况下，如果满足 case 1，则执行 block1，实际上我们只希望处理 block1，但执行完 block1 后，剩余的 block2、block3 和 blockdefault 也会被执行，尽管只有 case 1 被满足。为避免这种情况，我们在每个代码块末尾使用 break，如下所示：

```
switch(condition){  
case 1: block1;  
    break;  
case 2: block2;  
    break;  
case 3: block3;  
    break;  
default: blockdefault;  
    break;  
}
```

这样就只会执行一个代码块，控制权会跳出 switch 语句。

break 也可以用于其他条件和非条件循环，如 if、while、for 等；

示例：

```
if(condition1){  
    ...  
    if(condition2){  
        .....  
        break;  
    }  
    ...  
}
```

continue 指令：

continue 指令使程序跳过当前迭代中循环的其余部分，就好像已经到达语句块的末尾一样，从而跳转到下一次迭代。

语法是

```
continue;
```

示例考虑以下内容：

```
for(int i=0;i<10;i++){  
    if(i%2==0)  
        continue;  
    cout<<" @<<i;  
}
```

其输出结果为：

```
@1  
@3  
@5  
@7  
@9
```

当条件 `i%2==0` 满足时，代码中的 `continue` 会被执行，这会导致编译器跳过所有剩余的代码（打印 @ 和 i），并执行循环的增减语句。

in this case if case 1 is satisfied then block 1 is executed , what we really want is only the block1 to be processed but instead once the block1 is processed remaining blocks,block2,block3 and blockdefault are also processed even though only case 1 was satisfied.To avoid this we use break at the end of each block like :

```
switch(condition){  
case 1: block1;  
    break;  
case 2: block2;  
    break;  
case 3: block3;  
    break;  
default: blockdefault;  
    break;  
}
```

so only one block is processed and the control moves out of the switch loop.

break can also be used in other conditional and non conditional loops like if,while,for etc;

example:

```
if(condition1){  
    ...  
    if(condition2){  
        .....  
        break;  
    }  
    ...  
}
```

The continue instruction:

The continue instruction causes the program to skip the rest of the loop in the present iteration as if the end of the statement block would have been reached, causing it to jump to the following iteration.

The syntax is

```
continue;
```

Example consider the following :

```
for(int i=0;i<10;i++){  
    if(i%2==0)  
        continue;  
    cout<<"\n @"<<i;  
}
```

which produces the output:

```
@1  
@3  
@5  
@7  
@9
```

in this code whenever the condition `i%2==0` is satisfied `continue` is processed, this causes the compiler to skip all the remaining code(printing @ and i) and increment/decrement statement of the loop gets executed.

```

for (initialisation; condition; increment/decrement)
{
    ...
    if (True Condition)
        continue;           Continues Loop with
                            the Next Value
    ...
}

```

goto 指令：

它允许程序跳转到程序中的另一个绝对位置。你应谨慎使用此功能，因为其执行会忽略任何类型的嵌套限制。目标点由标签标识，标签随后作为 goto 指令的参数使用。标签由一个有效标识符后跟冒号 (:) 组成。

语法是

```

goto label;
...
label: statement;

```

注意：强烈不建议使用 goto 语句，因为它使程序的控制流程难以追踪，导致程序难以理解和修改。

goto label;

```

label :
statement 1;
statement 2;
statement 3;

```

Forward Reference

label :

```

statement 1;
statement 2;
statement 3;

```

goto label;

Backward Reference

```

for (initialisation; condition; increment/decrement)
{
    ...
    if (True Condition)
        continue;           Continues Loop with
                            the Next Value
    ...
}

```

The goto instruction:

It allows making an absolute jump to another point in the program. You should use this feature carefully since its execution ignores any type of nesting limitation. The destination point is identified by a label, which is then used as an argument for the goto instruction. A label is made of a valid identifier followed by a colon (:)

The syntax is

```

goto label;
...
label: statement;

```

Note: Use of goto statement is highly discouraged because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify.

goto label;

```

label :
statement 1;
statement 2;
statement 3;

```

Forward Reference

label :

```

statement 1;
statement 2;
statement 3;

```

goto label;

Backward Reference

示例：

```

int num = 1;
STEP:
do{

    if( num%2==0 )
    {
num = num + 1;

```

Example :

```

int num = 1;
STEP:
do{

    if( num%2==0 )
    {
        num = num + 1;

```

```

    goto STEP;
}

cout << "num 的值 : " << num << endl;
num = num + 1;
}while( num < 10 );

```

输出：

```

num 的值 : 1
num 的值 : 3
num 的值 : 5
num 的值 : 7
num 的值 : 9

```

每当条件 `num%2==0` 满足时，`goto` 会将执行控制转移到 `do-while`

循环的开始处。

退出函数：

`exit` 是在 `cstdlib` 中定义的函数。`exit` 的目的是以特定的退出

代码终止正在运行的程序。其原型为：

```
void exit (int exit code);
```

`cstdlib` 定义了标准退出代码 `EXIT_SUCCESS` 和 `EXIT_FAILURE`。

第15.12节：return

将控制权从函数返回给调用者。

如果 `return` 有操作数，该操作数将被转换为函数的返回类型，转换后的值将返回给调用者。

```

int f() {
    return 42;
}
int x = f(); // x 是 42
int g() {
    return 3.14;
}
int y = g(); // y 是 3

```

如果 `return` 没有操作数，函数必须具有 `void` 返回类型。作为特殊情况，`void` 返回的函数如果表达式类型为 `void`，也可以返回该表达式。

```

void f(int x) {
    if (x < 0) return;
    std::cout << sqrt(x);
}
int g() { return 42; }
void h() {
    return f(); // 调用 f, 然后返回
    return g(); // 格式错误
}

```

当 `main` 返回时，`std::exit` 会隐式调用并带有返回值，该值因此返回给

```

    goto STEP;
}

cout << "value of num : " << num << endl;
num = num + 1;
}while( num < 10 );

```

output：

```

value of num : 1
value of num : 3
value of num : 5
value of num : 7
value of num : 9

```

whenever the condition `num%2==0` is satisfied the `goto` sends the execution control to the beginning of the `do-while` loop.

The exit function:

`exit` is a function defined in `cstdlib`. The purpose of `exit` is to terminate the running program with a specific exit code. Its prototype is:

```
void exit (int exit code);
```

`cstdlib` defines the standard exit codes `EXIT_SUCCESS` and `EXIT_FAILURE`.

Section 15.12: return

Returns control from a function to its caller.

If `return` has an operand, the operand is converted to the function's return type, and the converted value is returned to the caller.

```

int f() {
    return 42;
}
int x = f(); // x is 42
int g() {
    return 3.14;
}
int y = g(); // y is 3

```

If `return` does not have an operand, the function must have `void` return type. As a special case, a `void`-returning function can also return an expression if the expression has type `void`.

```

void f(int x) {
    if (x < 0) return;
    std::cout << sqrt(x);
}
int g() { return 42; }
void h() {
    return f(); // calls f, then returns
    return g(); // ill-formed
}

```

When `main` returns, `std::exit` is implicitly called with the return value, and the value is thus returned to the

执行环境。(但是，从 main 返回会销毁自动局部变量，而直接调用 std::exit 则不会。)

```
int main(int argc, char** argv) {
    if (argc < 2) {
        std::cout << "缺少参数"; return EXIT_FAILURE; // 等
        同于：exit(EXIT_FAILURE);
    }
}
```

execution environment. (However, returning from `main` destroys automatic local variables, while calling `std::exit` directly does not.)

```
int main(int argc, char** argv) {
    if (argc < 2) {
        std::cout << "Missing argument\n";
        return EXIT_FAILURE; // equivalent to: exit(EXIT_FAILURE);
    }
}
```

第16章：元编程

在C++中，元编程指的是使用宏或模板在编译时生成代码。

一般来说，在这个角色中宏是不被看好的，更倾向于使用模板，尽管模板没有那么通用。

模板元编程通常利用编译时计算，无论是通过模板还是`constexpr`函数，来实现生成代码的目标，然而编译时计算本身并不是元编程。

第16.1节：计算阶乘

阶乘可以使用模板元编程技术在编译时计算。

```
#include <iostream>

template<unsigned int n>
struct factorial
{
    enum
    {
        value = n * factorial<n - 1>::value
    };
};

template<>
struct factorial<0>
{
    enum { value = 1 };
};

int main()
{
    std::cout << factorial<7>::value << std::endl; // 输出 "5040"
}
```

`factorial` 是一个结构体，但在模板元编程中它被视为一个模板元函数。按照惯例，模板元函数通过检查特定成员来求值，生成类型的元函数检查`::type`，生成值的元函数检查`::value`。

在上述代码中，我们通过用想要传递的参数实例化模板来计算`factorial`元函数，并使用`::value`来获取计算结果。

元函数本身依赖于递归地用更小的值实例化相同的元函数。

`factorial<0>`特化表示终止条件。模板元编程具有函数式编程语言的大部分限制，因此递归是主要的“循环”结构。

由于模板元函数在编译时执行，其结果可以用于需要编译时值的上下文中。例如：

```
int my_array[factorial<5>::value];
```

自动数组必须具有编译时定义的大小。元函数的结果是编译时常量，因此可以在这里使用。

限制：大多数编译器不允许递归深度超过一定限制。例如，g++编译器默认

Chapter 16: Metaprogramming

In C++ Metaprogramming refers to the use of macros or templates to generate code at compile-time.

In general, macros are frowned upon in this role and templates are preferred, although they are not as generic.

Template metaprogramming often makes use of compile-time computations, whether via templates or `constexpr` functions, to achieve its goals of generating code, however compile-time computations are not metaprogramming per se.

Section 16.1: Calculating Factorials

Factorials can be computed at compile-time using template metaprogramming techniques.

```
#include <iostream>

template<unsigned int n>
struct factorial
{
    enum
    {
        value = n * factorial<n - 1>::value
    };
};

template<>
struct factorial<0>
{
    enum { value = 1 };
};

int main()
{
    std::cout << factorial<7>::value << std::endl; // prints "5040"
}
```

`factorial` is a struct, but in template metaprogramming it is treated as a template metafunction. By convention, template metafunctions are evaluated by checking a particular member, either `::type` for metafunctions that result in types, or `::value` for metafunctions that generate values.

In the above code, we evaluate the `factorial` metafunction by instantiating the template with the parameters we want to pass, and using `::value` to get the result of the evaluation.

The metafunction itself relies on recursively instantiating the same metafunction with smaller values. The `factorial<0>` specialization represents the terminating condition. Template metaprogramming has most of the restrictions of a functional programming language, so recursion is the primary “looping” construct.

Since template metafunctions execute at compile time, their results can be used in contexts that require compile-time values. For example:

```
int my_array[factorial<5>::value];
```

Automatic arrays must have a compile-time defined size. And the result of a metafunction is a compile-time constant, so it can be used here.

Limitation: Most of the compilers won't allow recursion depth beyond a limit. For example, g++ compiler by default

将递归深度限制为256层。对于g++，程序员可以使用-ftemplate-depth-x选项设置递归深度。

版本 \geq C++11

自C++11以来，可以使用std::integral_constant模板进行此类模板计算：

```
#include <iostream>
#include <type_traits>

template<long long n>
struct factorial {
    std::integral_constant<long long, n * factorial<n - 1>::value> {};
};

template<>
struct factorial<0> {
    std::integral_constant<long long, 1> {};
};

int main()
{
    std::cout << factorial<7>::value << std::endl; // 输出 "5040"
}
```

此外，constexpr函数成为了更简洁的替代方案。

```
#include <iostream>

constexpr long long factorial(long long n)
{
    return (n == 0) ? 1 : n * factorial(n - 1);
}

int main()
{
    char test[factorial(3)]; std::cout
    << factorial(7) << "\n";
}
```

factorial()的函数体写成单条语句，是因为在C++11中constexpr函数只能使用语言的一个非常有限的子集。

版本 \geq C++14

自C++14起，许多对constexpr函数的限制被取消，现在可以更方便地编写：

```
constexpr long long factorial(long long n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
```

甚至：

```
constexpr long long factorial(int n)
{
    long long result = 1;
```

limits recursion depth to 256 levels. In case of g++, programmer can set recursion depth using -ftemplate-depth-X option.

Version \geq C++11

Since C++11, the std::integral_constant template can be used for this kind of template computation:

```
#include <iostream>
#include <type_traits>

template<long long n>
struct factorial {
    std::integral_constant<long long, n * factorial<n - 1>::value> {};
};

template<>
struct factorial<0> {
    std::integral_constant<long long, 1> {};
};

int main()
{
    std::cout << factorial<7>::value << std::endl; // prints "5040"
}
```

Additionally, constexpr functions become a cleaner alternative.

```
#include <iostream>

constexpr long long factorial(long long n)
{
    return (n == 0) ? 1 : n * factorial(n - 1);
}

int main()
{
    char test[factorial(3)];
    std::cout << factorial(7) << '\n';
}
```

The body of factorial() is written as a single statement because in C++11 constexpr functions can only use a quite limited subset of the language.

Version \geq C++14

Since C++14, many restrictions for constexpr functions have been dropped and they can now be written much more conveniently:

```
constexpr long long factorial(long long n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
```

Or even:

```
constexpr long long factorial(int n)
{
    long long result = 1;
```

```

for (int i = 1; i <= n; ++i) {
    result *= i;
}
return result;
}

```

版本 ≥ C++17

自 C++17 起，可以使用折叠表达式来计算阶乘：

```

#include <iostream>
#include <utility>

template <class T, T N, class I = std::make_integer_sequence<T, N>>
struct factorial;

模板 <类 T, T N, T... 是>
struct factorial<T,N,std::index_sequence<T, Is...>> {
    static constexpr T value = (static_cast<T>(1) * ... * (Is + 1));
};

int main() {
std::cout << factorial<int, 5>::value << std::endl;
}

```

第16.2节：遍历参数包

通常，我们需要对变参模板参数包中的每个元素执行操作。有许多方法可以做到这一点，且随着C++17的出现，解决方案变得更容易读写。假设我们只是想打印包中的每个元素。最简单的解决方案是递归：

```

版本 ≥ C++11
void print_all(std::ostream& os) {
    // 基础情况
}

template <class T, class... Ts>
void print_all(std::ostream& os, T const& first, Ts const&... rest) {
    os << first;

    print_all(os, rest...);
}

```

我们也可以使用展开技巧，在单个函数中完成所有的流操作。这种方法的优点是不需要第二个重载，但缺点是可读性不佳：

```

版本 ≥ C++11
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    using expander = int[];
    (void)expander{0,
        (void)(os << args), 0}...
}

```

有关此工作原理的解释，请参见T.C的精彩回答。

版本 ≥ C++17

在C++17中，我们获得了解决此问题的两个强大新工具。第一个是折叠表达式：

```

for (int i = 1; i <= n; ++i) {
    result *= i;
}
return result;
}

```

Version ≥ C++17

Since C++17 one can use fold expression to calculate factorial:

```

#include <iostream>
#include <utility>

template <class T, T N, class I = std::make_integer_sequence<T, N>>
struct factorial;

template <class T, T N, T... Is>
struct factorial<T,N,std::index_sequence<T, Is...>> {
    static constexpr T value = (static_cast<T>(1) * ... * (Is + 1));
};

int main() {
    std::cout << factorial<int, 5>::value << std::endl;
}

```

Section 16.2: Iterating over a parameter pack

Often, we need to perform an operation over every element in a variadic template parameter pack. There are many ways to do this, and the solutions get easier to read and write with C++17. Suppose we simply want to print every element in a pack. The simplest solution is to recurse:

```

Version ≥ C++11
void print_all(std::ostream& os) {
    // base case
}

template <class T, class... Ts>
void print_all(std::ostream& os, T const& first, Ts const&... rest) {
    os << first;

    print_all(os, rest...);
}

```

We could instead use the expander trick, to perform all the streaming in a single function. This has the advantage of not needing a second overload, but has the disadvantage of less than stellar readability:

```

Version ≥ C++11
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    using expander = int[];
    (void)expander{0,
        (void)(os << args), 0}...
}

```

For an explanation of how this works, see [T.C's excellent answer](#).

Version ≥ C++17

With C++17, we get two powerful new tools in our arsenal for solving this problem. The first is a fold-expression:

```
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    ((os << args), ...);
}
```

第二个是if constexpr，它允许我们将原来的递归解决方案写成一个函数：

```
template <class T, class... Ts>
void print_all(std::ostream& os, T const& first, Ts const&... rest) {
    os << first;

    if constexpr (sizeof...(rest) > 0) {
        // 只有当还有更多参数时，这行代码才会被实例化。
        // 如果rest...为空，则不会调用
        // print_all(os)。
    print_all(os, rest...);
    }
}
```

第16.3节：使用std::integer_sequence进行迭代

自C++14起，标准提供了类模板

```
template <class T, T... Ints>
class integer_sequence;

template <std::size_t... Ints>
using index_sequence = std::integer_sequence<std::size_t, Ints...>;
```

以及一个生成它的元函数：

```
template <class T, T N>
using make_integer_sequence = std::integer_sequence<T, /* 一个序列 0, 1, 2, ..., N-1 */ >;

template<std::size_t N>
using make_index_sequence = make_integer_sequence<std::size_t, N>;
```

虽然这在C++14中是标准提供的，但也可以使用C++11的工具来实现。

我们可以使用这个工具来调用带有std::tuple参数的函数（在C++17中标准化为std::apply）：

```
namespace detail {
    template <class F, class Tuple, std::size_t... Is>
    decltype(auto) apply_impl(F&& f, Tuple&& tpl, std::index_sequence<Is...>) {
        return std::forward<F>(f)(std::get<Is>(std::forward<Tuple>(tpl))...);
    }

    template <class F, class Tuple>
    decltype(auto) apply(F&& f, Tuple&& tpl) {
        return detail::apply_impl(std::forward<F>(f),
            std::forward<Tuple>(tpl),
            std::make_index_sequence<std::tuple_size<std::decay_t<Tuple>>::value>{});
    }

    // 这将打印 3
    int f(int, char, double);
}
```

```
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    ((os << args), ...);
}
```

And the second is `if constexpr`, which allows us to write our original recursive solution in a single function:

```
template <class T, class... Ts>
void print_all(std::ostream& os, T const& first, Ts const&... rest) {
    os << first;

    if constexpr (sizeof...(rest) > 0) {
        // this line will only be instantiated if there are further
        // arguments. if rest... is empty, there will be no call to
        // print_all(os).
        print_all(os, rest...);
    }
}
```

Section 16.3: Iterating with std::integer_sequence

Since C++14, the standard provides the class template

```
template <class T, T... Ints>
class integer_sequence;

template <std::size_t... Ints>
using index_sequence = std::integer_sequence<std::size_t, Ints...>;
```

and a generating metafunction for it:

```
template <class T, T N>
using make_integer_sequence = std::integer_sequence<T, /* a sequence 0, 1, 2, ..., N-1 */ >;

template<std::size_t N>
using make_index_sequence = make_integer_sequence<std::size_t, N>;
```

While this comes standard in C++14, this can be implemented using C++11 tools.

We can use this tool to call a function with a `std::tuple` of arguments (standardized in C++17 as `std::apply`):

```
namespace detail {
    template <class F, class Tuple, std::size_t... Is>
    decltype(auto) apply_impl(F&& f, Tuple&& tpl, std::index_sequence<Is...>) {
        return std::forward<F>(f)(std::get<Is>(std::forward<Tuple>(tpl))...);
    }

    template <class F, class Tuple>
    decltype(auto) apply(F&& f, Tuple&& tpl) {
        return detail::apply_impl(std::forward<F>(f),
            std::forward<Tuple>(tpl),
            std::make_index_sequence<std::tuple_size<std::decay_t<Tuple>>::value>{});
    }

    // this will print 3
    int f(int, char, double);
}
```

```
auto some_args = std::make_tuple(42, 'x', 3.14);
int r = apply(f, some_args); // 调用 f(42, 'x', 3.14)
```

第16.4节：标签分发

在编译时选择函数的一种简单方法是将函数分发到一对重载函数，这些函数将标签作为其中一个（通常是最后一个）参数。例如，为了实现 `std::advance()`，我们可以根据迭代器类别进行分发：

```
namespace details {
    template <class RAIter, class Distance>
    void advance(RAIter& it, Distance n, std::random_access_iterator_tag) {
        it += n;
    }

    template <class 双向迭代器, class 距离>
    void advance(双向迭代器& it, 距离 n, std::bidirectional_iterator_tag) {
        if (n > 0) {
            while (n--) ++it;
        }
        else {
            while (n++) --it;
        }
    }

    template <class 输入迭代器, class 距离>
    void advance(输入迭代器& it, 距离 n, std::input_iterator_tag) {
        while (n--) {
            ++it;
        }
    }

    template <class 迭代器, class 距离>
    void advance(迭代器& it, 距离 n) {
        details::advance(it, n,
                        typename std::iterator_traits<迭代器>::iterator_category{});
    }
}
```

重载的 `details::advance` 函数中的 `std::XY_iterator_tag` 参数是未使用的函数参数。实际实现并不重要（实际上它是完全空的）。它们唯一的目的是允许编译器根据调用 `details::advance` 时所用的标签类选择重载。

在此示例中，`advance` 使用 `iterator_traits<T>::iterator_category` 元函数，该函数根据 `迭代器` 的实际类型返回其中一个 `iterator_tag` 类。然后，默认构造的 `iterator_category<迭代器>::type` 对象让编译器选择 `details::advance` 的不同重载之一。

（此函数参数很可能被完全优化掉，因为它是一个空的默认构造对象
结构体且从未使用过。）

标签分发可以让你的代码比使用SFINAE和enable_if的等价代码更易读。

注意：虽然C++17的`if constexpr`可能简化了特别是`advance`的实现，但它不适合像标签分发那样的开放式实现。

第16.5节：检测表达式是否有效

可以检测某个操作符或函数是否可以在某个类型上调用。要测试一个类是否重载了

```
auto some_args = std::make_tuple(42, 'x', 3.14);
int r = apply(f, some_args); // calls f(42, 'x', 3.14)
```

Section 16.4: Tag Dispatching

A simple way of selecting between functions at compile time is to dispatch a function to an overloaded pair of functions that take a tag as one (usually the last) argument. For example, to implement `std::advance()`, we can dispatch on the iterator category:

```
namespace details {
    template <class RAIter, class Distance>
    void advance(RAIter& it, Distance n, std::random_access_iterator_tag) {
        it += n;
    }

    template <class BidirIter, class Distance>
    void advance(BidirIter& it, Distance n, std::bidirectional_iterator_tag) {
        if (n > 0) {
            while (n--) ++it;
        }
        else {
            while (n++) --it;
        }
    }

    template <class InputIter, class Distance>
    void advance(InputIter& it, Distance n, std::input_iterator_tag) {
        while (n--) {
            ++it;
        }
    }

    template <class Iter, class Distance>
    void advance(Iter& it, Distance n) {
        details::advance(it, n,
                        typename std::iterator_traits<Iter>::iterator_category{});
    }
}
```

The `std::XY_iterator_tag` arguments of the overloaded `details::advance` functions are unused function parameters. The actual implementation does not matter (actually it is completely empty). Their only purpose is to allow the compiler to select an overload based on which tag class `details::advance` is called with.

In this example, `advance` uses the `iterator_traits<T>::iterator_category` metafunction which returns one of the `iterator_tag` classes, depending on the actual type of `Iter`. A default-constructed object of the `iterator_category<Iter>::type` then lets the compiler select one of the different overloads of `details::advance`. (This function parameter is likely to be completely optimized away, as it is a default-constructed object of an empty `struct` and never used.)

Tag dispatching can give you code that's much easier to read than the equivalents using SFINAE and enable_if.

Note: while C++17's `if constexpr` may simplify the implementation of `advance` in particular, it is not suitable for open implementations unlike tag dispatching.

Section 16.5: Detect Whether Expression is Valid

It is possible to detect whether an operator or function can be called on a type. To test if a class has an overload of

std::hash, 可以这样做：

```
#include <functional> // 用于 std::hash
#include <type_traits> // 用于 std::false_type 和 std::true_type
#include <utility> // 用于 std::declval

template<class, class = void>
struct has_hash
    : std::false_type
{};

template<class T>
struct has_hash<T, decltype(std::hash<T>()(std::declval<T>()), void())>
    : std::true_type
{};

版本 ≥ C++17
```

自 C++17 起, std::void_t 可用于简化此类结构

```
#include <functional> // 用于 std::hash
#include <type_traits> // 用于 std::false_type, std::true_type, std::void_t
#include <utility> // 用于 std::declval

template<class, class = std::void_t> >
struct has_hash
    : std::false_type
{};

template<class T>
struct has_hash<T, std::void_t< decltype(std::hash<T>()(std::declval<T>())) >>
    : std::true_type
{};


```

其中 std::void_t 定义为：

```
template< class... > using void_t = void;
```

要检测某个运算符是否定义，例如 operator<，语法几乎相同：

```
template<class, class = void>
struct has_less_than
    : std::false_type
{};

template<class T>
struct has_less_than<T, decltype(std::declval<T>() < std::declval<T>(), void())>
    : std::true_type
{};


```

这些可以用来使用 std::unordered_map<T>, 前提是 T 对 std::hash 有重载，否则尝试使用 std::map<T>：

```
template <class K, class V>
using hash_invariant_map = std::conditional_t<
    has_hash<K>::value,
    std::unordered_map<K, V>,
    std::map<K, V>>;
```

std::hash, one can do this:

```
#include <functional> // 用于 std::hash
#include <type_traits> // 用于 std::false_type 和 std::true_type
#include <utility> // 用于 std::declval

template<class, class = void>
struct has_hash
    : std::false_type
{};

template<class T>
struct has_hash<T, decltype(std::hash<T>()(std::declval<T>()), void())>
    : std::true_type
{};

Version ≥ C++17
```

Since C++17, std::void_t can be used to simplify this type of construct

```
#include <functional> // 用于 std::hash
#include <type_traits> // 用于 std::false_type, std::true_type, std::void_t
#include <utility> // 用于 std::declval

template<class, class = std::void_t> >
struct has_hash
    : std::false_type
{};

template<class T>
struct has_hash<T, std::void_t< decltype(std::hash<T>()(std::declval<T>())) >>
    : std::true_type
{};


```

where std::void_t is defined as:

```
template< class... > using void_t = void;
```

For detecting if an operator, such as operator< is defined, the syntax is almost the same:

```
template<class, class = void>
struct has_less_than
    : std::false_type
{};

template<class T>
struct has_less_than<T, decltype(std::declval<T>() < std::declval<T>(), void())>
    : std::true_type
{};


```

These can be used to use a std::unordered_map<T> if T has an overload for std::hash, but otherwise attempt to use a std::map<T>:

```
template <class K, class V>
using hash_invariant_map = std::conditional_t<
    has_hash<K>::value,
    std::unordered_map<K, V>,
    std::map<K, V>>;
```

第16.6节：条件语句 if-then-else

版本 ≥ C++11

标准库头文件`<type_traits>`中的类型`std::conditional`可以基于编译时的布尔值选择其中一个类型：

```
template<typename T>
struct ValueOrPointer
{
    typename std::conditional<(sizeof(T) > sizeof(void*)), T*, T>::type vop;
};
```

该结构体包含一个指向T的指针（如果T的大小大于指针的大小），或者包含T本身（如果其大小小于或等于指针的大小）。因此`sizeof(ValueOrPointer)`总是小于等于`sizeof(void*)`。

第16.7节：给定任意类型T时的手动类型区分

在使用`std::enable_if`实现SFINAE时，通常需要访问辅助模板来判断给定类型T是否符合一组条件。

为帮助我们实现这一点，标准已经提供了两种类似于`true`和`false`的类型，分别是`std::true_type`和`std::false_type`。

以下示例展示了如何检测类型T是否为指针，`is_pointer`模板模仿了标准库`std::is_pointer`辅助模板的行为：

```
template <typename T>
struct is_pointer_ : std::false_type {};

template <typename T>
struct is_pointer_<T*> : std::true_type {};

template <typename T>
struct is_pointer : is_pointer_<typename std::remove_cv<T>::type> { }
```

上述代码包含三个步骤（有时只需两个）：

1. `is_pointer_`的第一个声明是默认情况，继承自`std::false_type`。该默认情况应始终继承自`std::false_type`，因为它相当于“`false`条件”。
2. 第二个声明针对指针类型`T*`特化了`is_pointer_`模板，而不关心`T`具体是什么。该版本继承自`std::true_type`。
3. 第三个声明（真正的声明）只是从`T`中移除任何不必要的信息（在本例中我们移除`const`和`volatile`限定符）然后回退到之前的两个声明之一。

由于`is_pointer<T>`是一个类，要访问其值你需要：

- 使用`::value`，例如`is_pointer<int>::value`-`value`是一个静态类成员，类型为`bool`，继承自`std::true_type`或`std::false_type`；
- 构造该类型的对象，例如`is_pointer<int>{}`-这可行是因为`std::is_pointer`继承了其默认构造函数，来自`std::true_type`或`std::false_type`（它们具有`constexpr`构造函数），且`std::true_type`和`std::false_type`都具有`constexpr`转换为`bool`的转换操作符。

Section 16.6: If-then-else

Version ≥ C++11

The type `std::conditional` in the standard library header `<type_traits>` can select one type or the other, based on a compile-time boolean value:

```
template<typename T>
struct ValueOrPointer
{
    typename std::conditional<(sizeof(T) > sizeof(void*)), T*, T>::type vop;
};
```

This struct contains a pointer to `T` if `T` is larger than the size of a pointer, or `T` itself if it is smaller or equal to a pointer's size. Therefore `sizeof(ValueOrPointer)` will always be <= `sizeof(void*)`.

Section 16.7: Manual distinction of types when given any type T

When implementing SFINAE using `std::enable_if`, it is often useful to have access to helper templates that determines if a given type `T` matches a set of criteria.

To help us with that, the standard already provides two types analog to `true` and `false` which are `std::true_type` and `std::false_type`.

The following example show how to detect if a type `T` is a pointer or not, the `is_pointer` template mimic the behavior of the standard `std::is_pointer` helper:

```
template <typename T>
struct is_pointer_ : std::false_type {};

template <typename T>
struct is_pointer_<T*> : std::true_type {};

template <typename T>
struct is_pointer : is_pointer_<typename std::remove_cv<T>::type> { }
```

There are three steps in the above code (sometimes you only need two):

1. The first declaration of `is_pointer_` is the *default case*, and inherits from `std::false_type`. The *default case* should always inherit from `std::false_type` since it is analogous to a "`false` condition".
2. The second declaration specialize the `is_pointer_` template for pointer `T*` without caring about what `T` is really. This version inherits from `std::true_type`.
3. The third declaration (the real one) simply remove any unnecessary information from `T` (in this case we remove `const` and `volatile` qualifiers) and then fall backs to one of the two previous declarations.

Since `is_pointer<T>` is a class, to access its value you need to either:

- Use `::value`, e.g. `is_pointer<int>::value` - `value` is a static class member of type `bool` inherited from `std::true_type` or `std::false_type`;
- Construct an object of this type, e.g. `is_pointer<int>{}` - This works because `std::is_pointer` inherits its default constructor from `std::true_type` or `std::false_type` (which have `constexpr` constructors) and both `std::true_type` and `std::false_type` have `constexpr` conversion operators to `bool`.

提供“辅助模板”以便直接访问值是一个好习惯：

```
template <typename T>
constexpr bool is_pointer_v = is_pointer<T>::value;
```

版本 ≥ C++17

在C++17及以上版本，大多数辅助模板已经提供了_v版本，例如：

```
template< class T > constexpr bool is_pointer_v = is_pointer<T>::value;
template< class T > constexpr bool is_reference_v = is_reference<T>::value;
```

第16.8节：使用C++11（及更高版本）计算幂

使用 C++11 及更高版本，编译时计算可以变得更加简单。例如，在编译时计算给定数字的幂将如下所示：

```
template <typename T>
constexpr T calculatePower(T value, unsigned power) {
    return power == 0 ? 1 : value * calculatePower(value, power-1);
}
```

关键字`constexpr`负责在编译时计算函数，且仅当满足所有要求时（详见`constexpr`关键字参考），例如所有参数必须在编译时已知。

注意：在C++11中，`constexpr`函数必须仅由一个`return`语句组成。

优点：与标准的编译时计算方式相比，这种方法也适用于运行时计算。也就是说，如果函数的参数在编译时未知（例如`value`和`power`作为用户输入），则函数在运行时执行，因此无需重复代码（这在旧版C++标准中是必须的）。

例如

```
void useExample() {
    constexpr int compileTimeCalculated = calculatePower(3, 3); // 在编译时计算,
                                                                // 因为两个参数在编译时已知
                                                                // 并用于常量表达式。
    int value;
    std::cin >> value;
    int runtimeCalculated = calculatePower(value, 3); // 运行时计算,
                                                       // 因为value仅在运行时已知。
}
```

版本 ≥ C++17

另一种在编译时计算幂的方法可以利用折叠表达式，如下所示：

```
#include <iostream>
#include <utility>

template <class T, T V, T N, class I = std::make_integer_sequence<T, N>>
struct power;

template <class T, T V, T N, T... Is>
struct power<T, V, N, std::integer_sequence<T, Is...>> {
    static constexpr T value = (static_cast<T>(1) * ... * (V * static_cast<bool>(Is + 1)));
};
```

It is a good habit to provide "helper helper templates" that let you directly access the value:

```
template <typename T>
constexpr bool is_pointer_v = is_pointer<T>::value;
```

Version ≥ C++17

In C++17 and above, most helper templates already provide a _v version, e.g.:

```
template< class T > constexpr bool is_pointer_v = is_pointer<T>::value;
template< class T > constexpr bool is_reference_v = is_reference<T>::value;
```

Section 16.8: Calculating power with C++11 (and higher)

With C++11 and higher calculations at compile time can be much easier. For example calculating the power of a given number at compile time will be following:

```
template <typename T>
constexpr T calculatePower(T value, unsigned power) {
    return power == 0 ? 1 : value * calculatePower(value, power-1);
}
```

Keyword `constexpr` is responsible for calculating function in compilation time, then and only then, when all the requirements for this will be met (see more at `constexpr` keyword reference) for example all the arguments must be known at compile time.

Note: In C++11 `constexpr` function must consist only from one return statement.

Advantages: Comparing this to the standard way of compile time calculation, this method is also useful for runtime calculations. It means, that if the arguments of the function are not known at the compilation time (e.g. `value` and `power` are given as input via user), then function is run in a compilation time, so there's no need to duplicate a code (as we would be forced in older standards of C++)�.

E.g.

```
void useExample() {
    constexpr int compileTimeCalculated = calculatePower(3, 3); // computes at compile time,
                                                                // as both arguments are known at compilation time
                                                                // and used for a constant expression.
    int value;
    std::cin >> value;
    int runtimeCalculated = calculatePower(value, 3); // runtime calculated,
                                                       // because value is known only at runtime.
}
```

Version ≥ C++17

Another way to calculate power at compile time can make use of fold expression as follows:

```
#include <iostream>
#include <utility>

template <class T, T V, T N, class I = std::make_integer_sequence<T, N>>
struct power;

template <class T, T V, T N, T... Is>
struct power<T, V, N, std::integer_sequence<T, Is...>> {
    static constexpr T value = (static_cast<T>(1) * ... * (V * static_cast<bool>(Is + 1)));
};
```

```
int main() {
std::cout << power<int, 4, 2>::value << std::endl;
}
```

第16.9节：支持可变参数数量的通用最小值/最大值

版本 > C++11

可以通过模板元编程编写一个通用函数（例如min），它接受各种数值类型和任意数量的参数。该函数为两个参数声明了一个min，并递归处理更多参数。

```
template <typename T1, typename T2>
auto min(const T1 &a, const T2 &b)
-> typename std::common_type<const T1&, const T2&>::type
{
    return a < b ? a : b;
}

template <typename T1, typename T2, typename ... Args>
auto min(const T1 &a, const T2 &b, const Args& ... args)
-> typename std::common_type<const T1&, const T2&, const Args& ...>::type
{
    return min(min(a, b), args...);
}

auto minimum = min(4, 5.8f, 3, 1.8, 3, 1.1, 9);
```

```
int main() {
std::cout << power<int, 4, 2>::value << std::endl;
}
```

Section 16.9: Generic Min/Max with variable argument count

Version > C++11

It's possible to write a generic function (for example min) which accepts various numerical types and arbitrary argument count by template meta-programming. This function declares a min for two arguments and recursively for more.

```
template <typename T1, typename T2>
auto min(const T1 &a, const T2 &b)
-> typename std::common_type<const T1&, const T2&>::type
{
    return a < b ? a : b;
}

template <typename T1, typename T2, typename ... Args>
auto min(const T1 &a, const T2 &b, const Args& ... args)
-> typename std::common_type<const T1&, const T2&, const Args& ...>::type
{
    return min(min(a, b), args...);
}

auto minimum = min(4, 5.8f, 3, 1.8, 3, 1.1, 9);
```

第17章：const关键字

第17.1节：避免const和非const getter方法中代码重复

在C++中，仅通过const限定符不同的方法可以重载。有时可能需要两个版本的getter，返回某个成员的引用。

设Foo为一个类，它有两个执行相同操作并返回类型为Bar对象引用的方法：

```
class Foo
{
public:
    Bar& GetBar(/* 一些参数 */)
    {
        /* 一些计算 */
        return bar;
    }

    const Bar& GetBar(/* 一些参数 */) const
    {
        /* 一些计算 */
        return bar;
    }

    // ...
};
```

它们之间唯一的区别是，一个方法是非常量的并返回非常量引用（可用于修改对象），另一个是常量的并返回常量引用。

为了避免代码重复，通常会想从一个方法调用另一个方法。然而，我们不能从常量方法调用非常量方法。但我们可以从非常量方法调用常量方法。这将要求我们使用 'const_cast' 来去除常量限定符。

解决方案是：

```
struct Foo
{
    Bar& GetBar(/*参数*/)
    {
        return const_cast<Bar&>(const_cast<const Foo*>(this)->GetBar(/*arguments*/));
    }

    const Bar& GetBar(/*arguments*/) const
    {
        /* 一些计算 */
        return foo;
    }
};
```

在上面的代码中，我们通过将 this 转换为 const 类型，从非 const 的 GetBar 调用 const 版本的 GetBar：
const_cast<const Foo*>(this)。由于我们从非 const 调用 const 方法，对象本身是非 const 的，因此允许去除 const。

Chapter 17: const keyword

Section 17.1: Avoiding duplication of code in const and non-const getter methods

In C++ methods that differs only by `const` qualifier can be overloaded. Sometimes there may be a need of two versions of getter that return a reference to some member.

Let Foo be a class, that has two methods that perform identical operations and returns a reference to an object of type Bar:

```
class Foo
{
public:
    Bar& GetBar(/* some arguments */)
    {
        /* some calculations */
        return bar;
    }

    const Bar& GetBar(/* some arguments */) const
    {
        /* some calculations */
        return bar;
    }

    // ...
};
```

The only difference between them is that one method is non-const and return a non-const reference (that can be used to modify object) and the second is const and returns const reference.

To avoid the code duplication, there is a temptation to call one method from another. However, we can not call non-const method from the const one. But we can call const method from non-const one. That will require us to use 'const_cast' to remove the const qualifier.

The solution is:

```
struct Foo
{
    Bar& GetBar(/arguments/)
    {
        return const_cast<Bar&>(const_cast<const Foo*>(this)->GetBar(/arguments__));
    }

    const Bar& GetBar(/arguments/) const
    {
        /* some calculations */
        return foo;
    }
};
```

In code above, we call const version of GetBar from the non-const GetBar by casting this to const type:
const_cast<const Foo*>(this). Since we call const method from non-const, the object itself is non-const, and casting away the const is allowed.

请查看以下更完整的示例：

```
#include <iostream>

class Student
{
public:
    char& GetScore(bool midterm)
    {
        return const_cast<char&>(const_cast<const Student*>(this)->GetScore(midterm));
    }

    const char& GetScore(bool midterm) const
    {
        if (midterm)
        {
            return midtermScore;
        }
        else
        {
            return finalScore;
        }
    }

private:
    char midtermScore;
    char finalScore;
};

int main()
{
    // 非const对象
    Student a;
    // 我们可以给引用赋值。调用的是非const版本的GetScore
    a.GetScore(true) = 'B';
    a.GetScore(false) = 'A';

    // const object
    const Student b(a);
    // 我们仍然可以调用 const 对象的 GetScore 方法，// 因为我们重载
    // 了 const 版本的 GetScore
    std::cout << b.GetScore(true) << b.GetScore(false) << "\n";
}
```

第17.2节：常量成员函数

类的成员函数可以声明为 `const`，这告诉编译器和未来的读者该函数不会修改对象：

```
class MyClass
{
private:
    int myInt_;
public:
    int myInt() const { return myInt_; }
    void setMyInt(int myInt) { myInt_ = myInt; }
};
```

在 `const` 成员函数中，`this` 指针实际上是一个 `const MyClass *` 而不是 `MyClass *`。这意味着你不能在函数内修改任何成员变量；编译器会发出警告。因此 `setMyInt`

Examine the following more complete example:

```
#include <iostream>

class Student
{
public:
    char& GetScore(bool midterm)
    {
        return const_cast<char&>(const_cast<const Student*>(this)->GetScore(midterm));
    }

    const char& GetScore(bool midterm) const
    {
        if (midterm)
        {
            return midtermScore;
        }
        else
        {
            return finalScore;
        }
    }

private:
    char midtermScore;
    char finalScore;
};

int main()
{
    // non-const object
    Student a;
    // We can assign to the reference. Non-const version of GetScore is called
    a.GetScore(true) = 'B';
    a.GetScore(false) = 'A';

    // const object
    const Student b(a);
    // We still can call GetScore method of const object,
    // because we have overloaded const version of GetScore
    std::cout << b.GetScore(true) << b.GetScore(false) << '\n';
}
```

Section 17.2: Const member functions

Member functions of a class can be declared `const`, which tells the compiler and future readers that this function will not modify the object:

```
class MyClass
{
private:
    int myInt_;
public:
    int myInt() const { return myInt_; }
    void setMyInt(int myInt) { myInt_ = myInt; }
};
```

In a `const` member function, the `this` pointer is effectively a `const MyClass *` instead of a `MyClass *`. This means that you cannot change any member variables within the function; the compiler will emit a warning. So `setMyInt`

不能被声明为const。

你几乎总是应该在可能的情况下将成员函数标记为const。只有const成员函数才能被调用用于const MyClass对象上。

静态方法不能声明为const。这是因为静态方法属于类，而不是在对象上调用；因此它永远不会修改对象的内部变量。所以将static方法声明为const是多余的。

第17.3节：常量局部变量

声明和使用。

```
// a 是常量整型，所以不能被修改
const int a = 15;
a = 12;           // 错误：不能给常量变量赋新值
a += 1;          // 错误：不能给常量变量赋新值
```

引用和指针的绑定

```
int &b = a;      // 错误：不能将非常量引用绑定到常量变量
const int &c = a; // 正确；c 是常量引用

int *d = &a;     // 错误：不能将指向非常量的指针绑定到常量变量
const int *e = &a // 正确；e 是指向常量的指针

int f = 0;
e = &f;          // 正确；e 是一个非常量指向常量的指针，
                  // 这意味着它可以重新绑定到新的 int* 或 const int*

*e = 1           // 错误：e 是指向常量的指针，意味着
                  // 不能通过解引用 e 来修改它指向的值

int *g = &f;
*g = 1;          // 好的；这个值仍然可以通过取消引用一个非const指针来更改
```

could not be declared const.

You should almost always mark member functions as const when possible. Only const member functions can be called on a const MyClass.

static methods cannot be declared as const. This is because a static method belongs to a class and is not called on object; therefore it can never modify object's internal variables. So declaring static methods as const would be redundant.

Section 17.3: Const local variables

Declaration and usage.

```
// a is const int, so it can't be changed
const int a = 15;
a = 12;           // Error: can't assign new value to const variable
a += 1;          // Error: can't assign new value to const variable
```

Binding of references and pointers

```
int &b = a;      // Error: can't bind non-const reference to const variable
const int &c = a; // OK; c is a const reference

int *d = &a;     // Error: can't bind pointer-to-non-const to const variable
const int *e = &a // OK; e is a pointer-to-const

int f = 0;
e = &f;          // OK; e is a non-const pointer-to-const,
                  // which means that it can be rebound to new int* or const int*

*e = 1           // Error: e is a pointer-to-const which means that
                  // the value it points to can't be changed through dereferencing e

int *g = &f;
*g = 1;          // OK; this value still can be changed through dereferencing
                  // a pointer-not-to-const
```

第17.4节：常量指针

```
int a = 0, b = 2;

const int* pA = &a; // 指向常量的指针。不能通过此指针修改'a'
int* const pB = &a; // 常量指针。`a`可以被修改，但此指针本身不能修改。
const int* const pC = &a; // 指向常量的常量指针。

// 错误：不能赋值给常量引用
*pA = b;

pA = &b;

// 错误：不能赋值给常量指针
pB = &b;

// 错误：不能赋值给常量引用
*pC = b;
```

Section 17.4: Const pointers

```
int a = 0, b = 2;

const int* pA = &a; // pointer-to-const. `a` can't be changed through this
int* const pB = &a; // const pointer. `a` can be changed, but this pointer can't.
const int* const pC = &a; // const pointer-to-const.

// Error: Cannot assign to a const reference
*pA = b;

pA = &b;

// Error: Cannot assign to const pointer
pB = &b;

// Error: Cannot assign to a const reference
*pC = b;
```

```
//错误：无法赋值给常量指针  
pC = &b;
```

```
//Error: Cannot assign to const pointer  
pC = &b;
```

第18章：mutable关键字

第18.1节：mutable lambda表达式

默认情况下，lambda的隐式operator()是const的。这不允许对lambda执行非const操作。为了允许修改成员，lambda可以被标记为mutable，这使得隐式的operator()变为非const：

```
int a = 0;

auto bad_counter = [a] {
    return a++; // 错误：operator() 是 const
                // 不能修改成员
};

auto good_counter = [a]() mutable {
    return a++; // 正确
}

good_counter(); // 0
good_counter(); // 1
good_counter(); // 2
```

第18.2节：非静态类成员修饰符

在此上下文中，mutable修饰符用于表示可以修改const对象的数据字段，而不影响对象的外部可见状态。

如果你正在考虑缓存一个昂贵计算的结果，你可能应该使用这个关键字。

如果你有一个锁（例如，`std::unique_lock`）数据成员，在const方法中进行加锁和解锁，你也可以使用这个关键字。

你不应该使用这个关键字来破坏对象的逻辑常量性。

带缓存的示例：

```
类 pi_calculator {
    公共:
        double get_pi() const {
            如果 (pi_calculated) {
                返回 pi;
            } 否则 {
                double new_pi = 0;
                对于 (int i = 0; i < 0000000000; ++i) {
                    // 一些计算以细化 new_pi
                }
                // 注意：如果 pi 和 pi_calculated 不是 mutable，我们将会收到错误
                编译器
                // 因为在 const 方法中我们不能修改非 mutable 字段
                pi = new_pi;
            }
            pi_calculated = true;
            返回 pi;
        }
    私有:
        mutable bool pi_calculated = false;
```

Chapter 18: mutable keyword

Section 18.1: mutable lambdas

By default, the implicit operator() of a lambda is `const`. This disallows performing non-`const` operations on the lambda. In order to allow modifying members, a lambda may be marked `mutable`, which makes the implicit operator() non-`const`:

```
int a = 0;

auto bad_counter = [a] {
    return a++; // error: operator() is const
                // cannot modify members
};

auto good_counter = [a]() mutable {
    return a++; // OK
}

good_counter(); // 0
good_counter(); // 1
good_counter(); // 2
```

Section 18.2: non-static class member modifier

`mutable` modifier in this context is used to indicate that a data field of a `const` object may be modified without affecting the externally-visible state of the object.

If you are thinking about caching a result of expensive computation, you should probably use this keyword.

If you have a lock (for example, `std::unique_lock`) data field which is locked and unlocked inside a `const` method, this keyword is also what you could use.

You should not use this keyword to break logical const-ness of an object.

Example with caching:

```
class pi_calculator {
    public:
        double get_pi() const {
            if (pi_calculated) {
                返回 pi;
            } 否则 {
                double new_pi = 0;
                for (int i = 0; i < 1000000000; ++i) {
                    // some calculation to refine new_pi
                }
                // note: if pi and pi_calculated were not mutable, we would get an error from a
                compiler
                // because in a const method we can not change a non-mutable field
                pi = new_pi;
                pi_calculated = true;
                返回 pi;
            }
        }
    私有:
        mutable bool pi_calculated = false;
```

```
    mutable double pi = 0;  
};
```

```
    mutable double pi = 0;  
};
```

第19章：Friend 关键字

设计良好的类封装其功能，隐藏其实现，同时提供一个简洁、有文档说明的接口。这允许在接口不变的前提下进行重新设计或更改。

在更复杂的场景中，可能需要多个类相互依赖对方的实现细节。友元类和函数允许这些同伴访问彼此的细节，而不会破坏已记录接口的封装性和信息隐藏。

第19.1节：友元函数

类或结构体可以声明任何函数为其友元。如果一个函数是某个类的友元函数，则它可以访问该类的所有受保护和私有成员：

```
// 函数的前置声明。
void friend_function();
void non_friend_function();

class PrivateHolder {
public:
PrivateHolder(int val) : private_value(val) {}
private:
    int private_value;
    // 将其中一个函数声明为友元函数。
    friend void friend_function();
};

void non_friend_function() {
    PrivateHolder ph(10);
    // 编译错误：private_value 是私有的。
    std::cout << ph.private_value << std::endl;
}

void friend_function() {
    // 可以：友元可以访问私有值。
    PrivateHolder ph(10);
    std::cout << ph.private_value << std::endl;
}
```

访问修饰符不会改变友元语义。友元的public、protected和private声明是等效的。

友元声明不会被继承。例如，如果我们对PrivateHolder进行子类化：

```
class PrivateHolderDerived : public PrivateHolder {
public:
PrivateHolderDerived(int val) : PrivateHolder(val) {}
private:
    int derived_private_value = 0;
};
```

并尝试访问它的成员，我们将得到以下结果：

```
void friend_function() {
    PrivateHolderDerived pd(20);
    // 可以。
    std::cout << pd.private_value << std::endl;
    // 编译错误：derived_private_value 是私有的。
```

Chapter 19: Friend keyword

Well-designed classes encapsulate their functionality, hiding their implementation while providing a clean, documented interface. This allows redesign or change so long as the interface is unchanged.

In a more complex scenario, multiple classes that rely on each others' implementation details may be required. Friend classes and functions allow these peers access to each others' details, without compromising the encapsulation and information hiding of the documented interface.

Section 19.1: Friend function

A class or a structure may declare any function it's friend. If a function is a friend of a class, it may access all its protected and private members:

```
// Forward declaration of functions.
void friend_function();
void non_friend_function();

class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
private:
    int private_value;
    // Declare one of the function as a friend.
    friend void friend_function();
};

void non_friend_function() {
    PrivateHolder ph(10);
    // Compilation error: private_value is private.
    std::cout << ph.private_value << std::endl;
}

void friend_function() {
    // OK: friends may access private values.
    PrivateHolder ph(10);
    std::cout << ph.private_value << std::endl;
}
```

Access modifiers do not alter friend semantics. Public, protected and private declarations of a friend are equivalent.

Friend declarations are not inherited. For example, if we subclass PrivateHolder:

```
class PrivateHolderDerived : public PrivateHolder {
public:
    PrivateHolderDerived(int val) : PrivateHolder(val) {}
private:
    int derived_private_value = 0;
};
```

and try to access its members, we'll get the following:

```
void friend_function() {
    PrivateHolderDerived pd(20);
    // OK.
    std::cout << pd.private_value << std::endl;
    // Compilation error: derived_private_value is private.
```

```
std::cout << pd.derived_private_value << std::endl;
}
```

注意PrivateHolderDerived成员函数无法访问PrivateHolder::private_value，但友元函数可以访问。

第19.2节：友元方法

方法也可以声明为友元，就像函数一样：

```
class Accesser {
public:
    void private_accesser();
};

class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
    friend void Accesser::private_accesser();
private:
    int private_value;
};

void Accesser::private_accesser() {
    PrivateHolder ph(10);
    // OK: 此方法被声明为友元。
    std::cout << ph.private_value << std::endl;
}
```

```
    std::cout << pd.derived_private_value << std::endl;
}
```

Note that PrivateHolderDerived member function cannot access PrivateHolder::private_value, while friend function can do it.

Section 19.2: Friend method

Methods may declared as friends as well as functions:

```
class Accesser {
public:
    void private_accesser();
};

class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
    friend void Accesser::private_accesser();
private:
    int private_value;
};

void Accesser::private_accesser() {
    PrivateHolder ph(10);
    // OK: this method is declares as friend.
    std::cout << ph.private_value << std::endl;
}
```

第19.3节：友元类

整个类可以被声明为友元。友元类声明意味着该友元类的任何成员都可以访问声明类的私有和受保护成员：

```
class Accesser {
public:
    void private_accesser1();
    void private_accesser2();
};

class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
    friend class Accesser;
private:
    int private_value;
};

void Accesser::private_accesser1() {
    PrivateHolder ph(10);
    // OK.
    std::cout << ph.private_value << std::endl;
}

void Accesser::private_accesser2() {
    PrivateHolder ph(10);
    // OK.
    std::cout << ph.private_value + 1 << std::endl;
}
```

Section 19.3: Friend class

A whole class may be declared as friend. Friend class declaration means that any member of the friend may access private and protected members of the declaring class:

```
class Accesser {
public:
    void private_accesser1();
    void private_accesser2();
};

class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
    friend class Accesser;
private:
    int private_value;
};

void Accesser::private_accesser1() {
    PrivateHolder ph(10);
    // OK.
    std::cout << ph.private_value << std::endl;
}

void Accesser::private_accesser2() {
    PrivateHolder ph(10);
    // OK.
    std::cout << ph.private_value + 1 << std::endl;
}
```

```
}
```

友元类声明不是自反的。如果类需要双向的私有访问，则两者都需要友元声明。

```
class Accesser {  
public:  
    void private_accesser1();  
    void private_accesser2();  
private:  
    int private_value = 0;  
};  
  
class PrivateHolder {  
public:  
PrivateHolder(int val) : private_value(val) {}  
    // Accesser 是 PrivateHolder 的友元  
    friend class Accesser;  
    void reverse_accesse() {  
        // 但 PrivateHolder 无法访问 Accesser 的成员。  
        Accesser a;  
        std::cout << a.private_value;  
    }  
private:  
    int private_value;  
};
```

```
}
```

Friend class declaration is not reflexive. If classes need private access in both directions, both of them need friend declarations.

```
class Accesser {  
public:  
    void private_accesser1();  
    void private_accesser2();  
private:  
    int private_value = 0;  
};  
  
class PrivateHolder {  
public:  
PrivateHolder(int val) : private_value(val) {}  
    // Accesser is a friend of PrivateHolder  
    friend class Accesser;  
    void reverse_accesse() {  
        // but PrivateHolder cannot access Accesser's members.  
        Accesser a;  
        std::cout << a.private_value;  
    }  
private:  
    int private_value;  
};
```

第20章：类型关键字

第20.1节：类 (class)

1. 介绍类类型的定义。

```
class foo {
    int x;
public:
    int get_x();
    void set_x(int new_x);
};
```

2. 引入了一个详细类型说明符，它指定后面的名称是一个类类型的名称。如果类名已经声明，即使被另一个名称隐藏，也能找到它。如果类名尚未声明，则进行前向声明。

```
class foo; // 详细类型说明符 -> 前向声明
class bar {
public:
    bar(foo& f);
};

void baz();
class baz; // 另一个详细类型说明符；另一个前向声明
            // 注意：该类与函数 void baz() 同名
class foo {
bar b;
    friend class baz; // 详述类型说明符指的是类，
                      // 而不是同名的函数
public:
foo();
};
```

3. 在模板声明中引入一个类型参数。

```
template <class T>
const T& min(const T& x, const T& y) {
    return b < a ? b : a;
}
```

4. 在模板模版参数的声明中，关键字class位于参数名称之前。
由于模板模版参数的实参只能是类模板，因此这里使用class是多余的。
但是，C++的语法要求必须使用它。

```
template <template <class T> class U>
//          ^^^^^^ 此处"class"的用法；
//          U是一个模板模版参数
void f() {
U<int>::do_it();
U<double>::do_it();
}
```

5. 注意，意义2和意义3可以在同一声明中结合使用。例如：

```
template <class T>
class foo {
```

Chapter 20: Type Keywords

Section 20.1: class

1. Introduces the definition of a class type.

```
class foo {
    int x;
public:
    int get_x();
    void set_x(int new_x);
};
```

2. Introduces an *elaborated type specifier*, which specifies that the following name is the name of a class type. If the class name has been declared already, it can be found even if hidden by another name. If the class name has not been declared already, it is forward-declared.

```
class foo; // elaborated type specifier -> forward declaration
class bar {
public:
    bar(foo& f);
};

void baz();
class baz; // another elaborated type specifier; another forward declaration
            // note: the class has the same name as the function void baz()
class foo {
bar b;
    friend class baz; // elaborated type specifier refers to the class,
                      // not the function of the same name
public:
foo();
};
```

3. Introduces a type parameter in the declaration of a template.

```
template <class T>
const T& min(const T& x, const T& y) {
    return b < a ? b : a;
}
```

4. In the declaration of a template template parameter, the keyword `class` precedes the name of the parameter. Since the argument for a template template parameter can only be a class template, the use of `class` here is redundant. However, the grammar of C++ requires it.

```
template <template <class T> class U>
//          ^^^^^^ "class" used in this sense here;
//          U is a template template parameter
void f() {
U<int>::do_it();
U<double>::do_it();
}
```

5. Note that sense 2 and sense 3 may be combined in the same declaration. For example:

```
template <class T>
class foo {
```

```
};

foo<class bar> x; // <- bar 不必事先出现过。
```

版本 \geq C++11

- 在枚举的声明或定义中，声明该枚举为作用域枚举 (scoped enum) 。

```
enum class Format {
    TEXT,
    PDF,
    OTHER,
};
Format f = F::TEXT;
```

第20.2节：枚举 (enum)

- 引入枚举类型的定义。

```
enum Direction {
    UP,
    左,
    下,
    右
};
方向 d = 向上;
```

版本 \geq C++11

在 C++11 中，enum 后面可以选择性地跟随 class 或 struct 来定义一个作用域枚举。此外，作用域枚举和非作用域枚举都可以通过在枚举名后加上 : T 来显式指定其底层类型，其中 T 指的是整数类型。

```
enum class 格式 : char {
    文本,
    PDF,
    其他
};
格式 f = 格式::文本;

enum 语言 : int {
    英语,
    法语,
    其他
};
```

普通 enum 中的枚举量也可以带有作用域运算符前缀，尽管它们仍被视为定义该 enum 的作用域内的成员。

语言 l1, l2;

```
l1 = 英语;
l2 = 语言::其他;
```

- 引入了一个 详细类型说明符，指定后续的名称是之前定义过的名称声明的枚举类型。（不能使用详细类型说明符来前向声明枚举类型。）一个

```
};

foo<class bar> x; // <- bar 不必事先出现过。
```

Version \geq C++11

- In the declaration or definition of an enum, declares the enum to be a scoped enum.

```
enum class Format {
    TEXT,
    PDF,
    OTHER,
};
Format f = F::TEXT;
```

Section 20.2: enum

- Introduces the definition of an enumeration type.

```
enum Direction {
    UP,
    LEFT,
    DOWN,
    RIGHT
};
Direction d = UP;
```

Version \geq C++11

In C++11, enum may optionally be followed by class or struct to define a scoped enum. Furthermore, both scoped and unscoped enums can have their underlying type explicitly specified by : T following the enum name, where T refers to an integer type.

```
enum class Format : char {
    TEXT,
    PDF,
    OTHER
};
Format f = Format::TEXT;

enum Language : int {
    ENGLISH,
    FRENCH,
    OTHER
};
```

Enumerators in normal enums may also be preceded by the scope operator, although they are still considered to be in the scope the enum was defined in.

```
Language l1, l2;

l1 = ENGLISH;
l2 = Language::OTHER;
```

- Introduces an elaborated type specifier, which specifies that the following name is the name of a previously declared enum type. (An elaborated type specifier cannot be used to forward-declare an enum type.) An

即使被另一个名称隐藏，枚举也可以这样命名。

```
enum Foo { FOO };
void Foo() {}
Foo foo = FOO;      // 错误；Foo 指的是函数
enum Foo foo = FOO; // 正确；Foo 指的是枚举类型
```

版本 ≥ C++11

- 引入一个不透明枚举声明，该声明声明了一个枚举但未定义它。它可以重新声明一个先前声明的枚举，或者前向声明一个之前未声明的枚举。

首次声明为作用域枚举的枚举不能再声明为无作用域枚举，反之亦然。所有枚举的声明必须在底层类型上保持一致。

在前向声明无作用域枚举时，必须显式指定底层类型，因为在枚举值未知之前无法推断。

```
enum class Format; // 底层类型隐式为 int
void f(Format f);
enum class Format {
    TEXT,
    PDF,
    OTHER,
};

enum Direction; // 格式错误；必须指定底层类型
```

enum can be named in this way even if hidden by another name.

```
enum Foo { FOO };
void Foo() {}
Foo foo = FOO;      // ill-formed; Foo refers to the function
enum Foo foo = FOO; // ok; Foo refers to the enum type
```

Version ≥ C++11

- Introduces an *opaque enum declaration*, which declares an enum without defining it. It can either redeclare a previously declared enum, or forward-declare an enum that has not been previously declared.

An enum first declared as scoped cannot later be declared as unscoped, or *vice versa*. All declarations of an enum must agree in underlying type.

When forward-declaring an unscoped enum, the underlying type must be explicitly specified, since it cannot be inferred until the values of the enumerators are known.

```
enum class Format; // underlying type is implicitly int
void f(Format f);
enum class Format {
    TEXT,
    PDF,
    OTHER,
};

enum Direction; // ill-formed; must specify underlying type
```

第20.3节：结构体

与类可互换，但有以下区别：

- 如果使用关键字**struct**定义类类型，则基类和成员的默认访问权限是**public**而非**private**。
- struct**不能用于声明模板类型参数或模板模板参数；只能使用**class**。

第20.4节：联合体

- 引入联合体类型的定义。

```
// 示例来自POSIX
union sigval {
    int     sival_int;
    void   *sival_ptr;
};
```

- 引入了一个详细类型说明符，用于指定后续名称是联合体类型的名称。如果联合体名称已被声明，即使被其他名称隐藏，也能找到该名称。如果联合体名称尚未声明，则进行前向声明。

```
union foo; // 详述类型说明符 -> 前向声明
class bar {
```

Section 20.3: struct

Interchangeable with **class**, except for the following differences:

- If a class type is defined using the keyword **struct**, then the default accessibility of bases and members is **public** rather than **private**.
- struct** cannot be used to declare a template type parameter or template template parameter; only **class** can.

Section 20.4: union

- Introduces the definition of a union type.

```
// Example is from POSIX
union sigval {
    int     sival_int;
    void   *sival_ptr;
};
```

- Introduces an *elaborated type specifier*, which specifies that the following name is the name of a union type. If the union name has been declared already, it can be found even if hidden by another name. If the union name has not been declared already, it is forward-declared.

```
union foo; // elaborated type specifier -> forward declaration
class bar {
```

```
public:  
bar(foo& f);  
};  
void baz();  
union baz; // 另一个详述类型说明符；另一个前向声明  
           // 注意：该类与函数 void baz() 同名  
union foo {  
    long l;  
    union baz* b; // 详述类型说明符指向类，  
                   // 不是同名函数  
};
```

```
public:  
bar(foo& f);  
};  
void baz();  
union baz; // another elaborated type specifier; another forward declaration  
           // note: the class has the same name as the function void baz()  
union foo {  
    long l;  
    union baz* b; // elaborated type specifier refers to the class,  
                   // not the function of the same name  
};
```


类型void*（“指向void的指针”）具有任何对象指针都可以转换为该类型并再转换回原指针且结果相同的特性。此特性使得void*类型适用于某些（类型不安全的）类型擦除接口，例如C风格API中的通用上下文（如qsort、pthread_create）。

任何表达式都可以转换为void类型的表达式；这称为丢弃值表达式：

```
static_cast<void>(std::printf("Hello, %s!", name)); // 丢弃返回值
```

这对于明确表示某表达式的值不重要且该表达式仅为副作用而求值时非常有用。

第21.6节：wchar_t

一种整数类型，足够大以表示最大支持的扩展字符集中的所有字符，也称为宽字符集。（假设wchar_t使用任何特定编码，如UTF-16，是不可移植的。）

通常在需要存储超过ASCII 255的字符时使用，因为它的大小比字符类型char更大。

```
const wchar_t message_ahmaric[] = L"Hello, world\\n"; //阿姆哈拉语的"hello, world"
const wchar_t message_chinese[] = L"你好，世界\\n"; //中文的"hello, world"
const wchar_t message_hebrew[] = L"שלום מילש\\n"; //希伯来语的"hello, world"
const wchar_t message_russian[] = L"Привет мир\\n"; //俄语的"hello, world"
const wchar_t message_tamil[] = L"ஹலஹா உலகம்\\n"; //泰米尔语的"hello, world"
```

第21.7节：float

一种浮点类型。在C++的三种浮点类型中范围最窄。

```
float area(float radius) {
    const float pi = 3.14159f;
    return pi*radius*radius;
}
```

第21.8节：double

一种浮点类型。其范围包含float的范围。与long结合时，表示long double浮点类型，其范围包含double的范围。

```
double area(double radius) {
    const double pi = 3.141592653589793;
    return pi*radius*radius;
}
```

第21.9节：long

表示一种有符号整数类型，其长度至少与int相同，且范围至少包括-2147483647到+2147483647（即，-(2^31 - 1)到+(2^31 - 1)）。该类型也可以写作long int。

```
const long approx_seconds_per_year = 60L*60L*24L*365L;
```

组合long double表示一种浮点类型，在三种浮点类型中范围最广

The type void* ("pointer to void") has the property that any object pointer can be converted to it and back and result in the same pointer. This feature makes the type void* suitable for certain kinds of (type-unsafe) type-erasing interfaces, for example for generic contexts in C-style APIs (e.g. qsort, pthread_create).

Any expression may be converted to an expression of type void; this is called a *discarded-value expression*:

```
static_cast<void>(std::printf("Hello, %s!\\n", name)); // discard return value
```

This may be useful to signal explicitly that the value of an expression is not of interest and that the expression is to be evaluated for its side effects only.

Section 21.6: wchar_t

An integer type large enough to represent all characters of the largest supported extended character set, also known as the wide-character set. (It is not portable to make the assumption that wchar_t uses any particular encoding, such as UTF-16.)

It is normally used when you need to store characters over ASCII 255, as it has a greater size than the character type char.

```
const wchar_t message_ahmaric[] = L"Hello, world\\n"; //Ahmaric for "hello, world"
const wchar_t message_chinese[] = L"你好，世界\\n"; // Chinese for "hello, world"
const wchar_t message_hebrew[] = L"שלום מילש\\n"; //Hebrew for "hello, world"
const wchar_t message_russian[] = L"Привет мир\\n"; //Russian for "hello, world"
const wchar_t message_tamil[] = L"ஹலஹா உலகம்\\n"; //Tamil for "hello, world"
```

Section 21.7: float

A floating point type. Has the narrowest range out of the three floating point types in C++.

```
float area(float radius) {
    const float pi = 3.14159f;
    return pi*radius*radius;
}
```

Section 21.8: double

A floating point type. Its range includes that of float. When combined with long, denotes the long double floating point type, whose range includes that of double.

```
double area(double radius) {
    const double pi = 3.141592653589793;
    return pi*radius*radius;
}
```

Section 21.9: long

Denotes a signed integer type that is at least as long as int, and whose range includes at least -2147483647 to +2147483647, inclusive (that is, -(2^31 - 1) to +(2^31 - 1)). This type can also be written as long int.

```
const long approx_seconds_per_year = 60L*60L*24L*365L;
```

The combination long double denotes a floating point type, which has the widest range out of the three floating

的浮点类型。

```
long double area(long double radius) {  
    const long double pi = 3.1415926535897932385L;  
    return pi*radius*radius;  
}
```

版本 \geq C++11

当**long**说明符出现两次，如**long long**时，表示一种有符号整数类型，其长度至少与**long**相同，且范围至少包括-9223372036854775807到+9223372036854775807（即， $-(2^{63}-1)$ 到 $(2^{63}-1)$ ）。

```
// 支持最大文件大小为2 TiB  
const long long max_file_size = 2LL << 40;
```

第21.10节：short

表示一种有符号整数类型，其长度至少与**char**相同，且范围至少包括-32767到+32767（含）。该类型也可以写作**short int**。

```
// (在过去的一年中)  
short 工作小时数(short 工作天数) {  
    return 8*工作天数;  
}
```

第21.11节：bool

一种整数类型，其值可以是**true**或**false**。

```
bool 是偶数(int x) {  
    return x%2 == 0;  
}  
const bool b = 是偶数(47); // false
```

point types.

```
long double area(long double radius) {  
    const long double pi = 3.1415926535897932385L;  
    return pi*radius*radius;  
}
```

Version \geq C++11

When the **long** specifier occurs twice, as in **long long**, it denotes a signed integer type that is at least as long as **long**, and whose range includes at least -9223372036854775807 to +9223372036854775807, inclusive (that is, $-(2^{63}-1)$ to $(2^{63}-1)$).

```
// support files up to 2 TiB  
const long long max_file_size = 2LL << 40;
```

Section 21.10: short

Denotes a signed integer type that is at least as long as **char**, and whose range includes at least -32767 to +32767, inclusive. This type can also be written as **short int**.

```
// (during the last year)  
short hours_worked(short days_worked) {  
    return 8*days_worked;  
}
```

Section 21.11: bool

An integer type whose value can be either **true** or **false**.

```
bool is_even(int x) {  
    return x%2 == 0;  
}  
const bool b = is_even(47); // false
```

第22章：变量声明 关键字

第22.1节：decltype

版本 ≥ C++11

返回其操作数的类型，操作数本身不被求值。

- 如果操作数 e 是一个没有额外括号的名称，则 decltype(e) 是 e 的声明类型。

```
int x = 42;
std::vector<decltype(x)> v(100, x); // v 是一个 vector<int>
```

- 如果操作数 e 是一个类成员访问且没有额外的括号，那么 decltype(e) 是被访问成员的声明类型。

```
结构体 S {
    int x = 42;
};

const S s;
decltype(s.x) y; // y 的类型是 int, 尽管 s.x 是 const
```

- 在所有其他情况下，decltype(e) 同时返回表达式 e 的类型和值类别，规则如下：

- 如果 e 是类型为 T 的左值，则 decltype(e) 是 T&。
- 如果 e 是类型为 T 的将亡值 (xvalue)，则 decltype(e) 是 T&&。
- 如果 e 是类型为 T 的纯右值 (prvalue)，则 decltype(e) 是 T。

这也包括多余括号的情况。

```
int f() { return 42; }
int& g() { static int x = 42; return x; }
int x = 42;
decltype(f()) a = f(); // a 的类型是 int
decltype(g()) b = g(); // b 的类型是 int&
decltype((x)) c = x; // c 的类型是 int&, 因为 x 是一个左值
```

版本 ≥ C++14

特殊形式 decltype(auto) 根据变量的初始化值推导变量类型，或根据函数定义中 return 语句推导函数返回类型，使用的是 decltype 的类型推导规则，而非 auto 的规则。

```
const int x = 123;
auto y = x; // y 的类型是 int
decltype(auto) z = x; // z 的类型是 const int, 即 x 的声明类型
```

第22.2节：const

一种类型说明符；当应用于某个类型时，生成该类型的 const 限定版本。有关 const 的含义详见 const 关键字。

```
const int x = 123;
x = 456; // 错误
```

Chapter 22: Variable Declaration Keywords

Section 22.1: decltype

Version ≥ C++11

Yields the type of its operand, which is not evaluated.

- If the operand e is a name without any additional parentheses, then `decltype(e)` is the *declared type* of e.

```
int x = 42;
std::vector<decltype(x)> v(100, x); // v is a vector<int>
```

- If the operand e is a class member access without any additional parentheses, then `decltype(e)` is the *declared type* of the member accessed.

```
struct S {
    int x = 42;
};
const S s;
decltype(s.x) y; // y has type int, even though s.x is const
```

- In all other cases, `decltype(e)` yields both the type and the value category of the expression e, as follows:

- If e is an lvalue of type T, then `decltype(e)` is T&.
- If e is an xvalue of type T, then `decltype(e)` is T&&.
- If e is a prvalue of type T, then `decltype(e)` is T.

This includes the case with extraneous parentheses.

```
int f() { return 42; }
int& g() { static int x = 42; return x; }
int x = 42;
decltype(f()) a = f(); // a has type int
decltype(g()) b = g(); // b has type int&
decltype((x)) c = x; // c has type int&, since x is an lvalue
```

Version ≥ C++14

The special form `decltype(auto)` deduces the type of a variable from its initializer or the return type of a function from the `return` statements in its definition, using the type deduction rules of `decltype` rather than those of `auto`.

```
const int x = 123;
auto y = x; // y has type int
decltype(auto) z = x; // z has type const int, the declared type of x
```

Section 22.2: const

A type specifier; when applied to a type, produces the const-qualified version of the type. See `const` keyword for details on the meaning of `const`.

```
const int x = 123;
x = 456; // error
```

```
int& r = x; // 错误

struct S {
    void f();
    void g() const;
};

const S s;
s.f(); // 错误
s.g(); // OK
```

第22.3节 : volatile

一种类型限定符；当应用于某种类型时，会生成该类型的volatile限定版本。volatile限定在类型系统中起到与const限定相同的作用，但volatile并不阻止对象被修改；相反，它强制编译器将对该对象的所有访问视为副作用。

在下面的示例中，如果memory_mapped_port不是volatile，编译器可能会优化函数，使其只执行最后一次写操作；如果int的大小大于1，这将是不正确的。volatile限定强制编译器将所有sizeof(int)的写操作视为不同的副作用，因此必须全部执行（按顺序）。

```
extern volatile char memory_mapped_port;
void write_to_device(int x) {
    const char* p = reinterpret_cast<const char*>(&x);
    for (int i = 0; i < sizeof(int); i++) {
        memory_mapped_port = p[i];
    }
}
```

第22.4节 : signed

一个关键字，是某些整数类型名称的一部分。

- 当单独使用时，隐含int，因此signed、signed int和int是相同的类型。
- 当与char结合时，产生类型为 signed char，这是一种不同于char的类型，即使char也是有符号的。 signed char的取值范围至少包括-127到+127（含）。
- 当与short、long或long long结合使用时，这是多余的，因为这些类型本身已经是有符号的。
- signed不能与bool、wchar_t、char16_t或char32_t结合使用。

示例：

```
signed char 摄氏温度; std::cin >> 摄氏温度;
if (摄氏温度 < -35) { std::cout << "冷天，是吧？"; }
```

第22.5节：无符号 (unsigned)

一种请求整数类型无符号版本的类型说明符。

- 单独使用时，隐含int，因此unsigned与unsigned int是相同的类型。
- 类型unsigned char不同于类型char，即使char是无符号的。它可以存储至少255以内的整数。
- unsigned也可以与short、long或long long结合使用。它不能与bool、wchar_t、char16_t或char32_t结合使用。

```
int& r = x; // error

struct S {
    void f();
    void g() const;
};

const S s;
s.f(); // error
s.g(); // OK
```

Section 22.3: volatile

A type qualifier; when applied to a type, produces the volatile-qualified version of the type. Volatile qualification plays the same role as `const` qualification in the type system, but `volatile` does not prevent objects from being modified; instead, it forces the compiler to treat all accesses to such objects as side effects.

In the example below, if `memory_mapped_port` were not volatile, the compiler could optimize the function so that it performs only the final write, which would be incorrect if `sizeof(int)` is greater than 1. The `volatile` qualification forces it to treat all `sizeof(int)` writes as different side effects and hence perform all of them (in order).

```
extern volatile char memory_mapped_port;
void write_to_device(int x) {
    const char* p = reinterpret_cast<const char*>(&x);
    for (int i = 0; i < sizeof(int); i++) {
        memory_mapped_port = p[i];
    }
}
```

Section 22.4: signed

A keyword that is part of certain integer type names.

- When used alone, `int` is implied, so that `signed`, `signed int`, and `int` are the same type.
- When combined with `char`, yields the type `signed char`, which is a different type from `char`, even if `char` is also signed. `signed char` has a range that includes at least -127 to +127, inclusive.
- When combined with `short`, `long`, or `long long`, it is redundant, since those types are already signed.
- `signed` cannot be combined with `bool`, `wchar_t`, `char16_t`, or `char32_t`.

Example:

```
signed char celsius_temperature;
std::cin >> celsius_temperature;
if (celsius_temperature < -35) {
    std::cout << "cold day, eh?\n";
}
```

Section 22.5: unsigned

A type specifier that requests the unsigned version of an integer type.

- When used alone, `int` is implied, so `unsigned` is the same type as `unsigned int`.
- The type `unsigned char` is different from the type `char`, even if `char` is unsigned. It can hold integers up to at least 255.
- `unsigned` can also be combined with `short`, `long`, or `long long`. It cannot be combined with `bool`, `wchar_t`, `char16_t`, or `char32_t`.

示例：

```
char invert_case_table[256] = { ..., 'a', 'b', 'c', ..., 'A', 'B', 'C', ... };
char invert_case(char c) {
    unsigned char index = c;
    return invert_case_table[index];
// 注意：直接返回 invert_case_table[c] 在 char 为有符号类型的实现中会
产生错误
}
```

Example:

```
char invert_case_table[256] = { ..., 'a', 'b', 'c', ..., 'A', 'B', 'C', ... };
char invert_case(char c) {
    unsigned char index = c;
    return invert_case_table[index];
// note: returning invert_case_table[c] directly does the
// wrong thing on implementations where char is a signed type
}
```

第23章：关键字

关键字具有由 C++ 标准定义的固定含义，不能用作标识符。在包含标准库头文件的任何翻译单元中，使用预处理器重新定义关键字是非法的。然而，关键字在属性内部会失去其特殊含义。

第23.1节：asm

asm 关键字接受一个操作数，该操作数必须是字符串字面量。它具有实现定义的含义，但通常会传递给实现的汇编器，汇编器的输出会被合并到翻译单元中。

asm 语句是一个定义，而非表达式，因此它可以出现在块作用域或命名空间作用域（包括全局作用域）。但是，由于内联汇编无法受 C++ 语言规则约束，asm 不能出现在constexpr函数内部。

示例：

```
[[noreturn]] void halt_system() {
    asm("hlt");
}
```

第23.2节：不同的关键字

void C++

1. 当用作函数返回类型时，void 关键字指定该函数不返回值。

当用于函数的参数列表时，void 指定该函数不接受参数。当用于指针声明时，void 指定该指针是“通用”的。

2. 如果指针的类型是 void *，该指针可以指向任何未用 const 或 volatile 关键字声明的变量。除非将 void 指针转换为其他类型，否则不能对其进行解引用。void 指针可以转换为任何其他类型的数据指针。
3. void 指针可以指向函数，但在 C++ 中不能指向类成员。

```
void vobject; // C2182
void *pv; // 合法
int *pint; int i;
int main() {
    pv = &i;
    // 在 C 中可选转换，在 C++ 中必须转换
    pint = (int *)pv;
```

Volatile C++

1. 一种类型限定符，用于声明对象可以被硬件在程序中修改。

```
volatile 声明符 ;
```

Chapter 23: Keywords

Keywords have fixed meaning defined by the C++ standard and cannot be used as identifiers. It is illegal to redefine keywords using the preprocessor in any translation unit that includes a standard library header. However, keywords lose their special meaning inside attributes.

Section 23.1: asm

The `asm` keyword takes a single operand, which must be a string literal. It has an implementation-defined meaning, but is typically passed to the implementation's assembler, with the assembler's output being incorporated into the translation unit.

The `asm` statement is a *definition*, not an *expression*, so it may appear either at block scope or namespace scope (including global scope). However, since inline assembly cannot be constrained by the rules of the C++ language, `asm` may not appear inside a `constexpr` function.

Example:

```
[[noreturn]] void halt_system() {
    asm("hlt");
}
```

Section 23.2: Different keywords

void C++

1. When used as a function return type, the void keyword specifies that the function does not return a value. When used for a function's parameter list, void specifies that the function takes no parameters. When used in the declaration of a pointer, void specifies that the pointer is "universal."
2. If a pointer's type is void *, the pointer can point to any variable that is not declared with the const or volatile keyword. A void pointer cannot be dereferenced unless it is cast to another type. A void pointer can be converted into any other type of data pointer.
3. A void pointer can point to a function, but not to a class member in C++.

```
void vobject; // C2182
void *pv; // okay
int *pint; int i;
int main() {
    pv = &i;
    // Cast optional in C required in C++
    pint = (int *)pv;
```

Volatile C++

1. A type qualifier that you can use to declare that an object can be modified in the program by the hardware.

```
volatile declarator ;
```

- virtual 关键字用于声明虚函数或虚基类。

```
virtual [类型-限定符] 成员-函数-声明符
virtual [访问-限定符] 基类-名称
```

参数

- 类型限定符 指定虚成员函数的返回类型。
- 成员函数声明符 声明成员函数。
- 访问限定符 定义对基类的访问级别，public（公有）、protected（保护）或private（私有）。可出现在前面或在 virtual 关键字之后。
- base-class-name 标识先前声明的类类型

this 指针

- this 指针是一个只能在类、结构体或联合体的非静态成员函数内访问的指针。它指向调用该成员函数的对象。静态成员函数没有 this 指针。

```
this->成员-标识符
```

对象的 this 指针不是对象本身的一部分；它不会反映在对该对象使用 sizeof 运算符的结果中。相反，当为对象调用非静态成员函数时，编译器会将该对象的地址作为隐藏参数传递给函数。例如，以下函数调用：

```
myDate.setMonth( 3 );
```

可以这样理解：

```
setMonth( &myDate, 3 );
```

对象的地址可以在成员函数内部通过 this 指针获得。this 的大多数用法是隐式的。

在引用类的成员时，显式使用 this 是合法的，尽管没有必要。例如：

```
void Date::setMonth( int mn )
{
month = mn;           // 这三条语句
    this->month = mn;   // 是等价的
    (*this).month = mn;
}
```

表达式 *this 通常用于从成员函数返回当前对象：return *this; this 指针也用于防止自我引用：

```
if (&Object != this) {
// 在自我引用的情况下不执行
```

try、throw 和 catch 语句 (C++)

- The virtual keyword declares a virtual function or a virtual base class.

```
virtual [type-specifiers] member-function-declarator
virtual [access-specifier] base-class-name
```

Parameters

- type-specifiers** Specifies the return type of the virtual member function.
- member-function-declarator** Declares a member function.
- access-specifier** Defines the level of access to the base class, public, protected or private. Can appear before or after the virtual keyword.
- base-class-name** Identifies a previously declared class type

this pointer

- The this pointer is a pointer accessible only within the nonstatic member functions of a class, struct, or union type. It points to the object for which the member function is called. Static member functions do not have a this pointer.

```
this->member-identifier
```

An object's this pointer is not part of the object itself; it is not reflected in the result of a sizeof statement on the object. Instead, when a nonstatic member function is called for an object, the address of the object is passed by the compiler as a hidden argument to the function. For example, the following function call:

```
myDate.setMonth( 3 );
```

can be interpreted this way:

```
setMonth( &myDate, 3 );
```

The object's address is available from within the member function as the this pointer. Most uses of this are implicit. It is legal, though unnecessary, to explicitly use this when referring to members of the class. For example:

```
void Date::setMonth( int mn )
{
month = mn;           // These three statements
    this->month = mn;   // are equivalent
    (*this).month = mn;
}
```

The expression *this is commonly used to return the current object from a member function: return *this; The this pointer is also used to guard against self-reference:

```
if (&Object != this) {
// do not execute in cases of self-reference
```

try, throw, and catch Statements (C++)

1. 在 C++ 中实现异常处理，使用 try、throw 和 catch 表达式。

2. 首先，使用 try 块包围一个或多个可能抛出异常的语句。

3. throw 表达式表示在 try 块中发生了异常情况——通常是错误——你

可以使用任何类型的对象作为 throw 表达式的操作数。通常，这个对象用于传达有关错误的信息。在大多数情况下，我们建议使用 std::exception 类或标准库中定义的派生类之一。如果这些都不合适，建议您从 std::exception 派生自己的异常类。

4. 为了处理可能抛出的异常，在 try 块后立即实现一个或多个 catch 块。每个 catch 块指定它可以处理的异常类型。

```
MyData md;
try {
    // 可能抛出异常的代码
    md = GetNetworkResource();
}
catch (const networkIOException& e) {
    // 当 try 块中抛出 networkIOException 类型的异常时执行的代码 // ...
}
```

// 记录异常对象中的错误信息
cerr << e.what();

```
}
catch (const myDataFormatException& e) {
    // 处理另一种异常类型的代码
    // ...
    cerr << e.what();
}
```

// 以下语法展示了一个 throw 表达式

```
MyData GetNetworkResource()
{
    // ...
    if (IOSuccess == false)
        throw networkIOException("Unable to connect");
    // ...
    if (readError)
        throw myDataFormatException("Format error");
    // ...
}
```

try 子句之后的代码是受保护的代码段。throw 表达式抛出——也就是引发——一个异常。catch 子句之后的代码块是异常处理程序。这个处理程序会捕获抛出的异常，前提是 throw 和 catch 表达式中的类型兼容。

```
try {
    throw CSomeOtherException();
}
catch(...) {
    // 捕获所有异常——危险！！！
    // 对异常做出响应（可能只是部分响应），然后
    // 重新抛出异常以将其传递给其他处理程序
    // ...
    throw;
}
```

友元 (C++)

1. To implement exception handling in C++, you use try, throw, and catch expressions.

2. First, use a try block to enclose one or more statements that might throw an exception.

3. A throw expression signals that an exceptional condition—often, an error—has occurred in a try block. You can use an object of any type as the operand of a throw expression. Typically, this object is used to communicate information about the error. In most cases, we recommend that you use the std::exception class or one of the derived classes that are defined in the standard library. If one of those is not appropriate, we recommend that you derive your own exception class from std::exception.

4. To handle exceptions that may be thrown, implement one or more catch blocks immediately following a try block. Each catch block specifies the type of exception it can handle.

```
MyData md;
try {
    // Code that could throw an exception
    md = GetNetworkResource();
}
catch (const networkIOException& e) {
    // Code that executes when an exception of type
    // networkIOException is thrown in the try block
    // ...
    // Log error message in the exception object
    cerr << e.what();
}
catch (const myDataFormatException& e) {
    // Code that handles another exception type
    // ...
    cerr << e.what();
}

// The following syntax shows a throw expression
MyData GetNetworkResource()
{
    // ...
    if (IOSuccess == false)
        throw networkIOException("Unable to connect");
    // ...
    if (readError)
        throw myDataFormatException("Format error");
    // ...
}
```

The code after the try clause is the guarded section of code. The throw expression throws—that is, raises—an exception. The code block after the catch clause is the exception handler. This is the handler that catches the exception that's thrown if the types in the throw and catch expressions are compatible.

```
try {
    throw CSomeOtherException();
}
catch(...) {
    // Catch all exceptions – dangerous!!!
    // Respond (perhaps only partially) to the exception, then
    // re-throw to pass the exception to some other handler
    // ...
    throw;
}
```

friend (C++)

1. 在某些情况下，授予非类成员函数或另一个独立类中的所有成员访问权限会更方便。只有类的实现者可以声明其友元。函数或类不能声明自己为任何类的友元。在类定义中，使用friend关键字和非成员函数或其他类的名称，授予其访问该类的私有和受保护成员的权限。在模板定义中，类型参数也可以被声明为友元。

2. 如果声明了一个之前未声明的友元函数，该函数将被导出到包含的非类作用域中。

```
class friend F
friend F;
class ForwardDeclared;// 已知类名。
class HasFriends
{
    friend int ForwardDeclared::IsAFriend(); // 预期出现C2039错误
};
```

友元函数

- 友元函数是指不是类成员的函数，但可以访问该类的私有和受保护成员。友元函数不被视为类成员；它们是被赋予特殊访问权限的普通外部函数。
- 友元不在类的作用域内，也不会通过成员选择运算符（. 和 ->）调用除非它们是另一个类的成员。
- 友元函数由授予访问权限的类声明。友元声明可以放在类声明的任何位置，不受访问控制关键字的影响。

```
#include <iostream>

using namespace std;
class Point
{
    friend void ChangePrivate( Point & );
public:
Point( void ) : m_i(0) {}
    void PrintPrivate( void ) {cout << m_i << endl; }

private:
int m_i;
};

void ChangePrivate ( Point &i ) { i.m_i++; }

int main()
{
Point sPoint;

    ChangePrivate(sPoint);
    sPoint.PrintPrivate();
    // 输出: 0
        1
}
```

1. In some circumstances, it is more convenient to grant member-level access to functions that are not members of a class or to all members in a separate class. Only the class implementer can declare who its friends are. A function or class cannot declare itself as a friend of any class. In a class definition, use the friend keyword and the name of a non-member function or other class to grant it access to the private and protected members of your class. In a template definition, a type parameter can be declared as a friend.

2. If you declare a friend function that was not previously declared, that function is exported to the enclosing nonclass scope.

```
class friend F
friend F;
class ForwardDeclared;// Class name is known.
class HasFriends
{
    friend int ForwardDeclared::IsAFriend(); // C2039 error expected
};
```

friend functions

- A friend function is a function that is not a member of a class but has access to the class's private and protected members. Friend functions are not considered class members; they are normal external functions that are given special access privileges.
- Friends are not in the class's scope, and they are not called using the member-selection operators (. and ->) unless they are members of another class.
- A friend function is declared by the class that is granting access. The friend declaration can be placed anywhere in the class declaration. It is not affected by the access control keywords.

```
#include <iostream>

using namespace std;
class Point
{
    friend void ChangePrivate( Point & );
public:
Point( void ) : m_i(0) {}
    void PrintPrivate( void ) {cout << m_i << endl; }

private:
int m_i;
};

void ChangePrivate ( Point &i ) { i.m_i++; }

int main()
{
    Point sPoint;
    sPoint.PrintPrivate();
    ChangePrivate(sPoint);
    sPoint.PrintPrivate();
    // Output: 0
        1
}
```

```

类 B;

类 A {
    公共:
        整数 Func1( B& b );

    私有:
        整数 Func2( B& b );
};

类 B {
    私有:
        整数 _b;

    // A::Func1 是类 B 的友元函数
    // 因此 A::Func1 可以访问 B 的所有成员
    友元 整数 A::Func1( B& );
};

整数 A::Func1( B& b ) { 返回 b._b; } // 正确
整数 A::Func2( B& b ) { 返回 b._b; } // C2248

```

第23.3节 : typename

- 当后跟限定名时，typename 指定它是一个类型名。这通常在模板，特别是当嵌套名称说明符是除当前实例化之外的依赖类型时。在此示例中，std::decay<T>依赖于模板参数T，因此为了命名嵌套类型type，我们需要在整个限定名称前加上typename。更多详情请参见“我为什么以及何时必须使用‘template’和‘typename’关键字？”

```

template <class T>
auto decay_copy(T& r) -> typename std::decay<T>::type;

```

- 在模板声明中引入类型参数。在此上下文中，它与class可以互换使用。

```

template <typename T>
const T& min(const T& x, const T& y) {
    return b < a ? b : a;
}

```

版本 ≥ C++17

- typename也可以用于声明模板模板参数，置于参数名称之前，类似于class参数，就像class一样。

```

template <template <class T> typename U>
void f() {
    U<int>::do_it();
    U<double>::do_it();
}

```

Class members as friends

```

class B;

class A {
    public:
        int Func1( B& b );

    private:
        int Func2( B& b );
};

class B {
    private:
        int _b;

    // A::Func1 is a friend function to class B
    // so A::Func1 has access to all members of B
    friend int A::Func1( B );
};

int A::Func1( B& b ) { return b._b; } // OK
int A::Func2( B& b ) { return b._b; } // C2248

```

Section 23.3: typename

- When followed by a qualified name, typename specifies that it is the name of a type. This is often required in templates, in particular, when the nested name specifier is a dependent type other than the current instantiation. In this example, std::decay<T> depends on the template parameter T, so in order to name the nested type type, we need to prefix the entire qualified name with typename. For more details, see [Where and why do I have to put the "template" and "typename" keywords?](#)

```

template <class T>
auto decay_copy(T& r) -> typename std::decay<T>::type;

```

- Introduces a type parameter in the declaration of a template. In this context, it is interchangeable with class.

```

template <typename T>
const T& min(const T& x, const T& y) {
    return b < a ? b : a;
}

```

Version ≥ C++17

- typename can also be used when declaring a template template parameter, preceding the name of the parameter, just like class.

```

template <template <class T> typename U>
void f() {
    U<int>::do_it();
    U<double>::do_it();
}

```

第23.4节 : explicit (显式)

1. 当应用于单参数构造函数时，防止该构造函数被用于执行隐式转换。

```
class MyVector {
public:
    explicit MyVector(uint64_t size);
};

MyVector v1(100); // ok
uint64_t len1 = 100;
MyVector v2{len1}; // ok, len1 是 uint64_t
int len2 = 100;
MyVector v3{len2}; // 格式错误, int到uint64_t的隐式转换
```

自从C++11引入初始化列表以来，在C++11及更高版本中，`explicit`可以应用于具有任意数量参数的构造函数，其含义与单参数情况相同。

```
结构体 S {
    explicit S(int x, int y);
};

S f() {
    return {12, 34}; // ill-formed
    return S{12, 34}; // ok
}
```

版本 ≥ C++11

2. 当应用于转换函数时，防止该转换函数被用于执行隐式转换。

```
class C {
    const int x;
public:
    C(int x) : x(x) {}
    explicit operator int() { return x; }
};

C c(42);
int x = c; // ill-formed
int y = static_cast<int>(c); // ok; explicit conversion
```

第23.5节 : sizeof

一元运算符，返回其操作数的字节大小，操作数可以是表达式或类型。如果操作数是表达式，则不对其求值。大小是类型为`std::size_t`的常量表达式。

如果操作数是类型，则必须加括号。

- 对函数类型使用`sizeof`是非法的。
- 对不完整类型（包括`void`）使用`sizeof`是非法的。
- 如果对引用类型`T&`或`T&&`使用`sizeof`，则等同于`sizeof(T)`。
- 当`sizeof`应用于类类型时，它返回该类型完整对象的字节数，包括中间或末尾的任何填充字节。因此，`sizeof`表达式的值永远不会是0。详情见对象类型的布局。

Section 23.4: explicit

1. When applied to a single-argument constructor, prevents that constructor from being used to perform implicit conversions.

```
class MyVector {
public:
    explicit MyVector(uint64_t size);
};

MyVector v1(100); // ok
uint64_t len1 = 100;
MyVector v2{len1}; // ok, len1 is uint64_t
int len2 = 100;
MyVector v3{len2}; // ill-formed, implicit conversion from int to uint64_t
```

Since C++11 introduced initializer lists, in C++11 and later, `explicit` can be applied to a constructor with any number of arguments, with the same meaning as in the single-argument case.

```
struct S {
    explicit S(int x, int y);
};

S f() {
    return {12, 34}; // ill-formed
    return S{12, 34}; // ok
}
```

Version ≥ C++11

2. When applied to a conversion function, prevents that conversion function from being used to perform implicit conversions.

```
class C {
    const int x;
public:
    C(int x) : x(x) {}
    explicit operator int() { return x; }
};

C c(42);
int x = c; // ill-formed
int y = static_cast<int>(c); // ok; explicit conversion
```

Section 23.5: sizeof

A unary operator that yields the size in bytes of its operand, which may be either an expression or a type. If the operand is an expression, it is not evaluated. The size is a constant expression of type `std::size_t`.

If the operand is a type, it must be parenthesized.

- It is illegal to apply `sizeof` to a function type.
- It is illegal to apply `sizeof` to an incomplete type, including `void`.
- If `sizeof` is applied to a reference type `T&` or `T&&`, it is equivalent to `sizeof(T)`.
- When `sizeof` is applied to a class type, it yields the number of bytes in a complete object of that type, including any padding bytes in the middle or at the end. Therefore, a `sizeof` expression can never have a value of 0. See layout of object types for more details.

- 类型char、signed char和unsigned char的大小为1。相反，字节被定义为存储一个char对象所需的内存量。这并不一定意味着8位，因为某些系统的char对象长度超过8位。

如果expr是一个表达式，`sizeof(expr)`等价于`sizeof(T)`，其中T是expr的类型。

```
int a[100];
std::cout << "数组 a 的字节数是: " << sizeof a;
memset(a, 0, sizeof a); // 将数组清零
版本 ≥ C++11
```

`sizeof...`操作符返回参数包中的元素数量。

```
template <class... T>
void f(T&... args) {
    std::cout << "函数 f 被调用，参数数量为 " << sizeof...(T) << " 个";}
```

第23.6节：noexcept

版本 ≥ C++11

1. 一元操作符，用于判断其操作数的求值是否可能传播异常。注意，调用函数的函数体不会被检查，因此`noexcept`可能产生假阴性。操作数不会被求值。

```
#include <iostream>
#include <stdexcept>
void foo() { throw std::runtime_error("oops"); }
void bar() {}
struct S {};
int main() {
    std::cout << noexcept(foo()) << "; // 输出 0    std::cout << noexcept(bar()) << "; // 输出 0    std::cout << noexcept(1 + 1) << ";
    // 输出 1    std::cout << noexcept(S()) << "; // 输出 1}
```

在此示例中，尽管`bar()`永远不会抛出异常，`noexcept(bar())`仍然为假，因为没有明确指定`bar()`不能传播异常这一事实。

2. 声明函数时，指定该函数是否可以传播异常。单独使用时，声明该函数不能传播异常。带有括号参数时，根据参数的真假值声明函数是否可以传播异常。

```
void f1() { throw std::runtime_error("oops"); }
void f2() noexcept(false) { throw std::runtime_error("oops"); }
void f3() {}
void f4() noexcept {}
void f5() noexcept(true) {}
void f6() noexcept {
    try {
        f1();
    } catch (const std::runtime_error&) {}
}
```

- The `char`, `signed char`, and `unsigned char` types have a size of 1. Conversely, a byte is defined to be the amount of memory required to store a `char` object. It does not necessarily mean 8 bits, as some systems have `char` objects longer than 8 bits.

If expr is an expression, `sizeof(expr)` is equivalent to `sizeof(T)` where T is the type of expr.

```
int a[100];
std::cout << "The number of bytes in `a` is: " << sizeof a;
memset(a, 0, sizeof a); // zeroes out the array
Version ≥ C++11
```

The `sizeof...` operator yields the number of elements in a parameter pack.

```
template <class... T>
void f(T&... args) {
    std::cout << "f was called with " << sizeof...(T) << " arguments\n";
}
```

Section 23.6: noexcept

Version ≥ C++11

1. A unary operator that determines whether the evaluation of its operand can propagate an exception. Note that the bodies of called functions are not examined, so `noexcept` can yield false negatives. The operand is not evaluated.

```
#include <iostream>
#include <stdexcept>
void foo() { throw std::runtime_error("oops"); }
void bar() {}
struct S {};
int main() {
    std::cout << noexcept(foo()) << '\n'; // prints 0
    std::cout << noexcept(bar()) << '\n'; // prints 0
    std::cout << noexcept(1 + 1) << '\n'; // prints 1
    std::cout << noexcept(S()) << '\n'; // prints 1
}
```

In this example, even though `bar()` can never throw an exception, `noexcept(bar())` is still false because the fact that `bar()` cannot propagate an exception has not been explicitly specified.

2. When declaring a function, specifies whether or not the function can propagate an exception. Alone, it declares that the function cannot propagate an exception. With a parenthesized argument, it declares that the function can or cannot propagate an exception depending on the truth value of the argument.

```
void f1() { throw std::runtime_error("oops"); }
void f2() noexcept(false) { throw std::runtime_error("oops"); }
void f3() {}
void f4() noexcept {}
void f5() noexcept(true) {}
void f6() noexcept {
    try {
        f1();
    } catch (const std::runtime_error&) {}
}
```

在此示例中，我们声明了f4、f5和f6不能传播异常。（尽管在执行f6期间可能会抛出异常，但该异常被捕获，不允许传播出函数。）我们声明了f2可能传播异常。当省略noexcept说明符时，等同于noexcept(false)，因此我们隐式声明了f1和f3可能传播异常，尽管实际上在执行f3期间不会抛出异常。

版本 ≥ C++17

函数是否为noexcept是函数类型的一部分：也就是说，在上面的例子中，f1、f2和f3的类型与f4、f5和f6不同。因此，noexcept在函数指针、模板参数等方面也很重要。

```
void g1() {}  
void g2() noexcept {}  
void (*p1)() noexcept = &g1; // ill-formed, since g1 is not noexcept  
void (*p2)() noexcept = &g2; // ok; types match  
void (*p3)() = &g1; // ok; types match  
void (*p4)() = &g2; // ok; implicit conversion
```

In this example, we have declared that f4, f5, and f6 cannot propagate exceptions. (Although an exception can be thrown during execution of f6, it is caught and not allowed to propagate out of the function.) We have declared that f2 may propagate an exception. When the `noexcept` specifier is omitted, it is equivalent to `noexcept(false)`, so we have implicitly declared that f1 and f3 may propagate exceptions, even though exceptions cannot actually be thrown during the execution of f3.

Version ≥ C++17

Whether or not a function is `noexcept` is part of the function's type: that is, in the example above, f1, f2, and f3 have different types from f4, f5, and f6. Therefore, `noexcept` is also significant in function pointers, template arguments, and so on.

```
void g1() {}  
void g2() noexcept {}  
void (*p1)() noexcept = &g1; // ill-formed, since g1 is not noexcept  
void (*p2)() noexcept = &g2; // ok; types match  
void (*p3)() = &g1; // ok; types match  
void (*p4)() = &g2; // ok; implicit conversion
```

第24章：从函数返回多个值

在许多情况下，从函数返回多个值是很有用的：例如，如果你想输入一个项目并返回价格和库存数量，这个功能就很有用。在C++中有很多方法可以做到这一点，大多数都涉及STL。然而，如果你出于某种原因想避免使用STL，仍然有几种方法可以实现，包括结构体/类和数组。

第24.1节：使用 std::tuple

版本 ≥ C++11

类型std::tuple可以将任意数量的值（可能包括不同类型的值）打包成一个单一的返回对象：

```
std::tuple<int, int, int, int> foo(int a, int b) { // 或者使用 auto (C++14)
    return std::make_tuple(a + b, a - b, a * b, a / b);
}
```

在 C++17 中，可以使用花括号初始化列表：

版本 ≥ C++17

```
std::tuple<int, int, int, int> foo(int a, int b) {
    return {a + b, a - b, a * b, a / b};
}
```

从返回的 tuple 中获取值可能比较繁琐，需要使用 std::get 模板函数：

```
auto mrvs = foo(5, 12);
auto add = std::get<0>(mrvs);
auto sub = std::get<1>(mrvs);
auto mul = std::get<2>(mrvs);
auto div = std::get<3>(mrvs);
```

如果类型可以在函数返回之前声明，则可以使用 std::tie 将 tuple 解包到已有变量中：

```
int add, sub, mul, div;
std::tie(add, sub, mul, div) = foo(5, 12);
```

如果不返回值中的某个值，可以使用 std::ignore：

```
std::tie(add, sub, std::ignore, div) = foo(5, 12);
```

版本 ≥ C++17

可以使用结构化绑定来避免使用 std::tie：

```
auto [add, sub, mul, div] = foo(5, 12);
```

如果您想返回一个左值引用的元组，而不是值的元组，请使用std::tie代替 std::make_tuple。

```
std::tuple<int&, int&> minmax( int& a, int& b ) {
```

Chapter 24: Returning several values from a function

There are many situations where it is useful to return several values from a function: for example, if you want to input an item and return the price and number in stock, this functionality could be useful. There are many ways to do this in C++, and most involve the STL. However, if you wish to avoid the STL for some reason, there are still several ways to do this, including structs/classes and arrays.

Section 24.1: Using std::tuple

Version ≥ C++11

The type `std::tuple` can bundle any number of values, potentially including values of different types, into a single return object:

```
std::tuple<int, int, int, int> foo(int a, int b) { // or auto (C++14)
    return std::make_tuple(a + b, a - b, a * b, a / b);
}
```

In C++17, a braced initializer list can be used:

```
Version ≥ C++17
std::tuple<int, int, int, int> foo(int a, int b) {
    return {a + b, a - b, a * b, a / b};
}
```

Retrieving values from the returned tuple can be cumbersome, requiring the use of the `std::get` template function:

```
auto mrvs = foo(5, 12);
auto add = std::get<0>(mrvs);
auto sub = std::get<1>(mrvs);
auto mul = std::get<2>(mrvs);
auto div = std::get<3>(mrvs);
```

If the types can be declared before the function returns, then `std::tie` can be employed to unpack a tuple into existing variables:

```
int add, sub, mul, div;
std::tie(add, sub, mul, div) = foo(5, 12);
```

If one of the returned values is not needed, `std::ignore` can be used:

```
std::tie(add, sub, std::ignore, div) = foo(5, 12);
```

Version ≥ C++17

Structured bindings can be used to avoid `std::tie`:

```
auto [add, sub, mul, div] = foo(5, 12);
```

If you want to return a tuple of lvalue references instead of a tuple of values, use `std::tie` in place of `std::make_tuple`.

```
std::tuple<int&, int&> minmax( int& a, int& b ) {
```

```

if (b < a)
    return std::tie(b, a);
else
    return std::tie(a, b);
}

```

这允许

```

void increase_least(int& a, int& b) {
    std::get<0>(minmax(a, b))++;
}

```

在某些罕见情况下，你会使用 `std::forward_as_tuple` 代替 `std::tie`；使用时请小心，因为临时对象可能不会持续足够长的时间被使用。

第24.2节：结构化绑定

版本 ≥ C++17

C++17引入了结构化绑定，这使得处理多返回类型更加容易，因为你不需要依赖 `std::tie()` 或手动拆包元组：

```

std::map<std::string, int> m;

// 将元素插入映射并检查插入是否成功
auto [iterator, success] = m.insert({"Hello", 42});

if (success) {
    // 你的代码写在这里
}

// 遍历所有元素，无需使用晦涩的 'first' 和 'second' 名称
for (auto const& [key, value] : m) {
    std::cout << "键 " << key << " 的值是 " << value << ";"
}

```

结构化绑定默认可用于 `std::pair`、`std::tuple` 以及任何非静态数据成员全部为公共直接成员或无歧义基类成员的类型：

```

struct A { int x; };
struct B : A { int y; };
B foo();

// 使用结构化绑定
const auto [x, y] = foo();

// 不使用结构化绑定的等效代码
const auto result = foo();
auto& x = result.x;
auto& y = result.y;

```

如果你使你的类型“类似元组”，它也会自动适用于你的类型。类似元组是指具有适当的 `tuple_size`、`tuple_element` 和 `get` 的类型：

```

namespace my_ns {
    struct my_type {
        int x;
    }
}

```

```

if (b < a)
    return std::tie(b, a);
else
    return std::tie(a, b);
}

```

which permits

```

void increase_least(int& a, int& b) {
    std::get<0>(minmax(a, b))++;
}

```

In some rare cases you'll use `std::forward_as_tuple` instead of `std::tie`; be careful if you do so, as temporaries may not last long enough to be consumed.

Section 24.2: Structured Bindings

Version ≥ C++17

C++17 introduces structured bindings, which makes it even easier to deal with multiple return types, as you do not need to rely upon `std::tie()` or do any manual tuple unpacking:

```

std::map<std::string, int> m;

// insert an element into the map and check if insertion succeeded
auto [iterator, success] = m.insert({"Hello", 42});

if (success) {
    // your code goes here
}

// iterate over all elements without having to use the cryptic 'first' and 'second' names
for (auto const& [key, value] : m) {
    std::cout << "The value for " << key << " is " << value << '\n';
}

```

Structured bindings can be used by default with `std::pair`, `std::tuple`, and any type whose non-static data members are all either public direct members or members of an unambiguous base class:

```

struct A { int x; };
struct B : A { int y; };
B foo();

// with structured bindings
const auto [x, y] = foo();

// equivalent code without structured bindings
const auto result = foo();
auto& x = result.x;
auto& y = result.y;

```

If you make your type "tuple-like" it will also automatically work with your type. A tuple-like is a type with appropriate `tuple_size`, `tuple_element` and `get` written:

```

namespace my_ns {
    struct my_type {
        int x;
    }
}

```

```

        double d;
std::string s;
};

struct my_type_view {
    my_type* ptr;
};

}

namespace std {
    template<>
    struct tuple_size<my_ns::my_type_view> : std::integral_constant<std::size_t, 3>
    {};

    template<> struct tuple_element<my_ns::my_type_view, 0>{ using type = int; };
    template<> struct tuple_element<my_ns::my_type_view, 1>{ using type = double; };
    template<> struct tuple_element<my_ns::my_type_view, 2>{ using type = std::string; };
}

namespace my_ns {
    template<std::size_t I>
    decltype(auto) get(my_type_view const& v) {
        if constexpr (I == 0)
            return v.ptr->x;
        else if constexpr (I == 1)
            return v.ptr->d;
        else if constexpr (I == 2)
            return v.ptr->s;
        static_assert(I < 3, "Only 3 elements");
    }
}

```

现在这可以工作了：

```

my_ns::my_type t{1, 3.14, "hello world"};

my_ns::my_type_view foo() {
    return {&t};
}

int main() {
    auto[x, d, s] = foo();
    std::cout << x << ',' << d << ',' << s << ";"
}

```

第24.3节：使用结构体

结构体 (struct) 可以用来捆绑多个返回值：

```

版本 ≥ C++11

结构体 foo_return_type {
    整数 add;
    int sub;
    int mul;
    int div;
};

foo_return_type foo(int a, int b) {
    return {a + b, a - b, a * b, a / b};
}

```

```

        double d;
        std::string s;
    };
    struct my_type_view {
        my_type* ptr;
    };
}

namespace std {
    template<>
    struct tuple_size<my_ns::my_type_view> : std::integral_constant<std::size_t, 3>
    {};

    template<> struct tuple_element<my_ns::my_type_view, 0>{ using type = int; };
    template<> struct tuple_element<my_ns::my_type_view, 1>{ using type = double; };
    template<> struct tuple_element<my_ns::my_type_view, 2>{ using type = std::string; };
}

namespace my_ns {
    template<std::size_t I>
    decltype(auto) get(my_type_view const& v) {
        if constexpr (I == 0)
            return v.ptr->x;
        else if constexpr (I == 1)
            return v.ptr->d;
        else if constexpr (I == 2)
            return v.ptr->s;
        static_assert(I < 3, "Only 3 elements");
    }
}

```

now this works:

```

my_ns::my_type t{1, 3.14, "hello world"};

my_ns::my_type_view foo() {
    return {&t};
}

int main() {
    auto[x, d, s] = foo();
    std::cout << x << ',' << d << ',' << s << '\n';
}

```

Section 24.3: Using struct

A struct can be used to bundle multiple return values:

```

Version ≥ C++11

struct foo_return_type {
    int add;
    int sub;
    int mul;
    int div;
};

foo_return_type foo(int a, int b) {
    return {a + b, a - b, a * b, a / b};
}

```

```
auto calc = foo(5, 12);
```

版本 < C++11

可以使用构造函数来简化返回值的构造，而不是赋值给各个字段：

```
结构体 foo_return_type {
    整数 add;
    int sub;
    int mul;
    int div;
foo_return_type(int add, int sub, int mul, int div)
    : add(add), sub(sub), mul(mul), div(div) {}

foo_return_type foo(int a, int b) {
    return foo_return_type(a + b, a - b, a * b, a / b);
}

foo_return_type calc = foo(5, 12);
```

函数 foo() 返回的各个结果可以通过访问结构体 calc 的成员变量来获取：

```
std::cout << calc.add << ' ' << calc.sub << ' ' << calc.mul << ' ' << calc.div << ";
```

输出：

```
17 -7 60 0
```

注意：使用 struct 时，返回值会被组合在一个对象中，并且可以通过有意义的名称访问。这也有助于减少在返回值作用域中创建的多余变量数量。

版本 ≥ C++17

为了展开从函数返回的 struct，可以使用结构化绑定。这使得输出参数与输入参数处于同等地位：

```
int a=5, b=12;
auto[add, sub, mul, div] = foo(a, b); std::cout
    << add << ' ' << sub << ' ' << mul << ' ' << div << ";
```

这段代码的输出与上面完全相同。函数仍然使用 struct 来返回值。这允许你单独处理各个字段。

第24.4节：使用输出参数

参数可用于返回一个或多个值；这些参数必须是非const指针或引用。

引用：

```
void calculate(int a, int b, int& c, int& d, int& e, int& f) {
    c = a + b;
    d = a - b;
```

```
auto calc = foo(5, 12);
```

Version < C++11

Instead of assignment to individual fields, a constructor can be used to simplify the constructing of returned values:

```
struct foo_return_type {
    int add;
    int sub;
    int mul;
    int div;
foo_return_type(int add, int sub, int mul, int div)
    : add(add), sub(sub), mul(mul), div(div) {}

foo_return_type foo(int a, int b) {
    return foo_return_type(a + b, a - b, a * b, a / b);
}

foo_return_type calc = foo(5, 12);
```

The individual results returned by the function foo() can be retrieved by accessing the member variables of the struct calc:

```
std::cout << calc.add << ' ' << calc.sub << ' ' << calc.mul << ' ' << calc.div << '\n' ;
```

Output:

```
17 -7 60 0
```

Note: When using a struct, the returned values are grouped together in a single object and accessible using meaningful names. This also helps to reduce the number of extraneous variables created in the scope of the returned values.

Version ≥ C++17

In order to unpack a struct returned from a function, structured bindings can be used. This places the out-parameters on an even footing with the in-parameters:

```
int a=5, b=12;
auto[add, sub, mul, div] = foo(a, b);
std::cout << add << ' ' << sub << ' ' << mul << ' ' << div << '\n' ;
```

The output of this code is identical to that above. The struct is still used to return the values from the function. This permits you do deal with the fields individually.

Section 24.4: Using Output Parameters

Parameters can be used for returning one or more values; those parameters are required to be non-const pointers or references.

References:

```
void calculate(int a, int b, int& c, int& d, int& e, int& f) {
    c = a + b;
    d = a - b;
```

```
e = a * b;  
f = a / b;  
}
```

指针：

```
void calculate(int a, int b, int* c, int* d, int* e, int* f) {  
    *c = a + b;  
    *d = a - b;  
    *e = a * b;  
    *f = a / b;  
}
```

一些库或框架使用空的 'OUT' #define 来明确指出函数签名中哪些参数是输出参数。这不会影响功能，且会被编译器优化掉，但使函数签名更清晰；

```
#define OUT  
  
void calculate(int a, int b, OUT int& c) {  
    c = a + b;  
}
```

第24.5节：使用函数对象消费者

我们可以提供一个消费者，该消费者将被调用并传入多个相关值：

```
版本 ≥ C++11  
  
template <class F>  
void foo(int a, int b, F consumer) {  
    consumer(a + b, a - b, a * b, a / b);  
}  
  
// 使用很简单.....也可以忽略某些结果  
foo(5, 12, [](int sum, int , int , int ){  
    std::cout << "sum is " << sum << ";\");
```

这被称为“继续传递风格” ([continuation passing style](#))。

你可以通过以下方式将返回元组的函数适配为继续传递风格的函数：

```
版本 ≥ C++17  
  
template<class Tuple>  
struct continuation {  
    Tuple t;  
    模板<类 F>  
    decltype(auto) operator->*(F&& f)&&{  
        return std::apply( std::forward<F>(f), std::move(t) );  
    }  
};  
std::tuple<int,int,int,int> foo(int a, int b);continuation(foo  
  
(5,12))->*[ ](int sum, auto&&...){ std::cout << "sum is " << s  
um << ";\");
```

更复杂的版本可以用 C++14 或 C++11 编写。

```
e = a * b;  
f = a / b;  
}
```

Pointers:

```
void calculate(int a, int b, int* c, int* d, int* e, int* f) {  
    *c = a + b;  
    *d = a - b;  
    *e = a * b;  
    *f = a / b;  
}
```

Some libraries or frameworks use an empty 'OUT' #define to make it abundantly obvious which parameters are output parameters in the function signature. This has no functional impact, and will be compiled out, but makes the function signature a bit clearer;

```
#define OUT  
  
void calculate(int a, int b, OUT int& c) {  
    c = a + b;  
}
```

Section 24.5: Using a Function Object Consumer

We can provide a consumer that will be called with the multiple relevant values:

```
Version ≥ C++11  
  
template <class F>  
void foo(int a, int b, F consumer) {  
    consumer(a + b, a - b, a * b, a / b);  
}  
  
// use is simple... ignoring some results is possible as well  
foo(5, 12, [](int sum, int , int , int ){  
    std::cout << "sum is " << sum << '\n';  
});
```

This is known as "[continuation passing style](#)".

You can adapt a function returning a tuple into a continuation passing style function via:

```
Version ≥ C++17  
  
template<class Tuple>  
struct continuation {  
    Tuple t;  
    模板<类 F>  
    decltype(auto) operator->*(F&& f)&&{  
        return std::apply( std::forward<F>(f), std::move(t) );  
    }  
};  
std::tuple<int,int,int,int> foo(int a, int b);  
  
continuation(foo(5,12))->*[ ](int sum, auto&&...){  
    std::cout << "sum is " << sum << '\n';  
};
```

with more complex versions being writable in C++14 or C++11.

第24.6节：使用 std::pair

结构模板 `std::pair` 可以将恰好两个任意类型的返回值捆绑在一起：

```
#include <utility>
std::pair<int, int> foo(int a, int b) {
    return std::make_pair(a+b, a-b);
}
```

在 C++11 或更高版本中，可以使用初始化列表代替 `std::make_pair`：

版本 ≥ C++11

```
#include <utility>
std::pair<int, int> foo(int a, int b) {
    return {a+b, a-b};
}
```

返回的 `std::pair` 的各个值可以通过 `pair` 的 `first` 和 `second` 成员对象获取：

```
std::pair<int, int> mrvs = foo(5, 12);
std::cout << mrvs.first + mrvs.second << std::endl;
```

输出：

10

第24.7节：使用 std::array

版本 ≥ C++11

容器 `std::array` 可以将固定数量的返回值打包在一起。这个数量必须在编译时已知，且所有返回值必须是相同类型：

```
std::array<int, 4> bar(int a, int b) {
    return { a + b, a - b, a * b, a / b };
}
```

这取代了形式为 `int bar[4]` 的 C 风格数组。优点是现在可以对其使用各种 C++ 标准函数。它还提供了有用的成员函数，如 `at`，它是带边界检查的安全成员访问函数，以及 `size`，可以让你返回数组的大小而无需计算。

第24.8节：使用输出迭代器

通过向函数传递输出迭代器，可以返回多个相同类型的值。这在泛型函数（如标准库的算法）中尤为常见。

示例：

```
template <typename Incrementable, typename OutputIterator>
void generate_sequence(Incrementable from, Incrementable to, OutputIterator output) {
    for (Incrementable k = from; k != to; ++k)
        *output++ = k;
```

Section 24.6: Using std::pair

The struct template `std::pair` can bundle together *exactly* two return values, of any two types:

```
#include <utility>
std::pair<int, int> foo(int a, int b) {
    return std::make_pair(a+b, a-b);
}
```

With C++11 or later, an initializer list can be used instead of `std::make_pair`:

Version ≥ C++11

```
#include <utility>
std::pair<int, int> foo(int a, int b) {
    return {a+b, a-b};
}
```

The individual values of the returned `std::pair` can be retrieved by using the pair's `first` and `second` member objects:

```
std::pair<int, int> mrvs = foo(5, 12);
std::cout << mrvs.first + mrvs.second << std::endl;
```

Output:

10

Section 24.7: Using std::array

Version ≥ C++11

The container `std::array` can bundle together a fixed number of return values. This number has to be known at compile-time and all return values have to be of the same type:

```
std::array<int, 4> bar(int a, int b) {
    return { a + b, a - b, a * b, a / b };
}
```

This replaces c style arrays of the form `int bar[4]`. The advantage being that various c++ std functions can now be used on it. It also provides useful member functions like `at` which is a safe member access function with bound checking, and `size` which allows you to return the size of the array without calculation.

Section 24.8: Using Output Iterator

Several values of the same type can be returned by passing an output iterator to the function. This is particularly common for generic functions (like the algorithms of the standard library).

Example:

```
template<typename Incrementable, typename OutputIterator>
void generate_sequence(Incrementable from, Incrementable to, OutputIterator output) {
    for (Incrementable k = from; k != to; ++k)
        *output++ = k;
```

```
}
```

示例用法：

```
std::vector<int> digits;
generate_sequence(0, 10, std::back_inserter(digits));
// digits 现在包含 {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

第24.9节：使用 std::vector

std::vector 对于返回同一类型的动态数量变量非常有用。以下示例使用 int 作为数据类型，但 std::vector 可以保存任何可平凡复制的类型：

```
#include <vector>
#include <iostream>

// 以下函数返回包含 'a' 和 'b' 之间所有整数的向量
// (该函数可以返回最多 std::vector::max_size 个元素，前提是
// 系统主内存能够容纳这么多项)
std::vector<int> fillVectorFrom(int a, int b) {
    std::vector<int> temp;
    for (int i = a; i <= b; i++) {
        temp.push_back(i);
    }
    return temp;
}

int main() {
    // 将函数内部创建的填充向量赋值给新向量 'v'
    std::vector<int> v = fillVectorFrom(1, 10);

    // 输出 "1 2 3 4 5 6 7 8 9 10 "
    for (int i = 0; i < v.size(); i++) {
        std::cout << v[i] << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

```
}
```

Example usage:

```
std::vector<int> digits;
generate_sequence(0, 10, std::back_inserter(digits));
// digits 现在包含 {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Section 24.9: Using std::vector

A std::vector can be useful for returning a dynamic number of variables of the same type. The following example uses int as data type, but a std::vector can hold any type that is trivially copyable:

```
#include <vector>
#include <iostream>

// the following function returns all integers between and including 'a' and 'b' in a vector
// (the function can return up to std::vector::max_size elements with the vector, given that
// the system's main memory can hold that many items)
std::vector<int> fillVectorFrom(int a, int b) {
    std::vector<int> temp;
    for (int i = a; i <= b; i++) {
        temp.push_back(i);
    }
    return temp;
}

int main() {
    // assigns the filled vector created inside the function to the new vector 'v'
    std::vector<int> v = fillVectorFrom(1, 10);

    // prints "1 2 3 4 5 6 7 8 9 10 "
    for (int i = 0; i < v.size(); i++) {
        std::cout << v[i] << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

第25章：多态性

第25.1节：定义多态类

典型的例子是一个抽象的形状类，然后可以派生出正方形、圆形和其他具体形状。

父类：

让我们从多态类开始：

```
class Shape {  
public:  
    virtual ~Shape() = default;  
    virtual double get_surface() const = 0;  
    virtual void describe_object() const { std::cout << "这是一个形状" << std::endl; }  
  
    double get_doubled_surface() const { return 2 * get_surface(); }  
};
```

如何理解这个定义？

- 你可以通过引入带有关键字virtual的成员函数来定义多态行为。这里get_surface()和describe_object()显然对于正方形和圆形的实现会不同。当函数在对象上被调用时，函数会在运行时根据对象的实际类来确定。
- 为抽象形状定义get_surface()没有意义。这就是为什么该函数后面跟着= 0。这意味着该函数是纯虚函数。
- 多态类应始终定义虚析构函数。
- 你可以定义非虚成员函数。当这些函数被对象调用时，函数的选择将取决于编译时使用的类。这里get_doubled_surface()就是以这种方式定义的。
- 包含至少一个纯虚函数的类是抽象类。抽象类不能被实例化。你只能拥有抽象类型的指针或引用。

派生类

一旦定义了多态基类，你就可以派生它。例如：

```
class Square : public Shape {  
    Point top_left;  
    double side_length;  
public:  
    Square (const Point& top_left, double side)  
        : top_left(top_left), side_length(side_length) {}  
  
    double get_surface() override { return side_length * side_length; }  
    void describe_object() override {  
        std::cout << "这是一个起点为 " << top_left.x << ", " << top_left.y  
              << ", 边长为 " << side_length << std::endl;  
    }  
};
```

Chapter 25: Polymorphism

Section 25.1: Define polymorphic classes

The typical example is an abstract shape class, that can then be derived into squares, circles, and other concrete shapes.

The parent class:

Let's start with the polymorphic class:

```
class Shape {  
public:  
    virtual ~Shape() = default;  
    virtual double get_surface() const = 0;  
    virtual void describe_object() const { std::cout << "this is a shape" << std::endl; }  
  
    double get_doubled_surface() const { return 2 * get_surface(); }  
};
```

How to read this definition ?

- You can define polymorphic behavior by introduced member functions with the keyword `virtual`. Here `get_surface()` and `describe_object()` will obviously be implemented differently for a square than for a circle. When the function is invoked on an object, function corresponding to the real class of the object will be determined at runtime.
- It makes no sense to define `get_surface()` for an abstract shape. This is why the function is followed by `= 0`. This means that the function is *pure virtual function*.
- A polymorphic class should always define a virtual destructor.
- You may define non virtual member functions. When these function will be invoked for an object, the function will be chosen depending on the class used at compile-time. Here `get_double_surface()` is defined in this way.
- A class that contains at least one pure virtual function is an abstract class. Abstract classes cannot be instantiated. You may only have pointers or references of an abstract class type.

Derived classes

Once a polymorphic base class is defined you can derive it. For example:

```
class Square : public Shape {  
    Point top_left;  
    double side_length;  
public:  
    Square (const Point& top_left, double side)  
        : top_left(top_left), side_length(side_length) {}  
  
    double get_surface() override { return side_length * side_length; }  
    void describe_object() override {  
        std::cout << "this is a square starting at " << top_left.x << ", " << top_left.y  
              << " with a length of " << side_length << std::endl;  
    }  
};
```

一些说明：

- 你可以定义或重写父类的任何虚函数。父类中函数是虚函数的事实使得它在派生类中也是虚函数。无需再次告诉编译器关键字 `virtual`。但建议在函数声明末尾添加关键字 `override`，以防止因函数签名细微差异而导致的潜在错误。
- 如果父类的所有纯虚函数都被定义，则可以实例化该类的对象，否则它也将成为抽象类。
- 你不必重写所有虚函数。如果父类的版本满足你的需求，可以保留父类版本。

实例化示例

```
int main() {  
  
    Square square(Point(10.0, 0.0), 6); // 我们知道它是一个正方形，编译器也知道  
    square.describe_object();  
    std::cout << "表面面积: " << square.get_surface() << std::endl;  
  
    Circle circle(Point(0.0, 0.0), 5);  
  
    Shape *ps = nullptr; // 我们还不知道对象的真实类型  
    ps = &circle; // 它是一个圆，但也可能是一个正方形  
    ps->describe_object();  
    std::cout << "Surface: " << ps->get_surface() << std::endl;  
}
```

第25.2节：安全向下转换

假设你有一个指向多态类对象的指针：

```
Shape *ps; // 参见定义多态类的示例  
ps = get_a_new_random_shape(); // 如果你还没有这样的函数，  
// 你也可以直接写 ps = new Square(0.0, 0.0, 5);
```

向下转换是指将一个通用的多态 `Shape` 类型转换为其派生的、更具体的形状类型，比如 `Square` 或 `Circle`。

为什么要向下转换？

大多数情况下，你不需要知道对象的真实类型，因为虚函数允许你独立于类型来操作对象：

```
std::cout << "Surface: " << ps->get_surface() << std::endl;
```

如果你不需要任何向下转换，那么你的设计就是完美的。

但是，有时你可能需要进行向下转换。一个典型的例子是当你想调用仅存在于子类中的非虚函数时。

例如考虑圆。只有圆才有直径。因此类可以定义为：

```
class Circle: public Shape { // 关于 Shape，请参见定义多态类的示例  
    Point center;  
    double radius;  
public:
```

Some explanations:

- You can define or override any of the virtual functions of the parent class. The fact that a function was virtual in the parent class makes it virtual in the derived class. No need to tell the compiler the keyword `virtual` again. But it's recommended to add the keyword `override` at the end of the function declaration, in order to prevent subtle bugs caused by unnoticed variations in the function signature.
- If all the pure virtual functions of the parent class are defined you can instantiate objects for this class, else it will also become an abstract class.
- You are not obliged to override all the virtual functions. You can keep the version of the parent if it suits your need.

Example of instantiation

```
int main() {  
  
    Square square(Point(10.0, 0.0), 6); // we know it's a square, the compiler also  
    square.describe_object();  
    std::cout << "Surface: " << square.get_surface() << std::endl;  
  
    Circle circle(Point(0.0, 0.0), 5);  
  
    Shape *ps = nullptr; // we don't know yet the real type of the object  
    ps = &circle; // it's a circle, but it could as well be a square  
    ps->describe_object();  
    std::cout << "Surface: " << ps->get_surface() << std::endl;  
}
```

Section 25.2: Safe downcasting

Suppose that you have a pointer to an object of a polymorphic class:

```
Shape *ps; // see example on defining a polymorphic class  
ps = get_a_new_random_shape(); // if you don't have such a function yet, you  
// could just write ps = new Square(0.0, 0.0, 5);
```

a downcast would be to cast from a general polymorphic `Shape` down to one of its derived and more specific shape like `Square` or `Circle`.

Why to downcast ?

Most of the time, you would not need to know which is the real type of the object, as the virtual functions allow you to manipulate your object independently of its type:

```
std::cout << "Surface: " << ps->get_surface() << std::endl;
```

If you don't need any downcast, your design would be perfect.

However, you may need sometimes to downcast. A typical example is when you want to invoke a non virtual function that exist only for the child class.

Consider for example circles. Only circles have a diameter. So the class would be defined as :

```
class Circle: public Shape { // for Shape, see example on defining a polymorphic class  
    Point center;  
    double radius;  
public:
```

```

Circle (const Point& center, double radius)
    : center(center), radius(radius) {}

double get_surface() const override { return r * r * M_PI; }

// 这仅适用于圆。对其他形状没有意义
double get_diameter() const { return 2 * r; }
};

```

成员函数 `get_diameter()` 仅存在于圆中。它没有为 `Shape` 对象定义：

```

Shape* ps = get_any_shape();
ps->get_diameter(); // 啊呀！！！编译错误

```

如何进行向下转换？

如果你确定 `ps` 指向一个圆，你可以选择使用 `static_cast`：

```
std::cout << "Diameter: " << static_cast<Circle*>(ps)->get_diameter() << std::endl;
```

这将起作用。但风险很大：如果 `ps` 不是一个 `Circle`，你的代码行为将是未定义的。

所以，与其玩俄罗斯轮盘赌，不如安全地使用 `dynamic_cast`。这专门用于多态类：

```

int main() {
Circle circle(Point(0.0, 0.0), 10);
Shape &shape = circle;

std::cout << "该形状的表面积为 " << shape.get_surface() << std::endl;

//shape.get_diameter(); // 啊呀！！！编译错误

Circle *pc = dynamic_cast<Circle*>(&shape); // 如果ps不是circle，则pc为nullptr
if (pc)
std::cout << "该形状是直径为 " << pc->get_diameter() << std::endl;
else
std::cout << "该形状不是圆形！" << std::endl;
}

```

注意，`dynamic_cast` 不能用于非多态类。你需要类或其父类中至少有一个虚函数才能使用它。

第25.3节：多态与析构函数

如果一个类打算以多态方式使用，派生实例以基类指针/引用存储，其基类的析构函数应当是 `virtual` 或 `protected` 的。在前者情况下，这将导致对象销毁时检查 `vtable`，自动根据动态类型调用正确的析构函数。在后者情况下，通过基类指针/引用销毁对象被禁用，只有显式将对象视为其实际类型时才能删除。

```

struct VirtualDestructor {
    virtual ~VirtualDestructor() = default;
};

struct VirtualDerived : VirtualDestructor {};

```

```

Circle (const Point& center, double radius)
    : center(center), radius(radius) {}

double get_surface() const override { return r * r * M_PI; }

// this is only for circles. Makes no sense for other shapes
double get_diameter() const { return 2 * r; }
};

```

The `get_diameter()` member function only exists for circles. It was not defined for a `Shape` object:

```

Shape* ps = get_any_shape();
ps->get_diameter(); // OUCH !!! Compilation error

```

How to downcast ?

If you'd know for sure that `ps` points to a `circle` you could opt for a `static_cast`:

```
std::cout << "Diameter: " << static_cast<Circle*>(ps)->get_diameter() << std::endl;
```

This will do the trick. But it's very risky: if `ps` appears to be anything else than a `Circle` the behavior of your code will be undefined.

So rather than playing Russian roulette, you should safely use a `dynamic_cast`. This is specifically for polymorphic classes :

```

int main() {
Circle circle(Point(0.0, 0.0), 10);
Shape &shape = circle;

std::cout << "The shape has a surface of " << shape.get_surface() << std::endl;

//shape.get_diameter(); // OUCH !!! Compilation error

Circle *pc = dynamic_cast<Circle*>(&shape); // will be nullptr if ps wasn't a circle
if (pc)
std::cout << "The shape is a circle of diameter " << pc->get_diameter() << std::endl;
else
std::cout << "The shape isn't a circle !" << std::endl;
}

```

Note that `dynamic_cast` is not possible on a class that is not polymorphic. You'd need at least one virtual function in the class or its parents to be able to use it.

Section 25.3: Polymorphism & Destructors

If a class is intended to be used polymorphically, with derived instances being stored as base pointers/references, its base class' destructor should be either `virtual` or `protected`. In the former case, this will cause object destruction to check the `vtable`, automatically calling the correct destructor based on the dynamic type. In the latter case, destroying the object through a base class pointer/reference is disabled, and the object can only be deleted when explicitly treated as its actual type.

```

struct VirtualDestructor {
    virtual ~VirtualDestructor() = default;
};

struct VirtualDerived : VirtualDestructor {};

```

```

struct ProtectedDestructor {
    protected:
~ProtectedDestructor() = default;
};

struct ProtectedDerived : ProtectedDestructor {
    ~ProtectedDerived() = default;
};

// ...

VirtualDestructor* vd = new VirtualDerived;
delete vd; // 在vtable中查找VirtualDestructor::~VirtualDestructor(), 发现是
// VirtualDerived::~VirtualDerived(), 调用该析构函数。

```

```

ProtectedDestructor* pd = new ProtectedDerived;
delete pd; // 错误：ProtectedDestructor::~ProtectedDestructor() 是受保护的。
delete static_cast<ProtectedDerived*>(pd); // 正确。

```

这两种做法都保证了派生类的析构函数总会在派生类实例上被调用，从而防止内存泄漏。

```

struct ProtectedDestructor {
    protected:
~ProtectedDestructor() = default;
};

struct ProtectedDerived : ProtectedDestructor {
    ~ProtectedDerived() = default;
};

// ...

VirtualDestructor* vd = new VirtualDerived;
delete vd; // Looks up VirtualDestructor::~VirtualDestructor() in vtable, sees it's
// VirtualDerived::~VirtualDerived(), calls that.

ProtectedDestructor* pd = new ProtectedDerived;
delete pd; // Error: ProtectedDestructor::~ProtectedDestructor() is protected.
delete static_cast<ProtectedDerived*>(pd); // Good.

```

Both of these practices guarantee that the derived class' destructor will always be called on derived class instances, preventing memory leaks.

第26章：参考文献

第26.1节：定义引用

引用的行为类似于常量指针，但并不完全相同。引用通过在类型名后加上一个&符号来定义
& 加在类型名后。

```
int i = 10;  
int &refi = i;
```

这里，`refi` 是绑定到 `i` 的引用。

引用抽象了指针的语义，表现得像是底层对象的别名：

```
refi = 20; // i = 20;
```

你也可以在一个定义中定义多个引用：

```
int i = 10, j = 20;  
int &refi = i, &refj = j;  
  
// 常见的陷阱：  
// int& refi = i, k = j;  
// refi 将是 int& 类型。  
// 但是，k 将是 int 类型，而不是 int&！
```

引用必须在定义时正确初始化，且之后不能修改。以下代码片段会导致编译错误：

```
int &i; // 错误：引用变量 'i' 的声明需要一个初始化器
```

你也不能像指针那样直接将引用绑定到`nullptr`：

```
int *const ptri = nullptr;  
int &refi = nullptr; // 错误：非 const 左值引用类型 'int' 不能绑定到类型为 'nullptr_t' 的临时对象
```

Chapter 26: References

Section 26.1: Defining a reference

References behaves similarly, but not entirely like const pointers. A reference is defined by suffixing an ampersand & to a type name.

```
int i = 10;  
int &refi = i;
```

Here, `refi` is a reference bound to `i`.

References abstracts the semantics of pointers, acting like an alias to the underlying object:

```
refi = 20; // i = 20;
```

You can also define multiple references in a single definition:

```
int i = 10, j = 20;  
int &refi = i, &refj = j;  
  
// Common pitfall :  
// int& refi = i, k = j;  
// refi will be of type int&.  
// though, k will be of type int, not int&!
```

References **must** be initialized correctly at the time of definition, and cannot be modified afterwards. The following piece of codes causes a compile error:

```
int &i; // error: declaration of reference variable 'i' requires an initializer
```

You also cannot bind directly a reference to `nullptr`, unlike pointers:

```
int *const ptri = nullptr;  
int &refi = nullptr; // error: non-const lvalue reference to type 'int' cannot bind to a temporary  
of type 'nullptr_t'
```

第27章：值语义和引用语义

第27.1节：定义

如果一个类型的对象的可观察状态在功能上与该类型的所有其他对象不同，则该类型具有值语义。这意味着如果你复制一个对象，你将得到一个新对象，对新对象的修改不会以任何方式影响旧对象。

大多数基本的C++类型具有值语义：

```
int i = 5;
int j = i; //复制
j += 20;
std::cout << i; //输出5；i不受j的影响。
```

大多数标准库定义的类型也具有值语义：

```
std::vector<int> v1(5, 12); //包含5个值的数组，每个值为12。
std::vector<int> v2 = v1; //复制该向量。
v2[3] = 6; v2[4] = 9;
std::cout << v1[3] << " " << v1[4]; //输出"12 12"，因为v1未被改变。
```

如果某类型的实例可以与另一个（外部）对象共享其可观察状态，使得操作一个对象会导致另一个对象的状态发生变化，则称该类型具有引用语义。

C++指针在指向哪个对象方面具有值语义，但在指向对象的状态方面具有引用语义：

```
int *pi = new int(4);
int *pi2 = pi;
pi = new int(16);
assert(pi2 != pi); //将始终通过。

int *pj = pi;
*pj += 5;
std::cout << *pi; //输出9，因为`pi`和`pj`引用同一个对象。
```

C++引用也具有引用语义。

第27.2节：深拷贝和移动支持

如果一个类型希望具有值语义，并且需要存储动态分配的对象，那么在拷贝操作时，该类型需要为这些对象分配新的副本。拷贝赋值时也必须这样做。

这种拷贝称为“深拷贝”。它实际上将本来是引用语义的内容转变为值语义：

```
结构体 Inner {整型 i;} ;

常量整型 NUM_INNER = 5;
类 Value
{
private:
内部*数组_;//通常具有引用语义。
```

Chapter 27: Value and Reference Semantics

Section 27.1: Definitions

A type has value semantics if the object's observable state is functionally distinct from all other objects of that type. This means that if you copy an object, you have a new object, and modifications of the new object will not be in any way visible from the old object.

Most basic C++ types have value semantics:

```
int i = 5;
int j = i; //Copied
j += 20;
std::cout << i; //Prints 5; i is unaffected by changes to j.
```

Most standard-library defined types have value semantics too:

```
std::vector<int> v1(5, 12); //array of 5 values, 12 in each.
std::vector<int> v2 = v1; //Copies the vector.
v2[3] = 6; v2[4] = 9;
std::cout << v1[3] << " " << v1[4]; //Writes "12 12", since v1 is unchanged.
```

A type is said to have reference semantics if an instance of that type can share its observable state with another object (external to it), such that manipulating one object will cause the state to change within another object.

C++ pointers have value semantics with regard to which object they point to, but they have reference semantics with regard to the state of the object they point to:

```
int *pi = new int(4);
int *pi2 = pi;
pi = new int(16);
assert(pi2 != pi); //Will always pass.

int *pj = pi;
*pj += 5;
std::cout << *pi; //Writes 9, since `pi` and `pj` reference the same object.
```

C++ references have reference semantics as well.

Section 27.2: Deep copying and move support

If a type wishes to have value semantics, and it needs to store objects that are dynamically allocated, then on copy operations, the type will need to allocate new copies of those objects. It must also do this for copy assignment.

This kind of copying is called a "deep copy". It effectively takes what would have otherwise been reference semantics and turns it into value semantics:

```
struct Inner {int i;} ;

const int NUM_INNER = 5;
class Value
{
private:
Inner *array_;//Normally has reference semantics.
```

```
public:  
Value() : 数组_(new 内部[NUM_INNER]){}  
  
~Value() {delete[] 数组_;
```

```
Value(const Value &val) : 数组_(new 内部[NUM_INNER])  
{  
    for(int i = 0; i < NUM_INNER; ++i)  
        数组_[i] = val.数组_[i];  
}
```

```
Value &operator=(const Value &val)  
{  
    for(int i = 0; i < NUM_INNER; ++i)  
        数组_[i] = val.数组_[i];  
    return *this;  
}  
};
```

版本 ≥ C++11

移动语义允许像Value这样的类型避免真正复制其引用的数据。如果用户以一种触发移动的方式使用该值，“被复制”的对象可以被置为空，丢失其引用的数据：

```
结构体 内部 {int i;};  
  
constexpr auto NUM_INNER = 5;  
类 Value  
{  
private:  
    内部*数组_;//通常具有引用语义。  
  
public:  
    Value() : 数组_(new 内部[NUM_INNER]){}  
  
    //即使是nullptr也可以删除  
    ~Value() {delete[] 数组_;}  
  
    Value(const Value &val) : array_(new Inner[NUM_INNER])  
    {  
        for(int i = 0; i < NUM_INNER; ++i)  
            数组_[i] = val.数组_[i];  
    }  
  
    Value &operator=(const Value &val)  
        for(int i = 0; i < NUM_INNER; ++i)  
            数组_[i] = val.数组_[i];  
        return *this;  
    }
```

//移动意味着不分配内存。

//不能抛出异常。

```
Value(Value &&val) noexcept : array_(val.array_)  
{  
    //我们已经窃取了旧值。  
    val.array_ = nullptr;  
}
```

//不能抛出异常。

```
Value &operator=(Value &&val) noexcept  
{
```

```
public:  
    Value() : array_(new Inner[NUM_INNER]){}  
  
    ~Value() {delete[] array_;}  
  
    Value(const Value &val) : array_(new Inner[NUM_INNER])  
    {  
        for(int i = 0; i < NUM_INNER; ++i)  
            array_[i] = val.array_[i];  
    }  
  
    Value &operator=(const Value &val)  
    {  
        for(int i = 0; i < NUM_INNER; ++i)  
            array_[i] = val.array_[i];  
        return *this;  
    }  
};  
Version ≥ C++11
```

Move semantics allow a type like Value to avoid truly copying its referenced data. If the user uses the value in a way that provokes a move, the "copied" from object can be left empty of the data it referenced:

```
struct Inner {int i;};  
  
constexpr auto NUM_INNER = 5;  
class Value  
{  
private:  
    Inner *array_;//Normally has reference semantics.  
  
public:  
    Value() : array_(new Inner[NUM_INNER]){}  
  
    //OK to delete even if nullptr  
    ~Value() {delete[] array_;}  
  
    Value(const Value &val) : array_(new Inner[NUM_INNER])  
    {  
        for(int i = 0; i < NUM_INNER; ++i)  
            array_[i] = val.array_[i];  
    }  
  
    Value &operator=(const Value &val)  
    {  
        for(int i = 0; i < NUM_INNER; ++i)  
            array_[i] = val.array_[i];  
        return *this;  
    }  
  
    //Movement means no memory allocation.  
    //Cannot throw exceptions.  
    Value(Value &&val) noexcept : array_(val.array_)  
    {  
        //We've stolen the old value.  
        val.array_ = nullptr;  
    }  
  
    //Cannot throw exceptions.  
    Value &operator=(Value &&val) noexcept  
{
```

```

//巧妙的技巧。由于`val`很快就会被销毁,
//我们交换他的数据和我们的数据。他的析构函数将销毁我们的数据。
std::swap(array_, val.array_);
}
};

```

实际上，如果我们想禁止深拷贝，同时仍允许对象被移动，我们甚至可以使这种类型不可复制。

```

结构体 内部 {int i;};

constexpr auto NUM_INNER = 5;
类 Value
{
private:
    内部*array_; //通常具有引用语义。

public:
    Value() : 数组_(new 内部[NUM_INNER]){}

    //即使是nullptr也可以删除
    ~Value() {delete[] array_;}

    Value(const Value &val) = delete;
    Value &operator=(const Value &val) = delete;

    //移动意味着不分配内存。
    //不能抛出异常。
    Value(Value &&val) noexcept : array_(val.array_)
    {
        //我们已经窃取了旧值。
        val.array_ = nullptr;
    }

    //不能抛出异常。
    Value &operator=(Value &&val) noexcept
    {
        //巧妙的技巧。由于`val`很快就会被销毁,
        //我们交换他的数据和我们的数据。他的析构函数将销毁我们的数据。
        std::swap(array_, val.array_);
    }
};

```

我们甚至可以通过使用unique_ptr来应用零规则（Rule of Zero）：

```

结构体 内部 {int i;};

constexpr auto NUM_INNER = 5;
类 Value
{
private:
    unique_ptr<Inner []>array_; //仅限移动类型。

public:
    Value() : 数组_(new 内部[NUM_INNER]){}

    //无需显式删除，甚至无需声明。
    ~Value() = default; {delete[] array_;}

    //无需显式删除，甚至无需声明。
    Value(const Value &val) = default;

```

```

//Clever trick. Since `val` is going to be destroyed soon anyway,
//we swap his data with ours. His destructor will destroy our data.
std::swap(array_, val.array_);
}
};

```

Indeed, we can even make such a type non-copyable, if we want to forbid deep copies while still allowing the object to be moved around.

```

struct Inner {int i;};

constexpr auto NUM_INNER = 5;
class Value
{
private:
    Inner *array_; //Normally has reference semantics.

public:
    Value() : array_(new Inner[NUM_INNER]){}

    //OK to delete even if nullptr
    ~Value() {delete[] array_;}

    Value(const Value &val) = delete;
    Value &operator=(const Value &val) = delete;

    //Movement means no memory allocation.
    //Cannot throw exceptions.
    Value(Value &&val) noexcept : array_(val.array_)
    {
        //We've stolen the old value.
        val.array_ = nullptr;
    }

    //Cannot throw exceptions.
    Value &operator=(Value &&val) noexcept
    {
        //Clever trick. Since `val` is going to be destroyed soon anyway,
        //we swap his data with ours. His destructor will destroy our data.
        std::swap(array_, val.array_);
    }
};

```

We can even apply the Rule of Zero, through the use of unique_ptr:

```

struct Inner {int i;};

constexpr auto NUM_INNER = 5;
class Value
{
private:
    unique_ptr<Inner []>array_; //Move-only type.

public:
    Value() : array_(new Inner[NUM_INNER]){}

    //No need to explicitly delete. Or even declare.
    ~Value() = default; {delete[] array_;}

    //No need to explicitly delete. Or even declare.
    Value(const Value &val) = default;

```

```
Value &operator=(const Value &val) = default;  
  
//将执行逐元素移动。  
Value(Value &&val) noexcept = default;  
  
//将执行逐元素移动。  
Value &operator=(Value &&val) noexcept = default;  
};
```

```
Value &operator=(const Value &val) = default;  
  
//Will perform an element-wise move.  
Value(Value &&val) noexcept = default;  
  
//Will perform an element-wise move.  
Value &operator=(Value &&val) noexcept = default;  
};
```

第28章：C++函数“值传递”与“引用传递”

本节的范围是解释函数调用时参数在理论和实现上的差异。

具体来说，参数可以看作函数调用前和函数内部的变量，其中这些变量的可见行为和访问权限因传递方式不同而异。

此外，本节还解释了函数调用后变量及其对应值的可重用性。

第28.1节：值传递

调用函数时，程序栈上会创建新的元素。这些元素包括一些关于函数的信息，以及参数和返回值的空间（内存位置）。

将参数传递给函数时，使用变量（或字面量）的值会被复制到函数参数的内存位置。这意味着现在有两个内存位置保存相同的值。在函数内部，我们只操作参数的内存位置。

函数执行结束后，程序栈上的内存会被弹出（移除），这会清除函数调用的所有数据，包括我们在函数内部使用的参数内存位置。因此，函数内部改变的值不会影响外部变量的值。

```
int func(int f, int b) {  
    //新变量被创建，外部的值被复制  
    //f 的值为 0  
    //inner_b 的值为 1  
    f = 1;  
    //f 的值为 1  
    b = 2;  
    //inner_b 的值为 2  
    return f+b;  
}  
  
int main(void) {  
    int a = 0;  
    int b = 1; //outer_b  
    int c;  
  
    c = func(a,b);  
    //返回值被复制给 c  
  
    //a 的值为 0  
    //outer_b 的值为 1 <--- outer_b 和 inner_b 是不同的变量  
    //c 的值为 3  
}
```

在这段代码中，我们在主函数内部创建变量。这些变量被赋予值。在调用函数时，会创建两个新变量：f 和 inner_b，其中 b 与外部变量同名，但不共享内存位置。变量 a<->f 和 b<->b 的行为是相同的。

下图象征了堆栈上发生的情况以及为什么变量 b 没有变化。该图并不完全准确，但强调了示例的重点。

Chapter 28: C++ function "call by value" vs. "call by reference"

The scope of this section is to explain the differences in theory and implementation for what happens with the parameters of a function upon calling.

In detail the parameters can be seen as variables before the function call and inside the function, where the visible behaviour and accessibility to these variables differs with the method used to hand them over.

Additionally, the reusability of variables and their respective values after the function call also is explained by this topic.

Section 28.1: Call by value

Upon calling a function there are new elements created on the program stack. These include some information about the function and also space (memory locations) for the parameters and the return value.

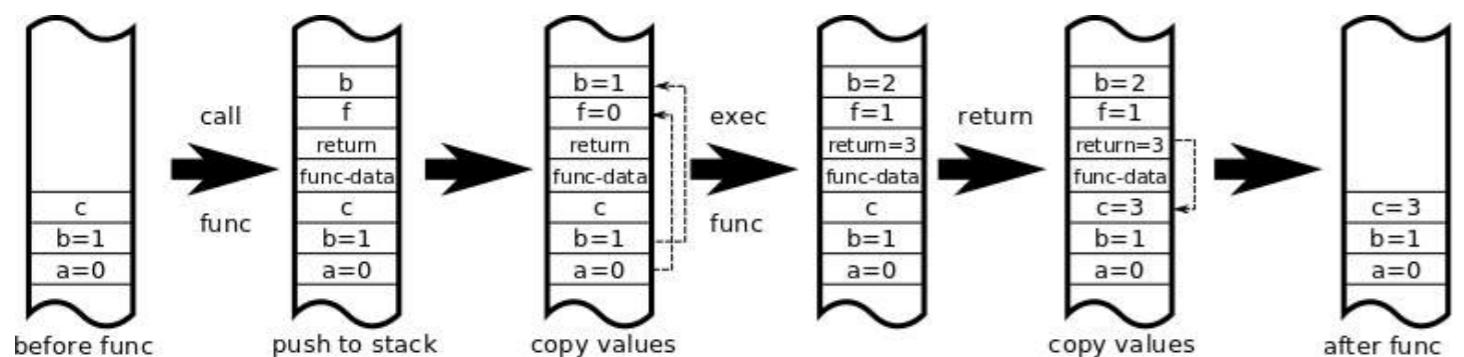
When handing over a parameter to a function the value of the used variable (or literal) is copied into the memory location of the function parameter. This implies that now there are two memory locations with the same value. Inside of the function we only work on the parameter memory location.

After leaving the function the memory on the program stack is popped (removed) which erases all data of the function call, including the memory location of the parameters we used inside. Thus, the values changed inside the function do not affect the outside variables values.

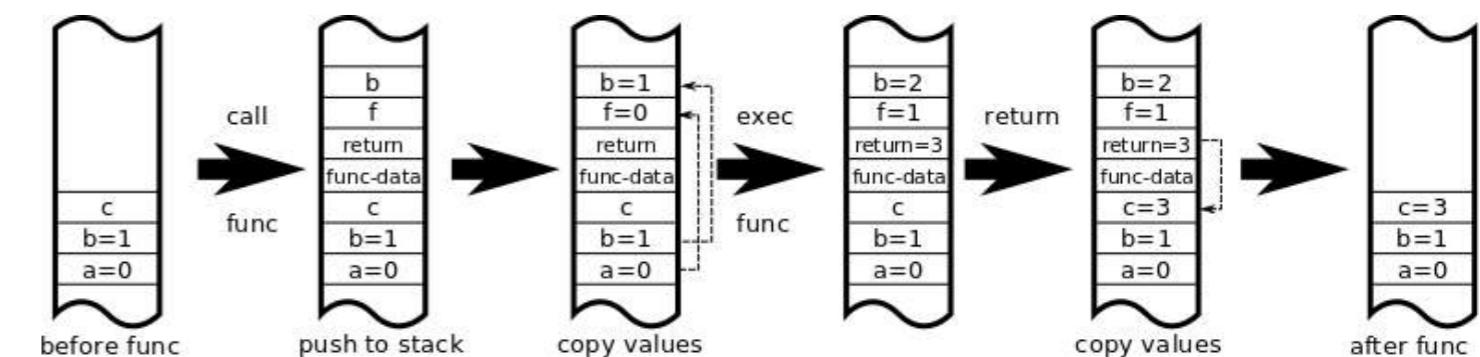
```
int func(int f, int b) {  
    //new variables are created and values from the outside copied  
    //f has a value of 0  
    //inner_b has a value of 1  
    f = 1;  
    //f has a value of 1  
    b = 2;  
    //inner_b has a value of 2  
    return f+b;  
}  
  
int main(void) {  
    int a = 0;  
    int b = 1; //outer_b  
    int c;  
  
    c = func(a,b);  
    //the return value is copied to c  
  
    //a has a value of 0  
    //outer_b has a value of 1 <--- outer_b and inner_b are different variables  
    //c has a value of 3  
}
```

In this code we create variables inside the main function. These get assigned values. Upon calling the functions there are two new variables created: f and inner_b where b shares the name with the outer variable it does not share the memory location. The behaviour of a<->f and b<->b is identical.

The following graphic symbolizes what is happening on the stack and why there is no change in variable b. The graphic is not fully accurate but emphasizes the example.



之所以称为“按值调用”，是因为我们不是传递变量本身，而只是传递这些变量的值。



It is called "call by value" because we do not hand over the variables but only the values of these variables.

第29章：复制与赋值

rhs 等号右侧，既适用于复制构造函数也适用于赋值构造函数。例如赋值构造函数：`MyClass operator=(MyClass& rhs);`
占位符 占位符

第29.1节：赋值运算符

赋值运算符是指用另一个已存在（先前初始化过的）对象的数据替换当前对象的数据。我们以此为例：

```
// 赋值运算符
#include <iostream>
#include <string>

using std::cout;
using std::endl;

class Foo
{
public:
    Foo(int data)
    {
        this->data = data;
    }
    ~Foo(){}
    Foo& operator=(const Foo& rhs)
    {
        data = rhs.data;
        return *this;
    }

    int data;
};

int main()
{
    Foo foo(2); //调用了 Foo(int data) 构造函数
    Foo foo2(42);
    foo = foo2; // 调用了赋值运算符
    cout << foo.data << endl; // 输出 42
}
```

你可以看到这里我在已经初始化了foo对象后调用了赋值运算符。然后我将foo2赋值给foo。所有在调用等号运算符时发生的变化都定义在你的`operator=`函数中。你可以在[这里](http://cpp.sh/3qtbm)看到可运行的输出：<http://cpp.sh/3qtbm>

第29.2节：拷贝构造函数

另一方面，拷贝构造函数与赋值构造函数完全相反。这次，它用于初始化一个尚不存在（或之前未初始化）的对象。这意味着它会复制你赋值给它的对象的所有数据，而实际上并没有初始化被复制的对象。现在让我们来看一下之前相同的代码，但将赋值构造函数修改为拷贝构造函数：

```
// 拷贝构造函数
#include <iostream>
#include <string>
```

Chapter 29: Copying vs Assignment

rhs Right Hand Side of the equality for both copy and assignment constructors. For example the assignment constructor : `MyClass operator=(MyClass& rhs);`
Placeholder Placeholder

Section 29.1: Assignment Operator

The Assignment Operator is when you replace the data with an already existing(previously initialized) object with some other object's data. Lets take this as an example:

```
// Assignment Operator
#include <iostream>
#include <string>

using std::cout;
using std::endl;

class Foo
{
public:
    Foo(int data)
    {
        this->data = data;
    }
    ~Foo(){}
    Foo& operator=(const Foo& rhs)
    {
        data = rhs.data;
        return *this;
    }

    int data;
};

int main()
{
    Foo foo(2); //Foo(int data) called
    Foo foo2(42);
    foo = foo2; // Assignment Operator Called
    cout << foo.data << endl; //Prints 42
}
```

You can see here I call the assignment operator when I already initialized the foo object. Then later I assign foo2 to foo . All the changes to appear when you call that equal sign operator is defined in your `operator=` function. You can see a runnable output here: <http://cpp.sh/3qtbm>

Section 29.2: Copy Constructor

Copy constructor on the other hand , is the complete opposite of the Assignment Constructor. This time, it is used to initialize an already nonexistent(or non-previously initialized) object. This means it copies all the data from the object you are assigning it to , without actually initializing the object that is being copied onto. Now Let's take a look at the same code as before but modify the assignment constructor to be a copy constructor :

```
// Copy Constructor
#include <iostream>
#include <string>
```

```

using std::cout;
using std::endl;

class Foo
{
public:
    Foo(int data)
    {
        this->data = data;
    }
    ~Foo() {}
    Foo(const Foo& rhs)
    {
        data = rhs.data;
    }

    int data;
};

int main()
{
    Foo foo(2); // 调用 Foo(int data)
    Foo foo2 = foo; // 调用拷贝构造函数
    cout << foo2.data << endl;
}

```

你可以看到这里 main 函数中的 Foo foo2 = foo; 我直接在实际初始化之前赋值对象，正如之前所说，这就是拷贝构造函数。并且注意我不需要为 foo2 对象传递 int 参数，因为我自动从对象 foo 中获取了之前的数据。下面是一个示例输出：

<http://cpp.sh/5iu7>

第29.3节：拷贝构造函数与赋值构造函数的比较

好了，我们已经简要了解了拷贝构造函数和赋值构造函数是什么，并给出了各自的示例，现在让我们在同一段代码中同时看看它们。这段代码将与上面两个类似。我们来看这个：

```

// 复制构造函数与赋值构造函数
#include <iostream>
#include <string>

using std::cout;
using std::endl;

class Foo
{
public:
    Foo(int data)
    {
        this->data = data;
    }
    ~Foo() {}
    Foo(const Foo& rhs)
    {
        data = rhs.data;
    }

    Foo& operator=(const Foo& rhs)
    {
        data = rhs.data;
    }

```

```

using std::cout;
using std::endl;

class Foo
{
public:
    Foo(int data)
    {
        this->data = data;
    }
    ~Foo() {}
    Foo(const Foo& rhs)
    {
        data = rhs.data;
    }

    int data;
};

int main()
{
    Foo foo(2); // Foo(int data) called
    Foo foo2 = foo; // Copy Constructor called
    cout << foo2.data << endl;
}

```

You can see here Foo foo2 = foo; in the main function I immediately assign the object before actually initializing it, which as said before means it's a copy constructor. And notice that I didn't need to pass the parameter int for the foo2 object since I automatically pulled the previous data from the object foo. Here is an example output :
<http://cpp.sh/5iu7>

Section 29.3: Copy Constructor Vs Assignment Constructor

Ok we have briefly looked over what the copy constructor and assignment constructor are above and gave examples of each now let's see both of them in the same code. This code will be similar as above two. Let's take this :

```

// Copy vs Assignment Constructor
#include <iostream>
#include <string>

using std::cout;
using std::endl;

class Foo
{
public:
    Foo(int data)
    {
        this->data = data;
    }
    ~Foo() {}
    Foo(const Foo& rhs)
    {
        data = rhs.data;
    }

    Foo& operator=(const Foo& rhs)
    {
        data = rhs.data;
    }

```

```

    return *this;
}

int data;
};

int main()
{
    Foo foo(2); //调用了Foo(int data) / 普通构造函数
    Foo foo2 = foo; //调用了拷贝构造函数
    cout << foo2.data << endl;

    Foo foo3(42);
    foo3=foo; //调用了赋值构造函数
    cout << foo3.data << endl;
}

```

输出：

```

2

```

Foo foo2 = foo;。因为我们之前没有初始化它。
然后接下来我们调用赋值运算符给 foo3，因为它已经被初始化了 foo3=foo;

```

    return *this;
}

int data;
};

int main()
{
    Foo foo(2); //Foo(int data) / Normal Constructor called
    Foo foo2 = foo; //Copy Constructor Called
    cout << foo2.data << endl;

    Foo foo3(42);
    foo3=foo; //Assignment Constructor Called
    cout << foo3.data << endl;
}

```

Output:

```

2

```

Here you can see we first call the copy constructor by executing the line Foo foo2 = foo;. Since we didn't initialize it previously. And then next we call the assignment operator on foo3 since it was already initialized foo3=foo;

第30章：指针

指针是指向内存中某个位置的地址。它们通常用于让函数或数据结构能够知道并修改内存，而无需复制所指向的内存。指针既可用于原始（内置）类型，也可用于用户定义的类型。

指针使用“解引用” *、“取地址” & 和“箭头” -> 运算符。'*' 和 '->' 运算符用于访问所指向的内存，& 运算符用于获取内存地址。

第30.1节：指针操作

指针有两个运算符：取地址运算符 (&)：返回其操作数的内存地址。

解引用运算符(*)：返回位于其操作符指定地址的变量的值。

```
int var = 20;
int *ptr;
ptr = &var;

cout << var << endl;
//输出 20 (var 的值)

cout << ptr << endl;
//输出 0x234f119 (变量的内存地址)

cout << *ptr << endl;
//输出 20 (存储在指针 ptr 中的变量的值)
```

星号 (*) 用于声明指针，目的是简单地表示它是一个指针。不要将其与解引用 (dereference) 操作符混淆，解引用操作符用于获取指定地址处存储的值。它们只是用相同符号表示的两种不同的东西。

第30.2节：指针基础

版本 < C++11

注意：在以下所有内容中，假设存在 C++11 常量 `nullptr`。对于早期版本，请将 `nullptr` 替换为 `NULL`，这是以前起类似作用的常量。

创建指针变量

可以使用特定的 * 语法创建指针变量，例如 `int *pointer_to_int;`

当变量是 指针类型 (`int *`) 时，它仅包含一个内存地址。该内存地址是存储底层类型 (`int`) 数据的位置。

比较变量的大小与指向相同类型的指针的大小时，区别很明显：

```
// 声明一个结构体类型 `big_struct`，包含
// 三个 long long int 类型的成员。
typedef struct {
    long long int foo1;
    long long int foo2;
    long long int foo3;
} big_struct;

// 创建一个类型为 `big_struct` 的变量 `bar`
```

Chapter 30: Pointers

A pointer is an address that refers to a location in memory. They're commonly used to allow functions or data structures to know of and modify memory without having to copy the memory referred to. Pointers are usable with both primitive (built-in) or user-defined types.

Pointers make use of the "dereference" * , "address of" & , and "arrow" -> operators. The '*' and '->' operators are used to access the memory being pointed at, and the & operator is used to get an address in memory.

Section 30.1: Pointer Operations

There are two operators for pointers: Address-of operator (&): Returns the memory address of its operand. Contents-of (Dereference) operator(*): Returns the value of the variable located at the address specified by its operator.

```
int var = 20;
int *ptr;
ptr = &var;

cout << var << endl;
//Outputs 20 (The value of var)

cout << ptr << endl;
//Outputs 0x234f119 (var's memory location)

cout << *ptr << endl;
//Outputs 20(The value of the variable stored in the pointer ptr)
```

The asterisk (*) is used in declaring a pointer for simple purpose of indicating that it is a pointer. Don't confuse this with the **dereference** operator, which is used to obtain the value located at the specified address. They are simply two different things represented with the same sign.

Section 30.2: Pointer basics

Version < C++11

Note: in all the following, the existence of the C++11 constant `nullptr` is assumed. For earlier versions, replace `nullptr` with `NULL`, the constant that used to play a similar role.

Creating a pointer variable

A pointer variable can be created using the specific * syntax, e.g. `int *pointer_to_int;`

When a variable is of a *pointer type* (`int *`), it just contains a memory address. The memory address is the location at which data of the *underlying type* (`int`) is stored.

The difference is clear when comparing the size of a variable with the size of a pointer to the same type:

```
// Declare a struct type `big_struct` that contains
// three long long ints.
typedef struct {
    long long int foo1;
    long long int foo2;
    long long int foo3;
} big_struct;

// Create a variable `bar` of type `big_struct`
```

```

big_struct bar;
// 创建一个类型为指向 `big_struct` 的指针变量 `p_bar`。
// 将其初始化为 `nullptr` (空指针)。
big_struct *p_bar0 = nullptr;

// 输出 `bar` 的大小
std::cout << "sizeof(bar) = " << sizeof(bar) << std::endl;
// 输出 `p_bar` 的大小。
std::cout << "sizeof(p_bar0) = " << sizeof(p_bar0) << std::endl;

/* 输出结果：
sizeof(bar) = 24
sizeof(p_bar0) = 8
*/

```

获取另一个变量的地址

指针之间可以像普通变量一样赋值；在这种情况下，**内存地址**是从一个指针复制到另一个指针的，而不是指针所指向的实际数据。此外，指针可以取值为nullptr，表示空内存位置。等于nullptr的指针包含无效的内存地址，因此不指向有效数据。

你可以通过在变量前加上取地址操作符来获取某类型变量的内存地址`&`。操作符`&`返回的值是指向该类型的指针，包含变量的内存地址（只要变量未超出作用域，该地址即为有效数据）。

```

// 将 `p_bar0` 复制到 `p_bar_1`。
big_struct *p_bar1 = p_bar0;

// 将 `bar` 的地址赋给 `p_bar_2`
big_struct *p_bar2 = &bar;

// 现在 p_bar1 是 nullptr, p_bar2 是 &bar.

p_bar0 = p_bar2;

// 现在 p_bar0 是 &bar.

p_bar2 = nullptr;

// p_bar0 == &bar
// p_bar1 == nullptr
// p_bar2 == nullptr

```

与引用相比：

- 给两个指针赋值不会覆盖被赋值指针所指向的内存；
- 指针可以为 null。
- 必须显式使用取地址符操作符。

访问指针所指内容

由于取地址需要使用`&`，访问内容同样需要使用解引用操作符`*`，作为前缀。当指针被解引用时，它变成底层类型的变量（实际上是对该变量的引用）。如果不是`const`，则可以读取和修改它。

```

(*p_bar0).foo1 = 5;

// `p_bar0` 指向 `bar`。这将打印 5。

```

```

big_struct bar;
// Create a variable `p_bar` of type `pointer to big_struct`.
// Initialize it to `nullptr` (a null pointer).
big_struct *p_bar0 = nullptr;

// Print the size of `bar`
std::cout << "sizeof(bar) = " << sizeof(bar) << std::endl;
// Print the size of `p_bar`.
std::cout << "sizeof(p_bar0) = " << sizeof(p_bar0) << std::endl;

/* Produces:
sizeof(bar) = 24
sizeof(p_bar0) = 8
*/

```

Taking the address of another variable

Pointers can be assigned between each other just as normal variables; in this case, it is the **memory address** that is copied from one pointer to another, **not the actual data** that a pointer points to. Moreover, they can take the value `nullptr` which represents a null memory location. A pointer equal to `nullptr` contains an invalid memory location and hence it does not refer to valid data.

You can get the memory address of a variable of a given type by prefixing the variable with the *address of operator*`&`. The value returned by `&` is a pointer to the underlying type which contains the memory address of the variable (which is valid data **as long as the variable does not go out of scope**).

```

// Copy `p_bar0` into `p_bar_1`.
big_struct *p_bar1 = p_bar0;

// Take the address of `bar` into `p_bar_2`
big_struct *p_bar2 = &bar;

// p_bar1 is now nullptr, p_bar2 is &bar.

p_bar0 = p_bar2;

// p_bar0 is now &bar.

p_bar2 = nullptr;

// p_bar0 == &bar
// p_bar1 == nullptr
// p_bar2 == nullptr

```

In contrast with references:

- assigning two pointers does not overwrite the memory that the assigned pointer refers to;
- pointers can be null.
- the *address of operator* is required explicitly.

Accessing the content of a pointer

As taking an address requires `&`，well accessing the content requires the usage of the *dereference operator*`*`, as a prefix. When a pointer is dereferenced, it becomes a variable of the underlying type (actually, a reference to it). It can then be read and modified, if not `const`.

```

(*p_bar0).foo1 = 5;

// `p_bar0` points to `bar`。This prints 5.

```

```

std::cout << "bar.foo1 = " << bar.foo1 << std::endl;

// 将 `p_bar0` 指向的值赋给 `baz`。
big_struct baz;
baz = *p_bar0;

// 现在 `baz` 包含了 `p_bar0` 指向数据的副本。
// 确实，它包含了 `bar` 的副本。

// 也打印 5
std::cout << "baz.foo1 = " << baz.foo1 << std::endl;

```

操作符 * 和 . 的组合简写为 ->：

```

std::cout << "bar.foo1 = " << (*p_bar0).foo1 << std::endl; // 打印 5
std::cout << "bar.foo1 = " << p_bar0->foo1 << std::endl; // 打印 5

```

取消引用无效指针

在取消引用无效指针时，应确保它指向有效数据。取消引用无效指针（或空指针）可能导致内存访问违规，或读取或写入垃圾数据。

```

big_struct *never_do_this() {
    // 这是一个局部变量。在 `never_do_this` 之外它不存在。
    big_struct retval;
    retval.foo1 = 11;
    // 这将返回 `retval` 的地址。
    return &retval;
    // `retval` 被销毁，任何使用 `never_do_this` 返回值的代码// 都持有一个指向包
    // 含垃圾数据（或不可访问）内存位置的指针。
}

}

```

在这种情况下，g++ 和 clang++ 会正确发出警告：

```

(Clang) 警告: 返回了与局部变量 'retval' 关联的栈内存地址 [-Wreturn-stack-address]
(Gcc) 警告: 返回了局部变量 'retval' 的地址 [-Wreturn-local-addr]

```

因此，当指针作为函数参数时必须小心，因为它们可能为空指针：

```

void naive_code(big_struct *ptr_big_struct) {
    // ... 一些没有检查 `ptr_big_struct` 是否有效的代码。
    ptr_big_struct->foo1 = 12;
}

// 段错误。
naive_code(nullptr);

```

第30.3节：指针运算

递增 / 递减

指针可以递增或递减（前缀和后缀）。递增指针会使指针值前进到数组中当前指向元素的下一个元素。递减指针会使其移动到数组中的前一个元素。

```

std::cout << "bar.foo1 = " << bar.foo1 << std::endl;

// Assign the value pointed to by `p_bar0` to `baz`.
big_struct baz;
baz = *p_bar0;

// Now `baz` contains a copy of the data pointed to by `p_bar0`.
// Indeed, it contains a copy of `bar`.

// Prints 5 as well
std::cout << "baz.foo1 = " << baz.foo1 << std::endl;

```

The combination of * and the operator . is abbreviated by ->:

```

std::cout << "bar.foo1 = " << (*p_bar0).foo1 << std::endl; // Prints 5
std::cout << "bar.foo1 = " << p_bar0->foo1 << std::endl; // Prints 5

```

Dereferencing invalid pointers

When dereferencing a pointer, you should make sure it points to valid data. Dereferencing an invalid pointer (or a null pointer) can lead to memory access violation, or to read or write garbage data.

```

big_struct *never_do_this() {
    // This is a local variable. Outside `never_do_this` it doesn't exist.
    big_struct retval;
    retval.foo1 = 11;
    // This returns the address of `retval`.
    return &retval;
    // `retval` is destroyed and any code using the value returned
    // by `never_do_this` has a pointer to a memory location that
    // contains garbage data (or is inaccessible).
}

```

In such scenario, g++ and clang++ correctly issue the warnings:

```

(Clang) warning: address of stack memory associated with local variable 'retval' returned [-Wreturn-stack-address]
(Gcc)   warning: address of local variable 'retval' returned [-Wreturn-local-addr]

```

Hence, care must be taken when pointers are arguments of functions, as they could be null:

```

void naive_code(big_struct *ptr_big_struct) {
    // ... some code which doesn't check if `ptr_big_struct` is valid.
    ptr_big_struct->foo1 = 12;
}

// Segmentation fault.
naive_code(nullptr);

```

Section 30.3: Pointer Arithmetic

Increment / Decrement

A pointer can be incremented or decremented (prefix and postfix). Incrementing a pointer advances the pointer value to the element in the array one element past the currently pointed to element. Decrementing a pointer moves it to the previous element in the array.

如果指针所指向的类型不完整，则不允许进行指针运算。void类型始终是不完整类型。

```
char* str = new char[10]; // str = 0x010
++str; // str = 0x011 在此情况下 sizeof(char) = 1 字节

int* arr = new int[10]; // arr = 0x00100
++arr; // arr = 0x00104 如果 sizeof(int) = 4 字节

void* ptr = (void*)new char[10];
++ptr; // void 是不完整类型。
```

如果对指向末尾元素的指针进行递增，则该指针指向数组末尾之后的一个元素。

这样的指针不能被解引用，但可以被递减。

对数组中末尾之后的元素指针进行递增，或对数组第一个元素指针进行递减，会导致未定义行为。

对于非数组对象的指针，在指针运算时可以将其视为大小为1的数组。

加法 / 减法

可以将整数值加到指针上；它们表现为递增，但递增的数量是特定的数字，而不是1。

整数值也可以从指针中减去，起到指针递减的作用。与递增/递减一样，指针必须指向完整类型。

```
char* str = new char[10]; // str = 0x010
str += 2; // str = 0x010 + 2 * sizeof(char) = 0x012

int* arr = new int[10]; // arr = 0x100
arr += 2; // arr = 0x100 + 2 * sizeof(int) = 0x108, assuming sizeof(int) == 4.
```

指针差值

可以计算两个指向相同类型的指针之间的差值。两个指针必须位于同一个数组对象内；否则结果未定义。

给定同一数组中的两个指针P和Q，如果P是数组中的第i个元素，Q是第j个元素，则 $P - Q$ 应为 $i - j$ 。结果的类型是std::ptrdiff_t，来自<cstddef>。

```
char* start = new char[10]; // str = 0x010
char* test = &start[5];
std::ptrdiff_t diff = test - start; // 等于5。
std::ptrdiff_t diff = start - test; // 等于-5；ptrdiff_t是有符号类型。
```

Pointer arithmetic is not permitted if the type that the pointer points to is not complete. void is always an incomplete type.

```
char* str = new char[10]; // str = 0x010
++str; // str = 0x011 in this case sizeof(char) = 1 byte

int* arr = new int[10]; // arr = 0x00100
++arr; // arr = 0x00104 if sizeof(int) = 4 bytes

void* ptr = (void*)new char[10];
++ptr; // void is incomplete.
```

If a pointer to the end element is incremented, then the pointer points to one element past the end of the array. Such a pointer cannot be dereferenced, but it can be decremented.

Incrementing a pointer to the one-past-the-end element in the array, or decrementing a pointer to the first element in an array yields undefined behavior.

A pointer to a non-array object can be treated, for the purposes of pointer arithmetic, as though it were an array of size 1.

Addition / Subtraction

Integer values can be added to pointers; they act as incrementing, but by a specific number rather than by 1. Integer values can be subtracted from pointers as well, acting as pointer decrementing. As with incrementing/decrementing, the pointer must point to a complete type.

```
char* str = new char[10]; // str = 0x010
str += 2; // str = 0x010 + 2 * sizeof(char) = 0x012

int* arr = new int[10]; // arr = 0x100
arr += 2; // arr = 0x100 + 2 * sizeof(int) = 0x108, assuming sizeof(int) == 4.
```

Pointer Differencing

The difference between two pointers to the same type can be computed. The two pointers must be within the same array object; otherwise undefined behavior results.

Given two pointers P and Q in the same array, if P is the i th element in the array, and Q is the j th element, then $P - Q$ shall be $i - j$. The type of the result is std::ptrdiff_t, from <cstddef>.

```
char* start = new char[10]; // str = 0x010
char* test = &start[5];
std::ptrdiff_t diff = test - start; // Equal to 5.
std::ptrdiff_t diff = start - test; // Equal to -5; ptrdiff_t is signed.
```

第31章：成员指针

第31.1节：指向静态成员函数的指针

静态成员函数就像普通的C/C++函数，只是带有作用域：

- 它在一个类中，因此需要用类名修饰它的名称；
- 它有访问权限，可能是public、protected或private。

所以，如果你可以访问该静态成员函数并正确修饰它，那么你可以像指向类外的普通函数一样指向该函数：

```
typedef int Fn(int); // Fn是一个接受int并返回int的函数类型

// 注意MyFn()的类型是'Fn'
int MyFn(int i) { return 2*i; }

class Class {
public:
    // 注意Static()的类型是'Fn'
    static int Static(int i) { return 3*i; }
}; // Class

int main() {
    Fn *fn; // fn是指向Fn类型的指针

    fn = &MyFn; // 指向一个函数
    fn(3); // 调用它
    fn = &Class::Static; // 指向另一个函数
    fn(4); // 调用它
} // main()
```

第31.2节：指向成员函数的指针

要访问类的成员函数，您需要拥有该特定实例的“句柄”，可以是实例本身，或者是指向它的指针或引用。给定一个类实例，如果语法正确，您可以使用指向成员的指针指向它的各种成员！当然，指针必须声明为与所指向的类型相同...

```
typedef int Fn(int); // Fn 是一种接受 int 并返回 int 的函数类型

class Class {
public:
    // 注意 A() 的类型是 'Fn'
    int A(int a) { return 2*a; }
    // 注意 B() 的类型是 'Fn'
    int B(int b) { return 3*b; }
}; // Class

int main() {
    Class c; // 需要一个 Class 实例来操作
    Class *p = &c; // 需要一个 Class 指针来操作

    Fn Class::*fn; // fn 是指向 Class 中 Fn 类型的指针

    fn = &Class::A; // fn 现在指向任何 Class 中的 A 函数
    (c.*fn)(5); // 通过 fn 向 c 的函数 A 传递参数 5
```

Chapter 31: Pointers to members

Section 31.1: Pointers to static member functions

A static member function is just like an ordinary C/C++ function, except with scope:

- It is inside a class, so it needs its name decorated with the class name;
- It has accessibility, with public, protected or private.

So, if you have access to the static member function and decorate it correctly, then you can point to the function like any normal function outside a class:

```
typedef int Fn(int); // Fn is a type-of function that accepts an int and returns an int

// Note that MyFn() is of type 'Fn'
int MyFn(int i) { return 2*i; }

class Class {
public:
    // Note that Static() is of type 'Fn'
    static int Static(int i) { return 3*i; }
}; // Class

int main() {
    Fn *fn; // fn is a pointer to a type-of Fn

    fn = &MyFn; // Point to one function
    fn(3); // Call it
    fn = &Class::Static; // Point to the other function
    fn(4); // Call it
} // main()
```

Section 31.2: Pointers to member functions

To access a member function of a class, you need to have a "handle" to the particular instance, as either the instance itself, or a pointer or reference to it. Given a class instance, you can point to various of its members with a pointer-to-member, IF you get the syntax correct! Of course, the pointer has to be declared to be of the same type as what you are pointing to...

```
typedef int Fn(int); // Fn is a type-of function that accepts an int and returns an int

class Class {
public:
    // Note that A() is of type 'Fn'
    int A(int a) { return 2*a; }
    // Note that B() is of type 'Fn'
    int B(int b) { return 3*b; }
}; // Class

int main() {
    Class c; // Need a Class instance to play with
    Class *p = &c; // Need a Class pointer to play with

    Fn Class::*fn; // fn is a pointer to a type-of Fn within Class

    fn = &Class::A; // fn now points to A within any Class
    (c.*fn)(5); // Pass 5 to c's function A (via fn)
```

```

fn = &Class::B; // fn 现在指向任何 Class 中的 B 函数
(p->*fn)(6); // 通过 p 调用 c 的函数 B (通过 fn) 传递参数 6
} // main()

```

与成员变量指针（前面的例子）不同，类实例与成员指针之间的关联必须用括号紧密绑定，这看起来有点奇怪（仿佛 `.*` 和 `->*` 还不够奇怪！）

第31.3节：指向成员变量的指针

要访问类的成员，您需要拥有该特定实例的“句柄”，可以是实例本身，或者是指向它的指针或引用。给定一个类实例，如果语法正确，您可以使用指向成员的指针指向它的各种成员！当然，指针必须声明为与所指向的类型相同...

```

class 类 {
public:
    int x, y, z;
    char m, n, o;
} // Class

int x; // 全局变量

int main() {
类 c; // 需要一个类实例来操作
    类 *p = &c; // 需要一个类指针来操作

    int *p_i; // 指向int的指针

p_i = &x; // 现在指向x
    p_i = &c.x; // 现在指向c的x

    int 类::*p_C_i; // 指向类中int的指针

p_C_i = &类::x; // 指向任何类中的x
    int i = c.*p_C_i; // 使用p_C_i从c的实例中获取x
    p_C_i = &类::y; // 指向任何类中的y
    i = c.*p_C_i; // 使用p_C_i从c的实例中获取y

    p_C_i = &类::m; // 错误！m是char类型，不是int！

    char 类::*p_C_c = &类::m; // 这样更好...
} // main()

```

指向成员的指针语法需要一些额外的语法元素：

- 要定义指针的类型，需要提及基类型，以及它是在一个类中：`int Class::*ptr;`。
- 如果你有一个类或引用，并想用它与成员指针一起使用，需要使用`.*`操作符（类似于`.操作符`）。
- 如果你有一个指向类的指针，并想用它与成员指针一起使用，需要使用`->*`操作符（类似于`->`操作符）。

第31.4节：指向静态成员变量的指针

`static` 成员变量就像普通的C/C++变量，只是作用域不同：

```

fn = &Class::B; // fn now points to B within any Class
(p->*fn)(6); // Pass 6 to c's (via p) function B (via fn)
} // main()

```

Unlike pointers to member variables (in the previous example), the association between the class instance and the member pointer need to be bound tightly together with parentheses, which looks a little strange (as though the `.*` and `->*` aren't strange enough!)

Section 31.3: Pointers to member variables

To access a member of a `class`, you need to have a "handle" to the particular instance, as either the instance itself, or a pointer or reference to it. Given a `class` instance, you can point to various of its members with a pointer-to-member, IF you get the syntax correct! Of course, the pointer has to be declared to be of the same type as what you are pointing to...

```

class Class {
public:
    int x, y, z;
    char m, n, o;
} // Class

int x; // Global variable

int main() {
    Class c; // Need a Class instance to play with
    Class *p = &c; // Need a Class pointer to play with

    int *p_i; // Pointer to an int

    p_i = &x; // Now pointing to x
    p_i = &c.x; // Now pointing to c's x

    int Class::*p_C_i; // Pointer to an int within Class

    p_C_i = &Class::x; // Point to x within any Class
    int i = c.*p_C_i; // Use p_C_i to fetch x from c's instance
    p_C_i = &Class::y; // Point to y within any Class
    i = c.*p_C_i; // Use p_C_i to fetch y from c's instance

    p_C_i = &Class::m; // ERROR! m is a char, not an int!

    char Class::*p_C_c = &Class::m; // That's better...
} // main()

```

The syntax of pointer-to-member requires some extra syntactic elements:

- To define the type of the pointer, you need to mention the base type, as well as the fact that it is inside a class: `int Class::*ptr;`.
- If you have a class or reference and want to use it with a pointer-to-member, you need to use the `.*` operator (akin to the `.` operator).
- If you have a pointer to a class and want to use it with a pointer-to-member, you need to use the `->*` operator (akin to the `->` operator).

Section 31.4: Pointers to static member variables

A `static` member variable is just like an ordinary C/C++ variable, except with scope:

- 它在一个类中，因此需要用类名修饰它的名称；
- 它有访问权限，可能是public、protected或private。

因此，如果你可以访问static成员变量并正确修饰它，那么你可以像指向普通变量一样指向该变量，且该变量位于class外部：

```
class 类 {  
public:  
    static int i;  
} // Class  
  
int Class::i = 1; // 定义i的值（以及它存储的位置！）  
  
int j = 2; // 另一个全局变量  
  
int main() {  
    int k = 3; // 局部变量  
  
    int *p;  
  
    p = &k; // 指向 k  
    *p = 2; // 修改它  
    p = &j; // 指向 j  
    *p = 3; // 修改它  
    p = &Class::i; // 指向 Class::i  
    *p = 4; // 修改它  
} // main()
```

- It is inside a `class`, so it needs its name decorated with the class name;
- It has accessibility, with `public`, `protected` or `private`.

So, if you have access to the `static` member variable and decorate it correctly, then you can point to the variable like any normal variable outside a `class`:

```
class Class {  
public:  
    static int i;  
} // Class  
  
int Class::i = 1; // Define the value of i (and where it's stored!)  
  
int j = 2; // Just another global variable  
  
int main() {  
    int k = 3; // Local variable  
  
    int *p;  
  
    p = &k; // Point to k  
    *p = 2; // Modify it  
    p = &j; // Point to j  
    *p = 3; // Modify it  
    p = &Class::i; // Point to Class::i  
    *p = 4; // Modify it  
} // main()
```

第32章：this 指针

第32.1节：this 指针

所有非静态成员函数都有一个隐藏参数，即指向类实例的指针，名为 `this`；该参数会被编译器自动插入到参数列表的开头，并由编译器完全处理。当在成员函数内部访问类的成员时，会通过 `this` 隐式访问；这使得编译器可以为所有实例使用单个非静态成员函数，并允许成员函数以多态方式调用其他成员函数。

```
struct ThisPointer {
    int i;

ThisPointer(int ii);
    virtual void func();
    int get_i() const;
    void set_i(int ii);
};

ThisPointer::ThisPointer(int ii) : i(ii) {}
// 编译器重写为：
ThisPointer::ThisPointer(int ii) : this->i(ii) {}
// 构造函数负责将分配的内存转换为 'this'.
// 由于构造函数负责创建对象，'this'在实例完全构造之前不会“完全”有效。

/* virtual */ void ThisPointer::func() {
    if (some_external_condition) {
        set_i(182);
    } else {
        i = 218;
    }
}
// 编译器重写为：
/* virtual */ void ThisPointer::func(ThisPointer* this) {
    if (some_external_condition) {
        this->set_i(182);
    } else {
        this->i = 218;
    }
}

int ThisPointer::get_i() const { return i; }
// 编译器重写为：
int ThisPointer::get_i(const ThisPointer* this) { return this->i; }

void ThisPointer::set_i(int ii) { i = ii; }
// 编译器重写为：
void ThisPointer::set_i(ThisPointer* this, int ii) { this->i = ii; }
```

在构造函数中，`this` 可以安全地（隐式或显式地）访问任何已经初始化的字段，或父类中的任何字段；相反，访问任何尚未初始化的字段，或派生类中的任何字段（无论是隐式还是显式）都是不安全的（因为派生类尚未构造，因此其字段既未初始化也不存在）。在构造函数中通过`this`调用虚函数也是不安全的，因为任何派生类的函数都不会被考虑（由于派生类尚未构造，因此其构造函数尚未更新虚函数表）。

Chapter 32: The This Pointer

Section 32.1: this Pointer

All non-static member functions have a hidden parameter, a pointer to an instance of the class, named `this`; this parameter is silently inserted at the beginning of the parameter list, and handled entirely by the compiler. When a member of the class is accessed inside a member function, it is silently accessed through `this`; this allows the compiler to use a single non-static member function for all instances, and allows a member function to call other member functions polymorphically.

```
struct ThisPointer {
    int i;

ThisPointer(int ii);
    virtual void func();
    int get_i() const;
    void set_i(int ii);
};

ThisPointer::ThisPointer(int ii) : i(ii) {}
// Compiler rewrites as:
ThisPointer::ThisPointer(int ii) : this->i(ii) {}
// Constructor is responsible for turning allocated memory into 'this'.
// As the constructor is responsible for creating the object, 'this' will not be "fully"
// valid until the instance is fully constructed.

/* virtual */ void ThisPointer::func() {
    if (some_external_condition) {
        set_i(182);
    } else {
        i = 218;
    }
}
// Compiler rewrites as:
/* virtual */ void ThisPointer::func(ThisPointer* this) {
    if (some_external_condition) {
        this->set_i(182);
    } else {
        this->i = 218;
    }
}

int ThisPointer::get_i() const { return i; }
// Compiler rewrites as:
int ThisPointer::get_i(const ThisPointer* this) { return this->i; }

void ThisPointer::set_i(int ii) { i = ii; }
// Compiler rewrites as:
void ThisPointer::set_i(ThisPointer* this, int ii) { this->i = ii; }
```

In a constructor, `this` can safely be used to (implicitly or explicitly) access any field that has already been initialised, or any field in a parent class; conversely, (implicitly or explicitly) accessing any fields that haven't yet been initialised, or any fields in a derived class, is unsafe (due to the derived class not yet being constructed, and thus its fields neither being initialised nor existing). It is also unsafe to call virtual member functions through `this` in the constructor, as any derived class functions will not be considered (due to the derived class not yet being constructed, and thus its constructor not yet updating the vtable).

还要注意，在构造函数中，对象的类型是该构造函数所构造的类型。即使对象被声明为派生类型，这一点仍然成立。例如，在下面的例子中，`ctd_good` 和 `ctd_bad` 在 `CtorThisBase()` 内部的类型是 `CtorThisBase`，在 `CtorThis()` 内部的类型是 `CtorThis`，尽管它们的规范类型是 `CtorThisDerived`。由于更派生的类是围绕基类构造的，实例会逐渐经过类层次结构，直到成为其预期类型的完全构造实例。

```
class CtorThisBase {
    short s;

public:
CtorThisBase() : s(516) {}

class CtorThis : public CtorThisBase {
    int i, j, k;

public:
    // 好的构造函数。
CtorThis() : i(s + 42), j(this->i), k(j) {}

    // 不好的构造函数。
CtorThis(int ii) : i(ii), j(this->k), k(b ? 51 : -51) {
    virt_func();
}

    virtual void virt_func() { i += 2; }

};

class CtorThisDerived : public CtorThis {
    bool b;

public:
CtorThisDerived() : b(true) {}
CtorThisDerived(int ii) : CtorThis(ii), b(false) {}

    void virt_func() override { k += (2 * i); }

// ...

CtorThisDerived ctd_good;
CtorThisDerived ctd_bad(3);
```

使用这些类和成员函数：

- 在良好构造函数中，对于`ctd_good`：
 - `CtorThisBase`在进入`CtorThis`构造函数时已完全构造完成。因此，`s`在初始化`i`时处于有效状态，因此可以访问。
 - `i`在达到`j(this->i)`之前已初始化。因此，`i`在初始化`j`时处于有效状态，因此可以访问。
 - `j`在达到`k(j)`之前已初始化。因此，`j`在初始化`k`时处于有效状态，因此可以访问。
- 在不良构造函数中，对于`ctd_bad`：
 - `k` 在达到 `j(this->k)` 后初始化。因此，`k` 在初始化 `j` 时处于无效状态，访问它会导致未定义行为。
 - `CtorThisDerived` 直到 `CtorThis` 构造完成后才构造。因此，`b` 在初始化 `k` 时处于无效状态，访问它会导致未定义行为。
 - 对象 `ctd_bad` 仍然是一个 `CtorThis`，直到它离开 `CtorThis()`，并且不会更新为使用

Also note that while in a constructor, the type of the object is the type which that constructor constructs. This holds true even if the object is declared as a derived type. For example, in the below example, `ctd_good` and `ctd_bad` are type `CtorThisBase` inside `CtorThisBase()`, and type `CtorThis` inside `CtorThis()`, even though their canonical type is `CtorThisDerived`. As the more-derived classes are constructed around the base class, the instance gradually goes through the class hierarchy until it is a fully-constructed instance of its intended type.

```
class CtorThisBase {
    short s;

public:
CtorThisBase() : s(516) {}

class CtorThis : public CtorThisBase {
    int i, j, k;

public:
    // Good constructor.
CtorThis() : i(s + 42), j(this->i), k(j) {}

    // Bad constructor.
CtorThis(int ii) : i(ii), j(this->k), k(b ? 51 : -51) {
    virt_func();
}

    virtual void virt_func() { i += 2; }

};

class CtorThisDerived : public CtorThis {
    bool b;

public:
CtorThisDerived() : b(true) {}
CtorThisDerived(int ii) : CtorThis(ii), b(false) {}

    void virt_func() override { k += (2 * i); }

// ...

CtorThisDerived ctd_good;
CtorThisDerived ctd_bad(3);
```

With these classes and member functions:

- In the good constructor, for `ctd_good`:
 - `CtorThisBase` is fully constructed by the time the `CtorThis` constructor is entered. Therefore, `s` is in a valid state while initialising `i`, and can thus be accessed.
 - `i` is initialised before `j(this->i)` is reached. Therefore, `i` is in a valid state while initialising `j`, and can thus be accessed.
 - `j` is initialised before `k(j)` is reached. Therefore, `j` is in a valid state while initialising `k`, and can thus be accessed.
- In the bad constructor, for `ctd_bad`:
 - `k` is initialised after `j(this->k)` is reached. Therefore, `k` is in an invalid state while initialising `j`, and accessing it causes undefined behaviour.
 - `CtorThisDerived` is not constructed until after `CtorThis` is constructed. Therefore, `b` is in an invalid state while initialising `k`, and accessing it causes undefined behaviour.
 - The object `ctd_bad` is still a `CtorThis` until it leaves `CtorThis()`, and will not be updated to use

CtorThisDerived 的虚函数表，直到 CtorThisDerived()。因此，virt_func() 将调用 CtorThis::virt_func()，无论是否打算调用 CtorThisDerived::virt_func()。

第32.2节：使用 this 指针访问成员数据

在这种情况下，使用 `this` 指针并非完全必要，但它会使代码对读者更清晰，表明某个函数或变量是类的成员。此情形下的示例：

```
// this 指针示例
#include <iostream>
#include <string>

using std::cout;
using std::endl;

class Class
{
public:
    Class();
    ~Class();
    int getPrivateNumber () const;
private:
    int private_number = 42;
};

Class::Class(){}
Class::~Class(){}

int Class::getPrivateNumber() const
{
    return this->private_number;
}

int main()
{
    Class class_example;
    cout << class_example.getPrivateNumber() << endl;
}
```

在这里查看实际效果。

第32.3节：使用this指针区分成员数据和参数

这是一种区分成员数据和参数的实际有效策略.....我们来看这个例子：

```
// 狗类示例
#include <iostream>
#include <string>

using std::cout;
using std::endl;

/*
 * @class 狗
 * @member 名字
 *     狗的名字
 * @function 吠叫
 */
```

CtorThisDerived's vtable until CtorThisDerived(). Therefore, virt_func() will call CtorThis::virt_func(), regardless of whether it is intended to call that or CtorThisDerived::virt_func().

Section 32.2: Using the this Pointer to Access Member Data

In this context, using the `this` pointer isn't entirely necessary, but it will make your code clearer to the reader, by indicating that a given function or variable is a member of the class. An example in this situation:

```
// Example for this pointer
#include <iostream>
#include <string>

using std::cout;
using std::endl;

class Class
{
public:
    Class();
    ~Class();
    int getPrivateNumber () const;
private:
    int private_number = 42;
};

Class::Class(){}
Class::~Class(){}

int Class::getPrivateNumber() const
{
    return this->private_number;
}

int main()
{
    Class class_example;
    cout << class_example.getPrivateNumber() << endl;
}
```

See it in action [here](#).

Section 32.3: Using the this Pointer to Differentiate Between Member Data and Parameters

This is an actual useful strategy to differentiate member data from parameters... Lets take this example :

```
// Dog Class Example
#include <iostream>
#include <string>

using std::cout;
using std::endl;

/*
 * @class Dog
 * @member name
 *     Dog's name
 * @function bark
 */
```

```

/*
* 狗在吠叫 !
* @function 获取名字
* 获取私有
* 名字变量
*/
class 狗
{
public:
    狗(std::string 名字);
    ~狗();
    void 吠叫() const;
    std::string 获取名字() const;
private:
    std::string 名字;
};

狗::狗(std::string 名字)
{
    /*
     * this->name 是类 dog 中的 name 变量。name 来自函数的参数
     */
    this->name = name;
}

Dog::~Dog(){}

void Dog::bark() const
{
    cout << "BARK" << endl;
}

std::string Dog::getName() const
{
    return this->name;
}

int main()
{
    Dog dog("Max");
    cout << dog.getName() << endl;
    dog.bark();
}

```

你可以看到在构造函数中我们执行了以下操作：

```
this->name = name;
```

这里，你可以看到我们将参数 name 赋值给类 Dog 的私有变量 name (this->name)。

要查看上述代码的输出：<http://cpp.sh/75r7>

第32.4节：this指针的CV限定符

this 也可以是 cv 限定的，和任何其他指针一样。然而，由于 this 参数未被列出

```

/*
* Dog Barks!
* @function getName
* To Get Private
* Name Variable
*/
class Dog
{
public:
    Dog(std::string name);
    ~Dog();
    void bark() const;
    std::string getName() const;
private:
    std::string name;
};

Dog::Dog(std::string name)
{
    /*
     * this->name is the
     * name variable from
     * the class dog . and
     * name is from the
     * parameter of the function
     */
    this->name = name;
}

Dog::~Dog(){}

void Dog::bark() const
{
    cout << "BARK" << endl;
}

std::string Dog::getName() const
{
    return this->name;
}

int main()
{
    Dog dog("Max");
    cout << dog.getName() << endl;
    dog.bark();
}

```

You can see here in the constructor we execute the following:

```
this->name = name;
```

Here , you can see we are assinging the parameter name to the name of the private variable from the class Dog(this->name) .

To see the output of above code : <http://cpp.sh/75r7>

Section 32.4: this Pointer CV-Qualifiers

this can also be cv-qualified, the same as any other pointer. However, due to the this parameter not being listed

在参数列表中，这需要特殊的语法；cv限定符列在参数列表之后，但在函数体之前。

```
struct ThisCVQ {
    void no_qualifier() {} // "this" is :ThisCVQ*
    void c_qualifier() const {} // "this" is :const ThisCVQ*
    void v_qualifier() volatile {} // "this" is :volatile ThisCVQ*
    void cv_qualifier() const volatile {} // "this" is :const volatile ThisCVQ*
};
```

由于这是一个参数，函数可以基于其 this cv 限定符进行重载。

```
struct CVOverload {
    int func() { return 3; }
    int func() const { return 33; }
    int func() volatile { return 333; }
    int func() const volatile { return 3333; }
};
```

当this是const（包括const volatile）时，函数无法通过它对成员变量进行写操作，无论是隐式还是显式。唯一的例外是mutable成员变量，无论是否为const，都可以被写入。正因为如此，const用于表示成员函数不会改变对象的逻辑状态（对象对外界的表达方式），即使它确实修改了物理状态（对象内部的实际表现）。

逻辑状态是对象在外部观察者看来呈现的方式。它并不直接与物理状态相关，实际上，甚至可能不会以物理状态的形式存储。只要外部观察者看不到任何变化，逻辑状态就是恒定的，即使你翻转对象中的每一个比特。

物理状态，也称为按位状态，是对象在内存中的存储方式。这是对象的细节，构成其数据的原始1和0。只有当对象在内存中的表示从未改变时，该对象才是物理上常量的。

注意，C++将const性基于逻辑状态，而非物理状态。

```
class DoSomethingComplexAndOrExpensive {
    mutable ResultType cached_result;
    mutable bool state_changed;

    ResultType calculate_result();
    void modify_somewhat(const Param& p);

    // ...

public:
    DoSomethingComplexAndOrExpensive(Param p) : state_changed(true) {
        modify_somewhat(p);
    }

    void change_state(Param p) {
        modify_somewhat(p);
        state_changed = true;
    }
}
```

in the parameter list, special syntax is required for this; the cv-qualifiers are listed after the parameter list, but before the function's body.

```
struct ThisCVQ {
    void no_qualifier() {} // "this" is :ThisCVQ*
    void c_qualifier() const {} // "this" is :const ThisCVQ*
    void v_qualifier() volatile {} // "this" is :volatile ThisCVQ*
    void cv_qualifier() const volatile {} // "this" is :const volatile ThisCVQ*
};
```

As `this` is a parameter, a function can be overloaded based on its `this` cv-qualifier(s).

```
struct CVOverload {
    int func() { return 3; }
    int func() const { return 33; }
    int func() volatile { return 333; }
    int func() const volatile { return 3333; }
};
```

When `this` is `const` (including `const volatile`), the function is unable to write to member variables through it, whether implicitly or explicitly. The sole exception to this is `mutable` member variables, which can be written regardless of `const`-ness. Due to this, `const` is used to indicate that the member function doesn't change the object's logical state (the way the object appears to the outside world), even if it does modify the physical state (the way the object looks under the hood).

Logical state is the way the object appears to outside observers. It isn't directly tied to physical state, and indeed, might not even be stored as physical state. As long as outside observers can't see any changes, the logical state is constant, even if you flip every single bit in the object.

Physical state, also known as bitwise state, is how the object is stored in memory. This is the object's nitty-gritty, the raw 1s and 0s that make up its data. An object is only physically constant if its representation in memory never changes.

Note that C++ bases `const`ness on logical state, not physical state.

```
class DoSomethingComplexAndOrExpensive {
    mutable ResultType cached_result;
    mutable bool state_changed;

    ResultType calculate_result();
    void modify_somewhat(const Param& p);

    // ...

public:
    DoSomethingComplexAndOrExpensive(Param p) : state_changed(true) {
        modify_somewhat(p);
    }

    void change_state(Param p) {
        modify_somewhat(p);
        state_changed = true;
    }
}
```

```

// 返回一些复杂和/或计算代价高的结果。
// 由于这没有修改逻辑状态的理由，因此标记为"const"。
ResultType get_result() const;
};

ResultType DoSomethingComplexAndOrExpensive::get_result() const {
    // cached_result 和 state_changed 即使在 const "this" 指针下也可以被修改。
    // 虽然该函数不修改逻辑状态，但它确实修改了物理状态
    // 通过缓存结果，避免每次调用函数时都重新计算。
    // 这由 cached_result 和 state_changed 被声明为 mutable 表示。

    if (state_changed) {
        cached_result = calculate_result();
        state_changed = false;
    }

    return cached_result;
}

```

注意，虽然技术上你可以使用 `const_cast` 对 `this` 进行非 cv 限定转换，但你真的，真的不应该这样做，应该使用 `mutable`。`const_cast` 在用于实际为 `const` 的对象时可能导致未定义行为，而 `mutable` 设计上是安全的。不过，在非常老的代码中你可能会遇到这种情况。

该规则的一个例外是用 `const` 访问器定义非 cv 限定访问器；因为如果调用非 cv 限定版本，保证对象不是 `const`，所以不存在未定义行为的风险。

```

class CVAccessor {
    int arr[5];

public:
    const int& get_arr_element(size_t i) const { return arr[i]; }

    int& get_arr_element(size_t i) {
        return const_cast<int&>(const_cast<const CVAccessor*>(this)->get_arr_element(i));
    }
};

```

这可以防止不必要的代码重复。

与普通指针一样，如果 `this` 是 `volatile`（包括 `const volatile`），则每次访问时都会从内存中加载，而不是缓存。这对优化的影响与将任何其他指针声明为 `volatile` 相同，因此应当小心。

请注意，如果一个实例是 cv 限定的，它只能访问那些 `this` 指针至少与实例本身具有相同 cv 限定的成员函数：

- 非 cv 实例可以访问任何成员函数。
- `const` 实例可以访问 `const` 和 `const volatile` 函数。
- `volatile` 实例可以访问 `volatile` 和 `const volatile` 函数。
- `const volatile` 实例可以访问 `const volatile` 函数。

这是 `const` 正确性的关键原则之一。

```

struct CVAcces {
    void func() {}
    void func_c() const {}
    void func_v() volatile {}
    void func_cv() const volatile {}
};

```

```

// Return some complex and/or expensive-to-calculate result.
// As this has no reason to modify logical state, it is marked as "const".
ResultType get_result() const;
};

ResultType DoSomethingComplexAndOrExpensive::get_result() const {
    // cached_result and state_changed can be modified, even with a const "this" pointer.
    // Even though the function doesn't modify logical state, it does modify physical state
    // by caching the result, so it doesn't need to be recalculated every time the function
    // is called. This is indicated by cached_result and state_changed being mutable.

    if (state_changed) {
        cached_result = calculate_result();
        state_changed = false;
    }

    return cached_result;
}

```

Note that while you technically *could* use `const_cast` on `this` to make it non-cv-qualified, you really, **REALLY** shouldn't, and should use `mutable` instead. A `const_cast` is liable to invoke undefined behaviour when used on an object that actually *is* `const`, while `mutable` is designed to be safe to use. It is, however, possible that you may run into this in extremely old code.

An exception to this rule is defining non-cv-qualified accessors in terms of `const` accessors; as the object is guaranteed to not be `const` if the non-cv-qualified version is called, there's no risk of UB.

```

class CVAccessor {
    int arr[5];

public:
    const int& get_arr_element(size_t i) const { return arr[i]; }

    int& get_arr_element(size_t i) {
        return const_cast<int&>(const_cast<const CVAccessor*>(this)->get_arr_element(i));
    }
};

```

This prevents unnecessary duplication of code.

As with regular pointers, if `this` is `volatile` (including `const volatile`)，it is loaded from memory each time it is accessed, instead of being cached. This has the same effects on optimisation as declaring any other pointer `volatile` would, so care should be taken.

Note that if an instance is cv-qualified, the only member functions it is allowed to access are member functions whose `this` pointer is at least as cv-qualified as the instance itself:

- Non-cv instances can access any member functions.
- `const` instances can access `const` and `const volatile` functions.
- `volatile` instances can access `volatile` and `const volatile` functions.
- `const volatile` instances can access `const volatile` functions.

This is one of the key tenets of `const` correctness.

```

struct CVAcces {
    void func() {}
    void func_c() const {}
    void func_v() volatile {}
    void func_cv() const volatile {}
};

```

```
};
```

```
CVAcces cva;  
cva.func(); // 正确。  
cva.func_c(); // 好的。  
cva.func_v(); // 好的。  
cva.func_cv(); // 好的。
```

```
const CVAcces c_cva;  
c_cva.func(); // 错误。  
c_cva.func_c(); // 好的。  
c_cva.func_v(); // 错误。  
c_cva.func_cv(); // 好的。
```

```
volatile CVAcces v_cva;  
v_cva.func(); // 错误。  
v_cva.func_c(); // 错误。  
v_cva.func_v(); // 好的。  
v_cva.func_cv(); // 好的。
```

```
const volatile CVAcces cv_cva;  
cv_cva.func(); // 错误。  
cv_cva.func_c(); // 错误。  
cv_cva.func_v(); // 错误。  
cv_cva.func_cv(); // 正确。
```

第32.5节：this指针的引用限定符

版本 ≥ C++11

类似于this的cv限定符，我们也可以将ref-qualifiers应用于*this。引用限定符用于选择普通引用语义和右值引用语义，允许编译器根据更合适的情况使用拷贝或移动语义，并且应用于*this而非this。

注意，尽管引用限定符使用引用语法，this本身仍然是一个指针。还要注意，引用限定符实际上并不改变*this的类型；只是通过将它们视为改变了类型，更容易描述和理解它们的效果。

```
struct RefQualifiers {  
    std::string s;  
  
    RefQualifiers(const std::string& ss = "无名者.") : s(ss) {}  
  
    // 普通版本。  
    void func() & { std::cout << "访问普通实例 " << s << std::endl; }  
    // 右值版本。  
    void func() && { std::cout << "访问临时实例 " << s << std::endl; }  
  
    const std::string& still_a_pointer() & { return this->s; }  
    const std::string& still_a_pointer() && { this->s = "Bob"; return this->s; }  
};  
  
// ...  
  
RefQualifiers rf("Fred");  
rf.func(); // 输出: 访问普通实例 Fred  
RefQualifiers{}.func(); // 输出: 访问临时实例 无名者
```

成员函数不能同时有带有和不带有引用限定符的重载；程序员必须在两者之间选择

```
};
```

```
CVAcces cva;  
cva.func(); // Good.  
cva.func_c(); // Good.  
cva.func_v(); // Good.  
cva.func_cv(); // Good.
```

```
const CVAcces c_cva;  
c_cva.func(); // Error.  
c_cva.func_c(); // Good.  
c_cva.func_v(); // Error.  
c_cva.func_cv(); // Good.
```

```
volatile CVAcces v_cva;  
v_cva.func(); // Error.  
v_cva.func_c(); // Error.  
v_cva.func_v(); // Good.  
v_cva.func_cv(); // Good.
```

```
const volatile CVAcces cv_cva;  
cv_cva.func(); // Error.  
cv_cva.func_c(); // Error.  
cv_cva.func_v(); // Error.  
cv_cva.func_cv(); // Good.
```

Section 32.5: this Pointer Ref-Qualifiers

Version ≥ C++11

Similarly to `this` cv-qualifiers, we can also apply *ref-qualifiers* to `*this`. Ref-qualifiers are used to choose between normal and rvalue reference semantics, allowing the compiler to use either copy or move semantics depending on which are more appropriate, and are applied to `*this` instead of `this`.

Note that despite ref-qualifiers using reference syntax, `this` itself is still a pointer. Also note that ref-qualifiers don't actually change the type of `*this`; it's just easier to describe and understand their effects by looking at them as if they did.

```
struct RefQualifiers {  
    std::string s;  
  
    RefQualifiers(const std::string& ss = "The nameless one.") : s(ss) {}  
  
    // Normal version.  
    void func() & { std::cout << "Accessed on normal instance " << s << std::endl; }  
    // Rvalue version.  
    void func() && { std::cout << "Accessed on temporary instance " << s << std::endl; }  
  
    const std::string& still_a_pointer() & { return this->s; }  
    const std::string& still_a_pointer() && { this->s = "Bob"; return this->s; }  
};  
  
// ...  
  
RefQualifiers rf("Fred");  
rf.func(); // Output: Accessed on normal instance Fred  
RefQualifiers{}.func(); // Output: Accessed on temporary instance The nameless one
```

A member function cannot have overloads both with and without ref-qualifiers; the programmer has to choose

其中之一。值得庆幸的是，cv限定符可以与引用限定符一起使用，从而允许`const`正确性规则得以遵守。

```
struct RefCV {  
    void func() & {}  
    void func() && {}  
    void func() const& {}  
    void func() const&& {}  
    void func() volatile& {}  
    void func() volatile&& {}  
    void func() const volatile& {}  
    void func() const volatile&& {}  
};
```

between one or the other. Thankfully, cv-qualifiers can be used in conjunction with ref-qualifiers, allowing `const` correctness rules to be followed.

```
struct RefCV {  
    void func() & {}  
    void func() && {}  
    void func() const& {}  
    void func() const&& {}  
    void func() volatile& {}  
    void func() volatile&& {}  
    void func() const volatile& {}  
    void func() const volatile&& {}  
};
```

第33章：智能指针

第33.1节：唯一所有权 (std::unique_ptr)

版本 ≥ C++11

`std::unique_ptr` 是一个类模板，用于管理动态存储对象的生命周期。与 `std::shared_ptr` 不同，动态对象在任何时候只被一个 `std::unique_ptr` 实例拥有，

```
// 创建一个值为20的动态int，由唯一指针拥有  
std::unique_ptr<int> ptr = std::make_unique<int>(20);
```

(注：`std::unique_ptr` 自C++11起可用，`std::make_unique` 自C++14起可用。) 只有变

量 `ptr` 持有指向动态分配的 `int` 的指针。当拥有对象的唯一指针超出作用域时，所拥有的对象将被删除，即如果对象是类类型，则调用其析构函数，并释放该对象的内存。

要对数组类型使用 `std::unique_ptr` 和 `std::make_unique`，请使用它们的数组特化版本：

```
// 创建一个值为59的int的unique_ptr  
std::unique_ptr<int> ptr = std::make_unique<int>(59);
```

```
// 创建一个包含15个int的数组的unique_ptr  
std::unique_ptr<int[]> ptr = std::make_unique<int[]>(15);
```

你可以像使用原始指针一样访问 `std::unique_ptr`，因为它重载了相关操作符。

你可以通过使用 `std::move` 将智能指针内容的所有权转移给另一个指针，这将导致原智能指针指向 `nullptr`。

```
// 1. std::unique_ptr  
std::unique_ptr<int> ptr = std::make_unique<int>();  
  
// 将值改为1  
*ptr = 1;  
  
// 2. std::unique_ptr (通过将 'ptr' 移动到 'ptr2'， 'ptr' 不再拥有该对象)  
std::unique_ptr<int> ptr2 = std::move(ptr);  
  
int a = *ptr2; // 'a' 的值是1  
int b = *ptr; // 未定义行为！'ptr' 是 'nullptr'  
// (因为上面的移动命令)
```

将 `unique_ptr` 作为参数传递给函数：

```
void foo(std::unique_ptr<int> ptr)  
{  
    // 你的代码写在这里  
}  
  
std::unique_ptr<int> ptr = std::make_unique<int>(59);  
foo(std::move(ptr))
```

从函数返回 `unique_ptr`。这是编写工厂函数的首选C++11方式，因为它清晰地

Chapter 33: Smart Pointers

Section 33.1: Unique ownership (std::unique_ptr)

Version ≥ C++11

A `std::unique_ptr` is a class template that manages the lifetime of a dynamically stored object. Unlike for `std::shared_ptr`, the dynamic object is owned by only one instance of a `std::unique_ptr` at any time,

```
// Creates a dynamic int with value of 20 owned by a unique pointer  
std::unique_ptr<int> ptr = std::make_unique<int>(20);
```

(Note: `std::unique_ptr` is available since C++11 and `std::make_unique` since C++14.)

Only the variable `ptr` holds a pointer to a dynamically allocated `int`. When a unique pointer that owns an object goes out of scope, the owned object is deleted, i.e. its destructor is called if the object is of class type, and the memory for that object is released.

To use `std::unique_ptr` and `std::make_unique` with array-types, use their array specializations:

```
// Creates a unique_ptr to an int with value 59  
std::unique_ptr<int> ptr = std::make_unique<int>(59);
```

```
// Creates a unique_ptr to an array of 15 ints  
std::unique_ptr<int[]> ptr = std::make_unique<int[]>(15);
```

You can access the `std::unique_ptr` just like a raw pointer, because it overloads those operators.

You can transfer ownership of the contents of a smart pointer to another pointer by using `std::move`, which will cause the original smart pointer to point to `nullptr`.

```
// 1. std::unique_ptr  
std::unique_ptr<int> ptr = std::make_unique<int>();  
  
// Change value to 1  
*ptr = 1;  
  
// 2. std::unique_ptr (by moving 'ptr' to 'ptr2', 'ptr' doesn't own the object anymore)  
std::unique_ptr<int> ptr2 = std::move(ptr);  
  
int a = *ptr2; // 'a' is 1  
int b = *ptr; // undefined behavior! 'ptr' is 'nullptr'  
// (because of the move command above)
```

Passing `unique_ptr` to functions as parameter:

```
void foo(std::unique_ptr<int> ptr)  
{  
    // Your code goes here  
}  
  
std::unique_ptr<int> ptr = std::make_unique<int>(59);  
foo(std::move(ptr))
```

Returning `unique_ptr` from functions. This is the preferred C++11 way of writing factory functions, as it clearly

传达返回值的所有权语义：调用者拥有返回的unique_ptr，并负责其生命周期。

```
std::unique_ptr<int> foo()
{
    std::unique_ptr<int> ptr = std::make_unique<int>(59);
    return ptr;
}

std::unique_ptr<int> ptr = foo();
```

与此相比：

```
int* foo_cpp03();

int* p = foo_cpp03(); // 我拥有p吗？我是否需要在某个时候删除它？
                      // 答案并不明显。
```

C++14之前的版本

类模板make_unique自C++14起提供。可以很容易地手动将其添加到C++11代码中：

```
template<typename T, typename... Args>
typename std::enable_if<!std::is_array<T>::value, std::unique_ptr<T>>::type
make_unique(Args&&... args)
{ return std::unique_ptr<T>(new T(std::forward<Args>(args)...)); }

// 对数组使用 make_unique
template<typename T>
typename std::enable_if<std::is_array<T>::value, std::unique_ptr<T>>::type
make_unique(size_t n)
{ return std::unique_ptr<T>(new typename std::remove_extent<T>::type[n]); }
```

版本 ≥ C++11

与“笨拙”的智能指针（`std::auto_ptr`）不同，`unique_ptr`也可以用于向量分配（不是 `std::vector`）。之前的示例是针对标量分配的。例如，要动态分配一个包含 10 个元素的整数数组，您需要将模板类型指定为 `int[]`（而不仅仅是 `int`）：

```
std::unique_ptr<int[]> arr_ptr = std::make_unique<int[]>(10);
```

可以简化为：

```
auto arr_ptr = std::make_unique<int[]>(10);
```

现在，您可以像使用数组一样使用 `arr_ptr`：

```
arr_ptr[2] = 10; // 修改第三个元素
```

您无需担心内存释放。该模板特化版本会适当地调用构造函数和析构函数。使用基于`unique_ptr`的向量版本或`vector`本身——这是个人选择。

在C++11之前的版本中，提供了`std::auto_ptr`。与`unique_ptr`不同，`auto_ptr`允许复制，复制后源ptr会失去所包含指针的所有权，目标则获得该所有权。

第33.2节：共享所有权 (`std::shared_ptr`)

类模板`std::shared_ptr`定义了一个共享指针，能够与

conveys the ownership semantics of the return: the caller owns the resulting unique_ptr and is responsible for it.

```
std::unique_ptr<int> foo()
{
    std::unique_ptr<int> ptr = std::make_unique<int>(59);
    return ptr;
}

std::unique_ptr<int> ptr = foo();
```

Compare this to:

```
int* foo_cpp03();

int* p = foo_cpp03(); // do I own p? do I have to delete it at some point?
                      // it's not readily apparent what the answer is.
```

Version < C++14

The class template `make_unique` is provided since C++14. It's easy to add it manually to C++11 code:

```
template<typename T, typename... Args>
typename std::enable_if<!std::is_array<T>::value, std::unique_ptr<T>>::type
make_unique(Args&&... args)
{ return std::unique_ptr<T>(new T(std::forward<Args>(args)...)); }

// Use make_unique for arrays
template<typename T>
typename std::enable_if<std::is_array<T>::value, std::unique_ptr<T>>::type
make_unique(size_t n)
{ return std::unique_ptr<T>(new typename std::remove_extent<T>::type[n]); }
```

Version ≥ C++11

Unlike the *dumb* smart pointer (`std::auto_ptr`), `unique_ptr` can also be instantiated with vector allocation (not `std::vector`). Earlier examples were for *scalar* allocations. For example to have a dynamically allocated integer array for 10 elements, you would specify `int[]` as the template type (and not just `int`):

```
std::unique_ptr<int[]> arr_ptr = std::make_unique<int[]>(10);
```

Which can be simplified with:

```
auto arr_ptr = std::make_unique<int[]>(10);
```

Now, you use `arr_ptr` as if it is an array:

```
arr_ptr[2] = 10; // Modify third element
```

You need not to worry about de-allocation. This template specialized version calls constructors and destructors appropriately. Using vectored version of `unique_ptr` or a `vector` itself - is a personal choice.

In versions prior to C++11, `std::auto_ptr` was available. Unlike `unique_ptr` it is allowed to copy `auto_ptrs`, upon which the source `ptr` will lose the ownership of the contained pointer and the target receives it.

Section 33.2: Sharing ownership (`std::shared_ptr`)

The class template `std::shared_ptr` defines a shared pointer that is able to share ownership of an object with

其他共享指针共享对象的所有权。这与表示独占所有权的std::unique_ptr形成对比。

共享行为通过一种称为引用计数的技术实现，引用计数记录指向该对象的共享指针数量。当该计数归零时，无论是通过最后一个std::shared_ptr实例的销毁还是重新赋值，都会自动销毁该对象。

```
// 创建：'firstShared' 是指向新创建的 'Foo' 实例的共享指针
std::shared_ptr<Foo> firstShared = std::make_shared<Foo>(/*args*/);
```

要创建多个共享同一对象的智能指针，我们需要创建另一个别名第一个共享指针的shared_ptr。以下是两种方法：

```
std::shared_ptr<Foo> secondShared(firstShared); // 第一种方法：拷贝构造
std::shared_ptr<Foo> secondShared;
secondShared = firstShared; // 第二种方式：赋值
```

以上任一方式都会使 secondShared 成为一个共享指针，与

firstShared 共享我们 Foo 实例的所有权。

智能指针的工作方式就像原始指针。这意味着，你可以使用 * 来解引用它们。常规的 -> 操作符也同样适用：

```
secondShared->test(); // 调用 Foo::test()
```

最后，当最后一个别名的 shared_ptr 超出作用域时，我们的 Foo 实例的析构函数会被调用。

警告：构造一个 shared_ptr 可能会抛出 bad_alloc 异常，当需要为共享所有权语义分配额外数据时。如果构造函数传入一个普通指针，它假定拥有该指针指向的对象，并且如果抛出异常会调用删除器。这意味着 shared_ptr<T>(new T(args)) 在 shared_ptr<T> 分配失败时不会泄漏 T 对象。然而，建议使用 make_shared<T>(args) 或 allocate_shared<T>(alloc, args)，这些方法使实现能够优化内存分配。

使用 shared_ptr 分配数组([])

版本 ≥ C++11 版本 < C++17

遗憾的是，没有直接的方法使用 make_shared<> 来分配数组。

可以使用 new 和 std::default_delete 为 shared_ptr<> 创建数组。

例如，要分配一个包含 10 个整数的数组，我们可以这样写代码

```
shared_ptr<int> sh(new int[10], std::default_delete<int[]>());
```

这里必须指定 std::default_delete，以确保分配的内存能够通过 delete[] 正确释放。

如果我们在编译时就知道大小，可以这样做：

```
template<class Arr>
struct shared_array_maker {};
template<class T, std::size_t N>
struct shared_array_maker<T[N]> {
    std::shared_ptr<T> operator() const {
```

other shared pointers. This contrasts to std::unique_ptr which represents exclusive ownership.

The sharing behavior is implemented through a technique known as reference counting, where the number of shared pointers that point to the object is stored alongside it. When this count reaches zero, either through the destruction or reassignment of the last std::shared_ptr instance, the object is automatically destroyed.

```
// Creation: 'firstShared' is a shared pointer for a new instance of 'Foo'
std::shared_ptr<Foo> firstShared = std::make_shared<Foo>(/*args*/);
```

To create multiple smart pointers that share the same object, we need to create another shared_ptr that aliases the first shared pointer. Here are 2 ways of doing it:

```
std::shared_ptr<Foo> secondShared(firstShared); // 1st way: Copy constructing
std::shared_ptr<Foo> secondShared;
secondShared = firstShared; // 2nd way: Assigning
```

Either of the above ways makes secondShared a shared pointer that shares ownership of our instance of Foo with firstShared.

The smart pointer works just like a raw pointer. This means, you can use * to dereference them. The regular -> operator works as well:

```
secondShared->test(); // Calls Foo::test()
```

Finally, when the last aliased shared_ptr goes out of scope, the destructor of our Foo instance is called.

Warning: Constructing a shared_ptr might throw a bad_alloc exception when extra data for shared ownership semantics needs to be allocated. If the constructor is passed a regular pointer it assumes to own the object pointed to and calls the deleter if an exception is thrown. This means shared_ptr<T>(new T(args)) will not leak a T object if allocation of shared_ptr<T> fails. However, it is advisable to use make_shared<T>(args) or allocate_shared<T>(alloc, args), which enable the implementation to optimize the memory allocation.

Allocating Arrays([]) using shared_ptr

Version ≥ C++11 Version < C++17

Unfortunately, there is no direct way to allocate Arrays using make_shared<>.

It is possible to create arrays for shared_ptr<> using new and std::default_delete.

For example, to allocate an array of 10 integers, we can write the code as

```
shared_ptr<int> sh(new int[10], std::default_delete<int[]>());
```

Specifying std::default_delete is mandatory here to make sure that the allocated memory is correctly cleaned up using delete[].

If we know the size at compile time, we can do it this way:

```
template<class Arr>
struct shared_array_maker {};
template<class T, std::size_t N>
struct shared_array_maker<T[N]> {
    std::shared_ptr<T> operator() const {
```

```

auto r = std::make_shared<std::array<T,N>>();
if (!r) return {};
return {r.data(), r};
}
template<class Arr>
auto make_shared_array()
-> decltype( shared_array_maker<Arr>{}() )
{ return shared_array_maker<Arr>{}(); }

```

那么`make_shared_array<int[10]>`将返回一个指向10个默认构造整数的`shared_ptr<int>`。

版本 ≥ C++17

在 C++17 中，`shared_ptr` 获得了对数组类型的特殊支持。不再需要显式指定数组删除器，且共享指针可以使用 `[]` 数组索引运算符进行解引用：

```

std::shared_ptr<int[]> sh(new int[10]);
sh[0] = 42;

```

共享指针可以指向其所拥有对象的子对象：

```

struct Foo { int x; };
std::shared_ptr<Foo> p1 = std::make_shared<Foo>();
std::shared_ptr<int> p2(p1, &p1->x);

```

`p2` 和 `p1` 都拥有类型为 `Foo` 的对象，但 `p2` 指向其 `int` 成员 `x`。这意味着如果 `p1` 超出作用域或被重新赋值，底层的 `Foo` 对象仍然存在，确保 `p2` 不会成为悬空指针。

重要提示：`shared_ptr` 只知道自身以及所有通过别名构造函数创建的其他 `shared_ptr`。它不知道任何其他指针，包括所有通过引用同一个 `Foo` 实例创建的其他 `shared_ptr`：

```

Foo *foo = new Foo;
std::shared_ptr<Foo> shared1(foo);
std::shared_ptr<Foo> shared2(foo); // 不要这样做

shared1.reset(); // 这将删除 foo，因为 shared1
                 // 是唯一拥有它的 shared_ptr

shared2->test(); // 未定义行为：shared2 的 foo 已经
                  // 被删除！

```

shared_ptr 的所有权转移

默认情况下，`shared_ptr` 会增加引用计数而不转移所有权。然而，可以使用 `std::move` 来实现所有权的转移：

```

shared_ptr<int> up = make_shared<int>();
// 转移所有权
shared_ptr<int> up2 = move(up);
// 此时，up 的引用计数为 0,
// 指针的所有权完全转移到 up2，引用计数为 1

```

第33.3节：带临时所有权的共享

```

auto r = std::make_shared<std::array<T,N>>();
if (!r) return {};
return {r.data(), r};
}
template<class Arr>
auto make_shared_array()
-> decltype( shared_array_maker<Arr>{}() )
{ return shared_array_maker<Arr>{}(); }

```

then `make_shared_array<int[10]>` returns a `shared_ptr<int>` pointing to 10 ints all default constructed.

Version ≥ C++17

With C++17, `shared_ptr` gained special support for array types. It is no longer necessary to specify the array-deleter explicitly, and the shared pointer can be dereferenced using the `[]` array index operator:

```

std::shared_ptr<int[]> sh(new int[10]);
sh[0] = 42;

```

Shared pointers can point to a sub-object of the object it owns:

```

struct Foo { int x; };
std::shared_ptr<Foo> p1 = std::make_shared<Foo>();
std::shared_ptr<int> p2(p1, &p1->x);

```

Both `p2` and `p1` own the object of type `Foo`, but `p2` points to its `int` member `x`. This means that if `p1` goes out of scope or is reassigned, the underlying `Foo` object will still be alive, ensuring that `p2` does not dangle.

Important: A `shared_ptr` only knows about itself and all other `shared_ptr` that were created with the alias constructor. It does not know about any other pointers, including all other `shared_ptr`s created with a reference to the same `Foo` instance:

```

Foo *foo = new Foo;
std::shared_ptr<Foo> shared1(foo);
std::shared_ptr<Foo> shared2(foo); // don't do this

shared1.reset(); // this will delete foo, since shared1
                 // was the only shared_ptr that owned it

shared2->test(); // UNDEFINED BEHAVIOR: shared2's foo has been
                  // deleted already!!

```

Ownership Transfer of shared_ptr

By default, `shared_ptr` increments the reference count and doesn't transfer the ownership. However, it can be made to transfer the ownership using `std::move`:

```

shared_ptr<int> up = make_shared<int>();
// Transferring the ownership
shared_ptr<int> up2 = move(up);
// At this point, the reference count of up = 0 and the
// ownership of the pointer is solely with up2 with reference count = 1

```

Section 33.3: Sharing with temporary ownership

(std::weak_ptr)

std::weak_ptr 的实例可以指向由 std::shared_ptr 实例拥有的对象，同时自身仅作为临时所有者。这意味着弱指针不会改变对象的引用计数，因此如果对象的所有共享指针都被重新赋值或销毁，弱指针不会阻止对象的删除。

在下面的示例中，使用了 std::weak_ptr 的实例，以确保树对象的销毁不会被阻止：

```
#include <memory>
#include <vector>

struct TreeNode {
    std::weak_ptr<TreeNode> parent;
    std::vector< std::shared_ptr<TreeNode> > children;
};

int main() {
    // 创建一个 TreeNode 作为根节点/父节点。
    std::shared_ptr<TreeNode> root(new TreeNode);

    // 给父节点添加 100 个子节点。
    for (size_t i = 0; i < 100; ++i) {
        std::shared_ptr<TreeNode> child(new TreeNode);
        root->children.push_back(child);
        child->parent = root;
    }

    // 重置 root 共享指针，销毁根对象，
    // 以及随后销毁其子节点。
    root.reset();
}
```

当子节点被添加到根节点的 children 中时，它们的 std::weak_ptr 成员 parent 被设置为根节点。成员 parent 被声明为弱指针而非共享指针，以避免增加根节点的引用计数。当 main() 结束时重置根节点，根节点被销毁。由于指向子节点的唯一剩余 std::shared_ptr 引用都包含在根节点的集合 children 中，所有子节点也随之被销毁。

由于控制块的实现细节，shared_ptr 分配的内存可能不会被释放，直到 shared_ptr 引用计数和 weak_ptr 引用计数都达到零。

```
#include <memory>
int main()
{
    {
        std::weak_ptr<int> wk;
        {
            // std::make_shared 通过只分配一次进行了优化
            // 而 std::shared_ptr<int>(new int(42)) 会分配两次。
            // std::make_shared 的缺点是控制块绑定到了我们的整数上
            std::shared_ptr<int> sh = std::make_shared<int>(42);
            wk = sh;
            // sh 的内存此时应该被释放...
        }
        // ... 但 wk 仍然存活并且需要访问控制块
    }
    // 现在内存被释放了 (sh 和 wk)
}
```

(std::weak_ptr)

Instances of std::weak_ptr can point to objects owned by instances of std::shared_ptr while only becoming temporary owners themselves. This means that weak pointers do not alter the object's reference count and therefore do not prevent an object's deletion if all of the object's shared pointers are reassigned or destroyed.

In the following example instances of std::weak_ptr are used so that the destruction of a tree object is not inhibited:

```
#include <memory>
#include <vector>

struct TreeNode {
    std::weak_ptr<TreeNode> parent;
    std::vector< std::shared_ptr<TreeNode> > children;
};

int main() {
    // Create a TreeNode to serve as the root/parent.
    std::shared_ptr<TreeNode> root(new TreeNode);

    // Give the parent 100 child nodes.
    for (size_t i = 0; i < 100; ++i) {
        std::shared_ptr<TreeNode> child(new TreeNode);
        root->children.push_back(child);
        child->parent = root;
    }

    // Reset the root shared pointer, destroying the root object, and
    // subsequently its child nodes.
    root.reset();
}
```

As child nodes are added to the root node's children, their std::weak_ptr member parent is set to the root node. The member parent is declared as a weak pointer as opposed to a shared pointer such that the root node's reference count is not incremented. When the root node is reset at the end of main(), the root is destroyed. Since the only remaining std::shared_ptr references to the child nodes were contained in the root's collection children, all child nodes are subsequently destroyed as well.

Due to control block implementation details, shared_ptr allocated memory may not be released until shared_ptr reference counter and weak_ptr reference counter both reach zero.

```
#include <memory>
int main()
{
    {
        std::weak_ptr<int> wk;
        {
            // std::make_shared is optimized by allocating only once
            // while std::shared_ptr<int>(new int(42)) allocates twice.
            // Drawback of std::make_shared is that control block is tied to our integer
            std::shared_ptr<int> sh = std::make_shared<int>(42);
            wk = sh;
            // sh memory should be released at this point...
        }
        // ... but wk is still alive and needs access to control block
    }
    // now memory is released (sh and wk)
}
```

}

由于 `std::weak_ptr` 不会保持其引用对象的存活，不能通过 `std::weak_ptr` 直接访问数据。取而代之的是它提供了一个 `lock()` 成员函数，尝试获取一个指向被引用对象的 `std::shared_ptr`：

```
#include <cassert>
#include <memory>
int main()
{
    {
        std::weak_ptr<int> wk;
        std::shared_ptr<int> sp;
        {
            std::shared_ptr<int> sh = std::make_shared<int>(42);
            wk = sh;
            // 调用 lock 会创建一个指向 wk 所引用对象的 shared_ptr
            sp = wk.lock();
            // sh 在此之后将被销毁，但 sp 仍然存活
        }
        // sp 仍然保持数据存活。
        // 此时我们甚至可以再次调用 lock()
        // 从 wk 获取另一个指向相同数据的 shared_ptr
        assert(*sp == 42);
        assert(!wk.expired());
        // 重置 sp 将删除数据,
        // 因为它当前是最后一个拥有所有权的 shared_ptr
        sp.reset();
        // 现在尝试锁定 wk 将返回一个空的 shared_ptr,
        // 因为数据已经被删除
        sp = wk.lock();
        assert(!sp);
        assert(wk.expired());
    }
}
```

}

Since `std::weak_ptr` does not keep its referenced object alive, direct data access through a `std::weak_ptr` is not possible. Instead it provides a `lock()` member function that attempts to retrieve a `std::shared_ptr` to the referenced object:

```
#include <cassert>
#include <memory>
int main()
{
    {
        std::weak_ptr<int> wk;
        std::shared_ptr<int> sp;
        {
            std::shared_ptr<int> sh = std::make_shared<int>(42);
            wk = sh;
            // calling lock will create a shared_ptr to the object referenced by wk
            sp = wk.lock();
            // sh will be destroyed after this point, but sp is still alive
        }
        // sp still keeps the data alive.
        // At this point we could even call lock() again
        // to retrieve another shared_ptr to the same data from wk
        assert(*sp == 42);
        assert(!wk.expired());
        // resetting sp will delete the data,
        // as it is currently the last shared_ptr with ownership
        sp.reset();
        // attempting to lock wk now will return an empty shared_ptr,
        // as the data has already been deleted
        sp = wk.lock();
        assert(!sp);
        assert(wk.expired());
    }
}
```

第33.4节：使用自定义删除器创建C接口的包装器

许多 C 接口（如 `SDL2`）都有自己的删除函数。这意味着你不能直接使用智能指针：

```
std::unique_ptr<SDL_Surface> a; // 不可用，不安全！
```

相反，你需要定义自己的删除器。这里的示例使用了 `SDL_Surface` 结构体，应该使用 `SDL_FreeSurface()` 函数释放，但它们应该可以适配许多其他 C 接口。

删除器必须可以用指针参数调用，因此可以是例如一个简单的函数指针：

```
std::unique_ptr<SDL_Surface, void(*)(SDL_Surface*)> a(pointer, SDL_FreeSurface);
```

任何其他可调用对象也可以，例如带有 `operator()` 的类：

```
struct SurfaceDeleter {
    void operator()(SDL_Surface* surf) {
        SDL_FreeSurface(surf);
    }
};
```

Section 33.4: Using custom deleters to create a wrapper to a C interface

Many C interfaces such as `SDL2` have their own deletion functions. This means that you cannot use smart pointers directly:

```
std::unique_ptr<SDL_Surface> a; // won't work, UNSAFE!
```

Instead, you need to define your own deleter. The examples here use the `SDL_Surface` structure which should be freed using the `SDL_FreeSurface()` function, but they should be adaptable to many other C interfaces.

The deleter must be callable with a pointer argument, and therefore can be e.g. a simple function pointer:

```
std::unique_ptr<SDL_Surface, void(*)(SDL_Surface*)> a(pointer, SDL_FreeSurface);
```

Any other callable object will work, too, for example a class with an `operator()`:

```
struct SurfaceDeleter {
    void operator()(SDL_Surface* surf) {
        SDL_FreeSurface(surf);
    }
};
```

```
std::unique_ptr<SDL_Surface, SurfaceDeleter> a(pointer, SurfaceDeleter{}); // 安全  
std::unique_ptr<SDL_Surface, SurfaceDeleter> b(pointer); // 等同于上面  
// 因为删除器是值初始化的
```

这不仅为你提供了安全、零开销（如果使用unique_ptr）的自动内存管理，还提供了异常安全。

注意，删除器是unique_ptr类型的一部分，且实现可以使用空基类优化来避免空自定义删除器导致大小变化。因此，虽然std::unique_ptr<SDL_Surface, SurfaceDeleter>和std::unique_ptr<SDL_Surface, void(*)(SDL_Surface*)>以类似方式解决相同问题，前者类型仍然只有一个指针大小，而后者类型必须持有两个指针：既有SDL_Surface*也有函数指针！当使用自由函数自定义删除器时，最好将函数包装在一个空类型中。

在引用计数很重要的情况下，可以使用shared_ptr代替unique_ptr。shared_ptr总是存储一个删除器，这会擦除删除器的类型，这在API中可能很有用。使用shared_ptr而不是unique_ptr的缺点包括存储删除器的内存开销更大，以及维护引用计数的性能开销。

```
// 删除器在构造时需要，并且是类型的一部分  
std::unique_ptr<SDL_Surface, void(*)(SDL_Surface*)> a(pointer, SDL_FreeSurface);  
  
// 删除器只在构造时需要，不是类型的一部分  
std::shared_ptr<SDL_Surface> b(pointer, SDL_FreeSurface);
```

版本 ≥ C++17

使用template auto，我们可以更轻松地封装自定义删除器：

```
template <auto DeleteFn>  
struct FunctionDeleter {  
    template <class T>  
    void operator()(T* ptr) {  
        DeleteFn(ptr);  
    }  
};  
  
template <class T, auto DeleteFn>  
using unique_ptr_deleter = std::unique_ptr<T, FunctionDeleter<DeleteFn>>;
```

这样，上面的例子就简化为：

```
unique_ptr_deleter<SDL_Surface, SDL_FreeSurface> c(pointer);
```

这里，auto的作用是处理所有的自由函数，无论它们是否返回void（例如SDL_FreeSurface）还是不返回（例如fclose）。

第33.5节：无移动语义的唯一所有权（auto_ptr）

版本 < C++11

注意：std::auto_ptr 在 C++11 中已被弃用，并将在 C++17 中移除。只有在被迫使用 C++03 或更早版本且愿意小心使用时，才应使用它。建议结合 std::move 使用 unique_ptr 来替代 std::auto_ptr 的行为。

```
std::unique_ptr<SDL_Surface, SurfaceDeleter> a(pointer, SurfaceDeleter{}); // safe  
std::unique_ptr<SDL_Surface, SurfaceDeleter> b(pointer); // equivalent to the above  
// as the deleter is value-initialized
```

This not only provides you with safe, zero overhead (if you use unique_ptr) automatic memory management, you also get exception safety.

Note that the deleter is part of the type for unique_ptr, and the implementation can use the empty base optimization to avoid any change in size for empty custom deleters. So while std::unique_ptr<SDL_Surface, SurfaceDeleter> and std::unique_ptr<SDL_Surface, void(*)(SDL_Surface*)> solve the same problem in a similar way, the former type is still only the size of a pointer while the latter type has to hold *two* pointers: both the *SDL_Surface** and the function pointer! When having free function custom deleters, it is preferable to wrap the function in an empty type.

In cases where reference counting is important, one could use a shared_ptr instead of an unique_ptr. The shared_ptr always stores a deleter, this erases the type of the deleter, which might be useful in APIs. The disadvantages of using shared_ptr over unique_ptr include a higher memory cost for storing the deleter and a performance cost for maintaining the reference count.

```
// deleter required at construction time and is part of the type  
std::unique_ptr<SDL_Surface, void(*)(SDL_Surface*)> a(pointer, SDL_FreeSurface);  
  
// deleter is only required at construction time, not part of the type  
std::shared_ptr<SDL_Surface> b(pointer, SDL_FreeSurface);
```

Version ≥ C++17

With template auto, we can make it even easier to wrap our custom deleters:

```
template <auto DeleteFn>  
struct FunctionDeleter {  
    template <class T>  
    void operator()(T* ptr) {  
        DeleteFn(ptr);  
    }  
};  
  
template <class T, auto DeleteFn>  
using unique_ptr_deleter = std::unique_ptr<T, FunctionDeleter<DeleteFn>>;
```

With which the above example is simply:

```
unique_ptr_deleter<SDL_Surface, SDL_FreeSurface> c(pointer);
```

Here, the purpose of auto is to handle all free functions, whether they return void (e.g. *SDL_FreeSurface*) or not (e.g. *fclose*).

Section 33.5: Unique ownership without move semantics (auto_ptr)

Version < C++11

NOTE: std::auto_ptr has been deprecated in C++11 and will be removed in C++17. You should only use this if you are forced to use C++03 or earlier and are willing to be careful. It is recommended to move to unique_ptr in combination with std::move to replace std::auto_ptr behavior.

在我们拥有 `std::unique_ptr` 和移动语义之前，我们有 `std::auto_ptr`。`std::auto_ptr` 提供唯一所有权，但在复制时会转移所有权。

与所有智能指针一样，`std::auto_ptr` 会自动清理资源（参见 RAI）：

```
{  
std::auto_ptr<int> p(new int(42));  
    std::cout << *p;  
} // p 在此处被删除，无内存泄漏
```

但只允许一个所有者：

```
std::auto_ptr<X> px = ...;  
std::auto_ptr<X> py = px;  
// px 现在为空
```

这允许使用 `std::auto_ptr` 明确且唯一地保持所有权，但存在意外丢失所有权的风险：

```
void f(std::auto_ptr<X> ) {  
    // 假设拥有 X 的所有权  
    // 在作用域结束时删除它  
};  
  
std::auto_ptr<X> px = ...;  
f(px); // f 获得了底层 X 的所有权  
// px 现在为空  
px->foo(); // 空指针异常 (NPE) !  
// px.>~auto_ptr() 不会删除
```

所有权的转移发生在“拷贝”构造函数中。`auto_ptr` 的拷贝构造函数和拷贝赋值运算符通过非常量引用参数接收操作数，以便对其进行修改。一个示例实现可能是：

```
template <typename T>  
class auto_ptr {  
    T* ptr;  
public:  
    auto_ptr(auto_ptr& rhs)  
        : ptr(rhs.release())  
    {}  
  
    auto_ptr& operator=(auto_ptr& rhs) {  
        reset(rhs.release());  
        return *this;  
    }  
  
    T* release() {  
        T* tmp = ptr;  
        ptr = nullptr;  
        return tmp;  
    }  
  
    void reset(T* tmp = nullptr) {  
        if (ptr != tmp) {  
            delete ptr;  
            ptr = tmp;  
        }  
    }  
}
```

Before we had `std::unique_ptr`, before we had move semantics, we had `std::auto_ptr`. `std::auto_ptr` provides unique ownership but transfers ownership upon copy.

As with all smart pointers, `std::auto_ptr` automatically cleans up resources (see RAI):

```
{  
    std::auto_ptr<int> p(new int(42));  
    std::cout << *p;  
} // p is deleted here, no memory leaked
```

but allows only one owner:

```
std::auto_ptr<X> px = ...;  
std::auto_ptr<X> py = px;  
// px is now empty
```

This allows to use `std::auto_ptr` to keep ownership explicit and unique at the danger of losing ownership unintended:

```
void f(std::auto_ptr<X> ) {  
    // assumes ownership of X  
    // deletes it at end of scope  
};  
  
std::auto_ptr<X> px = ...;  
f(px); // f acquires ownership of underlying X  
// px is now empty  
px->foo(); // NPE!  
// px.>~auto_ptr() does NOT delete
```

The transfer of ownership happened in the "copy" constructor. `auto_ptr`'s copy constructor and copy assignment operator take their operands by non-`const` reference so that they could be modified. An example implementation might be:

```
template <typename T>  
class auto_ptr {  
    T* ptr;  
public:  
    auto_ptr(auto_ptr& rhs)  
        : ptr(rhs.release())  
    {}  
  
    auto_ptr& operator=(auto_ptr& rhs) {  
        reset(rhs.release());  
        return *this;  
    }  
  
    T* release() {  
        T* tmp = ptr;  
        ptr = nullptr;  
        return tmp;  
    }  
  
    void reset(T* tmp = nullptr) {  
        if (ptr != tmp) {  
            delete ptr;  
            ptr = tmp;  
        }  
    }  
}
```

```
/* 其他函数 ... */  
};
```

这破坏了复制语义，复制语义要求复制一个对象后，你会得到两个等价的版本。对于任何可复制类型T，我应该能够写出：

```
T a = ...;  
T b(a);  
assert(b == a);
```

但对于auto_ptr，情况并非如此。因此，将auto_ptr放入容器中是不安全的。

第33.6节：转换std::shared_ptr指针

不能直接对std::shared_ptr使用static_cast、const_cast、dynamic_cast和reinterpret_cast来获取与作为参数传递的指针共享所有权的指针。相反，应使用函数std::static_pointer_cast、std::const_pointer_cast、std::dynamic_pointer_cast和std::reinterpret_pointer_cast：

```
struct Base { virtual ~Base() noexcept {} };  
struct Derived: Base {};  
auto derivedPtr(std::make_shared<Derived>());  
auto basePtr(std::static_pointer_cast<Base>(derivedPtr));  
auto constBasePtr(std::const_pointer_cast<Base const>(basePtr));  
auto constDerivedPtr(std::dynamic_pointer_cast<Derived const>(constBasePtr));
```

注意，std::reinterpret_pointer_cast在C++11和C++14中不可用，因为它仅由N3920提议，并于2014年2月被纳入库基础TSs。然而，它可以按如下方式实现：

```
template <typename 到, typename 从>  
inline std::shared_ptr<到> reinterpret_pointer_cast(  
    std::shared_ptr<从> const & ptr) noexcept  
{ return std::shared_ptr<到>(ptr, reinterpret_cast<到 *>(ptr.get())); }
```

第33.7节：编写智能指针：value_ptr

value_ptr 是一种表现得像值的智能指针。复制时，它会复制其内容。创建时，它会创建其内容。

```
// 类似于 std::default_delete：  
template<class T>  
struct default_copier {  
    // 复制器必须处理空的 T const* 输入并返回空指针：  
    T* operator()(T const* tin) const {  
        if (!tin) return nullptr;  
        return new T(*tin);  
    }  
    void operator()(void* dest, T const* tin) const {  
        if (!tin) return;  
        return new(dest) T(*tin);  
    }  
};  
// 处理空情况的标签类：  
struct empty_ptr_t {};  
constexpr empty_ptr_t empty_ptr{};  
// 值指针类型本身：
```

```
/* other functions ... */  
};
```

This breaks copy semantics, which require that copying an object leaves you with two equivalent versions of it. For any copyable type, T, I should be able to write:

```
T a = ...;  
T b(a);  
assert(b == a);
```

But for auto_ptr, this is not the case. As a result, it is not safe to put auto_ptrs in containers.

Section 33.6: Casting std::shared_ptr pointers

It is not possible to directly use `static_cast`, `const_cast`, `dynamic_cast` and `reinterpret_cast` on `std::shared_ptr` to retrieve a pointer sharing ownership with the pointer being passed as argument. Instead, the functions `std::static_pointer_cast`, `std::const_pointer_cast`, `std::dynamic_pointer_cast` and `std::reinterpret_pointer_cast` should be used:

```
struct Base { virtual ~Base() noexcept {} };  
struct Derived: Base {};  
auto derivedPtr(std::make_shared<Derived>());  
auto basePtr(std::static_pointer_cast<Base>(derivedPtr));  
auto constBasePtr(std::const_pointer_cast<Base const>(basePtr));  
auto constDerivedPtr(std::dynamic_pointer_cast<Derived const>(constBasePtr));
```

Note that `std::reinterpret_pointer_cast` is not available in C++11 and C++14, as it was only proposed by [N3920](#) and adopted into Library Fundamentals TS [in February 2014](#). However, it can be implemented as follows:

```
template <typename To, typename From>  
inline std::shared_ptr<To> reinterpret_pointer_cast(  
    std::shared_ptr<From> const & ptr) noexcept  
{ return std::shared_ptr<To>(ptr, reinterpret_cast<To *>(ptr.get())); }
```

Section 33.7: Writing a smart pointer: value_ptr

A value_ptr is a smart pointer that behaves like a value. When copied, it copies its contents. When created, it creates its contents.

```
// Like std::default_delete：  
template<class T>  
struct default_copier {  
    // a copier must handle a null T const* in and return null:  
    T* operator()(T const* tin) const {  
        if (!tin) return nullptr;  
        return new T(*tin);  
    }  
    void operator()(void* dest, T const* tin) const {  
        if (!tin) return;  
        return new(dest) T(*tin);  
    }  
};  
// tag class to handle empty case:  
struct empty_ptr_t {};  
constexpr empty_ptr_t empty_ptr{};  
// the value pointer type itself:
```

```

模板<类 T, 类 Copier=default_copier<T>, 类 Deleter=std::default_delete<T>,
    类 Base=std::unique_ptr<T, Deleter>>
>
结构体 value_ptr:Base, 私有 Copier {
    使用 copier_type=Copier;
    // 也继承自 unique_ptr 的类型定义

    使用 Base::Base;

value_ptr( T 常量引用& t ):
Base( std::make_unique<T>(t) ),
    Copier()
    {}
value_ptr( T 右值引用&& t ):
Base( std::make_unique<T>(std::move(t)) ),
    Copier()
    {}
    // 几乎不为空：
value_ptr():
Base( std::make_unique<T>() ),
    Copier()
    {}
value_ptr( empty_ptr_t ) {}

value_ptr( Base b, Copier c={} ):
Base(std::move(b)),
    Copier(std::move(c))
    {}

Copier const& get_copier() const {
    return *this;
}

value_ptr clone() const {
    return {
Base(
get_copier()(this->get()),
    this->get_deleter()
),
get_copier()
};
}
value_ptr(value_ptr&&)=default;
value_ptr& operator=(value_ptr&&)=default;

value_ptr(value_ptr const& o):value_ptr(o.clone()) {}
value_ptr& operator=(value_ptr const&o) {
    if (o && *this) {
        // 如果我们两个都非空，则赋值内容：
        **this = *o;
    } else {
        // 否则，赋值一个克隆（克隆本身也可能为空）：
        *this = o.clone();
    }
    return *this;
}
value_ptr& operator=( T const& t ) {
    if (*this) {
        **this = t;
    } 否则 {
        *this = value_ptr(t);
    }
}

```

```

template<class T, class Copier=default_copier<T>, class Deleter=std::default_delete<T>,
    class Base=std::unique_ptr<T, Deleter>>
>
struct value_ptr:Base, private Copier {
    using copier_type=Copier;
    // also typedefs from unique_ptr

    using Base::Base;

value_ptr( T const& t ):
Base( std::make_unique<T>(t) ),
    Copier()
    {}
value_ptr( T && t ):
Base( std::make_unique<T>(std::move(t)) ),
    Copier()
    {}
    // almost-never-empty:
value_ptr():
Base( std::make_unique<T>() ),
    Copier()
    {}
value_ptr( empty_ptr_t ) {}

value_ptr( Base b, Copier c={} ):
Base(std::move(b)),
    Copier(std::move(c))
    {}

Copier const& get_copier() const {
    return *this;
}

value_ptr clone() const {
    return {
Base(
get_copier()(this->get()),
    this->get_deleter()
),
get_copier()
};
}
value_ptr(value_ptr&&)=default;
value_ptr& operator=(value_ptr&&)=default;

value_ptr(value_ptr const& o):value_ptr(o.clone()) {}
value_ptr& operator=(value_ptr const&o) {
    if (o && *this) {
        // if we are both non-null, assign contents:
        **this = *o;
    } else {
        // otherwise, assign a clone (which could itself be null):
        *this = o.clone();
    }
    return *this;
}
value_ptr& operator=( T const& t ) {
    if (*this) {
        **this = t;
    } else {
        *this = value_ptr(t);
    }
}

```

```

    return *this;
}
value_ptr& operator=( T && t ) {
    if (*this) {
        **this = std::move(t);
    } else {
        *this = value_ptr(std::move(t));
    }
    return *this;
}
T& get() { return **this; }
T const& get() const { return **this; }
T* get_pointer() {
    if (!*this) return nullptr;
    return std::addressof(get());
}
T const* get_pointer() const {
    if (!*this) return nullptr;
    return std::addressof(get());
}
// operator-> from unique_ptr
};

template<class T, class... Args>
value_ptr<T> make_value_ptr( Args&&... args ) {
    return {std::make_unique<T>(std::forward<Args>(args)...)};
}

```

这个特定的 value_ptr 只有在使用 empty_ptr_t 构造或从中移动时才为空。它暴露了它是一个 unique_ptr 的事实，因此 explicit operator bool() const 可以在其上使用。.get() 已被更改为返回一个引用（因为它几乎从不为空），而.get_pointer()则返回一个指针。

这个智能指针对于pImpl情况非常有用，在这些情况下，我们希望实现值语义，但又不想将pImpl的内容暴露在实现文件之外。

使用非默认的Copier时，它甚至可以处理知道如何生成其派生类实例并将其转换为值类型的虚基类。

第33.8节：获取指向本对象的shared_ptr

enable_shared_from_this使您能够获取指向this的有效shared_ptr实例。

通过从类模板enable_shared_from_this派生您的类，您继承了一个方法shared_from_this，该方法返回指向this的shared_ptr实例。

注意对象必须首先作为shared_ptr创建：

```

#include <memory>
class A: public enable_shared_from_this<A> {
};
A* ap1 =new A();
shared_ptr<A> ap2(ap1); // 首先准备一个指向该对象的shared_ptr并持有它！
// 然后从对象本身获取指向该对象的shared_ptr
shared_ptr<A> ap3 = ap1->shared_from_this();
int c3 =ap3.use_count(); // =2: 指向同一个对象

```

注意(2) 你不能在构造函数内部调用 enable_shared_from_this。

```
#include <memory> // enable_shared_from_this
```

```

    return *this;
}
value_ptr& operator=( T && t ) {
    if (*this) {
        **this = std::move(t);
    } else {
        *this = value_ptr(std::move(t));
    }
    return *this;
}
T& get() { return **this; }
T const& get() const { return **this; }
T* get_pointer() {
    if (!*this) return nullptr;
    return std::addressof(get());
}
T const* get_pointer() const {
    if (!*this) return nullptr;
    return std::addressof(get());
}
// operator-> from unique_ptr
};

template<class T, class... Args>
value_ptr<T> make_value_ptr( Args&&... args ) {
    return {std::make_unique<T>(std::forward<Args>(args)...)};
}

```

This particular value_ptr is only empty if you construct it with empty_ptr_t or if you move from it. It exposes the fact it is a unique_ptr, so explicit operator bool() const works on it. .get() has been changed to return a reference (as it is almost never empty), and .get_pointer() returns a pointer instead.

This smart pointer can be useful for pImpl cases, where we want value-semantics but we also don't want to expose the contents of the pImpl outside of the implementation file.

With a non-default Copier, it can even handle virtual base classes that know how to produce instances of their derived and turn them into value-types.

Section 33.8: Getting a shared_ptr referring to this

enable_shared_from_this enables you to get a valid shared_ptr instance to this.

By deriving your class from the class template enable_shared_from_this, you inherit a method shared_from_this that returns a shared_ptr instance to this.

Note that the object must be created as a shared_ptr in first place:

```

#include <memory>
class A: public enable_shared_from_this<A> {
};
A* ap1 =new A();
shared_ptr<A> ap2(ap1); // First prepare a shared pointer to the object and hold it!
// Then get a shared pointer to the object from the object itself
shared_ptr<A> ap3 = ap1->shared_from_this();
int c3 =ap3.use_count(); // =2: pointing to the same object

```

Note(2) you cannot call enable_shared_from_this inside the constructor.

```
#include <memory> // enable_shared_from_this
```

```

class Widget : public std::enable_shared_from_this<Widget>
{
public:
    void DoSomething()
    {
        std::shared_ptr<Widget> self = shared_from_this();
        someEvent -> Register( self );
    }
private:
    ...
};

int main()
{
    ...
    auto w = std::make_shared<Widget>();
    w -> DoSomething();
    ...
}

```

如果你在一个不被 `shared_ptr` 所拥有的对象上使用 `shared_from_this()`, 例如局部自动对象或全局对象, 那么行为是未定义的。自 C++17 起, 它会抛出 `std::bad_alloc` 异常。

在构造函数中使用 `shared_from_this()` 等同于在一个不被 `shared_ptr` 所拥有的对象上使用它, 因为对象是在构造函数返回后才被 `shared_ptr` 拥有的。

```

class Widget : public std::enable_shared_from_this<Widget>
{
public:
    void DoSomething()
    {
        std::shared_ptr<Widget> self = shared_from_this();
        someEvent -> Register( self );
    }
private:
    ...
};

int main()
{
    ...
    auto w = std::make_shared<Widget>();
    w -> DoSomething();
    ...
}

```

If you use `shared_from_this()` on an object not owned by a `shared_ptr`, such as a local automatic object or a global object, then the behavior is undefined. Since C++17 it throws `std::bad_alloc` instead.

Using `shared_from_this()` from a constructor is equivalent to using it on an object not owned by a `shared_ptr`, because the objects is possessed by the `shared_ptr` after the constructor returns.

第34章：类/结构体

第34.1节：类基础

类 (class) 是一种用户定义的类型。类通过class、struct或union关键字引入。在口语中，“类”一词通常仅指非联合类。

类是类成员的集合，这些成员可以是：

- 成员变量（也称为“字段”）、
- 成员函数（也称为“方法”）、
- 成员类型或typedef（例如“嵌套类”）、
- 成员模板（任何类型：变量、函数、类或别名模板）

class和struct关键字，称为类关键字，在很大程度上是可以互换的，唯一的区别是使用class关键字声明的类的成员和基类的默认访问说明符是“私有”，而使用struct或union关键字声明的类的默认访问说明符是“公有”（参见访问修饰符）。

例如，以下代码片段是相同的：

```
struct Vector
{
    int x;
    int y;
    int z;
};

// 等同于
class Vector
{
public:
    int x;
    int y;
    int z;
};
```

通过声明一个类，程序中添加了一个新类型，并且可以通过该类实例化对象

```
Vector my_vector;
```

类的成员通过点语法访问。

```
my_vector.x = 10;
my_vector.y = my_vector.x + 1; // my_vector.y = 11;
my_vector.z = my_vector.y - 4; // my_vector.z = 7;
```

第34.2节：最终类和结构体

版本 ≥ C++11

使用final说明符可以禁止派生类。我们来声明一个最终类：

```
class A final {
};
```

现在任何尝试继承它的行为都会导致编译错误：

Chapter 34: Classes/Structures

Section 34.1: Class basics

A *class* is a user-defined type. A class is introduced with the `class`, `struct` or `union` keyword. In colloquial usage, the term "class" usually refers only to non-union classes.

A class is a collection of *class members*, which can be:

- member variables (also called "fields"),
- member functions (also called "methods"),
- member types or typedefs (e.g. "nested classes"),
- member templates (of any kind: variable, function, class or alias template)

The `class` and `struct` keywords, called *class keys*, are largely interchangeable, except that the default access specifier for members and bases is "private" for a class declared with the `class` key and "public" for a class declared with the `struct` or `union` key (cf. Access modifiers).

For example, the following code snippets are identical:

```
struct Vector
{
    int x;
    int y;
    int z;
};

// are equivalent to
class Vector
{
public:
    int x;
    int y;
    int z;
};
```

By declaring a class ` a new type is added to your program, and it is possible to instantiate objects of that class by

```
Vector my_vector;
```

Members of a class are accessed using dot-syntax.

```
my_vector.x = 10;
my_vector.y = my_vector.x + 1; // my_vector.y = 11;
my_vector.z = my_vector.y - 4; // my_vector.z = 7;
```

Section 34.2: Final classes and structs

Version ≥ C++11

Deriving a class may be forbidden with `final` specifier. Let's declare a final class:

```
class A final {
};
```

Now any attempt to subclass it will cause a compilation error:

```
// 编译错误：不能从最终类派生：  
class B : public A {  
};
```

最终类可以出现在类层次结构的任何位置：

```
class A {  
};  
  
// OK.  
class B final : public A {  
};  
  
// 编译错误：不能从最终类 B 派生。  
class C : public B {  
};
```

```
// Compilation error: cannot derive from final class:  
class B : public A {  
};
```

Final class may appear anywhere in class hierarchy:

```
class A {  
};  
  
// OK.  
class B final : public A {  
};  
  
// Compilation error: cannot derive from final class B.  
class C : public B {  
};
```

第34.3节：访问说明符

有三个关键字作为访问说明符。这些关键字限制了紧随说明符之后的类成员的访问，直到另一个说明符再次改变访问级别：

关键字	描述
public	所有人都可以访问
protected	只有类本身、派生类和友元可以访问
private	只有类本身和友元可以访问

当使用class关键字定义类型时，默认的访问说明符是private，但如果使用struct关键字定义类型，默认的访问说明符是public：

```
struct MyStruct { int x; };  
class MyClass { int x; };
```

```
MyStruct s;  
s.x = 9; // 合法，因为x是public的
```

```
MyClass c;  
c.x = 9; // 不合法，因为x是private的
```

访问说明符主要用于限制对内部字段和方法的访问，并强制程序员使用特定的接口，例如强制使用getter和setter而不是直接引用变量：

```
class MyClass {  
  
public: /* 方法 */  
  
    int x() const noexcept { return m_x; }  
    void setX(int const x) noexcept { m_x = x; }  
  
private: /* 字段 */  
  
    int m_x;  
};
```

使用protected对于允许类型的某些功能仅对派生类可访问非常有用，例如，在以下代码中，方法calculateValue()仅对从该类派生的类可访问

Section 34.3: Access specifiers

There are three keywords that act as **access specifiers**. These limit the access to class members following the specifier, until another specifier changes the access level again:

Keyword	Description
public	Everyone has access
protected	Only the class itself, derived classes and friends have access
private	Only the class itself and friends have access

When the type is defined using the `class` keyword, the default access specifier is `private`, but if the type is defined using the `struct` keyword, the default access specifier is `public`:

```
struct MyStruct { int x; };  
class MyClass { int x; };  
  
MyStruct s;  
s.x = 9; // well formed, because x is public  
  
MyClass c;  
c.x = 9; // ill-formed, because x is private
```

Access specifiers are mostly used to limit access to internal fields and methods, and force the programmer to use a specific interface, for example to force use of getters and setters instead of referencing a variable directly:

```
class MyClass {  
  
public: /* Methods */  
  
    int x() const noexcept { return m_x; }  
    void setX(int const x) noexcept { m_x = x; }  
  
private: /* Fields */  
  
    int m_x;  
};
```

Using `protected` is useful for allowing certain functionality of the type to be only accessible to the derived classes, for example, in the following code, the method `calculateValue()` is only accessible to classes deriving from the

基类 Plus2Base，例如 FortyTwo：

```
结构体 Plus2Base {
    int value() noexcept { return calculateValue() + 2; }
protected: /* 方法: */
    virtual int calculateValue() noexcept = 0;
};

结构体 FortyTwo: Plus2Base {
protected: /* 方法: */
    int calculateValue() noexcept final override { return 40; }
};
```

注意，friend 关键字可以用来为函数或类型添加访问例外，以访问受保护和私有成员。

public、protected 和 private 关键字也可以用来授予或限制对基类子对象的访问权限。

参见继承示例。

第34.4节：继承

类/结构体可以有继承关系。

如果一个类/结构体B继承自一个类/结构体A，这意味着B的父类是A。我们说B是从A派生的类/结构体，A是基类/结构体。

```
结构体 A
{
public:
    int p1;
protected:
    int p2;
private:
    int p3;
};

//使B公开继承（默认）自A
结构体 B : A
```

●
●
●

如果使用struct

关键字，则为public；如果使用class关键字，则为private。

甚至可以让class从struct派生（反之亦然）。在这种情况下，默认继承由子类控制，因此从class派生的struct默认是public继承，而从struct派生的class默认是private继承。

公有继承：

结构体 B : 公有继承 A // 或者直接写成 `struct B : A`

base class Plus2Base, such as FortyTwo:

```
struct Plus2Base {
    int value() noexcept { return calculateValue() + 2; }
protected: /* Methods: */
    virtual int calculateValue() noexcept = 0;
};

struct FortyTwo: Plus2Base {
protected: /* Methods: */
    int calculateValue() noexcept final override { return 40; }
};
```

Note that the `friend` keyword can be used to add access exceptions to functions or types for accessing protected and private members.

The `public`, `protected`, and `private` keywords can also be used to grant or limit access to base class subobjects. See the Inheritance example.

Section 34.4: Inheritance

Classes/structs can have inheritance relations.

If a class/struct B inherits from a class/struct A, this means that B has A as a parent. We say that B is a derived class/struct from A, and A is the base class/struct.

```
struct A
{
public:
    int p1;
protected:
    int p2;
private:
    int p3;
};

//Make B inherit publicly (default) from A
struct B : A
{
};
```

There are 3 forms of inheritance for a class/struct:

- `public`
- `private`
- `protected`

Note that the default inheritance is the same as the default visibility of members: `public` if you use the `struct` keyword, and `private` for the `class` keyword.

It's even possible to have a `class` derive from a `struct` (or vice versa). In this case, the default inheritance is controlled by the child, so a `struct` that derives from a `class` will default to public inheritance, and a `class` that derives from a `struct` will have private inheritance by default.

`public` inheritance:

```
struct B : public A // or just `struct B : A`
```

```

void foo()
{
p1 = 0; // 合法, p1 在 B 中是公有的
p2 = 0; // 合法, p2 在 B 中是保护的
p3 = 0; // 不合法, p3 在 A 中是私有的
}

B b;
b.p1 = 1; // 合法, p1 是公有的
b.p2 = 1; // 不合法, p2 是保护的
b.p3 = 1; // 不合法, p3 无法访问

```

私有继承：

结构体 B : 私有继承 A

```

{
    void foo()
    {
p1 = 0; //格式正确, p1 在 B 中是私有的
p2 = 0; //格式正确, p2 在 B 中是私有的
p3 = 0; //格式错误, p3 在 A 中是私有的
    }
}

B b;
b.p1 = 1; //格式错误, p1 是私有的
b.p2 = 1; //格式错误, p2 是私有的
b.p3 = 1; // 不合法, p3 无法访问

```

受保护的继承：

结构体 B : 受保护的 A

```

{
    void foo()
    {
p1 = 0; //格式正确, p1 在 B 中是受保护的
p2 = 0; //格式正确, p2 在 B 中是受保护的
p3 = 0; //格式错误, p3 在 A 中是私有的
    }
}

B b;
b.p1 = 1; //格式错误, p1 是受保护的
b.p2 = 1; // 不合法, p2 是保护的
b.p3 = 1; // 不合法, p3 无法访问

```

注意，虽然允许受保护的继承，但实际上很少使用。受保护继承在应用中的一个实例是部分基类特化（通常称为“受控多态”）。

当面向对象编程（OOP）相对较新时，（公有）继承常被说成是“是一个”（IS-A）关系。也就是说，只有当派生类的实例也是基类的实例时，公有继承才是正确的。

后来这一点被细化为里氏替换原则：只有当派生类的实例在任何可能的情况下都可以替代基类的实例（且仍然合理）时，才应该使用公有继承。

私有继承通常被认为体现了一种完全不同的关系：“以.....的方式实现”

```

void foo()
{
p1 = 0; //well formed, p1 is public in B
p2 = 0; //well formed, p2 is protected in B
p3 = 0; //ill formed, p3 is private in A
}

B b;
b.p1 = 1; //well formed, p1 is public
b.p2 = 1; //ill formed, p2 is protected
b.p3 = 1; //ill formed, p3 is inaccessible

```

private inheritance:

```

struct B : private A
{
    void foo()
    {
p1 = 0; //well formed, p1 is private in B
p2 = 0; //well formed, p2 is private in B
p3 = 0; //ill formed, p3 is private in A
    }
}

B b;
b.p1 = 1; //ill formed, p1 is private
b.p2 = 1; //ill formed, p2 is private
b.p3 = 1; //ill formed, p3 is inaccessible

```

protected inheritance:

```

struct B : protected A
{
    void foo()
    {
p1 = 0; //well formed, p1 is protected in B
p2 = 0; //well formed, p2 is protected in B
p3 = 0; //ill formed, p3 is private in A
    }
}

B b;
b.p1 = 1; //ill formed, p1 is protected
b.p2 = 1; //ill formed, p2 is protected
b.p3 = 1; //ill formed, p3 is inaccessible

```

Note that although `protected` inheritance is allowed, the actual use of it is rare. One instance of how `protected` inheritance is used in application is in partial base class specialization (usually referred to as "controlled polymorphism").

When OOP was relatively new, (public) inheritance was frequently said to model an "IS-A" relationship. That is, public inheritance is correct only if an instance of the derived class is also an instance of the base class.

This was later refined into the [Liskov Substitution Principle](#): public inheritance should only be used when/if an instance of the derived class can be substituted for an instance of the base class under any possible circumstance (and still make sense).

Private inheritance is typically said to embody a completely different relationship: "is implemented in terms of"

(有时称为“拥有 (HAS-A) ”关系)。例如，一个Stack类可以私有继承自一个Vector类。
私有继承与聚合的相似度远大于与公有继承的相似度。

保护继承几乎从不用，并且对于它所体现的关系类型没有普遍共识。

第34.5节：友元关系

friend关键字用于允许其他类和函数访问类的私有和保护成员，即使它们定义在类的作用域之外。

```
class Animal{
private:
    double weight;
    double height;
public:
    friend void printWeight(Animal animal);
    friend class AnimalPrinter;
    // 友元函数的一个常见用途是重载用于流操作的operator<<.
    friend std::ostream& operator<<(std::ostream& os, Animal animal);
};

void printWeight(动物 animal)
{
    std::cout << animal.weight << "";
}

类 动物打印器
{
public:
    void print(const 动物& animal)
    {
        // 由于声明了 `friend class AnimalPrinter;`，我们
        // 可以在这里访问私有成员。
        std::cout << animal.weight << ", " << animal.height << std::endl;
    }
}

std::ostream& operator<<(std::ostream& os, 动物 animal)
{
    os << "动物高度: " << animal.height << "";
    return os;
}

int main()
{
    动物 animal = {10, 5};
    printWeight(animal);

    动物打印器 aPrinter;
    aPrinter.print(animal);

    std::cout << animal;
}
```

10
10, 5
动物高度: 5

(sometimes called a "HAS-A" relationship). For example, a stack class could inherit privately from a Vector class.
Private inheritance bears a much greater similarity to aggregation than to public inheritance.

Protected inheritance is almost never used, and there's no general agreement on what sort of relationship it embodies.

Section 34.5: Friendship

The `friend` keyword is used to give other classes and functions access to private and protected members of the class, even though they are defined outside the class's scope.

```
class Animal{
private:
    double weight;
    double height;
public:
    friend void printWeight(Animal animal);
    friend class AnimalPrinter;
    // A common use for a friend function is to overload the operator<< for streaming.
    friend std::ostream& operator<<(std::ostream& os, Animal animal);
};

void printWeight(Animal animal)
{
    std::cout << animal.weight << "\n";
}

class AnimalPrinter
{
public:
    void print(const Animal& animal)
    {
        // Because of the `friend class AnimalPrinter;` declaration, we are
        // allowed to access private members here.
        std::cout << animal.weight << ", " << animal.height << std::endl;
    }
}

std::ostream& operator<<(std::ostream& os, Animal animal)
{
    os << "Animal height: " << animal.height << "\n";
    return os;
}

int main()
{
    Animal animal = {10, 5};
    printWeight(animal);

    AnimalPrinter aPrinter;
    aPrinter.print(animal);

    std::cout << animal;
}
```

10
10, 5
Animal height: 5

第34.6节：虚拟继承

使用继承时，可以指定virtual关键字：

```
struct A{};  
struct B: public virtual A{};
```

当类B具有虚基类A时，意味着A将驻留在继承树的最派生类中，因此最派生类也负责初始化该虚基类：

```
结构体 A  
{  
    int member;  
    A(int param)  
    {  
        member = param;  
    }  
}  
  
struct B: virtual A  
{  
    B(): A(5){}  
};  
  
struct C: B  
{  
    C(): /*A(88)*/ {}  
};  
  
void f()  
{  
    C object; //错误，因为C没有初始化它的间接虚基类`A`  
}
```

如果我们取消注释/*A(88)*/，就不会有任何错误，因为C现在初始化了它的间接虚基类A。

还要注意，当我们创建变量object时，最派生的类是C，所以C负责创建（调用构造函数）A，因此A::member的值是88，而不是5（如果我们创建的是B类型的对象，则值为5）。

这在解决菱形继承问题时非常有用：



B和C都继承自A，D继承自B和C，因此D中有两个A的实例！这导致通过D访问A的成员时出现歧义，因为编译器无法确定你想访问的是哪个类继承的成员（是B继承的那个，还是C继承的那个？）。

虚继承解决了这个问题：由于虚基类只存在于最派生对象中，D中只会有一个A的实例。

```
结构体 A  
{  
    void foo() {}
```

Section 34.6: Virtual Inheritance

When using inheritance, you can specify the `virtual` keyword:

```
struct A{};  
struct B: public virtual A{};
```

When class B has virtual base A it means that A **will reside in most derived class** of inheritance tree, and thus that most derived class is also responsible for initializing that virtual base:

```
struct A  
{  
    int member;  
    A(int param)  
    {  
        member = param;  
    }  
}  
  
struct B: virtual A  
{  
    B(): A(5){}  
};  
  
struct C: B  
{  
    C(): /*A(88)*/ {}  
};  
  
void f()  
{  
    C object; //error since C is not initializing its indirect virtual base `A`  
}
```

If we un-comment `/*A(88)*/` we won't get any error since C is now initializing its indirect virtual base A.

Also note that when we're creating variable object, most derived class is C, so C is responsible for creating(calling constructor of) A and thus value of `A::member` is 88, not 5 (as it would be if we were creating object of type B).

It is useful when solving the [diamond problem](#).



B and C both inherit from A, and D inherits from B and C, so **there are 2 instances of A in D!** This results in ambiguity when you're accessing member of A through D, as the compiler has no way of knowing from which class do you want to access that member (the one which B inherits, or the one that is inherited by C?).

Virtual inheritance solves this problem: Since virtual base resides only in most derived object, there will be only one instance of A in D.

```
struct A  
{  
    void foo() {}
```

```
};

结构体 B : 公有继承 /*虚继承*/ A {};
结构体 C : 公有继承 /*虚继承*/ A {};
```

```
结构体 D : 公有继承 B, 公有继承 C
{
    void bar()
    {
        foo(); //错误, 调用哪个 foo? 是 B::foo() 还是 C::foo()? ——歧义
    }
};
```

去掉注释可以解决歧义。

第34.7节：私有继承：限制基类接口

当需要限制类的公共接口时，私有继承非常有用：

```
class A {
public:
    int move();
    int turn();
};

class B : private A {
public:
    using A::turn;
};

B b;
b.move(); // 编译错误
b.turn(); // 正常
```

这种方法通过转换为 A 指针或引用，有效地防止访问 A 的公共方法：

```
B b;
A& a = static_cast<A&>(b); // 编译错误
```

在公有继承的情况下，这种转换将允许访问所有 A 的公共方法，尽管在派生类 B 中有其他防止这种情况的方法，比如隐藏：

```
class B : public A {
private:
    int move();
};
```

或者私有使用：

```
class B : public A {
private:
    using A::move;
};
```

那么在这两种情况下都是可能的：

```
B b;
```

```
};

struct B : public /*virtual*/ A {};
struct C : public /*virtual*/ A {};

struct D : public B, public C
{
    void bar()
    {
        foo(); //Error, which foo? B::foo() or C::foo()? - Ambiguous
    }
};
```

Removing the comments resolves the ambiguity.

Section 34.7: Private inheritance: restricting base class interface

Private inheritance is useful when it is required to restrict the public interface of the class:

```
class A {
public:
    int move();
    int turn();
};

class B : private A {
public:
    using A::turn;
};

B b;
b.move(); // compile error
b.turn(); // OK
```

This approach efficiently prevents an access to the A public methods by casting to the A pointer or reference:

```
B b;
A& a = static_cast<A&>(b); // compile error
```

In the case of public inheritance such casting will provide access to all the A public methods despite on alternative ways to prevent this in derived B, like hiding:

```
class B : public A {
private:
    int move();
};
```

or private using:

```
class B : public A {
private:
    using A::move;
};
```

then for both cases it is possible:

```
B b;
```

```
A& a = static_cast<A&>(b); // 公开继承时可行  
a.move(); // OK
```

第34.8节：访问类成员

要访问类对象的成员变量和成员函数，使用.运算符：

```
结构体 SomeStruct {  
    int a;  
    int b;  
    void foo() {}  
};  
  
SomeStruct var;  
// 访问var中的成员变量a。  
std::cout << var.a << std::endl;  
// 给var中的成员变量b赋值。  
var.b = 1;  
// 调用成员函数。  
var.foo();
```

通过指针访问类的成员时，通常使用->运算符。或者，也可以对实例进行解引用并使用.运算符，尽管这种方式较少见：

```
结构体 SomeStruct {  
    int a;  
    int b;  
    void foo() {}  
};  
  
SomeStruct var;  
SomeStruct *p = &var;  
// 通过指针访问var中的成员变量a。  
std::cout << p->a << std::endl;  
std::cout << (*p).a << std::endl;  
// 通过指针给var中的成员变量b赋值。  
p->b = 1;  
(*p).b = 1;  
// 通过指针调用成员函数。  
p->foo();  
(*p).foo();
```

访问静态类成员时，使用::运算符，但作用于类名而非其实例。或者，也可以通过实例或指向实例的指针使用.或->运算符访问静态成员，语法与访问非静态成员相同。

```
结构体 SomeStruct {  
    int a;  
    int b;  
    void foo() {}  
  
    static int c;  
    static void bar() {}  
};  
int SomeStruct::c;  
  
SomeStruct var;  
SomeStruct* p = &var;  
// 给结构体 SomeStruct 中的静态成员变量 c 赋值。
```

```
A& a = static_cast<A&>(b); // OK for public inheritance  
a.move(); // OK
```

Section 34.8: Accessing class members

To access member variables and member functions of an object of a class, the . operator is used:

```
struct SomeStruct {  
    int a;  
    int b;  
    void foo() {}  
};  
  
SomeStruct var;  
// Accessing member variable a in var.  
std::cout << var.a << std::endl;  
// Assigning member variable b in var.  
var.b = 1;  
// Calling a member function.  
var.foo();
```

When accessing the members of a class via a pointer, the -> operator is commonly used. Alternatively, the instance can be dereferenced and the . operator used, although this is less common:

```
struct SomeStruct {  
    int a;  
    int b;  
    void foo() {}  
};  
  
SomeStruct var;  
SomeStruct *p = &var;  
// Accessing member variable a in var via pointer.  
std::cout << p->a << std::endl;  
std::cout << (*p).a << std::endl;  
// Assigning member variable b in var via pointer.  
p->b = 1;  
(*p).b = 1;  
// Calling a member function via a pointer.  
p->foo();  
(*p).foo();
```

When accessing static class members, the :: operator is used, but on the name of the class instead of an instance of it. Alternatively, the static member can be accessed from an instance or a pointer to an instance using the . or -> operator, respectively, with the same syntax as accessing non-static members.

```
struct SomeStruct {  
    int a;  
    int b;  
    void foo() {}  
  
    static int c;  
    static void bar() {}  
};  
int SomeStruct::c;  
  
SomeStruct var;  
SomeStruct* p = &var;  
// Assigning static member variable c in struct SomeStruct.
```

```
SomeStruct::c = 5;
// 通过 var 和 p 访问结构体 SomeStruct 中的静态成员变量 c。
var.a = var.c;
var.b = p->c;
// 调用静态成员函数。
SomeStruct::bar();
var.bar();
p->bar();
```

背景

需要使用->操作符，因为成员访问操作符.的优先级高于解引用操作符*。

人们通常会认为*p.a会先解引用p（得到p指向对象的引用），然后访问其成员a。但实际上，它尝试先访问p的成员a，然后再解引用它。也就是说，*p.a等价于*(p.a)。在上面的例子中，这会导致编译错误，原因有两个：首先，p是一个指针，没有成员a；其次，a是一个整数，因此不能被解引用。

解决这个问题的一个不常用的方法是显式控制优先级：(*p).a但实际上，几乎总是使用->操作符。它是先解引用指针再访问成员的简写。也就是说，(*p).a与p->a完全相同。

::操作符是作用域操作符，用法类似于访问命名空间的成员。这是因为静态类成员被视为属于该类的作用域，但不被视为该类实例的成员。尽管静态成员不是实例成员，出于历史原因，仍允许使用普通的.和->访问静态成员；这对于模板中编写通用代码很有用，因为调用者无需关心某个成员函数是静态的还是非静态的。

第34.9节：成员类型和别名

class或struct也可以定义成员类型别名，这些别名包含在类中，并被视为类的成员。

```
struct IHaveATypedef {
    typedef int MyTypedef;
};

结构体 IHaveATemplateTypedef {
    模板<类型名 T>
    using MyTemplateTypedef = std::vector<T>;
};
```

像静态成员一样，这些typedef通过作用域运算符::访问。

```
IHaveATypedef::MyTypedef i = 5; // i是一个int类型。
```

```
IHaveATemplateTypedef::MyTemplateTypedef<int> v; // v是一个std::vector<int>。
```

与普通类型别名一样，每个成员类型别名允许引用其定义之前定义或别名的任何类型，但不允许引用之后的类型。同样，类定义外的typedef可以引用类定义内任何可访问的typedef，前提是它位于类定义之后。

```
template<typename T>
struct Helper {
```

```
SomeStruct::c = 5;
// Accessing static member variable c in struct SomeStruct, through var and p.
var.a = var.c;
var.b = p->c;
// Calling a static member function.
SomeStruct::bar();
var.bar();
p->bar();
```

Background

The -> operator is needed because the member access operator . has precedence over the dereferencing operator *.

One would expect that *p.a would dereference p (resulting in a reference to the object p is pointing to) and then accessing its member a. But in fact, it tries to access the member a of p and then dereference it. I.e. *p.a is equivalent to *(p.a). In the example above, this would result in a compiler error because of two facts: First, p is a pointer and does not have a member a. Second, a is an integer and, thus, can't be dereferenced.

The uncommonly used solution to this problem would be to explicitly control the precedence: (*p).a

Instead, the -> operator is almost always used. It is a short-hand for first dereferencing the pointer and then accessing it. I.e. (*p).a is exactly the same as p->a.

The :: operator is the scope operator, used in the same manner as accessing a member of a namespace. This is because a static class member is considered to be in that class' scope, but isn't considered a member of instances of that class. The use of normal . and -> is also allowed for static members, despite them not being instance members, for historical reasons; this is of use for writing generic code in templates, as the caller doesn't need to be concerned with whether a given member function is static or non-static.

Section 34.9: Member Types and Aliases

A **class** or **struct** can also define member type aliases, which are type aliases contained within, and treated as members of, the class itself.

```
struct IHaveATypedef {
    typedef int MyTypedef;
};

struct IHaveATemplateTypedef {
    template<typename T>
    using MyTemplateTypedef = std::vector<T>;
};
```

Like static members, these typedefs are accessed using the scope operator, ::.

```
IHaveATypedef::MyTypedef i = 5; // i is an int.
```

```
IHaveATemplateTypedef::MyTemplateTypedef<int> v; // v is a std::vector<int>.
```

As with normal type aliases, each member type alias is allowed to refer to any type defined or aliased before, but not after, its definition. Likewise, a typedef outside the class definition can refer to any accessible typedefs within the class definition, provided it comes after the class definition.

```
template<typename T>
struct Helper {
```

```

T get() const { return static_cast<T>(42); }

struct IHaveTypedefs {
//    typedef MyTypedef NonLinearTypedef; // 如果取消注释，将报错。
    typedef int MyTypedef;
    typedef Helper<MyTypedef> MyTypedefHelper;
};

IHaveTypedefs::MyTypedef      i; // x_i 是一个 int 类型。
IHaveTypedefs::MyTypedefHelper hi; // x_hi 是一个 Helper<int> 类型。

typedef IHaveTypedefs::MyTypedef TypedefBeFree;
TypedefBeFree ii;           // ii 是一个 int 类型。

```

成员类型别名可以在任何访问级别声明，并将遵守相应的访问修饰符。

```

class TypedefAccessLevels {
    typedef int PrvInt;

protected:
    typedef int ProInt;

public:
    typedef int PubInt;
};

TypedefAccessLevels::PrvInt prv_i; // 错误：TypedefAccessLevels::PrvInt 是私有的。
TypedefAccessLevels::ProInt pro_i; // 错误：TypedefAccessLevels::ProInt 是受保护的。
TypedefAccessLevels::PubInt pub_i; // 正确。

class Derived : public TypedefAccessLevels {
    PrvInt prv_i; // 错误：TypedefAccessLevels::PrvInt 是私有的。
    ProInt pro_i; // 正确。
    PubInt pub_i; // 正确。
};

```

这可以用来提供一个抽象层，使类的设计者能够更改其内部实现而不破坏依赖该类的代码。

```

class Something {
    friend class SomeComplexType;

    short s;
    // ...

public:
    typedef SomeComplexType MyHelper;

MyHelper get_helper() const { return MyHelper(8, s, 19.5, "shoe", false); }

    // ...
};

// ...

Something s;
Something::MyHelper hlp = s.get_helper();

```

在这种情况下，如果辅助类从SomeComplexType更改为其他类型，则只需修改**typedef**和

```

T get() const { return static_cast<T>(42); }

struct IHaveTypedefs {
//    typedef MyTypedef NonLinearTypedef; // Error if uncommented.
    typedef int MyTypedef;
    typedef Helper<MyTypedef> MyTypedefHelper;
};

IHaveTypedefs::MyTypedef      i; // x_i is an int.
IHaveTypedefs::MyTypedefHelper hi; // x_hi is a Helper<int>.

typedef IHaveTypedefs::MyTypedef TypedefBeFree;
TypedefBeFree ii;           // ii is an int.

```

Member type aliases can be declared with any access level, and will respect the appropriate access modifier.

```

class TypedefAccessLevels {
    typedef int PrvInt;

protected:
    typedef int ProInt;

public:
    typedef int PubInt;
};

TypedefAccessLevels::PrvInt prv_i; // Error: TypedefAccessLevels::PrvInt is private.
TypedefAccessLevels::ProInt pro_i; // Error: TypedefAccessLevels::ProInt is protected.
TypedefAccessLevels::PubInt pub_i; // Good.

class Derived : public TypedefAccessLevels {
    PrvInt prv_i; // Error: TypedefAccessLevels::PrvInt is private.
    ProInt pro_i; // Good.
    PubInt pub_i; // Good.
};

```

This can be used to provide a level of abstraction, allowing a class' designer to change its internal workings without breaking code that relies on it.

```

class Something {
    friend class SomeComplexType;

    short s;
    // ...

public:
    typedef SomeComplexType MyHelper;

MyHelper get_helper() const { return MyHelper(8, s, 19.5, "shoe", false); }

    // ...
};

// ...

Something s;
Something::MyHelper hlp = s.get_helper();

```

In this situation, if the helper class is changed from SomeComplexType to some other type, only the **typedef** and the

friend声明；只要辅助类提供相同的功能，任何使用它作为Something::MyHelper而不是直接指定名称的代码通常仍然可以正常工作，无需修改。通过这种方式，当底层实现发生变化时，我们将需要修改的代码量降到最低，使得类型名称只需在一个位置更改。

如果需要，也可以将其与decltype结合使用。

```
class SomethingElse {  
    AnotherComplexType<bool, int, SomeThirdClass> helper;  
  
public:  
    typedef decltype(helper) MyHelper;  
  
private:  
    InternalVariable<MyHelper> ivh;  
  
    // ...  
  
public:  
    MyHelper& get_helper() const { return helper; }  
  
    // ...  
};
```

在这种情况下，改变SomethingElse::helper的实现会自动改变我们的typedef，归功于decltype。这减少了当我们想要更改helper时所需的修改次数，从而降低了人为错误的风险。

然而，和所有事情一样，这也可能被过度使用。例如，如果typename只在内部使用一两次，且外部根本不使用，就没有必要为它提供别名。如果它在整个项目中被使用数百或数千次，或者名称足够长，那么提供typedef会比总是使用绝对名称更有用。必须在向前兼容性和便利性与产生的不必要的噪音之间取得平衡。

这也可以用于模板类，以便从类外部访问模板参数。

```
template<typename T>  
class SomeClass {  
    // ...  
  
public:  
    typedef T MyParam;  
    MyParam getParam() { return static_cast<T>(42); }  
};  
  
template<typename T>  
typename T::MyParam some_func(T& t) {  
    return t.getParam();  
}  
  
SomeClass<int> si;  
int i = some_func(si);
```

这通常用于容器，容器通常会将其元素类型和其他辅助类型作为成员类型别名提供。例如，C++标准库中的大多数容器都会提供以下12种辅助类型，以及它们可能需要的其他特殊类型。

```
template<typename T>
```

friend declaration would need to be modified; as long as the helper class provides the same functionality, any code that uses it as Something::MyHelper instead of specifying it by name will usually still work without any modifications. In this manner, we minimise the amount of code that needs to be modified when the underlying implementation is changed, such that the type name only needs to be changed in one location.

This can also be combined with decltype, if one so desires.

```
class SomethingElse {  
    AnotherComplexType<bool, int, SomeThirdClass> helper;  
  
public:  
    typedef decltype(helper) MyHelper;  
  
private:  
    InternalVariable<MyHelper> ivh;  
  
    // ...  
  
public:  
    MyHelper& get_helper() const { return helper; }  
  
    // ...  
};
```

In this situation, changing the implementation of SomethingElse::helper will automatically change the typedef for us, due to decltype. This minimises the number of modifications necessary when we want to change helper, which minimises the risk of human error.

As with everything, however, this can be taken too far. If the typename is only used once or twice internally and zero times externally, for example, there's no need to provide an alias for it. If it's used hundreds or thousands of times throughout a project, or if it has a long enough name, then it can be useful to provide it as a typedef instead of always using it in absolute terms. One must balance forwards compatibility and convenience with the amount of unnecessary noise created.

This can also be used with template classes, to provide access to the template parameters from outside the class.

```
template<typename T>  
class SomeClass {  
    // ...  
  
public:  
    typedef T MyParam;  
    MyParam getParam() { return static_cast<T>(42); }  
};  
  
template<typename T>  
typename T::MyParam some_func(T& t) {  
    return t.getParam();  
}  
  
SomeClass<int> si;  
int i = some_func(si);
```

This is commonly used with containers, which will usually provide their element type, and other helper types, as member type aliases. Most of the containers in the C++ standard library, for example, provide the following 12 helper types, along with any other special types they might need.

```
template<typename T>
```

```

class SomeContainer {
    // ...
public:
    // 让我们提供与大多数标准容器相同的辅助类型。
    typedef T value_type;
    typedef std::allocator<value_type> allocator_type;
    typedef value_type& reference;
    typedef const value_type& const_reference;
    typedef value_type* pointer;
    typedef const value_type* const_pointer;
    typedef MyIterator<value_type> iterator;
    typedef MyConstIterator<value_type> const_iterator;
    typedef std::reverse_iterator<iterator> reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
};

```

在 C++11 之前，它也常用于提供某种“模板 `typedef`”，因为当时该特性尚不可用；随着别名模板的引入，这种用法变得不那么常见，但在某些情况下仍然有用（并且在其他情况下与别名模板结合使用，对于获取复杂类型的单个组成部分，如函数指针，非常有用）。它们通常使用名称 `type` 作为类型别名。

```

template<typename T>
struct TemplateTypedef {
    typedef T type;
}

```

`TemplateTypedef<int>::type i;` // `i` 是一个 `int` 类型。

这通常用于具有多个模板参数的类型，以提供定义一个或多个参数的别名。

```

template<typename T, size_t SZ, size_t D>
class Array { /* ... */ };

template<typename T, size_t SZ>
struct OneDArray {
    typedef Array<T, SZ, 1> type;
};

template<typename T, size_t SZ>
struct TwoDArray {
    typedef Array<T, SZ, 2> type;
};

template<typename T>
struct MonoDisplayLine {
    typedef Array<T, 80, 1> type;
};

```

`OneDArray<int, 3>::type arr1i;` // `arr1i` 是一个 `Array<int, 3, 1>`。
`TwoDArray<short, 5>::type arr2s;` // `arr2s` 是一个 `Array<short, 5, 2>`。
`MonoDisplayLine<char>::type arr3c;` // `arr3c` 是一个 `Array<char, 80, 1>`。

```

class SomeContainer {
    // ...
public:
    // Let's provide the same helper types as most standard containers.
    typedef T value_type;
    typedef std::allocator<value_type> allocator_type;
    typedef value_type& reference;
    typedef const value_type& const_reference;
    typedef value_type* pointer;
    typedef const value_type* const_pointer;
    typedef MyIterator<value_type> iterator;
    typedef MyConstIterator<value_type> const_iterator;
    typedef std::reverse_iterator<iterator> reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
};

```

Prior to C++11, it was also commonly used to provide a "template `typedef`" of sorts, as the feature wasn't yet available; these have become a bit less common with the introduction of alias templates, but are still useful in some situations (and are combined with alias templates in other situations, which can be very useful for obtaining individual components of a complex type such as a function pointer). They commonly use the name `type` for their type alias.

```

template<typename T>
struct TemplateTypedef {
    typedef T type;
}

```

`TemplateTypedef<int>::type i;` // `i` is an `int`.

This was often used with types with multiple template parameters, to provide an alias that defines one or more of the parameters.

```

template<typename T, size_t SZ, size_t D>
class Array { /* ... */ };

template<typename T, size_t SZ>
struct OneDArray {
    typedef Array<T, SZ, 1> type;
};

template<typename T, size_t SZ>
struct TwoDArray {
    typedef Array<T, SZ, 2> type;
};

template<typename T>
struct MonoDisplayLine {
    typedef Array<T, 80, 1> type;
};

```

`OneDArray<int, 3>::type arr1i;` // `arr1i` is an `Array<int, 3, 1>`.
`TwoDArray<short, 5>::type arr2s;` // `arr2s` is an `Array<short, 5, 2>`.
`MonoDisplayLine<char>::type arr3c;` // `arr3c` is an `Array<char, 80, 1>`.

第34.10节：嵌套类/结构体

一个类或结构体也可以在自身内部包含另一个类/结构体定义，这称为“嵌套类”；在这种情况下，包含类被称为“外部类”。嵌套类定义被视为外部类的成员，但除此之外是独立的。

```
结构体 Outer {  
    结构体 Inner { };  
};
```

从外部访问外部类时，嵌套类使用作用域运算符访问。然而，从外部类内部，嵌套类可以不带限定符直接使用：

```
结构体 Outer {  
    结构体 Inner { };  
  
    Inner in;  
};  
  
// ...  
  
Outer o;  
Outer::Inner i = o.in;
```

与非嵌套的类/结构体一样，成员函数和静态变量可以在嵌套类内部定义，也可以在外部命名空间中定义。但是，不能在外部类中定义它们，因为外部类被视为与嵌套类不同的类。

```
// 错误示例。  
结构体 Outer {  
    结构体 Inner {  
        void do_something();  
    };  
  
    void Inner::do_something() {}  
};  
  
// 好的。  
结构体 Outer {  
    结构体 Inner {  
        void do_something();  
    };  
  
    void Outer::Inner::do_something() {}  
};
```

与非嵌套类一样，嵌套类可以先前向声明，稍后定义，前提是它们必须在被直接使用之前定义。

```
类 Outer {  
    类 Inner1;  
    类 Inner2;  
  
    类 Inner1 {};  
  
    Inner1 in1;
```

Section 34.10: Nested Classes/Structures

A `class` or `struct` can also contain another `class/struct` definition inside itself, which is called a "nested class"; in this situation, the containing class is referred to as the "enclosing class". The nested class definition is considered to be a member of the enclosing class, but is otherwise separate.

```
struct Outer {  
    struct Inner { };  
};
```

From outside of the enclosing class, nested classes are accessed using the scope operator. From inside the enclosing class, however, nested classes can be used without qualifiers:

```
struct Outer {  
    struct Inner { };  
  
    Inner in;  
};  
  
// ...  
  
Outer o;  
Outer::Inner i = o.in;
```

As with a non-nested `class/struct`, member functions and static variables can be defined either within a nested class, or in the enclosing namespace. However, they cannot be defined within the enclosing class, due to it being considered to be a different class than the nested class.

```
// Bad.  
struct Outer {  
    struct Inner {  
        void do_something();  
    };  
  
    void Inner::do_something() {}  
};  
  
// Good.  
struct Outer {  
    struct Inner {  
        void do_something();  
    };  
  
    void Outer::Inner::do_something() {}  
};
```

As with non-nested classes, nested classes can be forward declared and defined later, provided they are defined before being used directly.

```
class Outer {  
    class Inner1;  
    class Inner2;  
  
    class Inner1 {};  
  
    Inner1 in1;
```

```
Inner2* in2p;
```

```
public:  
Outer();  
~Outer();  
};  
  
class Outer::Inner2 {};  
  
Outer::Outer() : in1(Inner1()), in2p(new Inner2) {}  
Outer::~Outer() {  
    if (in2p) { delete in2p; }  
}
```

版本 < C++11

在C++11之前，嵌套类只能访问封闭类的类型名、static成员和枚举量；封闭类中定义的所有其他成员都是不可访问的。

版本 ≥ C++11

从C++11开始，嵌套类及其成员被视为封闭类的friend，可以根据通常的访问规则访问其所有成员；如果嵌套类的成员需要访问封闭类的一个或多个非静态成员，则必须传递一个实例：

```
class Outer {  
    结构体 Inner {  
        int get_sizeof_x() {  
            return sizeof(x); // 合法 (C++11) :x是未求值的，因此不需要实例。  
        }  
  
        int get_x() {  
            return x; // 非法：没有实例，无法访问非静态成员。  
        }  
  
        int get_x(Outer& o) {  
            return o.x; // 合法 (C++11) :作为Outer的成员，Inner可以访问私有成员。  
        }  
    };  
  
    int x;  
};
```

相反，封闭类不被视为嵌套类的友元，因此未经明确授权，不能访问其私有成员。

```
class Outer {  
    class Inner {  
        // friend class Outer;  
  
        int x;  
    };  
  
    Inner in;  
  
public:  
    int get_x() {  
        return in.x; // 错误：int Outer::Inner::x 是私有的。  
        // 取消注释上面的“friend”行以修复。  
    }
```

```
Inner2* in2p;
```

```
public:  
Outer();  
~Outer();  
};  
  
class Outer::Inner2 {};  
  
Outer::Outer() : in1(Inner1()), in2p(new Inner2) {}  
Outer::~Outer() {  
    if (in2p) { delete in2p; }  
}
```

Version < C++11

Prior to C++11, nested classes only had access to type names, static members, and enumerators from the enclosing class; all other members defined in the enclosing class were off-limits.

Version ≥ C++11

As of C++11, nested classes, and members thereof, are treated as if they were friends of the enclosing class, and can access all of its members, according to the usual access rules; if members of the nested class require the ability to evaluate one or more non-static members of the enclosing class, they must therefore be passed an instance:

```
class Outer {  
    struct Inner {  
        int get_sizeof_x() {  
            return sizeof(x); // Legal (C++11): x is unevaluated, so no instance is required.  
        }  
  
        int get_x() {  
            return x; // Illegal: Can't access non-static member without an instance.  
        }  
  
        int get_x(Outer& o) {  
            return o.x; // Legal (C++11): As a member of Outer, Inner can access private members.  
        }  
    };  
  
    int x;  
};
```

Conversely, the enclosing class is not treated as a friend of the nested class, and thus cannot access its private members without explicitly being granted permission.

```
class Outer {  
    class Inner {  
        // friend class Outer;  
  
        int x;  
    };  
  
    Inner in;  
  
public:  
    int get_x() {  
        return in.x; // Error: int Outer::Inner::x is private.  
        // Uncomment "friend" line above to fix.  
    }
```

```
};
```

嵌套类的友元不会自动被视为封闭类的友元；如果它们也需要成为封闭类的友元，则必须单独声明。反之，封闭类不会自动被视为嵌套类的友元，封闭类的友元也不会被视为嵌套类的友元。

```
class Outer {
    friend void barge_out(Outer& out, Inner& in);

    class Inner {
        friend void barge_in(Outer& out, Inner& in);

        int i;
    };

    int o;
};

void barge_in(Outer& out, Outer::Inner& in) {
    int i = in.i; // 正确。
    int o = out.o; // 错误：int Outer::o 是私有的。
}

void barge_out(Outer& out, Outer::Inner& in) {
    int i = in.i; // 错误：int Outer::Inner::i 是私有的。
    int o = out.o; // 正确。
}
```

与所有其他类成员一样，嵌套类只有在具有公共访问权限时，才能从类外部被命名访问。然而，只要不显式命名它们，你可以不受访问修饰符限制地访问它们。

```
class Outer {
    结构体 Inner {
        void func() { std::cout << "我没有私人禁忌。"; };

        public:
            static Inner make_Inner() { return Inner(); }
    };

    // ...

    Outer::Inner oi; // 错误：Outer::Inner 是私有的。

    auto oi = Outer::make_Inner(); // 正确。
    oi.func(); // 正确。
    Outer::make_Inner().func(); // 正确。
}
```

你也可以为嵌套类创建类型别名。如果类型别名包含在封闭类中，嵌套类型和类型别名可以有不同的访问修饰符。如果类型别名在封闭类外部，则要求嵌套类或其`typedef`必须是公共的。

```
class Outer {
    class Inner_ {};

    public:
        typedef Inner_ Inner;
```

```
};
```

Friends of a nested class are not automatically considered friends of the enclosing class; if they need to be friends of the enclosing class as well, this must be declared separately. Conversely, as the enclosing class is not automatically considered a friend of the nested class, neither will friends of the enclosing class be considered friends of the nested class.

```
class Outer {
    friend void barge_out(Outer& out, Inner& in);

    class Inner {
        friend void barge_in(Outer& out, Inner& in);

        int i;
    };

    int o;
};

void barge_in(Outer& out, Outer::Inner& in) {
    int i = in.i; // Good.
    int o = out.o; // Error: int Outer::o is private.
}

void barge_out(Outer& out, Outer::Inner& in) {
    int i = in.i; // Error: int Outer::Inner::i is private.
    int o = out.o; // Good.
}
```

As with all other class members, nested classes can only be named from outside the class if they have public access. However, you are allowed to access them regardless of access modifier, as long as you don't explicitly name them.

```
class Outer {
    struct Inner {
        void func() { std::cout << "I have no private taboo.\n"; }
    };

    public:
        static Inner make_Inner() { return Inner(); }
    };

    // ...

    Outer::Inner oi; // Error: Outer::Inner is private.

    auto oi = Outer::make_Inner(); // Good.
    oi.func(); // Good.
    Outer::make_Inner().func(); // Good.
```

You can also create a type alias for a nested class. If a type alias is contained in the enclosing class, the nested type and the type alias can have different access modifiers. If the type alias is outside the enclosing class, it requires that either the nested class, or a `typedef` thereof, be public.

```
class Outer {
    class Inner_ {};

    public:
        typedef Inner_ Inner;
```

```
};

typedef Outer::Inner ImOut; // 正确。
typedef Outer::Inner_ ImBad; // 错误。

// ...

Outer::Inner oi; // 正确。
Outer::Inner_ oi; // 错误。
ImOut      oi; // 正确。
```

与其他类一样，嵌套类既可以从其他类继承，也可以被其他类继承。

```
struct Base {};

结构体 Outer {
    struct Inner : Base {};
};

struct Derived : Outer::Inner {};
```

这在包含类被另一个类继承的情况下非常有用，允许程序员根据需要更新嵌套类。可以结合typedef为每个包含类的嵌套类提供一致的名称：

```
class BaseOuter {
    struct BaseInner_ {
        virtual void do_something() {}
        virtual void do_something_else();
    } b_in;

public:
    typedef BaseInner_ Inner;

    virtual ~BaseOuter() = default;

    virtual Inner& getInner() { return b_in; }
};

void BaseOuter::BaseInner_::do_something_else() {}

// ---


class DerivedOuter : public BaseOuter {
    // 注意限定typedef的使用；BaseOuter::BaseInner_是私有的。
    struct DerivedInner_ : BaseOuter::Inner {
        void do_something() override {}
        void do_something_else() override;
    } d_in;

public:
    typedef DerivedInner_ Inner;

    BaseOuter::Inner& getInner() override { return d_in; }
};

void DerivedOuter::DerivedInner_::do_something_else() {}

// ...
```

```
};

typedef Outer::Inner ImOut; // Good.
typedef Outer::Inner_ ImBad; // Error.

// ...

Outer::Inner oi; // Good.
Outer::Inner_ oi; // Error.
ImOut      oi; // Good.
```

As with other classes, nested classes can both derive from or be derived from by other classes.

```
struct Base {};

struct Outer {
    struct Inner : Base {};
};

struct Derived : Outer::Inner {};
```

This can be useful in situations where the enclosing class is derived from by another class, by allowing the programmer to update the nested class as necessary. This can be combined with a typedef to provide a consistent name for each enclosing class' nested class:

```
class BaseOuter {
    struct BaseInner_ {
        virtual void do_something() {}
        virtual void do_something_else();
    } b_in;

public:
    typedef BaseInner_ Inner;

    virtual ~BaseOuter() = default;

    virtual Inner& getInner() { return b_in; }
};

void BaseOuter::BaseInner_::do_something_else() {}

// ---


class DerivedOuter : public BaseOuter {
    // Note the use of the qualified typedef; BaseOuter::BaseInner_ is private.
    struct DerivedInner_ : BaseOuter::Inner {
        void do_something() override {}
        void do_something_else() override;
    } d_in;

public:
    typedef DerivedInner_ Inner;

    BaseOuter::Inner& getInner() override { return d_in; }
};

void DerivedOuter::DerivedInner_::do_something_else() {}

// ...
```

```
// 调用 BaseOuter::BaseInner_::do_something();
BaseOuter* b = new BaseOuter;
BaseOuter::Inner& bin = b->getInner();
bin.do_something();
b->getInner().do_something();

// 调用 DerivedOuter::DerivedInner_::do_something();
BaseOuter* d = new DerivedOuter;
BaseOuter::Inner& din = d->getInner();
din.do_something();
d->getInner().do_something();
```

在上述情况下，`BaseOuter` 和 `DerivedOuter` 都提供成员类型 `Inner`，分别作为 `BaseInner_` 和 `DerivedInner_`。这允许嵌套类型被派生而不破坏封闭类的接口，并允许嵌套类型以多态方式使用。

第34.11节：无名结构体/类

允许无名结构体（类型无名称）

```
void foo()
{
    struct /* 无名 */
    {
        float x;
        float y;
    } point;

    point.x = 42;
}
```

或者

```
结构体 圆
{
    struct /* 无名 */
    {
        float x;
        float y;
    } center; // 但这是成员名
    float radius;
};
```

之后

```
圆 circle;
circle.center.x = 42.f;
```

但不是匿名结构体（无名类型和无名对象）

```
结构体 无效圆
{
    结构体 /* 无名 */
    {
        float centerX;
        float centerY;
    }; // 也没有成员名。
    float radius;
};
```

注意：某些编译器允许将匿名结构体作为扩展。

```
// Calls BaseOuter::BaseInner_::do_something();
BaseOuter* b = new BaseOuter;
BaseOuter::Inner& bin = b->getInner();
bin.do_something();
b->getInner().do_something();

// Calls DerivedOuter::DerivedInner_::do_something();
BaseOuter* d = new DerivedOuter;
BaseOuter::Inner& din = d->getInner();
din.do_something();
d->getInner().do_something();
```

In the above case, both `BaseOuter` and `DerivedOuter` supply the member type `Inner`, as `BaseInner_` and `DerivedInner_`, respectively. This allows nested types to be derived without breaking the enclosing class' interface, and allows the nested type to be used polymorphically.

Section 34.11: Unnamed struct/class

Unnamed struct is allowed (type has no name)

```
void foo()
{
    struct /* No name */
    {
        float x;
        float y;
    } point;

    point.x = 42;
}
```

or

```
struct Circle
{
    struct /* No name */
    {
        float x;
        float y;
    } center; // but a member name
    float radius;
};
```

and later

```
Circle circle;
circle.center.x = 42.f;
```

but NOT *anonymous struct* (unnamed type and unnamed object)

```
struct InvalidCircle
{
    struct /* No name */
    {
        float centerX;
        float centerY;
    }; // No member either.
    float radius;
};
```

Note: Some compilers allow *anonymous struct* as extension.

- lambda可以看作是一种特殊的无名结构体。

- decltype允许获取无名结构体的类型：

```
decltype(circle.point) otherPoint;
```

- 无名结构体实例可以作为模板方法的参数：

```
void print_square_coordinates()
{
    const struct {float x; float y;} points[] = {
        {-1, -1}, {-1, 1}, {1, -1}, {1, 1}
    };

    // 范围基于 `template <class T, std::size_t N> std::begin(T (&)[N])`
    for (const auto& point : points) {
        std::cout << "(" << point.x << ", " << point.y << ")";
    }

    decltype(points[0]) topRightCorner{1, 1}; auto it =
        std::find(points, points + 4, topRightCorner); std::cout << "右上角
是第 " << 1 + std::distance(points, it) << " 个";
}
```

第34.12节：静态类成员

一个类也允许有静态成员，这些成员可以是变量或函数。它们被视为属于类的作用域，但不被当作普通成员；它们具有静态存储期（从程序开始到结束一直存在），不依赖于类的特定实例，并且整个类只有一份拷贝。

```
class 示例 {
    static int 实例数量;      // 静态数据成员（静态成员变量）。
    int i;                   // 非静态成员变量。

public:
    static std::string 静态字符串; // 静态数据成员（静态成员变量）。
    static int 静态函数();       // 静态成员函数。

    // 非静态成员函数可以修改静态成员变量。
    示例() { ++实例数量; }
    void 设置字符串(const std::string& 字符串);
};

int      示例::实例数量;
std::string 示例::静态字符串 = "Hello./";

// ...

示例一, 二, 三;
// 每个示例都有自己的“i”，因此：
// (&one.i != &two.i)
// (&one.i != &three.i)
// (&two.i != &three.i).
// 所有三个示例共享“num_instances”，因此：
```

- *lambda* can be seen as a special *unnamed struct*.

- *decltype* allows to retrieve the type of *unnamed struct*:

```
decltype(circle.point) otherPoint;
```

- *unnamed struct* instance can be parameter of template method:

```
void print_square_coordinates()
{
    const struct {float x; float y;} points[] = {
        {-1, -1}, {-1, 1}, {1, -1}, {1, 1}
    };

    // for range relies on `template <class T, std::size_t N> std::begin(T (&)[N])`
    for (const auto& point : points) {
        std::cout << "(" << point.x << ", " << point.y << ")\n";
    }

    decltype(points[0]) topRightCorner{1, 1};
    auto it = std::find(points, points + 4, topRightCorner);
    std::cout << "top right corner is the "
        << 1 + std::distance(points, it) << "th\n";
}
```

Section 34.12: Static class members

A class is also allowed to have `static` members, which can be either variables or functions. These are considered to be in the class' scope, but aren't treated as normal members; they have static storage duration (they exist from the start of the program to the end), aren't tied to a particular instance of the class, and only one copy exists for the entire class.

```
class Example {
    static int num_instances;      // Static data member (static member variable).
    int i;                      // Non-static member variable.

public:
    static std::string static_str; // Static data member (static member variable).
    static int static_func();     // Static member function.

    // Non-static member functions can modify static member variables.
    Example() { ++num_instances; }
    void set_str(const std::string& str);
};

int      Example::num_instances;
std::string Example::static_str = "Hello./";

// ...

示例 one, two, three;
// Each Example has its own "i", such that:
// (&one.i != &two.i)
// (&one.i != &three.i)
// (&two.i != &three.i).
// All three Examples share "num_instances", such that:
```

```
// (&one.num_instances == &two.num_instances)  
// (&one.num_instances == &three.num_instances)  
// (&two.num_instances == &three.num_instances)
```

静态成员变量不被视为在类内定义，仅被声明，因此它们的定义在类定义之外；程序员可以但不必在定义时初始化静态变量。定义成员变量时，省略关键字static。

```
class 示例 {  
    static int num_instances; // 声明。  
  
public:  
    static std::string static_str; // 声明。  
  
    // ...  
};  
  
int Example::num_instances; // 定义。零初始化。  
std::string Example::static_str = "Hello."; // 定义。
```

因此，静态变量可以是不完整类型（除void外），只要它们之后被定义为完整类型。

```
结构体 ForwardDeclared;  
  
类 ExIncomplete {  
    static ForwardDeclared fd;  
    static ExIncomplete i_contain_myself;  
    static int an_array[];  
};  
  
struct ForwardDeclared {};  
  
ForwardDeclared ExIncomplete::fd;  
ExIncomplete ExIncomplete::i_contain_myself;  
int ExIncomplete::an_array[5];
```

静态成员函数可以像普通成员函数一样在类定义内或外定义。与静态成员变量一样，在类定义外定义静态成员函数时，省略关键字static。

```
// 以上示例中，任意一种方式都可以...  
class Example {  
    // ...  
  
public:  
    static int static_func() { return num_instances; }  
  
    // ...  
  
    void set_str(const std::string& str) { static_str = str; }  
};  
  
// 或者...  
  
class Example { /* ... */ };  
  
int Example::static_func() { return num_instances; }
```

```
// (&one.num_instances == &two.num_instances)  
// (&one.num_instances == &three.num_instances)  
// (&two.num_instances == &three.num_instances)
```

Static member variables are not considered to be defined inside the class, only declared, and thus have their definition outside the class definition; the programmer is allowed, but not required, to initialise static variables in their definition. When defining the member variables, the keyword **static** is omitted.

```
class Example {  
    static int num_instances; // Declaration.  
  
public:  
    static std::string static_str; // Declaration.  
  
    // ...  
};  
  
int Example::num_instances; // Definition. Zero-initialised.  
std::string Example::static_str = "Hello."; // Definition.
```

Due to this, static variables can be incomplete types (apart from **void**), as long as they're later defined as a complete type.

```
struct ForwardDeclared;  
  
class ExIncomplete {  
    static ForwardDeclared fd;  
    static ExIncomplete i_contain_myself;  
    static int an_array[];  
};  
  
struct ForwardDeclared {};  
  
ForwardDeclared ExIncomplete::fd;  
ExIncomplete ExIncomplete::i_contain_myself;  
int ExIncomplete::an_array[5];
```

Static member functions can be defined inside or outside the class definition, as with normal member functions. As with static member variables, the keyword **static** is omitted when defining static member functions outside the class definition.

```
// For Example above, either...  
class Example {  
    // ...  
  
public:  
    static int static_func() { return num_instances; }  
  
    // ...  
  
    void set_str(const std::string& str) { static_str = str; }  
};  
  
// Or...  
  
class Example { /* ... */ };  
  
int Example::static_func() { return num_instances; }
```

```
void Example::set_str(const std::string& str) { static_str = str; }
```

如果静态成员变量被声明为const但不是volatile，并且是整型或枚举类型，则可以在类定义内部的声明处进行初始化。

```
enum E { VAL = 5 };
```

```
struct ExConst {
    const static int ci = 5;           // 好的。
    static const E ce = VAL;          // 好的。
    const static double cd = 5;        // 错误。
    static const volatile int cvi = 5; // 错误。

    const static double good_cd;
    static const volatile int good_cvi;
};
```

```
const double ExConst::good_cd = 5;      // 好的。
const volatile int ExConst::good_cvi = 5; // 好的。
```

版本 ≥ C++11

截至 C++11，LiteralType 类型（根据 `constexpr` 规则，可以在编译时构造的类型）的静态成员变量也可以声明为 `constexpr`；如果是这样，它们必须在类定义内进行初始化。

```
struct ExConstexpr {
    constexpr static int ci = 5;           // 正确。
    static constexpr double cd = 5;          // 正确。
    constexpr static int carr[] = { 1, 1, 2 }; // 正确。
    static constexpr ConstexprConstructibleClass c{}; // 正确。
    constexpr static int bad_ci;            // 错误。
};
```

```
constexpr int ExConstexpr::bad_ci = 5;      // 仍然是错误。
```

如果 `const` 或 `constexpr` 静态成员变量被 odr-used（非正式地说，如果它被取地址或赋给引用），那么它仍然必须在类定义外有一个单独的定义。该定义不允许包含初始化器。

```
struct ExODR {
    static const int odr_used = 5;
};
```

```
// const int ExODR::odr_used;
```

```
const int* odr_user = & ExODR::odr_used; // 错误；取消注释上面一行以解决。
```

由于静态成员不依赖于特定实例，因此可以使用作用域运算符`::`访问。

```
std::string str = Example::static_str;
```

它们也可以像普通的非静态成员一样访问。这具有历史意义，但比起作用域运算符，使用频率较低，以避免混淆成员是静态还是非静态。

```
Example ex;
std::string rts = ex.static_str;
```

```
void Example::set_str(const std::string& str) { static_str = str; }
```

If a static member variable is declared `const` but not `volatile`, and is of an integral or enumeration type, it can be initialised at declaration, inside the class definition.

```
enum E { VAL = 5 };
```

```
struct ExConst {
    const static int ci = 5;           // Good.
    static const E ce = VAL;          // Good.
    const static double cd = 5;        // Error.
    static const volatile int cvi = 5; // Error.

    const static double good_cd;
    static const volatile int good_cvi;
};
```

```
const double ExConst::good_cd = 5;      // Good.
const volatile int ExConst::good_cvi = 5; // Good.
```

Version ≥ C++11

As of C++11, static member variables of LiteralType types (types that can be constructed at compile time, according to `constexpr` rules) can also be declared as `constexpr`; if so, they must be initialised within the class definition.

```
struct ExConstexpr {
    constexpr static int ci = 5;           // Good.
    static constexpr double cd = 5;          // Good.
    constexpr static int carr[] = { 1, 1, 2 }; // Good.
    static constexpr ConstexprConstructibleClass c{}; // Good.
    constexpr static int bad_ci;            // Error.
};
```

```
constexpr int ExConstexpr::bad_ci = 5;      // Still an error.
```

If a `const` or `constexpr` static member variable is odr-used (informally, if it has its address taken or is assigned to a reference), then it must still have a separate definition, outside the class definition. This definition is not allowed to contain an initialiser.

```
struct ExODR {
    static const int odr_used = 5;
};

// const int ExODR::odr_used;

const int* odr_user = & ExODR::odr_used; // Error; uncomment above line to resolve.
```

As static members aren't tied to a given instance, they can be accessed using the scope operator, `::`.

```
std::string str = Example::static_str;
```

They can also be accessed as if they were normal, non-static members. This is of historical significance, but is used less commonly than the scope operator to prevent confusion over whether a member is static or non-static.

```
Example ex;
std::string rts = ex.static_str;
```

类成员能够像访问非静态类成员一样访问静态成员，无需限定作用域。

```
class ExTwo {
    static int num_instances;
    int my_num;

public:
ExTwo() : my_num(num_instances++) {}

    static int get_total_instances() { return num_instances; }
    int get_instance_number() const { return my_num; }
};

int ExTwo::num_instances;
```

它们不能是mutable，也不需要是；因为它们不依赖于任何特定实例，实例是否为const并不影响静态成员。

```
struct ExDontNeedMutable {
    int immuta;
    mutable int muta;

    static int i;

ExDontNeedMutable() : immuta(-5), muta(-5) {}
};

int ExDontNeedMutable::i;

// ...

const ExDontNeedMutable dnm;
dnm.immuta = 5; // 错误：无法修改只读对象。
dnm.muta = 5; // 正确。const对象的可变字段可以被写入。
dnm.i = 5; // 正确。静态成员无论实例是否为const都可以被写入。
```

静态成员遵守访问修饰符，就像非静态成员一样。

```
class ExAccess {
    static int prv_int;

protected:
    static int pro_int;

public:
    static int pub_int;
};

int ExAccess::prv_int;
int ExAccess::pro_int;
int ExAccess::pub_int;

// ...

int x1 = ExAccess::prv_int; // 错误：int ExAccess::prv_int 是私有的。
int x2 = ExAccess::pro_int; // 错误：int ExAccess::pro_int 是受保护的。
int x3 = ExAccess::pub_int; // 好的。
```

由于它们不绑定到特定实例，静态成员函数没有this指针；因此，除非传入实例，否则它们无法访问非静态成员变量。

Class members are able to access static members without qualifying their scope, as with non-static class members.

```
class ExTwo {
    static int num_instances;
    int my_num;

public:
ExTwo() : my_num(num_instances++) {}

    static int get_total_instances() { return num_instances; }
    int get_instance_number() const { return my_num; }
};

int ExTwo::num_instances;
```

They cannot be `mutable`, nor would they need to be; as they aren't tied to any given instance, whether an instance is or isn't const doesn't affect static members.

```
struct ExDontNeedMutable {
    int immuta;
    mutable int muta;

    static int i;

ExDontNeedMutable() : immuta(-5), muta(-5) {}
};

int ExDontNeedMutable::i;

// ...

const ExDontNeedMutable dnm;
dnm.immuta = 5; // Error: Can't modify read-only object.
dnm.muta = 5; // Good. Mutable fields of const objects can be written.
dnm.i = 5; // Good. Static members can be written regardless of an instance's const-ness.
```

Static members respect access modifiers, just like non-static members.

```
class ExAccess {
    static int prv_int;

protected:
    static int pro_int;

public:
    static int pub_int;
};

int ExAccess::prv_int;
int ExAccess::pro_int;
int ExAccess::pub_int;

// ...

int x1 = ExAccess::prv_int; // Error: int ExAccess::prv_int is private.
int x2 = ExAccess::pro_int; // Error: int ExAccess::pro_int is protected.
int x3 = ExAccess::pub_int; // Good.
```

As they aren't tied to a given instance, static member functions have no `this` pointer; due to this, they can't access non-static member variables unless passed an instance.

```

类 ExInstanceRequired {
    int i;

public:
ExInstanceRequired() : i(0) {}

    static void bad_mutate() { ++i *= 5; }           // 错误。
    static void good_mutate(ExInstanceRequired& e) { ++e.i *= 5; } // 正确。
};

```

由于没有this指针，它们的地址不能存储在指向成员函数的指针中，而是存储在普通的指向函数的指针中。

```

结构体 ExPointer {
    void nsfunc() {}
    static void sfunc() {}
};

typedef void (ExPointer::* mem_f_ptr)();
typedef void (*f_ptr)();

mem_f_ptr p_sf = &ExPointer::sfunc; // 错误。
f_ptr p_sf = &ExPointer::sfunc; // 正确。

```

由于没有this指针，它们也不能是const或volatile，也不能有引用限定符。它们也不能是虚函数。

```

struct ExCVQualifiersAndVirtual {
    static void func() {} // 正确。
    static void cfunc() const {} // 错误。
    static void vfunc() volatile {} // 错误。
    static void cvfunc() const volatile {} // 错误。
    static void rfunc() & {} // 错误。
    static void rvfunc() && {} // 错误。

    virtual static void vsfunc() {} // 错误。
    static virtual void svfunc() {} // 错误。
};

```

由于它们不绑定到特定实例，静态成员变量实际上被视为特殊的全局变量；它们在程序启动时创建，程序退出时销毁，无论类的任何实例是否存在。

每个静态成员变量只有一份副本（除非该变量被声明为thread_local（C++11或更高版本），在这种情况下每个线程有一份副本）。

静态成员变量与类具有相同的链接属性，无论类是外部链接还是内部链接。局部类和无名类不允许有静态成员。

第34.13节：多重继承

除了单继承之外：

```

class A {};
class B : public A {};

```

你也可以使用多重继承：

```

class A {};

```

```

class ExInstanceRequired {
    int i;

public:
ExInstanceRequired() : i(0) {}

    static void bad_mutate() { ++i *= 5; }           // Error.
    static void good_mutate(ExInstanceRequired& e) { ++e.i *= 5; } // Good.
};

```

Due to not having a `this` pointer, their addresses can't be stored in pointers-to-member-functions, and are instead stored in normal pointers-to-functions.

```

struct ExPointer {
    void nsfunc() {}
    static void sfunc() {}
};

typedef void (ExPointer::* mem_f_ptr)();
typedef void (*f_ptr)();

mem_f_ptr p_sf = &ExPointer::sfunc; // Error.
f_ptr p_sf = &ExPointer::sfunc; // Good.

```

Due to not having a `this` pointer, they also cannot be `const` or `volatile`, nor can they have ref-qualifiers. They also cannot be `virtual`.

```

struct ExCVQualifiersAndVirtual {
    static void func() {} // Good.
    static void cfunc() const {} // Error.
    static void vfunc() volatile {} // Error.
    static void cvfunc() const volatile {} // Error.
    static void rfunc() & {} // Error.
    static void rvfunc() && {} // Error.

    virtual static void vsfunc() {} // Error.
    static virtual void svfunc() {} // Error.
};

```

As they aren't tied to a given instance, static member variables are effectively treated as special global variables; they're created when the program starts, and destroyed when it exits, regardless of whether any instances of the class actually exist. Only a single copy of each static member variable exists (unless the variable is declared `thread_local` (C++11 or later), in which case there's one copy per thread).

Static member variables have the same linkage as the class, whether the class has external or internal linkage. Local classes and unnamed classes aren't allowed to have static members.

Section 34.13: Multiple Inheritance

Aside from single inheritance:

```

class A {};
class B : public A {};

```

You can also have multiple inheritance:

```

class A {};

```

```
class B {};
class C : public A, public B {};
```

C现在将同时继承自A和B。

注意：如果多个继承的class或struct中使用了相同的名称，可能会导致歧义。请小心！

多重继承中的歧义

多重继承在某些情况下可能有用，但有时在使用

多重继承时会遇到一些奇怪的问题。

例如：两个基类中有同名函数，且派生类未重写该函数，如果你使用派生类对象调用该函数，编译器会报错，因为它无法确定调用哪个函数。以下是一个关于多重继承中此类歧义的代码示例。

```
class base1
{
public:
    void function( )
    { // base1函数的代码 }
};

class base2
{
    void function( )
    { // base2 函数的代码 }
};

class 派生类 : 公有 基类1, 公有 基类2
{

};

int main()
{
    派生类 对象;

    // 错误，因为编译器无法确定调用哪个函数
    // 是基类1的 function() 还是基类2的 function()。
    对象.function( )
}
```

但是，这个问题可以通过使用作用域解析符来指定调用基类1还是基类2的函数来解决：

```
int main()
{
    对象.基类1::function( ); // 调用基类1的函数。
    对象.基类2::function( ); // 调用基类2的函数。
}
```

第34.14节：非静态成员函数

一个类可以有非静态成员函数，这些函数作用于类的各个实例。

```
class CL {
public:
    void 成员函数() {}
```

```
class B {};
class C : public A, public B {};
```

C will now have inherit from A and B at the same time.

Note: this can lead to ambiguity if the same names are used in multiple inherited classes or structs. Be careful!

Ambiguity in Multiple Inheritance

Multiple inheritance may be helpful in certain cases but, sometimes odd sort of problem encounters while using multiple inheritance.

For example: Two base classes have functions with same name which is not overridden in derived class and if you write code to access that function using object of derived class, compiler shows error because, it cannot determine which function to call. Here is a code for this type of ambiguity in multiple inheritance.

```
class base1
{
public:
    void function( )
    { //code for base1 function }
};

class base2
{
    void function( )
    { // code for base2 function }
};

class derived : public base1, public base2
{

};

int main()
{
    derived obj;

    // Error because compiler can't figure out which function to call
    // either function( ) of base1 or base2 .
    obj.function( )
}
```

But, this problem can be solved using scope resolution function to specify which function to class either base1 or base2:

```
int main()
{
    obj.base1::function( ); // Function of class base1 is called.
    obj.base2::function( ); // Function of class base2 is called.
}
```

Section 34.14: Non-static member functions

A class can have non-static member functions, which operate on individual instances of the class.

```
class CL {
public:
    void member_function() {}
```

```
};
```

这些函数是在类的实例上调用的，方式如下：

```
CL 实例;  
实例.成员函数();
```

它们可以在类定义内部或外部定义；如果在外部定义，则需指定其属于类的作用域。

```
结构体 ST {  
    void 在内部定义() {}  
    void 在外部定义();  
};  
void ST::在外部定义() {}
```

它们可以带有 CV 限定符和/或引用限定符，影响函数如何看到被调用的实例；函数将把实例视为具有指定的 cv 限定符（如果有的话）。调用哪个版本取决于实例的 cv 限定符。如果没有与实例相同 cv 限定符的版本，则如果有更严格 cv 限定符的版本，将调用该版本。

```
struct CVQualifiers {  
    void func()           {} // 1: 实例无cv限定符。  
    void func() const     {} // 2: 实例为const。  
  
    void cv_only() const volatile {}  
};  
  
CVQualifiers      non_cv_instance;  
const CVQualifiers      c_instance;  
  
non_cv_instance.func(); // 调用 #1。  
c_instance.func();     // 调用 #2。  
  
non_cv_instance.cv_only(); // 调用 const volatile 版本。  
c_instance.cv_only();   // 调用 const volatile 版本。  
版本 ≥ C++11
```

成员函数的引用限定符指示函数是否打算在右值实例上调用，并使用与函数cv限定符相同的语法。

```
struct RefQualifiers {  
    void func() & {} // 1: 在普通实例上调用。  
    void func() && {} // 2: 在右值 (临时) 实例上调用。  
};  
  
RefQualifiers rf;  
rf.func();          // 调用 #1。  
RefQualifiers{}.func(); // 调用 #2。
```

如果需要，CV限定符和引用限定符也可以组合使用。

```
结构体 BothCVAndRef {  
    void func() const& {} // 在普通实例上调用。视实例为const。  
    void func() && {} // 在临时实例上调用。  
};
```

```
};
```

These functions are called on an instance of the class, like so:

```
CL instance;  
instance.member_function();
```

They can be defined either inside or outside the class definition; if defined outside, they are specified as being in the class' scope.

```
struct ST {  
    void defined_inside() {}  
    void defined_outside();  
};  
void ST::defined_outside() {}
```

They can be CV-qualified and/or ref-qualified, affecting how they see the instance they're called upon; the function will see the instance as having the specified cv-qualifier(s), if any. Which version is called will be based on the instance's cv-qualifiers. If there is no version with the same cv-qualifiers as the instance, then a more-cv-qualified version will be called if available.

```
struct CVQualifiers {  
    void func()           {} // 1: Instance is non-cv-qualified.  
    void func() const     {} // 2: Instance is const.  
  
    void cv_only() const volatile {}  
};  
  
CVQualifiers      non_cv_instance;  
const CVQualifiers      c_instance;  
  
non_cv_instance.func(); // Calls #1.  
c_instance.func();     // Calls #2.  
  
non_cv_instance.cv_only(); // Calls const volatile version.  
c_instance.cv_only();   // Calls const volatile version.  
Version ≥ C++11
```

Member function ref-qualifiers indicate whether or not the function is intended to be called on rvalue instances, and use the same syntax as function cv-qualifiers.

```
struct RefQualifiers {  
    void func() & {} // 1: Called on normal instances.  
    void func() && {} // 2: Called on rvalue (temporary) instances.  
};  
  
RefQualifiers rf;  
rf.func();          // Calls #1.  
RefQualifiers{}.func(); // Calls #2.
```

CV-qualifiers and ref-qualifiers can also be combined if necessary.

```
struct BothCVAndRef {  
    void func() const& {} // Called on normal instances. Sees instance as const.  
    void func() && {} // Called on temporary instances.  
};
```

它们也可以是虚函数；这是多态的基础，允许子类提供与父类相同的接口，同时提供自己的功能。

```
结构体 Base {  
    virtual void func() {}  
};  
struct Derived {  
    virtual void func() {}  
};  
  
Base* bp = new Base;  
Base* dp = new Derived;  
bp.func(); // 调用 Base::func().  
dp.func(); // 调用 Derived::func().
```

更多信息，请参见此处。

They can also be virtual; this is fundamental to polymorphism, and allows a child class(es) to provide the same interface as the parent class, while supplying their own functionality.

```
struct Base {  
    virtual void func() {}  
};  
struct Derived {  
    virtual void func() {}  
};  
  
Base* bp = new Base;  
Base* dp = new Derived;  
bp.func(); // Calls Base::func().  
dp.func(); // Calls Derived::func().
```

For more information, see here.

第35章：函数重载

另见关于重载解析的单独主题

第35.1节：什么是函数重载？

函数重载是在同一作用域（称为作用域）中声明多个同名函数，唯一不同的是它们的签名，即它们接受的参数。

假设你正在编写一系列用于通用打印功能的函数，从std::string开始：

```
void print(const std::string &str)
{
    std::cout << "这是一个字符串: " << str << std::endl;
}
```

这可以正常工作，但假设你想要一个函数也接受一个int并打印它。你可以这样写：

```
void print_int(int num)
{
    std::cout << "这是一个整数: " << num << std::endl;
}
```

但是因为这两个函数接受不同的参数，你可以简单地写成：

```
void print(int num)
{
    std::cout << "这是一个整数: " << num << std::endl;
}
```

现在你有两个函数，名字都叫print，但签名不同。一个接受std::string，另一个接受int。现在你可以调用它们而不用担心不同的名字：

```
print("Hello world!"); //打印 "这是一个字符串: Hello world!"
print(1337);           //打印 "这是一个整数: 1337"
```

而不是：

```
print("Hello world!");
print_int(1337);
```

当你有重载函数时，编译器会根据你提供的参数推断调用哪个函数。编写函数重载时必须小心。例如，涉及隐式类型转换时：

```
void print(int num)
{
    std::cout << "这是一个整数: " << num << std::endl;
}
void print(double num)
{
    std::cout << "这是一个 double 类型: " << num << std::endl;
}
```

现在，当你写下以下代码时，哪个重载的print被调用并不立即清楚：

Chapter 35: Function Overloading

See also separate topic on Overload Resolution

Section 35.1: What is Function Overloading?

Function overloading is having multiple functions declared in the same scope with the exact same name exist in the same place (known as *scope*) differing only in their *signature*, meaning the arguments they accept.

Suppose you are writing a series of functions for generalized printing capabilities, beginning with std::string:

```
void print(const std::string &str)
{
    std::cout << "This is a string: " << str << std::endl;
}
```

This works fine, but suppose you want a function that also accepts an int and prints that too. You could write:

```
void print_int(int num)
{
    std::cout << "This is an int: " << num << std::endl;
}
```

But because the two functions accept different parameters, you can simply write:

```
void print(int num)
{
    std::cout << "This is an int: " << num << std::endl;
}
```

Now you have 2 functions, both named print, but with different signatures. One accepts std::string, the other one an int. Now you can call them without worrying about different names:

```
print("Hello world!"); //prints "This is a string: Hello world!"
print(1337);           //prints "This is an int: 1337"
```

Instead of:

```
print("Hello world!");
print_int(1337);
```

When you have overloaded functions, the compiler infers which of the functions to call from the parameters you provide it. Care must be taken when writing function overloads. For example, with implicit type conversions:

```
void print(int num)
{
    std::cout << "This is an int: " << num << std::endl;
}
void print(double num)
{
    std::cout << "This is a double: " << num << std::endl;
}
```

Now it's not immediately clear which overload of print is called when you write:

```
print(5);
```

你可能需要给编译器一些提示，比如：

```
print(static_cast<double>(5));
print(static_cast<int>(5));
print(5.0);
```

编写接受可选参数的重载函数时也需要注意：

```
// 错误代码
void print(int num1, int num2 = 0) // 如果未传入, num2 默认为 0
{
    std::cout << "这些是整数: " << num1 << " 和 " << num2 << std::endl;
}
void print(int num)
{
    std::cout << "这是一个 整数: " << num << std::endl;
}
```

因为编译器无法判断像 `print(17)` 这样的调用是针对第一个函数还是第二个函数
由于第二个参数是可选的，这将导致编译失败。

第35.2节：函数重载中的返回类型

注意，不能仅根据返回类型来重载函数。例如：

```
// 错误代码
std::string getValue()
{
    return "hello";
}

int getValue()
{
    return 0;
}

int x = getValue();
```

这将导致编译错误，因为编译器无法确定调用哪个版本的`getValue`，尽管返回类型被指定为`int`。

第35.3节：成员函数的cv限定符重载

类中的函数可以针对通过cv限定的该类引用访问时进行重载；这通常用于针对`const`进行重载，但也可以用于针对`volatile`和`const volatile`进行重载。这是因为所有非静态成员函数都将`this`作为隐藏参数，而cv限定符正是应用于该参数的。这通常用于针对`const`进行重载，但也可以用于`volatile`和`const volatile`。

这是必要的，因为成员函数只有在其cv限定符至少与调用它的实例的cv限定符相同或更严格时才能被调用。虽然非`const`实例可以调用`const`和非`const`成员，但`const`实例只能调用`const`成员。这允许函数根据调用实例的cv限定符表现出不同的行为，并允许程序员通过不提供带有某些限定符的版本来禁止函数被不希望的cv限定符调用。

```
print(5);
```

And you might need to give your compiler some clues, like:

```
print(static_cast<double>(5));
print(static_cast<int>(5));
print(5.0);
```

Some care also needs to be taken when writing overloads that accept optional parameters:

```
// WRONG CODE
void print(int num1, int num2 = 0) // num2 defaults to 0 if not included
{
    std::cout << "These are ints: " << num1 << " and " << num2 << std::endl;
}
void print(int num)
{
    std::cout << "This is an int: " << num << std::endl;
}
```

Because there's no way for the compiler to tell if a call like `print(17)` is meant for the first or second function
because of the optional second parameter, this will fail to compile.

Section 35.2: Return Type in Function Overloading

Note that you cannot overload a function based on its return type. For example:

```
// WRONG CODE
std::string getValue()
{
    return "hello";
}

int getValue()
{
    return 0;
}

int x = getValue();
```

This will cause a compilation error as the compiler will not be able to work out which version of `getValue` to call,
even though the return type is assigned to an `int`.

Section 35.3: Member Function cv-qualifier Overloading

Functions within a class can be overloaded for when they are accessed through a cv-qualified reference to that class; this is most commonly used to overload for `const`, but can be used to overload for `volatile` and `const volatile`, too. This is because all non-static member functions take `this` as a hidden parameter, which the cv-qualifiers are applied to. This is most commonly used to overload for `const`, but can also be used for `volatile` and `const volatile`.

This is necessary because a member function can only be called if it is at least as cv-qualified as the instance it's called on. While a non-`const` instance can call both `const` and non-`const` members, a `const` instance can only call `const` members. This allows a function to have different behaviour depending on the calling instance's cv-qualifiers, and allows the programmer to disallow functions for an undesired cv-qualifier(s) by not providing a version with that qualifier(s).

一个带有基本print方法的类可以这样进行const重载：

```
#include <iostream>

class Integer
{
public:
    Integer(int i_) : i{i_} {}

    void print()
    {
        std::cout << "int: " << i << std::endl;
    }

    void print() const
    {
        std::cout << "const int: " << i << std::endl;
    }

protected:
    int i;
};

int main()
{
    Integer i{5};
    const Integer &ic = i;

    i.print(); // 输出 "int: 5"
    ic.print(); // 输出 "const int: 5"
}
```

这是const正确性的一个关键原则：通过将成员函数标记为const，允许它们在const实例上被调用，这反过来又允许函数在不需要修改实例时，将实例作为const指针/引用传递。这使得代码可以通过将未修改的参数设为const，而将修改的参数不带cv限定符，来明确是否修改状态，从而使代码更安全、更易读。

```
class ConstCorrect
{
public:
    void good_func() const
    {
        std::cout << "我不在乎实例是否为const。" << std::endl;
    }

    void bad_func()
    {
        std::cout << "我只能在非const、非volatile实例上被调用。" << std::endl;
    }
};

void i_change_no_state(const ConstCorrect& cc)
{
    std::cout << "我可以接受 const 或非 const 的 ConstCorrect。" << std::endl;
    cc.good_func(); // 好的。可以从 const 或非 const 实例调用。
    cc.bad_func(); // 错误。只能从非常量实例调用。
}

void const_incorrect_func(ConstCorrect& cc)
{
```

A class with some basic print method could be const overloaded like so:

```
#include <iostream>

class Integer
{
public:
    Integer(int i_) : i{i_} {}

    void print()
    {
        std::cout << "int: " << i << std::endl;
    }

    void print() const
    {
        std::cout << "const int: " << i << std::endl;
    }

protected:
    int i;
};

int main()
{
    Integer i{5};
    const Integer &ic = i;

    i.print(); // prints "int: 5"
    ic.print(); // prints "const int: 5"
}
```

This is a key tenet of const correctness: By marking member functions as const, they are allowed to be called on const instances, which in turn allows functions to take instances as const pointers/references if they don't need to modify them. This allows code to specify whether it modifies state by taking unmodified parameters as const and modified parameters without cv-qualifiers, making code both safer and more readable.

```
class ConstCorrect
{
public:
    void good_func() const
    {
        std::cout << "I care not whether the instance is const." << std::endl;
    }

    void bad_func()
    {
        std::cout << "I can only be called on non-const, non-volatile instances." << std::endl;
    }
};

void i_change_no_state(const ConstCorrect& cc)
{
    std::cout << "I can take either a const or a non-const ConstCorrect." << std::endl;
    cc.good_func(); // Good. Can be called from const or non-const instance.
    cc.bad_func(); // Error. Can only be called from non-const instance.
}

void const_incorrect_func(ConstCorrect& cc)
{
```

```
cc.good_func(); // 正确。可以从常量或非常量实例调用。  
cc.bad_func(); // 错误。只能从非常量实例调用。  
}
```

一个常见的用法是将访问器声明为`const`, 将修改器声明为非`const`。

在`const`成员函数中不能修改任何类成员。如果确实需要修改某些成员, 比如锁定一个`std::mutex`, 可以将其声明为`mutable`:

```
class Integer  
{  
public:  
    Integer(int i_): i{i_}{}  
  
    int get() const  
    {  
        std::lock_guard<std::mutex> lock{mut};  
        return i;  
    }  
  
    void set(int i_)  
    {  
        std::lock_guard<std::mutex> lock{mut};  
        i = i_;  
    }  
  
protected:  
    int i;  
    mutable std::mutex mut;  
};
```

```
cc.good_func(); // Good. Can be called from const or non-const instance.  
cc.bad_func(); // Good. Can only be called from non-const instance.  
}
```

A common usage of this is declaring accessors as `const`, and mutators as non-`const`.

No class members can be modified within a `const` member function. If there is some member that you really need to modify, such as locking a `std::mutex`, you can declare it as `mutable`:

```
class Integer  
{  
public:  
    Integer(int i_): i{i_}{}  
  
    int get() const  
    {  
        std::lock_guard<std::mutex> lock{mut};  
        return i;  
    }  
  
    void set(int i_)  
    {  
        std::lock_guard<std::mutex> lock{mut};  
        i = i_;  
    }  
  
protected:  
    int i;  
    mutable std::mutex mut;  
};
```

第36章：运算符重载

在C++中，可以为用户自定义类型定义诸如+和-等运算符。例如，`<string>`头文件定义了一个+运算符用于字符串连接。这是通过使用operator关键字定义一个operator函数来实现的。

第36.1节：算术运算符

你可以重载所有基本的算术运算符：

- + 和 +=
- - 和 -=
- * 和 *=
- / 和 /=
- & 和 &=
- | 和 |=
- ^ 和 ^=
- >> 和 >>=
- << 和 <<=

所有运算符的重载方式相同。 向下滚动查看说明

类/结构体外的重载：

```
//operator+ 应该基于 operator+= 实现
T operator+(T lhs, const T& rhs)
{
    lhs += rhs;
    return lhs;
}

T& operator+=(T& lhs, const T& rhs)
{
    //执行加法
    return lhs;
}
```

类/结构体内的重载：

```
//operator+ 应该基于 operator+= 实现
T operator+(const T& rhs)
{
    *this += rhs;
    return *this;
}

T& operator+=(const T& rhs)
{
    //执行加法
    return *this;
}
```

注意：operator+ 应该返回非 const 值，因为返回引用没有意义（它返回一个new 对象），返回const值也不合适（通常不应返回const）。第一个参数是按值传递的，为什么？

Chapter 36: Operator Overloading

In C++, it is possible to define operators such as + and -> for user-defined types. For example, the `<string>` header defines a + operator to concatenate strings. This is done by defining an *operator function* using the operator keyword.

Section 36.1: Arithmetic operators

You can overload all basic arithmetic operators:

- + and +=
- - and -=
- * and *=
- / and /=
- & and &=
- | and |=
- ^ and ^=
- >> and >>=
- << and <<=

Overloading for all operators is the same. Scroll down for explanation

Overloading outside of `class/struct`:

```
//operator+ should be implemented in terms of operator+=
T operator+(T lhs, const T& rhs)
{
    lhs += rhs;
    return lhs;
}

T& operator+=(T& lhs, const T& rhs)
{
    //Perform addition
    return lhs;
}
```

Overloading inside of `class/struct`:

```
//operator+ should be implemented in terms of operator+=
T operator+(const T& rhs)
{
    *this += rhs;
    return *this;
}

T& operator+=(const T& rhs)
{
    //Perform addition
    return *this;
}
```

Note: operator+ should return by non-const value, as returning a reference wouldn't make sense (it returns a new object) nor would returning a const value (you should generally not return by const). The first argument is passed

因为

1. 你不能修改原始对象（对象 `foobar = foo + bar;` 之后不应该修改 `foo`, 毕竟这没有意义）
这没有意义)
2. 你不能将其设为 `const`, 因为你必须能够修改对象（因为 `operator+=` 是通过 `operator+=` 实现的，而 `operator+=` 会修改对象）

通过 `const&` 传递是一个选项，但这样你将不得不制作传入对象的临时副本。通过按值传递，编译器会为你完成这一步。

`operator+=` 返回自身的引用，因为这样可以链式调用（但不要使用同一个变量，否则由于序列点的原因会导致未定义行为）。

第一个参数是引用（我们想要修改它），但不是 `const`，因为那样你将无法修改它。第二个参数不应被修改，因此出于性能考虑通过 `const&` 传递（通过 `const` 引用传递比值传递更快）。

第36.2节：数组下标运算符

你甚至可以重载数组下标运算符`[]`。

你应该总是（99.98%的情况下）实现两个版本，一个 `const` 版本和一个非 `const` 版本，因为如果对象是 `const`，它不应该能够修改`[]` 返回的对象。

参数通过 `const&` 传递而不是值传递，因为引用传递比值传递更快，且使用 `const` 以防止运算符意外修改索引。

运算符返回引用，因为设计上你可以修改`[]` 返回的对象，例如：

```
std::vector<int> v{ 1 };
v[0] = 2; //将值1改为2
//如果不是返回引用则无法实现
```

你只能在 `class/struct` 内部重载：

```
//I 是索引类型，通常是 int
T& operator[](const I& index)
{
    //执行某些操作
    //返回某些值
}

//I 是索引类型，通常是 int
const T& operator[](const I& index) const
{
    //执行某些操作
    //返回某些值
}
```

多个下标操作符，`[][]...`，可以通过代理对象实现。以下简单的行优先矩阵类示例演示了这一点：

```
template<class T>
```

by value, why? Because

1. You can't modify the original object (`Object foobar = foo + bar;` shouldn't modify `foo` after all, it wouldn't make sense)
2. You can't make it `const`, because you will have to be able to modify the object (because `operator+=` is implemented in terms of `operator+=`, which modifies the object)

Passing by `const&` would be an option, but then you will have to make a temporary copy of the passed object. By passing by value, the compiler does it for you.

`operator+=` returns a reference to the itself, because it is then possible to chain them (don't use the same variable though, that would be undefined behavior due to sequence points).

The first argument is a reference (we want to modify it), but not `const`, because then you wouldn't be able to modify it. The second argument should not be modified, and so for performance reason is passed by `const&` (passing by `const` reference is faster than by value).

Section 36.2: Array subscript operator

You can even overload the array subscript operator `[]`.

You should **always** (99.98% of the time) implement 2 versions, a `const` and a not-`const` version, because if the object is `const`, it should not be able to modify the object returned by `[]`.

The arguments are passed by `const&` instead of by value because passing by reference is faster than by value, and `const` so that the operator doesn't change the index accidentally.

The operators return by reference, because by design you can modify the object `[]` return, i.e:

```
std::vector<int> v{ 1 };
v[0] = 2; //Changes value of 1 to 2
//wouldn't be possible if not returned by reference
```

You can **only** overload inside a `class/struct`:

```
//I is the index type, normally an int
T& operator[](const I& index)
{
    //Do something
    //return something
}

//I is the index type, normally an int
const T& operator[](const I& index) const
{
    //Do something
    //return something
}
```

Multiple subscript operators, `[][]...`, can be achieved via proxy objects. The following example of a simple row-major matrix class demonstrates this:

```
template<class T>
```

```

class matrix {
    // 使能 [][] 重载以访问矩阵元素的类
    template <class C>
    class proxy_row_vector {
        using reference = decltype(std::declval<C>()[0]);
        using const_reference = decltype(std::declval<C const>()[0]);
    public:
        proxy_row_vector(C& _vec, std::size_t _r_ind, std::size_t _cols)
            : vec(_vec), row_index(_r_ind), cols(_cols) {}
        const_reference operator[](std::size_t _col_index) const {
            return vec[row_index*cols + _col_index];
        }
        reference operator[](std::size_t _col_index) {
            return vec[row_index*cols + _col_index];
        }
    private:
        C& vec;
        std::size_t row_index; // 访问的行索引
        std::size_t cols; // 矩阵的列数
    };

    using const_proxy = proxy_row_vector<const std::vector<T>>;
    using proxy = proxy_row_vector<std::vector<T>>;
public:
    matrix() : mtx(), rows(0), cols(0) {}
    matrix(std::size_t _rows, std::size_t _cols)
        : mtx(_rows*_cols), rows(_rows), cols(_cols) {}

    // 调用 operator[] 后再调用另一个 [] 来访问矩阵元素
    const_proxy operator[](std::size_t _row_index) const {
        return const_proxy(mtx, _row_index, cols);
    }

    proxy operator[](std::size_t _row_index) {
        return proxy(mtx, _row_index, cols);
    }
private:
    std::vector<T> mtx;
    std::size_t rows;
    std::size_t cols;
};

```

第36.3节：转换运算符

你可以重载类型运算符，使你的类型能够隐式转换为指定类型。

转换运算符必须定义在类/结构体中：

```
operator T() const { /* 返回某些内容 */ }
```

注意：该运算符是`const`的，以允许`const`对象被转换。

示例：

```

结构体 Text
{
    std::string text;

    // 现在 Text 可以隐式转换为 const char*
    /*explicit*/ operator const char*() const { return text.data(); }

```

```

class matrix {
    // class enabling [][] overload to access matrix elements
    template <class C>
    class proxy_row_vector {
        using reference = decltype(std::declval<C>()[0]);
        using const_reference = decltype(std::declval<C const>()[0]);
    public:
        proxy_row_vector(C& _vec, std::size_t _r_ind, std::size_t _cols)
            : vec(_vec), row_index(_r_ind), cols(_cols) {}
        const_reference operator[](std::size_t _col_index) const {
            return vec[row_index*cols + _col_index];
        }
        reference operator[](std::size_t _col_index) {
            return vec[row_index*cols + _col_index];
        }
    private:
        C& vec;
        std::size_t row_index; // row index to access
        std::size_t cols; // number of columns in matrix
    };

    using const_proxy = proxy_row_vector<const std::vector<T>>;
    using proxy = proxy_row_vector<std::vector<T>>;
public:
    matrix() : mtx(), rows(0), cols(0) {}
    matrix(std::size_t _rows, std::size_t _cols)
        : mtx(_rows*_cols), rows(_rows), cols(_cols) {}

    // call operator[] followed by another [] call to access matrix elements
    const_proxy operator[](std::size_t _row_index) const {
        return const_proxy(mtx, _row_index, cols);
    }

    proxy operator[](std::size_t _row_index) {
        return proxy(mtx, _row_index, cols);
    }
private:
    std::vector<T> mtx;
    std::size_t rows;
    std::size_t cols;
};

```

Section 36.3: Conversion operators

You can overload type operators, so that your type can be implicitly converted into the specified type.

The conversion operator **must** be defined in a `class/struct`:

```
operator T() const { /* return something */ }
```

Note: the operator is `const` to allow `const` objects to be converted.

Example:

```

struct Text
{
    std::string text;

    // Now Text can be implicitly converted into a const char*
    /*explicit*/ operator const char*() const { return text.data(); }

```

```

// ^^^^^^
// 禁用隐式转换
};

文本 t;
t.text = "Hello world!";

//Ok
const char* copyoftext = t;

```

第36.4节：复数重访

下面的代码实现了一个非常简单的复数类型，其底层域会根据语言的类型提升规则，在应用四个基本运算符（+、-、* 和 /）与不同域的成员（无论是另一个complex<T>还是某个标量类型）时自动提升。

这旨在作为一个整体示例，涵盖运算符重载以及模板的基本使用。

```

#include <type_traits>

namespace not_std{

using std::decay_t;

//-----
// complex< value_t >
//-----

template<typename value_t>
结构体 复数
{
value_t x;
value_t y;

复数 &运算符 += (常量 value_t &x)
{
this->x += x;
    return *this;
}
复数 &运算符 += (常量 复数 &other)
{
this->x += other.x;
    this->y += other.y;
    返回 *this;
}

复数 &运算符 -= (常量 value_t &x)
{
this->x -= x;
    return *this;
}
复数 &运算符 -= (常量 复数 &other)
{
this->x -= other.x;
    this->y -= other.y;
    return *this;
}

complex &operator *= (const value_t &s)
{

```

```

// ^^^^^^
// to disable implicit conversion
};

Text t;
t.text = "Hello world!";

//Ok
const char* copyoftext = t;

```

Section 36.4: Complex Numbers Revisited

The code below implements a very simple complex number type for which the underlying field is automatically promoted, following the language's type promotion rules, under application of the four basic operators (+, -, *, and /) with a member of a different field (be it another `complex<T>` or some scalar type).

This is intended to be a holistic example covering operator overloading alongside basic use of templates.

```

#include <type_traits>

namespace not_std{

using std::decay_t;

//-----
// complex< value_t >
//-----

template<typename value_t>
struct complex
{
    value_t x;
    value_t y;

    complex &operator += (const value_t &x)
    {
        this->x += x;
        return *this;
    }
    complex &operator += (const complex &other)
    {
        this->x += other.x;
        this->y += other.y;
        return *this;
    }

    complex &operator -= (const value_t &x)
    {
        this->x -= x;
        return *this;
    }
    complex &operator -= (const complex &other)
    {
        this->x -= other.x;
        this->y -= other.y;
        return *this;
    }

    complex &operator *= (const value_t &s)
    {

```

```

this->x *= s;
this->y *= s;
    return *this;
}
complex &operator *= (const complex &other)
{
    (*this) = (*this) * other;
    return *this;
}

complex &operator /= (const value_t &s)
{
this->x /= s;
this->y /= s;
    return *this;
}
complex &operator /= (const complex &other)
{
    (*this) = (*this) / other;
    return *this;
}

complex(const value_t &x, const value_t &y)
: x{x}
, y{y}
{ }

template<typename other_value_t>
explicit complex(const complex<other_value_t> &other)
: x{static_cast<const value_t &>(other.x)}
, y{static_cast<const value_t &>(other.y)}
{ }

complex &operator = (const complex &) = default;
complex &operator = (complex &&) = default;
complex(const complex &) = default;
complex(complex &&) = default;
complex() = default;
};

// 绝对值的平方
template<typename value_t>
value_t absqr(const complex<value_t> &z)
{ return z.x*z.x + z.y*z.y; }

//-----
// 运算符 - (取反)
//-----


template<typename value_t>
complex<value_t> operator - (const complex<value_t> &z)
{ return {-z.x, -z.y}; }

//-----
// 运算符 +
//-----


template<typename left_t,typename right_t>
auto operator + (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x + b.x)>>
{ return{a.x + b.x, a.y + b.y}; }

```

```

this->x *= s;
this->y *= s;
    return *this;
}
complex &operator *= (const complex &other)
{
    (*this) = (*this) * other;
    return *this;
}

complex &operator /= (const value_t &s)
{
this->x /= s;
this->y /= s;
    return *this;
}
complex &operator /= (const complex &other)
{
    (*this) = (*this) / other;
    return *this;
}

complex(const value_t &x, const value_t &y)
: x{x}
, y{y}
{ }

template<typename other_value_t>
explicit complex(const complex<other_value_t> &other)
: x{static_cast<const value_t &>(other.x)}
, y{static_cast<const value_t &>(other.y)}
{ }

complex &operator = (const complex &) = default;
complex &operator = (complex &&) = default;
complex(const complex &) = default;
complex(complex &&) = default;
complex() = default;
};

// Absolute value squared
template<typename value_t>
value_t absqr(const complex<value_t> &z)
{ return z.x*z.x + z.y*z.y; }

//-----
// operator - (negation)
//-----


template<typename value_t>
complex<value_t> operator - (const complex<value_t> &z)
{ return {-z.x, -z.y}; }

//-----
// operator +
//-----


template<typename left_t,typename right_t>
auto operator + (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x + b.x)>>
{ return{a.x + b.x, a.y + b.y}; }

```

```

template<typename left_t,typename right_t>
auto operator + (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a + b.x)>>
{ return{a + b.x, b.y}; }

template<typename left_t,typename right_t>
auto operator + (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x + b)>>
{ return{a.x + b, a.y}; }

//-----
// operator -
//-----

template<typename left_t,typename right_t>
auto operator - (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x - b.x)>>
{ return { a.x - b.x, a.y - b.y }; }

template<typename left_t, typename right_t>
auto operator - (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a - b.x)>>
{ return { a - b.x, -b.y }; }

template<typename left_t, typename right_t>
auto operator - (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x - b)>>
{ return{a.x - b, a.y}; }

//-----
// operator *
//-----

template<typename left_t, typename right_t>
auto operator * (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x * b.x)>>;
{
    return {
        a.x*b.x - a.y*b.y,
        a.x*b.y + a.y*b.x
    };
}

template<typename left_t, typename right_t>
auto operator * (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a * b.x)>>
{ return {a * b.x, a * b.y}; }

template<typename left_t, typename right_t>
auto operator * (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x * b)>>
{ return {a.x * b, a.y * b}; }

//-----
// operator /
//-----

template<typename left_t, typename right_t>
auto operator / (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x / b.x)>>;
{
    const auto r = absqr(b);

```

```

template<typename left_t,typename right_t>
auto operator + (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a + b.x)>>
{ return{a + b.x, b.y}; }

template<typename left_t,typename right_t>
auto operator + (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x + b)>>
{ return{a.x + b, a.y}; }

//-----
// operator -
//-----

template<typename left_t,typename right_t>
auto operator - (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x - b.x)>>
{ return{a.x - b.x, a.y - b.y}; }

template<typename left_t, typename right_t>
auto operator - (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a - b.x)>>
{ return{a - b.x, -b.y}; }

template<typename left_t, typename right_t>
auto operator - (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x - b)>>
{ return{a.x - b, a.y}; }

//-----
// operator *
//-----

template<typename left_t, typename right_t>
auto operator * (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x * b.x)>>;
{
    return {
        a.x*b.x - a.y*b.y,
        a.x*b.y + a.y*b.x
    };
}

template<typename left_t, typename right_t>
auto operator * (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a * b.x)>>
{ return {a * b.x, a * b.y}; }

template<typename left_t, typename right_t>
auto operator * (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x * b)>>
{ return {a.x * b, a.y * b}; }

//-----
// operator /
//-----

template<typename left_t, typename right_t>
auto operator / (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x / b.x)>>;
{
    const auto r = absqr(b);

```

```

return {
    ( a.x*b.x + a.y*b.y ) / r,
    (-a.x*b.y + a.y*b.x) / r
};

template<typename left_t, typename right_t>
auto operator / (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a / b.x)>>
{
    const auto s = a/absqr(b);
    return {
        b.x * s,
        -b.y * s
    };
}

template<typename left_t, typename right_t>
auto operator / (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x / b)>>
{ return {a.x / b, a.y / b}; }

}// namespace not_std

int main(int argc, char **argv)
{
    using namespace not_std;

complex<float> fz{4.0f, 1.0f};

// 生成一个 complex<double>
auto dz = fz * 1.0;

// 仍然是 complex<double>
auto idz = 1.0f/dz;

// 也是 complex<double>
auto one = dz * idz;

// 又是一个 complex<double>
auto one_again = fz * idz;

// 运算符测试，仅确保所有代码能编译通过。

complex<float> a{1.0f, -2.0f};
complex<double> b{3.0, -4.0};

// 这些都是 complex<double>
auto c0 = a + b;
auto c1 = a - b;
auto c2 = a * b;
auto c3 = a / b;

// 这些都是 complex<float>
auto d0 = a + 1;
auto d1 = 1 + a;
auto d2 = a - 1;
auto d3 = 1 - a;
auto d4 = a * 1;
auto d5 = 1 * a;
auto d6 = a / 1;

```

```

return {
    ( a.x*b.x + a.y*b.y ) / r,
    (-a.x*b.y + a.y*b.x) / r
};

template<typename left_t, typename right_t>
auto operator / (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a / b.x)>>
{
    const auto s = a/absqr(b);
    return {
        b.x * s,
        -b.y * s
    };
}

template<typename left_t, typename right_t>
auto operator / (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x / b)>>
{ return {a.x / b, a.y / b}; }

}// namespace not_std

int main(int argc, char **argv)
{
    using namespace not_std;

complex<float> fz{4.0f, 1.0f};

// makes a complex<double>
auto dz = fz * 1.0;

// still a complex<double>
auto idz = 1.0f/dz;

// also a complex<double>
auto one = dz * idz;

// a complex<double> again
auto one_again = fz * idz;

// Operator tests, just to make sure everything compiles.

complex<float> a{1.0f, -2.0f};
complex<double> b{3.0, -4.0};

// All of these are complex<double>
auto c0 = a + b;
auto c1 = a - b;
auto c2 = a * b;
auto c3 = a / b;

// All of these are complex<float>
auto d0 = a + 1;
auto d1 = 1 + a;
auto d2 = a - 1;
auto d3 = 1 - a;
auto d4 = a * 1;
auto d5 = 1 * a;
auto d6 = a / 1;

```

```

auto d7 = 1 / a;

// 这些都是 complex<double>
auto e0 = b + 1;
auto e1 = 1 + b;
auto e2 = b - 1;
auto e3 = 1 - b;
auto e4 = b * 1;
auto e5 = 1 * b;
auto e6 = b / 1;
auto e7 = 1 / b;

return 0;
}

```

第36.5节：命名运算符

你可以通过标准C++运算符“引用”的方式扩展C++，添加命名运算符。

首先我们从一个十几行的库开始：

```

namespace named_operator {
    template<class D>struct make_operator{constexpr make_operator(){}};

    template<class T, char, class O> struct half_apply { T&& lhs; };

    template<class Lhs, class Op>
    half_apply<Lhs, '*', Op> operator*( Lhs&& lhs, make_operator<Op> ) {
        return {std::forward<Lhs>(lhs)};
    }

    template<class Lhs, class Op, class Rhs>
    auto operator*( half_apply<Lhs, '*', Op>&& lhs, Rhs&& rhs )
    -> decltype( named_invoke( std::forward<Lhs>(lhs), Op{}, std::forward<Rhs>(rhs) ) )
    {
        return named_invoke( std::forward<Lhs>(lhs), Op{}, std::forward<Rhs>(rhs) );
    }
}

```

这还什么都没做。

首先，追加向量

```

namespace my_ns {
    struct append_t : named_operator::make_operator<append_t> {};
    constexpr append_t append{};

    template<class T, class A0, class A1>
    std::vector<T, A0> named_invoke( std::vector<T, A0> lhs, append_t, std::vector<T, A1> const& rhs
    ) {
        lhs.insert( lhs.end(), rhs.begin(), rhs.end() );
        return std::move(lhs);
    }
}

using my_ns::append;

std::vector<int> a {1,2,3};
std::vector<int> b {4,5,6};

```

```
auto d7 = 1 / a;
```

```

// All of these are complex<double>
auto e0 = b + 1;
auto e1 = 1 + b;
auto e2 = b - 1;
auto e3 = 1 - b;
auto e4 = b * 1;
auto e5 = 1 * b;
auto e6 = b / 1;
auto e7 = 1 / b;

return 0;
}

```

Section 36.5: Named operators

You can extend C++ with named operators that are "quoted" by standard C++ operators.

First we start with a dozen-line library:

```

namespace named_operator {
    template<class D>struct make_operator{constexpr make_operator(){}};

    template<class T, char, class O> struct half_apply { T&& lhs; };

    template<class Lhs, class Op>
    half_apply<Lhs, '*', Op> operator*( Lhs&& lhs, make_operator<Op> ) {
        return {std::forward<Lhs>(lhs)};
    }

    template<class Lhs, class Op, class Rhs>
    auto operator*( half_apply<Lhs, '*', Op>&& lhs, Rhs&& rhs )
    -> decltype( named_invoke( std::forward<Lhs>(lhs), Op{}, std::forward<Rhs>(rhs) ) )
    {
        return named_invoke( std::forward<Lhs>(lhs), Op{}, std::forward<Rhs>(rhs) );
    }
}

```

this doesn't do anything yet.

First, appending vectors

```

namespace my_ns {
    struct append_t : named_operator::make_operator<append_t> {};
    constexpr append_t append{};

    template<class T, class A0, class A1>
    std::vector<T, A0> named_invoke( std::vector<T, A0> lhs, append_t, std::vector<T, A1> const& rhs
    ) {
        lhs.insert( lhs.end(), rhs.begin(), rhs.end() );
        return std::move(lhs);
    }
}

using my_ns::append;

std::vector<int> a {1,2,3};
std::vector<int> b {4,5,6};

```

```
auto c = a *append* b;
```

这里的*核心*是我们定义了一个类型为append_t的append对象：named_operator::make_operator<append_t>。

然后我们为想要的左右类型重载了named_invoke(lhs, append_t, rhs)函数。

该库重载了lhs*append_t，返回一个临时的half_apply对象。它还重载了half_apply*rhs以调用named_invoke(lhs, append_t, rhs)。

我们只需创建合适的append_t标记，并进行一个符合ADL规则的正确签名的named_invoke调用，所有功能就会连接并正常工作。

举一个更复杂的例子，假设你想对std::array的元素进行逐元素相乘：

```
template<class=void, std::size_t... Is>
auto indexer( std::index_sequence<Is...> ) {
    return [](&auto&& f) {
        return f( std::integral_constant<std::size_t, Is>{}... );
    };
}
template<std::size_t N>
auto indexer() { return indexer( std::make_index_sequence<N>{} ); }

namespace my_ns {
    struct e_times_t : named_operator::make_operator<e_times_t> {};
    constexpr e_times_t e_times{};

    template<class L, class R, std::size_t N,
             class Out=std::decay_t<decltype( std::declval<L const&>()*std::declval<R const&>() )>
    >
    std::array<Out, N> named_invoke( std::array<L, N> const& lhs, e_times_t, std::array<R, N> const& rhs ) {
        using result_type = std::array<Out, N>;
        auto index_over_N = indexer<N>();
        return index_over_N([&](&auto... is)->result_type {
            return {{
                (lhs[is] * rhs[is])...
            }};
        });
    }
}
```

实时示例。

此元素级数组代码可以扩展以处理元组、对或C风格数组，甚至可变长度的容器，前提是决定当长度不匹配时如何处理。

你也可以定义一个元素级操作符类型，得到 lhs *element_wise<'+*>* rhs。

编写 *dot* 和 *cross* 乘积操作符也是显而易见的用途。

* 的使用可以扩展以支持其他分隔符，比如 +。分隔符的优先级决定了命名操作符的优先级，这在将物理方程转换为C++时，尽量减少额外的 () 使用时可能很重要。

通过对上述库稍作修改，我们可以支持 ->*then* 操作符，并在标准更新之前扩展 std::function，或者编写单目 ->*bind*。它还可以拥有一个有状态的命名操作符，我们小心地将 Op 传递到最终的调用函数，允许：

```
auto c = a *append* b;
```

The core here is that we define an append object of type append_t: named_operator::make_operator<append_t>.

We then overload named_invoke(lhs, append_t, rhs) for the types we want on the right and left.

The library overloads lhs*append_t, returning a temporary half_apply object. It also overloads half_apply*rhs to call named_invoke(lhs, append_t, rhs).

We simply have to create the proper append_t token and do an ADL-friendly named_invoke of the proper signature, and everything hooks up and works.

For a more complex example, suppose you want to have element-wise multiplication of elements of a std::array:

```
template<class=void, std::size_t... Is>
auto indexer( std::index_sequence<Is...> ) {
    return [](&auto&& f) {
        return f( std::integral_constant<std::size_t, Is>{}... );
    };
}
template<std::size_t N>
auto indexer() { return indexer( std::make_index_sequence<N>{} ); }

namespace my_ns {
    struct e_times_t : named_operator::make_operator<e_times_t> {};
    constexpr e_times_t e_times{};

    template<class L, class R, std::size_t N,
             class Out=std::decay_t<decltype( std::declval<L const&>()*std::declval<R const&>() )>
    >
    std::array<Out, N> named_invoke( std::array<L, N> const& lhs, e_times_t, std::array<R, N> const& rhs ) {
        using result_type = std::array<Out, N>;
        auto index_over_N = indexer<N>();
        return index_over_N([&](&auto... is)->result_type {
            return {{
                (lhs[is] * rhs[is])...
            }};
        });
    }
}
```

live example.

This element-wise array code can be extended to work on tuples or pairs or C-style arrays, or even variable length containers if you decide what to do if the lengths don't match.

You could also an element-wise operator type and get lhs *element_wise<'+*>* rhs.

Writing a *dot* and *cross* product operators are also obvious uses.

The use of * can be extended to support other delimiters, like +. The delimiter precedence determines the precedence of the named operator, which may be important when translating physics equations over to C++ with minimal use of extra ()s.

With a slight change in the library above, we can support ->*then* operators and extend std::function prior to the standard being updated, or write monadic ->*bind*. It could also have a stateful named operator, where we carefully pass the Op down to the final invoke function, permitting:

```

named_operator<'*'> append = [](auto lhs, auto&& rhs) {
    using std::begin; using std::end;
    lhs.insert( end(lhs), begin(rhs), end(rhs) );
    return std::move(lhs);
};

```

在C++17中生成一个命名的容器追加操作符。

第36.6节：一元操作符

您可以重载这两个一元运算符：

- `++foo` 和 `foo++`
- `--foo` 和 `foo--`

两种类型的重载是相同的（`++` 和 `--`）。向下滚动查看说明

类/结构体外的重载：

```

//前缀运算符 ++foo
T& operator++(T& lhs)
{
    //执行加法
    return lhs;
}

//后缀运算符 foo++ (使用 int 参数区分前缀和后缀)
//应基于 ++foo (前缀运算符) 实现
T operator++(T& lhs, int)
{
    T t(lhs);
    ++lhs;
    return t;
}

```

类/结构体内的重载：

```

//前缀运算符 ++foo
T& operator++()
{
    //执行加法
    return *this;
}

//后缀运算符 foo++ (使用 int 参数区分前缀和后缀)
//应基于 ++foo (前缀运算符) 实现
T operator++(int)
{
    T t(*this);
    ++(*this);
    return t;
}

```

注意：前缀运算符返回自身的引用，以便你可以继续对其进行操作。第一个参数是一个引用，因为前缀运算符会修改对象，这也是它不是`const`的原因（否则你将无法修改它）。

```

named_operator<'*'> append = [](auto lhs, auto&& rhs) {
    using std::begin; using std::end;
    lhs.insert( end(lhs), begin(rhs), end(rhs) );
    return std::move(lhs);
};

```

generating a named container-appending operator in C++17.

Section 36.6: Unary operators

You can overload the 2 unary operators:

- `++foo` 和 `foo++`
- `--foo` 和 `foo--`

Overloading is the same for both types (`++` and `--`). Scroll down for explanation

Overloading outside of [class/struct](#):

```

//Prefix operator ++foo
T& operator++(T& lhs)
{
    //Perform addition
    return lhs;
}

//Postfix operator foo++ (int argument is used to separate pre- and postfix)
//Should be implemented in terms of ++foo (prefix operator)
T operator++(T& lhs, int)
{
    T t(lhs);
    ++lhs;
    return t;
}

```

Overloading inside of [class/struct](#):

```

//Prefix operator ++foo
T& operator++()
{
    //Perform addition
    return *this;
}

//Postfix operator foo++ (int argument is used to separate pre- and postfix)
//Should be implemented in terms of ++foo (prefix operator)
T operator++(int)
{
    T t(*this);
    ++(*this);
    return t;
}

```

Note: The prefix operator returns a reference to itself, so that you can continue operations on it. The first argument is a reference, as the prefix operator changes the object, that's also the reason why it isn't `const` (you wouldn't be able to modify it otherwise).

后缀运算符按值返回一个临时对象（之前的值），因此它不能是引用，因为那将是对临时对象的引用，而临时对象在函数结束时会被销毁，变成无效值。它也不能是const，因为你应该能够直接修改它。

第一个参数是对“调用”对象的非const引用，因为如果它是const，你将无法修改它；如果它不是引用，你也无法改变原始值。

正是因为后缀运算符重载中需要复制，所以最好养成在for循环中使用前缀++而不是后缀++的习惯。从for循环的角度来看，它们通常在功能上是等价的，但使用前缀++可能会有轻微的性能优势，尤其是对于成员较多的“重量级”类来说。以下是从循环中使用前缀++的示例：

```
for (list<string>::const_iterator it = tokens.begin();  
     it != tokens.end();  
     ++it) { // 不要使用 it++  
...  
}
```

第36.7节：比较运算符

你可以重载所有比较运算符：

- == 和 !=
- > 和 <
- >= 和 <=

推荐的重载所有这些运算符的方法是只实现两个运算符（== 和 <），然后用它们来定义其余的运算符。向下滚动查看说明

类/结构体外的重载：

```
//只实现这两个  
bool operator==(const T& lhs, const T& rhs) { /* 比较 */ }  
bool operator<(const T& lhs, const T& rhs) { /* 比较 */ }  
  
//现在你可以定义其余的  
bool operator!=(const T& lhs, const T& rhs) { return !(lhs == rhs); }  
bool operator>(const T& lhs, const T& rhs) { return rhs < lhs; }  
bool operator<=(const T& lhs, const T& rhs) { return !(lhs > rhs); }  
bool operator>=(const T& lhs, const T& rhs) { return !(lhs < rhs); }
```

类/结构体内的重载：

```
//注意这些函数是 const 的，因为如果不是 const，当对象是 const 时你将无法调用它们  
  
//只实现这两个  
bool operator==(const T& rhs) const { /* 比较 */ }  
bool operator<(const T& rhs) const { /* 比较 */ }  
  
//现在你可以定义其余的  
bool operator!=(const T& rhs) const { return !(*this == rhs); }  
bool operator>(const T& rhs) const { return rhs < *this; }  
bool operator<=(const T& rhs) const { return !(rhs > *this); }  
bool operator>=(const T& rhs) const { return !(rhs < *this); }
```

The postfix operator returns by value a temporary (the previous value), and so it cannot be a reference, as it would be a reference to a temporary, which would be garbage value at the end of the function, because the temporary variable goes out of scope). It also cannot be const, because you should be able to modify it directly.

The first argument is a non-const reference to the "calling" object, because if it were const, you wouldn't be able to modify it, and if it weren't a reference, you wouldn't change the original value.

It is because of the copying needed in postfix operator overloads that it's better to make it a habit to use prefix ++ instead of postfix ++ in for loops. From the for loop perspective, they're usually functionally equivalent, but there might be a slight performance advantage to using prefix ++, especially with "fat" classes with a lot of members to copy. Example of using prefix ++ in a for loop:

```
for (list<string>::const_iterator it = tokens.begin();  
     it != tokens.end();  
     ++it) { // Don't use it++  
...  
}
```

Section 36.7: Comparison operators

You can overload all comparison operators:

- == and !=
- > and <
- >= and <=

The recommended way to overload all those operators is by implementing only 2 operators (== and <) and then using those to define the rest. Scroll down for explanation

Overloading outside of class/struct:

```
//Only implement those 2  
bool operator==(const T& lhs, const T& rhs) { /* Compare */ }  
bool operator<(const T& lhs, const T& rhs) { /* Compare */ }  
  
//Now you can define the rest  
bool operator!=(const T& lhs, const T& rhs) { return !(lhs == rhs); }  
bool operator>(const T& lhs, const T& rhs) { return rhs < lhs; }  
bool operator<=(const T& lhs, const T& rhs) { return !(lhs > rhs); }  
bool operator>=(const T& lhs, const T& rhs) { return !(lhs < rhs); }
```

Overloading inside of class/struct:

```
//Note that the functions are const, because if they are not const, you wouldn't be able  
//to call them if the object is const  
  
//Only implement those 2  
bool operator==(const T& rhs) const { /* Compare */ }  
bool operator<(const T& rhs) const { /* Compare */ }  
  
//Now you can define the rest  
bool operator!=(const T& rhs) const { return !(*this == rhs); }  
bool operator>(const T& rhs) const { return rhs < *this; }  
bool operator<=(const T& rhs) const { return !(rhs > *this); }  
bool operator>=(const T& rhs) const { return !(rhs < *this); }
```

这些运算符显然返回一个bool，表示对应操作的true或false。

所有运算符的参数都是通过const&传递的，因为运算符只做比较，不应修改对象。通过&（引用）传递比值传递更快，为了确保运算符不修改对象，使用了const引用。

注意，类（class）或结构体（struct）内的运算符被定义为const，原因是如果函数不是const，就无法比较const对象，因为编译器无法确定运算符不会修改任何内容。

第36.8节：赋值运算符

赋值运算符是最重要的运算符之一，因为它允许你改变变量的状态。

如果你没有为你的类/结构体重载赋值运算符，编译器会自动生成它：自动生成的赋值运算符执行“成员逐一赋值”，即通过调用所有成员的赋值运算符，将一个对象逐个成员复制到另一个对象。当简单的成员逐一赋值不适合你的类/结构体时，比如你需要对对象执行深拷贝，就应该重载赋值运算符。

重载赋值运算符=很简单，但你应该遵循一些简单的步骤。

1. 检测自赋值。这个检查很重要，原因有两个：
 - 自赋值是无意义的复制，因此执行它没有意义；
 - 下一步在自赋值的情况下将无法正常工作。
2. 清理旧数据。旧数据必须被新数据替换。现在，你可以理解上一步的第二个原因：如果对象的内容被销毁，自赋值将无法完成复制。
3. 复制所有成员。如果你为你的类或结构体重载赋值运算符，编译器不会自动生成它，因此你需要负责从另一个对象复制所有成员。
编译器不会自动生成赋值运算符，因此你需要负责从另一个对象复制所有成员。
4. 返回*this。运算符自身通过引用返回，因为这允许链式调用（例如int b = (a = 6) + 4; //b == 10）。

```
//T 是某种类型
T& operator=(const T& other)
{
    //执行某些操作（如复制值）
    return *this;
}
```

注意：other 通过 const& 传递，因为被赋值的对象不应被修改，且通过引用传递比值传递更快，为了确保 operator= 不会意外修改它，使用了 const。

赋值运算符只能在 class/struct 中重载，因为 = 的左值总是 class/struct 本身。将其定义为自由函数无法保证这一点，因此不允许这样做。

当你在 class/struct 中声明它时，左值隐式为 class/struct 本身，所以没有问题。

The operators obviously return a `bool`, indicating `true` or `false` for the corresponding operation.

All of the operators take their arguments by `const&`, because the only thing that does operators do is compare, so they shouldn't modify the objects. Passing by & (reference) is faster than by value, and to make sure that the operators don't modify it, it is a `const`-reference.

Note that the operators inside the `class/struct` are defined as `const`, the reason for this is that without the functions being `const`, comparing `const` objects would not be possible, as the compiler doesn't know that the operators don't modify anything.

Section 36.8: Assignment operator

The assignment operator is one of the most important operators because it allows you to change the status of a variable.

If you do not overload the assignment operator for your `class/struct`, it is automatically generated by the compiler: the automatically-generated assignment operator performs a "memberwise assignment", ie by invoking assignment operators on all members, so that one object is copied to the other, a member at time. The assignment operator should be overloaded when the simple memberwise assignment is not suitable for your `class/struct`, for example if you need to perform a `deep copy` of an object.

Overloading the assignment operator = is easy, but you should follow some simple steps.

1. **Test for self-assignment.** This check is important for two reasons:
 - a self-assignment is a needless copy, so it does not make sense to perform it;
 - the next step will not work in the case of a self-assignment.
2. **Clean the old data.** The old data must be replaced with new ones. Now, you can understand the second reason of the previous step: if the content of the object was destroyed, a self-assignment will fail to perform the copy.
3. **Copy all members.** If you overload the assignment operator for your `class` or your `struct`, it is not automatically generated by the compiler, so you will need to take charge of copying all members from the other object.
4. **Return `*this`.** The operator returns by itself by reference, because it allows chaining (i.e. `int b = (a = 6) + 4; //b == 10`).

```
//T is some type
T& operator=(const T& other)
{
    //Do something (like copying values)
    return *this;
}
```

Note: other is passed by `const&`, because the object being assigned should not be changed, and passing by reference is faster than by value, and to make sure than `operator=` doesn't modify it accidentally, it is `const`.

The assignment operator can **only** to be overloaded in the `class/struct`, because the left value of = is **always** the `class/struct` itself. Defining it as a free function doesn't have this guarantee, and is disallowed because of that.

When you declare it in the `class/struct`, the left value is implicitly the `class/struct` itself, so there is no problem with that.

第36.9节：函数调用运算符

你可以重载函数调用运算符 ()：

重载必须在 class/struct 内进行：

```
//R -> 返回类型  
//Types -> 任意不同类型  
R operator()(Type name, Type2 name2, ...)  
{  
    //执行某些操作  
    //返回某些值  
}  
  
//像这样使用 (R 是返回类型, a 和 b 是变量)  
R foo = object(a, b, ...);
```

例如：

```
结构体 Sum  
{  
    int operator()(int a, int b)  
    {  
        返回 a + b;  
    }  
};  
  
//创建结构体实例  
Sum sum;  
int result = sum(1, 1); //result == 2
```

第36.10节：按位非运算符

重载按位非 (~) 相当简单。向下滚动查看说明

类/结构体外的重载：

```
T operator~(T lhs)  
{  
    //执行操作  
    返回 lhs;  
}
```

类/结构体内的重载：

```
T 操作符~()  
{  
    T t(*this);  
    //执行操作  
    return t;  
}
```

注意：operator~返回值是按值返回的，因为它必须返回一个新值（修改后的值），而不是返回值的引用（那将是对临时对象的引用，一旦操作符执行完毕，该临时对象将包含无效值）。也不是const，因为调用代码之后应该能够修改它（例如int a = ~a + 1；应该是可行的）。

Section 36.9: Function call operator

You can overload the function call operator ():

Overloading must be done inside of a [class/struct](#):

```
//R -> Return type  
//Types -> any different type  
R operator()(Type name, Type2 name2, ...)  
{  
    //Do something  
    //return something  
}  
  
//Use it like this (R is return type, a and b are variables)  
R foo = object(a, b, ...);
```

For example:

```
struct Sum  
{  
    int operator()(int a, int b)  
    {  
        return a + b;  
    }  
};  
  
//Create instance of struct  
Sum sum;  
int result = sum(1, 1); //result == 2
```

Section 36.10: Bitwise NOT operator

Overloading the bitwise NOT (~) is fairly simple. Scroll down for explanation

Overloading outside of [class/struct](#):

```
T operator~(T lhs)  
{  
    //Do operation  
    return lhs;  
}
```

Overloading inside of [class/struct](#):

```
T operator~()  
{  
    T t(*this);  
    //Do operation  
    return t;  
}
```

Note: operator~ returns by value, because it has to return a new value (the modified value), and not a reference to the value (it would be a reference to the temporary object, which would have garbage value in it as soon as the operator is done). Not [const](#) either because the calling code should be able to modify it afterwards (i.e. `int a = ~a + 1;` should be possible).

在class/struct内部，你必须创建一个临时对象，因为你不能修改this，否则会修改原始对象，而这不应该发生。

第36.11节：用于输入输出的位移操作符

操作符<<和>>通常用作“写入”和“读取”操作符：

- std::ostream重载了<<以将变量写入底层流（例如：std::cout）
- std::istream重载了>>以从底层流读取到变量（例如：std::cin）

它们的实现方式类似于你想在class/struct外“正常”重载它们，只是参数类型不同：

- 返回类型是你想重载的流（例如std::ostream），以引用传递，允许链式调用（链式调用示例：std::cout << a << b;）。示例：std::ostream&
- lhs 将与返回类型相同
- rhs 是你想允许重载的类型（即 T），为了性能原因通过 const& 传递而不是值传递（无论如何 rhs 不应被修改）。例如：const Vector&。

示例：

```
//重载 std::ostream 操作符<< 以允许从 Vector 输出
std::ostream& operator<<(std::ostream& lhs, const Vector& rhs)
{
lhs << "x: " << rhs.x << " y: " << rhs.y << " z: " << rhs.z << ";return lhs;
}

Vector v = { 1, 2, 3};

//现在你可以这样做
std::cout << v;
```

Inside the class/struct you have to make a temporary object, because you can't modify this, as it would modify the original object, which shouldn't be the case.

Section 36.11: Bit shift operators for I/O

The operators << and >> are commonly used as "write" and "read" operators:

- std::ostream overloads << to write variables to the underlying stream (example: std::cout)
- std::istream overloads >> to read from the underlying stream to a variable (example: std::cin)

The way they do this is similar if you wanted to overload them "normally" outside of the class/struct, except that specifying the arguments are not of the same type:

- Return type is the stream you want to overload from (for example, std::ostream) passed by reference, to allow chaining (Chaining: std::cout << a << b;). Example: std::ostream&
- lhs would be the same as the return type
- rhs is the type you want to allow overloading from (i.e. T), passed by const& instead of value for performance reason (rhs shouldn't be changed anyway). Example: const Vector&.

Example:

```
//Overload std::ostream operator<< to allow output from Vector's
std::ostream& operator<<(std::ostream& lhs, const Vector& rhs)
{
lhs << "x: " << rhs.x << " y: " << rhs.y << " z: " << rhs.z << '\n';
return lhs;
}

Vector v = { 1, 2, 3};

//Now you can do
std::cout << v;
```

第37章：函数模板重载

第37.1节：什么是有效的函数模板重载？

函数模板可以根据非模板函数重载的规则进行重载（同名，但参数类型不同），此外，如果满足以下条件，重载也是有效的

- 返回类型不同，或者
- 模板参数列表不同，除了参数命名和默认参数的存在（它们不属于签名的一部分）

对于普通函数，比较两个参数类型对编译器来说很简单，因为它拥有所有信息。但模板中的类型可能尚未确定。因此，判断两个参数类型是否相等的规则在这里是近似的，要求非依赖类型和值必须匹配，依赖类型和表达式的拼写必须相同（更准确地说，它们必须符合所谓的ODR规则），但模板参数可以重命名。然而，如果在不同拼写下，类型中的两个值被认为不同，但总是实例化为相同的值，则重载无效，但编译器不需要给出诊断。

```
template<typename T>
void f(T*) { }
```

```
template<typename T>
void f(T) { }
```

这是一个有效的重载，因为“T”和“T*”是不同的拼写。但以下是无效的，且不需要诊断

```
template<typename T>
void f(T (*x)[sizeof(T) + sizeof(T)]) { }
```

```
template<typename T>
void f(T (*x)[2 * sizeof(T)]) { }
```

Chapter 37: Function Template Overloading

Section 37.1: What is a valid function template overloading?

A function template can be overloaded under the rules for non-template function overloading (same name, but different parameter types) and in addition to that, the overloading is valid if

- The return type is different, or
- The template parameter list is different, except for the naming of parameters and the presence of default arguments (they are not part of the signature)

For a normal function, comparing two parameter types is easy for the compiler, since it has all information. But a type within a template may not be determined yet. Therefore, the rule for when two parameter types are equal is approximative here and says that the non-dependent types and values need to match and the spelling of dependent types and expressions needs to be the same (more precisely, they need to conform to the so-called ODR-rules), except that template parameters may be renamed. However, if under such different spellings, two values within the types are deemed different, but will always instantiate to the same values, the overloading is invalid, but no diagnostic is required from the compiler.

```
template<typename T>
void f(T*) { }
```

```
template<typename T>
void f(T) { }
```

This is a valid overload, as "T" and "T*" are different spellings. But the following is invalid, with no diagnostic required

```
template<typename T>
void f(T (*x)[sizeof(T) + sizeof(T)]) { }
```

```
template<typename T>
void f(T (*x)[2 * sizeof(T)]) { }
```

第38章：虚成员函数

第38.1节：最终虚函数

C++11引入了final说明符，如果出现在方法签名中，则禁止方法重写：

```
class Base {
public:
    virtual void foo() {
        std::cout << "Base::Foo";
    }

class Derived1 : public Base {
public:
    // 重写 Base::foo
    void foo() final {
        std::cout << "Derived1::Foo";
    }

class Derived2 : public Derived1 {
public:
    // 编译错误：无法重写 final 方法
    virtual void foo() {
        std::cout << "Derived2::Foo";
    }
};
```

限定符 final 只能用于 'virtual' 成员函数，不能应用于非虚成员函数

类似于 final，还有一个限定符叫做 'override'，它防止在派生类中重写 virtual 函数。

限定符 override 和 final 可以组合使用以达到预期效果：

```
class Derived1 : public Base {
public:
    void foo() final override {
        std::cout << "Derived1::Foo";
    }
};
```

第38.2节：在C++11及以后版本中使用override和virtual

在C++11及以后版本中，如果在函数签名末尾添加限定符override，则该限定符具有特殊含义。这表示该函数是

- 重写基类中存在的函数 &
- 基类函数是virtual

该限定符在运行时没有运行时意义，主要是作为编译器的指示

下面的示例将演示使用或不使用override时行为的变化。

不使用override：

Chapter 38: Virtual Member Functions

Section 38.1: Final virtual functions

C++11 introduced `final` specifier which forbids method overriding if appeared in method signature:

```
class Base {
public:
    virtual void foo() {
        std::cout << "Base::Foo\n";
    }

class Derived1 : public Base {
public:
    // Overriding Base::foo
    void foo() final {
        std::cout << "Derived1::Foo\n";
    }
};

class Derived2 : public Derived1 {
public:
    // Compilation error: cannot override final method
    virtual void foo() {
        std::cout << "Derived2::Foo\n";
    }
};
```

The specifier `final` can only be used with 'virtual' member function and can't be applied to non-virtual member functions

Like `final`, there is also an specifier caller 'override' which prevent overriding of `virtual` functions in the derived class.

The specifiers `override` and `final` may be combined together to have desired effect:

```
class Derived1 : public Base {
public:
    void foo() final override {
        std::cout << "Derived1::Foo\n";
    }
};
```

Section 38.2: Using override with virtual in C++11 and later

The specifier `override` has a special meaning in C++11 onwards, if appended at the end of function signature. This signifies that a function is

- Overriding the function present in base class &
- The Base class function is `virtual`

There is no run `time` significance of this specifier as is mainly meant as an indication for compilers

The example below will demonstrate the change in behaviour with our without using `override`.

Without `override`:

```
#include <iostream>

struct X {
    virtual void f() { std::cout << "X::f()"; };

struct Y : X {
    // Y::f()不会重写X::f(), 因为它的签名不同,
    // 但编译器会接受该代码 (并且默默忽略Y::f())。
    virtual void f(int a) { std::cout << a << ""; };
};
```

使用 `override`:

```
#include <iostream>

struct X {
    virtual void f() { std::cout << "X::f()"; };

struct Y : X {
    // 编译器会提醒你 Y::f() 实际上并没有覆盖任何内容。
    virtual void f(int a) override { std::cout << a << ""; };
};
```

注意 `override` 不是关键字，而是一个特殊标识符，只能出现在函数签名中。在所有其他上下文中，`override` 仍然可以作为标识符使用：

```
void foo() {
    int override = 1; // 合法。
    int virtual = 2; // 编译错误：关键字不能用作标识符。
}
```

第38.3节：虚函数与非虚成员函数

使用虚成员函数：

```
#include <iostream>

struct X {
    virtual void f() { std::cout << "X::f()"; };

struct Y : X {
    // 这里再次指定 virtual 是可选的
    // 因为可以从 X::f() 推断出来。
    virtual void f() { std::cout << "Y::f()"; };

void call(X& a) {
    a.f();
}

int main() {
    X x;
    Y y;
    call(x); // 输出 "X::f()"
    call(y); // 输出 "Y::f()"
};
```

```
#include <iostream>

struct X {
    virtual void f() { std::cout << "X::f()\n"; };

struct Y : X {
    // Y::f() will not override X::f() because it has a different signature,
    // but the compiler will accept the code (and silently ignore Y::f()).
    virtual void f(int a) { std::cout << a << "\n"; };
};
```

With `override`:

```
#include <iostream>

struct X {
    virtual void f() { std::cout << "X::f()\n"; };

struct Y : X {
    // The compiler will alert you to the fact that Y::f() does not
    // actually override anything.
    virtual void f(int a) override { std::cout << a << "\n"; };
};
```

Note that `override` is not a keyword, but a special identifier which only may appear in function signatures. In all other contexts `override` still may be used as an identifier:

```
void foo() {
    int override = 1; // OK.
    int virtual = 2; // Compilation error: keywords can't be used as identifiers.
}
```

Section 38.3: Virtual vs non-virtual member functions

With virtual member functions:

```
#include <iostream>

struct X {
    virtual void f() { std::cout << "X::f()\n"; };

struct Y : X {
    // Specifying virtual again here is optional
    // because it can be inferred from X::f().
    virtual void f() { std::cout << "Y::f()\n"; };
};

void call(X& a) {
    a.f();
}

int main() {
    X x;
    Y y;
    call(x); // outputs "X::f()"
    call(y); // outputs "Y::f()"
};
```

```
}
```

没有虚函数成员：

```
#include <iostream>

struct X {
    void f() { std::cout << "X::f()"; }

struct Y : X {
    void f() { std::cout << "Y::f()"; }

void call(X& a) {
    a.f();
}

int main() {
    X x;
    Y y;
    call(x); // 输出 "X::f()"
    call(y); // 输出 "X::f()"
}
```

```
}
```

Without virtual member functions:

```
#include <iostream>

struct X {
    void f() { std::cout << "X::f()\n"; }

struct Y : X {
    void f() { std::cout << "Y::f()\n"; }

void call(X& a) {
    a.f();
}

int main() {
    X x;
    Y y;
    call(x); // outputs "X::f()"
    call(y); // outputs "X::f()"
}
```

第38.4节：构造函数和析构函数中虚函数的行为

虚函数在构造函数和析构函数中的行为初次接触时常常令人困惑。

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

class base {
public:
    base() { f("base constructor"); }
    ~base() { f("base destructor"); }

    virtual const char* v() { return "base::v()"; }

    void f(const char* caller) {
        cout << "当从 " << caller << " 调用时, " << v() << " 被调用。";
    }
};

class derived : public base {
public:
    derived() { f("derived constructor"); }
    ~derived() { f("derived destructor"); }

    const char* v() override { return "derived::v()"; }
};

int main() {
    derived d;
}
```

输出：

Section 38.4: Behaviour of virtual functions in constructors and destructors

The behaviour of virtual functions in constructors and destructors is often confusing when first encountered.

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

class base {
public:
    base() { f("base constructor"); }
    ~base() { f("base destructor"); }

    virtual const char* v() { return "base::v()"; }

    void f(const char* caller) {
        cout << "When called from " << caller << ", " << v() << " gets called.\n";
    }
};

class derived : public base {
public:
    derived() { f("derived constructor"); }
    ~derived() { f("derived destructor"); }

    const char* v() override { return "derived::v()"; }
};

int main() {
    derived d;
}
```

Output:

当从 base 构造函数调用时，调用的是 base::v()。
 当从派生类构造函数调用时，会调用派生类的 v() 函数。
 当从派生类析构函数调用时，调用的是 derived::v()。
 当从基类析构函数调用时，调用的是 base::v()。

其原因在于派生类可能定义了额外的成员，这些成员在构造函数中尚未初始化，或在析构函数中已被销毁，调用其成员函数将是不安全的。因此，在 C++ 对象的构造和析构过程中，*this 的动态类型被视为构造函数或析构函数所属的类，而非更派生的类。

示例：

```
#include <iostream>
#include <memory>

using namespace std;
class base {
public:
base()
{
std::cout << "foo is " << foo() << std::endl;
}
virtual int foo() { return 42; }
};

class derived : public base {
unique_ptr<int> ptr_;
public:
derived(int i) : ptr_(new int(i*i)) { }
// 由于 C++ 的行为，以下内容不能在 derived::derived 之前调用，
// 如果可能的话.....程序将崩溃！
int foo() override { return *ptr_; }
};

int main() {
derived d(4);
}
```

When called from base constructor, base::v() gets called.
 When called from derived constructor, derived::v() gets called.
 When called from derived destructor, derived::v() gets called.
 When called from base destructor, base::v() gets called.

The reasoning behind this is that the derived class may define additional members which are not yet initialized (in the constructor case) or already destroyed (in the destructor case), and calling its member functions would be unsafe. Therefore during construction and destruction of C++ objects, the *dynamic type* of `*this` is considered to be the constructor's or destructor's class and not a more-derived class.

Example:

```
#include <iostream>
#include <memory>

using namespace std;
class base {
public:
base()
{
std::cout << "foo is " << foo() << std::endl;
}
virtual int foo() { return 42; }
};

class derived : public base {
unique_ptr<int> ptr_;
public:
derived(int i) : ptr_(new int(i*i)) { }
// The following cannot be called before derived::derived due to how C++ behaves,
// if it was possible... Kaboom!
int foo() override { return *ptr_; }
};

int main() {
derived d(4);
}
```

第38.5节：纯虚函数

我们也可以通过在声明后附加= 0来指定一个虚函数为纯虚函数（抽象函数）。包含一个或多个纯虚函数的类被视为抽象类，不能被实例化；只有定义了所有纯虚函数或继承了其定义的派生类才能被实例化。

```
struct Abstract {
virtual void f() = 0;
};

struct Concrete {
void f() override {}
};

Abstract a; // 错误。
Concrete c; // 正确。
```

即使函数被指定为纯虚函数，也可以为其提供默认实现。尽管如此，该函数仍被视为抽象函数，派生类必须定义它才能被实例化。在这种情况下，

Section 38.5: Pure virtual functions

We can also specify that a `virtual` function is *pure virtual* (abstract), by appending `= 0` to the declaration. Classes with one or more pure virtual functions are considered to be abstract, and cannot be instantiated; only derived classes which define, or inherit definitions for, all pure virtual functions can be instantiated.

```
struct Abstract {
virtual void f() = 0;
};

struct Concrete {
void f() override {}
};

Abstract a; // Error.
Concrete c; // Good.
```

Even if a function is specified as pure virtual, it can be given a default implementation. Despite this, the function will still be considered abstract, and derived classes will have to define it before they can be instantiated. In this case,

派生类版本的函数甚至可以调用基类版本的函数。

```
结构体 DefaultAbstract {
    虚函数 f() = 0;
};

void DefaultAbstract::f() {}

struct WhyWouldWeDoThis : DefaultAbstract {
    void f() override { DefaultAbstract::f(); }
};
```

我们可能想这样做的原因有几个：

- 如果我们想创建一个自身不能被实例化的类，但不阻止其派生类被实例化，我们可以将析构函数声明为纯虚函数。作为析构函数，无论如何都必须定义，如果我们想能够释放实例。并且由于析构函数很可能已经是虚函数，以防止多态使用时的内存泄漏，因此声明另一个函数为virtual不会带来不必要的性能损失。这在制作接口时非常有用。

```
struct Interface {
    virtual ~Interface() = 0;
};

Interface::~Interface() = default;

struct Implementation : Interface {};
// 如果未显式指定，编译器会自动定义 ~Implementation()，满足“必须在实例化前定义”的要求。
```

- 如果纯虚函数的大多数或所有实现都包含重复代码，则可以将该代码移到基类版本中，使代码更易于维护。

```
class SharedBase {
    State my_state;
    std::unique_ptr<Helper> my_helper;
    // ...

public:
    virtual void config(const Context& cont) = 0;
    // ...
};

/* virtual */ void SharedBase::config(const Context& cont) {
    my_helper = new Helper(my_state, cont.relevant_field);
    do_this();
    and_that();
}

class OneImplementation : public SharedBase {
    int i;
    // ...

public:
    void config(const Context& cont) override;
    // ...
};

void OneImplementation::config(const Context& cont) /* override */ {
    my_state = { cont.some_field, cont.another_field, i };
    SharedBase::config(cont);
    my_unique_setup();
}
```

the derived class' version of the function is even allowed to call the base class' version.

```
struct DefaultAbstract {
    virtual void f() = 0;
};

void DefaultAbstract::f() {}

struct WhyWouldWeDoThis : DefaultAbstract {
    void f() override { DefaultAbstract::f(); }
};
```

There are a couple of reasons why we might want to do this:

- If we want to create a class that can't itself be instantiated, but doesn't prevent its derived classes from being instantiated, we can declare the destructor as pure virtual. Being the destructor, it must be defined anyways, if we want to be able to deallocate the instance. And as the destructor is most likely already virtual to prevent memory leaks during polymorphic use, we won't incur an unnecessary performance hit from declaring another function `virtual`. This can be useful when making interfaces.

```
struct Interface {
    virtual ~Interface() = 0;
};

Interface::~Interface() = default;

struct Implementation : Interface {};
// ~Implementation() is automatically defined by the compiler if not explicitly
// specified, meeting the "must be defined before instantiation" requirement.
```

- If most or all implementations of the pure virtual function will contain duplicate code, that code can instead be moved to the base class version, making the code easier to maintain.

```
class SharedBase {
    State my_state;
    std::unique_ptr<Helper> my_helper;
    // ...

public:
    virtual void config(const Context& cont) = 0;
    // ...
};

/* virtual */ void SharedBase::config(const Context& cont) {
    my_helper = new Helper(my_state, cont.relevant_field);
    do_this();
    and_that();
}

class OneImplementation : public SharedBase {
    int i;
    // ...

public:
    void config(const Context& cont) override;
    // ...
};

void OneImplementation::config(const Context& cont) /* override */ {
    my_state = { cont.some_field, cont.another_field, i };
    SharedBase::config(cont);
    my_unique_setup();
}
```

```
// And so on, for other classes derived from SharedBase.
```

```
// And so on, for other classes derived from SharedBase.
```

第39章：内联函数

使用`inline`说明符定义的函数称为内联函数。内联函数可以被多次定义而不违反一一定义规则，因此可以在具有外部链接的头文件中定义。声明函数为内联函数是向编译器提示该函数应在代码生成时进行内联，但这并不保证一定会内联。

第39.1节：非成员内联函数定义

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

第39.2节：成员内联函数

```
// 头文件 (.hpp)
struct A
{
    void i_am_inlined()
    {
    }
};

struct B
{
    void i_am_NOT_inlined();
};

// 源文件 (.cpp)
void B::i_am_NOT_inlined()
```

第39.3节：什么是函数内联？

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

int main()
{
    int a = 1, b = 2;
    int c = add(a, b);
}
```

在上述代码中，当`add`被内联时，生成的代码将变成类似如下

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

内联函数已经看不到了，它的函数体被内联到了调用者的函数体中。如果`add`没有被内联，将会调用一个

Chapter 39: Inline functions

A function defined with the `inline` specifier is an inline function. An inline function can be multiply defined without violating the One Definition Rule, and can therefore be defined in a header with external linkage. Declaring a function inline hints to the compiler that the function should be inlined during code generation, but does not provide a guarantee.

Section 39.1: Non-member inline function definition

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

Section 39.2: Member inline functions

```
// header (.hpp)
struct A
{
    void i_am_inlined()
    {
    }
};

struct B
{
    void i_am_NOT_inlined();
};

// source (.cpp)
void B::i_am_NOT_inlined()
```

Section 39.3: What is function inlining?

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

int main()
{
    int a = 1, b = 2;
    int c = add(a, b);
}
```

In the above code, when `add` is inlined, the resulting code would become something like this

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

The inline function is nowhere to be seen, its body gets *inlined* into the caller's body. Had `add` not been inlined, a

函数。调用函数的开销——例如创建新的栈帧、复制参数、生成局部变量、跳转（丧失局部性引用从而导致缓存未命中）等——都必须承担。

第39.4节：非成员内联函数声明

```
inline int add(int x, int y);
```

function would be called. The overhead of calling a function -- such as creating a new stack frame, copying arguments, making local variables, jump (losing locality of reference and there by cache misses), etc. -- has to be incurred.

Section 39.4: Non-member inline function declaration

```
inline int add(int x, int y);
```

第40章：特殊成员函数

第40.1节：默认构造函数

默认构造函数是一种调用时不需要参数的构造函数。它以所构造的类型命名，并且是该类型的成员函数（所有构造函数都是如此）。

```
类 C{
    整数 i;
公共:
    // 默认构造函数定义
C()
: i(0){ // 成员初始化列表 -- 将 i 初始化为 0
    // 构造函数体 -- 这里可以做更复杂的操作
}
};

C c1; // 调用 C 的默认构造函数创建对象 c1
C c2 = C(); // 显式调用默认构造函数
C c3(); // 错误：由于“最令人困惑的解析”，这种直观写法不可行
C c4{}; // 但在 C++11 中，{} 可以以类似方式使用

C c5[2]; // 为数组的两个元素调用默认构造函数
C* c6 = new C[2]; // 为数组的两个元素调用默认构造函数
```

满足“无参数”要求的另一种方法是开发者为所有参数提供默认值：

```
类 D{
    整数 i;
    整数 j;
公共:
    // 也是一个默认构造函数（可以无参数调用）
D( 整数 i = 0, 整数 j = 42 )
: i(i), j(j){
}
};
```

```
D d; // 使用提供的参数默认值调用D的构造函数
```

在某些情况下（即开发者未提供构造函数且没有其他不合格条件时），编译器会隐式提供一个空的默认构造函数：

```
class C{
std::string s; // 注意：成员本身需要是可默认构造的
};

C c1; // 成功 —— C有一个隐式定义的默认构造函数
```

拥有其他类型的构造函数是前面提到的不合格条件之一：

```
类 C{
    整数 i;
公共:
C( int i ) : i(i){}
};
```

Chapter 40: Special Member Functions

Section 40.1: Default Constructor

A *default constructor* is a type of constructor that requires no parameters when called. It is named after the type it constructs and is a member function of it (as all constructors are).

```
class C{
    int i;
public:
    // the default constructor definition
C()
: i(0){ // member initializer list -- initialize i to 0
    // constructor function body -- can do more complex things here
}
};

C c1; // calls default constructor of C to create object c1
C c2 = C(); // calls default constructor explicitly
C c3(); // ERROR: this intuitive version is not possible due to "most vexing parse"
C c4{}; // but in C++11 {} CAN be used in a similar way

C c5[2]; // calls default constructor for both array elements
C* c6 = new C[2]; // calls default constructor for both array elements
```

Another way to satisfy the “no parameters” requirement is for the developer to provide default values for all parameters:

```
class D{
    int i;
    int j;
public:
    // also a default constructor (can be called with no parameters)
D( int i = 0, int j = 42 )
: i(i), j(j){
}
};
```

```
D d; // calls constructor of D with the provided default values for the parameters
```

Under some circumstances (i.e., the developer provides no constructors and there are no other disqualifying conditions), the compiler implicitly provides an empty default constructor:

```
class C{
    std::string s; // note: members need to be default constructible themselves
};

C c1; // will succeed -- C has an implicitly defined default constructor
```

Having some other type of constructor is one of the disqualifying conditions mentioned earlier:

```
class C{
    int i;
public:
    C( int i ) : i(i){}
};
```

```
C c1; // 编译错误：C没有（隐式定义的）默认构造函数
```

版本 < c++11

为了防止隐式默认构造函数的生成，常用的做法是将其声明为private（且不定义）。当有人尝试使用构造函数时，意图是导致编译错误（这要么导致访问私有成员错误，要么导致链接器错误，取决于编译器）。

为了确保定义了一个默认构造函数（功能上类似于隐式构造函数），开发者可以显式地编写一个空的构造函数。

版本 ≥ c++11

在 C++11 中，开发者还可以使用delete关键字来阻止编译器提供默认构造函数。

```
类 C{  
    整数 i;  
公共：  
    // 默认构造函数被显式删除  
    C() = delete;  
};
```

```
C c1; // 编译错误：C 的默认构造函数被删除
```

此外，开发者也可以显式表示希望编译器提供默认构造函数。

```
类 C{  
    整数 i;  
公共：  
    // 确实有自动生成的默认构造函数（与隐式构造函数相同）  
    C() = default;  
  
C( int i ) : i(i){}  
};
```

```
C c1; // 默认构造  
C c2( 1 ); // 使用接受 int 的构造函数构造
```

版本 ≥ c++14

你可以使用以下方法判断一个类型是否有默认构造函数（或是否为原始类型）

`std::is_default_constructible` 来自 `<type_traits>`：

```
class C1{};  
class C2{ public: C2(){} };  
class C3{ public: C3(int){} };  
  
using std::cout; using std::boolalpha; using std::endl;  
using std::is_default_constructible;  
cout << boolalpha << is_default_constructible<int>() << endl; // 输出 true  
cout << boolalpha << is_default_constructible<C1>() << endl; // 输出 true  
cout << boolalpha << is_default_constructible<C2>() << endl; // 输出 true  
cout << boolalpha << is_default_constructible<C3>() << endl; // 输出 false
```

版本 = c++11

在 C++11 中，仍然可以使用非函数对象版本的 `std::is_default_constructible`：

```
cout << boolalpha << is_default_constructible<C1>::value << endl; // 输出 true
```

```
C c1; // Compile ERROR: C has no (implicitly defined) default constructor  
Version < c++11
```

To prevent implicit default constructor creation, a common technique is to declare it as `private` (with no definition). The intention is to cause a compile error when someone tries to use the constructor (this either results in an Access to private error or a linker error, depending on the compiler).

To be sure a default constructor (functionally similar to the implicit one) is defined, a developer could write an empty one explicitly.

Version ≥ c++11

In C++11, a developer can also use the `delete` keyword to prevent the compiler from providing a default constructor.

```
class C{  
    int i;  
public:  
    // default constructor is explicitly deleted  
    C() = delete;  
};
```

```
C c1; // Compile ERROR: C has its default constructor deleted
```

Furthermore, a developer may also be explicit about wanting the compiler to provide a default constructor.

```
class C{  
    int i;  
public:  
    // does have automatically generated default constructor (same as implicit one)  
    C() = default;  
  
    C( int i ) : i(i){}  
};
```

```
C c1; // default constructed  
C c2( 1 ); // constructed with the int taking constructor
```

Version ≥ c++14

You can determine whether a type has a default constructor (or is a primitive type) using `std::is_default_constructible` from `<type_traits>`:

```
class C1{};  
class C2{ public: C2(){} };  
class C3{ public: C3(int){} };  
  
using std::cout; using std::boolalpha; using std::endl;  
using std::is_default_constructible;  
cout << boolalpha << is_default_constructible<int>() << endl; // prints true  
cout << boolalpha << is_default_constructible<C1>() << endl; // prints true  
cout << boolalpha << is_default_constructible<C2>() << endl; // prints true  
cout << boolalpha << is_default_constructible<C3>() << endl; // prints false
```

Version = c++11

In C++11, it is still possible to use the non-functor version of `std::is_default_constructible`:

```
cout << boolalpha << is_default_constructible<C1>::value << endl; // prints true
```

第40.2节：析构函数

析构函数是一种无参数的函数，当用户定义的对象即将被销毁时调用。它以被销毁类型的名称加上“~”前缀命名。

```
类 C{
    int* is;
    string s;
public:
C()
: is( new int[10] ){
}

~C(){ // 析构函数定义
    delete[] is;
}
};

类 C_child : public C{
    string s_ch;
public:
C_child(){}
~C_child(){} // 子类析构函数
};

void f(){
C c1; // 调用默认构造函数
C c2[2]; // 为两个元素调用默认构造函数
C* c3 = new C[2]; // 为数组的两个元素调用默认构造函数

C_child c_ch; // 析构时调用 s_ch 和 C 基类的析构函数 (进而调用 s 的析构函数)

delete[] c3; // 调用 c3[0] 和 c3[1] 的析构函数
} // 自动变量在此处被销毁——即 c1、c2 和 c_ch
```

在大多数情况下（即用户未提供析构函数，且没有其他不合格条件时），编译器会隐式提供默认析构函数：

```
class C{
    int i;
string s;
};

void f(){
C* c1 = new C;
    delete c1; // C 有析构函数
}
```

```
class C{
    int m;
private:
~C(){} // 非公共析构函数！
};

class C_container{
    C c;
};

void f()
```

Section 40.2: Destructor

A *destructor* is a function without arguments that is called when a user-defined object is about to be destroyed. It is named after the type it destructs with a ~ prefix.

```
class C{
    int* is;
    string s;
public:
C()
: is( new int[10] ){

}

~C(){ // destructor definition
    delete[] is;
}
};

class C_child : public C{
    string s_ch;
public:
C_child(){}
~C_child(){} // child destructor
};

void f(){
C c1; // calls default constructor
C c2[2]; // calls default constructor for both elements
C* c3 = new C[2]; // calls default constructor for both array elements

C_child c_ch; // when destructed calls destructor of s_ch and of C base (and in turn s)

delete[] c3; // calls destructors on c3[0] and c3[1]
} // automatic variables are destroyed here -- i.e. c1, c2 and c_ch
```

Under most circumstances (i.e., a user provides no destructor, and there are no other disqualifying conditions), the compiler provides a default destructor implicitly:

```
class C{
    int i;
string s;
};

void f(){
C* c1 = new C;
    delete c1; // C has a destructor
}
```

```
class C{
    int m;
private:
~C(){} // not public destructor!
};

class C_container{
    C c;
};

void f()
```

```
C_container* c_cont = new C_container;
    删除 c_cont; // 编译错误：C 没有可访问的析构函数
}
版本 > c++11
```

在 C++11 中，开发者可以通过阻止编译器提供默认析构函数来覆盖此行为。

```
类 C{
    int m;
公共:
~C() = delete; // 没有隐式析构函数
};

void f{
C c1;
} // 编译错误：C 没有析构函数
```

此外，开发者也可以明确表示希望编译器提供默认析构函数。

```
类 C{
    int m;
公共:
~C() = default; // 明确表示它有隐式/空的析构函数
};

void f(){
C c1;
} // C 有一个析构函数 -- c1 被正确销毁
版本 > c++11
```

你可以使用来自

```
class C1{ };
class C2{ public: ~C2() = delete };
class C3 : public C2{ };

using std::cout; using std::boolalpha; using std::endl;
using std::is_destructible;
cout << boolalpha << is_destructible<int>() << endl; // 输出 true
cout << boolalpha << is_destructible<C1>() << endl; // 输出 true
cout << boolalpha << is_destructible<C2>() << endl; // 输出 false
cout << boolalpha << is_destructible<C3>() << endl; // 输出 false
```

第40.3节：拷贝与交换

如果你正在编写一个管理资源的类，你需要实现所有特殊成员函数（参见三法则/五法则/零法则）。编写拷贝构造函数和赋值运算符的最直接方法是：

```
person(const person &other)
: name(new char[strlen(other.name) + 1])
, age(other.age)
{
std::strcpy(name, other.name);
}

person& operator=(person const& rhs) {
    if (this != &other) {
        删除 [] 名称;
```

```
C_container* c_cont = new C_container;
    delete c_cont; // Compile ERROR: C has no accessible destructor
}
Version > c++11
```

In C++11, a developer can override this behavior by preventing the compiler from providing a default destructor.

```
class C{
    int m;
public:
~C() = delete; // does NOT have implicit destructor
};

void f{
C c1;
} // Compile ERROR: C has no destructor
```

Furthermore, a developer may also be explicit about wanting the compiler to provide a default destructor.

```
class C{
    int m;
public:
~C() = default; // saying explicitly it does have implicit/empty destructor
};

void f(){
C c1;
} // C has a destructor -- c1 properly destroyed
Version > c++11
```

You can determine whether a type has a destructor (or is a primitive type) using `std::is_destructible` from `<type_traits>`:

```
class C1{ };
class C2{ public: ~C2() = delete };
class C3 : public C2{ };

using std::cout; using std::boolalpha; using std::endl;
using std::is_destructible;
cout << boolalpha << is_destructible<int>() << endl; // prints true
cout << boolalpha << is_destructible<C1>() << endl; // prints true
cout << boolalpha << is_destructible<C2>() << endl; // prints false
cout << boolalpha << is_destructible<C3>() << endl; // prints false
```

Section 40.3: Copy and swap

If you're writing a class that manages resources, you need to implement all the special member functions (see Rule of Three/Five/Zero). The most direct approach to writing the copy constructor and assignment operator would be:

```
person(const person &other)
: name(new char[strlen(other.name) + 1])
, age(other.age)
{
std::strcpy(name, other.name);
}

person& operator=(person const& rhs) {
    if (this != &other) {
        delete [] name;
```

```

name = new char[std::strlen(other.name) + 1];
    std::strcpy(name, other.name);
age = other.age;
}

return *this;
}

```

但这种方法存在一些问题。它无法满足强异常保证——如果new[]抛出异常，我们已经清理了this所拥有的资源，无法恢复。我们在复制赋值中重复了大量复制构造的逻辑。而且我们必须记住自赋值检查，这通常只是增加复制操作的开销，但仍然至关重要。

为了满足强异常保证并避免代码重复（在后续的移动赋值操作符中重复两次），我们可以使用复制-交换惯用法：

```

class person {
    char* name;
    int age;
public:
    /* 其他所有函数 ... */

    friend void swap(person& lhs, person& rhs) {
        using std::swap; // 启用ADL

        swap(lhs.name, rhs.name);
        swap(lhs.age, rhs.age);
    }

    person& operator=(person rhs) {
        swap(*this, rhs);
        return *this;
    }
};

```

为什么这样做有效？考虑当我们有

```

person p1 = ...;
person p2 = ...;
p1 = p2;

```

首先，我们从p2复制构造 rhs（这里我们不需要复制）。如果该操作抛出异常，operator=中不会做任何操作，p1保持不变。接下来，我们交换*this和 rhs之间的成员，然后 rhs超出作用域。当operator=结束时，这会隐式地清理 this的原始资源（通过析构函数，我们不需要复制）。自赋值也能正常工作——使用复制交换法效率较低（涉及额外的分配和释放），但如果这是不太可能的情况，我们不会因此降低典型用例的性能。

版本 ≥ C++11

上述形式对于移动赋值已经可以直接使用。

```
p1 = std::move(p2);
```

这里，我们从p2移动构造 rhs，其余部分同样有效。如果一个类是可移动但不可复制的，则无需删除复制赋值运算符，因为该赋值运算符由于复制构造函数被删除而本身就是非法的。

```

name = new char[std::strlen(other.name) + 1];
    std::strcpy(name, other.name);
age = other.age;
}

return *this;
}

```

But this approach has some problems. It fails the strong exception guarantee - if `new[]` throws, we've already cleared the resources owned by `this` and cannot recover. We're duplicating a lot of the logic of copy construction in copy assignment. And we have to remember the self-assignment check, which usually just adds overhead to the copy operation, but is still critical.

To satisfy the strong exception guarantee and avoid code duplication (double so with the subsequent move assignment operator), we can use the copy-and-swap idiom:

```

class person {
    char* name;
    int age;
public:
    /* all the other functions ... */

    friend void swap(person& lhs, person& rhs) {
        using std::swap; // enable ADL

        swap(lhs.name, rhs.name);
        swap(lhs.age, rhs.age);
    }

    person& operator=(person rhs) {
        swap(*this, rhs);
        return *this;
    }
};

```

Why does this work? Consider what happens when we have

```

person p1 = ...;
person p2 = ...;
p1 = p2;

```

First, we copy-construct rhs from p2 (which we didn't have to duplicate here). If that operation throws, we don't do anything in operator= and p1 remains untouched. Next, we swap the members between `*this` and rhs, and then rhs goes out of scope. When operator=, that implicitly cleans the original resources of `this` (via the destructor, which we didn't have to duplicate). Self-assignment works too - it's less efficient with copy-and-swap (involves an extra allocation and deallocation), but if that's the unlikely scenario, we don't slow down the typical use case to account for it.

Version ≥ C++11

The above formulation works as-is already for move assignment.

```
p1 = std::move(p2);
```

Here, we move-construct rhs from p2, and all the rest is just as valid. If a class is movable but not copyable, there is no need to delete the copy-assignment, since this assignment operator will simply be ill-formed due to the deleted copy constructor.

第40.4节：隐式移动和复制

请记住，声明析构函数会阻止编译器生成隐式的移动构造函数和移动赋值运算符。如果你声明了析构函数，记得也要为移动操作添加适当的定义。

此外，声明移动操作会抑制复制操作的生成，因此如果该类的对象需要复制语义，也应添加复制操作。

```
class Movable {
public:
    virtual ~Movable() noexcept = default;

    // 除非我们告诉编译器，否则它不会生成这些
    // 因为我们声明了析构函数
    Movable(Movable&&) noexcept = default;
    Movable& operator=(Movable&&) noexcept = default;

    // 声明移动操作将抑制复制操作的生成
    // 除非我们显式重新启用它们
    Movable(const Movable&) = default;
    Movable& operator=(const Movable&) = default;
};
```

Section 40.4: Implicit Move and Copy

Bear in mind that declaring a destructor inhibits the compiler from generating implicit move constructors and move assignment operators. If you declare a destructor, remember to also add appropriate definitions for the move operations.

Furthermore, declaring move operations will suppress the generation of copy operations, so these should also be added (if the objects of this class are required to have copy semantics).

```
class Movable {
public:
    virtual ~Movable() noexcept = default;

    // compiler won't generate these unless we tell it to
    // because we declared a destructor
    Movable(Movable&&) noexcept = default;
    Movable& operator=(Movable&&) noexcept = default;

    // declaring move operations will suppress generation
    // of copy operations unless we explicitly re-enable them
    Movable(const Movable&) = default;
    Movable& operator=(const Movable&) = default;
};
```

第41章：非静态成员函数

第41.1节：非静态成员函数

类 (class) 或结构体 (struct) 可以拥有成员函数以及成员变量。这些函数的语法大多与独立函数相似，可以定义在类定义内部或外部；如果定义在类定义外部，函数名需加上类名和作用域运算符 (::) 作为前缀。

```
class CL {  
public:  
    void definedInside() {}  
    void definedOutside();  
};  
void CL::definedOutside() {}
```

这些函数通过点运算符 (.) 在类的实例（或实例引用）上调用，或者通过箭头运算符 (->) 在实例指针上调用，每次调用都绑定到调用该函数的实例；当成员函数在实例上被调用时，它可以访问该实例的所有字段（通过this指针），但只能访问作为参数传入的其他实例的字段。

```
结构体 ST {  
ST(const std::string& ss = "Wolf", int ii = 359) : s(ss), i(ii) {}  
  
    int get_i() const { return i; }  
    bool compare_i(const ST& other) const { return (i == other.i); }  
  
private:  
std::string s;  
    int i;  
};  
ST st1;  
ST st2("Species", 8472);  
  
int i = st1.get_i(); // 可以访问 st1.i, 但不能访问 st2.i。  
bool b = st1.compare_i(st2); // 可以访问 st1 和 st2。
```

这些函数被允许访问成员变量和/或其他成员函数，无论变量或函数的访问修饰符如何。它们也可以乱序编写，访问在它们之前声明的成员变量和/或调用成员函数，因为整个类定义必须被解析后，编译器才能开始编译类。

```
class Access {  
public:  
Access(int i_ = 8088, int j_ = 8086, int k_ = 6502) : i(i_), j(j_), k(k_) {}  
  
    int i;  
    int get_k() const { return k; }  
    bool private_no_more() const { return i_be_private(); }  
protected:  
    int j;  
    int get_i() const { return i; }  
private:  
    int k;  
    int get_j() const { return j; }  
    bool i_be_private() const { return ((i > j) && (k < j)); }  
};
```

Chapter 41: Non-Static Member Functions

Section 41.1: Non-static Member Functions

A `class` or `struct` can have member functions as well as member variables. These functions have syntax mostly similar to standalone functions, and can be defined either inside or outside the class definition; if defined outside the class definition, the function's name is prefixed with the class' name and the scope (:) operator.

```
class CL {  
public:  
    void definedInside() {}  
    void definedOutside();  
};  
void CL::definedOutside() {}
```

These functions are called on an instance (or reference to an instance) of the class with the dot (.) operator, or a pointer to an instance with the arrow (->) operator, and each call is tied to the instance the function was called on; when a member function is called on an instance, it has access to all of that instance's fields (through the `this` pointer), but can only access other instances' fields if those instances are supplied as parameters.

```
struct ST {  
ST(const std::string& ss = "Wolf", int ii = 359) : s(ss), i(ii) {}  
  
    int get_i() const { return i; }  
    bool compare_i(const ST& other) const { return (i == other.i); }  
  
private:  
std::string s;  
    int i;  
};  
ST st1;  
ST st2("Species", 8472);  
  
int i = st1.get_i(); // Can access st1.i, but not st2.i.  
bool b = st1.compare_i(st2); // Can access st1 & st2.
```

These functions are allowed to access member variables and/or other member functions, regardless of either the variable or function's access modifiers. They can also be written out-of-order, accessing member variables and/or calling member functions declared before them, as the entire class definition must be parsed before the compiler can begin to compile a class.

```
class Access {  
public:  
Access(int i_ = 8088, int j_ = 8086, int k_ = 6502) : i(i_), j(j_), k(k_) {}  
  
    int i;  
    int get_k() const { return k; }  
    bool private_no_more() const { return i_be_private(); }  
protected:  
    int j;  
    int get_i() const { return i; }  
private:  
    int k;  
    int get_j() const { return j; }  
    bool i_be_private() const { return ((i > j) && (k < j)); }  
};
```

第41.2节：封装

成员函数的一个常见用途是封装，使用访问器（通常称为getter）和修改器（通常称为setter）来代替直接访问字段。

```
class 封装器 {
    int 封装的变量;

public:
    int get_encapsulated() const { return 封装的变量; }
    void set_encapsulated(int e) { 封装的变量 = e; }

    void some_func() {
        do_something_with(封装的变量);
    }
};
```

在类内部，封装的变量可以被任何非静态成员函数自由访问；在类外部，访问由成员函数控制，使用get_encapsulated()读取，使用set_encapsulated()修改。

这样可以防止对变量的无意修改，因为读取和写入使用了不同的函数。[关于getter和setter是否提供或破坏封装，有许多讨论，双方都有充分的论据；此类激烈争论超出本例范围。]

Section 41.2: Encapsulation

A common use of member functions is for encapsulation, using an *accessor* (commonly known as a *getter*) and a *mutator* (commonly known as a *setter*) instead of accessing fields directly.

```
class Encapsulator {
    int encapsulated;

public:
    int get_encapsulated() const { return encapsulated; }
    void set_encapsulated(int e) { encapsulated = e; }

    void some_func() {
        do_something_with(encapsulated);
    }
};
```

Inside the class, `encapsulated` can be freely accessed by any non-static member function; outside the class, access to it is regulated by member functions, using `get_encapsulated()` to read it and `set_encapsulated()` to modify it. This prevents unintentional modifications to the variable, as separate functions are used to read and write it. [There are many discussions on whether getters and setters provide or break encapsulation, with good arguments for both claims; such heated debate is outside the scope of this example.]

第41.3节：名称隐藏与导入

当基类提供一组重载函数，而派生类向该集合中添加另一个重载时，这会隐藏基类提供的所有重载。

```
结构体 HiddenBase {
    void f(int) { std::cout << "int" << std::endl; }
    void f(bool) { std::cout << "bool" << std::endl; }
    void f(std::string) { std::cout << "std::string" << std::endl; }
};

结构体 HidingDerived : HiddenBase {
    void f(float) { std::cout << "float" << std::endl; }
};

// ...

HiddenBase hb;
HidingDerived hd;
std::string s;

hb.f(1); // 输出: int
hb.f(true); // 输出: bool
hb.f(s); // 输出: std::string;

hd.f(1.0); // 输出: float
hd.f(3); // 输出: float
hd.f(true); // 输出: float
hd.f(s); // 错误: 无法将 std::string 转换为 float.
```

这是由于名称解析规则：在名称查找过程中，一旦找到正确的名称，我们就停止查找，即使我们显然还没有找到该名称对应的正确版本（例如 `hd.f(s)`）；因此，在派生类中重载函数会阻止名称查找发现基类中的重载函数。为避免这种情况，可以使用 `using` 声明将基类中的名称“导入”到派生类中，使

Section 41.3: Name Hiding & Importing

When a base class provides a set of overloaded functions, and a derived class adds another overload to the set, this hides all of the overloads provided by the base class.

```
struct HiddenBase {
    void f(int) { std::cout << "int" << std::endl; }
    void f(bool) { std::cout << "bool" << std::endl; }
    void f(std::string) { std::cout << "std::string" << std::endl; }
};

struct HidingDerived : HiddenBase {
    void f(float) { std::cout << "float" << std::endl; }
};

// ...

HiddenBase hb;
HidingDerived hd;
std::string s;

hb.f(1); // Output: int
hb.f(true); // Output: bool
hb.f(s); // Output: std::string;

hd.f(1.0); // Output: float
hd.f(3); // Output: float
hd.f(true); // Output: float
hd.f(s); // Error: Can't convert from std::string to float.
```

This is due to name resolution rules: During name lookup, once the correct name is found, we stop looking, even if we clearly haven't found the correct *version* of the entity with that name (such as with `hd.f(s)`); due to this, overloading the function in the derived class prevents name lookup from discovering the overloads in the base class. To avoid this, a `using-declaration` can be used to "import" names from the base class into the derived class, so

它们在名称查找时可用。

```
struct HidingDerived : HiddenBase {
    // 所有名为 HiddenBase::f 的成员在查找时都应视为 HidingDerived 的成员。
    using HiddenBase::f;

    void f(float) { std::cout << "float" << std::endl; }
};

// ...

HidingDerived hd;

hd.f(1.f); // 输出: float
hd.f(3); // 输出: int
hd.f(true); // 输出: bool
hd.f(s); // 输出: std::string
```

如果派生类使用 `using` 声明导入名称，但同时声明了与基类函数签名相同的函数，基类函数将被静默地覆盖或隐藏。

```
结构体 NamesHidden {
    虚函数 hide_me() {}
    虚函数 hide_me(float) {}
    函数 hide_me(int) {}
    函数 hide_me(bool) {}
};

结构体 NameHider : NamesHidden {
    使用 NamesHidden::hide_me;

    void hide_me() {} // 重写 NamesHidden::hide_me()。
    void hide_me(int) {} // 隐藏 NamesHidden::hide_me(int)。
};
```

`using` 声明也可以用来改变访问修饰符，前提是被导入的实体在基类中是 `public` 或 `protected` 的。

```
结构体 ProMem {
    保护成员:
        void func() {}
};

结构体 BecomesPub : ProMem {
    使用 ProMem::func;
};

// ...

ProMem pm;
BecomesPub bp;

pm.func(); // 错误: 受保护的。
bp.func(); // 好的。
```

同样地，如果我们明确想要调用继承层次中特定类的成员函数，可以在调用函数时限定函数名，指定该类的名称。

```
结构体 One {
```

that they will be available during name lookup.

```
struct HidingDerived : HiddenBase {
    // All members named HiddenBase::f shall be considered members of HidingDerived for lookup.
    using HiddenBase::f;

    void f(float) { std::cout << "float" << std::endl; }
};

// ...

HidingDerived hd;

hd.f(1.f); // Output: float
hd.f(3); // Output: int
hd.f(true); // Output: bool
hd.f(s); // Output: std::string
```

If a derived class imports names with a `using-declaration`, but also declares functions with the same signature as functions in the base class, the base class functions will silently be overridden or hidden.

```
结构体 NamesHidden {
    virtual void hide_me() {}
    virtual void hide_me(float) {}
    void hide_me(int) {}
    void hide_me(bool) {}
};

结构体 NameHider : NamesHidden {
    使用 NamesHidden::hide_me;

    void hide_me() {} // Overrides NamesHidden::hide_me().
    void hide_me(int) {} // Hides NamesHidden::hide_me(int).
};
```

A `using-declaration` can also be used to change access modifiers, provided the imported entity was `public` or `protected` in the base class.

```
结构体 ProMem {
    保护成员:
        void func() {}
};

结构体 BecomesPub : ProMem {
    使用 ProMem::func;
};

// ...

ProMem pm;
BecomesPub bp;

pm.func(); // Error: protected.
bp.func(); // Good.
```

Similarly, if we explicitly want to call a member function from a specific class in the inheritance hierarchy, we can qualify the function name when calling the function, specifying that class by name.

```
结构体 One {
```

```

虚函数 void f() { std::cout << "One." << std::endl; }
};

结构体 Two : One {
    void f() 重写 {
    One::f(); // this->One::f();
        std::cout << "Two." << std::endl;
    }
};

结构体 Three : Two {
    void f() 重写 {
    Two::f(); // this->Two::f();
        std::cout << "Three." << std::endl;
    }
};

// ...

Three t;

t.f();      // 正常语法。
t.Two::f(); // 调用 Two 中定义的 f() 版本。
t.One::f(); // 调用 One 中定义的 f() 版本。

```

第41.4节：虚成员函数

成员函数也可以声明为virtual。在这种情况下，如果通过指针或引用调用实例的成员函数，将不会直接访问；而是会在虚函数表（一个指向虚函数成员函数指针的列表，更常见的称为vtable或vftable）中查找函数，并使用该表调用适合实例动态（实际）类型的版本。如果函数是直接从类的变量调用，则不会进行查找。

```

结构体 Base {
    virtual void func() { std::cout << "在基类中." << std::endl; }
};

struct 派生类 : 基类 {
    void func() override { std::cout << "在派生类中." << std::endl; }
};

void slicer(基类 x) { x.func(); }

// ...

基类 b;
导出 d;

Base *pb = &b, *pd = &d; // 指针。
Base &rb = b, &rd = d; // 引用。

b.func(); // 输出：在 Base 中。
d.func(); // 输出：在 Derived 中。

pb->func(); // 输出：在 Base 中。
pd->func(); // 输出：在 Derived 中。

rb.func(); // 输出：在 Base 中。
rd.func(); // 输出：在 Derived 中。

```

```

virtual void f() { std::cout << "One." << std::endl; }
};

struct Two : One {
    void f() override {
    One::f(); // this->One::f();
        std::cout << "Two." << std::endl;
    }
};

struct Three : Two {
    void f() override {
    Two::f(); // this->Two::f();
        std::cout << "Three." << std::endl;
    }
};

// ...

Three t;

t.f();      // Normal syntax.
t.Two::f(); // Calls version of f() defined in Two.
t.One::f(); // Calls version of f() defined in One.

```

Section 41.4: Virtual Member Functions

Member functions can also be declared `virtual`. In this case, if called on a pointer or reference to an instance, they will not be accessed directly; rather, they will look up the function in the virtual function table (a list of pointers-to-member-functions for virtual functions, more commonly known as the vtable or vftable), and use that to call the version appropriate for the instance's dynamic (actual) type. If the function is called directly, from a variable of a class, no lookup is performed.

```

struct Base {
    virtual void func() { std::cout << "In Base." << std::endl; }
};

struct Derived : Base {
    void func() override { std::cout << "In Derived." << std::endl; }
};

void slicer(Base x) { x.func(); }

// ...

Base b;
Derived d;

Base *pb = &b, *pd = &d; // Pointers.
Base &rb = b, &rd = d; // References.

b.func(); // Output: In Base.
d.func(); // Output: In Derived.

pb->func(); // Output: In Base.
pd->func(); // Output: In Derived.

rb.func(); // Output: In Base.
rd.func(); // Output: In Derived.

```

```
slicer(b); // 输出：在 Base 中。  
slicer(d); // 输出：以 Base 为基准。
```

请注意，虽然 `pd` 是 `Base*`，且 `rd` 是 `Base&`，但对这两个调用中的任意一个调用 `func()` 时，调用的是 `Derived::func()` 而不是 `Base::func()`；这是因为 `Derived` 的 vtable 更新了 `Base::func()` 的入口，使其指向 `Derived::func()`。相反，注意将实例传递给 `slicer()` 总是调用 `Base::func()`，即使传入的实例是 `Derived`；这是由于所谓的 数据切片 (data slicing)，将 `Derived` 实例按值传递给 `Base` 参数时，`Derived` 实例中非 `Base` 部分将无法访问。

当成员函数被定义为虚函数时，所有具有相同签名的派生类成员函数都会覆盖它，无论覆盖函数是否显式声明为 `virtual`。这可能使程序员更难理解派生类，因为没有指示哪个函数是 `virtual`。

```
struct B {  
    virtual void f() {}  
};  
  
struct D : B {  
    void f() {} // 隐式为虚函数，覆盖了 B::f.  
               // 不过你得查看 B 才能知道这一点。  
};
```

但请注意，派生函数只有在签名匹配时才会覆盖基类函数；即使派生函数显式声明为 `virtual`，如果签名不匹配，它将创建一个新的虚函数。

```
struct BadB {  
    virtual void f() {}  
};  
  
结构体 BadD : BadB {  
    虚函数 f(int i) {} // 不会覆盖 BadB::f。  
};  
版本 ≥ C++11
```

从 C++11 开始，可以使用上下文敏感关键字 `override` 明确表示覆盖意图。这告诉编译器程序员期望它覆盖基类函数，如果它没有覆盖任何函数，编译器会报错。

```
结构体 CPP11B {  
    virtual void f() {}  
};  
  
结构体 CPP11D : CPP11B {  
    void f() override {}  
    void f(int i) override {} // 错误：实际上没有覆盖任何函数。  
};
```

这还有一个好处，就是告诉程序员该函数既是虚函数，也至少在一个基类中声明过，这可以使复杂类更易于理解。

当函数被声明为 `virtual`，并且在类定义外定义时，函数声明中必须包含 `virtual` 说明符，而定义中不应重复。

版本 ≥ C++11

```
slicer(b); // Output: In Base.  
slicer(d); // Output: In Base.
```

Note that while `pd` is `Base*`, and `rd` is a `Base&`, calling `func()` on either of the two calls `Derived::func()` instead of `Base::func()`; this is because the vtable for `Derived` updates the `Base::func()` entry to instead point to `Derived::func()`. Conversely, note how passing an instance to `slicer()` always results in `Base::func()` being called, even when the passed instance is a `Derived`; this is because of something known as *data slicing*, where passing a `Derived` instance into a `Base` parameter by value renders the portion of the `Derived` instance that isn't a `Base` instance inaccessible.

When a member function is defined as `virtual`, all derived class member functions with the same signature override it, regardless of whether the overriding function is specified as `virtual` or not. This can make derived classes harder for programmers to parse, however, as there's no indication as to which function(s) is/are `virtual`.

```
struct B {  
    virtual void f() {}  
};  
  
struct D : B {  
    void f() {} // Implicitly virtual, overrides B::f.  
               // You'd have to check B to know that, though.  
};
```

Note, however, that a derived function only overrides a base function if their signatures match; even if a derived function is explicitly declared `virtual`, it will instead create a new virtual function if the signatures are mismatched.

```
struct BadB {  
    virtual void f() {}  
};  
  
struct BadD : BadB {  
    virtual void f(int i) {} // Does NOT override BadB::f.  
};  
Version ≥ C++11
```

As of C++11, intent to override can be made explicit with the context-sensitive keyword `override`. This tells the compiler that the programmer expects it to override a base class function, which causes the compiler to omit an error if it *doesn't* override anything.

```
struct CPP11B {  
    virtual void f() {}  
};  
  
struct CPP11D : CPP11B {  
    void f() override {}  
    void f(int i) override {} // Error: Doesn't actually override anything.  
};
```

This also has the benefit of telling programmers that the function is both `virtual`, and also declared in at least one base class, which can make complex classes easier to parse.

When a function is declared `virtual`, and defined outside the class definition, the `virtual` specifier must be included in the function declaration, and not repeated in the definition.

Version ≥ C++11

这对override同样适用。

```
结构体 VB {
    虚函数 f(); // "virtual"写在这里。
    无返回值函数 g();
};

/* virtual */ 无返回值函数 VB::f() {} // 这里没有。
虚函数 VB::g() {} // 错误。
```

如果基类重载了一个虚函数，只有明确指定为虚函数的重载才是虚函数。

```
结构体 BOverload {
    虚函数 func() {}
    无返回值函数 func(int) {}
};

结构体 DOverload : BOverload {
    无返回值函数 func() 覆盖 {}
    无返回值函数 func(int) {}
};

// ...

BOverload* bo = new DOverload;
bo->func(); // 调用 DOverload::func().
bo->func(1); // 调用 BOverload::func(int).
```

欲了解更多信息，请参阅相关主题。

第41.5节：常量正确性

cv限定符的主要用途之一是常量正确性。这是一种保证只有需要修改对象的访问才能修改该对象的做法，且任何不需要修改对象的（成员或非成员）函数都没有对该对象的写访问权限（无论是直接还是间接）。这防止了无意的修改，使代码更不易出错。它还允许任何不需要修改状态的函数能够接受常量或非常量对象，而无需重写或重载函数。

常量正确性由于其本质，是自下而上的：任何不需要改变状态的类成员函数都声明为const，以便可以在const实例上调用。反过来，这允许传引用的参数在不需要修改时声明为const，这使得函数可以接受const或非const对象而不会报错，const性质可以以这种方式向外传播。

因此，getter函数通常是const，任何不需要修改逻辑状态的其他函数也是如此。

```
class ConstIncorrect {
    Field fld;

public:
    ConstIncorrect(const Field& f) : fld(f) {} // 修改。

    const Field& get_field() { return fld; } // 不修改；应为const。
    void set_field(const Field& f) { fld = f; } // 修改。

    void do_something(int i) { // 修改。
        fld.insert_value(i);
    }
    void do_nothing() { } // 不修改；应为 const。
};
```

This also holds true for override.

```
struct VB {
    virtual void f(); // "virtual" goes here.
    void g();
};
/* virtual */ void VB::f() {} // Not here.
virtual void VB::g() {} // Error.
```

If a base class overloads a virtual function, only overloads that are explicitly specified as virtual will be virtual.

```
struct BOverload {
    virtual void func() {}
    void func(int) {}
};

struct DOverload : BOverload {
    void func() override {}
    void func(int) {}
};

// ...

BOverload* bo = new DOverload;
bo->func(); // Calls DOverload::func().
bo->func(1); // Calls BOverload::func(int).
```

For more information, see the relevant topic.

Section 41.5: Const Correctness

One of the primary uses for this cv-qualifiers is const correctness. This is the practice of guaranteeing that only accesses that need to modify an object are able to modify the object, and that any (member or non-member) function that doesn't need to modify an object doesn't have write access to that object (whether directly or indirectly). This prevents unintentional modifications, making code less error-prone. It also allows any function that doesn't need to modify state to be able to take either a const or non-const object, without needing to rewrite or overload the function.

const correctness, due to its nature, starts at the bottom up: Any class member function that doesn't need to change state is declared as const, so that it can be called on const instances. This, in turn, allows passed-by-reference parameters to be declared const when they don't need to be modified, which allows functions to take either const or non-const objects without complaining, and const-ness can propagate outwards in this manner. Due to this, getters are frequently const, as are any other functions that don't need to modify logical state.

```
class ConstIncorrect {
    Field fld;

public:
    ConstIncorrect(const Field& f) : fld(f) {} // Modifies.

    const Field& get_field() { return fld; } // Doesn't modify; should be const.
    void set_field(const Field& f) { fld = f; } // Modifies.

    void do_something(int i) { // Modifies.
        fld.insert_value(i);
    }
    void do_nothing() { } // Doesn't modify; should be const.
};
```

```

class ConstCorrect {
    Field fld;

public:
ConstCorrect(const Field& f) : fld(f) {}      // 非 const: 会修改。

const Field& get_field() const { return fld; } // const: 不修改。
void set_field(const Field& f) { fld = f; }   // 非 const: 会修改。

    void do_something(int i) {                  // 非 const: 会修改。
fld.insert_value(i);
}
    void do_nothing() const { }                // const: 不修改。
};

// ...

const ConstIncorrect i_cant_do_anything(make_me_a_field());
// 现在, 让我们读取它...
Field f = i_cant_do_anything.get_field();
// 错误: 丢失 cv 限定符, get_field() 不是 const。
i_cant_do_anything.do_nothing();
// 错误: 同上。
// 哟呀。

const ConstCorrect but_i_can(make_me_a_field());
// 现在, 让我们读取它...
Field f = but_i_can.get_field(); // 好的。
but_i_can.do_nothing();        // 好的。

```

正如对 ConstIncorrect 和 ConstCorrect 的注释所示, 正确地使用 cv 限定符修饰函数也起到了文档说明的作用。如果一个类是 const 正确的, 任何不是 const 的函数都可以安全地假设会改变状态, 而任何是 const 的函数都可以安全地假设不会改变状态。

```

class ConstCorrect {
    Field fld;

public:
ConstCorrect(const Field& f) : fld(f) {}      // Not const: Modifies.

const Field& get_field() const { return fld; } // const: Doesn't modify.
void set_field(const Field& f) { fld = f; }   // Not const: Modifies.

    void do_something(int i) {                  // Not const: Modifies.
fld.insert_value(i);
}
    void do_nothing() const { }                // const: Doesn't modify.
};

// ...

const ConstIncorrect i_cant_do_anything(make_me_a_field());
// Now, let's read it...
Field f = i_cant_do_anything.get_field();
// Error: Loses cv-qualifiers, get_field() isn't const.
i_cant_do_anything.do_nothing();
// Error: Same as above.
// Oops.

const ConstCorrect but_i_can(make_me_a_field());
// Now, let's read it...
Field f = but_i_can.get_field(); // Good.
but_i_can.do_nothing();        // Good.

```

As illustrated by the comments on ConstIncorrect and ConstCorrect, properly cv-qualifying functions also serves as documentation. If a class is `const` correct, any function that isn't `const` can safely be assumed to change state, and any function that is `const` can safely be assumed not to change state.

第42章：常量类成员函数

第42.1节：常量成员函数

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

using namespace std;

class A {
public:
    map<string, string> * mapOfStrings;
public:
    A() {
        mapOfStrings = new map<string, string>();
    }

    void insertEntry(string const & key, string const & value) const {
        (*mapOfStrings)[key] = value;           // 这行代码能用吗？能用。
        delete mapOfStrings;                  // 这行代码也能用
        mapOfStrings = new map<string, string>(); // 这行代码*不能*用
    }

    void refresh() {
        delete mapOfStrings;
    mapOfStrings = new map<string, string>(); // 由于refresh是非const函数，这行代码能用
    }

    void getEntry(string const & key) const {
        cout << mapOfStrings->at(key);
    }
};

int main(int argc, char* argv[]) {

    A var;
    var.insertEntry("abc", "abcValue");
    var.getEntry("abc");
    getchar();
    return 0;
}
```

Chapter 42: Constant class member functions

Section 42.1: constant member function

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

using namespace std;

class A {
public:
    map<string, string> * mapOfStrings;
public:
    A() {
        mapOfStrings = new map<string, string>();
    }

    void insertEntry(string const & key, string const & value) const {
        (*mapOfStrings)[key] = value;           // This works? Yes it does.
        delete mapOfStrings;                  // This also works
        mapOfStrings = new map<string, string>(); // This * does * not work
    }

    void refresh() {
        delete mapOfStrings;
        mapOfStrings = new map<string, string>(); // Works as refresh is non const function
    }

    void getEntry(string const & key) const {
        cout << mapOfStrings->at(key);
    }
};

int main(int argc, char* argv[]) {

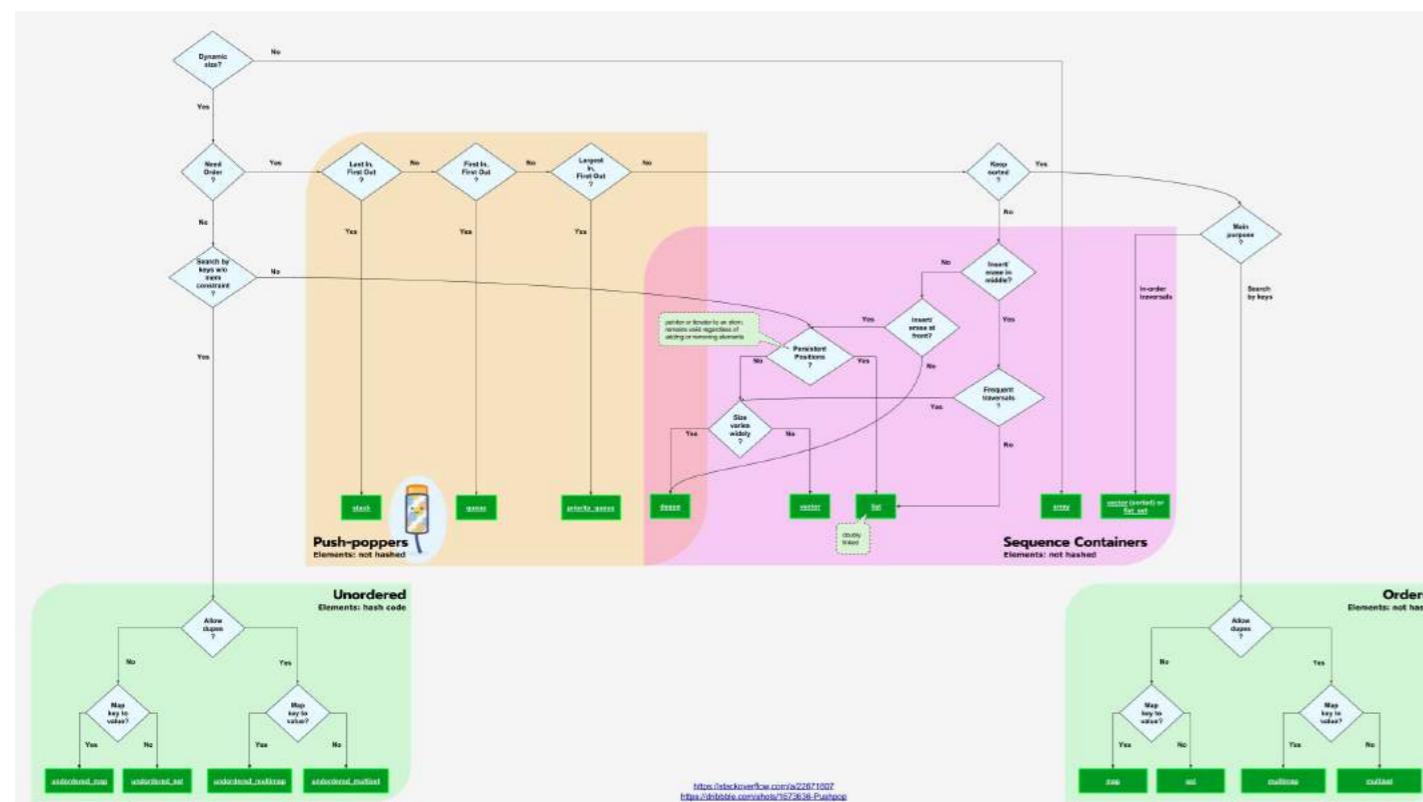
    A var;
    var.insertEntry("abc", "abcValue");
    var.getEntry("abc");
    getchar();
    return 0;
}
```

第43章：C++容器

C++容器存储一组元素。容器包括向量、列表、映射等。使用模板，C++容器可以包含基本类型（例如int）或自定义类（例如MyClass）的集合。

第43.1节：C++容器流程图

选择使用哪种C++容器可能很棘手，下面是一个简单的流程图，帮助决定哪种容器适合这项工作。



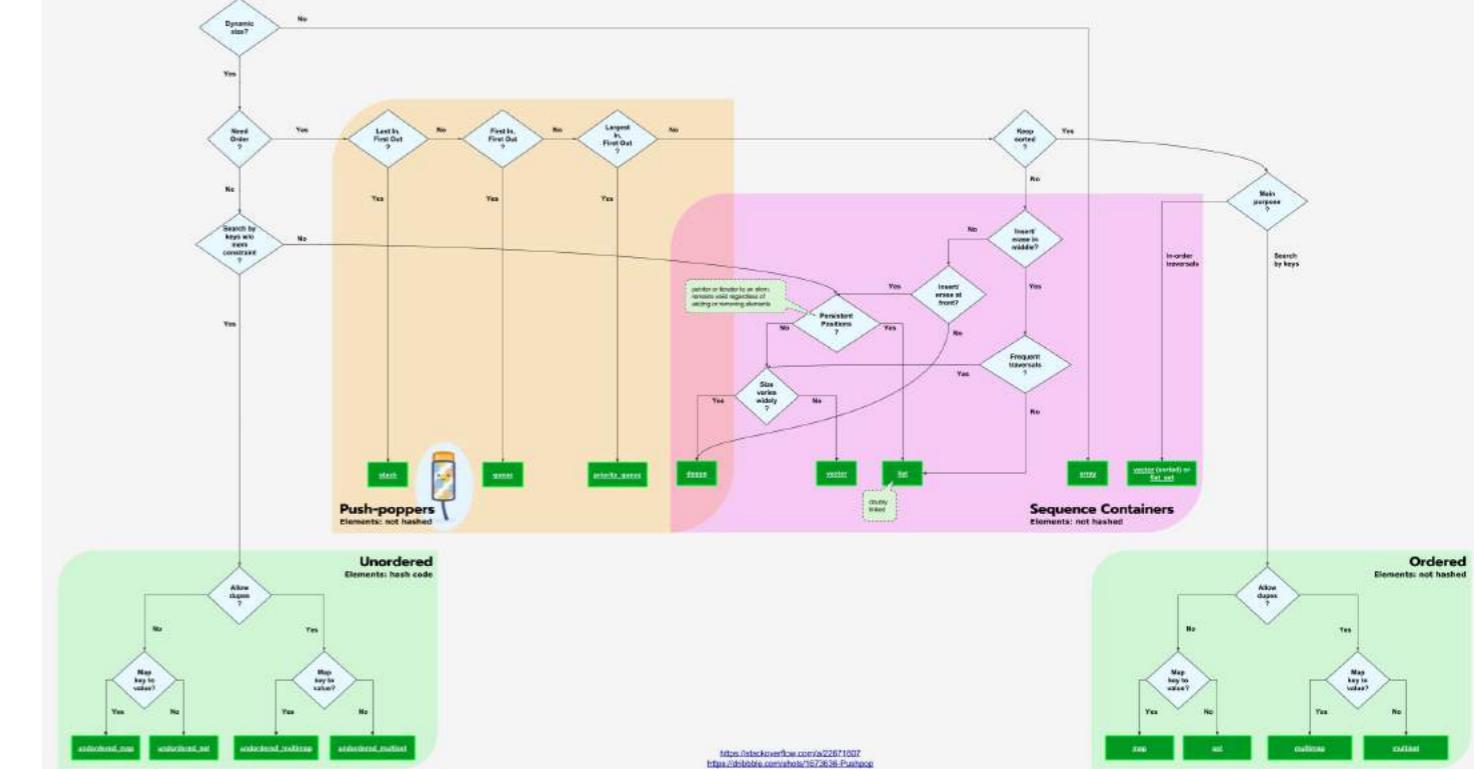
此流程图基于Mikael Persson的帖子。流程图中的小图形来自Megan Hopkins。

Chapter 43: C++ Containers

C++ containers store a collection of elements. Containers include vectors, lists, maps, etc. Using Templates, C++ containers contain collections of primitives (e.g. ints) or custom classes (e.g. MyClass).

Section 43.1: C++ Containers Flowchart

Choosing which C++ Container to use can be tricky, so here's a simple flowchart to help decide which Container is right for the job.



This flowchart was based on [Mikael Persson's post](#). This little graphic in the flowchart is from [Megan Hopkins](#).

第44章：命名空间

用于防止使用多个库时的名称冲突，命名空间是函数、类、类型等的声明性前缀。

第44.1节：什么是命名空间？

C++命名空间是一组C++实体（函数、类、变量），其名称以命名空间名称为前缀。在命名空间内编写代码时，属于该命名空间的命名实体无需加命名空间前缀，但命名空间外的实体必须使用完全限定名。完全限定名的格式为`<name space>::<entity>`。例如：

```
namespace Example
{
    const int test = 5;

    const int test2 = test + 12; // 在 `Example` 命名空间内有效
}

const int test3 = test + 3; // 失败；`test` 在命名空间外未找到。

const int test3 = Example::test + 3; // 成功；使用了完全限定名。
```

命名空间对于将相关定义分组非常有用。打个比方，购物中心通常被划分为多个商店，每个商店销售特定类别的商品。一个商店可能卖电子产品，而另一个商店可能卖鞋子。这些商店类型的逻辑划分帮助顾客找到他们想要的商品。命名空间帮助C++程序员，就像顾客一样，通过逻辑组织函数、类和变量，方便他们找到所需的内容。例如：

```
namespace Electronics
{
    int TotalStock;
    class Headphones
    {
        // 耳机的描述（颜色、品牌、型号等）
    };
    class Television
    {
        // 电视的描述（颜色、品牌、型号等）
    };
}

namespace Shoes
{
    int TotalStock;
    class 凉鞋
    {
        // 凉鞋的描述（颜色、品牌、型号等）
    };
    class 拖鞋
    {
        // 拖鞋的描述（颜色、品牌、型号等）
    };
}
```

预定义了一个命名空间，即没有名称的全局命名空间，但可以用`::`表示。例如：

Chapter 44: Namespaces

Used to prevent name collisions when using multiple libraries, a namespace is a declarative prefix for functions, classes, types, etc.

Section 44.1: What are namespaces?

A C++ namespace is a collection of C++ entities (functions, classes, variables), whose names are prefixed by the name of the namespace. When writing code within a namespace, named entities belonging to that namespace need not be prefixed with the namespace name, but entities outside of it must use the fully qualified name. The fully qualified name has the format `<namespace>::<entity>`. Example:

```
namespace Example
{
    const int test = 5;

    const int test2 = test + 12; // Works within `Example` namespace
}

const int test3 = test + 3; // Fails; `test` not found outside of namespace.

const int test3 = Example::test + 3; // Works; fully qualified name used.
```

Namespaces are useful for grouping related definitions together. Take the analogy of a shopping mall. Generally a shopping mall is split up into several stores, each store selling items from a specific category. One store might sell electronics, while another store might sell shoes. These logical separations in store types help the shoppers find the items they're looking for. Namespaces help C++ programmers, like shoppers, find the functions, classes, and variables they're looking for by organizing them in a logical manner. Example:

```
namespace Electronics
{
    int TotalStock;
    class Headphones
    {
        // Description of a Headphone (color, brand, model number, etc.)
    };
    class Television
    {
        // Description of a Television (color, brand, model number, etc.)
    };
}

namespace Shoes
{
    int TotalStock;
    class Sandal
    {
        // Description of a Sandal (color, brand, model number, etc.)
    };
    class Slipper
    {
        // Description of a Slipper (color, brand, model number, etc.)
    };
}
```

There is a single namespace predefined, which is the global namespace that has no name, but can be denoted by`::`. Example:

```

void bar() {
    // 定义在全局命名空间中
}
namespace foo {
    void bar() {
        // 定义在命名空间 foo 中
    }
    void barbar() {
        bar(); // 调用 foo::bar()
        ::bar(); // 调用定义在全局命名空间的 bar()
    }
}

```

第44.2节：参数依赖查找

当调用一个没有显式命名空间限定符的函数时，如果该函数的某个参数类型也属于某个命名空间，编译器可以选择调用该命名空间内的函数。这称为“参数依赖查找”（Argument Dependent Lookup，简称 ADL）：

```

namespace Test
{
    int call(int i);

    class SomeClass {...};

    int call_too(const SomeClass &data);
}

```

call(5); // 失败。不是限定的函数名。

```

Test::SomeClass data;

call_too(data); // 成功

```

call 失败是因为它的参数类型都不属于 Test 命名空间。call_too 能成功是因为 SomeClass 是 Test 的成员，因此符合 ADL 规则。

ADL 何时不会发生

如果普通的无限定查找找到类成员、在块作用域声明的函数或非函数类型的内容，则不会发生ADL（论域查找）。例如：

```

void foo();
namespace N {
    struct X {};
    void foo(X) { std::cout << '1'; }
    void qux(X) { std::cout << '2'; }
}

struct C {
    void foo() {}
    void bar() {}
foo(N::X{}); // 错误：ADL被禁用且C::foo()不接受任何参数
}

void bar() {
    extern void foo(); // 重新声明了 ::foo
foo(N::X{}); // 错误：ADL被禁用且::foo()不接受任何参数

```

```

void bar() {
    // defined in global namespace
}
namespace foo {
    void bar() {
        // defined in namespace foo
    }
    void barbar() {
        bar(); // calls foo::bar()
        ::bar(); // calls bar() defined in global namespace
    }
}

```

Section 44.2: Argument Dependent Lookup

When calling a function without an explicit namespace qualifier, the compiler can choose to call a function within a namespace if one of the parameter types to that function is also in that namespace. This is called "Argument Dependent Lookup", or ADL:

```

namespace Test
{
    int call(int i);

    class SomeClass {...};

    int call_too(const SomeClass &data);
}

call(5); // Fails. Not a qualified function name.

Test::SomeClass data;

call_too(data); // Succeeds

```

call fails because none of its parameter types come from the Test namespace. call_too works because SomeClass is a member of Test and therefore it qualifies for ADL rules.

When does ADL not occur

ADL does not occur if normal unqualified lookup finds a class member, a function that has been declared at block scope, or something that is not of function type. For example:

```

void foo();
namespace N {
    struct X {};
    void foo(X) { std::cout << '1'; }
    void qux(X) { std::cout << '2'; }
}

struct C {
    void foo() {}
    void bar() {}
    foo(N::X{}); // error: ADL is disabled and C::foo() takes no arguments
}

void bar() {
    extern void foo(); // redeclares ::foo
    foo(N::X{}); // error: ADL is disabled and ::foo() doesn't take any arguments
}

```

```

}

int qux;

void baz() {
    qux(N::X{}); // 错误：变量声明禁用了“qux”的ADL
}

```

第44.3节：扩展命名空间

命名空间的一个有用特性是你可以扩展它们（向其中添加成员）。

```

namespace Foo
{
    void bar() {}
}

//其他一些内容

namespace Foo
{
    void bar2() {}
}

```

第44.4节：使用指令

关键字 'using' 有三种用法。与关键字 'namespace' 结合使用时，您可以编写一个 'using 指令'：

如果你不想在命名空间Foo中的每个内容前都写Foo::，可以使用`using namespace Foo;`来导入Foo中的所有内容。

```

namespace Foo
{
    void bar() {}
    void baz() {}
}

//必须使用 Foo::bar()
Foo::bar();

//导入 Foo
using namespace Foo;
bar(); //可以
baz(); //可以

```

也可以只导入命名空间中的选定实体，而不是整个命名空间：

```

using Foo::bar;
bar(); //可以，已被专门导入
baz(); //不可以，未被导入

```

需要注意的是：在头文件中使用`using namespace`在大多数情况下被视为不良风格。如果这样做，该命名空间会被导入到包含该头文件的每个文件中。由于没有办法“取消使用”一个命名空间，这可能导致命名空间污染（全局命名空间中出现更多或意外的符号）或者更糟的是，产生冲突。以下示例说明了这个问题：

```
***** foo.h *****
```

```

}

int qux;

void baz() {
    qux(N::X{}); // error: variable declaration disables ADL for "qux"
}

```

Section 44.3: Extending namespaces

A useful feature of [namespaces](#) is that you can expand them (add members to it).

```

namespace Foo
{
    void bar() {}
}

//some other stuff

namespace Foo
{
    void bar2() {}
}

```

Section 44.4: Using directive

The keyword 'using' has three flavors. Combined with keyword 'namespace' you write a 'using directive':

If you don't want to write `Foo::` in front of every stuff in the namespace Foo, you can use `using namespace Foo;` to import every single thing out of Foo.

```

namespace Foo
{
    void bar() {}
    void baz() {}
}

//Have to use Foo::bar()
Foo::bar();

//Import Foo
using namespace Foo;
bar(); //OK
baz(); //OK

```

It is also possible to import selected entities in a namespace rather than the whole namespace:

```

using Foo::bar;
bar(); //OK, was specifically imported
baz(); // Not OK, was not imported

```

A word of caution: `using namespaces` in header files is seen as bad style in most cases. If this is done, the namespace is imported in *every* file that includes the header. Since there is no way of "un-using" a namespace, this can lead to namespace pollution (more or unexpected symbols in the global namespace) or, worse, conflicts. See this example for an illustration of the problem:

```
***** foo.h *****
```

```

namespace Foo
{
    class C;
}

***** bar.h *****
namespace Bar
{
    class C;
}

***** baz.h *****
#include "foo.h"
using namespace Foo;

***** main.cpp *****
#include "bar.h"
#include "baz.h"

using namespace Bar;
C c; // 错误：Bar::C 和 Foo::C 之间存在歧义

```

在类作用域中不能出现using-directive。

第44.5节：创建命名空间

创建命名空间非常简单：

```

//创建命名空间 foo
namespace Foo
{
    //在命名空间 foo 中声明函数 bar
    void bar() {}
}

```

调用bar时，必须先指定命名空间，然后是作用域解析运算符::：

```
Foo::bar();
```

允许在一个命名空间中创建另一个命名空间，例如：

```

namespace A
{
    namespace B
    {
        namespace C
        {
            void bar() {}
        }
    }
}

```

版本 ≥ C++17

上述代码可以简化为以下内容：

```

namespace A::B::C
{
    void bar() {}
}

```

```

namespace Foo
{
    class C;
}

***** bar.h *****
namespace Bar
{
    class C;
}

***** baz.h *****
#include "foo.h"
using namespace Foo;

***** main.cpp *****
#include "bar.h"
#include "baz.h"

using namespace Bar;
C c; // error: Ambiguity between Bar::C and Foo::C

```

A *using-directive* cannot occur at class scope.

Section 44.5: Making namespaces

Creating a namespace is really easy:

```

//Creates namespace foo
namespace Foo
{
    //Declares function bar in namespace foo
    void bar() {}
}

```

To call bar, you have to specify the namespace first, followed by the scope resolution operator :::

```
Foo::bar();
```

It is allowed to create one namespace in another, for example:

```

namespace A
{
    namespace B
    {
        namespace C
        {
            void bar() {}
        }
    }
}

```

Version ≥ C++17

The above code could be simplified to the following:

```

namespace A::B::C
{
    void bar() {}
}

```

}

第44.6节：无名/匿名命名空间

无名命名空间可用于确保名称具有内部链接（只能被当前翻译单元引用）。这种命名空间的定义方式与其他命名空间相同，但没有名称：

```
namespace {
    int foo = 42;
}
```

`foo` 仅在其出现的翻译单元中可见。

建议不要在头文件中使用无名命名空间，因为这会为每个包含它的翻译单元生成内容的副本。如果定义了非常量全局变量，这一点尤其重要。

```
// foo.h
namespace {
std::string globalString;
}

// 1.cpp
#include "foo.h" //< 生成 unnamed_namespace{1.cpp}::globalString ...

globalString = "Initialize";

// 2.cpp
#include "foo.h" //< 生成 unnamed_namespace{2.cpp}::globalString ...

std::cout << globalString; //< 将始终打印空字符串
```

}

Section 44.6: Unnamed/anonymous namespaces

An unnamed namespace can be used to ensure names have internal linkage (can only be referred to by the current translation unit). Such a namespace is defined in the same way as any other namespace, but without the name:

```
namespace {
    int foo = 42;
}
```

`foo` 只能在其出现的翻译单元中可见。

It is recommended to never use unnamed namespaces in header files as this gives a version of the content for every translation unit it is included in. This is especially important if you define non-const globals.

```
// foo.h
namespace {
    std::string globalString;
}

// 1.cpp
#include "foo.h" //< Generates unnamed_namespace{1.cpp}::globalString ...

globalString = "Initialize";

// 2.cpp
#include "foo.h" //< Generates unnamed_namespace{2.cpp}::globalString ...

std::cout << globalString; //< Will always print the empty string
```

第44.7节：紧凑嵌套命名空间

版本 ≥ C++17

```
namespace a {
    namespace b {
        template<class T>
        struct qualifies : std::false_type {};
    }

    namespace other {
        struct bob {};
    }

    namespace a::b {
        template<>
        struct qualifies<::other::bob> : std::true_type {};
    }
}
```

从 C++17 开始，可以通过 `namespace a::b` 一步进入 `a` 和 `b` 命名空间。

第44.8节：命名空间别名

可以使用 `namespace` 标识符 = 语法为命名空间指定别名（即同一命名空间的另一个名称）。通过使用别名名称限定，可以访问别名命名空间的成员。在

Version ≥ C++17

```
namespace a {
    namespace b {
        template<class T>
        struct qualifies : std::false_type {};
    }

    namespace other {
        struct bob {};
    }

    namespace a::b {
        template<>
        struct qualifies<::other::bob> : std::true_type {};
    }
}
```

You can enter both the `a` and `b` namespaces in one step with `namespace a::b` starting in C++17.

Section 44.8: Namespace alias

A namespace can be given an alias (*i.e.*, another name for the same namespace) using the `namespace identifier =` syntax. Members of the aliased namespace can be accessed by qualifying them with the name of the alias. In the

下面的示例中，嵌套命名空间 AReallyLongName::AnotherReallyLongName 输入起来不方便，因此函数 qux 在本地声明了别名 N。之后可以简单地使用 N:: 访问该命名空间的成员。

```
namespace AReallyLongName {
    namespace AnotherReallyLongName {
        int foo();
        int bar();
        void baz(int x, int y);
    }
}
void qux() {
    namespace N = AReallyLongName::AnotherReallyLongName;
    N::baz(N::foo(), N::bar());
}
```

following example, the nested namespace AReallyLongName::AnotherReallyLongName is inconvenient to type, so the function qux locally declares an alias N. Members of that namespace can then be accessed simply using N::.

```
namespace AReallyLongName {
    namespace AnotherReallyLongName {
        int foo();
        int bar();
        void baz(int x, int y);
    }
}
void qux() {
    namespace N = AReallyLongName::AnotherReallyLongName;
    N::baz(N::foo(), N::bar());
}
```

第44.9节：内联命名空间

版本 ≥ C++11

内联命名空间 将内联命名空间的内容包含在封闭命名空间中，因此

```
命名空间 Outer
{
    内联命名空间 Inner
    {
        void foo();
    }
}
```

大致等同于

```
命名空间 Outer
{
    命名空间 Inner
    {
        void foo();
    }

    using Inner::foo;
}
```

但来自 Outer::Inner:: 的元素与那些关联到 Outer:: 的元素是相同的。

因此以下等价

```
Outer::foo();
Outer::Inner::foo();
```

使用namespace Inner;的替代方案在某些复杂部分如模板特化时并不等价：

用于

```
#include <outer.h> // 见下文

class MyCustomType;
命名空间 Outer
{
```

Section 44.9: Inline namespace

Version ≥ C++11

inline namespace 包括内联命名空间的内容在封闭命名空间中，所以

```
namespace Outer
{
    inline namespace Inner
    {
        void foo();
    }
}
```

大致等同于

```
namespace Outer
{
    命名空间 Inner
    {
        void foo();
    }

    using Inner::foo;
}
```

但来自 Outer::Inner:: 的元素与那些关联到 Outer:: 的元素是相同的。

因此以下等价

```
Outer::foo();
Outer::Inner::foo();
```

The alternative using namespace Inner; would not be equivalent for some tricky parts as template specialization:

For

```
#include <outer.h> // See below

class MyCustomType;
namespace Outer
{
```

```
template <>
void foo<MyCustomType>() { std::cout << "特化"; }
}
```

- 内联命名空间允许对Outer::foo进行特化

```
// outer.h
// 为简化省略了包含保护

namespace Outer
{
    // 内联命名空间 Inner
    {
        template <typename T>
        void foo() { std::cout << "通用"; }
    }
}
```

- 而using namespace不允许对Outer::foo进行特化

```
// outer.h
// 为简化省略了包含保护

namespace Outer
{
    // 命名空间 Inner
    {
        template <typename T>
        void foo() { std::cout << "通用"; }
    }
    using namespace Inner;
    // 无法对 `Outer::foo` 进行特化
    // 应该是 `Outer::Inner::foo`。
}
}
```

内联命名空间是一种允许多个版本共存并默认使用inline版本的方法

```
namespace MyNamespace
{
    // 内联最后一个版本
    inline namespace Version2
    {
        void foo(); // 新版本
        void bar();
    }

    namespace Version1 // 旧版本
    {
        void foo();
    }
}
```

以及使用方式

```
MyNamespace::Version1::foo(); // 旧版本
MyNamespace::Version2::foo(); // 新版本
MyNamespace::foo();          // 默认版本 : MyNamespace::Version1::foo();
```

```
template <>
void foo<MyCustomType>() { std::cout << "Specialization"; }
}
```

- The inline namespace allows the specialization of Outer::foo

```
// outer.h
// include guard omitted for simplification

namespace Outer
{
    inline namespace Inner
    {
        template <typename T>
        void foo() { std::cout << "Generic"; }
    }
}
```

- Whereas the using namespace doesn't allow the specialization of Outer::foo

```
// outer.h
// include guard omitted for simplification

namespace Outer
{
    namespace Inner
    {
        template <typename T>
        void foo() { std::cout << "Generic"; }
    }
    using namespace Inner;
    // Specialization of `Outer::foo` is not possible
    // it should be `Outer::Inner::foo`.
}
}
```

Inline namespace is a way to allow several version to cohabit and defaulting to the inline one

```
namespace MyNamespace
{
    // Inline the last version
    inline namespace Version2
    {
        void foo(); // New version
        void bar();
    }

    namespace Version1 // The old one
    {
        void foo();
    }
}
```

And with usage

```
MyNamespace::Version1::foo(); // old version
MyNamespace::Version2::foo(); // new version
MyNamespace::foo();          // default version : MyNamespace::Version1::foo();
```

第44.10节：为长命名空间创建别名

这通常用于重命名或缩短对长命名空间的引用，例如引用库的组件。

```
namespace boost
{
    namespace multiprecision
    {
        class Number ...
    }
}

namespace Name1 = boost::multiprecision;

// 两种类型声明是等价的
boost::multiprecision::Number X // 写出完整的命名空间路径，更长
Name1::Number Y // 使用别名，更短
```

Section 44.10: Aliasing a long namespace

This is usually used for renaming or shortening long namespace references such referring to components of a library.

```
namespace boost
{
    namespace multiprecision
    {
        class Number ...
    }
}

namespace Name1 = boost::multiprecision;

// Both Type declarations are equivalent
boost::multiprecision::Number X // Writing the full namespace path, longer
Name1::Number Y // using the name alias, shorter
```

第44.11节：别名声明的作用域

别名声明会受到之前using语句的影响

```
namespace boost
{
    namespace multiprecision
    {
        class Number ...
    }
}

using namespace boost;

// 两个命名空间是等价的
namespace Name1 = boost::multiprecision;
namespace Name2 = multiprecision;
```

然而，当你遇到如下情况时，更容易混淆你正在为哪个命名空间创建别名：

```
namespace boost
{
    namespace multiprecision
    {
        class Number ...
    }
}

namespace numeric
{
    namespace multiprecision
    {
        class Number ...
    }
}

使用命名空间 numeric;
使用命名空间 boost;
```

Section 44.11: Alias Declaration scope

Alias Declaration are affected by preceding *using* statements

```
namespace boost
{
    namespace multiprecision
    {
        class Number ...
    }
}

using namespace boost;

// Both Namespace are equivalent
namespace Name1 = boost::multiprecision;
namespace Name2 = multiprecision;
```

However, it is easier to get confused over which namespace you are aliasing when you have something like this:

```
namespace boost
{
    namespace multiprecision
    {
        class Number ...
    }
}

namespace numeric
{
    namespace multiprecision
    {
        class Number ...
    }
}

using namespace numeric;
using namespace boost;
```

```
// 不推荐使用，因为  
// 不明确 Name1 是指 numeric::multiprecision 还是 boost::multiprecision  
namespace Name1 = multiprecision;  
  
// 为了清晰起见，建议使用绝对路径  
  
namespace Name2 = numeric::multiprecision;  
namespace Name3 = boost::multiprecision;
```

```
// Not recommended as  
// its not explicitly clear whether Name1 refers to  
// numeric::multiprecision or boost::multiprecision  
namespace Name1 = multiprecision;  
  
// For clarity, its recommended to use absolute paths  
// instead  
namespace Name2 = numeric::multiprecision;  
namespace Name3 = boost::multiprecision;
```

第45章：头文件

第45.1节：基本示例

下面的示例将包含一段代码块，表示将被拆分到多个源文件中，如
// 文件名 注释所示。

源文件

```
// my_function.h

/* 注意这个头文件只包含了一个函数的声明。
 * 头文件中的函数通常不定义声明的实现
 * 除非代码必须在编译时进一步处理，比如模板。
 */

/* 通常头文件还会包含预处理器保护，以确保每个头文件
 * 不会被包含两次。
 *
 * 保护通过检查头文件唯一的预处理器标识符是否已定义来实现， * 只有在之前未包含过该头文件时
才包含它。
 */

#ifndef MY_FUNCTION_H
#define MY_FUNCTION_H

// global_value 和 my_function() 将会被
// 如果此头文件被不同文件包含，则被视为相同的构造体。
const int global_value = 42;
int my_function();

#endif // MY_FUNCTION_H
```

```
// my_function.cpp

/* 注意对应的源文件如何包含头文件中定义的接口， * 以便编译器知道源文件正在实现什么内容。
 *
 * 在本例中，源文件需要知道仅在 my_function.h 中定义的全局常量* global_value。如果不包
含头文件，源文件将无法编译。
 */

#include "my_function.h" // 或 #include "my_function.hpp"
int my_function() {
    return global_value; // 返回 42;
}
```

头文件随后被其他想要使用头文件接口定义的功能但不需要了解其实现细节的源文件包含（从而减少代码耦合）。以下程序使用了上述定义的头文件my_function.h：

```
// main.cpp

#include <iostream>      // 一个 C++ 标准库头文件。
#include "my_function.h" // 个人头文件

int main(int argc, char** argv) {
    std::cout << my_function() << std::endl;
```

Chapter 45: Header Files

Section 45.1: Basic Example

The following example will contain a block of code that is meant to be split into several source files, as denoted by // filename comments.

Source Files

```
// my_function.h

/* Note how this header contains only a declaration of a function.
 * Header functions usually do not define implementations for declarations
 * unless code must be further processed at compile time, as in templates.
 */

/* Also, usually header files include preprocessor guards so that every header
 * is never included twice.
 *
 * The guard is implemented by checking if a header-file unique preprocessor
 * token is defined, and only including the header if it hasn't been included
 * once before.
 */
#ifndef MY_FUNCTION_H
#define MY_FUNCTION_H

// global_value and my_function() will be
// recognized as the same constructs if this header is included by different files.
const int global_value = 42;
int my_function();

#endif // MY_FUNCTION_H
```

```
// my_function.cpp

/* Note how the corresponding source file for the header includes the interface
 * defined in the header so that the compiler is aware of what the source file is
 * implementing.
 *
 * In this case, the source file requires knowledge of the global constant
 * global_value only defined in my_function.h. Without inclusion of the header
 * file, this source file would not compile.
 */
#include "my_function.h" // or #include "my_function.hpp"
int my_function() {
    return global_value; // return 42;
}
```

Header files are then included by other source files that want to use the functionality defined by the header interface, but don't require knowledge of its implementation (thus, reducing code coupling). The following program makes use of the header `my_function.h` as defined above:

```
// main.cpp

#include <iostream>      // A C++ Standard Library header.
#include "my_function.h" // A personal header

int main(int argc, char** argv) {
    std::cout << my_function() << std::endl;
```

```
    return 0;  
}
```

编译过程

由于头文件通常是编译过程工作流的一部分，使用头文件/源文件约定的典型编译过程通常会执行以下操作。

假设头文件和源代码文件已经在同一目录下，程序员会执行以下命令：

```
g++ -c my_function.cpp      # 编译源文件 my_function.cpp  
                             # --> 生成目标文件 my_function.o  
  
g++ main.cpp my_function.o  # 链接包含  
                             # int my_function() 实现的目标文件  
                             # 与 main.cpp 编译后的目标文件链接  
                             # 然后生成最终可执行文件 a.out
```

或者，如果想先将 main.cpp 编译成目标文件，然后只将目标文件链接为最终步骤：

```
g++ -c my_function.cpp  
g++ -c main.cpp  
  
g++ main.o my_function.o
```

第45.2节：头文件中的模板

模板需要在编译时生成代码：例如，一个模板函数在被源代码中使用并参数化后，实际上会被转换成多个不同的函数。

这意味着模板函数、成员函数和类的定义不能委托给单独的源代码文件，因为任何使用模板构造的代码都需要知道其定义，才能生成相应的派生代码。

因此，如果将模板代码放在头文件中，必须同时包含其定义。下面是一个示例：

```
// templated_function.h  
  
template <typename T>  
T* null_T_pointer()  
T* type_point = NULL; // 或者，对于C++11及以后版本，可以用nullptr代替NULL  
  
    return type_point;  
}
```

```
    return 0;  
}
```

The Compilation Process

Since header files are often part of a compilation process workflow, a typical compilation process making use of the header/source file convention will usually do the following.

Assuming that the header file and source code file is already in the same directory, a programmer would execute the following commands:

```
g++ -c my_function.cpp      # Compiles the source file my_function.cpp  
                             # --> object file my_function.o  
  
g++ main.cpp my_function.o  # Links the object file containing the  
                             # implementation of int my_function()  
                             # to the compiled, object version of main.cpp  
                             # and then produces the final executable a.out
```

Alternatively, if one wishes to compile main.cpp to an object file first, and then link only object files together as the final step:

```
g++ -c my_function.cpp  
g++ -c main.cpp  
  
g++ main.o my_function.o
```

Section 45.2: Templates in Header Files

Templates require compile-time generation of code: a templated function, for example, will be effectively turned into multiple distinct functions once a templated function is parameterized by use in source code.

This means that template function, member function, and class definitions cannot be delegated to a separate source code file, as any code that will use any templated construct requires knowledge of its definition to generally generate any derivative code.

Thus, templated code, if put in headers, must also contain its definition. An example of this is below:

```
// templated_function.h  
  
template <typename T>  
T* null_T_pointer()  
T* type_point = NULL; // or, alternatively, nullptr instead of NULL  
                      // for C++11 or later  
    return type_point;  
}
```

第46章：using声明

using声明将之前在其他地方声明的单个名称引入当前作用域。

第46.1节：从命名空间单独导入名称

一旦使用using将命名空间std中的名称cout引入到main函数的作用域中，
std::cout对象就可以单独称为cout。

```
#include <iostream>
int main() {
    using std::cout;
    cout << "Hello, world!"}
```

第46.2节：重新声明基类成员以 避免名称隐藏

如果using-declaration出现在类作用域中，只允许重新声明基类的成员。例如，在类作用域中不允许使用using std::cout。

通常，重新声明的名称是本来会被隐藏的名称。例如，在下面的代码中，d1.foo仅指代Derived1::foo(const char*)，会导致编译错误。函数Base::foo(int)被隐藏，完全不被考虑。然而，d2.foo(42)是允许的，因为using-declaration将Base::foo(int)引入了Derived2中名为foo的实体集合。名称查找随后找到两个foo，重载解析选择了Base::foo。

```
结构体 Base {
    void foo(int);
};

struct Derived1 : Base {
    void foo(const char*);
};

struct Derived2 : Base {
    using Base::foo;
    void foo(const char*);
};

int main() {
    Derived1 d1;
    d1.foo(42); // 错误
    Derived2 d2;
    d2.foo(42); // 正确
}
```

第46.3节：继承构造函数

版本 ≥ C++11

作为特殊情况，类作用域内的using声明可以引用直接基类的构造函数。那些构造函数随后被继承到派生类，并可用于初始化派生类。

```
结构体 Base {
    Base(int x, const char* s);
};

struct Derived1 : Base {
    Derived1(int x, const char* s) : Base(x, s) {}
};
```

Chapter 46: Using declaration

A using declaration introduces a single name into the current scope that was previously declared elsewhere.

Section 46.1: Importing names individually from a namespace

Once using is used to introduce the name cout from the namespace std into the scope of the main function, the std::cout object can be referred to as cout alone.

```
#include <iostream>
int main() {
    using std::cout;
    cout << "Hello, world!\n";
}
```

Section 46.2: Redeclaring members from a base class to avoid name hiding

If a using-declaration occurs at class scope, it is only allowed to redeclare a member of a base class. For example, using std::cout is not allowed at class scope.

Often, the name redeclared is one that would otherwise be hidden. For example, in the below code, d1.foo only refers to Derived1::foo(const char*) and a compilation error will occur. The function Base::foo(int) is hidden not considered at all. However, d2.foo(42) is fine because the using-declaration brings Base::foo(int) into the set of entities named foo in Derived2. Name lookup then finds both foos and overload resolution selects Base::foo.

```
struct Base {
    void foo(int);
};

struct Derived1 : Base {
    void foo(const char*);
};

struct Derived2 : Base {
    using Base::foo;
    void foo(const char*);
};

int main() {
    Derived1 d1;
    d1.foo(42); // error
    Derived2 d2;
    d2.foo(42); // OK
}
```

Section 46.3: Inheriting constructors

Version ≥ C++11

As a special case, a using-declaration at class scope can refer to the constructors of a direct base class. Those constructors are then inherited by the derived class and can be used to initialize the derived class.

```
struct Base {
    Base(int x, const char* s);
};

struct Derived1 : Base {
    Derived1(int x, const char* s) : Base(x, s) {}
};
```

```
struct Derived2 : Base {  
    using Base::Base;  
};  
int main() {  
    Derived1 d1(42, "Hello, world");  
    Derived2 d2(42, "Hello, world");  
}
```

在上述代码中，Derived1 和 Derived2 都有构造函数，将参数直接转发给对应的 Base 构造函数。 Derived1 显式地执行转发，而 Derived2 利用 C++11 的继承构造函数特性，隐式地完成转发。

```
struct Derived2 : Base {  
    using Base::Base;  
};  
int main() {  
    Derived1 d1(42, "Hello, world");  
    Derived2 d2(42, "Hello, world");  
}
```

In the above code, both Derived1 and Derived2 have constructors that forward the arguments directly to the corresponding constructor of Base. Derived1 performs the forwarding explicitly, while Derived2, using the C++11 feature of inheriting constructors, does so implicitly.

第47章：std::string

字符串是表示字符序列的对象。标准的 `string` 类提供了一种简单、安全且多功能的替代方案，用于处理文本和其他字符序列时，避免直接使用显式的 `char` 数组。C++ 的 `string` 类属于 `std` 命名空间，并于1998年被标准化。

第47.1节：分词 (Tokenize)

按运行时开销从低到高列出：

1. `std::strtok` 是标准提供的开销最低的分词方法，它还允许在分词间修改分隔符，但它在现代 C++ 中存在三个问题：

- `std::strtok` 不能同时用于多个 `string` (尽管某些实现扩展支持此功能，如：`strtok_s`)
- 出于同样的原因，`std::strtok` 不能在多个线程中同时使用 (不过这可能依赖于实现，例如：Visual Studio 的实现是线程安全的) 调用 `std::strtok` 会修改其操作的 `std::string`，因此不能用于 `const string`、`const char*` 或字面量字符串。若要用 `std::strtok` 对这些进行分词，或对内容需保留的 `std::string` 进行操作，必须先复制输入，然后对副本进行操作。

通常这些选项的开销会隐藏在分词的分配成本中，但如果需要最廉价的算法且无法克服 `std::strtok` 的问题，可以考虑手写解决方案。

```
// 用于分词的字符串
std::string str{ "快速的棕色狐狸" };
// 用于存储分词的向量
vector<std::string> tokens;

for (auto i = strtok(&str[0], " "); i != NULL; i = strtok(NULL, " "))
    tokens.push_back(i);
```

实时示例

2. `std::istream_iterator` 使用流的提取运算符进行迭代。如果输入的 `std::string` 是以空白字符分隔，这能够扩展 `std::strtok` 的选项，消除其困难，实现内联分词，从而支持生成一个 `const vector<string>`，并且增加对多重分隔空白字符的支持：

```
// 要分词的字符串
const std::string str("The quick brown fox");std::istringstream
m is(str);
// 用于存储分词的向量
const std::vector<std::string> tokens = std::vector<std::string>(
    std::istream_iterator<std::string>(is),
    std::istream_iterator<std::string>());
```

实时示例

3. `std::regex_token_iterator` 使用 `std::regex` 进行迭代分词。它提供了更灵活的分隔符定义。例如，非分隔的逗号和空白字符：

Chapter 47: std::string

Strings are objects that represent sequences of characters. The standard `string` class provides a simple, safe and versatile alternative to using explicit arrays of `chars` when dealing with text and other sequences of characters. The C++ `string` class is part of the `std` namespace and was standardized in 1998.

Section 47.1: Tokenize

Listed from least expensive to most expensive at run-time:

1. `std::strtok` is the cheapest standard provided tokenization method, it also allows the delimiter to be modified between tokens, but it incurs 3 difficulties with modern C++:
 - `std::strtok` cannot be used on multiple strings at the same time (though some implementations do extend to support this, such as: `strtok_s`)
 - For the same reason `std::strtok` cannot be used on multiple threads simultaneously (this may however be implementation defined, for example: [Visual Studio's implementation is thread safe](#))
 - Calling `std::strtok` modifies the `std::string` it is operating on, so it cannot be used on `const strings`, `const char*s`, or literal strings, to tokenize any of these with `std::strtok` or to operate on a `std::string` whose contents need to be preserved, the input would have to be copied, then the copy could be operated on

Generally any of these options cost will be hidden in the allocation cost of the tokens, but if the cheapest algorithm is required and `std::strtok`'s difficulties are not overcomable consider a [hand-spun solution](#).

```
// String to tokenize
std::string str{ "The quick brown fox" };
// Vector to store tokens
vector<std::string> tokens;

for (auto i = strtok(&str[0], " "); i != NULL; i = strtok(NULL, " "))
    tokens.push_back(i);
```

Live Example

2. The `std::istream_iterator` uses the stream's extraction operator iteratively. If the input `std::string` is white-space delimited this is able to expand on the `std::strtok` option by eliminating its difficulties, allowing inline tokenization thereby supporting the generation of a `const vector<string>`, and by adding support for multiple delimiting white-space character:

```
// String to tokenize
const std::string str("The quick \tbrown \nfox");
std::istringstream is(str);
// Vector to store tokens
const std::vector<std::string> tokens = std::vector<std::string>(
    std::istream_iterator<std::string>(is),
    std::istream_iterator<std::string>());
```

Live Example

3. The `std::regex_token_iterator` uses a `std::regex` to iteratively tokenize. It provides for a more flexible delimiter definition. For example, non-delimited commas and white-space:

版本 ≥ C++11

```
// 要分词的字符串  
const std::string str{ "The ,qu\\,ick ,brown, fox" };  
const std::regex re{ "\\s*(?:[^\\\\,]|\\\\,)*?\\s*(?:,|$)" };  
// 用于存储标记的向量  
const std::vector<std::string> tokens{  
    std::sregex_token_iterator(str.begin(), str.end(), re, 1),  
    std::sregex_token_iterator()  
};
```

实时示例

更多细节请参见 regex_token_iterator 示例。

第47.2节：(const) char*的转换

为了获取 const char*类型的 std::string 数据访问，可以使用字符串的 c_str() 成员函数。

请记住，该指针仅在 std::string 对象处于作用域内且未被修改时有效，这意味着只能调用对象的 const 方法。

版本 ≥ C++17

data() 成员函数可用于获取可修改的 char*，进而操作 std::string 对象的数据。

版本 ≥ C++11

也可以通过取第一个字符的地址来获得可修改的 char* : &s[0]。在 C++11 中，这保证返回一个格式良好、以空字符结尾的字符串。注意，即使 s 为空，&s[0] 也是格式良好的，而 &s.front() 在 s 为空时是未定义的。

版本 ≥ C++11

```
std::string str("这是一个字符串。");  
const char* cstr = str.c_str(); // cstr 指向："这是一个字符串。\\0"  
const char* data = str.data(); // data 指向："这是一个字符串。\\0"  
  
std::string str("这是一个字符串。");  
  
// 将 str 的内容复制到 untie lifetime (解除生命周期绑定) 中，脱离 std::string 对象  
std::unique_ptr<char []> cstr = std::make_unique<char []>(str.size() + 1);  
  
// 上面代码的替代方案（无异常安全性）：  
// char* cstr_unsafe = new char[str.size() + 1];  
  
std::copy(str.data(), str.data() + str.size(), cstr);  
cstr[str.size()] = '\\0'; // 需要添加一个空终止符  
  
// delete[] cstr_unsafe;  
std::cout << cstr.get();
```

第47.3节：使用 std::string_view 类

版本 ≥ C++17

C++17 引入了 std::string_view，它只是一个非拥有的 const char 范围，可以实现为一对指针或一个指针加长度。它是需要不可修改字符串数据的函数的更优参数类型。在 C++17 之前，有三种选择：

```
void foo(std::string const& s); // C++17 之前，单个参数，可能会发生分配
```

Version ≥ C++11

```
// String to tokenize  
const std::string str{ "The ,qu\\,ick ,\\tbrown, fox" };  
const std::regex re{ "\\s*(?:[^\\\\,]|\\\\,)*?\\s*(?:,|$)" };  
// Vector to store tokens  
const std::vector<std::string> tokens{  
    std::sregex_token_iterator(str.begin(), str.end(), re, 1),  
    std::sregex_token_iterator()  
};
```

Live Example

See the regex_token_iterator Example for more details.

Section 47.2: Conversion to (const) char*

In order to get **const char*** access to the data of a **std::string** you can use the string's **c_str()** member function.

Keep in mind that the pointer is only valid as long as the **std::string** object is within scope and remains unchanged, that means that only **const** methods may be called on the object.

Version ≥ C++17

The **data()** member function can be used to obtain a modifiable **char***, which can be used to manipulate the **std::string** object's data.

Version ≥ C++11

A modifiable **char*** can also be obtained by taking the address of the first character: **&s[0]**. Within C++11, this is guaranteed to yield a well-formed, null-terminated string. Note that **&s[0]** is well-formed even if **s** is empty, whereas **&s.front()** is undefined if **s** is empty.

Version ≥ C++11

```
std::string str("This is a string.");  
const char* cstr = str.c_str(); // cstr points to: "This is a string.\\0"  
const char* data = str.data(); // data points to: "This is a string.\\0"  
  
std::string str("This is a string.");  
  
// Copy the contents of str to untie lifetime from the std::string object  
std::unique_ptr<char []> cstr = std::make_unique<char []>(str.size() + 1);  
  
// Alternative to the line above (no exception safety):  
// char* cstr_unsafe = new char[str.size() + 1];  
  
std::copy(str.data(), str.data() + str.size(), cstr);  
cstr[str.size()] = '\\0'; // A null-terminator needs to be added  
  
// delete[] cstr_unsafe;  
std::cout << cstr.get();
```

Section 47.3: Using the std::string_view class

Version ≥ C++17

C++17 introduces **std::string_view**, which is simply a non-owning range of **const chars**, implementable as either a pair of pointers or a pointer and a length. It is a superior parameter type for functions that require non-modifiable string data. Before C++17, there were three options for this:

```
void foo(std::string const& s); // pre-C++17, single argument, could incur
```

```

// 如果调用者的数据不是字符串
// (例如字符串字面量或 vector<char>)

void foo(const char* s, size_t len); // C++17 之前, 两个参数, 必须到处传递两者

void foo(const char* s);           // C++17 之前, 单个参数, 但需要调用
                                    // strlen()

template <class StringT>
void foo(StringT const& s);       // C++17 之前, 调用者可以传递任意字符数据
                                    // 提供者, 但现在 foo() 必须定义在头文件中

```

所有这些都可以替换为：

```

void foo(std::string_view s);      // C++17 之后, 单参数, 更紧密的耦合// 无论调用者如何存储数据, 都不
                                    // 会产生拷贝

```

注意 `std::string_view` 不能 修改其底层数据。

`string_view` 在你想避免不必要的拷贝时非常有用。

它提供了 `std::string` 的一个有用子集功能, 尽管其中一些函数的行为有所不同：

```

std::string str = "lllloooonnnngggg ssssttrriiinnnggg"; // 一个非常长的字符串// 不好的方式 - 'strin
g::substr' 返回一个新的字符串 (如果字符串很长, 开销很大) std::cout << str.substr(15, 10) << "';

// 好的方式 - 不会创建拷贝 !
std::string_view view = str;// strin

g_view::substr 返回一个新的 string_viewstd::cout << view.su
bstr(15, 10) << ";

```

第47.4节：转换为 `std::wstring`

在 C++ 中, 字符序列通过使用本地字符类型特化 `std::basic_string` 类来表示。标准库定义的两个主要集合是 `std::string` 和 `std::wstring` :

- `std::string` 是由类型为 `char` 的元素构成的
- `std::wstring` 是由类型为 `wchar_t` 的元素构成的

要在这两种类型之间转换, 使用 `wstring_convert` :

```

#include <string>
#include <codecvt>
#include <locale>

std::string input_str = "this is a -string-, which is a sequence based on the -char- type.";
std::wstring input_wstr = L"this is a -wide- string, which is based on the -wchar_t- type.";

// 转换
std::wstring str_turned_to_wstr =
std::wstring_convert<std::codecvt_utf8<wchar_t>>().from_bytes(input_str);

```

```

// allocation if caller's data was not in a string
// (e.g. string literal or vector<char> )

void foo(const char* s, size_t len); // pre-C++17, two arguments, have to pass them
                                    // both everywhere

void foo(const char* s);           // pre-C++17, single argument, but need to call
                                    // strlen()

template <class StringT>
void foo(StringT const& s);       // pre-C++17, caller can pass arbitrary char data
                                    // provider, but now foo() has to live in a header

```

All of these can be replaced with:

```

void foo(std::string_view s);      // post-C++17, single argument, tighter coupling
                                    // zero copies regardless of how caller is storing
                                    // the data

```

Note that `std::string_view` cannot modify its underlying data.

`string_view` 是有用的当你想避免不必要的拷贝。

It offers a useful subset of the functionality that `std::string` does, although some of the functions behave differently:

```

std::string str = "lllloooonnnngggg ssssttrriiinnnggg"; //A really long string

//Bad way - 'string::substr' returns a new string (expensive if the string is long)
std::cout << str.substr(15, 10) << '\n';

//Good way - No copies are created!
std::string_view view = str;

// string_view::substr returns a new string_view
std::cout << view.substr(15, 10) << '\n';

```

Section 47.4: Conversion to `std::wstring`

In C++, sequences of characters are represented by specializing the `std::basic_string` class with a native character type. The two major collections defined by the standard library are `std::string` and `std::wstring`:

- `std::string` is built with elements of type `char`
- `std::wstring` is built with elements of type `wchar_t`

To convert between the two types, use `wstring_convert`:

```

#include <string>
#include <codecvt>
#include <locale>

std::string input_str = "this is a -string-, which is a sequence based on the -char- type.";
std::wstring input_wstr = L"this is a -wide- string, which is based on the -wchar_t- type.";

// conversion
std::wstring str_turned_to_wstr =
std::wstring_convert<std::codecvt_utf8<wchar_t>>().from_bytes(input_str);

```

```
std::string wstr_turned_to_str =  
std::wstring_convert<std::codecvt_utf8<wchar_t>>().to_bytes(input_wstr);
```

为了提高可用性和/或可读性，你可以定义函数来执行转换：

```
#include <string>  
#include <codecvt>  
#include <locale>  
  
using convert_t = std::codecvt_utf8<wchar_t>;  
std::wstring_convert<convert_t, wchar_t> strconverter;  
  
std::string to_string(std::wstring wstr)  
{  
    return strconverter.to_bytes(wstr);  
}  
  
std::wstring to_wstring(std::string str)  
{  
    return strconverter.from_bytes(str);  
}
```

示例用法：

```
std::wstring a_wide_string = to_wstring("Hello World!");
```

这当然比 `std::wstring_convert<std::codecvt_utf8<wchar_t>>().from_bytes("Hello World!")` 更易读。

请注意，`char` 和 `wchar_t` 并不表示编码，也不指示字节大小。例如，`wchar_t` 通常实现为2字节数据类型，并且在 Windows 下通常包含UTF-16编码数据（或Windows 2000之前版本的UCS-2），而在Linux 下实现为4字节数据类型，使用UTF-32编码。这与较新的类型 `char16_t` 和 `char32_t` 不同，后者在C++11中引入，保证足够大以分别容纳任何UTF16或UTF32“字符”（更准确地说是 `code point`）。

第47.5节：字典序比较

两个 `std::string` 可以使用操作符 `==`、`!=`、`<`、`<=`、`>` 和 `>=` 进行字典序比较：

```
std::string str1 = "Foo";  
std::string str2 = "Bar";  
  
assert(!(str1 < str2));  
assert(str1 > str2);  
assert(!(str1 <= str2));  
assert(str1 >= str2);  
assert(!(str1 == str2));  
assert(str1 != str2);
```

所有这些函数都使用底层的`std::string::compare()`方法来执行比较，并为了方便返回布尔值。这些函数的操作可以按以下方式理解，无论实际实现如何：

- operator`==`:

如果`str1.length() == str2.length()`且每对字符都匹配，则返回`true`，否则返回

```
std::string wstr_turned_to_str =  
std::wstring_convert<std::codecvt_utf8<wchar_t>>().to_bytes(input_wstr);
```

In order to improve usability and/or readability, you can define functions to perform the conversion:

```
#include <string>  
#include <codecvt>  
#include <locale>  
  
using convert_t = std::codecvt_utf8<wchar_t>;  
std::wstring_convert<convert_t, wchar_t> strconverter;  
  
std::string to_string(std::wstring wstr)  
{  
    return strconverter.to_bytes(wstr);  
}  
  
std::wstring to_wstring(std::string str)  
{  
    return strconverter.from_bytes(str);  
}
```

Sample usage:

```
std::wstring a_wide_string = to_wstring("Hello World!");
```

That's certainly more readable than `std::wstring_convert<std::codecvt_utf8<wchar_t>>().from_bytes("Hello World!")`.

Please note that `char` and `wchar_t` do not imply encoding, and gives no indication of size in bytes. For instance, `wchar_t` is commonly implemented as a 2-bytes data type and typically contains UTF-16 encoded data under Windows (or UCS-2 in versions prior to Windows 2000) and as a 4-bytes data type encoded using UTF-32 under Linux. This is in contrast with the newer types `char16_t` and `char32_t`, which were introduced in C++11 and are guaranteed to be large enough to hold any UTF16 or UTF32 “character” (or more precisely, `code point`) respectively.

Section 47.5: Lexicographical comparison

Two `std::strings` can be compared lexicographically using the operators `==`, `!=`, `<`, `<=`, `>`, and `>=`:

```
std::string str1 = "Foo";  
std::string str2 = "Bar";  
  
assert(!(str1 < str2));  
assert(str1 > str2);  
assert(!(str1 <= str2));  
assert(str1 >= str2);  
assert(!(str1 == str2));  
assert(str1 != str2);
```

All these functions use the underlying `std::string::compare()` method to perform the comparison, and return for convenience boolean values. The operation of these functions may be interpreted as follows, regardless of the actual implementation:

- operator`==`:

If `str1.length() == str2.length()` and each character pair matches, then returns `true`, otherwise returns

false.

- operator!=:

如果 `str1.length() != str2.length()` 或有一对字符不匹配，则返回 `true`，否则返回 `false`。

- operator<或operator>:

找到第一个不同的字符对，比较它们然后返回布尔结果。

- operator<=或operator>=:

找到第一个不同的字符对，比较它们然后返回布尔结果。

注意：“字符对”一词指的是两个字符串中相同位置的对应字符。为了更好理解，假设两个示例字符串为 `str1` 和 `str2`，它们的长度分别为 `n` 和 `m`，则两个字符串的字符对指的是每个 `str1[i]` 和 `str2[i]` 的配对，其中 `i = 0, 1, 2, ..., max(n,m)`。如果对于任意 `i` 对应的字符不存在，即当 `i` 大于或等于 `n` 或 `m` 时，则视为最低值。

以下是使用 `<` 的示例：

```
std::string str1 = "Barr";
std::string str2 = "Bar";

assert(str2 < str1);
```

步骤如下：

1. 比较第一个字符，`'B' == 'B'` - 继续。
2. 比较第二个字符，`'a' == 'a'` - 继续。
3. 比较第三个字符，`'r' == 'r'` - 继续。
4. `str2` 范围现已耗尽，而 `str1` 范围仍有字符。因此，`str2 < str1`。

第47.6节：修剪起始/结束字符

此示例需要包含头文件 `<algorithm>`、`<locale>` 和 `<utility>`。

版本 \geq C++11

对序列或字符串进行 `trim` 意味着移除所有符合某个谓词的前导和尾随元素（或字符）。我们先修剪尾随元素，因为这不涉及移动任何元素，然后再修剪前导元素。注意，以下泛化适用于所有类型的 `std::basic_string`（例如 `std::string` 和 `std::wstring`），并且意外地也适用于序列容器（例如 `std::vector` 和 `std::list`）。

```
template <typename 序列, // 任何 basic_string、vector、list 等。
          typename Pred> // 元素（字符）类型的谓词
Sequence& trim(Sequence& seq, Pred pred) {
    return trim_start(trim_end(seq, pred), pred);
}
```

false.

- operator!=:

If `str1.length() != str2.length()` or one character pair doesn't match, returns `true`, otherwise it returns `false`.

- operator< or operator>:

Finds the first different character pair, compares them then returns the boolean result.

- operator<= or operator>=:

Finds the first different character pair, compares them then returns the boolean result.

Note: The term **character pair** means the corresponding characters in both strings of the same positions. For better understanding, if two example strings are `str1` and `str2`, and their lengths are `n` and `m` respectively, then character pairs of both strings means each `str1[i]` and `str2[i]` pairs where `i = 0, 1, 2, ..., max(n,m)`. If for any `i` where the corresponding character does not exist, that is, when `i` is greater than or equal to `n` or `m`, it would be considered as the lowest value.

Here is an example of using `<`:

```
std::string str1 = "Barr";
std::string str2 = "Bar";

assert(str2 < str1);
```

The steps are as follows:

1. Compare the first characters, `'B' == 'B'` - move on.
2. Compare the second characters, `'a' == 'a'` - move on.
3. Compare the third characters, `'r' == 'r'` - move on.
4. The `str2` range is now exhausted, while the `str1` range still has characters. Thus, `str2 < str1`.

Section 47.6: Trimming characters at start/end

This example requires the headers `<algorithm>`, `<locale>`, and `<utility>`.

Version \geq C++11

To `trim` a sequence or string means to remove all leading and trailing elements (or characters) matching a certain predicate. We first trim the trailing elements, because it doesn't involve moving any elements, and then trim the leading elements. Note that the generalizations below work for all types of `std::basic_string` (e.g. `std::string` and `std::wstring`)，and accidentally also for sequence containers (e.g. `std::vector` and `std::list`).

```
template <typename Sequence, // any basic_string, vector, list etc.
          typename Pred> // a predicate on the element (character) type
Sequence& trim(Sequence& seq, Pred pred) {
    return trim_start(trim_end(seq, pred), pred);
}
```

去除尾部元素涉及找到最后一个不匹配谓词的元素，并从那里开始删除：

```
template <typename Sequence, typename Pred>
Sequence& trim_end(Sequence& seq, Pred pred) {
    auto last = std::find_if_not(seq.rbegin(),
                                 seq.rend(),
                                 pred);
    seq.erase(last.base(), seq.end());
    return seq;
}
```

去除开头元素涉及找到第一个不匹配谓词的元素，并删除到该位置为止：

```
template <typename Sequence, typename Pred>
Sequence& trim_start(Sequence& seq, Pred pred) {
    auto first = std::find_if_not(seq.begin(),
                                  seq.end(),
                                  pred);
    seq.erase(seq.begin(), first);
    return seq;
}
```

为了专门处理std::string中去除空白字符，可以使用std::isspace()函数作为谓词：

```
std::string& trim(std::string& str, const std::locale& loc = std::locale()) {
    return trim(str, [&loc](const char c){ return std::isspace(c, loc); });
}

std::string& trim_start(std::string& str, const std::locale& loc = std::locale()) {
    return trim_start(str, [&loc](const char c){ return std::isspace(c, loc); });
}

std::string& trim_end(std::string& str, const std::locale& loc = std::locale()) {
    return trim_end(str, [&loc](const char c){ return std::isspace(c, loc); });
}
```

同样，我们可以对std::iswspace()函数用于std::wstring等类型。

如果你想创建一个new序列作为修剪后的副本，可以使用一个单独的函数：

```
template <typename Sequence, typename Pred>
Sequence trim_copy(Sequence seq, Pred pred) { // 注意：按值传递seq
    trim(seq, pred);
    return seq;
}
```

第47.7节：字符串替换

按位置替换

要替换std::string的一部分，可以使用std::string的replace方法。

replace有许多有用的重载：

Trimming the trailing elements involves finding the *last* element not matching the predicate, and erasing from there on:

```
template <typename Sequence, typename Pred>
Sequence& trim_end(Sequence& seq, Pred pred) {
    auto last = std::find_if_not(seq.rbegin(),
                                seq.rend(),
                                pred);
    seq.erase(last.base(), seq.end());
    return seq;
}
```

Trimming the leading elements involves finding the *first* element not matching the predicate and erasing up to there:

```
template <typename Sequence, typename Pred>
Sequence& trim_start(Sequence& seq, Pred pred) {
    auto first = std::find_if_not(seq.begin(),
                                 seq.end(),
                                 pred);
    seq.erase(seq.begin(), first);
    return seq;
}
```

To specialize the above for trimming whitespace in a std::string we can use the `std::isspace()` function as a predicate:

```
std::string& trim(std::string& str, const std::locale& loc = std::locale()) {
    return trim(str, [&loc](const char c){ return std::isspace(c, loc); });
}

std::string& trim_start(std::string& str, const std::locale& loc = std::locale()) {
    return trim_start(str, [&loc](const char c){ return std::isspace(c, loc); });
}

std::string& trim_end(std::string& str, const std::locale& loc = std::locale()) {
    return trim_end(str, [&loc](const char c){ return std::isspace(c, loc); });
}
```

Similarly, we can use the `std::iswspace()` function for std::wstring etc.

If you wish to create a new sequence that is a trimmed copy, then you can use a separate function:

```
template <typename Sequence, typename Pred>
Sequence trim_copy(Sequence seq, Pred pred) { // NOTE: passing seq by value
    trim(seq, pred);
    return seq;
}
```

Section 47.7: String replacement

Replace by position

To replace a portion of a std::string you can use the method `replace` from std::string.

replace has a lot of useful overloads:

```

//定义字符串
std::string str = "Hello foo, bar and world!";
std::string alternate = "Hello foobar";

//1)
str.replace(6, 3, "bar"); //"Hello bar, bar and world!"

//2)
str.replace(str.begin() + 6, str.end(), "nobody!"); //"Hello nobody!"

//3)
str.replace(19, 5, alternate, 6, 6); //"Hello foo, bar and foobar!"
版本 ≥ C++14

//4)
str.replace(19, 5, alternate, 6); //"Hello foo, bar and foobar!"

//5)
str.replace(str.begin(), str.begin() + 5, str.begin() + 6, str.begin() + 9);
//"foo foo, bar and world!"

//6)
str.replace(0, 5, 3, 'z'); //"zzz foo, bar and world!"

//7)
str.replace(str.begin() + 6, str.begin() + 9, 3, 'x'); //"Hello xxx, bar and world!"
版本 ≥ C++11

//8)
str.replace(str.begin(), str.begin() + 5, { 'x', 'y', 'z' }); //"xyz foo, bar and world!"

```

将字符串中的某个子串替换为另一个字符串

只替换str中第一次出现的replace为with：

```

std::string replaceString(std::string str,
                         const std::string& replace,
                         const std::string& with){
    std::size_t pos = str.find(replace);
    if (pos != std::string::npos)
        str.replace(pos, replace.length(), with);
    return str;
}

```

替换str中所有出现的replace为with：

```

std::string replaceStringAll(std::string str,
                            const std::string& replace,
                            const std::string& with) {
    if(!replace.empty()) {
        std::size_t pos = 0;
        while ((pos = str.find(replace, pos)) != std::string::npos) {
            str.replace(pos, replace.length(), with);
            pos += with.length();
        }
    }
    return str;
}

```

第47.8节：转换为std::string

std::ostringstream 可用于通过插入对象将任何可流式传输的类型转换为字符串表示

```

//Define string
std::string str = "Hello foo, bar and world!";
std::string alternate = "Hello foobar";

//1)
str.replace(6, 3, "bar"); //"Hello bar, bar and world!"

//2)
str.replace(str.begin() + 6, str.end(), "nobody!"); //"Hello nobody!"

//3)
str.replace(19, 5, alternate, 6, 6); //"Hello foo, bar and foobar!"
Version ≥ C++14

//4)
str.replace(19, 5, alternate, 6); //"Hello foo, bar and foobar!"

//5)
str.replace(str.begin(), str.begin() + 5, str.begin() + 6, str.begin() + 9);
//"foo foo, bar and world!"

//6)
str.replace(0, 5, 3, 'z'); //"zzz foo, bar and world!"

//7)
str.replace(str.begin() + 6, str.begin() + 9, 3, 'x'); //"Hello xxx, bar and world!"
Version ≥ C++11

//8)
str.replace(str.begin(), str.begin() + 5, { 'x', 'y', 'z' }); //"xyz foo, bar and world!"

```

Replace occurrences of a string with another string

Replace only the first occurrence of replace with with in str:

```

std::string replaceString(std::string str,
                         const std::string& replace,
                         const std::string& with){
    std::size_t pos = str.find(replace);
    if (pos != std::string::npos)
        str.replace(pos, replace.length(), with);
    return str;
}

```

Replace all occurrence of replace with with in str:

```

std::string replaceStringAll(std::string str,
                            const std::string& replace,
                            const std::string& with) {
    if(!replace.empty()) {
        std::size_t pos = 0;
        while ((pos = str.find(replace, pos)) != std::string::npos) {
            str.replace(pos, replace.length(), with);
            pos += with.length();
        }
    }
    return str;
}

```

Section 47.8: Converting to std::string

std::ostringstream can be used to convert any streamable type to a string representation, by inserting the object

使用流插入运算符<<将内容写入`std::ostringstream`对象，然后将整个`std::ostringstream`转换为`std::string`。

例如，对于int：

```
#include <sstream>

int main()
{
    int val = 4;
    std::ostringstream str;
    str << val;
    std::string converted = str.str();
    return 0;
}
```

编写您自己的转换函数，很简单：

```
template<class T>
std::string toString(const T& x)
{
    std::ostringstream ss;
    ss << x;
    return ss.str();
}
```

可行，但不适合性能关键的代码。

用户自定义类如果需要，可以实现流插入运算符：

```
std::ostream operator<<( std::ostream& out, const A& a )
{
    // 将 a 的字符串表示写入 out
    return out;
}
```

版本 ≥ C++11

除了流之外，自 C++11 起，你还可以使用 `std::to_string` (和 `std::to_wstring`) 函数，该函数对所有基本类型进行了重载，返回其参数的字符串表示。

```
std::string s = to_string(0x12f3); // 之后字符串 s 包含 "4851"
```

第47.9节：拆分

使用`std::string::substr`来分割字符串。该成员函数有两个变体。

第一个参数是起始位置，返回的子字符串应从该位置开始。起始位置必须在范围`(0, str.length()]`内有效：

```
std::string str = "Hello foo, bar and world!";
std::string newstr = str.substr(11); // "bar and world!"
```

第二个参数是起始位置和新子字符串的总长度。不管长度是多少，子字符串都不会超过源字符串的末尾：

```
std::string str = "Hello foo, bar and world!";
```

into a `std::ostringstream` object (with the stream insertion operator <<) and then converting the whole `std::ostringstream` to a `std::string`.

For `int` for instance:

```
#include <sstream>

int main()
{
    int val = 4;
    std::ostringstream str;
    str << val;
    std::string converted = str.str();
    return 0;
}
```

Writing your own conversion function, the simple:

```
template<class T>
std::string toString(const T& x)
{
    std::ostringstream ss;
    ss << x;
    return ss.str();
}
```

works but isn't suitable for performance critical code.

User-defined classes may implement the stream insertion operator if desired:

```
std::ostream operator<<( std::ostream& out, const A& a )
{
    // write a string representation of a to out
    return out;
}
```

Version ≥ C++11

Aside from streams, since C++11 you can also use the `std::to_string` (and `std::to_wstring`) function which is overloaded for all fundamental types and returns the string representation of its parameter.

```
std::string s = to_string(0x12f3); // after this the string s contains "4851"
```

Section 47.9: Splitting

Use `std::string::substr` to split a string. There are two variants of this member function.

The first takes a *starting position* from which the returned substring should begin. The starting position must be valid in the range `(0, str.length())`:

```
std::string str = "Hello foo, bar and world!";
std::string newstr = str.substr(11); // "bar and world!"
```

The second takes a starting position and a total *length* of the new substring. Regardless of the *length*, the substring will never go past the end of the source string:

```
std::string str = "Hello foo, bar and world!";
```

```
std::string newstr = str.substr(15, 3); // "and"
```

注意你也可以调用substr不带参数的版本，这种情况下会返回字符串的一个完全副本

```
std::string str = "Hello foo, bar and world!";
std::string newstr = str.substr(); // "Hello foo, bar and world!"
```

第47.10节：访问字符

有多种方法可以从std::string中提取字符，每种方法都有细微的不同。

```
std::string str("Hello world!");
```

operator[](n)

返回索引为 n 的字符的引用。

std::string::operator[] 不进行边界检查，也不会抛出异常。调用者负责断言索引在字符串范围内：

```
char c = str[6]; // 'w'
```

at(n)

返回索引为 n 的字符的引用。

std::string::at 会进行边界检查，如果索引不在字符串范围内，会抛出 std::out_of_range 异常：

```
char c = str.at(7); // 'o'
```

版本 ≥ C++11

注意：如果字符串为空，这两个示例都会导致未定义行为。

front()

返回第一个字符的引用：

```
char c = str.front(); // 'H'
```

back()

返回对最后一个字符的引用：

```
char c = str.back(); // '!'
```

第47.11节：检查一个字符串是否是另一个字符串的前缀

版本 ≥ C++14

在C++14中，这可以通过std::mismatch轻松完成，该函数返回两个范围中第一个不匹配的元素对：

```
std::string prefix = "foo";
```

```
std::string newstr = str.substr(15, 3); // "and"
```

Note that you can also call substr with no arguments, in this case an exact copy of the string is returned

```
std::string str = "Hello foo, bar and world!";
std::string newstr = str.substr(); // "Hello foo, bar and world!"
```

Section 47.10: Accessing a character

There are several ways to extract characters from a std::string and each is subtly different.

```
std::string str("Hello world!");
```

operator[](n)

Returns a reference to the character at index n.

std::string::operator[] is not bounds-checked and does not throw an exception. The caller is responsible for asserting that the index is within the range of the string:

```
char c = str[6]; // 'w'
```

at(n)

Returns a reference to the character at index n.

std::string::at is bounds checked, and will throw std::out_of_range if the index is not within the range of the string:

```
char c = str.at(7); // 'o'
```

Version ≥ C++11

Note: Both of these examples will result in undefined behavior if the string is empty.

front()

Returns a reference to the first character:

```
char c = str.front(); // 'H'
```

back()

Returns a reference to the last character:

```
char c = str.back(); // '!'
```

Section 47.11: Checking if a string is a prefix of another

Version ≥ C++14

In C++14, this is easily done by std::mismatch which returns the first mismatching pair from two ranges:

```
std::string prefix = "foo";
```

```
std::string string = "foobar";  
  
bool isPrefix = std::mismatch(prefix.begin(), prefix.end(),  
    string.begin(), string.end()).first == prefix.end();
```

注意，C++14之前存在一个范围加半范围的mismatch()版本，但当第二个字符串比第一个短时，这种用法是不安全的。

C++14之前的版本

我们仍然可以使用范围加半范围版本的std::mismatch()，但需要先检查第一个字符串的长度是否不大于第二个字符串：

```
bool isPrefix = prefix.size() <= string.size() &&  
    std::mismatch(prefix.begin(), prefix.end(),  
        string.begin(), string.end()).first == prefix.end();
```

版本 ≥ C++17

使用std::string_view，我们可以编写想要的直接比较，而无需担心分配开销或复制：

```
bool isPrefix(std::string_view prefix, std::string_view full)  
{  
    return prefix == full.substr(0, prefix.size());  
}
```

第47.12节：遍历每个字符

版本 ≥ C++11

std::string 支持迭代器，因此你可以使用基于范围的循环来遍历每个字符：

```
std::string str = "Hello World!";  
for (auto c : str)  
    std::cout << c;
```

你也可以使用“传统”的for循环来遍历每个字符：

```
std::string str = "Hello World!";  
for (std::size_t i = 0; i < str.length(); ++i)  
    std::cout << str[i];
```

第47.13节：转换为整数/浮点类型

一个包含数字的std::string可以使用转换函数转换为整数类型或浮点类型。

请注意，所有这些函数在遇到非数字字符时都会停止解析输入字符串，因此“123abc”将被转换为123。

std::ato*系列函数将C风格字符串（字符数组）转换为整数或浮点类型：

```
std::string ten = "10";  
  
double num1 = std::atof(ten.c_str());
```

```
std::string string = "foobar";
```

```
bool isPrefix = std::mismatch(prefix.begin(), prefix.end(),  
    string.begin(), string.end()).first == prefix.end();
```

Note that a range-and-a-half version of mismatch() existed prior to C++14, but this is unsafe in the case that the second string is the shorter of the two.

Version < C++14

We can still use the range-and-a-half version of std::mismatch(), but we need to first check that the first string is at most as big as the second:

```
bool isPrefix = prefix.size() <= string.size() &&  
    std::mismatch(prefix.begin(), prefix.end(),  
        string.begin(), string.end()).first == prefix.end();
```

Version ≥ C++17

With std::string_view, we can write the direct comparison we want without having to worry about allocation overhead or making copies:

```
bool isPrefix(std::string_view prefix, std::string_view full)  
{  
    return prefix == full.substr(0, prefix.size());  
}
```

Section 47.12: Looping through each character

Version ≥ C++11

std::string 支持迭代器，因此你可以使用一个基于范围的循环来遍历每个字符：

```
std::string str = "Hello World!";  
for (auto c : str)  
    std::cout << c;
```

你也可以使用一个“传统”的for循环来遍历每个字符：

```
std::string str = "Hello World!";  
for (std::size_t i = 0; i < str.length(); ++i)  
    std::cout << str[i];
```

Section 47.13: Conversion to integers/floating point types

A std::string 包含一个数字可以被转换为一个整数类型或一个浮点类型，使用转换函数。

Note that 所有这些函数在遇到非数字字符时都会停止解析输入字符串，因此“123abc”将被转换为123。

The std::ato* 家族函数将C风格字符串（字符数组）转换为整数或浮点类型：

```
std::string ten = "10";  
  
double num1 = std::atof(ten.c_str());
```

```
int num2 = std::atoi(ten.c_str());
long num3 = std::atol(ten.c_str());
版本 ≥ C++11
long long num4 = std::atoll(ten.c_str());
```

然而，不建议使用这些函数，因为如果解析字符串失败，它们会返回0。这很糟糕，因为0也可能是一个有效结果，例如输入字符串是“0”，因此无法判断转换是否真的失败。

更新的std::sto*系列函数将std::string转换为整数或浮点类型，如果无法解析输入，则会抛出异常。你应该尽可能使用这些函数：

```
版本 ≥ C++11
std::string ten = "10";

int num1 = std::stoi(ten);
long num2 = std::stol(ten);
long long num3 = std::stoll(ten);

float num4 = std::stof(ten);
double num5 = std::stod(ten);
long double num6 = std::stold(ten);
```

此外，这些函数还支持八进制和十六进制字符串，这一点与std::ato*系列不同。第二个参数是指向输入字符串中第一个未转换字符的指针（此处未示例），第三个参数是使用的进制。0表示自动检测八进制（以0开头）和十六进制（以0x或0X开头），其他值则表示使用的进制。

```
std::string ten = "10";
std::string ten_octal = "12";
std::string ten_hex = "0xA";

int num1 = std::stoi(ten, 0, 2); // 返回 2
int num2 = std::stoi(ten_octal, 0, 8); // 返回 10
long num3 = std::stol(ten_hex, 0, 16); // 返回 10
long num4 = std::stol(ten_hex); // 返回 0
long num5 = std::stol(ten_hex, 0, 0); // 返回 10, 因为它检测到了前导的 0x
```

第 47.14 节：连接

你可以使用重载的 + 和 += 运算符连接 std::string。使用 + 运算符：

```
std::string hello = "Hello";
std::string world = "world";
std::string helloworld = hello + world; // "Helloworld"
```

使用 += 运算符：

```
std::string hello = "Hello";
std::string world = "world";
hello += world; // "Helloworld"
```

你也可以追加 C 字符串，包括字符串字面量：

```
std::string hello = "Hello";
std::string world = "world";
```

```
int num2 = std::atoi(ten.c_str());
long num3 = std::atol(ten.c_str());
Version ≥ C++11
long long num4 = std::atoll(ten.c_str());
```

However, use of these functions is discouraged because they return 0 if they fail to parse the string. This is bad because 0 could also be a valid result, if for example the input string was "0", so it is impossible to determine if the conversion actually failed.

The newer std::sto* family of functions convert std::strings to integer or floating-point types, and throw exceptions if they could not parse their input. You should use these functions if possible:

```
Version ≥ C++11
std::string ten = "10";

int num1 = std::stoi(ten);
long num2 = std::stol(ten);
long long num3 = std::stoll(ten);

float num4 = std::stof(ten);
double num5 = std::stod(ten);
long double num6 = std::stold(ten);
```

Furthermore, these functions also handle octal and hex strings unlike the std::ato* family. The second parameter is a pointer to the first unconverted character in the input string (not illustrated here), and the third parameter is the base to use. 0 is automatic detection of octal (starting with 0) and hex (starting with 0x or 0X), and any other value is the base to use

```
std::string ten = "10";
std::string ten_octal = "12";
std::string ten_hex = "0xA";

int num1 = std::stoi(ten, 0, 2); // Returns 2
int num2 = std::stoi(ten_octal, 0, 8); // Returns 10
long num3 = std::stol(ten_hex, 0, 16); // Returns 10
long num4 = std::stol(ten_hex); // Returns 0
long num5 = std::stol(ten_hex, 0, 0); // Returns 10 as it detects the leading 0x
```

Section 47.14: Concatenation

You can concatenate std::strings using the overloaded + and += operators. Using the + operator:

```
std::string hello = "Hello";
std::string world = "world";
std::string helloworld = hello + world; // "Helloworld"
```

Using the += operator:

```
std::string hello = "Hello";
std::string world = "world";
hello += world; // "Helloworld"
```

You can also append C strings, including string literals:

```
std::string hello = "Hello";
std::string world = "world";
```

```
const char *comma = ", ";
std::string newhelloworld = hello + comma + world + "!"; // "Hello, world!"
```

你也可以使用 `push_back()` 来添加单个 `char` :

```
std::string s = "a, b, ";
s.push_back('c'); // "a, b, c"
```

还有 `append()`, 功能和 `+=` 差不多 :

```
std::string app = "test and ";
app.append("test"); // "test and test"
```

第47.15节：字符编码转换

使用C++11进行编码转换很简单，大多数编译器能够通过 `<codecvt>` 和 `<locale>` 头文件以跨平台的方式处理它。

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

int main() {
    // 在wstring和utf8字符串之间转换
    wstring_convert<codecvt_utf8_utf16<wchar_t>> wchar_to_utf8;
    // 在u16string和utf8字符串之间转换
    wstring_convert<codecvt_utf8_utf16<char16_t>, char16_t> utf16_to_utf8;

    wstring wstr = L"foobar";
    string utf8str = wchar_to_utf8.to_bytes(wstr);
    wstring wstr2 = wchar_to_utf8.from_bytes(utf8str);

    wcout << wstr << endl;
    cout << utf8str << endl;
    wcout << wstr2 << endl;

    u16string u16str = u"foobar";
    string utf8str2 = utf16_to_utf8.to_bytes(u16str);
    u16string u16str2 = utf16_to_utf8.from_bytes(utf8str2);

    return 0;
}
```

请注意，Visual Studio 2015 提供了对这些转换的支持，但其库实现中的一个错误

要求在处理`char16_t`时使用不同的`wstring_convert`模板：

```
using utf16_char = unsigned short;
wstring_convert<codecvt_utf8_utf16<utf16_char>, utf16_char> conv_utf8_utf16;

void strings::utf16_to_utf8(const std::u16string& utf16, std::string& utf8)
{
    std::basic_string<utf16_char> tmp;
    tmp.resize(utf16.length());
    std::copy(utf16.begin(), utf16.end(), tmp.begin());
    utf8 = conv_utf8_utf16.to_bytes(tmp);
}
```

```
const char *comma = ", ";
std::string newhelloworld = hello + comma + world + "!"; // "Hello, world!"
```

You can also use `push_back()` to push back individual `chars`:

```
std::string s = "a, b, ";
s.push_back('c'); // "a, b, c"
```

There is also `append()`, which is pretty much like `+=`:

```
std::string app = "test and ";
app.append("test"); // "test and test"
```

Section 47.15: Converting between character encodings

Converting between encodings is easy with C++11 and most compilers are able to deal with it in a cross-platform manner through `<codecvt>` and `<locale>` headers.

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

int main() {
    // converts between wstring and utf8 string
    wstring_convert<codecvt_utf8_utf16<wchar_t>> wchar_to_utf8;
    // converts between u16string and utf8 string
    wstring_convert<codecvt_utf8_utf16<char16_t>, char16_t> utf16_to_utf8;

    wstring wstr = L"foobar";
    string utf8str = wchar_to_utf8.to_bytes(wstr);
    wstring wstr2 = wchar_to_utf8.from_bytes(utf8str);

    wcout << wstr << endl;
    cout << utf8str << endl;
    wcout << wstr2 << endl;

    u16string u16str = u"foobar";
    string utf8str2 = utf16_to_utf8.to_bytes(u16str);
    u16string u16str2 = utf16_to_utf8.from_bytes(utf8str2);

    return 0;
}
```

Mind that Visual Studio 2015 provides supports for these conversion but a [bug](#) in their library implementation requires to use a different template for `wstring_convert` when dealing with `char16_t`:

```
using utf16_char = unsigned short;
wstring_convert<codecvt_utf8_utf16<utf16_char>, utf16_char> conv_utf8_utf16;

void strings::utf16_to_utf8(const std::u16string& utf16, std::string& utf8)
{
    std::basic_string<utf16_char> tmp;
    tmp.resize(utf16.length());
    std::copy(utf16.begin(), utf16.end(), tmp.begin());
    utf8 = conv_utf8_utf16.to_bytes(tmp);
}
```

```

void strings::utf8_to_utf16(const std::string& utf8, std::u16string& utf16)
{
    std::basic_string<utf16_char> tmp = conv_utf8_utf16.from_bytes(utf8);
    utf16.clear();
    utf16.resize(tmp.length());
    std::copy(tmp.begin(), tmp.end(), utf16.begin());
}

```

第47.16节：在字符串中查找字符

要查找一个字符或另一个字符串，可以使用`std::string::find`。它返回第一个匹配的第一个字符的位置。如果未找到匹配项，函数返回`std::string::npos`

```

std::string str = "Curiosity killed the cat";
auto it = str.find("cat");

if (it != std::string::npos)    std::cout
    << "Found at position: " << it << "\n";
else
    std::cout << "Not found!";

```

Found at position: 21

搜索功能通过以下函数得到了进一步扩展：

```

find_first_of // 查找字符的首次出现
find_first_not_of // 查找字符的首次未出现
find_last_of // 查找字符的最后一次出现
find_last_not_of // 查找字符的最后一次未出现

```

这些函数允许你从字符串末尾搜索字符，也可以查找不存在的情况（即字符串中没有的字符）。下面是一个示例：

```

std::string str = "dog dog cat cat"; std::cout
    << "Found at position: " << str.find_last_of("gzx") << "\n";

```

找到的位置：6

注意：请注意，上述函数不是搜索子字符串，而是搜索包含在搜索字符串中的字符。在此例中，最后一次出现的'g'位于位置6（其他字符未找到）。

```

void strings::utf8_to_utf16(const std::string& utf8, std::u16string& utf16)
{
    std::basic_string<utf16_char> tmp = conv_utf8_utf16.from_bytes(utf8);
    utf16.clear();
    utf16.resize(tmp.length());
    std::copy(tmp.begin(), tmp.end(), utf16.begin());
}

```

Section 47.16: Finding character(s) in a string

To find a character or another string, you can use `std::string::find`. It returns the position of the first character of the first match. If no matches were found, the function returns `std::string::npos`

```

std::string str = "Curiosity killed the cat";
auto it = str.find("cat");

if (it != std::string::npos)
    std::cout << "Found at position: " << it << "\n";
else
    std::cout << "Not found!\n";

```

Found at position: 21

The search opportunities are further expanded by the following functions:

```

find_first_of // Find first occurrence of characters
find_first_not_of // Find first absence of characters
find_last_of // Find last occurrence of characters
find_last_not_of // Find last absence of characters

```

These functions can allow you to search for characters from the end of the string, as well as find the negative case (ie. characters that are not in the string). Here is an example:

```

std::string str = "dog dog cat cat";
std::cout << "Found at position: " << str.find_last_of("gzx") << "\n";

```

Found at position: 6

Note: Be aware that the above functions do not search for substrings, but rather for characters contained in the search string. In this case, the last occurrence of 'g' was found at position 6 (the other characters weren't found).

第48章：std::array

参数	定义
class T	指定数组成员的数据类型
std::size_t N	指定数组中成员的数量

第48.1节：初始化std::array

初始化std::array<T, N>, 其中T是标量类型, N是类型T的元素数量

如果T是标量类型, std::array可以通过以下方式初始化：

```
// 1) 使用聚合初始化  
std::array<int, 3> a{ 0, 1, 2 };  
// 或者等价写法  
std::array<int, 3> a = { 0, 1, 2 };  
  
// 2) 使用拷贝构造函数  
std::array<int, 3> a{ 0, 1, 2 };  
std::array<int, 3> a2(a);  
// 或者等价写法  
std::array<int, 3> a2 = a;  
  
// 3) 使用移动构造函数  
std::array<int, 3> a = std::array<int, 3>{ 0, 1, 2 };
```

初始化std::array<T, N>, 其中T是非标量类型, N是类型T的元素数量

如果T是非标量类型, std::array可以通过以下方式初始化：

```
结构体A { int values[3]; }; // 一个聚合类型  
  
// 1) 使用带有花括号省略的聚合初始化  
// 仅当T是聚合类型时有效！  
std::array<A, 2> a{ 0, 1, 2, 3, 4, 5 };  
// 或者等价写法  
std::array<A, 2> a = { 0, 1, 2, 3, 4, 5 };  
  
// 2) 使用带有子元素花括号初始化的聚合初始化  
std::array<A, 2> a{ A{ 0, 1, 2 }, A{ 3, 4, 5 } };  
// 或者等价写法  
std::array<A, 2> a = { A{ 0, 1, 2 }, A{ 3, 4, 5 } };  
  
// 3)  
std::array<A, 2> a{ { 0, 1, 2 }, { 3, 4, 5 } };  
// 或者等价写法  
std::array<A, 2> a = { { 0, 1, 2 }, { 3, 4, 5 } };  
  
// 4) 使用拷贝构造函数  
std::array<A, 2> a{ 1, 2, 3 };  
std::array<A, 2> a2(a);  
// 或者等价写法  
std::array<A, 2> a2 = a;  
  
// 5) 使用移动构造函数  
std::array<A, 2> a = std::array<A, 2>{ 0, 1, 2, 3, 4, 5 };
```

Chapter 48: std::array

Parameter	Definition
class T	Specifies the data type of array members
std::size_t N	Specifies the number of members in the array

Section 48.1: Initializing an std::array

Initializing std::array<T, N>, where T is a scalar type and N is the number of elements of type T

If T is a scalar type, std::array can be initialized in the following ways:

```
// 1) Using aggregate-initialization  
std::array<int, 3> a{ 0, 1, 2 };  
// or equivalently  
std::array<int, 3> a = { 0, 1, 2 };  
  
// 2) Using the copy constructor  
std::array<int, 3> a{ 0, 1, 2 };  
std::array<int, 3> a2(a);  
// or equivalently  
std::array<int, 3> a2 = a;  
  
// 3) Using the move constructor  
std::array<int, 3> a = std::array<int, 3>{ 0, 1, 2 };
```

Initializing std::array<T, N>, where T is a non-scalar type and N is the number of elements of type T

If T is a non-scalar type std::array can be initialized in the following ways:

```
struct A { int values[3]; }; // An aggregate type  
  
// 1) Using aggregate initialization with brace elision  
// It works only if T is an aggregate type!  
std::array<A, 2> a{ 0, 1, 2, 3, 4, 5 };  
// or equivalently  
std::array<A, 2> a = { 0, 1, 2, 3, 4, 5 };  
  
// 2) Using aggregate initialization with brace initialization of sub-elements  
std::array<A, 2> a{ A{ 0, 1, 2 }, A{ 3, 4, 5 } };  
// or equivalently  
std::array<A, 2> a = { A{ 0, 1, 2 }, A{ 3, 4, 5 } };  
  
// 3)  
std::array<A, 2> a{ { 0, 1, 2 }, { 3, 4, 5 } };  
// or equivalently  
std::array<A, 2> a = { { 0, 1, 2 }, { 3, 4, 5 } };  
  
// 4) Using the copy constructor  
std::array<A, 2> a{ 1, 2, 3 };  
std::array<A, 2> a2(a);  
// or equivalently  
std::array<A, 2> a2 = a;  
  
// 5) Using the move constructor  
std::array<A, 2> a = std::array<A, 2>{ 0, 1, 2, 3, 4, 5 };
```

第48.2节：元素访问

1. at(pos)

返回位置 pos 处元素的引用，并进行边界检查。如果 pos 不在容器范围内，将抛出类型为 std::out_of_range 的异常。

复杂度为常数时间 O(1)。

```
#include <array>

int main()
{
    std::array<int, 3> arr;

    // 写入值
    arr.at(0) = 2;
    arr.at(1) = 4;
    arr.at(2) = 6;

    // 读取值
    int a = arr.at(0); // a 现在是 2
    int b = arr.at(1); // b 现在是 4
    int c = arr.at(2); // c 现在是 6

    return 0;
}
```

2) operator[pos]

返回对位置 pos 处元素的引用，不进行边界检查。如果 pos 不在容器范围内，可能会发生运行时错误。此方法提供的元素访问等同于经典数组，因此比 at(pos) 更高效。

复杂度为常数时间 O(1)。

```
#include <array>

int main()
{
    std::array<int, 3> arr;

    // 写入值
    arr[0] = 2;
    arr[1] = 4;
    arr[2] = 6;

    // 读取值
    int a = arr[0]; // a 现在是 2
    int b = arr[1]; // b 现在是 4
    int c = arr[2]; // c 现在是 6

    return 0;
}
```

3) std::get<pos>

这个非成员函数返回对编译时常量位置 pos 处元素的引用，且不进行

Section 48.2: Element access

1. at(pos)

Returns a reference to the element at position pos with bounds checking. If pos is not within the range of the container, an exception of type std::out_of_range is thrown.

The complexity is constant O(1).

```
#include <array>

int main()
{
    std::array<int, 3> arr;

    // write values
    arr.at(0) = 2;
    arr.at(1) = 4;
    arr.at(2) = 6;

    // read values
    int a = arr.at(0); // a is now 2
    int b = arr.at(1); // b is now 4
    int c = arr.at(2); // c is now 6

    return 0;
}
```

2) operator[pos]

Returns a reference to the element at position pos without bounds checking. If pos is not within the range of the container, a runtime *segmentation violation* error can occur. This method provides element access equivalent to classic arrays and thereof more efficient than at(pos).

The complexity is constant O(1).

```
#include <array>

int main()
{
    std::array<int, 3> arr;

    // write values
    arr[0] = 2;
    arr[1] = 4;
    arr[2] = 6;

    // read values
    int a = arr[0]; // a is now 2
    int b = arr[1]; // b is now 4
    int c = arr[2]; // c is now 6

    return 0;
}
```

3) std::get<pos>

This **non-member** function returns a reference to the element at **compile-time constant** position pos without

边界检查。如果pos不在容器的范围内，可能会发生运行时错误。

复杂度为常数时间 O(1)。

```
#include <array>

int main()
{
    std::array<int, 3> arr;

    // 写入值
    std::get<0>(arr) = 2;
    std::get<1>(arr) = 4;
    std::get<2>(arr) = 6;

    // 读取值
    int a = std::get<0>(arr); // a 现在是 2
    int b = std::get<1>(arr); // b 现在是 4
    int c = std::get<2>(arr); // c 现在是 6

    return 0;
}
```

4) front()

返回容器中第一个元素的引用。在空容器上调用front()是未定义的。

复杂度为常数时间 O(1)。

注意：对于容器 c，表达式 c.front() 等价于 *c.begin()。

```
#include <array>

int main()
{
    std::array<int, 3> arr{ 2, 4, 6 };

    int a = arr.front(); // a 现在是 2

    return 0;
}
```

5) back()

返回容器中最后一个元素的引用。在空容器上调用back()是未定义的。

复杂度为常数时间 O(1)。

```
#include <array>

int main()
{
    std::array<int, 3> arr{ 2, 4, 6 };

    int a = arr.back(); // a 现在是 6

    return 0;
}
```

bounds checking. If pos is not within the range of the container, a runtime *segmentation violation* error can occur.

The complexity is constant O(1).

```
#include <array>

int main()
{
    std::array<int, 3> arr;

    // write values
    std::get<0>(arr) = 2;
    std::get<1>(arr) = 4;
    std::get<2>(arr) = 6;

    // read values
    int a = std::get<0>(arr); // a is now 2
    int b = std::get<1>(arr); // b is now 4
    int c = std::get<2>(arr); // c is now 6

    return 0;
}
```

4) front()

Returns a reference to the first element in container. Calling front() on an empty container is undefined.

The complexity is constant O(1).

Note: For a container c, the expression c.front() is equivalent to *c.begin().

```
#include <array>

int main()
{
    std::array<int, 3> arr{ 2, 4, 6 };

    int a = arr.front(); // a is now 2

    return 0;
}
```

5) back()

Returns reference to the last element in the container. Calling back() on an empty container is undefined.

The complexity is constant O(1).

```
#include <array>

int main()
{
    std::array<int, 3> arr{ 2, 4, 6 };

    int a = arr.back(); // a is now 6

    return 0;
}
```

6) data()

返回指向作为元素存储的底层数组的指针。该指针保证范围 `[data();data() + size()]` 始终是一个有效范围，即使容器为空（此时 `data()` 不可解引用）。

复杂度为常数时间 $O(1)$ 。

```
#include <iostream>
#include <cstring>
#include <array>

int main ()
{
    const char* cstr = "Test string";
    std::array<char, 12> arr;

    std::memcpy(arr.data(), cstr, 12); // 将 cstr 复制到 arr

    std::cout << arr.data(); // 输出: Test string

    return 0;
}
```

第48.3节：遍历数组

`std::array` 作为 STL 容器，可以使用基于范围的 `for` 循环，类似于其他容器如 `vector`

```
int main() {
    std::array<int, 3> arr = { 1, 2, 3 };
    for (auto i : arr)
        cout << i << ";"
}
```

第48.4节：检查数组大小

与C风格数组相比，`std::array`的主要优点之一是我们可以使用`size()`成员函数检查数组的大小

```
int main() {
    std::array<int, 3> arr = { 1, 2, 3 };
    cout << arr.size() << endl;
}
```

第48.5节：一次性更改所有数组元素

可以对`std::array`使用成员函数`fill()`，在初始化后一次性更改所有元素的值

```
int main() {

    std::array<int, 3> arr = { 1, 2, 3 };
    // 将数组所有元素更改为100
    arr.fill(100);

}
```

6) data()

Returns pointer to the underlying array serving as element storage. The pointer is such that range `[data(); data() + size()]` is always a valid range, even if the container is empty (`data()` is not dereferenceable in that case).

The complexity is constant $O(1)$.

```
#include <iostream>
#include <cstring>
#include <array>

int main ()
{
    const char* cstr = "Test string";
    std::array<char, 12> arr;

    std::memcpy(arr.data(), cstr, 12); // copy cstr to arr

    std::cout << arr.data(); // outputs: Test string

    return 0;
}
```

Section 48.3: Iterating through the Array

`std::array` being a STL container, can use range-based for loop similar to other containers like `vector`

```
int main() {
    std::array<int, 3> arr = { 1, 2, 3 };
    for (auto i : arr)
        cout << i << '\n';
}
```

Section 48.4: Checking size of the Array

One of the main advantage of `std::array` as compared to C style array is that we can check the size of the array using `size()` member function

```
int main() {
    std::array<int, 3> arr = { 1, 2, 3 };
    cout << arr.size() << endl;
}
```

Section 48.5: Changing all array elements at once

The member function `fill()` can be used on `std::array` for changing the values at once post initialization

```
int main() {

    std::array<int, 3> arr = { 1, 2, 3 };
    // change all elements of the array to 100
    arr.fill(100);

}
```

第49章：std::vector

向量（vector）是一种动态数组，存储自动管理。向量中的元素访问效率与数组相当，且向量的优势在于其大小可以动态变化。

在存储方面，向量的数据通常放置在动态分配的内存中，因此需要一些额外开销；相反，C数组和std::array使用相对于声明位置的自动存储，因此没有任何开销。

第49.1节：访问元素

访问std::vector中元素的两种主要方式

- 基于索引的访问
- 迭代器

基于索引的访问：

这可以通过下标运算符[]，或成员函数at()来完成。

两者都返回对std::vector中相应位置元素的引用（除非是vector<bool>），因此可以读取也可以修改（如果vector不是const）。

[]和at()的区别在于[]不保证执行任何边界检查，而at()会。对于[]，访问index < 0或index >= size的元素是未定义行为，而at()会抛出std::out_of_range异常。

注意：下面的示例使用C++11风格的初始化以便清晰，但这些运算符可以用于所有版本（除非标注为C++11）。

版本 ≥ C++11

```
std::vector<int> v{ 1, 2, 3 };

// 使用 []
int a = v[1];    // a 是 2
v[1] = 4;        // v 现在包含 { 1, 4, 3 }

// 使用 at()
int b = v.at(2); // b 是 3
v.at(2) = 5;     // v 现在包含 { 1, 4, 5 }
int c = v.at(3); // 抛出 std::out_of_range 异常
```

由于at()方法执行边界检查并可能抛出异常，因此它比[]慢。这使得在操作语义保证索引在边界内的情况下，[]成为首选代码。无论如何，对向量元素的访问都是常数时间的。这意味着访问向量的第一个元素与访问第二个元素、第三个元素等的时间成本相同。

例如，考虑以下循环

```
for (std::size_t i = 0; i < v.size(); ++i) {
    v[i] = 1;
}
```

这里我们知道索引变量 i 总是在边界内，因此每次调用operator[]时检查 i 是否越界将浪费CPU周期。

Chapter 49: std::vector

A vector is a dynamic array with automatically handled storage. The elements in a vector can be accessed just as efficiently as those in an array with the advantage being that vectors can dynamically change in size.

In terms of storage the vector data is (usually) placed in dynamically allocated memory thus requiring some minor overhead; conversely C-arrays and std::array use automatic storage relative to the declared location and thus do not have any overhead.

Section 49.1: Accessing Elements

There are two primary ways of accessing elements in a [std::vector](#)

- index-based access
- iterators

Index-based access:

This can be done either with the subscript operator [] or the member function [at\(\)](#).

Both return a reference to the element at the respective position in the std::vector (unless it's a vector<bool>), so that it can be read as well as modified (if the vector is not const).

[] and at() differ in that [] is not guaranteed to perform any bounds checking, while at() does. Accessing elements where index < 0 or index >= size is undefined behavior for [] , while at() throws a [std::out_of_range](#) exception.

Note: The examples below use C++11-style initialization for clarity, but the operators can be used with all versions (unless marked C++11).

```
Version ≥ C++11
std::vector<int> v{ 1, 2, 3 };

// using []
int a = v[1];    // a is 2
v[1] = 4;        // v now contains { 1, 4, 3 }

// using at()
int b = v.at(2); // b is 3
v.at(2) = 5;     // v now contains { 1, 4, 5 }
int c = v.at(3); // throws std::out_of_range exception
```

Because the at() method performs bounds checking and can throw exceptions, it is slower than []. This makes [] preferred code where the semantics of the operation guarantee that the index is in bounds. In any case, accesses to elements of vectors are done in constant time. That means accessing to the first element of the vector has the same cost (in time) of accessing the second element, the third element and so on.

For example, consider this loop

```
for (std::size_t i = 0; i < v.size(); ++i) {
    v[i] = 1;
}
```

Here we know that the index variable i is always in bounds, so it would be a waste of CPU cycles to check that i is in bounds for every call to operator[].

`front()`和`back()`成员函数分别允许轻松访问向量的第一个和最后一个元素。这些位置经常被使用，且这些特殊访问器比使用`[]`的替代方案更具可读性：

```
std::vector<int> v{ 4, 5, 6 }; // 在C++11之前，这写法更冗长

int a = v.front(); // a是4, v.front()等同于v[0]
v.front() = 3; // 现在v包含{3, 5, 6}
int b = v.back(); // b是6, v.back()等同于v[v.size() - 1]
v.back() = 7; // 现在v包含{3, 5, 7}
```

注意：在空向量上调用`front()`或`back()`是未定义行为。你需要在调用`front()`或`back()`之前，使用`empty()`成员函数（用于检查容器是否为空）确认容器非空。下面是使用`'empty()'`检测空向量的简单示例：

```
int main ()
{
std::vector<int> v;
int sum (0);

for (int i=1;i<=10;i++) v.push_back(i); //创建并初始化向量

while (!v.empty()) //循环直到向量为空
{
    sum += v.back(); //保持一个运行总计
    v.pop_back(); //弹出元素，将其从向量中移除
}

std::cout << "total: " << sum << " "; //向用户输出总数

return 0;
}
```

上面的示例创建了一个包含从1到10的数字序列的向量。然后它不断弹出向量中的元素，直到向量为空（使用“`empty()`”）以防止未定义行为。接着计算并显示向量中数字的总和给用户。

版本 ≥ C++11

_____ `data()` 方法返回一个指向 `std::vector` 用于内部存储其元素的原始内存的指针。这通常用于将向量数据传递给期望C风格数组的遗留代码。

```
std::vector<int> v{ 1, 2, 3, 4 }; // v 包含 {1, 2, 3, 4}
int* p = v.data(); // p 指向 1
*p = 4; // v 现在包含 {4, 2, 3, 4}
++p; // p 指向 2
*p = 3; // v 现在包含 {4, 3, 3, 4}
p[1] = 2; // v 现在包含 {4, 3, 2, 4}
*(p + 2) = 1; // v 现在包含 {4, 3, 2, 1}
```

版本 < C++11

在 C++11 之前，`data()` 方法可以通过调用 `front()` 并取返回值的地址来模拟：

```
std::vector<int> v(4);
int* ptr = &(v.front()); // 或 &v[0]
```

这是可行的，因为向量总是保证将其元素存储在连续的内存位置，

The `front()` and `back()` member functions allow easy reference access to the first and last element of the vector, respectively. These positions are frequently used, and the special accessors can be more readable than their alternatives using `[]`:

```
std::vector<int> v{ 4, 5, 6 }; // In pre-C++11 this is more verbose

int a = v.front(); // a is 4, v.front() is equivalent to v[0]
v.front() = 3; // v now contains {3, 5, 6}
int b = v.back(); // b is 6, v.back() is equivalent to v[v.size() - 1]
v.back() = 7; // v now contains {3, 5, 7}
```

Note: It is undefined behavior to invoke `front()` or `back()` on an empty vector. You need to check that the container is not empty using the `empty()` member function (which checks if the container is empty) before calling `front()` or `back()`. A simple example of the use of 'empty()' to test for an empty vector follows:

```
int main ()
{
std::vector<int> v;
int sum (0);

for (int i=1;i<=10;i++) v.push_back(i); //create and initialize the vector

while (!v.empty()) //loop through until the vector tests to be empty
{
    sum += v.back(); //keep a running total
    v.pop_back(); //pop out the element which removes it from the vector
}

std::cout << "total: " << sum << '\n'; //output the total to the user

return 0;
}
```

The example above creates a vector with a sequence of numbers from 1 to 10. Then it pops the elements of the vector out until the vector is empty (using 'empty()') to prevent undefined behavior. Then the sum of the numbers in the vector is calculated and displayed to the user.

Version ≥ C++11

The `data()` method returns a pointer to the raw memory used by the `std::vector` to internally store its elements. This is most often used when passing the vector data to legacy code that expects a C-style array.

```
std::vector<int> v{ 1, 2, 3, 4 }; // v contains {1, 2, 3, 4}
int* p = v.data(); // p points to 1
*p = 4; // v now contains {4, 2, 3, 4}
++p; // p points to 2
*p = 3; // v now contains {4, 3, 3, 4}
p[1] = 2; // v now contains {4, 3, 2, 4}
*(p + 2) = 1; // v now contains {4, 3, 2, 1}
```

Version < C++11

Before C++11, the `data()` method can be simulated by calling `front()` and taking the address of the returned value:

```
std::vector<int> v(4);
int* ptr = &(v.front()); // or &v[0]
```

This works because vectors are always guaranteed to store their elements in contiguous memory locations,

假设向量的内容没有重载一元 `operator&`。如果重载了，你需要在 C++11 之前重新实现 `std::addressof`。它还假设向量不为空。

迭代器：

迭代器在示例“遍历 `std::vector`”和文章迭代器中有更详细的说明。简而言之，它们的作用类似于指向向量元素的指针：

```
版本 ≥ C++11
std::vector<int> v{ 4, 5, 6 };

auto it = v.begin();
int i = *it;           // i 是 4
++it;
i = *it;           // i 是 5
*it = 6;           // v 包含 { 4, 6, 6 }
auto e = v.end();    // e 指向 v 末尾元素之后的位置。它可以用来
                     // 检查迭代器是否到达了 vector 的末尾：
++it;
it == v.end();      // false, it 指向位置 2 的元素 (值为 6)
++it;
it == v.end();      // true
```

标准规定 `std::vector<T>` 的迭代器实际上是 `T*` 类型，但大多数标准库并不这样实现。不这样做可以改善错误信息，捕获非可移植代码，并且可以在非发布版本中为迭代器添加调试检查。然后，在发布版本中，包装底层指针的类会被优化掉。

你可以保存对 `vector` 元素的引用或指针以进行间接访问。除非你在该元素之前或该元素处添加/删除元素，或者导致 `vector` 容量发生变化，否则这些对 `vector` 元素的引用或指针保持稳定且访问定义有效。这与迭代器失效的规则相同。

```
版本 ≥ C++11
std::vector<int> v{ 1, 2, 3 };
int* p = v.data() + 1;    // p 指向元素 2
v.insert(v.begin(), 0);   // p 现在无效，访问 *p 是未定义行为。
p = v.data() + 1;        // p 指向元素 1
v.reserve(10);           // p 现在无效，访问 *p 是未定义行为。
p = v.data() + 1;        // p 指向元素 1
v.erase(v.begin());      // p 现在无效，访问 *p 是未定义行为。
```

第49.2节：初始化 `std::vector`

声明时可以通过多种方式初始化一个`std::vector`：

```
版本 ≥ C++11
std::vector<int> v{ 1, 2, 3 }; // v 变为 {1, 2, 3}

// 与 std::vector<int> v(3, 6) 不同
std::vector<int> v{ 3, 6 };    // v 变为 {3, 6}

// 与 C++11 中的 std::vector<int> v{3, 6} 不同
std::vector<int> v(3, 6);     // v 变为 {6, 6, 6}

std::vector<int> v(4);       // v 变为 {0, 0, 0, 0}
```

一个 `vector` 可以通过多种方式从另一个容器初始化：

assuming the contents of the vector doesn't override unary `operator&`. If it does, you'll have to re-implement `std::addressof` in pre-C++11. It also assumes that the vector isn't empty.

Iterators:

Iterators are explained in more detail in the example "Iterating over `std::vector`" and the article Iterators. In short, they act similarly to pointers to the elements of the vector:

```
Version ≥ C++11
std::vector<int> v{ 4, 5, 6 };

auto it = v.begin();
int i = *it;           // i is 4
++it;
i = *it;           // i is 5
*it = 6;           // v contains { 4, 6, 6 }
auto e = v.end();    // e points to the element after the end of v. It can be
                     // used to check whether an iterator reached the end of the vector:
++it;
it == v.end();      // false, it points to the element at position 2 (with value 6)
++it;
it == v.end();      // true
```

It is consistent with the standard that a `std::vector<T>`'s iterators actually *be* `T*`s, but most standard libraries do not do this. Not doing this both improves error messages, catches non-portable code, and can be used to instrument the iterators with debugging checks in non-release builds. Then, in release builds, the class wrapping around the underlying pointer is optimized away.

You can persist a reference or a pointer to an element of a vector for indirect access. These references or pointers to elements in the vector remain stable and access remains defined unless you add/remove elements at or before the element in the vector, or you cause the vector capacity to change. This is the same as the rule for invalidating iterators.

```
Version ≥ C++11
std::vector<int> v{ 1, 2, 3 };
int* p = v.data() + 1;    // p points to 2
v.insert(v.begin(), 0);   // p is now invalid, accessing *p is a undefined behavior.
p = v.data() + 1;        // p points to 1
v.reserve(10);           // p is now invalid, accessing *p is a undefined behavior.
p = v.data() + 1;        // p points to 1
v.erase(v.begin());      // p is now invalid, accessing *p is a undefined behavior.
```

Section 49.2: Initializing a `std::vector`

A `std::vector` can be initialized in several ways while declaring it:

```
Version ≥ C++11
std::vector<int> v{ 1, 2, 3 }; // v becomes {1, 2, 3}

// Different from std::vector<int> v(3, 6)
std::vector<int> v{ 3, 6 };    // v becomes {3, 6}

// Different from std::vector<int> v{3, 6} in C++11
std::vector<int> v(3, 6);     // v becomes {6, 6, 6}

std::vector<int> v(4);       // v becomes {0, 0, 0, 0}
```

A vector can be initialized from another container in several ways:

拷贝构造（仅限于另一个 vector），从 v2 复制数据：

```
std::vector<int> v(v2);
std::vector<int> v = v2;
```

移动构造（仅限于另一个 vector），从 v2 移动数据：

```
std::vector<int> v(std::move(v2));
std::vector<int> v = std::move(v2);
```

迭代器（范围）拷贝构造，将元素复制到 v 中：

```
// 从另一个向量
std::vector<int> v(v2.begin(), v2.begin() + 3); // v 变为 {v2[0], v2[1], v2[2]}
```

```
// 从数组
int z[] = { 1, 2, 3, 4 };
std::vector<int> v(z, z + 3); // v 变为 {1, 2, 3}
```

```
// 从列表
std::list<int> list1{ 1, 2, 3 };
std::vector<int> v(list1.begin(), list1.end()); // v 变为 {1, 2, 3}
```

版本 ≥ C++11

使用迭代器移动构造，利用 `std::make_move_iterator`，将元素移动到 v 中：

```
// 从另一个向量
std::vector<int> v(std::make_move_iterator(v2.begin()),
                   std::make_move_iterator(v2.end()));
```

```
// 从列表
std::list<int> list1{ 1, 2, 3 };
std::vector<int> v(std::make_move_iterator(list1.begin()),
                   std::make_move_iterator(list1.end()));
```

借助 `assign()` 成员函数，`std::vector` 可以在构造后重新初始化：

```
v.assign(4, 100); // v 变为 {100, 100, 100, 100}
```

```
v.assign(v2.begin(), v2.begin() + 3); // v 变为 {v2[0], v2[1], v2[2]}
```

```
int z[] = { 1, 2, 3, 4 };
v.assign(z + 1, z + 4); // v 变为 {2, 3, 4}
```

第49.3节：删除元素

删除最后一个元素：

```
std::vector<int> v{ 1, 2, 3 };
v.pop_back(); // v 变为 {1, 2}
```

删除所有元素：

```
std::vector<int> v{ 1, 2, 3 };
v.clear(); // v 变为空向量
```

按索引删除元素：

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(v.begin() + 3); // v 变为 {1, 2, 3, 5, 6}
```

Copy construction (from another vector only), which copies data from v2:

```
std::vector<int> v(v2);
std::vector<int> v = v2;
```

Version ≥ C++11

Move construction (from another vector only), which moves data from v2:

```
std::vector<int> v(std::move(v2));
std::vector<int> v = std::move(v2);
```

Iterator (range) copy-construction, which copies elements into v:

```
// from another vector
std::vector<int> v(v2.begin(), v2.begin() + 3); // v becomes {v2[0], v2[1], v2[2]}
```

```
// from an array
int z[] = { 1, 2, 3, 4 };
std::vector<int> v(z, z + 3); // v becomes {1, 2, 3}
```

```
// from a list
std::list<int> list1{ 1, 2, 3 };
std::vector<int> v(list1.begin(), list1.end()); // v becomes {1, 2, 3}
```

Version ≥ C++11

Iterator move-construction, using `std::make_move_iterator`, which moves elements into v:

```
// from another vector
std::vector<int> v(std::make_move_iterator(v2.begin()),
                   std::make_move_iterator(v2.end()));
```

```
// from a list
std::list<int> list1{ 1, 2, 3 };
std::vector<int> v(std::make_move_iterator(list1.begin()),
                   std::make_move_iterator(list1.end()));
```

With the help of the `assign()` member function, a `std::vector` can be reinitialized after its construction:

```
v.assign(4, 100); // v becomes {100, 100, 100, 100}
```

```
v.assign(v2.begin(), v2.begin() + 3); // v becomes {v2[0], v2[1], v2[2]}
```

```
int z[] = { 1, 2, 3, 4 };
v.assign(z + 1, z + 4); // v becomes {2, 3, 4}
```

Section 49.3: Deleting Elements

Deleting the last element:

```
std::vector<int> v{ 1, 2, 3 };
v.pop_back(); // v becomes {1, 2}
```

Deleting all elements:

```
std::vector<int> v{ 1, 2, 3 };
v.clear(); // v becomes an empty vector
```

Deleting element by index:

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(v.begin() + 3); // v becomes {1, 2, 3, 5, 6}
```

注意：对于 vector 删除非最后一个元素时，所有被删除元素之后的元素都必须被复制或移动以填补空缺，详见下方说明及 std::list。

删除一个范围内的所有元素：

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(v.begin() + 1, v.begin() + 5); // v 变为 {1, 6}
```

注意：上述方法不会改变 vector 的容量，只会改变大小。详见 Vector 大小和容量。

erase 方法用于移除一段元素，常作为 erase-remove 习惯用法的一部分。即，先由 std::remove 将部分元素移动到 vector 末尾，然后由 erase 将其删除。对于任何小于 vector 最后索引的下标，这是一种相对低效的操作，因为所有被删除区段之后的元素都必须重新定位。对于需要高效删除容器中任意元素的性能关键应用，请参见 std::list。

按值删除元素：

```
std::vector<int> v{ 1, 1, 2, 2, 3, 3 };
int value_to_remove = 2;
v.erase(std::remove(v.begin(), v.end(), value_to_remove), v.end()); // v 变为 {1, 1, 3, 3}
```

按条件删除元素：

```
// std::remove_if 需要一个函数，该函数以向量元素为参数，返回 true,
// 如果该元素应被删除
bool _predicate(const int& element) {
    return (element > 3); // 这将导致所有大于3的元素被删除
}

...
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(std::remove_if(v.begin(), v.end(), _predicate), v.end()); // v 变为 {1, 2, 3}
```

通过 lambda 删除元素，无需创建额外的谓词函数

版本 ≥ C++11

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(std::remove_if(v.begin(), v.end(),
    [] (auto& element){return element > 3;} ), v.end()
);
```

通过循环按条件删除元素：

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
std::vector<int>::iterator it = v.begin();
while (it != v.end()) {
    if (condition)
        it = v.erase(it); // 删除后，'it' 将被设置为 v 中的下一个元素
    else
        ++it; // 手动将 'it' 设置为 v 中的下一个元素
}
```

虽然在删除时不应该递增 `it` 很重要，但当在循环中重复擦除时，你应该考虑使用不同的方法。考虑使用 `remove_if` 以获得更高效的方式。

从反向循环中按条件删除元素：

```
std::vector<int> v{ -1, 0, 1, 2, 3, 4, 5, 6 };
typedef std::vector<int>::reverse_iterator rev_itr;
rev_itr it = v.rbegin();

while (it != v.rend()) { // 循环结束后 v 中只会剩下 '0'
    int value = *it;
    if (value) {
        ++it;
    }
}
```

Note: For a vector deleting an element which is not the last element, all elements beyond the deleted element have to be copied or moved to fill the gap, see the note below and std::list.

Deleting all elements in a range:

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(v.begin() + 1, v.begin() + 5); // v 变为 {1, 6}
```

Note: The above methods do not change the capacity of the vector, only the size. See Vector Size and Capacity.

The erase method, which removes a range of elements, is often used as a part of the erase-remove idiom. That is, first std::remove moves some elements to the end of the vector, and then erase chops them off. This is a relatively inefficient operation for any indices less than the last index of the vector because all elements after the erased segments must be relocated to new positions. For speed critical applications that require efficient removal of arbitrary elements in a container, see std::list.

Deleting elements by value:

```
std::vector<int> v{ 1, 1, 2, 2, 3, 3 };
int value_to_remove = 2;
v.erase(std::remove(v.begin(), v.end(), value_to_remove), v.end()); // v 变为 {1, 1, 3, 3}
```

Deleting elements by condition:

```
// std::remove_if 需要一个函数，该函数以向量元素为参数并返回 true,
// 如果该元素应被删除
bool _predicate(const int& element) {
    return (element > 3); // 这将导致所有大于3的元素被删除
}

...
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(std::remove_if(v.begin(), v.end(), _predicate), v.end()); // v 变为 {1, 2, 3}
```

Deleting elements by lambda, without creating additional predicate function

Version ≥ C++11

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(std::remove_if(v.begin(), v.end(),
    [] (auto& element){return element > 3;} ), v.end()
);
```

Deleting elements by condition from a loop:

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
std::vector<int>::iterator it = v.begin();
while (it != v.end()) {
    if (condition)
        it = v.erase(it); // 在擦除后，'it' 将被设置为 v 中的下一个元素
    else
        ++it; // 手动将 'it' 设置为 v 中的下一个元素
}
```

While it is important *not* to increment `it` in case of a deletion, you should consider using a different method when erasing repeatedly in a loop. Consider `remove_if` for a more efficient way.

Deleting elements by condition from a reverse loop:

```
std::vector<int> v{ -1, 0, 1, 2, 3, 4, 5, 6 };
typedef std::vector<int>::reverse_iterator rev_itr;
rev_itr it = v.rbegin();

while (it != v.rend()) { // 循环结束后 v 中只会剩下 '0'
    int value = *it;
    if (value) {
        ++it;
    }
}
```

```
// 以下代码行的解释见下文。
it = rev_itr(v.erase(it.base()));
} else
    ++it;
}
```

注意前面循环中的几点：

- 给定一个指向某元素的反向迭代器 `it`, 方法 `base` 返回指向同一元素的常规（非反向）迭代器。
- `vector::erase(iterator)` 删除由迭代器指向的元素，并返回指向被删除元素后继元素的迭代器。
- `reverse_iterator::reverse_iterator(iterator)` 从普通迭代器构造一个反向迭代器。

综合起来，语句 `it = rev_itr(v.erase(it.base()))` 的意思是：取反向迭代器 `it`, 让 `v` 删除其对应的普通迭代器指向的元素；然后用返回的迭代器构造一个反向迭代器，并赋值给反向迭代器 `it`。

```
// See explanation below for the following line.
it = rev_itr(v.erase(it.base()));
} else
    ++it;
}
```

Note some points for the preceding loop:

- Given a reverse iterator `it` pointing to some element, the method `base` gives the regular (non-reverse) iterator pointing to the same element.
- `vector::erase(iterator)` erases the element pointed to by an iterator, and returns an iterator to the element that followed the given element.
- `reverse_iterator::reverse_iterator(iterator)` constructs a reverse iterator from an iterator.

Put altogether, the line `it = rev_itr(v.erase(it.base()))` says: take the reverse iterator `it`, have `v` erase the element pointed by its regular iterator; take the resulting iterator, construct a reverse iterator from it, and assign it to the reverse iterator `it`.

使用 `v.clear()` 删除所有元素不会释放内存 (`vector` 的 `capacity()` 保持不变)。要回收空间，请使用：

```
std::vector<int>().swap(v);
```

版本 ≥ C++11

shrink_to_fit() 用于释放未使用的 `vector` 容量：

```
v.shrink_to_fit();
```

`shrink_to_fit` 并不保证一定能回收空间，但大多数当前实现都会回收。

第49.4节：遍历 std::vector

你可以通过多种方式遍历 `std::vector`。以下各节中，`v` 定义如下：

```
std::vector<int> v;
```

正向迭代

版本 ≥ C++11

```
// 基于范围的 for 循环
for(const auto& value: v) {
    std::cout << value << " ";
}
```

```
// 使用带迭代器的 for 循环
for(auto it = std::begin(v); it != std::end(v); ++it) {
    std::cout << *it << " ";
}
```

```
// 使用 for_each 算法，使用函数或函数对象：
void fun(int const& value) {
    std::cout << value << " ";
}
```

```
std::for_each(std::begin(v), std::end(v), fun);
```

Deleting all elements using `v.clear()` does not free up memory (`capacity()` of the vector remains unchanged). To reclaim space, use:

```
std::vector<int>().swap(v);
```

Version ≥ C++11

shrink_to_fit() frees up unused vector capacity:

```
v.shrink_to_fit();
```

The `shrink_to_fit` does not guarantee to really reclaim space, but most current implementations do.

Section 49.4: Iterating Over std::vector

You can iterate over a `std::vector` in several ways. For each of the following sections, `v` is defined as follows:

```
std::vector<int> v;
```

Iterating in the Forward Direction

Version ≥ C++11

```
// Range based for
for(const auto& value: v) {
    std::cout << value << "\n";
}
```

```
// Using a for loop with iterator
for(auto it = std::begin(v); it != std::end(v); ++it) {
    std::cout << *it << "\n";
}
```

```
// Using for_each algorithm, using a function or functor:
void fun(int const& value) {
    std::cout << value << "\n";
}
```

```
std::for_each(std::begin(v), std::end(v), fun);
```

```
// 使用 for_each 算法。使用 lambda 表达式：
std::for_each(std::begin(v), std::end(v), [](int const& value) {    std::cout << va
lue << "";
});

版本 < C++11
// 使用带迭代器的 for 循环
for(std::vector<int>::iterator it = std::begin(v); it != std::end(v); ++it) {    std::cout << *it << ""
;
}

// 使用带索引的 for 循环for(std::size_t
i = 0; i < v.size(); ++i) {    std::cout << v[i] << "";
}
```

反向迭代

版本 ≥ C++14

```
// 没有标准方法使用基于范围的 for 来实现此功能。
// 见下文替代方案。

// 使用 for_each 算法
// 注意：为清晰起见使用了 lambda，但函数或函数对象同样适用
std::for_each(std::rbegin(v), std::rend(v), [](auto const& value) {
std::cout << value << "");}

// 使用带迭代器的 for 循环for(auto rit =
std::rbegin(v); rit != std::rend(v); ++rit) {    std::cout << *rit << "";
}

// 使用带索引的 for 循环for(std::size_t
i = 0; i < v.size(); ++i) {    std::cout << v[v.size() -
1 - i] << "";}
}
```

虽然范围基的 for 循环没有内置的方式来反向迭代；但修复这个问题相对简单。范围基的 for 循环使用 begin() 和 end() 来获取迭代器，因此通过一个包装对象来模拟这一点可以实现我们所需的结果。

版本 ≥ C++14

```
template<class C>
struct ReverseRange {
C c; // 可以是引用或拷贝，如果原始对象是临时的
ReverseRange(C&& cin): c(std::forward<C>(cin)) {}
ReverseRange(ReverseRange&&)=default;
ReverseRange& operator=(ReverseRange&&)=delete;
    auto begin() const {return std::rbegin(c);}
    auto end() const {return std::rend(c);}
};

// C 旨在被推断，并完美转发到
template<class C>
ReverseRange<C> make_ReverseRange(C&& c) {return {std::forward<C>(c)};}

int main() {
std::vector<int> v { 1,2,3,4};for(auto const
    & value: make_ReverseRange(v)) {    std::cout << v
alue << "";}
}
```

强制常量元素

```
// Using for_each algorithm. Using a lambda:
std::for_each(std::begin(v), std::end(v), [](int const& value) {
    std::cout << value << "\n";
});

Version < C++11
// Using a for loop with iterator
for(std::vector<int>::iterator it = std::begin(v); it != std::end(v); ++it) {
    std::cout << *it << "\n";
}

// Using a for loop with index
for(std::size_t i = 0; i < v.size(); ++i) {
    std::cout << v[i] << "\n";
}

Iterating in the Reverse Direction
Version ≥ C++14
// There is no standard way to use range based for for this.
// See below for alternatives.

// Using for_each algorithm
// Note: Using a lambda for clarity. But a function or functor will work
std::for_each(std::rbegin(v), std::rend(v), [](auto const& value) {
    std::cout << value << "\n";
});

// Using a for loop with iterator
for(auto rit = std::rbegin(v); rit != std::rend(v); ++rit) {
    std::cout << *rit << "\n";
}

// Using a for loop with index
for(std::size_t i = 0; i < v.size(); ++i) {
    std::cout << v[v.size() - 1 - i] << "\n";
}
```

Though there is no built-in way to use the range based for to reverse iterate; it is relatively simple to fix this. The range based for uses begin() and end() to get iterators and thus simulating this with a wrapper object can achieve the results we require.

```
Version ≥ C++14
template<class C>
struct ReverseRange {
C c; // could be a reference or a copy, if the original was a temporary
ReverseRange(C&& cin): c(std::forward<C>(cin)) {}
ReverseRange(ReverseRange&&)=default;
ReverseRange& operator=(ReverseRange&&)=delete;
    auto begin() const {return std::rbegin(c);}
    auto end() const {return std::rend(c);}
};

// C is meant to be deduced, and perfect forwarded into
template<class C>
ReverseRange<C> make_ReverseRange(C&& c) {return {std::forward<C>(c)};}

int main() {
    std::vector<int> v { 1,2,3,4};
    for(auto const& value: make_ReverseRange(v)) {
        std::cout << value << "\n";
    }
}
```

Enforcing const elements

自 C++11 起，`cbegin()` 和 `cend()` 方法允许你为一个 `vector` 获取一个常量迭代器，即使该 `vector` 不是常量。常量迭代器允许你读取但不能修改 `vector` 的内容，这对于强制常量正确性非常有用：

```
版本 ≥ C++11
// 正向迭代
for (auto pos = v.cbegin(); pos != v.cend(); ++pos) {
    // pos 的类型是 vector<T>::const_iterator
    // *pos = 5; // 编译错误 - 不能通过 const 迭代器写入
}

// 反向迭代
for (auto pos = v.crbegin(); pos != v.crend(); ++pos) {
    // pos 的类型是 vector<T>::const_iterator
    // *pos = 5; // 编译错误 - 不能通过 const 迭代器写入
}

// 期望 Functor::operator()(T&)
for_each(v.begin(), v.end(), Functor());

// 期望 Functor::operator()(const T&)
for_each(v.cbegin(), v.cend(), Functor())
版本 ≥ C++17
```

as_const 将此扩展到范围迭代：

```
for (auto const& e : std::as_const(v)) { std::cout
    << e << " ";
}
```

这在早期版本的 C++ 中很容易实现：

```
版本 ≥ C++14
template <class T>
constexpr std::add_const_t<T>& as_const(T& t) noexcept {
    return t;
}
```

关于效率的说明

由于类 `std::vector` 基本上是管理动态分配的连续数组的类，这里解释的相同原理也适用于 C++ 向量。按照行优先顺序原则通过索引访问向量的内容效率更高。当然，每次访问向量时，其管理内容也会被加载到缓存中，但正如多次讨论过的（特别是在这里和这里），迭代 `std::vector` 与原始数组的性能差异可以忽略不计。因此，C 语言中原始数组的效率原则同样适用于 C++ 的 `std::vector`。

第49.5节：`vector<bool>`：众多规则中的例外

标准（第23.3.7节）规定提供了 `vector<bool>` 的特化版本，通过打包 `bool` 值来优化空间，使每个值仅占用一位。由于在 C++ 中位不可寻址，这意味着对 `vector` 的若干要求不适用于 `vector<bool>`：

- 存储的数据不要求是连续的，因此 `vector<bool>` 不能传递给期望 `bool` 数组的 C 接口。
- `at()`、`operator []` 和迭代器解引用不会返回 `bool` 的引用。相反，它们返回一个

Since C++11 the `cbegin()` and `cend()` methods allow you to obtain a *constant iterator* for a `vector`, even if the `vector` is non-const. A constant iterator allows you to read but not modify the contents of the `vector` which is useful to enforce const correctness:

```
Version ≥ C++11
// forward iteration
for (auto pos = v.cbegin(); pos != v.cend(); ++pos) {
    // type of pos is vector<T>::const_iterator
    // *pos = 5; // Compile error - can't write via const iterator
}

// reverse iteration
for (auto pos = v.crbegin(); pos != v.crend(); ++pos) {
    // type of pos is vector<T>::const_iterator
    // *pos = 5; // Compile error - can't write via const iterator
}

// expects Functor::operator()(T&)
for_each(v.begin(), v.end(), Functor());

// expects Functor::operator()(const T&)
for_each(v.cbegin(), v.cend(), Functor())
Version ≥ C++17
```

as_const 扩展了此范围迭代：

```
for (auto const& e : std::as_const(v)) {
    std::cout << e << '\n';
}
```

This is easy to implement in earlier versions of C++:

```
Version ≥ C++14
template <class T>
constexpr std::add_const_t<T>& as_const(T& t) noexcept {
    return t;
}
```

A Note on Efficiency

Since the class `std::vector` is basically a class that manages a dynamically allocated contiguous array, the same principle explained here applies to C++ vectors. Accessing the vector's content by index is much more efficient when following the row-major order principle. Of course, each access to the vector also puts its management content into the cache as well, but as has been debated many times (notably [here](#) and [here](#)), the difference in performance for iterating over a `std::vector` compared to a raw array is negligible. So the same principle of efficiency for raw arrays in C also applies for C++'s `std::vector`.

Section 49.5: `vector<bool>`: The Exception To So Many, So Many Rules

The standard (section 23.3.7) specifies that a specialization of `vector<bool>` is provided, which optimizes space by packing the `bool` values, so that each takes up only one bit. Since bits aren't addressable in C++, this means that several requirements on `vector` are not placed on `vector<bool>`:

- The data stored is not required to be contiguous, so a `vector<bool>` can't be passed to a C API which expects a `bool` array.
- `at()`, `operator []`, and dereferencing of iterators do not return a reference to `bool`. Rather they return a

代理对象，该对象通过重载赋值运算符（不完美地）模拟对bool的引用。例如，以下代码对std::vector<bool>可能无效，因为迭代器解引用不返回引用：

版本 ≥ C++11

```
std::vector<bool> v = {true, false};  
for (auto &b: v) {} // 错误
```

同样，期望bool&参数的函数不能用于operator []或at()作用于vector<bool>的结果，或用于解引用其迭代器的结果：

```
void f(bool& b);  
f(v[0]);           // 错误  
f(*v.begin());    // 错误
```

std::vector<bool>的实现依赖于编译器和架构。该特化通过将 n 个布尔值打包到最低可寻址内存单元中实现。这里，n 是最低可寻址内存的位数。在大多数现代系统中，这通常是1字节或8位。这意味着一个字节可以存储8个布尔值。这比传统实现中1个布尔值占用1字节的方式有了改进。

注意：以下示例展示了传统实现与优化实现中单个字节的可能按位值

vector<bool>。这在所有架构中并不总是成立。然而，这是可视化优化的一个好方法。在下面的示例中，一个字节表示为 [x, x, x, x, x, x, x, x]。

传统的 std::vector<char> 存储 8 个布尔值：

版本 ≥ C++11

```
std::vector<char> trad_vect = {true, false, false, false, true, false, true, true};
```

按位表示：

```
[0,0,0,0,0,0,0,1], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0],  
[0,0,0,0,0,0,0,1], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,1], [0,0,0,0,0,0,0,1]
```

专门化的 std::vector<bool> 存储 8 个布尔值：

版本 ≥ C++11

```
std::vector<bool> optimized_vect = {true, false, false, false, true, false, true, true};
```

按位表示：

```
[1,0,0,0,1,0,1,1]
```

注意在上述示例中，传统版本的 std::vector<bool> 中，8 个布尔值占用 8 字节内存，而在优化版本的 std::vector<bool> 中，它们只使用 1 字节内存。这是对内存使用的显著改进。如果需要将 vector<bool> 传递给 C 风格的 API，可能需要将值复制到数组，或者找到更好的使用该 API 的方法，以防内存和性能受到影响。

第 49.6 节：插入元素

在向量末尾添加元素（通过复制/移动）：

```
结构体 Point {  
    双精度 x, y;
```

proxy object that (imperfectly) simulates a reference to a `bool` by overloading its assignment operators. As an example, the following code may not be valid for `std::vector<bool>`, because dereferencing an iterator does not return a reference:

Version ≥ C++11

```
std::vector<bool> v = {true, false};  
for (auto &b: v) {} // error
```

Similarly, functions expecting a `bool&` argument cannot be used with the result of operator [] or at() applied to `vector<bool>`, or with the result of dereferencing its iterator:

```
void f(bool& b);  
f(v[0]);           // error  
f(*v.begin());    // error
```

The implementation of `std::vector<bool>` is dependent on both the compiler and architecture. The specialisation is implemented by packing n Booleans into the lowest addressable section of memory. Here, n is the size in bits of the lowest addressable memory. In most modern systems this is 1 byte or 8 bits. This means that one byte can store 8 Boolean values. This is an improvement over the traditional implementation where 1 Boolean value is stored in 1 byte of memory.

Note: The below example shows possible bitwise values of individual bytes in a traditional vs. optimized `vector<bool>`. This will not always hold true in all architectures. It is, however, a good way of visualising the optimization. In the below examples a byte is represented as [x, x, x, x, x, x, x, x].

Traditional std::vector<char> storing 8 Boolean values:

Version ≥ C++11

```
std::vector<char> trad_vect = {true, false, false, false, true, false, true, true};
```

Bitwise representation:

```
[0,0,0,0,0,0,0,1], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0],  
[0,0,0,0,0,0,0,1], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,1], [0,0,0,0,0,0,0,1]
```

Specialized std::vector<bool> storing 8 Boolean values:

Version ≥ C++11

```
std::vector<bool> optimized_vect = {true, false, false, false, true, false, true, true};
```

Bitwise representation:

```
[1,0,0,0,1,0,1,1]
```

Notice in the above example, that in the traditional version of `std::vector<bool>`, 8 Boolean values take up 8 bytes of memory, whereas in the optimized version of `std::vector<bool>`, they only use 1 byte of memory. This is a significant improvement on memory usage. If you need to pass a `vector<bool>` to an C-style API, you may need to copy the values to an array, or find a better way to use the API, if memory and performance are at risk.

Section 49.6: Inserting Elements

Appending an element at the end of a vector (by copying/moving):

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

```
Point(双精度 x, 双精度 y) : x(x), y(y) {}
};

std::vector<Point> v;
Point p(10.0, 2.0);
v.push_back(p); // p 被复制到向量中。
版本 ≥ C++11
```

通过在末尾原地构造元素来追加元素：

```
std::vector<Point> v;
v.emplace_back(10.0, 2.0); // 参数被传递给给定类型（这里是 Point）的构造函数。对象在向量中被构造
                           //，避免了复制。
```

注意，`std::vector` 出于性能原因不具有 `push_front()` 成员函数。在开头添加元素会导致向量中所有现有元素被移动。如果你需要频繁在容器开头插入元素，建议使用 `std::list` 或 `std::deque`。

在向量的任意位置插入元素：

```
std::vector<int> v{ 1, 2, 3 };
v.insert(v.begin(), 9); // 现在 v 包含 {9, 1, 2, 3}
```

版本 ≥ C++11

通过原地构造在向量的任意位置插入元素：

```
std::vector<int> v{ 1, 2, 3 };
v.emplace(v.begin() + 1, 9); // 现在 v 包含 {1, 9, 2, 3}
```

在向量的任意位置插入另一个向量：

```
std::vector<int> v(4); // 包含：0, 0, 0, 0
std::vector<int> v2(2, 10); // 包含：10, 10
v.insert(v.begin() + 2, v2.begin(), v2.end()); // 包含：0, 0, 10, 10, 0, 0
```

在向量的任意位置插入数组：

```
std::vector<int> v(4); // 包含：0, 0, 0, 0
int a[] = {1, 2, 3}; // 包含：1, 2, 3
v.insert(v.begin() + 1, a, a + sizeof(a) / sizeof(a[0])); // 包含：0, 1, 2, 3, 0, 0, 0
```

如果事先知道插入多个元素后向量的大小，使用`reserve()`以避免多次重新分配（参见向量大小和容量）：

```
std::vector<int> v;
v.reserve(100);
for(int i = 0; i < 100; ++i)
    v.emplace_back(i);
```

务必不要在这种情况下调用`resize()`，否则你会无意中创建一个包含200个元素的向量，而只有后面的一百个元素具有你预期的值。

第49.7节：将`std::vector`用作C数组

有几种方法可以将`std::vector`用作C数组（例如，为了与C库兼容）。这是可能的，因为向量中的元素是连续存储的。

```
Point(double x, double y) : x(x), y(y) {}
};

std::vector<Point> v;
Point p(10.0, 2.0);
v.push_back(p); // p is copied into the vector.
Version ≥ C++11
```

Appending an element at the end of a vector by constructing the element in place:

```
std::vector<Point> v;
v.emplace_back(10.0, 2.0); // The arguments are passed to the constructor of the
                           // given type (here Point). The object is constructed
                           // in the vector, avoiding a copy.
```

Note that `std::vector` does not have a `push_front()` member function due to performance reasons. Adding an element at the beginning causes all existing elements in the vector to be moved. If you want to frequently insert elements at the beginning of your container, then you might want to use `std::list` or `std::deque` instead.

Inserting an element at any position of a vector:

```
std::vector<int> v{ 1, 2, 3 };
v.insert(v.begin(), 9); // v now contains {9, 1, 2, 3}
Version ≥ C++11
```

Inserting an element at any position of a vector by constructing the element in place:

```
std::vector<int> v{ 1, 2, 3 };
v.emplace(v.begin() + 1, 9); // v now contains {1, 9, 2, 3}
```

Inserting another vector at any position of the vector:

```
std::vector<int> v(4); // contains: 0, 0, 0, 0
std::vector<int> v2(2, 10); // contains: 10, 10
v.insert(v.begin() + 2, v2.begin(), v2.end()); // contains: 0, 0, 10, 10, 0, 0
```

Inserting an array at any position of a vector:

```
std::vector<int> v(4); // contains: 0, 0, 0, 0
int a[] = {1, 2, 3}; // contains: 1, 2, 3
v.insert(v.begin() + 1, a, a + sizeof(a) / sizeof(a[0])); // contains: 0, 1, 2, 3, 0, 0, 0
```

Use `reserve()` before inserting multiple elements if resulting vector size is known beforehand to avoid multiple reallocations (see vector size and capacity):

```
std::vector<int> v;
v.reserve(100);
for(int i = 0; i < 100; ++i)
    v.emplace_back(i);
```

Be sure to not make the mistake of calling `resize()` in this case, or you will inadvertently create a vector with 200 elements where only the latter one hundred will have the value you intended.

Section 49.7: Using `std::vector` as a C array

There are several ways to use a `std::vector` as a C array (for example, for compatibility with C libraries). This is possible because the elements in a vector are stored contiguously.

版本 ≥ C++11

```
std::vector<int> v{ 1, 2, 3 };
int* p = v.data();
```

与基于之前C++标准的解决方案（见下文）相比，成员函数.data()也可以应用于空向量，因为在这种情况下它不会导致未定义行为。

在C++11之前，如果向量不为空，你会取向量第一个元素的地址来获得等效指针，这两种方法是可互换的：

```
int* p = &v[0];      // 结合下标运算符和0字面量
int* p = &v.front(); // 显式引用第一个元素
```

注意：如果向量为空，v[0]和v.front()是未定义的，不能使用。

在存储向量数据的基地址时，请注意许多操作（如push_back、resize等）可能会改变向量的数据内存位置，从而使之前的数据指针失效。例如：

```
std::vector<int> v;
int* p = v.data();
v.resize(42);      // 内存位置已更改；p的值现在无效
```

第49.8节：在std::vector中查找元素

定义在<algorithm>头文件中的函数std::find，可以用来在std::vector中查找元素。

std::find使用operator==来比较元素是否相等。它返回一个指向范围内第一个与值相等元素的迭代器。

如果未找到该元素，std::find返回std::vector::end（如果vector是常量，则返回std::vector::cend）。

版本 < C++11

```
static const int arr[] = {5, 4, 3, 2, 1};
std::vector<int> v (arr, arr + sizeof(arr) / sizeof(arr[0]) );

std::vector<int>::iterator it = std::find(v.begin(), v.end(), 4);
std::vector<int>::difference_type index = std::distance(v.begin(), it);
// `it` 指向vector的第二个元素，`index` 是1

std::vector<int>::iterator missing = std::find(v.begin(), v.end(), 10);
std::vector<int>::difference_type index_missing = std::distance(v.begin(), missing);
// `missing` 是 v.end()，`index_missing` 是5 (即vector的大小)
```

版本 ≥ C++11

```
std::vector<int> v { 5, 4, 3, 2, 1 };

auto it = std::find(v.begin(), v.end(), 4);
auto index = std::distance(v.begin(), it);
// `it` 指向vector的第二个元素，`index` 是1

auto missing = std::find(v.begin(), v.end(), 10);
auto index_missing = std::distance(v.begin(), missing);
// `missing` 是 v.end()，`index_missing` 是5 (即vector的大小)
```

如果需要在大型vector中执行多次搜索，那么你可能需要考虑先对vector进行排序，

Version ≥ C++11

```
std::vector<int> v{ 1, 2, 3 };
int* p = v.data();
```

In contrast to solutions based on previous C++ standards (see below), the member function .data() may also be applied to empty vectors, because it doesn't cause undefined behavior in this case.

Before C++11, you would take the address of the vector's first element to get an equivalent pointer, if the vector isn't empty, these both methods are interchangeable:

```
int* p = &v[0];      // combine subscript operator and 0 literal
int* p = &v.front(); // explicitly reference the first element
```

Note: If the vector is empty, v[0] and v.front() are undefined and cannot be used.

When storing the base address of the vector's data, note that many operations (such as push_back, resize, etc.) can change the data memory location of the vector, thus invalidating previous data pointers. For example:

```
std::vector<int> v;
int* p = v.data();
v.resize(42);      // internal memory location changed; value of p is now invalid
```

Section 49.8: Finding an Element in std::vector

The function `std::find`, defined in the `<algorithm>` header, can be used to find an element in a `std::vector`.

`std::find` uses the operator`==` to compare elements for equality. It returns an iterator to the first element in the range that compares equal to the value.

If the element in question is not found, `std::find` returns `std::vector::end` (or `std::vector::cend` if the vector is `const`).

Version < C++11

```
static const int arr[] = {5, 4, 3, 2, 1};
std::vector<int> v (arr, arr + sizeof(arr) / sizeof(arr[0]) );

std::vector<int>::iterator it = std::find(v.begin(), v.end(), 4);
std::vector<int>::difference_type index = std::distance(v.begin(), it);
// `it` points to the second element of the vector, `index` is 1
```

```
std::vector<int>::iterator missing = std::find(v.begin(), v.end(), 10);
std::vector<int>::difference_type index_missing = std::distance(v.begin(), missing);
// `missing` is v.end(), `index_missing` is 5 (ie. size of the vector)
```

Version ≥ C++11

```
std::vector<int> v { 5, 4, 3, 2, 1 };

auto it = std::find(v.begin(), v.end(), 4);
auto index = std::distance(v.begin(), it);
// `it` points to the second element of the vector, `index` is 1

auto missing = std::find(v.begin(), v.end(), 10);
auto index_missing = std::distance(v.begin(), missing);
// `missing` is v.end(), `index_missing` is 5 (ie. size of the vector)
```

If you need to perform many searches in a large vector, then you may want to consider sorting the vector first,

在使用binary_search算法之前。

要查找向量中第一个满足条件的元素，可以使用std::find_if。除了传递给std::find的两个参数外，std::find_if还接受第三个参数，该参数是一个函数对象或指向谓词函数的函数指针。谓词应接受容器中的一个元素作为参数，并返回一个可转换为bool的值，同时不修改容器：

```
版本 < C++11
bool isEven(int val) {
    return (val % 2 == 0);
}
```

```
struct moreThan {
moreThan(int limit) : _limit(limit) {}

    bool operator()(int val) {
        return val > _limit;
    }

    int _limit;
};
```

```
static const int arr[] = {1, 3, 7, 8};
std::vector<int> v (arr, arr + sizeof(arr) / sizeof(arr[0]));

std::vector<int>::iterator it = std::find_if(v.begin(), v.end(), isEven);
// `it` 指向 8，即第一个偶数元素
```

```
std::vector<int>::iterator missing = std::find_if(v.begin(), v.end(), moreThan(10));
// `missing` 是 v.end()，因为没有元素大于 10
```

版本 ≥ C++11

```
// 查找第一个偶数值
std::vector<int> v = {1, 3, 7, 8};
auto it = std::find_if(v.begin(), v.end(), [] (int val){return val % 2 == 0;});
// `it` 指向 8，第一个偶数元素
```

```
auto missing = std::find_if(v.begin(), v.end(), [] (int val){return val > 10;});
// `missing` 是 v.end()，因为没有元素大于 10
```

第 49.9 节：连接向量

可以使用成员函数 `insert()` 将一个 `std::vector` 追加到另一个向量后面：

```
std::vector<int> a = {0, 1, 2, 3, 4};
std::vector<int> b = {5, 6, 7, 8, 9};

a.insert(a.end(), b.begin(), b.end());
```

但是，如果尝试将一个向量追加到自身，这种方法会失败，因为标准规定传递给 `insert()` 的迭代器不得来自与接收对象元素相同的范围。

版本 ≥ C++11

除了使用向量的成员函数外，还可以使用函数 `std::begin()` 和 `std::end()`：

```
a.insert(std::end(a), std::begin(b), std::end(b));
```

这是一个更通用的解决方案，例如，因为 `b` 也可以是一个数组。不过，这个解决方案也不适用

before using the `binary_search` algorithm.

To find the first element in a vector that satisfies a condition, `std::find_if` can be used. In addition to the two parameters given to `std::find`, `std::find_if` accepts a third argument which is a function object or function pointer to a predicate function. The predicate should accept an element from the container as an argument and return a value convertible to `bool`, without modifying the container:

```
Version < C++11
bool isEven(int val) {
    return (val % 2 == 0);
}
```

```
struct moreThan {
moreThan(int limit) : _limit(limit) {}

    bool operator()(int val) {
        return val > _limit;
    }

    int _limit;
};
```

```
static const int arr[] = {1, 3, 7, 8};
std::vector<int> v (arr, arr + sizeof(arr) / sizeof(arr[0]));

std::vector<int>::iterator it = std::find_if(v.begin(), v.end(), isEven);
// `it` points to 8, the first even element
```

```
std::vector<int>::iterator missing = std::find_if(v.begin(), v.end(), moreThan(10));
// `missing` is v.end(), as no element is greater than 10
```

Version ≥ C++11

```
// find the first value that is even
std::vector<int> v = {1, 3, 7, 8};
auto it = std::find_if(v.begin(), v.end(), [] (int val){return val % 2 == 0;});
// `it` points to 8, the first even element
```

```
auto missing = std::find_if(v.begin(), v.end(), [] (int val){return val > 10;});
// `missing` is v.end(), as no element is greater than 10
```

Section 49.9: Concatenating Vectors

One `std::vector` can be appended to another by using the member function `insert()`:

```
std::vector<int> a = {0, 1, 2, 3, 4};
std::vector<int> b = {5, 6, 7, 8, 9};

a.insert(a.end(), b.begin(), b.end());
```

However, this solution fails if you try to append a vector to itself, because the standard specifies that iterators given to `insert()` must not be from the same range as the receiver object's elements.

Version ≥ C++11

Instead of using the vector's member functions, the functions `std::begin()` and `std::end()` can be used:

```
a.insert(std::end(a), std::begin(b), std::end(b));
```

This is a more general solution, for example, because `b` can also be an array. However, also this solution doesn't

允许你将一个向量追加到自身。

如果接收向量中元素的顺序无关紧要，考虑每个向量中的元素数量可以避免不必要的复制操作：

```
if (b.size() < a.size())
a.insert(a.end(), b.begin(), b.end());
else
b.insert(b.end(), a.begin(), a.end());
```

第49.10节：使用向量的矩阵

通过将向量定义为向量的向量，可以将向量用作二维矩阵。

一个具有3行4列且每个单元初始化为0的矩阵可以定义为：

```
std::vector<std::vector<int>> matrix(3, std::vector<int>(4));
```

版本 ≥ C++11

使用初始化列表或其他方式初始化它们的语法与普通向量类似。

```
std::vector<std::vector<int>> matrix = { {0,1,2,3},
                                            {4,5,6,7},
                                            {8,9,10,11}
                                         };
```

可以像访问二维数组一样访问该向量中的值

```
int var = matrix[0][2];
```

遍历整个矩阵类似于遍历普通向量，但多了一个维度。

```
for(int i = 0; i < 3; ++i)
{
    for(int j = 0; j < 4; ++j)
    {
        std::cout << matrix[i][j] << std::endl;
    }
}
版本 ≥ C++11
for(auto& row: matrix)
{
    for(auto& col : row)
    {
        std::cout << col << std::endl;
    }
}
```

向量的向量是一种表示矩阵的方便方式，但它不是最高效的：各个向量在内存中分散存放，且该数据结构不利于缓存。

此外，在一个合适的矩阵中，每一行的长度必须相同（这对于向量的向量来说并非如此）。这种额外的灵活性可能成为错误的来源。

allow you to append a vector to itself.

If the order of the elements in the receiving vector doesn't matter, considering the number of elements in each vector can avoid unnecessary copy operations:

```
if (b.size() < a.size())
a.insert(a.end(), b.begin(), b.end());
else
b.insert(b.end(), a.begin(), a.end());
```

Section 49.10: Matrices Using Vectors

Vectors can be used as a 2D matrix by defining them as a vector of vectors.

A matrix with 3 rows and 4 columns with each cell initialised as 0 can be defined as:

```
std::vector<std::vector<int>> matrix(3, std::vector<int>(4));
```

Version ≥ C++11

The syntax for initializing them using initialiser lists or otherwise are similar to that of a normal vector.

```
std::vector<std::vector<int>> matrix = { {0,1,2,3},
                                            {4,5,6,7},
                                            {8,9,10,11}
                                         };
```

Values in such a vector can be accessed similar to a 2D array

```
int var = matrix[0][2];
```

Iterating over the entire matrix is similar to that of a normal vector but with an extra dimension.

```
for(int i = 0; i < 3; ++i)
{
    for(int j = 0; j < 4; ++j)
    {
        std::cout << matrix[i][j] << std::endl;
    }
}
Version ≥ C++11
for(auto& row: matrix)
{
    for(auto& col : row)
    {
        std::cout << col << std::endl;
    }
}
```

A vector of vectors is a convenient way to represent a matrix but it's not the most efficient: individual vectors are scattered around memory and the data structure isn't cache friendly.

Also, in a proper matrix, the length of every row must be the same (this isn't the case for a vector of vectors). The additional flexibility can be a source of errors.

第49.11节：使用排序向量进行快速元素查找

<algorithm> 头文件提供了许多用于处理排序向量的有用函数。

使用排序向量的重要前提是存储的值可以用 < 进行比较。

可以使用函数 std::sort() 对未排序的向量进行排序：

```
std::vector<int> v;
// 在这里添加一些代码以填充 v 的元素
std::sort(v.begin(), v.end());
```

排序向量允许使用函数 std::lower_bound() 进行高效的元素查找。与 std::find() 不同，它对向量执行高效的二分查找。缺点是它只对已排序的输入范围给出有效结果：

```
// 在向量中查找第一个值为 42 的元素
std::vector<int>::iterator it = std::lower_bound(v.begin(), v.end(), 42);
if (it != v.end() && *it == 42) {
    // 找到了该元素！
}
```

注意：如果请求的值不在向量中，std::lower_bound() 将返回指向第一个大于请求值的元素的迭代器。此行为允许我们在已排序的向量中将新元素插入到正确的位置：

```
int const new_element = 33;
v.insert(std::lower_bound(v.begin(), v.end(), new_element), new_element);
```

如果你需要一次插入大量元素，先对所有元素调用push_back()，然后在所有元素插入完成后调用std::sort()，可能会更高效。在这种情况下，排序的额外开销可以抵消在向量末尾插入新元素而非中间插入所节省的成本。

如果你的向量包含多个相同值的元素，std::lower_bound() 会尝试返回指向所查找值的第一个元素的迭代器。然而，如果你需要在所查找值的最后一个元素之后插入新元素，应使用函数std::upper_bound()，因为这样会减少元素的移动：

```
v.insert(std::upper_bound(v.begin(), v.end(), new_element), new_element);
```

如果你同时需要上界和下界的迭代器，可以使用函数std::equal_range()，通过一次调用高效地获取它们：

```
std::pair<std::vector<int>::iterator,
std::vector<int>::iterator> rg = std::equal_range(v.begin(), v.end(), 42);
std::vector<int>::iterator lower_bound = rg.first;
std::vector<int>::iterator upper_bound = rg.second;
```

为了测试一个元素是否存在于已排序的向量中（虽然不限于向量），你可以使用函数std::binary_search()：

```
bool exists = std::binary_search(v.begin(), v.end(), value_to_find);
```

Section 49.11: Using a Sorted Vector for Fast Element Lookup

The [<algorithm>](#) header provides a number of useful functions for working with sorted vectors.

An important prerequisite for working with sorted vectors is that the stored values are comparable with <.

An unsorted vector can be sorted by using the function [std::sort\(\)](#):

```
std::vector<int> v;
// add some code here to fill v with some elements
std::sort(v.begin(), v.end());
```

Sorted vectors allow efficient element lookup using the function [std::lower_bound\(\)](#). Unlike [std::find\(\)](#), this performs an efficient binary search on the vector. The downside is that it only gives valid results for sorted input ranges:

```
// search the vector for the first element with value 42
std::vector<int>::iterator it = std::lower_bound(v.begin(), v.end(), 42);
if (it != v.end() && *it == 42) {
    // we found the element!
}
```

Note: If the requested value is not part of the vector, [std::lower_bound\(\)](#) will return an iterator to the first element that is greater than the requested value. This behavior allows us to insert a new element at its right place in an already sorted vector:

```
int const new_element = 33;
v.insert(std::lower_bound(v.begin(), v.end(), new_element), new_element);
```

If you need to insert a lot of elements at once, it might be more efficient to call push_back() for all them first and then call std::sort() once all elements have been inserted. In this case, the increased cost of the sorting can pay off against the reduced cost of inserting new elements at the end of the vector and not in the middle.

If your vector contains multiple elements of the same value, [std::lower_bound\(\)](#) will try to return an iterator to the first element of the searched value. However, if you need to insert a new element after the last element of the searched value, you should use the function [std::upper_bound\(\)](#) as this will cause less shifting around of elements:

```
v.insert(std::upper_bound(v.begin(), v.end(), new_element), new_element);
```

If you need both the upper bound and the lower bound iterators, you can use the function [std::equal_range\(\)](#) to retrieve both of them efficiently with one call:

```
std::pair<std::vector<int>::iterator,
std::vector<int>::iterator> rg = std::equal_range(v.begin(), v.end(), 42);
std::vector<int>::iterator lower_bound = rg.first;
std::vector<int>::iterator upper_bound = rg.second;
```

In order to test whether an element exists in a sorted vector (although not specific to vectors), you can use the function [std::binary_search\(\)](#):

```
bool exists = std::binary_search(v.begin(), v.end(), value_to_find);
```

第49.12节：减少向量的容量

`std::vector` 会在插入时根据需要自动增加其容量，但在删除元素后从不减少容量。

```
// 初始化一个包含100个元素的向量  
std::vector<int> v(100);  
  
// 向量的容量始终至少与其大小相等  
auto const old_capacity = v.capacity();  
// old_capacity >= 100  
  
// 删除一半的元素  
v.erase(v.begin() + 50, v.end()); // 将大小从100减少到50 (v.size() == 50) ,  
// 但容量不变 (v.capacity() == old_capacity)
```

要减少容量，可以将向量的内容复制到一个新的临时向量。新向量的容量将是存储原向量所有元素所需的最小容量。如果原向量的大小减少显著，那么新向量的容量减少也可能显著。然后我们可以交换原向量和临时向量，以保持其最小化的容量：

```
std::vector<int>(v).swap(v);
```

版本 ≥ C++11

在 C++11 中，我们可以使用 `shrink_to_fit()` 成员函数达到类似效果：

```
v.shrink_to_fit();
```

注意：`shrink_to_fit()` 成员函数只是一个请求，并不保证一定会减少容量。

第49.13节：向量的大小和容量

向量大小就是向量中元素的数量：

- 当前向量的大小可以通过`size()`成员函数查询。方便的`empty()`函数在大小为0时返回`true`。

```
vector<int> v = { 1, 2, 3 }; // 大小为3  
const vector<int>::size_type size = v.size();  
cout << size << endl; // 输出3  
cout << boolalpha << v.empty() << endl; // 输出false
```

- 默认构造的向量大小为0：

```
vector<int> v; // 大小为0  
cout << v.size() << endl; // 输出0
```

- 向量中添加N个元素会使大小增加N（例如通过`push_back()`、`insert()`或`resize()`函数）。

- 从向量中移除N个元素会使大小减少N（例如通过`pop_back()`、`erase()`或`clear()`函数）。

- Vector 对其大小有一个实现相关的上限，但在达到该上限之前，你很可能会耗尽内存：

```
vector<int> v;
```

Section 49.12: Reducing the Capacity of a Vector

A `std::vector` automatically increases its capacity upon insertion as needed, but it never reduces its capacity after element removal.

```
// Initialize a vector with 100 elements  
std::vector<int> v(100);  
  
// The vector's capacity is always at least as large as its size  
auto const old_capacity = v.capacity();  
// old_capacity >= 100  
  
// Remove half of the elements  
v.erase(v.begin() + 50, v.end()); // Reduces the size from 100 to 50 (v.size() == 50),  
// but not the capacity (v.capacity() == old_capacity)
```

To reduce its capacity, we can copy the contents of a vector to a new temporary vector. The new vector will have the minimum capacity that is needed to store all elements of the original vector. If the size reduction of the original vector was significant, then the capacity reduction for the new vector is likely to be significant. We can then swap the original vector with the temporary one to retain its minimized capacity:

```
std::vector<int>(v).swap(v);  
Version ≥ C++11
```

In C++11 we can use the `shrink_to_fit()` member function for a similar effect:

```
v.shrink_to_fit();
```

Note: The `shrink_to_fit()` member function is a request and doesn't guarantee to reduce capacity.

Section 49.13: Vector size and capacity

Vector size is simply the number of elements in the vector:

- Current vector **size** is queried by `size()` member function. Convenience `empty()` function returns `true` if size is 0:

```
vector<int> v = { 1, 2, 3 }; // size is 3  
const vector<int>::size_type size = v.size();  
cout << size << endl; // prints 3  
cout << boolalpha << v.empty() << endl; // prints false
```

- Default constructed vector starts with a size of 0:

```
vector<int> v; // size is 0  
cout << v.size() << endl; // prints 0
```

- Adding N elements to vector increases **size** by N (e.g. by `push_back()`, `insert()` or `resize()` functions).

- Removing N elements from vector decreases **size** by N (e.g. by `pop_back()`, `erase()` or `clear()` functions).

- Vector has an implementation-specific upper limit on its size, but you are likely to run out of RAM before reaching it:

```
vector<int> v;
```

```
const vector<int>::size_type max_size = v.max_size();
cout << max_size << endl; // 打印一个很大的数字
v.resize( max_size ); // 可能无法正常工作
v.push_back( 1 ); // 肯定无法正常工作
```

常见错误：size 不一定是（甚至通常不是）int 类型：

```
// !!!错误!!!恶意!!!
vector<int> v_bad( N, 1 ); // 构造大小为 N 的向量
for( int i = 0; i < v_bad.size(); ++i ) { // size 不应该是 int 类型!
    do_something( v_bad[i] );
}
```

向量容量 与 **size** 不同。虽然 **size** 只是向量当前拥有的元素数量，**capacity** 是为分配/预留的元素数量。这很有用，因为过于频繁地（重新）分配过大内存可能代价很高。

1. 当前向量容量通过capacity()成员函数查询。容量总是大于或等于大小：

```
vector<int> v = { 1, 2, 3 }; // 大小为3, 容量 >= 3
const vector<int>::size_type capacity = v.capacity();
cout << capacity << endl; // 输出大于等于3的数字
```

2. 你可以通过reserve(N)函数手动预留容量（它将向量容量更改为N）：

```
// !!!错误!!!恶意!!!
vector<int> v_bad;
for( int i = 0; i < 10000; ++i ) {
    v_bad.push_back( i ); // 可能会有大量重新分配
}

// 好的
vector<int> v_good;
v_good.reserve( 10000 ); // 好的！只分配一次
for( int i = 0; i < 10000; ++i ) {
    v_good.push_back( i ); // 不再需要分配内存
}
```

3. 您可以通过shrink_to_fit()请求释放多余的容量（但实现不一定必须遵守）。这对于节省已使用的内存很有用：这对于节省已使用的内存很有用：

```
vector<int> v = { 1, 2, 3, 4, 5 }; // 大小为5, 假设容量为6
v.shrink_to_fit(); // 容量变为5（或者可能仍为6） cout << boolalpha << v
.capacity() == v.size() << endl; // 可能打印true（但也可能是false）
```

Vector部分自动管理容量，当你添加元素时，它可能决定增长。实现者喜欢使用2或1.5作为增长因子（黄金比例是理想值——但由于是无理数，实际应用中不太可行）。另一方面，vector通常不会自动缩小。例如：

```
vector<int> v; // 容量可能是0（但不保证）
v.push_back( 1 ); // 容量为某个初始值，可能是1
v.clear(); // 大小为0，但容量仍然和之前一样！
```

```
const vector<int>::size_type max_size = v.max_size();
cout << max_size << endl; // prints some large number
v.resize( max_size ); // probably won't work
v.push_back( 1 ); // definitely won't work
```

Common mistake: **size** is not necessarily (or even usually) **int**:

```
// !!!bad!!!evil!!!
vector<int> v_bad( N, 1 ); // constructs large N size vector
for( int i = 0; i < v_bad.size(); ++i ) { // size is not supposed to be int!
    do_something( v_bad[i] );
}
```

Vector capacity differs from **size**. While **size** is simply how many elements the vector currently has, **capacity** is for how many elements it allocated/reserved memory for. That is useful, because too frequent (re)allocations of too large sizes can be expensive.

1. Current vector **capacity** is queried by capacity() member function. **Capacity** is always greater or equal to **size**:

```
vector<int> v = { 1, 2, 3 }; // size is 3, capacity is >= 3
const vector<int>::size_type capacity = v.capacity();
cout << capacity << endl; // prints number >= 3
```

2. You can manually reserve capacity by reserve(N) function (it changes vector capacity to N):

```
// !!!bad!!!evil!!!
vector<int> v_bad;
for( int i = 0; i < 10000; ++i ) {
    v_bad.push_back( i ); // possibly lot of reallocations
}

// good
vector<int> v_good;
v_good.reserve( 10000 ); // good! only one allocation
for( int i = 0; i < 10000; ++i ) {
    v_good.push_back( i ); // no allocations needed anymore
}
```

3. You can request for the excess capacity to be released by shrink_to_fit() (but the implementation doesn't have to obey you). This is useful to conserve used memory:

```
vector<int> v = { 1, 2, 3, 4, 5 }; // size is 5, assume capacity is 6
v.shrink_to_fit(); // capacity is 5 (or possibly still 6)
cout << boolalpha << v.capacity() == v.size() << endl; // prints likely true (but possibly false)
```

Vector partly manages capacity automatically, when you add elements it may decide to grow. Implementers like to use 2 or 1.5 for the grow factor (golden ratio would be the ideal value - but is impractical due to being rational number). On the other hand vector usually do not automatically shrink. For example:

```
vector<int> v; // capacity is possibly (but not guaranteed) to be 0
v.push_back( 1 ); // capacity is some starter value, likely 1
v.clear(); // size is 0 but capacity is still same as before!
```

```
v = { 1, 2, 3, 4 }; // 大小为4，假设容量为4。
v.push_back( 5 ); // 容量增长——假设增长到6 (1.5倍因子)
v.push_back( 6 ); // 容量未变化
v.push_back( 7 ); // 容量增长——假设增长到9 (1.5倍)
// 依此类推
v.pop_back(); v.pop_back(); v.pop_back(); v.pop_back(); // 容量保持不变
```

第49.14节：迭代器/指针失效

指向`std::vector`的迭代器和指针可能会失效，但仅在执行某些操作时会发生。

使用失效的迭代器/指针将导致未定义行为。

使迭代器/指针失效的操作包括：

- 任何改变vector容量的插入操作都会使所有迭代器/指针失效：

```
vector<int> v(5); // 向量大小为5；容量未知。
int *p1 = &v[0];
v.push_back(2); // 由于容量未知，p1可能已失效。

v.reserve(20); // 容量现在至少为20。
int *p2 = &v[0];
v.push_back(4); // p2 并未失效，因为 `v` 的大小现在是 7。
v.insert(v.end(), 30, 9); // 在末尾插入 30 个元素。大小超过了
// 请求的容量 20，因此 `p2` (很可能) 失效。
int *p3 = &v[0];
v.reserve(v.capacity() + 20); // 容量超出，因此 `p3` 失效。
```

版本 ≥ C++11

```
auto old_cap = v.capacity();
v.shrink_to_fit();
if(old_cap != v.capacity())
    // 迭代器已失效。
```

- 任何不增加容量的插入操作，仍会使指向插入位置及其之后元素的迭代器/指针失效。这包括 `end` 迭代器：

```
vector<int> v(5);
v.reserve(20); // 容量至少为 20。
int *p1 = &v[0];
int *p2 = &v[3];
v.insert(v.begin() + 2, 5, 0); // `p2` 已失效，但由于容量
// 未改变，`p1` 仍然有效。
int *p3 = &v[v.size() - 1];
v.push_back(10); // 容量未改变，因此 `p3` 和 `p1` 仍然有效。
```

- 任何删除操作都会使指向被删除元素及其之后元素的迭代器/指针失效。这包括 `end` 迭代器：

```
vector<int> v(10);
int *p1 = &v[0];
int *p2 = &v[5];
v.erase(v.begin() + 3, v.end()); // `p2` 已失效，但 `p1` 仍然有效。
```

- `operator=` (复制、移动或其他) 和 `clear()` 会使所有指向向量的迭代器/指针失效。

```
v = { 1, 2, 3, 4 }; // size is 4, and lets assume capacity is 4.
v.push_back( 5 ); // capacity grows - let's assume it grows to 6 (1.5 factor)
v.push_back( 6 ); // no change in capacity
v.push_back( 7 ); // capacity grows - let's assume it grows to 9 (1.5 factor)
// and so on
v.pop_back(); v.pop_back(); v.pop_back(); v.pop_back(); // capacity stays the same
```

Section 49.14: Iterator/Pointer Invalidation

Iterators and pointers pointing into an `std::vector` can become invalid, but only when performing certain operations. Using invalid iterators/pointers will result in undefined behavior.

Operations which invalidate iterators/pointers include:

- Any insertion operation which changes the capacity of the vector will invalidate *all* iterators/pointers:

```
vector<int> v(5); // Vector has a size of 5; capacity is unknown.
int *p1 = &v[0];
v.push_back(2); // p1 may have been invalidated, since the capacity was unknown.

v.reserve(20); // Capacity is now at least 20.
int *p2 = &v[0];
v.push_back(4); // p2 is *not* invalidated, since the size of `v` is now 7.
v.insert(v.end(), 30, 9); // Inserts 30 elements at the end. The size exceeds the
// requested capacity of 20, so `p2` is (probably) invalidated.
int *p3 = &v[0];
v.reserve(v.capacity() + 20); // Capacity exceeded, thus `p3` is invalid.
```

Version ≥ C++11

```
auto old_cap = v.capacity();
v.shrink_to_fit();
if(old_cap != v.capacity())
    // Iterators were invalidated.
```

- Any insertion operation, which does not increase the capacity, will still invalidate iterators/pointers pointing to elements at the insertion position and past it. This includes the `end` iterator:

```
vector<int> v(5);
v.reserve(20); // Capacity is at least 20.
int *p1 = &v[0];
int *p2 = &v[3];
v.insert(v.begin() + 2, 5, 0); // `p2` is invalidated, but since the capacity
// did not change, `p1` remains valid.
int *p3 = &v[v.size() - 1];
v.push_back(10); // The capacity did not change, so `p3` and `p1` remain valid.
```

- Any removal operation will invalidate iterators/pointers pointing to the removed elements and to any elements past the removed elements. This includes the `end` iterator:

```
vector<int> v(10);
int *p1 = &v[0];
int *p2 = &v[5];
v.erase(v.begin() + 3, v.end()); // `p2` is invalid, but `p1` remains valid.
```

- `operator=` (copy, move, or otherwise) and `clear()` will invalidate all iterators/pointers pointing into the vector.

第49.15节：在向量中查找最大和最小元素及其对应索引

要查找存储在向量中的最大或最小元素，可以分别使用方法 `std::max_element` 和 `std::min_element`。这些方法定义在 `<algorithm>` 头文件中。如果有多个元素等于最大（最小）元素，方法将返回指向第一个此类元素的迭代器。返回 `v.end()` 用于空向量。

```
std::vector<int> v = {5, 2, 8, 10, 9};
int maxElementIndex = std::max_element(v.begin(), v.end()) - v.begin();
int maxElement = *std::max_element(v.begin(), v.end());

int minElementIndex = std::min_element(v.begin(), v.end()) - v.begin();
int minElement = *std::min_element(v.begin(), v.end());

std::cout << "maxElementIndex:" << maxElementIndex << ", maxElement:" << maxElement << "; std::cout << "minElementIndex:" << minElementIndex << ", minElement:" << minElement << ";
```

输出：

```
maxElementIndex:3, maxElement:10
minElementIndex:1, minElement:2
```

版本 ≥ C++11

可以使用以下方法同时获取向量中的最小元素和最大元素

`std::minmax_element`，该方法也定义在 `<algorithm>` 头文件中：

```
std::vector<int> v = {5, 2, 8, 10, 9}; auto min
max = std::minmax_element(v.begin(), v.end()); std::cout << "minimum element: " << *minmax.first << "; std::cout << "maximum element: " << *minmax.second << ";
```

输出：

```
minimum element: 2
maximum element: 10
```

第49.16节：将数组转换为 `std::vector`

可以通过使用 `std::begin` 和 `std::end` 轻松地将数组转换为 `std::vector`：

```
int values[5] = {1, 2, 3, 4, 5}; // 源数组
std::vector<int> v(std::begin(values), std::end(values)); // 复制数组到新向量
for(auto &x: v)
    std::cout << x << " ";
    std::cout << std::endl;
```

Section 49.15: Find max and min Element and Respective Index in a Vector

To find the largest or smallest element stored in a vector, you can use the methods `std::max_element` and `std::min_element`, respectively. These methods are defined in `<algorithm>` header. If several elements are equivalent to the greatest (smallest) element, the methods return the iterator to the first such element. Return `v.end()` for empty vectors.

```
std::vector<int> v = {5, 2, 8, 10, 9};
int maxElementIndex = std::max_element(v.begin(), v.end()) - v.begin();
int maxElement = *std::max_element(v.begin(), v.end());

int minElementIndex = std::min_element(v.begin(), v.end()) - v.begin();
int minElement = *std::min_element(v.begin(), v.end());

std::cout << "maxElementIndex:" << maxElementIndex << ", maxElement:" << maxElement << '\n';
std::cout << "minElementIndex:" << minElementIndex << ", minElement:" << minElement << '\n';
```

Output:

```
maxElementIndex:3, maxElement:10
minElementIndex:1, minElement:2
```

Version ≥ C++11

The minimum and maximum element in a vector can be retrieved at the same time by using the method `std::minmax_element`, which is also defined in `<algorithm>` header:

```
std::vector<int> v = {5, 2, 8, 10, 9};
auto minmax = std::minmax_element(v.begin(), v.end());

std::cout << "minimum element: " << *minmax.first << '\n';
std::cout << "maximum element: " << *minmax.second << '\n';
```

Output:

```
minimum element: 2
maximum element: 10
```

Section 49.16: Converting an array to `std::vector`

An array can easily be converted into a `std::vector` by using `std::begin` and `std::end`:

```
int values[5] = {1, 2, 3, 4, 5}; // source array
std::vector<int> v(std::begin(values), std::end(values)); // copy array to new vector
for(auto &x: v)
    std::cout << x << " ";
    std::cout << std::endl;
```

```
int main(int argc, char* argv[]) {
    // 将主函数参数转换为字符串向量。
    std::vector<std::string> args(argv, argv + argc);
}
```

C++11 的 `initializer_list` 也可以用来一次性初始化向量

```
initializer_list<int> arr = { 1,2,3,4,5 };
vector<int> vec1 {arr};

for (auto & i : vec1)
    cout << i << endl;
```

第49.17节：返回大型向量的函数

版本 ≥ C++11

在 C++11 中，编译器被要求对被返回的局部变量进行隐式移动。此外，大多数编译器在许多情况下可以执行拷贝省略，完全省略移动操作。因此，返回可以廉价移动的大型对象不再需要特殊处理：

```
#include <vector>
#include <iostream>

// 如果编译器无法执行命名返回值优化 (NRVO)
// 并完全省略移动操作，则需要将 v 移动到返回值中。
std::vector<int> fillVector(int a, int b) {
    std::vector<int> v;
    v.reserve(b-a+1);
    for (int i = a; i <= b; i++) {
        v.push_back(i);
    }
    return v; // implicit move
}

int main() { // declare and fill vector
    std::vector<int> vec = fillVector(1, 10);

    // 打印向量
    for (auto value : vec)
        std::cout << value << " "; // 这将打印 "1 2 3 4 5 6 7 8 9 10"

    std::cout << std::endl;

    return 0;
}
```

版本 < C++11

在 C++11 之前，复制省略已经被允许并由大多数编译器实现。然而，由于缺乏移动语义，在遗留代码或必须使用不支持此优化的旧版本编译器编译的代码中，你会发现向量被作为输出参数传递以防止不必要的复制：

```
#include <vector>
#include <iostream>
```

```
int main(int argc, char* argv[]) {
    // convert main arguments into a vector of strings.
    std::vector<std::string> args(argv, argv + argc);
}
```

A C++11 `initializer_list` can also be used to initialize the vector at once

```
initializer_list<int> arr = { 1,2,3,4,5 };
vector<int> vec1 {arr};

for (auto & i : vec1)
    cout << i << endl;
```

Section 49.17: Functions Returning Large Vectors

Version ≥ C++11

In C++11, compilers are required to implicitly move from a local variable that is being returned. Moreover, most compilers can perform copy elision in many cases and elide the move altogether. As a result of this, returning large objects that can be moved cheaply no longer requires special handling:

```
#include <vector>
#include <iostream>

// If the compiler is unable to perform named return value optimization (NRVO)
// and elide the move altogether, it is required to move from v into the return value.
std::vector<int> fillVector(int a, int b) {
    std::vector<int> v;
    v.reserve(b-a+1);
    for (int i = a; i <= b; i++) {
        v.push_back(i);
    }
    return v; // implicit move
}

int main() { // declare and fill vector
    std::vector<int> vec = fillVector(1, 10);

    // print vector
    for (auto value : vec)
        std::cout << value << " "; // this will print "1 2 3 4 5 6 7 8 9 10"

    std::cout << std::endl;

    return 0;
}
```

Version < C++11

Before C++11, copy elision was already allowed and implemented by most compilers. However, due to the absence of move semantics, in legacy code or code that has to be compiled with older compiler versions which don't implement this optimization, you can find vectors being passed as output arguments to prevent the unneeded copy:

```
#include <vector>
#include <iostream>
```

```

// 通过引用传递 std::vector
void fillVectorFrom_By_Ref(int a, int b, std::vector<int> &v) {
    assert(v.empty());
    v.reserve(b-a+1);
    for (int i = a; i <= b; i++) {
        v.push_back(i);
    }
}

int main() {// 声明向量
    std::vector<int> vec;

    // 填充向量
    fillVectorFrom_By_Ref(1, 10, vec);
    // 打印向量
    for (std::vector<int>::const_iterator it = vec.begin(); it != vec.end(); ++it)
        std::cout << *it << " "; // 这将打印 "1 2 3 4 5 6 7 8 9 10 "
    std::cout << std::endl;
    return 0;
}

```

```

// passing a std::vector by reference
void fillVectorFrom_By_Ref(int a, int b, std::vector<int> &v) {
    assert(v.empty());
    v.reserve(b-a+1);
    for (int i = a; i <= b; i++) {
        v.push_back(i);
    }
}

int main() {// declare vector
    std::vector<int> vec;

    // fill vector
    fillVectorFrom_By_Ref(1, 10, vec);
    // print vector
    for (std::vector<int>::const_iterator it = vec.begin(); it != vec.end(); ++it)
        std::cout << *it << " "; // this will print "1 2 3 4 5 6 7 8 9 10 "
    std::cout << std::endl;
    return 0;
}

```

第50章：std::map

- 要使用任何 std::map 或 std::multimap，需包含头文件 `<map>`。
- `std::map` 和 `std::multimap` 都会根据键的升序对元素进行排序。对于 `std::multimap`，相同键的值之间不进行排序。
- `std::map` 和 `std::multimap` 的基本区别在于，`std::map` 不允许相同键有重复值，而 `std::multimap` 允许。
- 映射 (Map) 是以二叉搜索树实现的。因此 `search()`、`insert()`、`erase()` 平均时间复杂度为 $\Theta(\log n)$ 。若需常数时间操作，请使用 `std::unordered_map`。
- `size()` 和 `empty()` 函数的时间复杂度为 $\Theta(1)$ ，节点数被缓存，避免每次调用时遍历树。

第50.1节：访问元素

`std::map` 接受 (key, value) 键值对作为输入。

考虑以下 `std::map` 初始化的示例：

```
std::map < std::string, int > ranking { std::make_pair("stackoverflow", 2),  
                                         std::make_pair("docs-beta", 1) };
```

在 `std::map` 中，可以按如下方式插入元素：

```
ranking["stackoverflow"] = 2;  
ranking["docs-beta"] = 1;
```

在上述示例中，如果键 `stackoverflow` 已存在，其值将被更新为 2。如果该键尚不存在，则会创建一个新条目。

在 `std::map` 中，可以通过将键作为索引直接访问元素：

```
std::cout << ranking["stackoverflow"] << std::endl;
```

注意，使用 `operator[]` 访问 `map` 时，实际上会向 `map` 中插入一个带有查询键的新值。这意味着即使键已存储在 `map` 中，也不能在 `const std::map` 上使用它。为防止这种插入，可以先检查元素是否存在（例如使用 `find()`），或者使用下面介绍的 `at()`。

版本 ≥ C++11

可以使用 `at()` 访问 `std::map` 的元素：

```
std::cout << ranking.at("stackoverflow") << std::endl;
```

请注意，如果容器不包含请求的元素，`at()` 将抛出 `std::out_of_range` 异常。

在 `std::map` 和 `std::multimap` 两种容器中，都可以使用迭代器访问元素：

版本 ≥ C++11

```
// 使用 begin() 的示例
```

Chapter 50: std::map

- To use any of `std::map` or `std::multimap` the header file `<map>` should be included.
- `std::map` and `std::multimap` both keep their elements sorted according to the ascending order of keys. In case of `std::multimap`, no sorting occurs for the values of the same key.
- The basic difference between `std::map` and `std::multimap` is that the `std::map` one does not allow duplicate values for the same key where `std::multimap` does.
- Maps are implemented as binary search trees. So `search()`, `insert()`, `erase()` takes $\Theta(\log n)$ time in average. For constant time operation use `std::unordered_map`.
- `size()` and `empty()` functions have $\Theta(1)$ time complexity, number of nodes is cached to avoid walking through tree each time these functions are called.

Section 50.1: Accessing elements

An `std::map` takes (key, value) pairs as input.

Consider the following example of `std::map` initialization:

```
std::map < std::string, int > ranking { std::make_pair("stackoverflow", 2),  
                                         std::make_pair("docs-beta", 1) };
```

In an `std::map` , elements can be inserted as follows:

```
ranking["stackoverflow"] = 2;  
ranking["docs-beta"] = 1;
```

In the above example, if the key `stackoverflow` is already present, its value will be updated to 2. If it isn't already present, a new entry will be created.

In an `std::map` , elements can be accessed directly by giving the key as an index:

```
std::cout << ranking["stackoverflow"] << std::endl;
```

Note that using the `operator[]` on the map will actually *insert a new value* with the queried key into the map. This means that you cannot use it on a `const std::map`, even if the key is already stored in the map. To prevent this insertion, check if the element exists (for example by using `find()`) or use `at()` as described below.

Version ≥ C++11

Elements of a `std::map` can be accessed with `at()`:

```
std::cout << ranking.at("stackoverflow") << std::endl;
```

Note that `at()` will throw an `std::out_of_range` exception if the container does not contain the requested element.

In both containers `std::map` and `std::multimap`, elements can be accessed using iterators:

Version ≥ C++11

```
// Example using begin()
```

```

std::multimap < int, std::string > mmap { std::make_pair(2, "stackoverflow"),
                                         std::make_pair(1, "docs-beta"),
                                         std::make_pair(2, "stackexchange") };

auto it = mmap.begin();
std::cout << it->first << ":" << it->second << std::endl; // 输出: "1 : docs-beta"
it++;
std::cout << it->first << ":" << it->second << std::endl; // 输出: "2 : stackoverflow"
it++;
std::cout << it->first << ":" << it->second << std::endl; // 输出: "2 : stackexchange"

// 使用 rbegin() 的示例
std::map < int, std::string > mp { std::make_pair(2, "stackoverflow"),
                                   std::make_pair(1, "docs-beta"),
                                   std::make_pair(2, "stackexchange") };

auto it2 = mp.rbegin();
std::cout << it2->first << ":" << it2->second << std::endl; // 输出: "2 : stackoverflow"
it2++;
std::cout << it2->first << ":" << it2->second << std::endl; // 输出: "1 : docs-beta"

```

第50.2节：插入元素

只有当键尚未存在于std::map中时，才能向其中插入元素。例如：

```
std::map< std::string, size_t > fruits_count;
```

- 通过insert()成员函数可以向std::map中插入键值对。它需要一个pair作为参数：

```

fruits_count.insert({"grapes", 20});
fruits_count.insert(make_pair("orange", 30));
fruits_count.insert(pair<std::string, size_t>("banana", 40));
fruits_count.insert(map<std::string, size_t>::value_type("cherry", 50));

```

insert()函数返回一个由迭代器和bool值组成的pair：

- 如果插入成功，迭代器指向新插入的元素，且bool值为true。
- 如果已经存在相同的key元素，插入失败。此时，迭代器指向导致冲突的元素，且bool值为false。

以下方法可用于结合插入和查找操作：

```

auto success = fruits_count.insert({"grapes", 20});
if (!success.second) { // 我们的映射中已经有"grapes"
    success.first->second += 20; // 访问迭代器以更新值
}

```

- 为了方便，std::map容器提供了下标操作符来访问映射中的元素，如果元素不存在，还可以插入新元素：

```
fruits_count["apple"] = 10;
```

虽然更简单，但它阻止用户检查元素是否已经存在。如果元素缺失，`std::map::operator[]`会隐式创建该元素，先用默认构造函数初始化，然后再用提供的值覆盖它。

```

std::multimap < int, std::string > mmap { std::make_pair(2, "stackoverflow"),
                                         std::make_pair(1, "docs-beta"),
                                         std::make_pair(2, "stackexchange") };

auto it = mmap.begin();
std::cout << it->first << ":" << it->second << std::endl; // Output: "1 : docs-beta"
it++;
std::cout << it->first << ":" << it->second << std::endl; // Output: "2 : stackoverflow"
it++;
std::cout << it->first << ":" << it->second << std::endl; // Output: "2 : stackexchange"

// Example using rbegin()
std::map < int, std::string > mp { std::make_pair(2, "stackoverflow"),
                                   std::make_pair(1, "docs-beta"),
                                   std::make_pair(2, "stackexchange") };

auto it2 = mp.rbegin();
std::cout << it2->first << ":" << it2->second << std::endl; // Output: "2 : stackoverflow"
it2++;
std::cout << it2->first << ":" << it2->second << std::endl; // Output: "1 : docs-beta"

```

Section 50.2: Inserting elements

An element can be inserted into a std::map only if its key is not already present in the map. Given for example:

```
std::map< std::string, size_t > fruits_count;
```

- A key-value pair is inserted into a std::map through the `insert()` member function. It requires a pair as an argument:

```

fruits_count.insert({"grapes", 20});
fruits_count.insert(make_pair("orange", 30));
fruits_count.insert(pair<std::string, size_t>("banana", 40));
fruits_count.insert(map<std::string, size_t>::value_type("cherry", 50));

```

The `insert()` function returns a pair consisting of an iterator and a `bool` value:

- If the insertion was successful, the iterator points to the newly inserted element, and the `bool` value is true.
- If there was already an element with the same key, the insertion fails. When that happens, the iterator points to the element causing the conflict, and the `bool` value is false.

The following method can be used to combine insertion and searching operation:

```

auto success = fruits_count.insert({"grapes", 20});
if (!success.second) { // we already have 'grapes' in the map
    success.first->second += 20; // access the iterator to update the value
}

```

- For convenience, the std::map container provides the subscript operator to access elements in the map and to insert new ones if they don't exist:

```
fruits_count["apple"] = 10;
```

While simpler, it prevents the user from checking if the element already exists. If an element is missing, `std::map::operator[]` implicitly creates it, initializing it with the default constructor before overwriting it with the supplied value.

- `insert()` 可以使用一对对的花括号列表一次添加多个元素。这个版本的 `insert()` 返回 `void` :

```
fruits_count.insert({{"apricot", 1}, {"jackfruit", 1}, {"lime", 1}, {"mango", 7}});
```

- `insert()` 也可以使用表示 `value_type` 值的开始和结束的迭代器来添加元素 :

```
std::map< std::string, size_t > fruit_list{ {"lemon", 0}, {"olive", 0}, {"plum", 0}};
fruits_count.insert(fruit_list.begin(), fruit_list.end());
```

示例：

```
std::map<std::string, size_t> fruits_count;
std::string fruit;
while(std::cin >> fruit){
    // 插入一个以 'fruit' 为键，值为 '1' 的元素
    // (如果键已存在于 fruits_count 中, insert 不执行任何操作)
    auto ret = fruits_count.insert({fruit, 1});
    if(!ret.second){           // 'fruit' 已经存在于 map 中
        ++ret.first->second; // 计数器加一
    }
}
```

插入操作的时间复杂度为 $O(\log n)$ ，因为 `std::map` 是以树结构实现的。

版本 \geq C++11

可以使用 `make_pair()` 和 `emplace()` 显式构造一个 `pair` :

```
std::map< std::string , int > runs;
runs.emplace("Babe Ruth", 714);
runs.insert(make_pair("Barry Bonds", 762));
```

如果我们知道新元素将被插入的位置，则可以使用 `emplace_hint()` 指定一个迭代器 `hint`。如果新元素可以插入到 `hint` 之前，则插入操作可以在常数时间内完成。否则，它的行为与 `emplace()` 相同：

```
std::map< std::string , int > runs;
auto it = runs.emplace("Barry Bonds", 762); // 获取指向插入元素的迭代器
// 下一个元素将位于 "Barry Bonds" 之前，因此它被插入在 'it' 之前
runs.emplace_hint(it, "Babe Ruth", 714);
```

第50.3节：在 `std::map` 或 `std::multimap` 中搜索

有多种方法可以在`std::map`或`std::multimap`中搜索键。

- 要获取键的第一个出现位置的迭代器，可以使用`find()`函数。如果键不存在，则返回`end()`。

```
std::multimap<int,int> mmp{ {1,2},{3,4},{6,5},{8,9},{3,4},{6,7} };
auto it = mmp.find(6);
if(it!=mmp.end())
std::cout << it->first << ", " << it->second << std::endl; //输出: 6, 5
else
```

- `insert()` can be used to add several elements at once using a braced list of pairs. This version of `insert()` returns `void`:

```
fruits_count.insert({{"apricot", 1}, {"jackfruit", 1}, {"lime", 1}, {"mango", 7}});
```

- `insert()` can also be used to add elements by using iterators denoting the begin and end of `value_type` values:

```
std::map< std::string, size_t > fruit_list{ {"lemon", 0}, {"olive", 0}, {"plum", 0}};
fruits_count.insert(fruit_list.begin(), fruit_list.end());
```

Example:

```
std::map<std::string, size_t> fruits_count;
std::string fruit;
while(std::cin >> fruit){
    // insert an element with 'fruit' as key and '1' as value
    // (if the key is already stored in fruits_count, insert does nothing)
    auto ret = fruits_count.insert({fruit, 1});
    if(!ret.second){           // 'fruit' is already in the map
        ++ret.first->second; // increment the counter
    }
}
```

Time complexity for an insertion operation is $O(\log n)$ because `std::map` are implemented as trees.

Version \geq C++11

A pair can be constructed explicitly using `make_pair()` and `emplace()`:

```
std::map< std::string , int > runs;
runs.emplace("Babe Ruth", 714);
runs.insert(make_pair("Barry Bonds", 762));
```

If we know where the new element will be inserted, then we can use `emplace_hint()` to specify an iterator hint. If the new element can be inserted just before `hint`, then the insertion can be done in constant time. Otherwise it behaves in the same way as `emplace()`:

```
std::map< std::string , int > runs;
auto it = runs.emplace("Barry Bonds", 762); // get iterator to the inserted element
// the next element will be before "Barry Bonds", so it is inserted before 'it'
runs.emplace_hint(it, "Babe Ruth", 714);
```

Section 50.3: Searching in `std::map` or in `std::multimap`

There are several ways to search a key in `std::map` or in `std::multimap`.

- To get the iterator of the first occurrence of a key, the `find()` function can be used. It returns `end()` if the key does not exist.

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
auto it = mmp.find(6);
if(it!=mmp.end())
    std::cout << it->first << ", " << it->second << std::endl; //prints: 6, 5
else
```

```

std::cout << "值不存在！" << std::endl;

it = mmp.find(66);
if(it!=mmp.end())
    std::cout << it->first << ", " << it->second << std::endl;
else
    std::cout << "值不存在！" << std::endl; // 这行将被执行。

```

- 另一种判断条目是否存在于`std::map`或`std::multimap`中的方法是使用`count()`函数，该函数统计与给定键关联的值的数量。由于`std::map`每个键只关联一个值，其`count()`函数只能返回0（如果键不存在）或1（如果存在）。对于`std::multimap`，`count()`可以返回大于1的值，因为同一个键可能关联多个值。

```

std::map< int , int > mp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
if(mp.count(3) > 0) // 3 作为键存在于 map 中
std::cout << "键存在！" << std::endl; // 这行代码将被执行。
否则
std::cout << "键不存在！" << std::endl;

```

如果你只关心某个元素是否存在，`find`函数更好：它明确表达了你的意图，并且对于`multimap`来说，一旦找到第一个匹配元素就可以停止。

- 对于`std::multimap`，可能有多个元素具有相同的键。要获取这个范围，使用`equal_range()`函数，它返回一个`std::pair`，分别包含迭代器的下界（包含）和上界（不包含）。如果键不存在，两个迭代器都会指向`end()`。

```

auto eqr = mmp.equal_range(6);
auto st = eqr.first, en = eqr.second;
for(auto it = st; it != en; ++it){
    std::cout << it->first << ", " << it->second << std::endl;
}
// 输出: 6, 5
//       6, 7

```

第50.4节：初始化 `std::map` 或 `std::multimap`

`std::map` 和 `std::multimap` 都可以通过提供以逗号分隔的键值对来初始化。键值对可以通过`{key, value}`提供，也可以显式地通过`std::make_pair(key, value)`创建。由于`std::map`不允许重复键，且逗号运算符从右向左执行，右边的键值对会被左边具有相同键的键值对覆盖。

```

std::multimap <int, std::string> mmp { std::make_pair(2, "stackoverflow"),
                                         std::make_pair(1, "docs-beta"),
                                         std::make_pair(2, "stackexchange") };
// 1 docs-beta
// 2 stackoverflow
// 2 stackexchange

std::map <int, std::string> mp { std::make_pair(2, "stackoverflow"),
                                 std::make_pair(1, "docs-beta"),
                                 std::make_pair(2, "stackexchange") };
// 1 docs-beta
// 2 stackoverflow

```

```

std::cout << "Value does not exist!" << std::endl;

it = mmp.find(66);
if(it!=mmp.end())
    std::cout << it->first << ", " << it->second << std::endl;
else
    std::cout << "Value does not exist!" << std::endl; // This line would be executed.

```

- 另一种方式在`std::map`或`std::multimap`中查找条目是否存在是使用`count()`函数，该函数统计与给定键关联的值的数量。由于`std::map`每个键只关联一个值，其`count()`函数只能返回0（如果键不存在）或1（如果存在）。对于`std::multimap`，`count()`可以返回大于1的值，因为同一个键可能关联多个值。

```

std::map< int , int > mp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
if(mp.count(3) > 0) // 3 exists as a key in map
    std::cout << "The key exists!" << std::endl; // This line would be executed.
else
    std::cout << "The key does not exist!" << std::endl;

```

If you only care whether some element exists, `find` is strictly better: it documents your intent and, for `multimaps`, it can stop once the first matching element has been found.

- 在`std::multimap`的情况下，可能有多个元素具有相同的键。要获取这个范围，使用`equal_range()`函数，它返回一个`std::pair`，分别包含迭代器的下界（包含）和上界（不包含）。如果键不存在，两个迭代器都会指向`end()`。

```

auto eqr = mmp.equal_range(6);
auto st = eqr.first, en = eqr.second;
for(auto it = st; it != en; ++it){
    std::cout << it->first << ", " << it->second << std::endl;
}
// prints: 6, 5
//         6, 7

```

Section 50.4: Initializing a `std::map` or `std::multimap`

`std::map` 和 `std::multimap` 两者都可以通过提供以逗号分隔的键值对来初始化。键值对可以通过`{key, value}`提供，也可以显式地通过`std::make_pair(key, value)`创建。由于`std::map`不允许重复键，且逗号运算符从右向左执行，右边的键值对会被左边具有相同键的键值对覆盖。

```

std::multimap < int, std::string > mmp { std::make_pair(2, "stackoverflow"),
                                             std::make_pair(1, "docs-beta"),
                                             std::make_pair(2, "stackexchange") };
// 1 docs-beta
// 2 stackoverflow
// 2 stackexchange

std::map < int, std::string > mp { std::make_pair(2, "stackoverflow"),
                                    std::make_pair(1, "docs-beta"),
                                    std::make_pair(2, "stackexchange") };
// 1 docs-beta
// 2 stackoverflow

```

两者都可以通过迭代器进行初始化。

```
// 来自 std::map 或 std::multimap 迭代器
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {6, 8}, {3, 4},
                                {6, 7} };
                                // {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 8}, {6, 7}, {8, 9}
auto it = mmp.begin();
std::advance(it,3); //光标移动到第一个 {6, 5}
std::map< int, int > mp(it, mmp.end()); // {6, 5}, {8, 9}

// 来自 std::pair 数组
std::pair< int, int > arr[10];
arr[0] = {1, 3};
arr[1] = {1, 5};
arr[2] = {2, 5};
arr[3] = {0, 1};
std::map< int, int > mp(arr, arr+4); // {0, 1}, {1, 3}, {2, 5}

// 来自 std::vector 的 std::pair
std::vector< std::pair<int, int> > v{ {1, 5}, {5, 1}, {3, 6}, {3, 2} };
std::multimap< int, int > mp(v.begin(), v.end());
                                // {1, 5}, {3, 6}, {3, 2}, {5, 1}
```

第50.5节：检查元素数量

容器std::map有一个成员函数empty()，返回true或false，取决于map是否为空。成员函数size()返回存储在std::map容器中的元素数量：

```
std::map<std::string, int> rank {{"facebook.com", 1}, {"google.com", 2}, {"youtube.com", 3}};
if(!rank.empty()){
    std::cout << "rank map中的元素数量: " << rank.size() << std::endl;
}
else{
    std::cout << "rank map为空" << std::endl;
}
```

第50.6节：Map的类型

常规Map

Map是一种关联容器，包含键值对。

```
#include <string>
#include <map>
std::map<std::string, size_t> fruits_count;
```

在上述示例中，std::string 是 key 类型，size_t 是 value。

键在映射中充当索引。每个键必须唯一，且必须有序。

- 如果需要多个相同键的元素，可以考虑使用 multimap（如下所述）
- 如果你的值类型没有指定任何排序，或者你想覆盖默认排序，可以提供一个：

```
#include <string>
#include <map>
```

Both could be initialized with iterator.

```
// From std::map or std::multimap iterator
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {6, 8}, {3, 4},
                                {6, 7} };
                                // {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 8}, {6, 7}, {8, 9}
auto it = mmp.begin();
std::advance(it,3); //moved cursor on first {6, 5}
std::map< int, int > mp(it, mmp.end()); // {6, 5}, {8, 9}

//From std::pair array
std::pair< int, int > arr[10];
arr[0] = {1, 3};
arr[1] = {1, 5};
arr[2] = {2, 5};
arr[3] = {0, 1};
std::map< int, int > mp(arr, arr+4); // {0, 1}, {1, 3}, {2, 5}

//From std::vector of std::pair
std::vector< std::pair<int, int> > v{ {1, 5}, {5, 1}, {3, 6}, {3, 2} };
std::multimap< int, int > mp(v.begin(), v.end());
                                // {1, 5}, {3, 6}, {3, 2}, {5, 1}
```

Section 50.5: Checking number of elements

The container std::map has a member function empty(), which returns true or false, depending on whether the map is empty or not. The member function size() returns the number of element stored in a std::map container:

```
std::map<std::string, int> rank {{"facebook.com", 1}, {"google.com", 2}, {"youtube.com", 3}};
if(!rank.empty()){
    std::cout << "Number of elements in the rank map: " << rank.size() << std::endl;
}
else{
    std::cout << "The rank map is empty" << std::endl;
}
```

Section 50.6: Types of Maps

Regular Map

A map is an associative container, containing key-value pairs.

```
#include <string>
#include <map>
std::map<std::string, size_t> fruits_count;
```

In the above example, std::string is the key type, and size_t is a value.

The key acts as an index in the map. Each key must be unique, and must be ordered.

- If you need multiple elements with the same key, consider using multimap (explained below)
- If your value type does not specify any ordering, or you want to override the default ordering, you may provide one:

```
#include <string>
#include <map>
```

```

#include <cstring>
struct StrLess {
    bool operator()(const std::string& a, const std::string& b) {
        return strncmp(a.c_str(), b.c_str(), 8)<0;
        //只比较前8个字符
    }
}
std::map<std::string, size_t, StrLess> fruits_count2;

```

如果StrLess比较器对两个键返回false，则即使它们的实际内容不同，也被视为相同。

```

#include <cstring>
struct StrLess {
    bool operator()(const std::string& a, const std::string& b) {
        return strncmp(a.c_str(), b.c_str(), 8)<0;
        //compare only up to 8 first characters
    }
}
std::map<std::string, size_t, StrLess> fruits_count2;

```

If StrLess comparator returns `false` for two keys, they are considered the same even if their actual contents differ.

多重映射

多重映射允许在映射中存储多个具有相同键的键值对。除此之外，它的接口和创建方式与普通映射非常相似。

```

#include <string>
#include <map>
std::multimap<std::string, size_t> fruits_count;
std::multimap<std::string, size_t, StrLess> fruits_count2;

```

哈希映射（无序映射）

哈希映射存储键值对，类似于普通映射。但它不会根据键对元素进行排序。相反，使用键的哈希值来快速访问所需的键值对。

```

#include <string>
#include <unordered_map>
std::unordered_map<std::string, size_t> fruits_count;

```

无序映射通常更快，但元素不会以任何可预测的顺序存储。例如，遍历一个`unordered_map`中的所有元素时，元素的顺序看起来是随机的。

第50.7节：删除元素

删除所有元素：

```

std::multimap<int,int> mmp{ {1,2},{3,4},{6,5},{8,9},{3,4},{6,7} };
mmp.clear(); //清空multimap

```

借助迭代器从某处删除元素：

```

std::multimap<int,int> mmp{ {1,2},{3,4},{6,5},{8,9},{3,4},{6,7} };
// {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 7}, {8, 9}
auto it = mmp.begin();
std::advance(it,3); // 将光标移动到第一个 {6, 5}
mmp.erase(it); // {1, 2}, {3, 4}, {3, 4}, {6, 7}, {8, 9}

```

删除一个范围内的所有元素：

```

std::multimap<int,int> mmp{ {1,2},{3,4},{6,5},{8,9},{3,4},{6,7} };
// {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 7}, {8, 9}
auto it = mmp.begin();
auto it2 = it;

```

Multi-Map

Multimap allows multiple key-value pairs with the same key to be stored in the map. Otherwise, its interface and creation is very similar to the regular map.

```

#include <string>
#include <map>
std::multimap<std::string, size_t> fruits_count;
std::multimap<std::string, size_t, StrLess> fruits_count2;

```

Hash-Map (Unordered Map)

A hash map stores key-value pairs similar to a regular map. It does not order the elements with respect to the key though. Instead, a hash value for the key is used to quickly access the needed key-value pairs.

```

#include <string>
#include <unordered_map>
std::unordered_map<std::string, size_t> fruits_count;

```

Unordered maps are usually faster, but the elements are not stored in any predictable order. For example, iterating over all elements in an `unordered_map` gives the elements in a seemingly random order.

Section 50.7: Deleting elements

Removing all elements:

```

std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
mmp.clear(); //empty multimap

```

Removing element from somewhere with the help of iterator:

```

std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
// {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 7}, {8, 9}
auto it = mmp.begin();
std::advance(it,3); // moved cursor on first {6, 5}
mmp.erase(it); // {1, 2}, {3, 4}, {3, 4}, {6, 7}, {8, 9}

```

Removing all elements in a range:

```

std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
// {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 7}, {8, 9}
auto it = mmp.begin();
auto it2 = it;

```

```
it++; // 将第一个光标移动到第一个 {3, 4}
std::advance(it2, 3); // 将第二个光标移动到第一个 {6, 5}
mmp.erase(it, it2); // {1, 2}, {6, 5}, {6, 7}, {8, 9}
```

删除所有具有指定值作为键的元素：

```
std::multimap<int,int> mmp{ {1,2},{3,4},{6,5},{8,9},{3,4},{6,7} };
// {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 7}, {8, 9}
mmp.erase(6); // {1, 2}, {3, 4}, {3, 4}, {8, 9}
```

移除满足谓词 pred 的元素：

```
std::map<int,int> m;
auto it = m.begin();
while (it != m.end())
{
    if (pred(*it))
        it = m.erase(it);
    else
        ++it;
}
```

第50.8节：遍历 std::map 或 std::multimap

std::map 或 std::multimap 可以通过以下方式遍历：

```
std::multimap<int,int> mmp{ {1,2},{3,4},{6,5},{8,9},{3,4},{6,7} };

// 基于范围的循环 - 自 C++11 起
for(const auto &x: mmp)
    std::cout << x.first << ":" << x.second << std::endl;

// 前向迭代器 for 循环：它会从第一个元素循环到最后一个元素
// it 将是 std::map< int, int >::iterator 类型
for (auto it = mmp.begin(); it != mmp.end(); ++it)
    std::cout << it->first << ":" << it->second << std::endl; // 使用迭代器做某些操作

// 反向迭代器 for 循环：它将从最后一个元素循环到第一个元素
// it 将是 std::map< int, int >::reverse_iterator
for (auto it = mmp.rbegin(); it != mmp.rend(); ++it)
    std::cout << it->first << " " << it->second << std::endl; // 使用迭代器做某些操作
```

在遍历 std::map 或 std::multimap 时，优先使用 auto 以避免无用的隐式转换（详见该 SO 回答）。

第50.9节：使用用户自定义类型作为

为了能够将类用作 map 中的键，键只需满足可复制和可赋值的要求。map 中的排序由模板的第三个参数（以及构造函数的参数，如果使用的话）定义。该参数默认是 std::less<KeyType>，默认使用<操作符，但并不要求必须使用默认。只需编写一个比较操作符（最好是函数对象）：

```
struct CmpMyType
{
    bool operator()( MyType const& lhs, MyType const& rhs ) const
    {
        // 比较逻辑
    }
}
```

```
it++; // moved first cursor on first {3, 4}
std::advance(it2, 3); // moved second cursor on first {6, 5}
mmp.erase(it, it2); // {1, 2}, {6, 5}, {6, 7}, {8, 9}
```

Removing all elements having a provided value as key:

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
// {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 7}, {8, 9}
mmp.erase(6); // {1, 2}, {3, 4}, {3, 4}, {8, 9}
```

Removing elements that satisfy a predicate pred:

```
std::map<int,int> m;
auto it = m.begin();
while (it != m.end())
{
    if (pred(*it))
        it = m.erase(it);
    else
        ++it;
}
```

Section 50.8: Iterating over std::map or std::multimap

std::map 或 std::multimap 可以通过以下方式遍历：

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };

// Range based loop - since C++11
for(const auto &x: mmp)
    std::cout << x.first << ":" << x.second << std::endl;

// Forward iterator for loop: it would loop through first element to last element
// it will be a std::map< int, int >::iterator
for (auto it = mmp.begin(); it != mmp.end(); ++it)
    std::cout << it->first << ":" << it->second << std::endl; // Do something with iterator

// Backward iterator for loop: it would loop through last element to first element
// it will be a std::map< int, int >::reverse_iterator
for (auto it = mmp.rbegin(); it != mmp.rend(); ++it)
    std::cout << it->first << " " << it->second << std::endl; // Do something with iterator
```

While iterating over a std::map or a std::multimap, the use of `auto` is preferred to avoid useless implicit conversions (see [this SO answer](#) for more details).

Section 50.9: Creating std::map with user-defined types as key

In order to be able to use a class as the key in a map, all that is required of the key is that it be copyable and assignable. The ordering within the map is defined by the third argument to the template (and the argument to the constructor, if used). This *defaults* to `std::less<KeyType>`, which defaults to the `<` operator, but there's no requirement to use the defaults. Just write a comparison operator (preferably as a functional object):

```
struct CmpMyType
{
    bool operator()( MyType const& lhs, MyType const& rhs ) const
    {
        // Comparison logic
    }
}
```

```
// ...
}
};
```

在 C++ 中，“compare”谓词必须是严格弱序。特别地，`compare(X,X)` 对于任何 X 必须返回 `false`。即如果 `CmpMyTypee()(a, b)` 返回 `true`，那么 `CmpMyType()(b, a)` 必须返回 `false`；如果两者都返回 `false`，则认为元素相等（属于同一等价类）。

严格弱序

这是一个数学术语，用于定义两个对象之间的关系。

其定义如下：

如果对于两个对象 `x` 和 `y`, $f(x, y)$ 和 $f(y, x)$ 都为假，则这两个对象是等价的。注意，根据不可自反性不变式，一个对象总是等价于它自身的。

在 C++ 中，这意味着如果你有两个给定类型的对象，使用小于运算符 `<` 比较时，应返回以下值。

```
X    a;
X    b;
```

条件:	测试:	结果
a 等价于 b:	<code>a < b</code>	假
a 等价于 b	<code>b < a</code>	假
a 小于 b	<code>a < b</code>	真
a 小于 b	<code>b < a</code>	假
b 小于 a	<code>a < b</code>	假
b 小于 a	<code>b < a</code>	真

你如何定义等价/小于完全取决于对象的类型。

```
// ...
}
};
```

In C++, the "compare" predicate must be a strict weak ordering. In particular, `compare(X,X)` must return `false` for any X. i.e. if `CmpMyType()(a, b)` returns true, then `CmpMyType()(b, a)` must return false, and if both return false, the elements are considered equal (members of the same equivalence class).

Strict Weak Ordering

This is a mathematical term to define a relationship between two objects.

Its definition is:

Two objects `x` and `y` are equivalent if both $f(x, y)$ and $f(y, x)$ are false. Note that an object is always (by the irreflexivity invariant) equivalent to itself.

In terms of C++ this means if you have two objects of a given type, you should return the following values when compared with the operator `<`.

```
X    a;
X    b;
```

Condition:	Test:	Result
a is equivalent to b:	<code>a < b</code>	<code>false</code>
a is equivalent to b	<code>b < a</code>	<code>false</code>
a is less than b	<code>a < b</code>	<code>true</code>
a is less than b	<code>b < a</code>	<code>false</code>
b is less than a	<code>a < b</code>	<code>false</code>
b is less than a	<code>b < a</code>	<code>true</code>

How you define equivalent/less is totally dependent on the type of your object.

第51章：std::optional

第51.1节：使用optional表示值的缺失

在C++17之前，指针值为nullptr通常表示值的缺失。这对于已经由指针管理且动态分配的大型对象来说是一个很好的解决方案。然而，这种解决方案对于像int这样很少动态分配或由指针管理的小型或原始类型并不适用。std::optional提供了一个针对这一常见问题的可行解决方案。

在这个例子中，定义了struct Person。一个人可能有宠物，但不是必须的。因此，Person的pet成员被声明为带有std::optional包装器的类型。

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

struct Animal {
    std::string name;
};

struct Person {
    std::string name;
    std::optional<Animal> pet;
};

int main() {
    Person person;
    person.name = "John";

    if (person.pet) {
        std::cout << person.name << "'s pet's name is " <<
            person.pet->name << std::endl;
    }
    else {
        std::cout << person.name << " 独自一人。" << std::endl;
    }
}
```

第51.2节：作为返回值的optional

```
std::optional<float> divide(float a, float b) {
    if (b!=0.f) return a/b;
    return {};
}
```

这里我们返回分数 a/b ，但如果它未定义（将是无穷大），我们则返回空的optional。

一个更复杂的例子：

```
template<class Range, class Pred>
auto find_if( Range&& r, Pred&& p ) {
    using std::begin; using std::end;
    auto b = begin(r), e = end(r);
    auto r = std::find_if(b, e , p );
    using iterator = decltype(r);
```

Chapter 51: std::optional

Section 51.1: Using optionals to represent the absence of a value

Before C++17, having pointers with a value of `nullptr` commonly represented the absence of a value. This is a good solution for large objects that have been dynamically allocated and are already managed by pointers. However, this solution does not work well for small or primitive types such as `int`, which are rarely ever dynamically allocated or managed by pointers. `std::optional` provides a viable solution to this common problem.

In this example, `struct Person` is defined. It is possible for a person to have a pet, but not necessary. Therefore, the `pet` member of `Person` is declared with an `std::optional` wrapper.

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

struct Animal {
    std::string name;
};

struct Person {
    std::string name;
    std::optional<Animal> pet;
};

int main() {
    Person person;
    person.name = "John";

    if (person.pet) {
        std::cout << person.name << "'s pet's name is " <<
            person.pet->name << std::endl;
    }
    else {
        std::cout << person.name << " is alone." << std::endl;
    }
}
```

Section 51.2: optional as return value

```
std::optional<float> divide(float a, float b) {
    if (b!=0.f) return a/b;
    return {};
}
```

Here we return either the fraction a/b , but if it is not defined (would be infinity) we instead return the empty optional.

A more complex case:

```
template<class Range, class Pred>
auto find_if( Range&& r, Pred&& p ) {
    using std::begin; using std::end;
    auto b = begin(r), e = end(r);
    auto r = std::find_if(b, e , p );
    using iterator = decltype(r);
```

```

if (r==e)
    return std::optional<iterator>();
return std::optional<iterator>(r);
}
template<class 范围, class T>
auto find( 范围&& r, T const& t ) {
    return find_if( std::forward<范围>(r), [&t](auto&& x){return x==t;} );
}

```

find(some_range, 7) 在容器或范围 some_range 中搜索等于数字 7 的元素。

find_if 使用谓词进行搜索。

如果未找到，则返回空的 optional；如果找到，则返回包含指向该元素的迭代器的 optional。

这允许你这样做：

```

if (find( vec, 7 )) {
    // 代码
}

```

甚至可以这样写

```

if (auto oit = find( vec, 7 )) {
    vec.erase(*oit);
}

```

无需处理 begin/end 迭代器和测试。

第51.3节：value_or

```

void print_name( std::ostream& os, std::optional<std::string> const& name ) { std::cout "Name is: " << name.value_or("<name missing>") << ";
}

```

value_or 要么返回存储在optional中的值，要么返回参数（如果没有存储任何值）。

这使你可以处理可能为空的optional，并在实际需要值时提供默认行为。通过这种方式，“默认行为”的决定可以推迟到最合适且立即需要的时刻，而不是在某个引擎深处生成某个默认值。

第51.4节：介绍

Optionals（也称为Maybe类型）用于表示内容可能存在也可能不存在的类型。

它们在C++17中作为std::optional类实现。例如，类型为std::optional<int>的对象可能包含某个int类型的值，也可能不包含任何值。

Optionals通常用于表示可能不存在的值，或作为可能无法返回有意义结果的函数的返回类型。

其他处理optional的方法

解决std::optional问题还有许多其他方法，但都不够完整：使用指针、使用哨兵值，或使用pair<bool, T>。

可选类型与指针

```

if (r==e)
    return std::optional<iterator>();
return std::optional<iterator>(r);
}
template<class Range, class T>
auto find( Range&& r, T const& t ) {
    return find_if( std::forward<Range>(r), [&t](auto&& x){return x==t;} );
}

```

find(some_range, 7) searches the container or range some_range for something equal to the number 7. find_if does it with a predicate.

It returns either an empty optional if it was not found, or an optional containing an iterator to the element if it was.

This allows you to do:

```

if (find( vec, 7 )) {
    // code
}

```

or even

```

if (auto oit = find( vec, 7 )) {
    vec.erase(*oit);
}

```

without having to mess around with begin/end iterators and tests.

Section 51.3: value_or

```

void print_name( std::ostream& os, std::optional<std::string> const& name ) {
    std::cout "Name is: " << name.value_or("<name missing>") << '\n';
}

```

value_or either returns the value stored in the optional, or the argument if there is nothing stored there.

This lets you take the maybe-null optional and give a default behavior when you actually need a value. By doing it this way, the "default behavior" decision can be pushed back to the point where it is best made and immediately needed, instead of generating some default value deep in the guts of some engine.

Section 51.4: Introduction

Optionals (also known as Maybe types) are used to represent a type whose contents may or may not be present. They are implemented in C++17 as the std::optional class. For example, an object of type std::optional<int> may contain some value of type int, or it may contain no value.

Optionals are commonly used either to represent a value that may not exist or as a return type from a function that can fail to return a meaningful result.

Other approaches to optional

There are many other approaches to solving the problem that std::optional solves, but none of them are quite complete: using a pointer, using a sentinel, or using a pair<bool, T>.

Optional vs Pointer

在某些情况下，我们可以提供指向现有对象的指针或nullptr来表示失败。但这仅限于对象已经存在的情况——作为值类型的optional也可以用来返回新对象，而无需进行内存分配。

可选类型与哨兵值

一种常见的习惯用法是使用特殊值来表示该值无意义。对于整型类型，这可能是0或-1，或者对于指针是nullptr。然而，这会减少有效值的空间（你无法区分有效的0和无意义的0），而且许多类型没有自然的哨兵值选择。

可选类型与std::pair<bool, T>

另一种常见的习惯用法是提供一个pair，其中一个元素是bool，表示该值是否有意义。

这依赖于值类型在错误情况下可默认构造，而这对某些类型来说是不可能的，对另一些类型来说虽然可能但不理想。对于错误情况，optional<T>不需要构造任何东西。

第51.5节：使用optional表示函数失败

在C++17之前，函数通常通过几种方式之一来表示失败：

- 返回了空指针。
 - 例如，在一个没有委托的 App 实例上调用函数 Delegate *App::get_delegate() 会返回 nullptr
 - 这是对动态分配或由指针管理的大型对象的一个好解决方案，但对于通常在栈上分配并通过复制传递的小型对象来说，这不是一个好的解决方案。
- 保留了返回类型的一个特定值来表示失败。
 - 例如，在两个不相连的顶点上调用函数 unsigned shortest_path_distance(Vertex a, Vertex b) 可能返回零以表示这一事实。
- 该值与一个 bool 配对，以指示返回值是否有意义。
 - 例如，调用函数 std::pair<int, bool> parse(const std::string &str)，传入一个不是整数的字符串参数，会返回一个 int 未定义且 bool 设置为 false 的对。

在这个例子中，约翰有两只宠物，Fluffy 和 Furball。然后调用函数 Person::pet_with_name() 来获取约翰的宠物 Whiskers。由于约翰没有名为 Whiskers 的宠物，函数失败并返回 std::nullopt。

```
#include <iostream>
#include <optional>
#include <string>
#include <vector>

struct Animal {
    std::string name;
};

struct Person {
    std::string name;
    std::vector<Animal> pets;
};

std::optional<Animal> 带名字的宠物(const std::string &名字) {
    for (const Animal &宠物 : pets) {
```

In some cases, we can provide a pointer to an existing object or nullptr to indicate failure. But this is limited to those cases where objects already exist - optional, as a value type, can also be used to return new objects without resorting to memory allocation.

Optional vs Sentinel

A common idiom is to use a special value to indicate that the value is meaningless. This may be 0 or -1 for integral types, or nullptr for pointers. However, this reduces the space of valid values (you cannot differentiate between a valid 0 and a meaningless 0) and many types do not have a natural choice for the sentinel value.

Optional vs std::pair<bool, T>

Another common idiom is to provide a pair, where one of the elements is a bool indicating whether or not the value is meaningful.

This relies upon the value type being default-constructible in the case of error, which is not possible for some types and possible but undesirable for others. An optional<T>, in the case of error, does not need to construct anything.

Section 51.5: Using optionals to represent the failure of a function

Before C++17, a function typically represented failure in one of several ways:

- A null pointer was returned.
 - e.g. Calling a function Delegate *App::get_delegate() on an App instance that did not have a delegate would return nullptr.
 - This is a good solution for objects that have been dynamically allocated or are large and managed by pointers, but isn't a good solution for small objects that are typically stack-allocated and passed by copying.
- A specific value of the return type was reserved to indicate failure.
 - e.g. Calling a function unsigned shortest_path_distance(Vertex a, Vertex b) on two vertices that are not connected may return zero to indicate this fact.
- The value was paired together with a bool to indicate if the returned value was meaningful.
 - e.g. Calling a function std::pair<int, bool> parse(const std::string &str) with a string argument that is not an integer would return a pair with an undefined int and a bool set to false.

In this example, John is given two pets, Fluffy and Furball. The function Person::pet_with_name() is then called to retrieve John's pet Whiskers. Since John does not have a pet named Whiskers, the function fails and std::nullopt is returned instead.

```
#include <iostream>
#include <optional>
#include <string>
#include <vector>

struct Animal {
    std::string name;
};

struct Person {
    std::string name;
    std::vector<Animal> pets;
};

std::optional<Animal> pet_with_name(const std::string &name) {
    for (const Animal &pet : pets) {
```

```

    if (宠物.名字 == 名字) {
        return 宠物;
    }
}
return std::nullopt;
};

int main() {
人物 约翰;
约翰.名字 = "John";

动物 毛茸茸;
毛茸茸.名字 = "Fluffy";
约翰.宠物.push_back(毛茸茸);

动物 毛球;
毛球.名字 = "Furball";
约翰.宠物.push_back(毛球);

std::optional<Animal> 胡须 = john.pet_with_name("Whiskers");
if (胡须) {
    std::cout << "约翰有一只名叫Whiskers的宠物。" << std::endl;
} else {
    std::cout << "Whiskers一定不属于约翰。" << std::endl;
}
}

```

```

    if (pet.name == name) {
        return pet;
    }
}
return std::nullopt;
};

int main() {
Person john;
john.name = "John";

Animal fluffy;
fluffy.name = "Fluffy";
john.pets.push_back(fluffy);

Animal furball;
furball.name = "Furball";
john.pets.push_back(furball);

std::optional<Animal> whiskers = john.pet_with_name("Whiskers");
if (whiskers) {
    std::cout << "John has a pet named Whiskers." << std::endl;
} else {
    std::cout << "Whiskers must not belong to John." << std::endl;
}
}

```

第52章：std::function：封装任何可调用元素

第52.1节：简单用法

```
#include <iostream>
#include <functional>
std::function<void(int, const std::string&)> myFuncObj;
void theFunc(int i, const std::string& s)
{
    std::cout << s << ":" << i << std::endl;
}
int main(int argc, char *argv[])
{
    myFuncObj = theFunc;
    myFuncObj(10, "hello world");
}
```

第52.2节：std::function 与 std::bind 的使用

考虑一种需要带参数回调函数的情况。使用 std::function 结合 std::bind 提供了如下所示的强大设计构造。

```
类 A
{
    公共:
        std::function<void(int, const std::string&)> m_CbFunc = nullptr;
        void foo()
        {
            如果 (m_CbFunc)
            {
                m_CbFunc(100, "事件触发");
            }
        }
};

类 B
{
    公共:
        B()
        {
            auto aFunc = std::bind(&B::eventHandler, this, std::placeholders::_1,
                std::placeholders::_2);
            anObjA.m_CbFunc = aFunc;
        }
        void eventHandler(int i, const std::string& s)
        {
            std::cout << s << ":" << i << std::endl;
        }

        void DoSomethingOnA()
        {
            anObjA.foo();
        }
};

A anObjA;
```

Chapter 52: std::function: To wrap any element that is callable

Section 52.1: Simple usage

```
#include <iostream>
#include <functional>
std::function<void(int, const std::string&)> myFuncObj;
void theFunc(int i, const std::string& s)
{
    std::cout << s << ":" << i << std::endl;
}
int main(int argc, char *argv[])
{
    myFuncObj = theFunc;
    myFuncObj(10, "hello world");
}
```

Section 52.2: std::function used with std::bind

Think about a situation where we need to callback a function with arguments. std::function used with std::bind gives a very powerful design construct as shown below.

```
class A
{
public:
    std::function<void(int, const std::string&)> m_CbFunc = nullptr;
    void foo()
    {
        if (m_CbFunc)
        {
            m_CbFunc(100, "event fired");
        }
    }
};

class B
{
public:
    B()
    {
        auto aFunc = std::bind(&B::eventHandler, this, std::placeholders::_1,
            std::placeholders::_2);
        anObjA.m_CbFunc = aFunc;
    }
    void eventHandler(int i, const std::string& s)
    {
        std::cout << s << ":" << i << std::endl;
    }

    void DoSomethingOnA()
    {
        anObjA.foo();
    }
};

A anObjA;
```

```

int main(int argc, char *argv[])
{
B anObjB;
anObjB.DoSomethingOnA();
}

```

第52.3节：将std::function绑定到不同的可调用类型

```

/*
* 本示例展示了使用std::function调用的几种方式
* a) 类C函数
* b) 类成员函数
* c) operator()
* d) lambda函数
*
* 函数调用可以通过以下方式进行：
* a) 使用正确的参数 * b) 参数
顺序、类型和数量不同

#include <iostream>
#include <functional>
#include <iostream>
#include <vector>

using std::cout;
using std::endl;
using namespace std::placeholders;

```

```

// 简单的函数调用
double foo_fn(int x, float y, double z)
{
    double res = x + y + z;
    std::cout << "foo_fn 调用时的参数: "
        << x << ", " << y << ", " << z
        << " 结果是 : " << res
        << std::endl;
    return res;
}

// 带有成员函数调用的结构体
struct foo_struct
{
    // 成员函数调用
    double foo_fn(int x, float y, double z)
    {
        double res = x + y + z;
        std::cout << "foo_struct::foo_fn 调用时的参数: "
            << x << ", " << y << ", " << z
            << " 结果是 : " << res
            << std::endl;
        return res;
    }
    // 这个成员函数有不同的签名 - 但也可以使用
    // 请注意参数顺序也发生了变化
    double foo_fn_4(int x, double z, float y, long xx)
    {
        double res = x + y + z + xx;
        std::cout << "foo_struct::foo_fn_4 调用, 参数为: "
    }
}
```

```

int main(int argc, char *argv[])
{
B anObjB;
anObjB.DoSomethingOnA();
}

```

Section 52.3: Binding std::function to a different callable types

```

/*
* This example show some ways of using std::function to call
* a) C-like function
* b) class-member function
* c) operator()
* d) lambda function
*
* Function call can be made:
* a) with right arguments
* b) arguments with different order, types and count
*/
#include <iostream>
#include <functional>
#include <iostream>
#include <vector>

using std::cout;
using std::endl;
using namespace std::placeholders;

// simple function to be called
double foo_fn(int x, float y, double z)
{
    double res = x + y + z;
    std::cout << "foo_fn called with arguments: "
        << x << ", " << y << ", " << z
        << " result is : " << res
        << std::endl;
    return res;
}

// structure with member function to call
struct foo_struct
{
    // member function to call
    double foo_fn(int x, float y, double z)
    {
        double res = x + y + z;
        std::cout << "foo_struct::foo_fn called with arguments: "
            << x << ", " << y << ", " << z
            << " result is : " << res
            << std::endl;
        return res;
    }
    // this member function has different signature - but it can be used too
    // please note that argument order is changed too
    double foo_fn_4(int x, double z, float y, long xx)
    {
        double res = x + y + z + xx;
        std::cout << "foo_struct::foo_fn_4 called with arguments: "
    }
}
```

```

        << x << ", " << z << ", " << y << ", " << xx
        << " 结果是 : " << res
        << std::endl;
    return res;
}
// 重载的 operator() 使整个对象可调用
double operator()(int x, float y, double z)
{
    double res = x + y + z;
std::cout << "foo_struct::operator() 调用, 参数为: "
        << x << ", " << y << ", " << z
        << " 结果是 : " << res
        << std::endl;
    return res;
};

int main(void)
{
    // 类型定义
    using function_type = std::function<double(int, float, double)>;
    // foo_struct 实例
foo_struct fs;

    // 这里我们将存储所有绑定的函数
std::vector<function_type> bindings;

    // 变量 #1 - 你可以使用简单函数
function_type var1 = foo_fn;
bindings.push_back(var1);

    // 变量 #2 - 你可以使用成员函数
function_type var2 = std::bind(&foo_struct::foo_fn, fs, _1, _2, _3);
bindings.push_back(var2);

    // 变量 #3 - 你可以使用不同签名的成员函数
    // foo_fn_4 有不同数量的参数和类型
function_type var3 = std::bind(&foo_struct::foo_fn_4, fs, _1, _3, _2, _01);
bindings.push_back(var3);

    // 变量 #4 - 你可以使用重载了 operator() 的对象
function_type var4 = fs;
bindings.push_back(var4);

    // 变量 #5 - 你可以使用 lambda 函数
function_type var5 = [](int x, float y, double z)
{
    double res = x + y + z;
std::cout << "lambda 调用时的参数: "
        << x << ", " << y << ", " << z
        << " 结果是 : " << res
        << std::endl;
    return res;
};
bindings.push_back(var5);

std::cout << "测试存储的函数及参数: x = 1, y = 2, z = 3"
        << std::endl;

for (auto f : bindings)

```

```

        << x << ", " << z << ", " << y << ", " << xx
        << " result is : " << res
        << std::endl;
    return res;
}
// overloaded operator() makes whole object to be callable
double operator()(int x, float y, double z)
{
    double res = x + y + z;
std::cout << "foo_struct::operator() called with arguments: "
        << x << ", " << y << ", " << z
        << " result is : " << res
        << std::endl;
    return res;
};

int main(void)
{
    // typedefs
    using function_type = std::function<double(int, float, double)>;
    // foo_struct instance
foo_struct fs;

    // here we will store all binded functions
std::vector<function_type> bindings;

    // var #1 - you can use simple function
function_type var1 = foo_fn;
bindings.push_back(var1);

    // var #2 - you can use member function
function_type var2 = std::bind(&foo_struct::foo_fn, fs, _1, _2, _3);
bindings.push_back(var2);

    // var #3 - you can use member function with different signature
    // foo_fn_4 has different count of arguments and types
function_type var3 = std::bind(&foo_struct::foo_fn_4, fs, _1, _3, _2, _01);
bindings.push_back(var3);

    // var #4 - you can use object with overloaded operator()
function_type var4 = fs;
bindings.push_back(var4);

    // var #5 - you can use lambda function
function_type var5 = [](int x, float y, double z)
{
    double res = x + y + z;
std::cout << "lambda called with arguments: "
        << x << ", " << y << ", " << z
        << " result is : " << res
        << std::endl;
    return res;
};
bindings.push_back(var5);

std::cout << "Test stored functions with arguments: x = 1, y = 2, z = 3"
        << std::endl;

for (auto f : bindings)

```

```
f(1, 2, 3);  
}
```

Live

输出：

```
测试存储的函数及参数: x = 1, y = 2, z = 3  
foo_fn 调用参数: 1, 2, 3 结果是 : 6  
foo_struct::foo_fn 调用参数: 1, 2, 3 结果是 : 6  
foo_struct::foo_fn_4 调用参数: 1, 3, 2, 0 结果是 : 6  
foo_struct::operator() 调用参数: 1, 2, 3 结果是 : 6  
lambda 调用参数: 1, 2, 3 结果是 : 6
```

第52.4节：在std::tuple中存储函数参数

有些程序需要存储参数以便将来调用某些函数。

此示例展示了如何使用存储在 std::tuple 中的参数调用任意函数

```
#include <iostream>  
#include <functional>  
#include <tuple>  
#include <iostream>  
  
// 简单的函数调用  
double foo_fn(int x, float y, double z)  
{  
    double res = x + y + z;  
    std::cout << "foo_fn called. x = " << x << " y = " << y << " z = " << z  
          << " res=" << res;  
    return res;  
}  
  
// tuple展开的辅助工具  
template<int ...> struct seq {};  
template<int N, int ...S> struct gens : gens<N-1, N-1, S...> {};  
template<int ...S> struct gens<0, S...>{ typedef seq<S...> type; };  
  
// 调用辅助工具  
template<typename FN, typename P, int ...S>  
double call_fn_internal(const FN& fn, const P& params, const seq<S...>)  
{  
    return fn(std::get<S>(params) ...);  
}  
// 使用存储在 std::tuple 中的参数调用函数  
template<typename Ret, typename ...Args>  
Ret call_fn(const std::function<Ret(Args...)>& fn,  
            const std::tuple<Args...>& params)  
{  
    return call_fn_internal(fn, params, typename gens<sizeof...(Args)>::type());  
}  
  
int main(void)  
{  
    // 参数  
    std::tuple<int, float, double> t = std::make_tuple(1, 5, 10);  
    // 要调用的函数
```

```
f(1, 2, 3);  
}
```

Live

Output:

```
Test stored functions with arguments: x = 1, y = 2, z = 3  
foo_fn called with arguments: 1, 2, 3 result is : 6  
foo_struct::foo_fn called with arguments: 1, 2, 3 result is : 6  
foo_struct::foo_fn_4 called with arguments: 1, 3, 2, 0 result is : 6  
foo_struct::operator() called with arguments: 1, 2, 3 result is : 6  
lambda called with arguments: 1, 2, 3 result is : 6
```

Section 52.4: Storing function arguments in std::tuple

Some programs need to store arguments for future calling of some function.

This example shows how to call any function with arguments stored in std::tuple

```
#include <iostream>  
#include <functional>  
#include <tuple>  
#include <iostream>  
  
// simple function to be called  
double foo_fn(int x, float y, double z)  
{  
    double res = x + y + z;  
    std::cout << "foo_fn called. x = " << x << " y = " << y << " z = " << z  
          << " res=" << res;  
    return res;  
}  
  
// helpers for tuple unrolling  
template<int ...> struct seq {};  
template<int N, int ...S> struct gens : gens<N-1, N-1, S...> {};  
template<int ...S> struct gens<0, S...>{ typedef seq<S...> type; };  
  
// invocation helper  
template<typename FN, typename P, int ...S>  
double call_fn_internal(const FN& fn, const P& params, const seq<S...>)  
{  
    return fn(std::get<S>(params) ...);  
}  
// call function with arguments stored in std::tuple  
template<typename Ret, typename ...Args>  
Ret call_fn(const std::function<Ret(Args...)>& fn,  
            const std::tuple<Args...>& params)  
{  
    return call_fn_internal(fn, params, typename gens<sizeof...(Args)>::type());  
}  
  
int main(void)  
{  
    // arguments  
    std::tuple<int, float, double> t = std::make_tuple(1, 5, 10);  
    // function to call
```

```
std::function<double(int, float, double)> fn = foo_fn;
// 使用存储的参数调用函数
call_fn(fn, t);
}
```

Live

输出：

```
foo_fn 被调用。 x = 1 y = 5 z = 10 res=16
```

第52.5节：带有lambda和std::bind的std::function

```
#include <iostream>
#include <functional>

using std::placeholders::_1; // 用于std::bind示例

int stdf_foobar (int x, std::function<int(int)> moo)
{
    return x + moo(x); // 调用了std::function moo
}

int foo (int x) { return 2+x; }

int foo_2 (int x, int y) { return 9*x + y; }

int main()
{
    int a = 2;

    /* 函数指针 */
std::cout << stdf_foobar(a, &foo) << std::endl; // 6 ( 2 + (2+2) )
    // 也可以是: stdf_foobar(2, foo)

    /* Lambda 表达式 */
    /* 来自 Lambda 表达式的无名闭包可以
     * 存储在 std::function 对象中:
     */
    int capture_value = 3;
    std::cout << stdf_foobar(a,
                            [capture_value](int param) -> int { return 7 + capture_value * param;
})
        << std::endl;
    // 结果: 15 == value + (7 * capture_value * value) == 2 + (7 + 3 * 2)

    /* std::bind 表达式 */
    /* std::bind 表达式的结果可以被传递。*/
    /* 例如, 通过将参数绑定到函数指针调用:
     */
    int b = stdf_foobar(a, std::bind(foo_2, _1, 3));
    std::cout << b << std::endl;
    // b == 23 == 2 + ( 9*2 + 3 )
    int c = stdf_foobar(a, std::bind(foo_2, 5, _1));
    std::cout << c << std::endl;
    // c == 49 == 2 + ( 9*5 + 2 )

    return 0;
}
```

```
std::function<double(int, float, double)> fn = foo_fn;
// invoke a function with stored arguments
call_fn(fn, t);
}
```

Live

Output:

```
foo_fn called. x = 1 y = 5 z = 10 res=16
```

Section 52.5: std::function with lambda and std::bind

```
#include <iostream>
#include <functional>

using std::placeholders::_1; // to be used in std::bind example

int stdf_foobar (int x, std::function<int(int)> moo)
{
    return x + moo(x); // std::function moo called
}

int foo (int x) { return 2+x; }

int foo_2 (int x, int y) { return 9*x + y; }

int main()
{
    int a = 2;

    /* Function pointers */
std::cout << stdf_foobar(a, &foo) << std::endl; // 6 ( 2 + (2+2) )
    // can also be: stdf_foobar(2, foo)

    /* Lambda expressions */
    /* An unnamed closure from a lambda expression can be
     * stored in a std::function object:
     */
    int capture_value = 3;
    std::cout << stdf_foobar(a,
                            [capture_value](int param) -> int { return 7 + capture_value * param;
})
        << std::endl;
    // result: 15 == value + (7 * capture_value * value) == 2 + (7 + 3 * 2)

    /* std::bind expressions */
    /* The result of a std::bind expression can be passed.
     * For example by binding parameters to a function pointer call:
     */
    int b = stdf_foobar(a, std::bind(foo_2, _1, 3));
    std::cout << b << std::endl;
    // b == 23 == 2 + ( 9*2 + 3 )
    int c = stdf_foobar(a, std::bind(foo_2, 5, _1));
    std::cout << c << std::endl;
    // c == 49 == 2 + ( 9*5 + 2 )

    return 0;
}
```

第52.6节：`function` 的开销

`std::function` 可能会导致显著的开销。因为 `std::function` 具有[值语义][1]，它必须将给定的可调用对象复制或移动到自身内部。但由于它可以接受任意类型的可调用对象，因此通常需要动态分配内存来实现这一点。

一些 `function` 实现具有所谓的“小对象优化”，即小类型（如函数指针、成员指针或状态非常少的函数对象）会直接存储在 `function` 对象中。但即使如此，这也仅在类型是 `noexcept` 移动构造的情况下有效。此外，C++ 标准并不要求所有实现都必须提供此功能。

考虑以下情况：

```
//头文件
using MyPredicate = std::function<bool(const MyValue &, const MyValue &)>;
void SortMyContainer(MyContainer &C, const MyPredicate &pred);

//源文件
void SortMyContainer(MyContainer &C, const MyPredicate &pred)
{
    std::sort(C.begin(), C.end(), pred);
}
```

模板参数将是 `SortMyContainer` 的首选解决方案，但假设出于某种原因这不可行或不理想。`SortMyContainer` 不需要在调用之外存储 `pred`。尽管如此，`pred` 如果传入的函数对象体积较大，可能会分配内存。

`function` 会分配内存，因为它需要复制/移动某些东西；`function` 接管它所给的可调用对象的所有权。但 `SortMyContainer` 不需要拥有该可调用对象；它只是引用它。因此这里使用 `function` 有些过度；它可能高效，也可能不高效。

标准库中没有仅仅引用可调用对象的函数类型。因此必须找到替代方案，或者你可以选择接受这种开销。

此外，`function` 没有有效的方法来控制对象内存分配的来源。是的，它有接受 `allocator` 的构造函数，但 [许多实现并未正确实现它们……甚至根本没有实现][2]。

版本 ≥ C++17

接受 `allocator` 的 `function` 构造函数不再是类型的一部分。因此，无法管理内存分配。

调用函数也比直接调用内容要慢。由于任何函数实例都可能持有任何可调用对象，通过函数的调用必须是间接的。调用函数的开销相当于虚函数调用的开销。

Section 52.6: `function` overhead

`std::function` 可能会导致显著的开销。Because `std::function` has [value semantics][1], it must copy or move the given callable into itself. But since it can take callables of an arbitrary type, it will frequently have to allocate memory dynamically to do this.

Some function implementations have so-called "small object optimization", where small types (like function pointers, member pointers, or functors with very little state) will be stored directly in the function object. But even this only works if the type is `noexcept` move constructible. Furthermore, the C++ standard does not require that all implementations provide one.

Consider the following:

```
//Header file
using MyPredicate = std::function<bool(const MyValue &, const MyValue &)>;
void SortMyContainer(MyContainer &C, const MyPredicate &pred);

//Source file
void SortMyContainer(MyContainer &C, const MyPredicate &pred)
{
    std::sort(C.begin(), C.end(), pred);
}
```

A template parameter would be the preferred solution for `SortMyContainer`, but let us assume that this is not possible or desirable for whatever reason. `SortMyContainer` does not need to store `pred` beyond its own call. And yet, `pred` may well allocate memory if the functor given to it is of some non-trivial size.

`function` allocates memory because it needs something to copy/move into; `function` takes ownership of the callable it is given. But `SortMyContainer` does not need to own the callable; it's just referencing it. So using `function` here is overkill; it may be efficient, but it may not.

There is no standard library function type that merely references a callable. So an alternate solution will have to be found, or you can choose to live with the overhead.

Also, `function` has no effective means to control where the memory allocations for the object come from. Yes, it has constructors that take an `allocator`, but [many implementations do not implement them correctly... or even at all][2].

Version ≥ C++17

The `function` constructors that take an `allocator` no longer are part of the type. Therefore, there is no way to manage the allocation.

Calling a function is also slower than calling the contents directly. Since any `function` instance could hold any callable, the call through a function must be indirect. The overhead of calling `function` is on the order of a virtual function call.

第53章：std::forward_list

std::forward_list 是一种容器，支持在容器中的任意位置快速插入和删除元素。不支持快速随机访问。它被实现为单链表，基本上与其在C语言中的实现没有任何额外开销。与std::list相比，当不需要双向迭代时，该容器提供了更节省空间的存储。

第53.1节：示例

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

template<typename T>
std::ostream& operator<<(std::ostream& s, const std::forward_list<T>& v) {
    s.put('[');
    char comma[3] = {"\0", ' ', '\0'};
    for (const auto& e : v) {
        s <<逗号 << e;
       逗号[0] = ',';
    }
    return s << ']';
}

int main()
{
    // c++11 初始化列表语法：
    std::forward_list<std::string> words1 {"the", "frogurt", "is", "also", "cursed"};    std::cout << "words1: "
    << words1 << ";

    // words2 == words1
    std::forward_list<std::string> words2(words1.begin(), words1.end());    std::cout << "wor
    ds2: " << words2 << ";

    // words3 == words1
    std::forward_list<std::string> words3(words1);    std::cout <<
    "words3: " << words3 << ";

    // words4 是 {"Mo", "Mo", "Mo", "Mo", "Mo"}    st
    d::forward_list<std::string> words4(5, "Mo");    std::cout <<
    "words4: " << words4 << ";
}
```

输出：

```
words1: [the, frogurt, is, also, cursed]
words2: [the, frogurt, is, also, cursed]
words3: [the, frogurt, is, also, cursed]
words4: [Mo, Mo, Mo, Mo, Mo]
```

第53.2节：方法

方法名称	定义
operator=	将值赋给容器
assign	将值赋给容器
get_allocator	返回关联的分配器

Chapter 53: std::forward_list

std::forward_list is a container that supports fast insertion and removal of elements from anywhere in the container. Fast random access is not supported. It is implemented as a singly-linked list and essentially does not have any overhead compared to its implementation in C. Compared to std::list this container provides more space efficient storage when bidirectional iteration is not needed.

Section 53.1: Example

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

template<typename T>
std::ostream& operator<<(std::ostream& s, const std::forward_list<T>& v) {
    s.put('[');
    char comma[3] = {"\0", ' ', '\0'};
    for (const auto& e : v) {
        s << comma << e;
        comma[0] = ',';
    }
    return s << ']';
}

int main()
{
    // c++11 initializer list syntax:
    std::forward_list<std::string> words1 {"the", "frogurt", "is", "also", "cursed"};
    std::cout << "words1: " << words1 << '\n';

    // words2 == words1
    std::forward_list<std::string> words2(words1.begin(), words1.end());
    std::cout << "words2: " << words2 << '\n';

    // words3 == words1
    std::forward_list<std::string> words3(words1);
    std::cout << "words3: " << words3 << '\n';

    // words4 is {"Mo", "Mo", "Mo", "Mo", "Mo"}
    std::forward_list<std::string> words4(5, "Mo");
    std::cout << "words4: " << words4 << '\n';
}
```

Output:

```
words1: [the, frogurt, is, also, cursed]
words2: [the, frogurt, is, also, cursed]
words3: [the, frogurt, is, also, cursed]
words4: [Mo, Mo, Mo, Mo, Mo]
```

Section 53.2: Methods

Method name	Definition
operator=	assigns values to the container
assign	assigns values to the container
get_allocator	returns the associated allocator

元素访问

前面 访问第一个元素

迭代器

before_begin 返回指向开始前元素的迭代器

cbefore_begin 返回指向开始前元素的常量迭代器

begin 返回指向开始的迭代器

cbegin 返回指向开始的常量迭代器

end 返回指向末尾的迭代器

cend 返回指向末尾的迭代器

容量

空 检查容器是否为空

最大大小 返回可能的最大元素数量

修改器

清除 清空内容

插入后 在某个元素之后插入元素

emplace_after 在某个元素之后原地构造元素

erase_after 删除某个元素之后的元素

push_front 在开头插入元素

emplace_front 在开头原地构造元素

pop_front 删除第一个元素

resize 更改存储的元素数量

swap 交换内容

操作

合并 合并两个已排序的列表

splice_after 从另一个 forward_list 移动元素

移除 移除满足特定条件的元素

remove_if 移除满足特定条件的元素

反转 反转元素的顺序

唯一的 移除连续重复元素

排序 对元素进行排序

Element access

front access the first element

Iterators

before_begin returns an iterator to the element before beginning

cbefore_begin returns a constant iterator to the element before beginning

begin returns an iterator to the beginning

cbegin returns a const iterator to the beginning

end returns an iterator to the end

cend returns a iterator to the end

Capacity

empty checks whether the container is empty

max_size returns the maximum possible number of elements

Modifiers

clear clears the contents

insert_after inserts elements after an element

emplace_after constructs elements in-place after an element

erase_after erases an element after an element

push_front inserts an element to the beginning

emplace_front constructs an element in-place at the beginning

pop_front removes the first element

resize changes the number of elements stored

swap swaps the contents

Operations

merge merges two sorted lists

splice_after moves elements from another forward_list

remove removes elements satisfying specific criteria

remove_if removes elements satisfying specific criteria

reverse reverses the order of the elements

unique removes consecutive duplicate elements

sort sorts the elements

第54章：std::pair

第54.1节：比较运算符

这些运算符的参数是lhs和rhs

- operator== 测试lhs和rhs对中的两个元素是否相等。如果lhs.first == rhs.first且lhs.second==rhs.second，则返回值为true，否则为false

```
std::pair<int, int> p1 = std::make_pair(1, 2);
std::pair<int, int> p2 = std::make_pair(2, 2);
```

```
if (p1 == p2)
std::cout << "equals";
else
```

std::cout << "not equal"//语句将显示此内容，因为它们不相同

- operator!= 测试 lhs 和 rhs 对中的任意元素是否不相等。如果 lhs.first != rhs.first 或 lhs.second != rhs.second 中任一成立，则返回 true，否则返回 false。

- operator< 测试 lhs.first 是否小于 rhs.first，若是则返回 true。否则，如果 rhs.first 小于 lhs.first，则返回 false。否则，如果 lhs.second 小于 rhs.second 返回 true，否则返回 false。

- operator<= 返回 !(rhs<lhs)

- operator> 返回 rhs<lhs

- operator>= 返回 !(lhs<rhs)

另一个使用成对容器的示例。它使用了 operator<，因为需要对容器进行排序。

```
#include <iostream>
#include <utility>
#include <vector>
#include <algorithm>
#include <string>

int main()
{
    std::vector<std::pair<int, std::string>> v = { {2, "baz"}, {2, "bar"}, {1, "foo"} };
    std::sort(v.begin(), v.end());

    for(const auto& p: v) {
        std::cout << "(" << p.first << ", " << p.second << " ) ";
        //输出: (1,foo) (2,bar) (2,baz)
    }
}
```

第54.2节：创建pair并访问元素

pair允许我们将两个对象视为一个对象。pair可以借助模板函数轻松构造 std::make_pair。

另一种方法是先创建pair，之后再赋值其元素（first和second）。

Chapter 54: std::pair

Section 54.1: Compare operators

Parameters of these operators are lhs and rhs

- operator== tests if both elements on lhs and rhs pair are equal. The return value is `true` if both `lhs.first == rhs.first` AND `lhs.second == rhs.second`, otherwise `false`

```
std::pair<int, int> p1 = std::make_pair(1, 2);
std::pair<int, int> p2 = std::make_pair(2, 2);
```

```
if (p1 == p2)
    std::cout << "equals";
else
```

std::cout << "not equal"//statement will show this, because they are not identical

- operator!= tests if any elements on lhs and rhs pair are not equal. The return value is `true` if either `lhs.first != rhs.first` OR `lhs.second != rhs.second`, otherwise return `false`.

- operator< tests if `lhs.first < rhs.first`, returns `true`. Otherwise, if `rhs.first < lhs.first` returns `false`. Otherwise, if `lhs.second < rhs.second` returns `true`, otherwise, returns `false`.

- operator<= returns !(rhs<lhs)

- operator> returns rhs<lhs

- operator>= returns !(lhs<rhs)

Another example with containers of pairs. It uses operator< because it needs to sort container.

```
#include <iostream>
#include <utility>
#include <vector>
#include <algorithm>
#include <string>

int main()
{
    std::vector<std::pair<int, std::string>> v = { {2, "baz"}, {2, "bar"}, {1, "foo"} };
    std::sort(v.begin(), v.end());

    for(const auto& p: v) {
        std::cout << "(" << p.first << ", " << p.second << " ) ";
        //output: (1,foo) (2,bar) (2,baz)
    }
}
```

Section 54.2: Creating a Pair and accessing the elements

Pair allows us to treat two objects as one object. Pairs can be easily constructed with the help of template function `std::make_pair`.

Alternative way is to create pair and assign its elements (first and second) later.

```

#include <iostream>
#include <utility>

int main()
{
std::pair<int,int> p = std::make_pair(1,2); //创建pair
    std::cout << p.first << " " << p.second << std::endl; //访问元素

//我们也可以先创建pair，之后再赋值元素
std::pair<int,int> p1;
p1.first = 3;
p1.second = 4;
std::cout << p1.first << " " << p1.second << std::endl;

//我们也可以使用构造函数创建一个pair
std::pair<int,int> p2 = std::pair<int,int>(5, 6);
std::cout << p2.first << " " << p2.second << std::endl;

return 0;
}

```

```

#include <iostream>
#include <utility>

int main()
{
    std::pair<int,int> p = std::make_pair(1,2); //Creating the pair
    std::cout << p.first << " " << p.second << std::endl; //Accessing the elements

//We can also create a pair and assign the elements later
std::pair<int,int> p1;
p1.first = 3;
p1.second = 4;
std::cout << p1.first << " " << p1.second << std::endl;

//We can also create a pair using a constructor
std::pair<int,int> p2 = std::pair<int,int>(5, 6);
std::cout << p2.first << " " << p2.second << std::endl;

return 0;
}

```

第55章：std::atomics

第55.1节：原子类型

std::atomic 模板的每个实例化和完全特化都定义了一个原子类型。如果一个线程写入一个原子对象，而另一个线程从中读取，则行为是定义良好的（有关数据竞争的详细信息，请参见内存模型）

此外，对原子对象的访问可能会建立线程间同步，并按照 std::memory_order 指定的顺序对非原子内存访问进行排序。

std::atomic 可以用任何 TriviallyCopyable 类型 T 实例化。 std::atomic 既不可复制也不可移动。

标准库为以下类型提供了 std::atomic 模板的特化：

1. 为类型bool及其typedef名称定义了一个完全特化，该特化被视为非特化处理

std::atomic<T> 除了具有标准布局、平凡的默认构造函数、平凡的析构函数外，还支持聚合初始化语法：

类型定义名称 完全特化

std::atomic_bool std::atomic<bool>

- 2) 对整型的完全特化和类型定义如下：

类型定义名称	完全特化
std::atomic_char	std::atomic<char>
std::atomic_char	std::atomic<char>
std::atomic_schar	std::atomic<signed char>
std::atomic_uchar	std::atomic<unsigned char>
std::atomic_short	std::atomic<short>
std::atomic_ushort	std::atomic<unsigned short>
std::atomic_int	std::atomic<int>
std::atomic_uint	std::atomic<unsigned int>
std::atomic_long	std::atomic<long>
std::atomic_ulong	std::atomic<unsigned long>
std::atomic_llong	std::atomic<long long>
std::atomic_ullong	std::atomic<unsigned long long>
std::atomic_char16_t	std::atomic<char16_t>
std::atomic_char32_t	std::atomic<char32_t>
std::atomic_wchar_t	std::atomic<wchar_t>
std::atomic_int8_t	std::atomic<std::int8_t>
std::atomic_uint8_t	std::atomic<std::uint8_t>
std::atomic_int16_t	std::atomic<std::int16_t>
std::atomic_uint16_t	std::atomic<std::uint16_t>
std::atomic_int32_t	std::atomic<std::int32_t>
std::atomic_uint32_t	std::atomic<std::uint32_t>
std::atomic_int64_t	std::atomic<std::int64_t>
std::atomic_uint64_t	std::atomic<std::uint64_t>
std::atomic_int_least8_t	std::atomic<std::int_least8_t>
std::atomic_uint_least8_t	std::atomic<std::uint_least8_t>

Chapter 55: std::atomics

Section 55.1: atomic types

Each instantiation and full specialization of the std::atomic template defines an atomic type. If one thread writes to an atomic object while another thread reads from it, the behavior is well-defined (see memory model for details on data races)

In addition, accesses to atomic objects may establish inter-thread synchronization and order non-atomic memory accesses as specified by std::memory_order.

std::atomic may be instantiated with any TriviallyCopyable type T. std::atomic is neither copyable nor movable.

The standard library provides specializations of the std::atomic template for the following types:

1. One full specialization for the type bool and its typedef name is defined that is treated as a non-specialized std::atomic<T> except that it has standard layout, trivial default constructor, trivial destructors, and supports aggregate initialization syntax:

TypeDef name	Full specialization
std::atomic_bool	std::atomic<bool>

2) Full specializations and typedefs for integral types, as follows:

TypeDef name	Full specialization
std::atomic_char	std::atomic<char>
std::atomic_char	std::atomic<char>
std::atomic_schar	std::atomic<signed char>
std::atomic_uchar	std::atomic<unsigned char>
std::atomic_short	std::atomic<short>
std::atomic_ushort	std::atomic<unsigned short>
std::atomic_int	std::atomic<int>
std::atomic_uint	std::atomic<unsigned int>
std::atomic_long	std::atomic<long>
std::atomic_ulong	std::atomic<unsigned long>
std::atomic_llong	std::atomic<long long>
std::atomic_ullong	std::atomic<unsigned long long>
std::atomic_char16_t	std::atomic<char16_t>
std::atomic_char32_t	std::atomic<char32_t>
std::atomic_wchar_t	std::atomic<wchar_t>
std::atomic_int8_t	std::atomic<std::int8_t>
std::atomic_uint8_t	std::atomic<std::uint8_t>
std::atomic_int16_t	std::atomic<std::int16_t>
std::atomic_uint16_t	std::atomic<std::uint16_t>
std::atomic_int32_t	std::atomic<std::int32_t>
std::atomic_uint32_t	std::atomic<std::uint32_t>
std::atomic_int64_t	std::atomic<std::int64_t>
std::atomic_uint64_t	std::atomic<std::uint64_t>
std::atomic_int_least8_t	std::atomic<std::int_least8_t>
std::atomic_uint_least8_t	std::atomic<std::uint_least8_t>

```

std::atomic_int_least16_t std::atomic<std::int_least16_t>
std::atomic_uint_least16_t std::atomic<std::uint_least16_t>
std::atomic_int_least32_t std::atomic<std::int_least32_t>
std::atomic_uint_least32_t std::atomic<std::uint_least32_t>
std::atomic_int_least64_t std::atomic<std::int_least64_t>
std::atomic_uint_least64_t std::atomic<std::uint_least64_t>
std::atomic_int_fast8_t   std::atomic<std::int_fast8_t>
std::atomic_uint_fast8_t std::atomic<std::uint_fast8_t>
std::atomic_int_fast16_t std::atomic<std::int_fast16_t>
std::atomic_uint_fast16_t std::atomic<std::uint_fast16_t>
std::atomic_int_fast32_t std::atomic<std::int_fast32_t>
std::atomic_uint_fast32_t std::atomic<std::uint_fast32_t>
std::atomic_int_fast64_t std::atomic<std::int_fast64_t>
std::atomic_uint_fast64_t std::atomic<std::uint_fast64_t>
std::atomic_intptr_t     std::atomic<std::intptr_t>
std::atomic_uintptr_t   std::atomic<std::uintptr_t>
std::atomic_size_t       std::atomic<std::size_t>
std::atomic_ptrdiff_t   std::atomic<std::ptrdiff_t>
std::atomic_intmax_t    std::atomic<std::intmax_t>
std::atomic_uintmax_t   std::atomic<std::uintmax_t>

```

使用 std::atomic_int 的简单示例

```

#include <iostream>          // std::cout
#include <atomic>           // std::atomic, std::memory_order_relaxed
#include <thread>           // std::thread

std::atomic_int foo (0);

void set_foo(int x) {
    foo.store(x, std::memory_order_relaxed); // 原子地设置值
}

void print_foo() {
    int x;
    do {
        x = foo.load(std::memory_order_relaxed); // 原子地获取值
    } while (x==0);
    std::cout << "foo: " << x << "\n";
}

int main ()
{
    std::thread first (print_foo);
    std::thread second (set_foo, 10);
    first.join();
    //second.join();
    return 0;
}
//output: foo: 10

```

```

std::atomic_int_least16_t std::atomic<std::int_least16_t>
std::atomic_uint_least16_t std::atomic<std::uint_least16_t>
std::atomic_int_least32_t std::atomic<std::int_least32_t>
std::atomic_uint_least32_t std::atomic<std::uint_least32_t>
std::atomic_int_least64_t std::atomic<std::int_least64_t>
std::atomic_uint_least64_t std::atomic<std::uint_least64_t>
std::atomic_int_fast8_t   std::atomic<std::int_fast8_t>
std::atomic_uint_fast8_t std::atomic<std::uint_fast8_t>
std::atomic_int_fast16_t std::atomic<std::int_fast16_t>
std::atomic_uint_fast16_t std::atomic<std::uint_fast16_t>
std::atomic_int_fast32_t std::atomic<std::int_fast32_t>
std::atomic_uint_fast32_t std::atomic<std::uint_fast32_t>
std::atomic_int_fast64_t std::atomic<std::int_fast64_t>
std::atomic_uint_fast64_t std::atomic<std::uint_fast64_t>
std::atomic_intptr_t     std::atomic<std::intptr_t>
std::atomic_uintptr_t   std::atomic<std::uintptr_t>
std::atomic_size_t       std::atomic<std::size_t>
std::atomic_ptrdiff_t   std::atomic<std::ptrdiff_t>
std::atomic_intmax_t    std::atomic<std::intmax_t>
std::atomic_uintmax_t   std::atomic<std::uintmax_t>

```

Simple example of using std::atomic_int

```

#include <iostream>          // std::cout
#include <atomic>           // std::atomic, std::memory_order_relaxed
#include <thread>           // std::thread

std::atomic_int foo (0);

void set_foo(int x) {
    foo.store(x, std::memory_order_relaxed); // set value atomically
}

void print_foo() {
    int x;
    do {
        x = foo.load(std::memory_order_relaxed); // get value atomically
    } while (x==0);
    std::cout << "foo: " << x << '\n';
}

int main ()
{
    std::thread first (print_foo);
    std::thread second (set_foo, 10);
    first.join();
    //second.join();
    return 0;
}
//output: foo: 10

```

第56章：std::variant

第56.1节：创建伪方法指针

这是一个高级示例。

你可以使用variant实现轻量级类型擦除。

```
模板<类 F>
结构体 伪方法 {
    F f;
    // 启用 C++17 类型推断：
    伪方法( F&& fin ):f(std::move(fin)) {}

    // Koenig 查找 operator->*, 由于这是一个伪方法，因此是合适的：
    模板<类 Variant> // 也许添加 SFINAE 测试以确保左侧实际上是一个 variant。
    友元 decltype(auto) operator->*( Variant&& var, 伪方法 const& method ) {
        // var->*method 返回一个完美转发函数调用的 lambda,
        // 基本上表现得像一个方法指针：
        返回 [&](auto&&...args)->decltype(auto) {
            // 使用 visit 获取 variant 的类型：
            返回 std::visit(
                [&](auto&& self)->decltype(auto) {
                    // decltype(x)(x) 是 lambda 中的完美转发：
                    返回 方法. f( decltype(self)(self), decltype(args)(args)... );
                },
                std::forward<Var>(var)
            );
        };
    }
};
```

这会创建一个类型，该类型重载了左侧为Variant的operator->*运算符。

```
// C++17 类型推导，用于在此处查找 `print` 的模板参数。
// 一个伪方法 lambda 应该将 `self` 作为第一个参数，然后是
// 其余参数，最后调用该操作：
伪方法 print = [](auto&& self, auto&&...args)->decltype(auto) {
    返回 decltype(self)(self).print( decltype(args)(args)... );
};
```

现在如果我们有两个各自带有print方法的类型：

```
结构体 A {
    void print( std::ostream& os ) const {
        os << "A";
    }
};
结构体 B {
    void print( std::ostream& os ) const {
        os << "B";
    }
};
```

请注意它们是无关的类型。我们可以：

```
std::variant<A,B> var = A{};
```

Chapter 56: std::variant

Section 56.1: Create pseudo-method pointers

This is an advanced example.

You can use variant for light weight type erasure.

```
template<class F>
struct pseudo_method {
    F f;
    // enable C++17 class type deduction:
    pseudo_method( F&& fin ):f(std::move(fin)) {}

    // Koenig lookup operator->*, as this is a pseudo-method it is appropriate:
    template<class Variant> // maybe add SFINAE test that LHS is actually a variant.
    友元 decltype(auto) operator->*( Variant&& var, pseudo_method const& method ) {
        // var->*method returns a lambda that perfect forwards a function call,
        // behaving like a method pointer basically:
        返回 [&](auto&&...args)->decltype(auto) {
            // use visit to get the type of the variant:
            返回 std::visit(
                [&](auto&& self)->decltype(auto) {
                    // decltype(x)(x) is perfect forwarding in a lambda:
                    返回 method.f( decltype(self)(self), decltype(args)(args)... );
                },
                std::forward<Var>(var)
            );
        };
    }
};
```

this creates a type that overloads operator->* with a Variant on the left hand side.

```
// C++17 class type deduction to find template argument of `print` here.
// a pseudo-method lambda should take `self` as its first argument, then
// the rest of the arguments afterwards, and invoke the action:
pseudo_method print = [](auto&& self, auto&&...args)->decltype(auto) {
    返回 decltype(self)(self).print( decltype(args)(args)... );
};
```

Now if we have 2 types each with a print method:

```
struct A {
    void print( std::ostream& os ) const {
        os << "A";
    }
};
struct B {
    void print( std::ostream& os ) const {
        os << "B";
    }
};
```

note that they are unrelated types. We can:

```
std::variant<A,B> var = A{};
```

```
(var->*print)(std::cout);
```

它会直接将调用分派给A::print(std::cout)。如果我们用B{}初始化var，则会分派给B::print(std::cout)。

如果我们创建一个新类型C：

```
struct C {};
```

那么：

```
std::variant<A,B,C> var = A{};
(var->*print)(std::cout);
```

将无法编译，因为没有C.print(std::cout)方法。

扩展上述内容将允许检测并使用自由函数print，可能结合在print伪方法中使用if constexpr。

当前的实际示例使用boost::variant代替std::variant。

第56.2节：基本的std::variant用法

这将创建一个variant（带标签的联合体），可以存储int或string中的任意一种。

```
std::variant<int, std::string> var;
```

我们可以在其中存储任意一种类型：

```
var = "hello"s;
```

我们可以通过std::visit访问内容：

```
// 输出 "hello":
visit([](auto&&e) { std::cout << e << ";"}, var);
```

通过传入多态lambda或类似的函数对象。

如果我们确定知道它的类型，可以这样获取：

```
auto str = std::get<std::string>(var);
```

但如果获取错误会抛出异常。get_if：

```
auto* str = std::get_if<std::string>(&var);
```

如果猜错则返回 nullptr。

变体保证不进行动态内存分配（除了其包含的类型分配的内存）。变体中只存储一种类型，在极少数情况下（涉及赋值时的异常且没有安全的回退方式）变体可能变为空。

变体允许你安全高效地在一个变量中存储多种值类型。它们基本上是智能的、类型安全的

```
(var->*print)(std::cout);
```

and it will dispatch the call directly to A::print(std::cout) for us. If we instead initialized the var with B{}, it would dispatch to B::print(std::cout).

If we created a new type C:

```
struct C {};
```

then:

```
std::variant<A,B,C> var = A{};
(var->*print)(std::cout);
```

will fail to compile, because there is no C.print(std::cout) method.

Extending the above would permit free function prints to be detected and used, possibly with use of if constexpr within the print pseudo-method.

[live example](#) currently using boost::variant in place of std::variant.

Section 56.2: Basic std::variant use

This creates a variant (a tagged union) that can store either an int or a string.

```
std::variant< int, std::string > var;
```

We can store one of either type in it:

```
var = "hello"s;
```

And we can access the contents via std::visit:

```
// Prints "hello\n":
visit( [](auto&& e) {
    std::cout << e << '\n';
}, var );
```

by passing in a polymorphic lambda or similar function object.

If we are certain we know what type it is, we can get it:

```
auto str = std::get<std::string>(var);
```

but this will throw if we get it wrong. get_if:

```
auto* str = std::get_if<std::string>(&var);
```

returns nullptr if you guess wrong.

Variants guarantee no dynamic memory allocation (other than which is allocated by their contained types). Only one of the types in a variant is stored there, and in rare cases (involving exceptions while assigning and no safe way to back out) the variant can become empty.

Variants let you store multiple value types in one variable safely and efficiently. They are basically smart, type-safe

第56.3节：构造 `std::variant`

这不包括分配器。

```
struct A {};
struct B { B()=default; B(B const&)=default; B(int){}; };
struct C { C()=delete; C(int) {}; C(C const&)=default; };
struct D { D( std::initializer_list<int> ) {}; D(D const&)=default; D()=default; };

std::variant<A,B> var_ab0; // 包含一个 A()
std::variant<A,B> var_ab1 = 7; // 包含一个 B(7)
std::variant<A,B> var_ab2 = var_ab1; // 包含一个 B(7)
std::variant<A,B,C> var_abc0{ std::in_place_type<C>, 7 }; // 包含一个 C(7)
std::variant<C> var_c0; // 非法, C 没有默认构造函数
std::variant<A,D> var_ad0( std::in_place_type<D>, {1,3,3,4} ); // 包含 D{1,3,3,4}
std::variant<A,D> var_ad1( std::in_place_index<0> ); // 包含 A{}
std::variant<A,D> var_ad2( std::in_place_index<1>, {1,3,3,4} ); // 包含 D{1,3,3,4}
```

unions.

Section 56.3: Constructing a `std::variant`

This does not cover allocators.

```
struct A {};
struct B { B()=default; B(B const&)=default; B(int){}; };
struct C { C()=delete; C(int) {}; C(C const&)=default; };
struct D { D( std::initializer_list<int> ) {}; D(D const&)=default; D()=default; };

std::variant<A,B> var_ab0; // contains a A()
std::variant<A,B> var_ab1 = 7; // contains a B(7)
std::variant<A,B> var_ab2 = var_ab1; // contains a B(7)
std::variant<A,B,C> var_abc0{ std::in_place_type<C>, 7 }; // contains a C(7)
std::variant<C> var_c0; // illegal, no default ctor for C
std::variant<A,D> var_ad0( std::in_place_type<D>, {1,3,3,4} ); // contains D{1,3,3,4}
std::variant<A,D> var_ad1( std::in_place_index<0> ); // contains A{}
std::variant<A,D> var_ad2( std::in_place_index<1>, {1,3,3,4} ); // contains D{1,3,3,4}
```

第57章：std::iomanip

第57.1节：std::setprecision

当在表达式中使用 `out << setprecision(n)` 或 `in >> setprecision(n)` 时，设置流 `out` 或 `in` 的精度参数为恰好 `n`。该函数的参数是整数，表示新的精度值。

示例：

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <limits>
int main()
{
    const long double pi = std::acos(-1.0);
    std::cout << "默认精度 (6): " << pi << "    << std::precision(10):    " <<
        std::setprecision(10) << pi << "    最大精度:    "
        << std::setprecision(std::numeric_limits<long double>::digits10 + 1) << pi << ";
}
//输出
//默认精度 (6): 3.14159
//std::precision(10): 3.141592654
//最大精度: 3.141592653589793239
```

第57.2节：std::setfill

当在表达式 `out << setfill(c)` 中使用时，会将流 `out` 的填充字符设置为 `c`。

注意：当前的填充字符可以通过 `std::ostream::fill` 获得。

示例：

```
#include <iostream>
#include <iomanip>
int main()
{
    std::cout << "默认填充: " << std::setw(10) << 42 << "    << "设置填充('*'): " << std
        ::setfill('*')
        << std::setw(10) << 42 << ";"
}
//输出:
//默认填充: 42
//设置填充字符('*'): *****42
```

第57.3节：std::setiosflags

当在表达式 `out << setiosflags(mask)` 或 `in >> setiosflags(mask)` 中使用时，设置流 `out` 或 `in` 的所有格式标志，如掩码所指定。

所有 `std::ios_base::fmtflags` 的列表：

- dec - 对整数输入输出使用十进制
- oct - 对整数输入输出使用八进制

Chapter 57: std::iomanip

Section 57.1: std::setprecision

When used in an expression `out << setprecision(n)` or `in >> setprecision(n)`, sets the precision parameter of the stream `out` or `in` to exactly `n`. Parameter of this function is integer, which is new value for precision.

Example:

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <limits>
int main()
{
    const long double pi = std::acos(-1.0);
    std::cout << "default precision (6): " << pi << '\n'
        << "std::precision(10):    " << std::setprecision(10) << pi << '\n'
        << "max precision:    "
        << std::setprecision(std::numeric_limits<long double>::digits10 + 1)
        << pi << '\n';
}
//Output
//default precision (6): 3.14159
//std::precision(10): 3.141592654
//max precision: 3.141592653589793239
```

Section 57.2: std::setfill

When used in an expression `out << setfill(c)` sets the fill character of the stream `out` to `c`.

Note: The current fill character may be obtained with `std::ostream::fill`.

Example:

```
#include <iostream>
#include <iomanip>
int main()
{
    std::cout << "default fill: " << std::setw(10) << 42 << '\n'
        << "setfill('*'): " << std::setfill('*')
        << std::setw(10) << 42 << '\n';
}
//output:
//default fill: 42
//setfill('*'): *****42
```

Section 57.3: std::setiosflags

When used in an expression `out << setiosflags(mask)` or `in >> setiosflags(mask)`, sets all format flags of the stream `out` or `in` as specified by the mask.

List of all `std::ios_base::fmtflags`:

- dec - use decimal base for integer I/O
- oct - use octal base for integer I/O

- hex - 对整数输入输出使用十六进制
- basefield - dec|oct|hex|0 适用于掩码操作
- left - 左对齐 (在右侧添加填充字符)
- right - 右对齐 (在左侧添加填充字符)
- internal - 内部对齐 (在指定的内部位置添加填充字符)
- adjustfield - left|right|internal。适用于掩码操作
- scientific - 使用科学计数法生成浮点类型，或与fixed结合时使用十六进制表示法
- fixed - 使用定点表示法生成浮点类型，或与scientific结合时使用十六进制表示法
- floatfield - scientific|fixed|(scientific|fixed)|0。适用于掩码操作
- boolalpha - 以字母数字格式插入和提取bool类型
- showbase - 为整数输出生成表示数字进制的前缀，货币输入输出中需要货币符号
- showpoint - 无条件为浮点数输出生成小数点字符
- showpos - 为非负数值输出生成+字符
- skipws - 在某些输入操作前跳过前导空白
- unitbuf 每次输出操作后刷新输出缓冲区
- uppercase - 在某些输出操作中将特定小写字母替换为大写字母

操纵器示例：

```
#include <iostream>
#include <string>
#include<iomanip>
int main()
{
    int l_iTemp = 47;
    std::cout << std::resetiosflags(std::ios_base::basefield);
    std::cout << std::setiosflags( std::ios_base::oct) << l_iTemp << std::endl;
    //输出: 57
    std::cout << std::resetiosflags(std::ios_base::basefield);
    std::cout << std::setiosflags( std::ios_base::hex) << l_iTemp << std::endl;
    //输出: 2f
    std::cout << std::setiosflags( std::ios_base::uppercase) << l_iTemp << std::endl;
    //输出 2F
    std::cout << std::setfill('0') << std::setw(12);
    std::cout << std::resetiosflags(std::ios_base::uppercase);
    std::cout << std::setiosflags( std::ios_base::right) << l_iTemp << std::endl;
    //输出: 00000000002f

    std::cout << std::resetiosflags(std::ios_base::basefield | std::ios_base::adjustfield);
    std::cout << std::setfill('.') << std::setw(10);
    std::cout << std::setiosflags( std::ios_base::left) << l_iTemp << std::endl;
    //输出: 47.......

    std::cout << std::resetiosflags(std::ios_base::adjustfield) << std::setfill('#');
    std::cout << std::setiosflags(std::ios_base::internal | std::ios_base::showpos);
    std::cout << std::setw(10) << l_iTemp << std::endl;
    //输出 +#####47

    double l_dTemp = -1.2;
    double pi = 3.14159265359;
    std::cout << pi << " " << l_dTemp << std::endl;
    //输出 +3.14159 -1.2
    std::cout << std::setiosflags(std::ios_base::showpoint) << l_dTemp << std::endl;
    //输出 -1.20000
    std::cout << std::setiosflags(std::ios_base::scientific) << pi << std::endl;
    //输出: +3.141593e+00
}
```

- hex - use hexadecimal base for integer I/O
- basefield - dec|oct|hex|0 useful for masking operations
- left - left adjustment(add fill characters to the right)
- right - right adjustment (adds fill characters to the left)
- internal - internal adjustment (adds fill characters to the internal designated point)
- adjustfield - left|right|internal. Useful for masking operations
- scientific - generate floating point types using scientific notation, or hex notation if combined with fixed
- fixed - generate floating point types using fixed notation, or hex notation if combined with scientific
- floatfield - scientific|fixed|(scientific|fixed)|0. Useful for masking operations
- boolalpha - insert and extract bool type in alphanumeric format
- showbase - generate a prefix indicating the numeric base for integer output, require the currency indicator in monetary I/O
- showpoint - generate a decimal-point character unconditionally for floating-point number output
- showpos - generate a + character for non-negative numeric output
- skipws - skip leading whitespace before certain input operations
- unitbuf flush the output after each output operation
- uppercase - replace certain lowercase letters with their uppercase equivalents in certain output operations

Example of manipulators:

```
#include <iostream>
#include <string>
#include<iomanip>
int main()
{
    int l_iTemp = 47;
    std::cout << std::resetiosflags(std::ios_base::basefield);
    std::cout << std::setiosflags( std::ios_base::oct) << l_iTemp << std::endl;
    //output: 57
    std::cout << std::resetiosflags(std::ios_base::basefield);
    std::cout << std::setiosflags( std::ios_base::hex) << l_iTemp << std::endl;
    //output: 2f
    std::cout << std::setiosflags( std::ios_base::uppercase) << l_iTemp << std::endl;
    //output 2F
    std::cout << std::setfill('0') << std::setw(12);
    std::cout << std::resetiosflags(std::ios_base::uppercase);
    std::cout << std::setiosflags( std::ios_base::right) << l_iTemp << std::endl;
    //output: 00000000002f

    std::cout << std::resetiosflags(std::ios_base::basefield | std::ios_base::adjustfield);
    std::cout << std::setfill('.') << std::setw(10);
    std::cout << std::setiosflags( std::ios_base::left) << l_iTemp << std::endl;
    //output: 47.......

    std::cout << std::resetiosflags(std::ios_base::adjustfield) << std::setfill('#');
    std::cout << std::setiosflags(std::ios_base::internal | std::ios_base::showpos);
    std::cout << std::setw(10) << l_iTemp << std::endl;
    //output +#####47

    double l_dTemp = -1.2;
    double pi = 3.14159265359;
    std::cout << pi << " " << l_dTemp << std::endl;
    //output +3.14159 -1.2
    std::cout << std::setiosflags(std::ios_base::showpoint) << l_dTemp << std::endl;
    //output -1.20000
    std::cout << std::setiosflags(std::ios_base::scientific) << pi << std::endl;
    //output: +3.141593e+00
}
```

```

std::cout<<std::resetiosflags(std::ios_base::floatfield);
std::cout<<setiosflags(std::ios_base::fixed)<<pi<<std::endl;
//输出: +3.141593
bool b = true;
std::cout<<std::setiosflags(std::ios_base::unitbuf|std::ios_base::boolalpha)<<b;
//输出: true
return 0;
}

```

第57.4节 : std::setw

```

int val = 10;
// val 将被打印在输出控制台的最左端：
std::cout << val << std::endl;
// val 将从字段的右端开始，在长度为10的输出字段中打印：
std::cout << std::setw(10) << val << std::endl;

```

输出结果为：

```

10
 10
1234567890

```

(最后一行用于帮助查看字符偏移量)。

有时我们需要设置输出字段的宽度，通常是在需要获得某种结构化且规范的布局时。可以使用**std::setw**（属于**std::iomanip**）来实现。

std::setw的语法是：

```
std::setw(int n)
```

其中 **n** 是要设置的输出字段的长度

```

std::cout<<std::resetiosflags(std::ios_base::floatfield);
std::cout<<setiosflags(std::ios_base::fixed)<<pi<<std::endl;
//output: +3.141593
bool b = true;
std::cout<<std::setiosflags(std::ios_base::unitbuf|std::ios_base::boolalpha)<<b;
//output: true
return 0;
}

```

Section 57.4: std::setw

```

int val = 10;
// val will be printed to the extreme left end of the output console:
std::cout << val << std::endl;
// val will be printed in an output field of length 10 starting from right end of the field:
std::cout << std::setw(10) << val << std::endl;

```

This outputs:

```

10
 10
1234567890

```

(where the last line is there to aid in seeing the character offsets).

Sometimes we need to set the width of the output field, usually when we need to get the output in some structured and proper layout. That can be done using **std::setw** of **std::iomanip**.

The syntax for **std::setw** is:

```
std::setw(int n)
```

where **n** is the length of the output field to be set

第58章：std::any

第58.1节：基本用法

```
std::any an_object{ std::string("hello world") };
if (an_object.has_value()) {
    std::cout << std::any_cast<std::string>(an_object) << ";"

try {
    std::any_cast<int>(an_object); } catch(std
::bad_any_cast&) { std::cout <<
"Wrong type";}

std::any_cast<std::string&>(an_object) = "42";std::cout <<
std::any_cast<std::string>(an_object) << ";"
```

输出

```
hello world
Wrong type
42
```

Chapter 58: std::any

Section 58.1: Basic usage

```
std::any an_object{ std::string("hello world") };
if (an_object.has_value()) {
    std::cout << std::any_cast<std::string>(an_object) << '\n';
}

try {
    std::any_cast<int>(an_object);
} catch(std::bad_any_cast&) {
    std::cout << "Wrong type\n";
}

std::any_cast<std::string&>(an_object) = "42";
std::cout << std::any_cast<std::string>(an_object) << '\n';
```

Output

```
hello world
Wrong type
42
```

第59章：std::set 和 std::multiset

set 是一种元素有序且唯一的容器类型。multiset 类似，但在 multiset 中，多个元素可以具有相同的值。

第59.1节：更改 set 的默认排序方式

set 和 multiset 有默认的比较方法，但在某些情况下你可能需要重载它们。

假设我们在 set 中存储字符串值，但我们知道这些字符串只包含数字值。默认情况下，排序将是字典序字符串比较，因此顺序不会与数字排序匹配。如果你想应用等同于 int 值的排序，则需要一个函数对象来重载比较方法：

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

struct custom_compare final
{
    bool operator() (const std::string& left, const std::string& right) const
    {
        int nLeft = atoi(left.c_str());
        int nRight = atoi(right.c_str());
        return nLeft < nRight;
    }
};

int main ()
{
    std::set<std::string> sut({"1", "2", "5", "23", "6", "290"});

    std::cout << "### Default sort on std::set<std::string> :" << std::endl;
    for (auto &&data: sut)
        std::cout << data << std::endl;

    std::set<std::string, custom_compare> sut_custom({"1", "2", "5", "23", "6", "290"}, 
                                                    custom_compare{}); //< 比较对象可选
    // 因为它是默认可构造的。
}

std::cout << std::endl << "### 自定义排序的集合 :" << std::endl;
for (auto &&data : sut_custom)
    std::cout << data << std::endl;

auto compare_via_lambda = [] (auto &&lhs, auto &&rhs){ return lhs > rhs; };
using set_via_lambda = std::set<std::string, decltype(compare_via_lambda)>;
set_via_lambda sut_reverse_via_lambda({"1", "2", "5", "23", "6", "290"}, 
                                      compare_via_lambda);

std::cout << std::endl << "### Lambda 排序的集合 :" << std::endl;
for (auto &&data : sut_reverse_via_lambda)
    std::cout << data << std::endl;

    return 0;
}
```

输出将是：

Chapter 59: std::set and std::multiset

set is a type of container whose elements are sorted and unique. multiset is similar, but, in the case of multiset, multiple elements can have the same value.

Section 59.1: Changing the default sort of a set

set and multiset have default compare methods, but in some cases you may need to overload them.

Let's imagine we are storing string values in a set, but we know those strings contain only numeric values. By default the sort will be a lexicographical string comparison, so the order won't match the numerical sort. If you want to apply a sort equivalent to what you would have with `int` values, you need a functor to overload the compare method:

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

struct custom_compare final
{
    bool operator() (const std::string& left, const std::string& right) const
    {
        int nLeft = atoi(left.c_str());
        int nRight = atoi(right.c_str());
        return nLeft < nRight;
    }
};

int main ()
{
    std::set<std::string> sut({"1", "2", "5", "23", "6", "290"});

    std::cout << "### Default sort on std::set<std::string> :" << std::endl;
    for (auto &&data: sut)
        std::cout << data << std::endl;

    std::set<std::string, custom_compare> sut_custom {"1", "2", "5", "23", "6", "290"}, 
                                                    custom_compare{}); //< Compare object optional
    // as its default constructible.

    std::cout << std::endl << "### Custom sort on set :" << std::endl;
    for (auto &&data : sut_custom)
        std::cout << data << std::endl;

    auto compare_via_lambda = [] (auto &&lhs, auto &&rhs){ return lhs > rhs; };
    using set_via_lambda = std::set<std::string, decltype(compare_via_lambda)>;
    set_via_lambda sut_reverse_via_lambda {"1", "2", "5", "23", "6", "290"}, 
                                         compare_via_lambda);

    std::cout << std::endl << "### Lambda sort on set :" << std::endl;
    for (auto &&data : sut_reverse_via_lambda)
        std::cout << data << std::endl;

    return 0;
}
```

Output will be:

std::set<std::string> 的默认排序：

自定义排序的集合
1 2 3 2 9 0 5 6 #

Lambda 排序的集合
6 5 2 9 0 2 3 2 9 0

#

在其各自的上下文中都有用。

set添加比较操作，每种方

默认排序

这将使用键（第一个模板参数）的比较运算符。通常，键已经为std::less<T>函数提供了一个很好的默认值。除非该函数被特化，否则它使用对象的operator<。这在其他代码也尝试使用某种排序时尤其有用，因为这可以保证整个代码库的一致性。

以这种方式编写代码，将减少在键发生变化时更新代码的工作量，例如：一个包含两个成员的类变成包含三个成员的类。通过更新类中的operator<，所有相关的地方都会被更新。

正如你所料，使用默认排序是一个合理的默认选择。

自定义排序

通过带有比较运算符的对象添加自定义排序通常在默认比较不符合要求时使用。在上面的例子中，这是因为字符串指代的是整数。在其他情况下，当你想基于所指对象比较（智能）指针，或者因为你需要不同的比较约束（例如：按std::pair的first值比较）时，也常常使用自定义排序。

创建比较运算符时，应保证排序的稳定性。如果插入后比较运算符的结果发生变化，将导致未定义行为。作为良好实践，比较运算符应仅使用常量数据（const成员、const函数等）。

如上例所示，你经常会遇到没有成员的类作为比较运算符。这会导致默认构造函数和拷贝构造函数的生成。默认构造函数允许你在构造时省略实例，而拷贝构造函数是必需的，因为set会拷贝比较运算符的实例。

Lambda排序

Lambda是编写函数对象的简短方式。这使得比较运算符的代码行数减少，从而使

Default sort on std::set<std::string> :

1
2
23
290
5
6

Custom sort on set :

1
2
5
6
23
290

Lambda sort on set :

6
5
290
23
2
1

In the example above, one can find 3 different ways of adding compare operations to the std::set, each of them is useful in its own context.

Default sort

This will use the compare operator of the key (first template argument). Often, the key will already provide a good default for the std::less<T> function. Unless this function is specialized, it uses the operator< of the object. This is especially useful when other code also tries to use some ordering, as this allows consistency over the whole code base.

Writing the code this way, will reduce the effort to update your code when the key changes is API, like: a class containing 2 members which changes to a class containing 3 members. By updating the operator< in the class, all occurrences will get updated.

As you might expect, using the default sort is a reasonable default.

Custom sort

Adding a custom sort via an object with a compare operator is often used when the default comparison doesn't comply. In the example above this is because the strings are referring to integers. In other cases, it's often used when you want to compare (smart) pointers based upon the object they refer to or because you need different constraints for comparing (example: comparing std::pair by the value of first).

When creating a compare operator, this should be a stable sorting. If the result of the compare operator changes after insert, you will have undefined behavior. As a good practice, your compare operator should only use the constant data (const members, const functions ...).

As in the example above, you will often encounter classes without members as compare operators. This results in default constructors and copy constructors. The default constructor allows you to omit the instance at construction time and the copy constructor is required as the set takes a copy of the compare operator.

Lambda sort

Lambdas are a shorter way to write function objects. This allows writing the compare operator on less lines, making

使整体代码更易读。

使用lambda的缺点是每个lambda在编译时都会获得一个特定的类型，因此`decltype(lambda)`对于同一编译单元（cpp文件）的每次编译都会不同，且在多个编译单元（通过头文件包含时）中也是如此。因此，建议在头文件中使用函数对象作为比较操作符。

这种结构通常出现在函数的局部作用域内使用`std::set`时，而当作为函数参数或类成员使用时，优先使用函数对象。

其他排序选项

由于`std::set`的比较操作符是模板参数，所有可调用对象都可以用作比较操作符，上述示例只是特定情况。这些可调用对象唯一的限制是：

- 它们必须是可拷贝构造的
- 它们必须能以键类型的两个参数调用。（允许隐式转换，但不推荐，因为可能影响性能）

第59.2节：从集合中删除值

如果你只是想将你的set/multiset重置为空，最明显的方法是使用`clear`：

```
std::set<int> sut;
sut.insert(10);
sut.insert(15);
sut.insert(22);
sut.insert(3);
sut.clear(); // sut的大小为0
```

然后可以使用`erase`方法。它提供了一些看起来与插入相当的可能性：

```
std::set<int> sut;
std::set<int>::iterator it;

sut.insert(10);
sut.insert(15);
sut.insert(22);
sut.insert(3);
sut.insert(30);
sut.insert(33);
sut.insert(45);

// 基本删除
sut.erase(3);

// 使用迭代器
it = sut.find(22);
sut.erase(it);

// 删除一段范围的值
it = sut.find(33);
sut.erase(it, sut.end());

std::cout << std::endl << "测试集合包含：" << std::endl;
for (it = sut.begin(); it != sut.end(); ++it)
{
    std::cout << *it << std::endl;
```

the overall code more readable.

The disadvantage of the use of lambdas is that each lambda gets a specific type at compile time, so`decltype(lambda)` will be different for each compilation of the same compilation unit (cpp file) as over multiple compilation units (when included via header file). For this reason, its recommended to use function objects as compare operator when used within header files.

This construction is often encountered when a`std::set` is used within the local scope of a function instead, while the function object is preferred when used as function arguments or class members.

Other sort options

As the compare operator of`std::set` is a template argument, all callable objects can be used as compare operator and the examples above are only specific cases. The only restrictions these callable objects have are:

- They must be copy constructable
- They must be callable with 2 arguments of the type of the key. (implicit conversions are allowed, though not recommended as it can hurt performance)

Section 59.2: Deleting values from a set

The most obvious method, if you just want to reset your set/multiset to an empty one, is to use`clear`:

```
std::set<int> sut;
sut.insert(10);
sut.insert(15);
sut.insert(22);
sut.insert(3);
sut.clear(); // size of sut is 0
```

Then the`erase` method can be used. It offers some possibilities looking somewhat equivalent to the insertion:

```
std::set<int> sut;
std::set<int>::iterator it;

sut.insert(10);
sut.insert(15);
sut.insert(22);
sut.insert(3);
sut.insert(30);
sut.insert(33);
sut.insert(45);

// Basic deletion
sut.erase(3);

// Using iterator
it = sut.find(22);
sut.erase(it);

// Deleting a range of values
it = sut.find(33);
sut.erase(it, sut.end());

std::cout << std::endl << "Set under test contains:" << std::endl;
for (it = sut.begin(); it != sut.end(); ++it)
{
    std::cout << *it << std::endl;
```

}

输出将是：

测试的集合包含：

10

15

30

所有这些方法同样适用于多重集合。请注意，如果你请求从多重集合中删除一个元素，且该元素存在多次，**所有等价的值都会被删除**。

第59.3节：在集合中插入值

集合可以使用三种不同的插入方法。

- 首先，简单地插入值。此方法返回一个对，允许调用者检查插入是否真正发生。
- 其次，通过给出值将被插入位置的提示来插入。目的是在这种情况下优化插入时间，但知道值应该插入的位置并不常见。**在这种情况下要小心；给出提示的方式因编译器版本而异。**
- 最后，您可以通过给出起始指针和结束指针来插入一系列值。起始指针所指的值将被包含在插入中，结束指针所指的值则不包含在内。

```
#include <iostream>
#include <set>

int main ()
{
    std::set<int> sut;
    std::set<int>::iterator it;
    std::pair<std::set<int>::iterator, bool> ret;

    // 基本插入操作
    sut.insert(7);
    sut.insert(5);
    sut.insert(12);

    ret = sut.insert(23);
    if (ret.second==true)
        std::cout << "# 23 已被插入！" << std::endl;

    ret = sut.insert(23); // 由于是集合且23已存在，此次插入应失败
    if (ret.second==false)
        std::cout << "# 23 已经存在于集合中！" << std::endl;

    // 带提示的插入以优化性能
    it = sut.end();
    // 此情况针对 C++11 及以上版本进行了优化
}
```

}

Output will be:

Set under test contains:

10

15

30

All those methods also apply to multiset. Please note that if you ask to delete an element from a multiset, and it is present multiple times, **all the equivalent values will be deleted**.

Section 59.3: Inserting values in a set

Three different methods of insertion can be used with sets.

- First, a simple insert of the value. This method returns a pair allowing the caller to check whether the insert really occurred.
- Second, an insert by giving a hint of where the value will be inserted. The objective is to optimize the insertion time in such a case, but knowing where a value should be inserted is not the common case. **Be careful in that case; the way to give a hint differs with compiler versions.**
- Finally you can insert a range of values by giving a starting and an ending pointer. The starting one will be included in the insertion, the ending one is excluded.

```
#include <iostream>
#include <set>

int main ()
{
    std::set<int> sut;
    std::set<int>::iterator it;
    std::pair<std::set<int>::iterator, bool> ret;

    // Basic insert
    sut.insert(7);
    sut.insert(5);
    sut.insert(12);

    ret = sut.insert(23);
    if (ret.second==true)
        std::cout << "# 23 has been inserted!" << std::endl;

    ret = sut.insert(23); // since it's a set and 23 is already present in it, this insert should fail
    if (ret.second==false)
        std::cout << "# 23 already present in set!" << std::endl;

    // Insert with hint for optimization
    it = sut.end();
    // This case is optimized for C++11 and above
}
```

```

// 对于更早版本, 指向插入位置前面的元素
sut.insert(it, 30);

// 插入一系列值
std::set<int> sut2;
sut2.insert(20);
sut2.insert(30);
sut2.insert(45);
std::set<int>::iterator itStart = sut2.begin();
std::set<int>::iterator itEnd = sut2.end();

sut.insert (itStart, itEnd); // 第二个迭代器不包含在插入范围内

std::cout << std::endl << "测试集合包含：" << std::endl;
for (it = sut.begin(); it != sut.end(); ++it)
{
    std::cout << *it << std::endl;
}

return 0;
}

```

输出将是：

23 已插入！

23 已存在于集合中！

测试的集合包含:

5

7

12

20

23

30

```

// For earlier version, point to the element preceding your insertion
sut.insert(it, 30);

// inserting a range of values
std::set<int> sut2;
sut2.insert(20);
sut2.insert(30);
sut2.insert(45);
std::set<int>::iterator itStart = sut2.begin();
std::set<int>::iterator itEnd = sut2.end();

sut.insert (itStart, itEnd); // second iterator is excluded from insertion

std::cout << std::endl << "Set under test contains:" << std::endl;
for (it = sut.begin(); it != sut.end(); ++it)
{
    std::cout << *it << std::endl;
}

return 0;
}

```

Output will be:

23 has been inserted!

23 already present in set!

Set under test contains:

5

7

12

20

23

30

第59.4节：在多重集合中插入值

集合中的所有插入方法同样适用于多重集合。不过，还有另一种可能，即提供一个初始化列表：

```
auto il = { 7, 5, 12 };
std::multiset<int> msut;
msut.insert(il);
```

第59.5节：在集合和多重集中搜索值

在std::set或std::multiset中搜索给定值有多种方法：

要获取键的第一个出现位置的迭代器，可以使用find()函数。如果键不存在，则返回end()。

```
std::set<int> sut;
sut.insert(10);
sut.insert(15);
sut.insert(22);
sut.insert(3); // 包含 3, 10, 15, 22

auto itS = sut.find(10); // 找到该值，所以 *itS == 10
itS = sut.find(555); // 未找到该值，所以 itS == sut.end()

std::multiset<int> msut;
sut.insert(10);
sut.insert(15);
sut.insert(22);
sut.insert(15);
sut.insert(3); // 包含 3, 10, 15, 15, 22

auto itMS = msut.find(10);
```

另一种方法是使用count()函数，该函数统计在set/multiset中找到的对应值的数量（对于set，返回值只能是0或1）。使用上述相同的值，结果如下：

```
int result = sut.count(10); // result == 1
result = sut.count(555); // result == 0

result = msut.count(10); // result == 1
result = msut.count(15); // result == 2
```

对于std::multiset，可能有多个元素具有相同的值。要获取这个范围，可以使用equal_range()函数。它返回一个std::pair，分别包含迭代器的下界（包含）和上界（不包含）。如果键不存在，两个迭代器都会指向最近的较大值（基于用于排序该multiset的比较方法）。

```
auto eqr = msut.equal_range(15);
auto st = eqr.first; // 指向第一个元素 '15'
auto en = eqr.second; // 指向元素 '22'
```

Section 59.4: Inserting values in a multiset

All the insertion methods from sets also apply to multisets. Nevertheless, another possibility exists, which is providing an initializer_list:

```
auto il = { 7, 5, 12 };
std::multiset<int> msut;
msut.insert(il);
```

Section 59.5: Searching values in set and multiset

There are several ways to search a given value in std::set or in std::multiset:

To get the iterator of the first occurrence of a key, the find() function can be used. It returns end() if the key does not exist.

```
std::set<int> sut;
sut.insert(10);
sut.insert(15);
sut.insert(22);
sut.insert(3); // contains 3, 10, 15, 22

auto itS = sut.find(10); // the value is found, so *itS == 10
itS = sut.find(555); // the value is not found, so itS == sut.end()

std::multiset<int> msut;
sut.insert(10);
sut.insert(15);
sut.insert(22);
sut.insert(15);
sut.insert(3); // contains 3, 10, 15, 15, 22

auto itMS = msut.find(10);
```

Another way is using the count() function, which counts how many corresponding values have been found in the set/multiset (in case of a set, the return value can be only 0 or 1). Using the same values as above, we will have:

```
int result = sut.count(10); // result == 1
result = sut.count(555); // result == 0

result = msut.count(10); // result == 1
result = msut.count(15); // result == 2
```

In the case of std::multiset, there could be several elements having the same value. To get this range, the equal_range() function can be used. It returns std::pair having iterator lower bound (inclusive) and upper bound (exclusive) respectively. If the key does not exist, both iterators would point to the nearest superior value (based on compare method used to sort the given multiset).

```
auto eqr = msut.equal_range(15);
auto st = eqr.first; // point to first element '15'
auto en = eqr.second; // point to element '22'
```

```
eqr = msut.equal_range(9); // eqr.first 和 eqr.second 都指向元素 '10'
```

```
eqr = msut.equal_range(9); // both eqr.first and eqr.second point to element '10'
```

第60章：std::integer_sequence

类模板std::integer_sequence<Type, Values...>表示类型为Type的一系列值，其中Type是内置整数类型之一。这些序列用于实现类或函数模板，这些模板利用位置访问的优势。标准库还包含“工厂”类型，仅根据元素数量创建递增的整数值序列。

第60.1节：将std::tuple<T...>转换为函数参数

std::tuple<T...>可用于传递多个值。例如，它可用于将一系列参数存储到某种队列中。处理这样的元组时，需要将其元素转换为函数调用参数：

```
#include <array>
#include <iostream>
#include <string>
#include <tuple>
#include <utility>

// -----
// 要调用的示例函数：
void f(int i, std::string const& s) {    std::cout
ut << "f(" << i << ", " << s << ")";}

void f(int i, double d, std::string const& s) {    std::cout
<< "f(" << i << ", " << d << ", " << s << ")";}

void f(char c, int i, double d, std::string const& s) {    std::cout <
< "f(" << c << ", " << i << ", " << d << ", " << s << ")";}

void f(int i, int j, int k) {
    std::cout << "f(" << i << ", " << j << ", " << k << ")";}

// -----
// 实际展开元组的函数：
template <typename 元组, std::size_t... I>
void 处理(元组 const& 元组, std::index_sequence<I...>) {
    f(std::get<I>(元组)...);
}

// 调用的接口。遗憾的是，它需要分派到另一个函数
// 以推断由 std::make_index_sequence<N> 创建的索引序列
template <typename Tuple>
void process(Tuple const& tuple) {
    process(tuple, std::make_index_sequence<std::tuple_size<Tuple>::value>());
}

// -----
int main() {
    process(std::make_tuple(1, 3.14, std::string("foo")));
    process(std::make_tuple('a', 2, 2.71, std::string("bar")));
    process(std::make_pair(3, std::string("pair")));
    process(std::array<int, 3>{ 1, 2, 3 });
}
```

只要一个类支持std::get<I>(object)和std::tuple_size<T>::value，就可以通过上述process()函数进行扩展。该函数本身完全独立于参数的数量。

Chapter 60: std::integer_sequence

The class template std::integer_sequence<Type, Values...> represents a sequence of values of type Type where Type is one of the built-in integer types. These sequences are used when implementing class or function templates which benefit from positional access. The standard library also contains "factory" types which create ascending sequences of integer values just from the number of elements.

Section 60.1: Turn a std::tuple<T...> into function parameters

A std::tuple<T...> can be used to pass multiple values around. For example, it could be used to store a sequence of parameters into some form of a queue. When processing such a tuple its elements need to be turned into function call arguments:

```
#include <array>
#include <iostream>
#include <string>
#include <tuple>
#include <utility>

// -----
// Example functions to be called:
void f(int i, std::string const& s) {
    std::cout << "f(" << i << ", " << s << ")\\n";
}
void f(int i, double d, std::string const& s) {
    std::cout << "f(" << i << ", " << d << ", " << s << ")\\n";
}
void f(char c, int i, double d, std::string const& s) {
    std::cout << "f(" << c << ", " << i << ", " << d << ", " << s << ")\\n";
}
void f(int i, int j, int k) {
    std::cout << "f(" << i << ", " << j << ", " << k << ")\\n";
}

// -----
// The actual function expanding the tuple:
template <typename Tuple, std::size_t... I>
void process(Tuple const& tuple, std::index_sequence<I...>) {
    f(std::get<I>(tuple)...);
}

// The interface to call. Sadly, it needs to dispatch to another function
// to deduce the sequence of indices created from std::make_index_sequence<N>
template <typename Tuple>
void process(Tuple const& tuple) {
    process(tuple, std::make_index_sequence<std::tuple_size<Tuple>::value>());
}

// -----
int main() {
    process(std::make_tuple(1, 3.14, std::string("foo")));
    process(std::make_tuple('a', 2, 2.71, std::string("bar")));
    process(std::make_pair(3, std::string("pair")));
    process(std::array<int, 3>{ 1, 2, 3 });
}
```

As long as a class supports std::get<I>(object) and std::tuple_size<T>::value it can be expanded with the above process() function. The function itself is entirely independent of the number of arguments.

第60.2节：创建由整数组成的参数包

`std::integer_sequence`本身是用于保存一系列整数，这些整数可以转换成参数包。它的主要价值在于可以创建“工厂”类模板来生成这些序列：

```
#include <iostream>
#include <initializer_list>
#include <utility>

template <typename T, T... I>
void print_sequence(std::integer_sequence<T, I...>) { std::initializer_list<bool>{ bool(std::cout << I << ' ')... }; std::cout << "\n"; }

template <int Offset, typename T, T... I>
void print_offset_sequence(std::integer_sequence<T, I...>) {
    print_sequence(std::integer_sequence<T, T(I + Offset)...>());
}

int main() {
    // 明确指定序列：
    print_sequence(std::integer_sequence<int, 1, 2, 3>());
    print_sequence(std::integer_sequence<char, 'f', 'o', 'o'>());

    // 生成序列：
    print_sequence(std::make_index_sequence<10>());
    print_sequence(std::make_integer_sequence<short, 10>());
    print_offset_sequence<'A'>(std::make_integer_sequence<char, 26>());
}
```

函数模板`print_sequence()` 使用了一个 `std::initializer_list<bool>`，在展开整数序列时保证了求值顺序，并且避免创建未使用的数组变量。

第60.3节：将索引序列转换为元素的副本

在逗号表达式中展开索引参数包并使用一个值，会为每个索引创建该值的副本。遗憾的是，`gcc` 和 `clang` 认为索引没有影响，并对此发出警告（`gcc` 可以通过将索引强制转换为 `void` 来消除警告）：

```
#include <algorithm>
#include <array>
#include <iostream>
#include <iterator>
#include <string>
#include <utility>

模板 <typename T, std::size_t... I>
std::array<T, sizeof...(I)> make_array(T const& value, std::index_sequence<I...>) {
    return std::array<T, sizeof...(I)>{ (I, value)... };
}

模板 <int N, typename T>
std::array<T, N> make_array(T const& value) {
    return make_array(value, std::make_index_sequence<N>());
}

int main() {
```

Section 60.2: Create a parameter pack consisting of integers

`std::integer_sequence` itself is about holding a sequence of integers which can be turned into a parameter pack. Its primary value is the possibility to create "factory" class templates creating these sequences:

```
#include <iostream>
#include <initializer_list>
#include <utility>

template <typename T, T... I>
void print_sequence(std::integer_sequence<T, I...>) {
    std::initializer_list<bool>{ bool(std::cout << I << ' ')... };
    std::cout << "\n";
}

template <int Offset, typename T, T... I>
void print_offset_sequence(std::integer_sequence<T, I...>) {
    print_sequence(std::integer_sequence<T, T(I + Offset)...>());
}

int main() {
    // explicitly specify sequences:
    print_sequence(std::integer_sequence<int, 1, 2, 3>());
    print_sequence(std::integer_sequence<char, 'f', 'o', 'o'>());

    // generate sequences:
    print_sequence(std::make_index_sequence<10>());
    print_sequence(std::make_integer_sequence<short, 10>());
    print_offset_sequence<'A'>(std::make_integer_sequence<char, 26>());
}
```

The `print_sequence()` function template uses an `std::initializer_list<bool>` when expanding the integer sequence to guarantee the order of evaluation and not creating an unused [array] variable.

Section 60.3: Turn a sequence of indices into copies of an element

Expanding the parameter pack of indices in a comma expression with a value creates a copy of the value for each of the indices. Sadly, `gcc` and `clang` think the index has no effect and warn about it (`gcc` can be silenced by casting the index to `void`):

```
#include <algorithm>
#include <array>
#include <iostream>
#include <iterator>
#include <string>
#include <utility>

template <typename T, std::size_t... I>
std::array<T, sizeof...(I)> make_array(T const& value, std::index_sequence<I...>) {
    return std::array<T, sizeof...(I)>{ (I, value)... };
}

template <int N, typename T>
std::array<T, N> make_array(T const& value) {
    return make_array(value, std::make_index_sequence<N>());
}

int main() {
```

```
auto array = make_array<20>(std::string("value"));    std::cop  
y(array.begin(), array.end(),           std::ostream  
am_iterator<std::string>(std::cout, " "));    std::cout << "";  
}
```

```
auto array = make_array<20>(std::string("value"));  
std::copy(array.begin(), array.end(),  
         std::ostream_iterator<std::string>(std::cout, " "));  
std::cout << "\n";
```

第61章：使用 std::unordered_map

std::unordered_map 只是一个关联容器。它基于键及其映射工作。键顾名思义，帮助保证映射中的唯一性。而映射的值只是与键相关联的内容。键和值的数据类型可以是任何预定义的数据类型或用户自定义类型。

第61.1节：声明与使用

如前所述，你可以声明任何类型的 unordered_map。我们来声明一个名为 first 的 unordered_map，键为字符串类型，值为整数类型。

```
unordered_map<string, int> first; //声明映射
first["One"] = 1; // 使用 [] 操作符插入值
first["Two"] = 2;
first["Three"] = 3;
first["Four"] = 4;
first["Five"] = 5;

pair <string,int> bar = make_pair("Nine", 9); //make a pair of same type
first.insert(bar); //can also use insert to feed the values
```

第61.2节：一些基本函数

```
unordered_map<数据类型, 数据类型> 变量名; //声明

变量名[键值] = 映射值; //插入值

变量名.find(键值); //返回指向键值的迭代器

变量名.begin(); //指向第一个元素的迭代器

变量名.end(); //指向最后一个元素的下一个位置的迭代器
```

Chapter 61: Using std::unordered_map

std::unordered_map is just an associative container. It works on keys and their maps. Key as the names goes, helps to have uniqueness in the map. While the mapped value is just a content that is associated with the key. The data types of this key and map can be any of the predefined data type or user-defined.

Section 61.1: Declaration and Usage

As already mentioned you can declare an unordered map of any type. Let's have a unordered map named first with string and integer type.

```
unordered_map<string, int> first; //declaration of the map
first["One"] = 1; // [] operator used to insert the value
first["Two"] = 2;
first["Three"] = 3;
first["Four"] = 4;
first["Five"] = 5;

pair <string,int> bar = make_pair("Nine", 9); //make a pair of same type
first.insert(bar); //can also use insert to feed the values
```

Section 61.2: Some Basic Functions

```
unordered_map<data_type, data_type> variable_name; //declaration

variable_name[key_value] = mapped_value; //inserting values

variable_name.find(key_value); //returns iterator to the key value

variable_name.begin(); // iterator to the first element

variable_name.end(); // iterator to the last + 1 element
```

第62章：标准库算法

第62.1节：std::next_permutation

```
template< class 迭代器 >
bool next_permutation( 迭代器 first, 迭代器 last );
template< class 迭代器, class 比较器 >
bool next_permutation( 迭代器 first, 迭代器 last, 比较函数 cmpFun );
```

效果：

将范围 [first, last) 内的数据序列调整为下一个字典序更高的排列。如果提供了cmpFun，则排列规则自定义。

参数：

first- 要排列的范围起始位置，包含该位置
last - 要排列的范围结束位置，不包含该位置

返回值：

如果存在这样的排列则返回 true。
否则将范围交换为字典序最小的排列并返回 false。

复杂度：

O(n), n 是从first到last的距离。

示例：

```
std::vector<int> v { 1, 2, 3 };
do
{
    for( int i = 0; i < v.size(); i += 1 )
    {
        std::cout << v[i];
    }
    std::cout << std::endl;
}while( std::next_permutation( v.begin(), v.end() ) );
```

按字典序递增顺序打印1,2,3的所有排列情况。

输出:

123

函数>

函数 for_each(输入迭代器 first, 输入迭代器 last, 函数 f);

效果：

将 f 应用于范围 [first, last) 中每个迭代器解引用的结果，起始于 first 并

Chapter 62: Standard Library Algorithms

Section 62.1: std::next_permutation

```
template< class Iterator >
bool next_permutation( Iterator first, Iterator last );
template< class Iterator, class Compare >
bool next_permutation( Iterator first, Iterator last, Compare cmpFun );
```

Effects:

Sift the data sequence of the range [first, last) into the next lexicographically higher permutation. If cmpFun is provided, the permutation rule is customized.

Parameters:

first- the beginning of the range to be permuted, inclusive
last - the end of the range to be permuted, exclusive

Return Value:

Returns true if such permutation exists.
Otherwise the range is swapped to the lexicographically smallest permutation and return false.

Complexity:

O(n), n is the distance from first to last.

Example:

```
std::vector< int > v { 1, 2, 3 };
do
{
    for( int i = 0; i < v.size(); i += 1 )
    {
        std::cout << v[i];
    }
    std::cout << std::endl;
}while( std::next_permutation( v.begin(), v.end() ) );
```

print all the permutation cases of 1,2,3 in lexicographically-increasing order.

output:

123
132
213
231
312
321

Section 62.2: std::for_each

```
template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f);
```

Effects:

Applies f to the result of dereferencing every iterator in the range [first, last) starting from first and

继续至 last - 1。

参数：

first, last - 应用 f 的范围。

f - 可调用对象，应用于范围 [first, last) 中每个迭代器解引用的结果。

返回值：

f (直到 C++11) 和 std::move(f) (自 C++11 起)。

复杂度：

精确应用 f 共 last - first 次。

示例：

版本 ≥ c++11

```
std::vector<int> v { 1, 2, 4, 8, 16 };
std::for_each(v.begin(), v.end(), [](int elem) { std::cout << elem << " "; });
```

对向量 v 的每个元素应用给定函数，将该元素打印到 stdout。

第62.3节：std::accumulate

定义于头文件 <numeric>

```
template<class 输入迭代器, class T>
T accumulate(输入迭代器 first, 输入迭代器 last, T init); // (1)

template<class 输入迭代器, class T, class 二元操作>
T accumulate(输入迭代器 first, 输入迭代器 last, T init, 二元操作 f); // (2)
```

效果：

std::accumulate 使用 f 函数对范围 [first, last) 执行折叠操作，起始累加值为 init。

实际上等价于：

```
T acc = init;
for (auto it = first; first != last; ++it)
    acc = f(acc, *it);
return acc;
```

在版本 (1) 中，使用了运算符+代替了f，因此对容器的累积相当于容器元素的求和。

参数：

first, last - 应用 f 的范围。

init - 累加器的初始值。

f - 二元折叠函数。

返回值：

proceeding to last - 1.

Parameters:

first, last - the range to apply f to.

f - callable object which is applied to the result of dereferencing every iterator in the range [first, last).

Return value:

f (until C++11) and std::move(f) (since C++11).

Complexity:

Applies f exactly last - first times.

Example:

Version ≥ c++11

```
std::vector<int> v { 1, 2, 4, 8, 16 };
std::for_each(v.begin(), v.end(), [](int elem) { std::cout << elem << " "; });
```

Applies the given function for every element of the vector v printing this element to [stdout](#).

Section 62.3: std::accumulate

Defined in header <numeric>

```
template<class InputIterator, class T>
T accumulate(InputIterator first, InputIterator last, T init); // (1)
```

```
template<class InputIterator, class T, class BinaryOperation>
T accumulate(InputIterator first, InputIterator last, T init, BinaryOperation f); // (2)
```

Effects:

std::accumulate performs fold operation using f function on range [first, last) starting with init as accumulator value.

Effectively it's equivalent of:

```
T acc = init;
for (auto it = first; first != last; ++it)
    acc = f(acc, *it);
return acc;
```

In version (1) operator+ is used in place of f, so accumulate over container is equivalent of sum of container elements.

Parameters:

first, last - the range to apply f to.

init - initial value of accumulator.

f - binary folding function.

Return value:

f 应用的累积值。

复杂度：

$O(n \times k)$, 其中 n 是从 first 到 last 的距离, $O(k)$ 是 f 函数的复杂度。

示例：

简单求和示例：

```
std::vector<int> v { 2, 3, 4 };
auto sum = std::accumulate(v.begin(), v.end(), 1);
std::cout << sum << std::endl;
```

输出：

10

将数字转换为数字：

```
版本 < c++11
类 Converter {
公共:
    int operator()(int a, int d) const { 返回 a * 10 + d; }
};
```

及以后版本

```
const int ds[3] = {1, 2, 3};
int n = std::accumulate(ds, ds + 3, 0, Converter());
std::cout << n << std::endl;

版本 ≥ c++11
const std::vector<int> ds = {1, 2, 3};
int n = std::accumulate(ds.begin(), ds.end(),
    0,
    [] (int a, int d) { 返回 a * 10 + d; });
std::cout << n << std::endl;
```

输出：

123

第62.4节：std::find

```
template <class InputIterator, class T>
InputIterator find (InputIterator first, InputIterator last, const T& val);
```

效果

在范围 [first, last) 内查找 val 的第一次出现

参数

first => 指向范围起始的迭代器 last => 指向范围末尾的迭代器 val => 要在范围内查找的值

Accumulated value of f applications.

Complexity:

$O(n \times k)$, where n is the distance from first to last, $O(k)$ is complexity of f function.

Example:

Simple sum example:

```
std::vector<int> v { 2, 3, 4 };
auto sum = std::accumulate(v.begin(), v.end(), 1);
std::cout << sum << std::endl;
```

Output:

10

Convert digits to number:

```
Version < c++11
class Converter {
public:
    int operator()(int a, int d) const { return a * 10 + d; }
};
```

and later

```
const int ds[3] = {1, 2, 3};
int n = std::accumulate(ds, ds + 3, 0, Converter());
std::cout << n << std::endl;

Version ≥ c++11
const std::vector<int> ds = {1, 2, 3};
int n = std::accumulate(ds.begin(), ds.end(),
    0,
    [] (int a, int d) { return a * 10 + d; });
std::cout << n << std::endl;
```

Output:

123

Section 62.4: std::find

```
template <class InputIterator, class T>
InputIterator find (InputIterator first, InputIterator last, const T& val);
```

Effects

Finds the first occurrence of val within the range [first, last)

Parameters

first => iterator pointing to the beginning of the range last => iterator pointing to the end of the range val => The value to find within the range

返回值

指向范围内第一个等于(==) val 的元素的迭代器，如果未找到 val，则迭代器指向 last。

示例

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

int main(int argc, const char * argv[]) {

    //创建一个向量
    vector<int> intVec {4, 6, 8, 9, 10, 30, 55, 100, 45, 2, 4, 7, 9, 43, 48};

    //定义迭代器
    vector<int>::iterator itr_9;
    vector<int>::iterator itr_43;
    vector<int>::iterator itr_50;

    //调用 find 函数
    itr_9 = find(intVec.begin(), intVec.end(), 9); //出现两次
    itr_43 = find(intVec.begin(), intVec.end(), 43); //出现一次

    //向量中不存在的值
    itr_50 = find(intVec.begin(), intVec.end(), 50); //不存在

    cout << "首次出现的值: " << *itr_9 << endl;
    cout << "唯一出现的值: " << *itr_43 << endl;

    /*
    让我们通过查看 9 后面的元素来证明 itr_9 指向的是 9 的第一次出现,
    该元素应该是 10,
    而不是 43
    */
    cout << "第一个9之后的元素: " << *(itr_9 + 1) << endl;

    /*
    为了避免解引用 intVec.end(), 我们来看一下
    结束之前的元素
    */
    cout << "最后一个元素: " << *(itr_50 - 1) << endl;

    return 0;
}
```

输出

```
第一个出现的: 9
唯一出现的: 43
第一个9之后的元素: 10
最后一个元素: 48
```

Return

An iterator that points to the first element within the range that is equal(==) to val, the iterator points to last if val is not found.

Example

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

int main(int argc, const char * argv[]) {

    //create a vector
    vector<int> intVec {4, 6, 8, 9, 10, 30, 55, 100, 45, 2, 4, 7, 9, 43, 48};

    //define iterators
    vector<int>::iterator itr_9;
    vector<int>::iterator itr_43;
    vector<int>::iterator itr_50;

    //calling find
    itr_9 = find(intVec.begin(), intVec.end(), 9); //occurs twice
    itr_43 = find(intVec.begin(), intVec.end(), 43); //occurs once

    //a value not in the vector
    itr_50 = find(intVec.begin(), intVec.end(), 50); //does not occur

    cout << "first occurrence of: " << *itr_9 << endl;
    cout << "only occurrence of: " << *itr_43 << endl;

    /*
    let's prove that itr_9 is pointing to the first occurrence
    of 9 by looking at the element after 9, which should be 10
    not 43
    */
    cout << "element after first 9: " << *(itr_9 + 1) << endl;

    /*
    to avoid dereferencing intVec.end(), lets look at the
    element right before the end
    */
    cout << "last element: " << *(itr_50 - 1) << endl;

    return 0;
}
```

Output

```
first occurrence of: 9
only occurrence of: 43
element after first 9: 10
last element: 48
```

第62.5节 : std::min_element

```
template <class ForwardIterator>
ForwardIterator min_element (ForwardIterator first, ForwardIterator last);

template <class ForwardIterator, class Compare>
ForwardIterator min_element (ForwardIterator first, ForwardIterator last, Compare comp);
```

效果

在范围内查找最小元素

参数

first - 指向范围起始的迭代器
last - 指向范围末尾的迭代器 comp - 一个函数指针或函数对象，接受两个参数并返回true或false，表示第一个参数是否小于第二个参数。该函数不应修改输入

返回值

指向范围内最小元素的迭代器

复杂度

与比较元素数量减一成线性关系。

示例

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <utility> //用于使用 make_pair

using namespace std;

//比较两个 pair 的函数
bool pairLessThanFunction(const pair<string, int> &p1, const pair<string, int> &p2)
{
    return p1.second < p2.second;
}

int main(int argc, const char * argv[])
{
    vector<int> intVec {30,200,167,56,75,94,10,73,52,6,39,43};

    vector<pair<string, int>> pairVector = {make_pair("y", 25), make_pair("b", 2), make_pair("z", 26), make_pair("e", 5)};

    // 默认使用 < 运算符
    auto minInt = min_element(intVec.begin(), intVec.end());

    // 使用 pairLessThanFunction
    auto minPairFunction = min_element(pairVector.begin(), pairVector.end(), pairLessThanFunction);

    // 输出 intVector 的最小值
    cout << "min int from default: " << *minInt << endl;
}
```

Section 62.5: std::min_element

```
template <class ForwardIterator>
ForwardIterator min_element (ForwardIterator first, ForwardIterator last);

template <class ForwardIterator, class Compare>
ForwardIterator min_element (ForwardIterator first, ForwardIterator last, Compare comp);
```

Effects

Finds the minimum element in a range

Parameters

first - iterator pointing to the beginning of the range
last - iterator pointing to the end of the range comp - a function pointer or function object that takes two arguments and returns true or false indicating whether argument is less than argument 2. This function should not modify inputs

Return

Iterator to the minimum element in the range

Complexity

Linear in one less than the number of elements compared.

Example

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <utility> //to use make_pair

using namespace std;

//function compare two pairs
bool pairLessThanFunction(const pair<string, int> &p1, const pair<string, int> &p2)
{
    return p1.second < p2.second;
}

int main(int argc, const char * argv[])
{
    vector<int> intVec {30,200,167,56,75,94,10,73,52,6,39,43};

    vector<pair<string, int>> pairVector = {make_pair("y", 25), make_pair("b", 2), make_pair("z", 26), make_pair("e", 5)};

    // default using < operator
    auto minInt = min_element(intVec.begin(), intVec.end());

    //Using pairLessThanFunction
    auto minPairFunction = min_element(pairVector.begin(), pairVector.end(), pairLessThanFunction);

    //print minimum of intVector
    cout << "min int from default: " << *minInt << endl;
}
```

```

// 输出 pairVector 的最小值
cout << "min pair from PairLessThanFunction: " << (*minPairFunction).second << endl;

return 0;
}

```

输出

```

min int from default: 6
min pair from PairLessThanFunction: 2

```

第62.6节 : std::find_if

```

template <class 输入迭代器, class 一元谓词>
输入迭代器 find_if (输入迭代器 first, 输入迭代器 last, 一元谓词 pred);

```

效果

查找范围内第一个使谓词函数 pred 返回 true 的元素。

参数

first => 指向范围起始的迭代器 last => 指向范围末尾的迭代器 pred =>
谓词函数 (返回 true 或 false)

返回值

返回一个迭代器，指向范围内第一个使谓词函数 pred 返回 true 的元素。
如果未找到该元素，迭代器指向 last

示例

```

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

/*
定义一些用作谓词的函数
*/

// 如果 x 是 10 的倍数则返回 true
bool is10的倍数(int x) {
    return x % 10 == 0;
}

// 如果元素大于传入参数则返回 true
class Greater {
    int _than;

public:
    Greater(int th):_than(th){

    }
    bool operator()(int data) const

```

```

//print minimum of pairVector
cout << "min pair from PairLessThanFunction: " << (*minPairFunction).second << endl;

return 0;
}

```

Output

```

min int from default: 6
min pair from PairLessThanFunction: 2

```

Section 62.6: std::find_if

```

template <class InputIterator, class UnaryPredicate>
InputIterator find_if (InputIterator first, InputIterator last, UnaryPredicate pred);

```

Effects

Finds the first element in a range for which the predicate function pred returns true.

Parameters

first => iterator pointing to the beginning of the range last => iterator pointing to the end of the range pred =>
predicate function(returns true or false)

Return

An iterator that points to the first element within the range the predicate function pred returns true for. The
iterator points to last if val is not found

Example

```

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

/*
define some functions to use as predicates
*/

//Returns true if x is multiple of 10
bool multOf10(int x) {
    return x % 10 == 0;
}

//returns true if item greater than passed in parameter
class Greater {
    int _than;

public:
    Greater(int th):_than(th){

    }
    bool operator()(int data) const

```

```

{
    return data > _than;
}
};

int main()
{
vector<int> myvec {2, 5, 6, 10, 56, 7, 48, 89, 850, 7, 456};

//使用lambda函数
vector<int>::iterator gt10 = find_if(myvec.begin(), myvec.end(), [](int x){return x>10;}); // >=
C++11

//使用函数指针
vector<int>::iterator pow10 = find_if(myvec.begin(), myvec.end(), multOf10);

//使用函数对象
vector<int>::iterator gt5 = find_if(myvec.begin(), myvec.end(), Greater(5));

//not Found
vector<int>::iterator nf = find_if(myvec.begin(), myvec.end(), Greater(1000)); // nf 指向
myvec.end()

//检查指针是否指向 myvec.end()
if(nf != myvec.end()) {
    cout << "nf 指向: " << *nf << endl;
}
else {
    cout << "未找到项目" << endl;
}

cout << "第一个大于 10 的项目: " << *gt10 << endl;
cout << "第一个乘以 10 的项目: " << *pow10 << endl;
cout << "第一个大于 5 的项目: " << *gt5 << endl;

return 0;
}

```

输出

```

未找到项目
第一个大于 10 的项目: 56
第一个乘以 10 的项目: 10
第一个大于 5 的项目: 6

```

第 62.7 节：使用 std::nth_element 查找中位数（或其他分位数）

`std::nth_element` 算法接受三个迭代器：指向开始位置、第 n 个位置和结束位置的迭代器。函数返回后，第 n 个元素（按顺序）将是第 n 小的元素。（该函数有更复杂的重载，例如接受比较函数对象的版本；详见上述链接了解所有变体。）

注意 此函数非常高效——它具有线性复杂度。

```

{
    return data > _than;
}
};

int main()
{
vector<int> myvec {2, 5, 6, 10, 56, 7, 48, 89, 850, 7, 456};

//with a lambda function
vector<int>::iterator gt10 = find_if(myvec.begin(), myvec.end(), [](int x){return x>10;}); // >=
C++11

//with a function pointer
vector<int>::iterator pow10 = find_if(myvec.begin(), myvec.end(), multOf10);

//with functor
vector<int>::iterator gt5 = find_if(myvec.begin(), myvec.end(), Greater(5));

//not Found
vector<int>::iterator nf = find_if(myvec.begin(), myvec.end(), Greater(1000)); // nf points to
myvec.end()

//check if pointer points to myvec.end()
if(nf != myvec.end()) {
    cout << "nf points to: " << *nf << endl;
}
else {
    cout << "item not found" << endl;
}

cout << "First item > 10: " << *gt10 << endl;
cout << "First Item n * 10: " << *pow10 << endl;
cout << "First Item > 5: " << *gt5 << endl;

return 0;
}

```

Output

```

item not found
First item > 10: 56
First Item n * 10: 10
First Item > 5: 6

```

Section 62.7: Using std::nth_element To Find The Median (Or Other Quantiles)

The `std::nth_element` algorithm takes three iterators: an iterator to the beginning, n th position, and end. Once the function returns, the n th element (by order) will be the n th smallest element. (The function has more elaborate overloads, e.g., some taking comparison functors; see the above link for all the variations.)

Note This function is very efficient - it has linear complexity.

为了举例说明，我们将长度为 n 的序列的中位数定义为位于位置 $\lceil n / 2 \rceil$ 的元素。例如，长度为5的序列的中位数是第3小的元素，长度为6的序列的中位数也是第3小的元素。

要使用此函数查找中位数，我们可以使用以下方法。假设我们从

```
std::vector<int> v{5, 1, 2, 3, 4};

std::vector<int>::iterator b = v.begin();
std::vector<int>::iterator e = v.end();

std::vector<int>::iterator med = b;
std::advance(med, v.size() / 2);

// 这使得第2个位置存放中位数。
std::nth_element(b, med, e);

// 中位数现在位于 v[2]。
```

要查找第 p 分位数，我们需要修改上述部分代码：

```
const std::size_t pos = p * std::distance(b, e);

std::advance(nth, pos);
```

并查找位于 pos 位置的分位数。

第62.8节 : std::count

```
template <class InputIterator, class T>
typename iterator_traits<InputIterator>::difference_type
count (InputIterator first, InputIterator last, const T& val);
```

效果

计算等于 val 的元素数量

参数

$first \Rightarrow$ 指向范围起始的迭代器
 $last \Rightarrow$ 指向范围末尾的迭代器
 $val \Rightarrow$ 将统计该值在范围内出现的次数

返回值

范围内等于($==$) val 的元素数量。

示例

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

int main(int argc, const char * argv[]) {
```

For the sake of this example, let's define the median of a sequence of length n as the element that would be in position $\lceil n / 2 \rceil$. For example, the median of a sequence of length 5 is the 3rd smallest element, and so is the median of a sequence of length 6.

To use this function to find the median, we can use the following. Say we start with

```
std::vector<int> v{5, 1, 2, 3, 4};

std::vector<int>::iterator b = v.begin();
std::vector<int>::iterator e = v.end();

std::vector<int>::iterator med = b;
std::advance(med, v.size() / 2);

// This makes the 2nd position hold the median.
std::nth_element(b, med, e);

// The median is now at v[2].
```

To find the p th quantile, we would change some of the lines above:

```
const std::size_t pos = p * std::distance(b, e);

std::advance(nth, pos);
```

and look for the quantile at position pos .

Section 62.8: std::count

```
template <class InputIterator, class T>
typename iterator_traits<InputIterator>::difference_type
count (InputIterator first, InputIterator last, const T& val);
```

Effects

Counts the number of elements that are equal to val

Parameters

$first \Rightarrow$ iterator pointing to the beginning of the range
 $last \Rightarrow$ iterator pointing to the end of the range
 $val \Rightarrow$ The occurrence of this value in the range will be counted

Return

The number of elements in the range that are equal($==$) to val .

Example

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

int main(int argc, const char * argv[]) {
```

```

//创建向量
vector<int> intVec{4,6,8,9,10,30,55,100,45,2,4,7,9,43,48};

//统计9、55和101的出现次数
size_t count_9 = count(intVec.begin(), intVec.end(), 9); //出现两次
size_t count_55 = count(intVec.begin(), intVec.end(), 55); //出现一次
size_t count_101 = count(intVec.begin(), intVec.end(), 101); //出现一次

//打印结果
cout << "有 " << count_9 << " 个9" << endl;
cout << "有 " << count_55 << " 个55" << endl;
cout << "有 " << count_101 << " 个101" << endl;

//查找向量中第一个等于4的元素
vector<int>::iterator itr_4 = find(intVec.begin(), intVec.end(), 4);

// 从第一个元素开始计算它在向量中的出现次数
size_t count_4 = count(itr_4, intVec.end(), *itr_4); // 应该是2

cout << "有 " << count_4 << " 个 " << *itr_4 << endl;

return 0;
}

```

输出

```

有 2 个9
有 1 个55
有 0 个101
有 2 个4

```

第62.9节 : std::count_if

```

template <class 输入迭代器, class 一元谓词>
typename iterator_traits<输入迭代器>::difference_type
count_if (输入迭代器 first, 输入迭代器 last, 一元谓词 red);

```

效果

计算指定范围内满足给定谓词函数为真的元素数量

参数

first => 指向范围起始的迭代器 last => 指向范围末尾的迭代器 red =>
谓词函数 (返回true或false)

返回值

指定范围内谓词函数返回true的元素数量。

示例

```

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

```

```

//create vector
vector<int> intVec{4,6,8,9,10,30,55,100,45,2,4,7,9,43,48};

//count occurrences of 9, 55, and 101
size_t count_9 = count(intVec.begin(), intVec.end(), 9); //occurs twice
size_t count_55 = count(intVec.begin(), intVec.end(), 55); //occurs once
size_t count_101 = count(intVec.begin(), intVec.end(), 101); //occurs once

//print result
cout << "There are " << count_9 << " 9s" << endl;
cout << "There is " << count_55 << " 55" << endl;
cout << "There is " << count_101 << " 101" << endl;

//find the first element == 4 in the vector
vector<int>::iterator itr_4 = find(intVec.begin(), intVec.end(), 4);

//count its occurrences in the vector starting from the first one
size_t count_4 = count(itr_4, intVec.end(), *itr_4); // should be 2

cout << "There are " << count_4 << " " << *itr_4 << endl;

return 0;
}

```

Output

```

There are 2 9s
There is 1 55
There is 0 101
There are 2 4

```

Section 62.9: std::count_if

```

template <class InputIterator, class UnaryPredicate>
typename iterator_traits<InputIterator>::difference_type
count_if (InputIterator first, InputIterator last, UnaryPredicate red);

```

Effects

Counts the number of elements in a range for which a specified predicate function is true

Parameters

first => iterator pointing to the beginning of the range last => iterator pointing to the end of the range red =>
predicate function(returns true or false)

Return

The number of elements within the specified range for which the predicate function returned true.

Example

```

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

```

```

/*
定义几个用作谓词的函数
*/

//如果数字是奇数则返回true
bool isOdd(int i){
    return i%2 == 1;
}

//如果数字大于构造函数参数值则返回true的函数对象

class Greater {
    int _than;
public:
Greater(int th): _than(th){}
    bool operator()(int i){
        return i > _than;
    }
};

int main(int argc, const char * argv[]) {

    //创建一个向量
vector<int> myvec = {1,5,8,0,7,6,4,5,2,1,5,0,6,9,7};

    //使用lambda函数统计偶数个数
    size_t evenCount = count_if(myvec.begin(), myvec.end(), [](int i){return i % 2 == 0;}); // >=
C++11

    //使用函数指针统计向量前半部分的奇数个数
    size_t oddCount = count_if(myvec.begin(), myvec.end()- myvec.size()/2, isOdd);

    //使用函数对象统计大于5的数字个数
    size_t greaterCount = count_if(myvec.begin(), myvec.end(), Greater(5));

    cout << "vector size: " << myvec.size() << endl;
    cout << "even numbers: " << evenCount << " found" << endl;
    cout << "odd numbers: " << oddCount << " found" << endl;
    cout << "numbers > 5: " << greaterCount << " found" << endl;

    return 0;
}

```

输出

```

向量大小: 15
偶数: 7 找到
奇数: 4 找到
数字 > 5: 6 找到

```

```

/*
Define a few functions to use as predicates
*/

//return true if number is odd
bool isOdd(int i){
    return i%2 == 1;
}

//functor that returns true if number is greater than the value of the constructor parameter
provided
class Greater {
    int _than;
public:
Greater(int th): _than(th){}
    bool operator()(int i){
        return i > _than;
    }
};

int main(int argc, const char * argv[]) {

    //create a vector
vector<int> myvec = {1,5,8,0,7,6,4,5,2,1,5,0,6,9,7};

    //using a lambda function to count even numbers
    size_t evenCount = count_if(myvec.begin(), myvec.end(), [](int i){return i % 2 == 0;}); // >=
C++11

    //using function pointer to count odd number in the first half of the vector
    size_t oddCount = count_if(myvec.begin(), myvec.end()- myvec.size()/2, isOdd);

    //using a functor to count numbers greater than 5
    size_t greaterCount = count_if(myvec.begin(), myvec.end(), Greater(5));

    cout << "vector size: " << myvec.size() << endl;
    cout << "even numbers: " << evenCount << " found" << endl;
    cout << "odd numbers: " << oddCount << " found" << endl;
    cout << "numbers > 5: " << greaterCount << " found" << endl;

    return 0;
}

```

Output

```

vector size: 15
even numbers: 7 found
odd numbers: 4 found
numbers > 5: 6 found

```

第63章：ISO C++标准

1998年，首次发布了使C++成为内部标准化语言的标准。从那时起，C++不断发展，形成了不同的C++方言。在本页，您可以找到所有不同标准的概述及其相较于前一版本的变化。关于如何使用这些特性的详细说明，请参见更专业的页面。

第63.1节：当前工作草案

所有已发布的ISO标准均可从ISO商店 (<http://www.iso.org>) 购买。不过，[C++标准的工作草案是公开免费提供的](#)。

标准的不同版本：

- 即将发布（有时称为C++20或C++2a）：[当前工作草案 \(HTML版本\)](#)
- 提议版（有时称为C++17或C++1z）：[2017年3月工作草案N4659](#)
- C++14（有时称为C++1y）：[2014年11月工作草案N4296](#)
- C++11（有时称为C++0x）：[2011年2月工作草案N3242](#)
- C++03
- C++98

第63.2节：C++17

C++17 标准功能已完整，并已提议标准化。在支持这些功能的实验性编译器中，通常称为 C++1z。

语言扩展

- 折叠表达式
- 使用auto声明非类型模板参数
- 保证拷贝省略
- 构造函数的模板参数推导
- 结构化绑定
- 紧凑的嵌套命名空间
- 新属性：`[[fallthrough]]`, `[[nodiscard]]`, `[[maybe_unused]]`
- `static_assert` 的默认消息
- `if` 和 `switch` 中的初始化器
- 内联变量
- `if constexpr`
- 表达式求值顺序保证
- 为超对齐数据分配动态内存

库扩展

- `std::optional`
- `std::variant`
- `std::string_view`
- 关联容器的 `merge()` 和 `extract()`
- 带有 `<filesystem>` 头文件的文件系统库。
- 大多数标准算法的并行版本（在 `<algorithm>` 头文件中）。
- 在 `<cmath>` 头文件中添加数学特殊函数。
- 在 `map<>`、`unordered_map<>`、`set<>` 和 `unordered_set<>` 之间移动节点

Chapter 63: The ISO C++ Standard

In 1998, there was a first publication of the standard making C++ an internally standardized language. From that time, C++ has evolved resulting in different dialects of C++. On this page, you can find an overview of all different standards and their changes compared to the previous version. The details on how to use these features is described on more specialized pages.

Section 63.1: Current Working Drafts

All published ISO standards are available for sale from the ISO store (<http://www.iso.org>). The working drafts of the C++ standards are publicly available for free though.

The different versions of the standard:

- Upcoming (Sometimes referred as C++20 or C++2a): [Current working draft \(HTML-version\)](#)
- Proposed (Sometimes referred as C++17 or C++1z): [March 2017 working draft N4659](#).
- C++14 (Sometimes referred as C++1y): [November 2014 working draft N4296](#)
- C++11 (Sometimes referred as C++0x): [February 2011 working draft N3242](#)
- C++03
- C++98

Section 63.2: C++17

The C++17 standard is feature complete and has been proposed for standardization. In compilers with experimental support for these features, it is usually referred to as C++1z.

Language Extensions

- Fold Expressions
- Declaring non-type template arguments with `auto`
- Guaranteed copy elision
- Template parameter deduction for constructors
- Structured bindings
- Compact nested namespaces
- New attributes: `[[fallthrough]]`, `[[nodiscard]]`, `[[maybe_unused]]`
- Default message for `static_assert`
- Initializers in `if` and `switch`
- Inline variables
- `if constexpr`
- Order of expression evaluation guarantees
- Dynamic memory allocation for over-aligned data

Library Extensions

- `std::optional`
- `std::variant`
- `std::string_view`
- `merge()` and `extract()` for associative containers
- A file system library with the `<filesystem>` header.
- Parallel versions of most of the standard algorithms (in the `<algorithm>` header).
- Addition of mathematical special functions in the `<cmath>` header.
- Moving nodes between `map<>`, `unordered_map<>`, `set<>`, and `unordered_set<>`

第63.3节：C++11

C++11标准是对C++标准的重大扩展。以下是根据isocpp常见问题解答分组的更改概述，并附有指向更详细文档的链接。

语言扩展

通用特性

- auto
- decltype
- 范围for语句
- 初始化列表
- 统一初始化语法和语义
- 右值引用和移动语义
- Lambda表达式
- noexcept防止异常传播
- constexpr
- nullptr – 空指针字面量
- 复制和重新抛出异常
- 内联命名空间
- 用户自定义字面量

类

- =default 和 =delete
- 默认移动和复制的控制
- 委托构造函数
- 类内成员初始化器
- 继承构造函数
- 重写控制：override
- 重写控制：final
- 显式转换运算符

其他类型

- 枚举类
- 长长整数-更长的整数
- 扩展整数类型
- 广义联合体
- 广义POD (Plain Old Data) 类型

模板

- 外部模板
- 模板别名
- 可变参数模板
- 本地类型作为模板参数

并发

- 并发内存模型
- 并发中的动态初始化和销毁
- 线程局部存储

杂项语言特性

Section 63.3: C++11

The C++11 standard is a major extension to the C++ standard. Below you can find an overview of the changes as they have been grouped on the [isocpp FAQ](#) with links to more detailed documentation.

Language Extensions

General Features

- auto
- decltype
- Range-for statement
- Initializer lists
- Uniform initialization syntax and semantics
- Rvalue references and move semantics
- Lambdas
- noexcept to prevent exception propagation
- constexpr
- nullptr – a null pointer literal
- Copying and rethrowing exceptions
- Inline namespaces
- User-defined literals

Classes

- =default and =delete
- Control of default move and copy
- Delegating constructors
- In-class member initializers
- Inherited constructors
- Override controls: override
- Override controls: final
- Explicit conversion operators

Other Types

- enum class
- long long – a longer integer
- Extended integer types
- Generalized unions
- Generalized PODs

Templates

- Extern templates
- Template aliases
- Variadic templates
- Local types as template arguments

Concurrency

- Concurrency memory model
- Dynamic initialization and destruction with concurrency
- Thread-local storage

Miscellaneous Language Features

- C++11中`_cplusplus`的值是多少？
- 后缀返回类型语法
- 防止窄化
- 右尖括号
- `static_assert`编译时断言
- 原始字符串字面量
- 属性
- 对齐
- C99特性

库扩展

通用

- `unique_ptr`
- `shared_ptr`
- `weak_ptr`
- 垃圾回收 ABI
- 元组
- 类型特性
- 函数与绑定
- 正则表达式
- 时间工具
- 随机数生成
- 作用域分配器

容器与算法

- 算法改进
- 容器改进
- 无序`*`容器
- `std::array`
- `forward_list`

并发

- 线程
- 互斥
- 锁
- 条件变量
- 原子操作
- 期货和承诺
- 异步
- 放弃一个进程

第63.4节：C++14

C++14标准通常被称为C++11的错误修正。它只包含有限的更改列表，其中大多数是对C++11中新特性的扩展。下面您可以找到根据isocpp常见问题解答分组的更改概览，并附有更详细文档的链接。

语言扩展

- 二进制字面量
- 泛化的返回类型推断

- What is the value of `_cplusplus` for C++11?
- Suffix return type syntax
- Preventing narrowing
- Right-angle brackets
- `static_assert` compile-time assertions
- Raw string literals
- Attributes
- Alignment
- C99 features

Library Extensions

General

- `unique_ptr`
- `shared_ptr`
- `weak_ptr`
- Garbage collection ABI
- `tuple`
- Type traits
- function and bind
- Regular Expressions
- Time utilities
- Random number generation
- Scoped allocators

Containers and Algorithms

- Algorithms improvements
- Container improvements
- `unordered_*` containers
- `std::array`
- `forward_list`

Concurrency

- Threads
- Mutual exclusion
- Locks
- Condition variables
- Atomics
- Futures and promises
- `async`
- Abandoning a process

Section 63.4: C++14

The C++14 standard is often referred to as a bugfix for C++11. It contains only a limited list of changes of which most are extensions to the new features in C++11. Below you can find an overview of the changes as they have been grouped on the [isocpp FAQ](#) with links to more detailed documentation.

Language Extensions

- Binary literals
- Generalized return type deduction

- decltype(auto)
- 泛化的lambda捕获
- 通用lambda
- 变量模板
- 扩展的constexpr
- [[deprecated]]属性
- 数字分隔符

库扩展

- 共享锁定
- 用户自定义字面量用于std::类型
- std::make_unique
- 类型转换_t别名
- 按类型访问元组 (例如get<string>(t))
- 透明操作符函数对象 (例如greater<>(x))
- std::quoted

已弃用 / 已移除

- std::gets在C++11中被弃用，并在C++14中移除
- std::random_shuffle已被弃用

第63.5节：C++98

C++98是C++的第一个标准化版本。由于它是作为C的扩展开发的，许多将C++与C区分开的特性被添加进来。

语言扩展（相对于C89/C90）

- 类，派生类，虚成员函数，常成员函数
- 函数重载，操作符重载
- 单行注释（在C语言中随C99标准引入）
- 引用
- new 和 delete
- 布尔类型（已在C语言的C99标准中引入）
- 模板
- 命名空间
- 异常
- 特定类型转换

库扩展

- 标准模板库

第63.6节：C++03

C++03标准主要解决了C++98标准的缺陷报告。除这些缺陷外，它只添加了一个新特性。

语言扩展

- 值初始化

- decltype(auto)
- Generalized lambda captures
- Generic lambdas
- Variable templates
- Extended constexpr
- The [[deprecated]] attribute
- Digit separators

Library Extensions

- Shared locking
- User-defined literals for std:: types
- std::make_unique
- Type transformation _t aliases
- Addressing tuples by type (e.g. get<string>(t))
- Transparent Operator Function (e.g. greater<>(x))
- std::quoted

Deprecated / Removed

- std::gets was deprecated in C++11 and removed from C++14
- std::random_shuffle is deprecated

Section 63.5: C++98

C++98 is the first standardized version of C++. As it was developed as an extension to C, many of the features which set apart C++ from C are added.

Language Extensions (in respect to C89/C90)

- Classes, Derived classes, virtual member functions, const member functions
- Function overloading, Operator overloading
- Single line comments (Has been introduced in the C-language with C99 standard)
- References
- new and delete
- boolean type (Has been introduced in the C-language with C99 standard)
- templates
- namespaces
- exceptions
- specific casts

Library Extensions

- The Standard Template Library

Section 63.6: C++03

The C++03 standard mainly addresses defect reports of the C++98 standard. Apart from these defects, it only adds one new feature.

Language Extensions

- Value initialization

第63.7节：C++20

C++20是即将发布的C++标准，目前正在开发中，基于C++17标准。其进展可以在官方ISO cpp网站上跟踪。

以下特性是已被接受用于下一版C++标准的内容，目标为2020.

语言扩展

目前尚未接受任何语言扩展。

库扩展

目前尚未接受任何库扩展。

Section 63.7: C++20

C++20 is the upcoming standard of C++, currently in development, based upon the C++17 standard. Its progress can be tracked on the [official ISO cpp website](#).

The following features are simply what has been accepted for the next release of the C++ standard, targeted for 2020.

Language Extensions

No language extensions have been accepted for now.

Library Extensions

No library extensions have been accepted for now.

第64章：内联变量

允许在多个翻译单元中定义内联变量而不违反一定义规则。如果它被多次定义，链接器将在最终程序中将所有定义合并为单个对象。

第64.1节：在类定义中定义静态数据成员

如果类的静态数据成员被声明为`inline`，则可以在类定义内完全定义它。例如，以下类可以定义在头文件中。在C++17之前，必须提供一个`.cpp`文件来包含`Foo::num_instances`的定义，以确保它只被定义一次，但在C++17中，内联变量`Foo::num_instances`的多重定义都指向同一个`int`对象。

```
// 警告：线程不安全...
class Foo {
public:
    Foo() { ++num_instances; }
    ~Foo() { --num_instances; }
    inline static int num_instances = 0;
};
```

作为特殊情况，`constexpr` 静态数据成员隐式为内联。

```
类 MyString {
public:
    MyString() { /* ... */ }
    // ...
    static constexpr int max_size = INT_MAX / 2;
};

// 在 C++14 中，此定义需要在单个翻译单元中：
// constexpr int MyString::max_size;
```

Chapter 64: Inline variables

An inline variable is allowed to be defined in multiple translation units without violating the One Definition Rule. If it is multiply defined, the linker will merge all definitions into a single object in the final program.

Section 64.1: Defining a static data member in the class definition

A static data member of the class may be fully defined within the class definition if it is declared `inline`. For example, the following class may be defined in a header. Prior to C++17, it would have been necessary to provide a `.cpp` file to contain the definition of `Foo::num_instances` so that it would be defined only once, but in C++17 the multiple definitions of the inline variable `Foo::num_instances` all refer to the same `int` object.

```
// warning: not thread-safe...
class Foo {
public:
    Foo() { ++num_instances; }
    ~Foo() { --num_instances; }
    inline static int num_instances = 0;
};
```

As a special case, a `constexpr` static data member is implicitly inline.

```
class MyString {
public:
    MyString() { /* ... */ }
    // ...
    static constexpr int max_size = INT_MAX / 2;
};

// in C++14, this definition was required in a single translation unit:
// constexpr int MyString::max_size;
```

第65章：随机数生成

第65.1节：真正的随机值生成器

为了生成可用于加密的真正随机值，必须使用 `std::random_device` 作为生成器。

```
#include <iostream>
#include <random>

int main()
{
    std::random_device crypto_random_generator;
    std::uniform_int_distribution<int> int_distribution(0, 9);

    int actual_distribution[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

    for (int i = 0; i < 10000; i++) {
        int result = int_distribution(crypto_random_generator);
        actual_distribution[result]++;
    }

    for (int i = 0; i < 10; i++) {
        std::cout << actual_distribution[i] << " ";
    }

    return 0;
}
```

`std::random_device` 的使用方式与伪随机数生成器相同。

然而，如果实现中没有可用的非确定性源（例如硬件设备），`std::random_device` 可能会基于实现定义的伪随机数引擎来实现。

通过 `entropy` 成员函数（当生成器完全确定时返回零）应该可以检测到此类实现，但许多流行库（包括 GCC 的 `libstdc++` 和 LLVM 的 `libc++`）即使在使用高质量外部随机性的情况下，也总是返回零。

第 65.2 节：生成伪随机数

伪随机数生成器生成的值可以根据之前生成的值被猜测出来。

换句话说：它是确定性的。在需要真正随机数的情况下，不要使用伪随机数生成器。

```
#include <iostream>
#include <random>

int main()
{
    std::default_random_engine 伪随机生成器;
    std::uniform_int_distribution<int> 整数分布(0, 9);

    int actual_distribution[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

    for (int i = 0; i < 10000; i++) {
        int 结果 = 整数分布(伪随机生成器);
        实际分布[结果]++;
    }
}
```

Chapter 65: Random number generation

Section 65.1: True random value generator

To generate true random values that can be used for cryptography `std::random_device` has to be used as generator.

```
#include <iostream>
#include <random>

int main()
{
    std::random_device crypto_random_generator;
    std::uniform_int_distribution<int> int_distribution(0, 9);

    int actual_distribution[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

    for(int i = 0; i < 10000; i++) {
        int result = int_distribution(crypto_random_generator);
        actual_distribution[result]++;
    }

    for(int i = 0; i < 10; i++) {
        std::cout << actual_distribution[i] << " ";
    }

    return 0;
}
```

`std::random_device` is used in the same way as a pseudo random value generator is used.

However `std::random_device` **may be implemented in terms of an implementation-defined pseudo-random number engine** if a non-deterministic source (e.g. a hardware device) isn't available to the implementation.

Detecting such implementations should be possible via the `entropy` member function (which return zero when the generator is completely deterministic), but many popular libraries (both GCC's `libstdc++` and LLVM's `libc++`) always return zero, even when they're using high-quality external randomness.

Section 65.2: Generating a pseudo-random number

A pseudo-random number generator generates values that can be guessed based on previously generated values. In other words: it is deterministic. Do not use a pseudo-random number generator in situations where a true random number is required.

```
#include <iostream>
#include <random>

int main()
{
    std::default_random_engine pseudo_random_generator;
    std::uniform_int_distribution<int> int_distribution(0, 9);

    int actual_distribution[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

    for(int i = 0; i < 10000; i++) {
        int result = int_distribution(pseudo_random_generator);
        actual_distribution[result]++;
    }
}
```

```

}

for(int i = 0; i <= 9; i++) {
    std::cout << 实际分布[i] << " ";
}

return 0;
}

```

这段代码创建了一个随机数生成器，以及一个在区间[0,9]内均匀生成整数的分布。然后统计每个结果生成的次数。

模板参数 `std::uniform_int_distribution<T>` 指定了应生成的整数类型。使用 `std::uniform_real_distribution<T>` 来生成浮点数或双精度数。

第65.3节：为多个分布使用生成器

随机数生成器可以（且应该）用于多个分布。

```

#include <iostream>
#include <random>

int main()
{
    std::default_random_engine 伪随机生成器;
    std::uniform_int_distribution<int> 整数分布(0, 9);
    std::uniform_real_distribution<float> 浮点分布(0.0, 1.0);
    std::discrete_distribution<int> 作弊骰子({1,1,1,1,1,100});

    std::cout << int_distribution(pseudo_random_generator) << std::endl;
    std::cout << float_distribution(pseudo_random_generator) << std::endl;
    std::cout << (rigged_dice(pseudo_random_generator) + 1) << std::endl;

    return 0;
}

```

在此示例中，仅定义了一个生成器。随后它被用来在三种不同的分布中生成随机值。`rigged_dice` 分布将生成一个介于0到5之间的值，但几乎总是生成 5，因为生成 5 的概率是 100 / 105。

```

}

for(int i = 0; i <= 9; i++) {
    std::cout << actual_distribution[i] << " ";
}

return 0;
}

```

This code creates a random number generator, and a distribution that generates integers in the range [0,9] with equal likelihood. It then counts how many times each result was generated.

The template parameter of `std::uniform_int_distribution<T>` specifies the type of integer that should be generated. Use `std::uniform_real_distribution<T>` to generate floats or doubles.

Section 65.3: Using the generator for multiple distributions

The random number generator can (and should) be used for multiple distributions.

```

#include <iostream>
#include <random>

int main()
{
    std::default_random_engine pseudo_random_generator;
    std::uniform_int_distribution<int> int_distribution(0, 9);
    std::uniform_real_distribution<float> float_distribution(0.0, 1.0);
    std::discrete_distribution<int> rigged_dice({1,1,1,1,1,100});

    std::cout << int_distribution(pseudo_random_generator) << std::endl;
    std::cout << float_distribution(pseudo_random_generator) << std::endl;
    std::cout << (rigged_dice(pseudo_random_generator) + 1) << std::endl;

    return 0;
}

```

In this example, only one generator is defined. It is subsequently used to generate a random value in three different distributions. The `rigged_dice` distribution will generate a value between 0 and 5, but almost always generates a 5, because the chance to generate a 5 is 100 / 105.

第66章：使用 <chrono>

头文件的日期和时间

第66.1节：使用 <chrono> 测量时间

system_clock 可用于测量程序执行某部分期间经过的时间。

```
版本 = c++11
#include <iostream>
#include <chrono>
#include <thread>

int main() {
    auto start = std::chrono::system_clock::now(); // 该变量和“end”的类型是
std::chrono::time_point
    { // 测试代码
std::this_thread::sleep_for(std::chrono::seconds(2));
}
    auto end = std::chrono::system_clock::now();

std::chrono::duration<double> elapsed = end - start;
    std::cout << "Elapsed time: " << elapsed.count() << "s";
}
```

在此示例中，sleep_for 用于使活动线程休眠一段以
std::chrono::seconds 为单位计量的时间，但大括号内的代码可以是任何需要一定时间执行的函数调用。

第66.2节：计算两个日期之间的天数

本示例展示如何计算两个日期之间的天数。日期由年/月/日指定，另外还包括时/分/秒。

程序计算自2000年以来的天数。

```
#include <iostream>
#include <string>
#include <chrono>
#include <ctime>

/**
* 从原始日期创建一个 std::tm 结构体。
*
* |param year (必须大于等于1900)
* |param month 从一月开始的月份 - [1, 12]
* |param day 当月的天数 - [1, 31]
* |param minutes 小时后的分钟数 - [0, 59]
* |param seconds 分钟后的秒数 - [0, 61] (直到C++11) / [0, 60] (自C++11起)
*
* 基于 http://en.cppreference.com/w/cpp/chrono/c/tm
*/
std::tm CreateTmStruct(int year, int month, int day, int hour, int minutes, int seconds) {
    struct tm tm_ret = {0};

    tm_ret.tm_sec = seconds;
    tm_ret.tm_min = minutes;
    tm_ret.tm_hour = hour;
    tm_ret.tm_mday = day;
```

Chapter 66: Date and time using <chrono> header

Section 66.1: Measuring time using <chrono>

The system_clock can be used to measure the time elapsed during some part of a program's execution.

```
Version = c++11
#include <iostream>
#include <chrono>
#include <thread>

int main() {
    auto start = std::chrono::system_clock::now(); // This and "end"'s type is
std::chrono::time_point
    { // The code to test
        std::this_thread::sleep_for(std::chrono::seconds(2));
    }
    auto end = std::chrono::system_clock::now();

    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Elapsed time: " << elapsed.count() << "s";
}
```

In this example, sleep_for was used to make the active thread sleep for a time period measured in std::chrono::seconds, but the code between braces could be any function call that takes some time to execute.

Section 66.2: Find number of days between two dates

This example shows how to find number of days between two dates. A date is specified by year/month/day of month, and additionally hour/minute/second.

Program calculates number of days in years since 2000.

```
#include <iostream>
#include <string>
#include <chrono>
#include <ctime>

/**
* Creates a std::tm structure from raw date.
*
* |param year (must be 1900 or greater)
* |param month months since January - [1, 12]
* |param day day of the month - [1, 31]
* |param minutes minutes after the hour - [0, 59]
* |param seconds seconds after the minute - [0, 61](until C++11) / [0, 60] (since C++11)
*
* Based on http://en.cppreference.com/w/cpp/chrono/c/tm
*/
std::tm CreateTmStruct(int year, int month, int day, int hour, int minutes, int seconds) {
    struct tm tm_ret = {0};

    tm_ret.tm_sec = seconds;
    tm_ret.tm_min = minutes;
    tm_ret.tm_hour = hour;
    tm_ret.tm_mday = day;
```

```

tm_ret.tm_mon = month - 1;
tm_ret.tm_year = year - 1900;

return tm_ret;
}

int get_days_in_year(int year) {

using namespace std;
using namespace std::chrono;

// 我们希望结果以天为单位
typedef duration<int, ratio_multiply<hours::period, ratio<24> >::type> days;

// 创建起始时间段
std::tm tm_start = CreateTmStruct(year, 1, 1, 0, 0, 0);
auto tms = system_clock::from_time_t(std::mktime(&tm_start));

// 创建结束时间段
std::tm tm_end = CreateTmStruct(year + 1, 1, 1, 0, 0, 0);
auto tme = system_clock::from_time_t(std::mktime(&tm_end));

// 计算这两个日期之间的时间间隔
auto diff_in_days = std::chrono::duration_cast<days>(tme - tms);

return diff_in_days.count();
}

int main()
{
    for (int year = 2000; year <= 2016; ++year)
        std::cout << "在 " << year << " 年有 " << get_days_in_year(year) << " 天" << "";
}

```

```

tm_ret.tm_mon = month - 1;
tm_ret.tm_year = year - 1900;

return tm_ret;
}

int get_days_in_year(int year) {

using namespace std;
using namespace std::chrono;

// We want results to be in days
typedef duration<int, ratio_multiply<hours::period, ratio<24> >::type> days;

// Create start time span
std::tm tm_start = CreateTmStruct(year, 1, 1, 0, 0, 0);
auto tms = system_clock::from_time_t(std::mktime(&tm_start));

// Create end time span
std::tm tm_end = CreateTmStruct(year + 1, 1, 1, 0, 0, 0);
auto tme = system_clock::from_time_t(std::mktime(&tm_end));

// Calculate time duration between those two dates
auto diff_in_days = std::chrono::duration_cast<days>(tme - tms);

return diff_in_days.count();
}

int main()
{
    for (int year = 2000; year <= 2016; ++year)
        std::cout << "There are " << get_days_in_year(year) << " days in " << year << "\n";
}

```

第67章：排序

第67.1节：排序和序列容器

`std::sort`, 定义在标准库头文件`algorithm`中，是一个用于排序由一对迭代器定义的值范围的标准库算法。`std::sort`将一个用于比较两个值的函数对象作为最后一个参数；这就是它确定顺序的方式。注意，`std::sort`不是稳定排序。

比较函数必须对元素施加严格的弱序关系。简单的“小于”（或“大于”）比较即可。

具有随机访问迭代器的容器可以使用`std::sort`算法进行排序：

```
版本 ≥ C++11
#include <vector>
#include <algorithm>

std::vector<int> MyVector = {3, 1, 2}

//默认的<
std::sort(MyVector.begin(), MyVector.end());
```

`std::sort`要求其迭代器为随机访问迭代器。序列容器`std::list`和`std::forward_list`（需要C++11）不提供随机访问迭代器，因此不能与`std::sort`一起使用。然而，它们有自己的`sort`成员函数，使用适用于其自身迭代器类型的排序算法。

```
版本 ≥ C++11
#include <list>
#include <algorithm>

std::list<int> myList = {3, 1, 2}

//默认的<
//仅限整个列表。
myList.sort();
```

它们的成员`sort`函数总是对整个列表进行排序，因此不能对元素的子范围进行排序。然而，由于`list`和`forward_list`具有快速的拼接操作，你可以将要排序的元素从列表中提取出来，排序后，再高效地放回原处，方法如下：

```
void sort_sublist(std::list<int>& mylist, std::list<int>::const_iterator start,
std::list<int>::const_iterator end) {
    //提取并排序由start和end迭代器表示的半开子范围
    std::list<int> tmp;
    tmp.splice(tmp.begin(), list, start, end);
    tmp.sort();
    //将范围重新插入到我们提取它的位置
    list.splice(end, tmp);
}
```

第67.2节：使用`std::map`进行排序（升序和降序）

此示例使用`map`按`key`的升序对元素进行排序。你可以使用任何类型，包括类，

Chapter 67: Sorting

Section 67.1: Sorting and sequence containers

`std::sort`, found in the standard library header `algorithm`, is a standard library algorithm for sorting a range of values, defined by a pair of iterators. `std::sort` takes as the last parameter a functor used to compare two values; this is how it determines the order. Note that `std::sort` is not stable.

The comparison function *must* impose a Strict, Weak Ordering on the elements. A simple less-than (or greater-than) comparison will suffice.

A container with random-access iterators can be sorted using the `std::sort` algorithm:

```
Version ≥ C++11
#include <vector>
#include <algorithm>

std::vector<int> MyVector = {3, 1, 2}

//Default comparison of <
std::sort(MyVector.begin(), MyVector.end());
```

`std::sort` requires that its iterators are random access iterators. The sequence containers `std::list` and `std::forward_list` (requiring C++11) do not provide random access iterators, so they cannot be used with `std::sort`. However, they do have `sort` member functions which implement a sorting algorithm that works with their own iterator types.

```
Version ≥ C++11
#include <list>
#include <algorithm>

std::list<int> myList = {3, 1, 2}

//Default comparison of <
//Whole list only.
myList.sort();
```

Their member `sort` functions always sort the entire list, so they cannot sort a sub-range of elements. However, since `list` and `forward_list` have fast splicing operations, you could extract the elements to be sorted from the list, sort them, then stuff them back where they were quite efficiently like this:

```
void sort_sublist(std::list<int>& mylist, std::list<int>::const_iterator start,
std::list<int>::const_iterator end) {
    //extract and sort half-open sub range denoted by start and end iterator
    std::list<int> tmp;
    tmp.splice(tmp.begin(), list, start, end);
    tmp.sort();
    //re-insert range at the point we extracted it from
    list.splice(end, tmp);
}
```

Section 67.2: sorting with `std::map` (ascending and descending)

This example sorts elements in **ascending** order of a **key** using a `map`. You can use any type, including class,

替代下面示例中的std::string。

```
#include <iostream>
#include <utility>
#include <map>

int main()
{
    std::map<double, std::string> sorted_map;
    // 根据行星大小排序名称
    sorted_map.insert(std::make_pair(0.3829, "水星"));
    sorted_map.insert(std::make_pair(0.9499, "金星"));
    sorted_map.insert(std::make_pair(1, "地球"));
    sorted_map.insert(std::make_pair(0.532, "火星"));
    sorted_map.insert(std::make_pair(10.97, "木星"));
    sorted_map.insert(std::make_pair(9.14, "土星"));
    sorted_map.insert(std::make_pair(3.981, "天王星"));
    sorted_map.insert(std::make_pair(3.865, "海王星"));

    for (auto const& entry: sorted_map)
    {
        std::cout << entry.second << " (" << entry.first << " 倍地球半径) " << ";"
    }
}
```

输出：

```
水星 (0.3829 倍地球半径)
火星 (0.532 倍地球半径)
金星 (0.9499 倍地球半径)
地球 (1 倍地球半径)
海王星 (3.865 倍地球半径)
天王星 (3.981 倍地球半径)
土星 (9.14 倍地球半径)
木星 (10.97 倍地球半径)
```

如果可能存在键相同的条目，请使用multimap代替map（如下面的示例所示）。

要以降序方式排序元素，请使用合适的比较函数对象（std::greater<>）声明映射：

```
#include <iostream>
#include <utility>
#include <map>

int main()
{
    std::multimap<int, std::string, std::greater<int>> sorted_map;
    // 根据腿的数量降序排序动物名称
    sorted_map.insert(std::make_pair(6, "虫子"));
    sorted_map.insert(std::make_pair(4, "猫"));
    sorted_map.insert(std::make_pair(100, "蜈蚣"));
    sorted_map.insert(std::make_pair(2, "鸡"));
    sorted_map.insert(std::make_pair(0, "鱼"));
    sorted_map.insert(std::make_pair(4, "马"));
    sorted_map.insert(std::make_pair(8, "蜘蛛"));

    for (auto const& entry: sorted_map)
    {
        std::cout << entry.second << " (有 " << entry.first << " 条腿) " << ";"
    }
}
```

instead of std::string, in the example below.

```
#include <iostream>
#include <utility>
#include <map>

int main()
{
    std::map<double, std::string> sorted_map;
    // Sort the names of the planets according to their size
    sorted_map.insert(std::make_pair(0.3829, "Mercury"));
    sorted_map.insert(std::make_pair(0.9499, "Venus"));
    sorted_map.insert(std::make_pair(1, "Earth"));
    sorted_map.insert(std::make_pair(0.532, "Mars"));
    sorted_map.insert(std::make_pair(10.97, "Jupiter"));
    sorted_map.insert(std::make_pair(9.14, "Saturn"));
    sorted_map.insert(std::make_pair(3.981, "Uranus"));
    sorted_map.insert(std::make_pair(3.865, "Neptune"));

    for (auto const& entry: sorted_map)
    {
        std::cout << entry.second << " (" << entry.first << " of Earth's radius) " << '\n';
    }
}
```

Output:

```
Mercury (0.3829 of Earth's radius)
Mars (0.532 of Earth's radius)
Venus (0.9499 of Earth's radius)
Earth (1 of Earth's radius)
Neptune (3.865 of Earth's radius)
Uranus (3.981 of Earth's radius)
Saturn (9.14 of Earth's radius)
Jupiter (10.97 of Earth's radius)
```

If entries with equal keys are possible, use multimap instead of map (like in the following example).

To sort elements in **descending** manner, declare the map with a proper comparison functor (std::greater<>):

```
#include <iostream>
#include <utility>
#include <map>

int main()
{
    std::multimap<int, std::string, std::greater<int>> sorted_map;
    // Sort the names of animals in descending order of the number of legs
    sorted_map.insert(std::make_pair(6, "bug"));
    sorted_map.insert(std::make_pair(4, "cat"));
    sorted_map.insert(std::make_pair(100, "centipede"));
    sorted_map.insert(std::make_pair(2, "chicken"));
    sorted_map.insert(std::make_pair(0, "fish"));
    sorted_map.insert(std::make_pair(4, "horse"));
    sorted_map.insert(std::make_pair(8, "spider"));

    for (auto const& entry: sorted_map)
    {
        std::cout << entry.second << " (has " << entry.first << " legs) " << '\n';
    }
}
```

```
}
```

输出

```
蜈蚣 (有100条腿)
蜘蛛 (有8条腿)
虫子 (有6条腿)
猫 (有4条腿)
马 (有4条腿)
鸡 (有2条腿)
鱼 (有0条腿)
```

```
}
```

Output

```
centipede (has 100 legs)
spider (has 8 legs)
bug (has 6 legs)
cat (has 4 legs)
horse (has 4 legs)
chicken (has 2 legs)
fish (has 0 legs)
```

第67.3节：通过重载的less运算符对序列容器进行排序

如果没有传入排序函数，`std::sort` 将通过调用元素对的 `operator<` 来对元素进行排序，该运算符必须返回一个上下文可转换为 `bool`（或直接为 `bool`）的类型。基本类型（整数、浮点数、指针等）已经内置了比较运算符。

我们可以重载该运算符，使默认的 `sort` 调用能够作用于用户自定义类型。

```
// 包含序列容器
#include <vector>
#include <deque>
#include <list>

// 引入排序算法
#include <algorithm>

class Base {
    公共:

        // 构造函数，将变量设置为v的值
        Base(int v): variable(v) {}

        // 使用variable提供全序的less运算符
        // `this` 总是表示比较的左侧。
        bool operator<(const Base &b) const {
            return this->variable < b.variable;
        }

        int variable;
};

int main() {
    std::vector <Base> vector;
    std::deque <Base> deque;
    std::list <Base> list;

        // 创建2个元素用于排序
        Base a(10);
        Base b(5);

        // 将它们插入容器尾部
        vector.push_back(a);
        vector.push_back(b);
}
```

Section 67.3: Sorting sequence containers by overloaded less operator

If no ordering function is passed, `std::sort` will order the elements by calling `operator<` on pairs of elements, which must return a type contextually convertible to `bool` (or just `bool`). Basic types (integers, floats, pointers etc) have already built-in comparison operators.

We can overload this operator to make the default `sort` call work on user-defined types.

```
// Include sequence containers
#include <vector>
#include <deque>
#include <list>

// Insert sorting algorithm
#include <algorithm>

class Base {
    public:

        // Constructor that sets variable to the value of v
        Base(int v): variable(v) {}

        // Use variable to provide total order operator less
        // `this` always represents the left-hand side of the compare.
        bool operator<(const Base &b) const {
            return this->variable < b.variable;
        }

        int variable;
};

int main() {
    std::vector <Base> vector;
    std::deque <Base> deque;
    std::list <Base> list;

        // Create 2 elements to sort
        Base a(10);
        Base b(5);

        // Insert them into backs of containers
        vector.push_back(a);
        vector.push_back(b);
}
```

```

deque.push_back(a);
deque.push_back(b);

list.push_back(a);
list.push_back(b);

// 现在使用 operator<(const Base &b) 函数排序数据
std::sort(vector.begin(), vector.end());
std::sort(deque.begin(), deque.end());
// 由于设计原因, list 必须用不同方式排序
list.sort();

return 0;
}

```

第67.4节：使用比较函数排序序列容器

```

// 包含序列容器
#include <vector>
#include <deque>
#include <list>

// 插入排序算法
#include <algorithm>

class Base {
public:

    // 构造函数, 将变量设置为v的值
    Base(int v): variable(v) {
    }

    int variable;
};

bool compare(const Base &a, const Base &b) {
    return a.variable < b.variable;
}

int main() {
    std::vector <Base> vector;
    std::deque <Base> deque;
    std::list <Base> list;

    // 创建2个元素用于排序
    Base a(10);
    Base b(5);

    // 将它们插入到容器的尾部
    vector.push_back(a);
    vector.push_back(b);

    deque.push_back(a);
    deque.push_back(b);

    list.push_back(a);
    list.push_back(b);

    // 现在使用比较函数对数据进行排序
}

```

```

deque.push_back(a);
deque.push_back(b);

list.push_back(a);
list.push_back(b);

// Now sort data using operator<(const Base &b) function
std::sort(vector.begin(), vector.end());
std::sort(deque.begin(), deque.end());
// List must be sorted differently due to its design
list.sort();

return 0;
}

```

Section 67.4: Sorting sequence containers using compare function

```

// Include sequence containers
#include <vector>
#include <deque>
#include <list>

// Insert sorting algorithm
#include <algorithm>

class Base {
public:

    // Constructor that set variable to the value of v
    Base(int v): variable(v) {
    }

    int variable;
};

bool compare(const Base &a, const Base &b) {
    return a.variable < b.variable;
}

int main() {
    std::vector <Base> vector;
    std::deque <Base> deque;
    std::list <Base> list;

    // Create 2 elements to sort
    Base a(10);
    Base b(5);

    // Insert them into backs of containers
    vector.push_back(a);
    vector.push_back(b);

    deque.push_back(a);
    deque.push_back(b);

    list.push_back(a);
    list.push_back(b);

    // Now sort data using comparing function
}

```

```

std::sort(vector.begin(), vector.end(), compare);
std::sort(deque.begin(), deque.end(), compare);
list.sort(compare);

return 0;
}

```

第67.5节：使用lambda表达式排序序列容器 (C++11)

版本 ≥ C++11

```

// 包含序列容器
#include <vector>
#include <deque>
#include <list>
#include <array>
#include <forward_list>

// 包含排序算法
#include <algorithm>

class Base {
public:

    // 构造函数，将变量设置为v的值
    Base(int v): variable(v) {
    }

    int variable;
};

int main() {
    // 创建2个元素用于排序
    Base a(10);
    Base b(5);

    // 我们使用的是C++11，所以使用初始化列表来插入元素。
    std::vector<Base> vector = {a, b};
    std::deque<Base> deque = {a, b};
    std::list<Base> list = {a, b};
    std::array<Base, 2> array = {a, b};
    std::forward_list<Base> flist = {a, b};

    // 我们可以使用内联lambda表达式对数据进行排序
    std::sort(std::begin(vector), std::end(vector),
        [] (const Base &a, const Base &b) { return a.variable < b.variable;});

    // 我们也可以传递一个 lambda 对象作为比较器
    // 并多次重用该 lambda
    auto compare = [] (const Base &a, const Base &b) {
        return a.variable < b.variable;};
    std::sort(std::begin(deque), std::end(deque), compare);
    std::sort(std::begin(array), std::end(array), compare);
    list.sort(compare);
    flist.sort(compare);

    return 0;
}

```

```

std::sort(vector.begin(), vector.end(), compare);
std::sort(deque.begin(), deque.end(), compare);
list.sort(compare);

return 0;
}

```

Section 67.5: Sorting sequence containers using lambda expressions (C++11)

Version ≥ C++11

```

// Include sequence containers
#include <vector>
#include <deque>
#include <list>
#include <array>
#include <forward_list>

// Include sorting algorithm
#include <algorithm>

class Base {
public:

    // Constructor that set variable to the value of v
    Base(int v): variable(v) {
    }

    int variable;
};

int main() {
    // Create 2 elements to sort
    Base a(10);
    Base b(5);

    // We're using C++11, so let's use initializer lists to insert items.
    std::vector<Base> vector = {a, b};
    std::deque<Base> deque = {a, b};
    std::list<Base> list = {a, b};
    std::array<Base, 2> array = {a, b};
    std::forward_list<Base> flist = {a, b};

    // We can sort data using an inline lambda expression
    std::sort(std::begin(vector), std::end(vector),
        [] (const Base &a, const Base &b) { return a.variable < b.variable;});

    // We can also pass a lambda object as the comparator
    // and reuse the lambda multiple times
    auto compare = [] (const Base &a, const Base &b) {
        return a.variable < b.variable;};
    std::sort(std::begin(deque), std::end(deque), compare);
    std::sort(std::begin(array), std::end(array), compare);
    list.sort(compare);
    flist.sort(compare);

    return 0;
}

```

第67.6节：排序内置数组

`sort` 算法对由两个迭代器定义的序列进行排序。这足以对内置（也称为 C 风格）数组进行排序。

```
版本 ≥ C++11
int arr1[] = {36, 24, 42, 60, 59};

// 按升序排序数字
sort(std::begin(arr1), std::end(arr1));

// 按降序排序数字
sort(std::begin(arr1), std::end(arr1), std::greater<int>());
```

在 C++11 之前，数组末尾必须通过数组大小“计算”得出：

```
版本 < C++11
// 使用硬编码的数组大小
sort(arr1, arr1 + 5);

// 或者，使用表达式
const size_t arr1_size = sizeof(arr1) / sizeof(*arr1);
sort(arr1, arr1 + arr1_size);
```

第67.7节：使用指定顺序对序列容器进行排序

如果容器中的值已经重载了某些运算符，`std::sort` 可以配合特化的函数对象用于升序或降序排序：

```
版本 ≥ C++11
#include <vector>
#include <algorithm>
#include <functional>

std::vector<int> v = {5,1,2,4,3};

// 升序排序 (1,2,3,4,5)
std::sort(v.begin(), v.end(), std::less<int>());

// 或者直接：
std::sort(v.begin(), v.end());

// 降序排序 (5,4,3,2,1)
std::sort(v.begin(), v.end(), std::greater<int>());

// 或者直接：
std::sort(v.rbegin(), v.rend());
版本 ≥ C++14
```

在 C++14 中，我们不需要为比较函数对象提供模板参数，而是让对象根据传入的参数自动推断：

```
std::sort(v.begin(), v.end(), std::less<>()); // 升序
std::sort(v.begin(), v.end(), std::greater<>()); // 降序
```

Section 67.6: Sorting built-in arrays

The `sort` algorithm sorts a sequence defined by two iterators. This is enough to sort a built-in (also known as c-style) array.

```
Version ≥ C++11
int arr1[] = {36, 24, 42, 60, 59};

// sort numbers in ascending order
sort(std::begin(arr1), std::end(arr1));

// sort numbers in descending order
sort(std::begin(arr1), std::end(arr1), std::greater<int>());
```

Prior to C++11, end of array had to be "calculated" using the size of the array:

```
Version < C++11
// Use a hard-coded number for array size
sort(arr1, arr1 + 5);

// Alternatively, use an expression
const size_t arr1_size = sizeof(arr1) / sizeof(*arr1);
sort(arr1, arr1 + arr1_size);
```

Section 67.7: Sorting sequence containers with specified ordering

If the values in a container have certain operators already overloaded, `std::sort` can be used with specialized functors to sort in either ascending or descending order:

```
Version ≥ C++11
#include <vector>
#include <algorithm>
#include <functional>

std::vector<int> v = {5,1,2,4,3};

// sort in ascending order (1,2,3,4,5)
std::sort(v.begin(), v.end(), std::less<int>());

// Or just:
std::sort(v.begin(), v.end());

// sort in descending order (5,4,3,2,1)
std::sort(v.begin(), v.end(), std::greater<int>());

// Or just:
std::sort(v.rbegin(), v.rend());
Version ≥ C++14
```

In C++14, we don't need to provide the template argument for the comparison function objects and instead let the object deduce based on what it gets passed in:

```
std::sort(v.begin(), v.end(), std::less<>()); // ascending order
std::sort(v.begin(), v.end(), std::greater<>()); // descending order
```

第68章：枚举

第68.1节：枚举的迭代

没有内置的方法来遍历枚举。

但有几种方法

- 对于仅包含连续值的枚举：

```
枚举 E {
    Begin,
    E1 = Begin,
    E2,
    // ...
    En,
    End
};

对于 (E e = E::Begin; e != E::End; ++e) {
    // 使用 e 执行操作
}
```

版本 \geq C++11

对于枚举类，需要实现operator ++：

```
E& 操作符 ++ (E& e)
{
    if (e == E::End) {
        throw std::out_of_range("for E& operator ++ (E&)");
    }
    e = E(static_cast<std::underlying_type<E>::type>(e) + 1);
    return e;
}
```

- 使用容器作为 std::vector

```
枚举 E {
    E1 = 4,
    E2 = 8,
    // ...
    En
};

std::vector<E> build_all_E()
{
    const E all[] = {E1, E2, /*...*/ En};

    return std::vector<E>(all, all + sizeof(all) / sizeof(E));
}

std::vector<E> all_E = build_all_E();
```

然后

```
for (std::vector<E>::const_iterator it = all_E.begin(); it != all_E.end(); ++it) {
```

Chapter 68: Enumeration

Section 68.1: Iteration over an enum

There is no built-in to iterate over enumeration.

But there are several ways

- for enum with only consecutive values:

```
enum E {
    Begin,
    E1 = Begin,
    E2,
    // ...
    En,
    End
};

for (E e = E::Begin; e != E::End; ++e) {
    // Do job with e
}
```

Version \geq C++11

with enum class, operator ++ has to be implemented:

```
E& operator ++ (E& e)
{
    if (e == E::End) {
        throw std::out_of_range("for E& operator ++ (E&)");
    }
    e = E(static_cast<std::underlying_type<E>::type>(e) + 1);
    return e;
}
```

- using a container as std::vector

```
enum E {
    E1 = 4,
    E2 = 8,
    // ...
    En
};

std::vector<E> build_all_E()
{
    const E all[] = {E1, E2, /*...*/ En};

    return std::vector<E>(all, all + sizeof(all) / sizeof(E));
}

std::vector<E> all_E = build_all_E();
```

and then

```
for (std::vector<E>::const_iterator it = all_E.begin(); it != all_E.end(); ++it) {
```

```
E e = *it;
// 使用 e 执行操作;
}
```

版本 \geq C++11

- 或者使用 `std::initializer_list` 和更简单的语法：

```
枚举 E {
    E1 = 4,
    E2 = 8,
    // ...
    En
};

constexpr std::initializer_list<E> all_E = {E1, E2, /*...*/ En};
```

然后

```
for (auto e : all_E) {
    // 使用 e 执行操作
}
```

第 68.2 节：作用域枚举

C++11 引入了所谓的 作用域枚举。这些枚举的成员必须使用 `enumname::membername` 进行限定。作用域枚举使用 `enum class` 语法声明。例如，存储彩虹颜色：

```
enum class rainbow {
    RED,
    ORANGE,
    YELLOW,
    GREEN,
    BLUE,
    INDIGO,
    VIOLET
};
```

访问特定颜色：

```
rainbow r = rainbow::INDIGO;
```

枚举类 (`enum class`) 不能隐式转换为 `int` 类型，必须进行强制转换。因此 `int x = rainbow::RED` 是无效的。

作用域枚举 (Scoped enums) 还允许你指定底层类型，即用于表示成员的类型。默认情况下是 `int`。在井字棋游戏中，你可以将棋子存储为

```
enum class piece : char {
    EMPTY = '\0',
    X = 'X',
    O = 'O',
}
```

如你所见，`enum` 的最后一个成员后可以有一个尾随逗号。

```
E e = *it;
// Do job with e;
}
```

Version \geq C++11

- or `std::initializer_list` and a simpler syntax:

```
enum E {
    E1 = 4,
    E2 = 8,
    // ...
    En
};

constexpr std::initializer_list<E> all_E = {E1, E2, /*...*/ En};
```

and then

```
for (auto e : all_E) {
    // Do job with e
}
```

Section 68.2: Scoped enums

C++11 introduces what are known as *scoped enums*. These are enumerations whose members must be qualified with `enumname::membername`. Scoped enums are declared using the `enum class` syntax. For example, to store the colors in a rainbow:

```
enum class rainbow {
    RED,
    ORANGE,
    YELLOW,
    GREEN,
    BLUE,
    INDIGO,
    VIOLET
};
```

To access a specific color:

```
rainbow r = rainbow::INDIGO;
```

`enum classes` cannot be implicitly converted to `ints` without a cast. So `int x = rainbow::RED` is invalid.

Scoped enums also allow you to specify the *underlying type*, which is the type used to represent a member. By default it is `int`. In a Tic-Tac-Toe game, you may store the piece as

```
enum class piece : char {
    EMPTY = '\0',
    X = 'X',
    O = 'O',
}
```

As you may notice, `enums` can have a trailing comma after the last member.

第68.3节：C++11中的枚举前置声明

作用域枚举：

```
...
enum class Status; // 前向声明
Status doWork(); // 使用前向声明
...
enum class Status { Invalid, Success, Fail };
Status doWork() // 实现时需要完整声明
{
    return Status::Success;
}
```

无作用域枚举：

```
...
enum Status: int; // 前向声明，需显式指定类型
Status doWork(); // 使用前向声明
...
enum Status: int{ Invalid=0, Success, Fail }; // 必须与前向声明的类型匹配
static_assert( Success == 1 );
```

详细的多文件示例见此处：[盲目水果商示例](#)

第68.4节：基本枚举声明

标准枚举允许用户为一组整数声明一个有用的名字。这些名称统称为枚举量。枚举及其相关的枚举量定义如下：

```
enum myEnum
{
    枚举名1,
    枚举名2,
};
```

枚举是一种类型，这种类型与所有其他类型不同。在本例中，该类型的名称是myEnum。
该类型的对象预期将取枚举中的枚举值。

在枚举中声明的枚举值是该枚举类型的常量值。虽然枚举值是在类型内声明的，但访问名称时不需要使用作用域运算符::。
因此，第一个枚举值的名称是enumName1。

版本 ≥ C++11

作用域运算符可以选择性地用于访问枚举中的枚举值。因此，enumName1也可以写作myEnum::enumName1。

枚举值被赋予从0开始、每个枚举值递增1的整数值。因此，在上述情况下，enumName1的值为0，而enumName2的值为1。

枚举值也可以由用户指定特定的值；该值必须是整型常量表达式。
未显式提供值的枚举值，其值将被设置为前一个枚举值的值加1。

```
enum myEnum
```

Section 68.3: Enum forward declaration in C++11

Scoped enumerations:

```
...
enum class Status; // Forward declaration
Status doWork(); // Use the forward declaration
...
enum class Status { Invalid, Success, Fail };
Status doWork() // Full declaration required for implementation
{
    return Status::Success;
}
```

Unscoped enumerations:

```
...
enum Status: int; // Forward declaration, explicit type required
Status doWork(); // Use the forward declaration
...
enum Status: int{ Invalid=0, Success, Fail }; // Must match forward declare type
static_assert( Success == 1 );
```

An in-depth multi-file example can be found here: [Blind fruit merchant example](#)

Section 68.4: Basic Enumeration Declaration

Standard enumerations allow users to declare a useful name for a set of integers. The names are collectively referred to as enumerators. An enumeration and its associated enumerators are defined as follows:

```
enum myEnum
{
    enumName1,
    enumName2,
};
```

An enumeration is a *type*, one which is distinct from all other types. In this case, the name of this type is myEnum.
Objects of this type are expected to assume the value of an enumerator within the enumeration.

The enumerators declared within the enumeration are constant values of the type of the enumeration. Though the enumerators are declared within the type, the scope operator :: is not needed to access the name. So the name of the first enumerator is enumName1.

Version ≥ C++11

The scope operator can be optionally used to access an enumerator within an enumeration. So enumName1 can also be spelled myEnum::enumName1.

Enumerators are assigned integer values starting from 0 and increasing by 1 for each enumerator in an enumeration. So in the above case, enumName1 has the value 0, while enumName2 has the value 1.

Enumerators can also be assigned a specific value by the user; this value must be an integral constant expression.
Enumerators whose values are not explicitly provided will have their value set to the value of the previous enumerator + 1.

```
enum myEnum
```

```
{
enumName1 = 1, // 值为1
enumName2 = 2, // 值为2
enumName3, // 值为3, 前一个值+1
enumName4 = 7, // 值为7
enumName5, // 值将是 8
enumName6 = 5, // 值将是 5, 允许向后赋值
enumName7 = 3, // 值将是 3, 允许重复使用数字
enumName8 = enumName4 + 2, // 值将是 9, 允许使用之前的枚举并进行调整
};
```

第 68.5 节：switch 语句中的枚举

枚举常用于 switch 语句，因此它们常出现在状态机中。实际上，switch 语句结合枚举的一个有用特性是，如果 switch 中没有包含 default 语句，且枚举的所有值并未被使用，编译器会发出警告。

```
enum State {
    start,
    middle,
    end
};

...

switch(myState) {
    case start:
        ...
    case middle:
        ...
} // 警告：switch 中未处理枚举值 'end' [-Wswitch]
```

```
{
enumName1 = 1, // value will be 1
enumName2 = 2, // value will be 2
enumName3, // value will be 3, previous value + 1
enumName4 = 7, // value will be 7
enumName5, // value will be 8
enumName6 = 5, // value will be 5, legal to go backwards
enumName7 = 3, // value will be 3, legal to reuse numbers
enumName8 = enumName4 + 2, // value will be 9, legal to take prior enums and adjust them
};
```

Section 68.5: Enumeration in switch statements

A common use for enumerators is for switch statements and so they commonly appear in state machines. In fact a useful feature of switch statements with enumerations is that if no default statement is included for the switch, and not all values of the enum have been utilized, the compiler will issue a warning.

```
enum State {
    start,
    middle,
    end
};

...

switch(myState) {
    case start:
        ...
    case middle:
        ...
} // warning: enumeration value 'end' not handled in switch [-Wswitch]
```

第69章：迭代

第69.1节：break

跳出最近的包围循环或 switch 语句。

```
// 将数字打印到文件中，每行一个
for (const int num : num_list) {
    errno = 0;
    fprintf(file, "%d", num);if (errno
    == ENOSPC) {
        fprintf(stderr, "设备空间不足；输出将被截断");break;
    }
}
```

第69.2节：continue

跳转到最小包围循环的末尾。

```
int sum = 0;
for (int i = 0; i < N; i++) {
    int x;
    std::cin >> x;
    if (x < 0) continue;
    sum += x;
    // 等价于: if (x >= 0) sum += x;
}
```

第69.3节：do

介绍do-while循环。

```
// 从标准输入获取下一个非空白字符
char read_char() {
    char c;
    do {
        c = getchar();
    } while (isspace(c));
    return c;
}
```

第69.4节：while

介绍了while循环。

```
int i = 0;
// 打印10个星号
while (i < 10) {
    putchar('*');
    i++;
}
```

Chapter 69: Iteration

Section 69.1: break

Jumps out of the nearest enclosing loop or switch statement.

```
// print the numbers to a file, one per line
for (const int num : num_list) {
    errno = 0;
    fprintf(file, "%d\n", num);
    if (errno == ENOSPC) {
        fprintf(stderr, "no space left on device; output will be truncated\n");
        break;
    }
}
```

Section 69.2: continue

Jumps to the end of the smallest enclosing loop.

```
int sum = 0;
for (int i = 0; i < N; i++) {
    int x;
    std::cin >> x;
    if (x < 0) continue;
    sum += x;
    // equivalent to: if (x >= 0) sum += x;
}
```

Section 69.3: do

Introduces a do-while loop.

```
// Gets the next non-whitespace character from standard input
char read_char() {
    char c;
    do {
        c = getchar();
    } while (isspace(c));
    return c;
}
```

Section 69.4: while

Introduces a while loop.

```
int i = 0;
// print 10 asterisks
while (i < 10) {
    putchar('*');
    i++;
}
```

第69.5节：基于范围的for循环

```
std::vector<int> primes = {2, 3, 5, 7, 11, 13};

for(auto prime : primes) {
    std::cout << prime << std::endl;
}
```

第69.6节：for循环

介绍for循环，或在C++11及以后版本中的基于范围的for循环。

```
// 打印10个星号
for (int i = 0; i < 10; i++) {
    putchar('*');
}
```

Section 69.5: range-based for loop

```
std::vector<int> primes = {2, 3, 5, 7, 11, 13};

for(auto prime : primes) {
    std::cout << prime << std::endl;
}
```

Section 69.6: for

Introduces a for loop or, in C++11 and later, a range-based for loop.

```
// print 10 asterisks
for (int i = 0; i < 10; i++) {
    putchar('*');
}
```

第70章：正则表达式

签名	描述
<pre>bool regex_match(双向迭代器 first, 双向迭代器 last, smatch& sm, const regex& re, regex_constraints::match_flag_type flags)</pre>	双向迭代器 是任何提供递增和递减操作符的字符迭代器 smatch 可以是 cmatch 或任何其他接受 双向迭代器 类型的 match_results 变体 smatch参数可以省略，如果不需要正则表达式的结果 返回值 表示 re 是否匹配由 first 和 last 定义的整个字符序列
<pre>bool regex_match(const 字符串& str, smatch& sm, const regex re&, regex_constraints::match_flag_type flags)</pre>	字符串 可以是 const char* 或一个左值字符串 (string) , 接受右值字符串的函数被显式删除smatch 可以是 cmatch 或任何其他接受 str 类型的 match_results 变体 参数 smatch 可以省略，如果不需要正则表达式的结果 返回值 表示 re 是否匹配由 str 定义的整个字符序列

正则表达式 (有时称为reges或regesps) 是一种文本语法，用于表示可以在所操作的字符串中匹配的模式。

正则表达式，自C++11引入后，可选择支持返回匹配字符串数组，或另一种文本语法来定义如何替换操作字符串中的匹配模式。

第70.1节：基本的regex_match和regex_search示例

```
const auto input = "有些人，当面对问题时，会想\"我知道了，我要用正则表达式。\"";  
smatch sm;  
  
cout << input << endl;  
  
// 如果input以包含一个以"reg"开头的单词和另一个以"ex"开头的单词的引号结尾，则捕获input的前半部分  
  
if (regex_match(input, sm, regex("(.*).*\breg.*\bex.*"\s*"))){  
    const auto capture = sm[1].str();  
  
    cout << " << capture << endl; // 输出: "有些人，当面对问题时，会想"  
  
    // 在capture中搜索" a problem"或"# problems"  
    if(regex_search(capture, sm, regex("(a|d+)\s+problems?"))){  
        const auto count = sm[1] == "a"s ? 1 : stoi(sm[1]);  
  
        cout << " << count << (count > 1 ? " problems" : " problem"); // 输出: "1problem"  
  
        cout << "现在他们有 " << count + 1 << " 个问题。"; // 输出: "现在他们有 2个问题"  
    }  
}
```

实时示例

第70.2节：regex_iterator 示例

当捕获处理必须迭代进行时， regex_iterator 是一个不错的选择。对 regex_iterator 解引用返回一个 match_result。这对于条件捕获或具有

Chapter 70: Regular expressions

Signature	Description
<pre>bool regex_match(BidirectionalIterator first, BidirectionalIterator last, smatch& sm, const regex& re, regex_constraints::match_flag_type flags)</pre>	BidirectionalIterator is any character iterator that provides increment and decrement operators smatch may be cmatch or any other other variant of match_results that accepts the type of BidirectionalIterator the smatch argument may be omitted if the results of the regex are not needed Returns whether re matches the entire character sequence defined by first and last
<pre>bool regex_match(const string& str, smatch& sm, const regex re&, regex_constraints::match_flag_type flags)</pre>	string may be either a const char* or an L-Value string, the functions accepting an R-Value string are explicitly deleted smatch may be cmatch or any other other variant of match_results that accepts the type of str the smatch argument may be omitted if the results of the regex are not needed Returns whether re matches the entire character sequence defined by str

Regular Expressions (sometimes called reges or regesps) are a textual syntax which represents the patterns which can be matched in the strings operated upon.

Regular Expressions, introduced in [c++11](#), may optionally support a return array of matched strings or another textual syntax defining how to replace matched patterns in strings operated upon.

Section 70.1: Basic regex_match and regex_search Examples

```
const auto input = "Some people, when confronted with a problem, think \"I know, I'll use regular  
expressions.\s";  
smatch sm;  
  
cout << input << endl;  
  
// If input ends in a quotation that contains a word that begins with "reg" and another word  
beginning with "ex" then capture the preceding portion of input  
if (regex_match(input, sm, regex("(.*).*\breg.*\bex.*"\s*"))){  
    const auto capture = sm[1].str();  
  
    cout << '\t' << capture << endl; // Outputs: "\tSome people, when confronted with a problem,  
think\n"  
  
    // Search our capture for "a problem" or "# problems"  
    if(regex_search(capture, sm, regex("(a|d+)\s+problems?"))){  
        const auto count = sm[1] == "a"s ? 1 : stoi(sm[1]);  
  
        cout << '\t' << count << (count > 1 ? " problems\n" : " problem\n"); // Outputs: "\t1  
problem\n"  
        cout << "Now they have " << count + 1 << " problems.\n"; // Outputs: "Now they have 2  
problems\n"  
    }  
}
```

Live Example

Section 70.2: regex_iterator Example

When processing of captures has to be done iteratively a regex_iterator is a good choice. Dereferencing a regex_iterator returns a match_result. This is great for conditional captures or captures which have

相互依赖的捕获非常有用。假设我们想要对一些C++代码进行分词。给定：

```
enum TOKENS {
    NUMBER,
    ADDITION,
    SUBTRACTION,
    MULTIPLICATION,
    DIVISION,
    EQUALITY,
    OPEN_PARENTHESIS,
    CLOSE_PARENTHESIS
};
```

我们可以用 `regex_iterator` 对这个字符串进行分词：const auto input = "42/2 + -8=

```
(2 + 2) * 2 * 2 - 3\n, 方便如下:\n\nvector<TOKENS> tokens;\nconst regex re{ "\\\s*(\\(?)\\s*(-?\\s*\\d+)\\s*(\\(?)\\s*(?:\\+)|(-)|(\\*)|(/)|(=))" };\n\nfor_each(sregex_iterator(cbegin(input), cend(input), re), sregex_iterator(), [&](const auto& i) {\n    if(i[1].length() > 0) {\n        tokens.push_back(OPEN_PARENTHESIS);\n    }\n\n    tokens.push_back(i[2].str().front() == '-' ? NEGATIVE_NUMBER : NON_NEGATIVE_NUMBER);\n\n    if(i[3].length() > 0) {\n        tokens.push_back(CLOSE_PARENTHESIS);\n    }\n\n    auto it = next(cbegin(i), 4);\n\n    for(int result = ADDITION; it != cend(i); ++result, ++it) {\n        if(it->length() > 0U) {\n            tokens.push_back(static_cast<TOKENS>(result));\n            break;\n        }\n    }\n});\n\nmatch_results<string::const_reverse_iterator> sm;\n\nif(regex_search(cbegin(input), cend(input), sm, regex{ tokens.back() == SUBTRACTION ?\n    "\\\\s*\\d+\\s*-\\s*(-?)" : "\\\\s*\\d+\\s*(?-)" })) {\n    tokens.push_back(sm[1].length() == 0 ? NON_NEGATIVE_NUMBER : NEGATIVE_NUMBER);\n}
```

实时示例

使用正则表达式迭代器时一个显著的陷阱是，`regex`参数必须是左值，右值将无法工作：Visual Studio regex iterator Bug？

第70.3节：锚点

C++只提供4个锚点：

- ^ 表示字符串的开始
- \$ 表示字符串的结束
- \b 表示一个\W字符或字符串的开始或结束

interdependence. Let's say that we want to tokenize some C++ code. Given:

```
enum TOKENS {\n    NUMBER,\n    ADDITION,\n    SUBTRACTION,\n    MULTIPLICATION,\n    DIVISION,\n    EQUALITY,\n    OPEN_PARENTHESIS,\n    CLOSE_PARENTHESIS\n};
```

We can tokenize this string: const auto input = "42/2 + -8\t=\n(2 + 2) * 2 * 2 - 3"s with a `regex_iterator` like this:

```
vector<TOKENS> tokens;\nconst regex re{ "\\\s*(\\(?)\\s*(-?\\s*\\d+)\\s*(\\(?)\\s*(?:\\+)|(-)|(\\*)|(/)|(=))" };\n\nfor_each(sregex_iterator(cbegin(input), cend(input), re), sregex_iterator(), [&](const auto& i) {\n    if(i[1].length() > 0) {\n        tokens.push_back(OPEN_PARENTHESIS);\n    }\n\n    tokens.push_back(i[2].str().front() == '-' ? NEGATIVE_NUMBER : NON_NEGATIVE_NUMBER);\n\n    if(i[3].length() > 0) {\n        tokens.push_back(CLOSE_PARENTHESIS);\n    }\n\n    auto it = next(cbegin(i), 4);\n\n    for(int result = ADDITION; it != cend(i); ++result, ++it) {\n        if(it->length() > 0U) {\n            tokens.push_back(static_cast<TOKENS>(result));\n            break;\n        }\n    }\n});\n\nmatch_results<string::const_reverse_iterator> sm;\n\nif(regex_search(cbegin(input), cend(input), sm, regex{ tokens.back() == SUBTRACTION ?\n    "\\\\s*\\d+\\s*-\\s*(-?)" : "\\\\s*\\d+\\s*(?-)" })) {\n    tokens.push_back(sm[1].length() == 0 ? NON_NEGATIVE_NUMBER : NEGATIVE_NUMBER);\n}
```

Live Example

A notable gotcha with regex iterators is that the `regex` argument must be an L-value, an R-value will not work: [Visual Studio regex iterator Bug?](#)

Section 70.3: Anchors

C++ provides only 4 anchors:

- ^ which asserts the start of the string
- \$ which asserts the end of the string
- \b which asserts a \W character or the beginning or end of the string

- \B 表示一个\w字符

举个例子，假设我们想捕获带符号的数字：

```
auto input = "+1--12*123/+1234"s;
smatch sm;

if(regex_search(input, sm, regex{ "(?:^|\\b\\W)([+-]?\\d+)" })) {
    do {
        cout << sm[1] << endl;
    } 当(正则表达式搜索(输入, sm, 正则表达式{ "(?:^|\\W|\\b\\W)([+-]?\\d+)" }));
}
```

[实时示例](#)

这里一个重要的注意点是锚点不会消耗任何字符。

第70.4节：regex_replace 示例

这段代码接受各种大括号风格并将其转换为统一的大括号风格 (One_True_Brace_Style)：

```
const auto 输入 = "if (KnR)foo();if (spaces) {    foo();}if(allman){foo();}if (horstmann){foo
();}if (pico){foo(); }if(whitesmiths){foo();}"字符串;

cout << 输入 << regex_replace(输入, 正则表达式("(.+?)\\s*\\{?\\s*(.+)\\}\\s*"), "$1{$2}") << endl;
```

[实时示例](#)

第70.5节：regex_token_iterator 示例

`std::regex_token_iterator` 提供了一个强大的工具，用于提取逗号分隔值 (CSV) 文件中的元素。除了迭代的优势外，这个迭代器还能捕获转义逗号，而其他方法则难以做到：

```
const auto 输入 = "please split,this, csv, ,line,\\,\"字符串;const 正则表达
式 re{ "(?:[^\\\\\\,]|\\\\\\,.)+(:|\\$)" };
const vector<string> m_vecFields{ sregex_token_iterator(cbegin(input), cend(input), re, 1),
sregex_token_iterator() };

cout << input << endl;

copy(cbegin(m_vecFields), cend(m_vecFields), ostream_iterator<string>(cout, ""));
```

[实时示例](#)

使用正则表达式迭代器时一个显著的注意点是，`regex`参数必须是左值。右值将无法工作。

第70.6节：量词

假设给定了`const string input`作为要验证的电话号码。我们可以先要求一个数字输入，使用零次或多次量词：`regex_match(input, regex("\\d*"))`或者一次或多次量词：`regex_match(input, regex("\\d+"))`但如果`input`包含无效的

- \B which asserts a \w character

Let's say for example we want to capture a number *with* its sign:

```
auto input = "+1--12*123/+1234"s;
smatch sm;

if(regex_search(input, sm, regex{ "(?:^|\\b\\W)([+-]?\\d+)" })) {
    do {
        cout << sm[1] << endl;
        input = sm.suffix().str();
    } while(regex_search(input, sm, regex{ "(?:^|\\W|\\b\\W)([+-]?\\d+)" }));
}
```

[Live Example](#)

An important note here is that the anchor does not consume any characters.

Section 70.4: regex_replace Example

This code takes in various brace styles and converts them to One True Brace Style:

```
const auto input = "if (KnR)\n\tfoo();\nif (spaces) {\n    foo();\n}\nif\n(allman)\n{\n\tfoo();\n}\nif\n(horstmann)\n{\n\tfoo();\n}\nif\n(pico)\n{\n\tfoo();\n}\nif\n(whitesmiths)\n{\n\tfoo();\n}\ns;

cout << input << regex_replace(input, regex("(.+?)\\s*\\{?\\s*(.+)\\}\\s*"), "$1{$2}") << endl;
```

[Live Example](#)

Section 70.5: regex_token_iterator Example

A `std::regex_token_iterator` provides a tremendous tool for extracting elements of a Comma Separated Value file. Aside from the advantages of iteration, this iterator is also able to capture escaped commas where other methods struggle:

```
const auto input = "please split,this, csv, ,line,\\,\"字符串;
const regex re{ "(?:[^\\\\\\,]|\\\\\\,.)+(:|\\$)" };
const vector<string> m_vecFields{ sregex_token_iterator(cbegin(input), cend(input), re, 1),
sregex_token_iterator() };

cout << input << endl;

copy(cbegin(m_vecFields), cend(m_vecFields), ostream_iterator<string>(cout, "\n"));
```

[Live Example](#)

A notable gotcha with regex iterators is, that the `regex` argument must be an L-value. An R-value will not work.

Section 70.6: Quantifiers

Let's say that we're given `const string input` as a phone number to be validated. We could start by requiring a numeric input with a **zero or more quantifier**: `regex_match(input, regex("\\d*"))` or a **one or more quantifier**: `regex_match(input, regex("\\d+"))` But both of those really fall short if `input` contains an invalid

数字字符串，如："123"。我们使用至少n次量词来确保至少有7位数字：

```
regex_match(input, regex("\d{7}"))
```

这将保证我们至少得到一个电话号码的数字，但input也可能包含过长的数字字符串比如："123456789012"。所以我们使用**n到m之间的量词**，确保input至少7位数字但不超过11位：

```
regex_match(input, regex("\d{7,11}"));
```

这使我们更接近目标，但仍然接受范围在[7, 11]内的非法数字字符串，例如："123456789"。所以让我们用一个**惰性量词**使国家代码变为可选：

```
regex_match(input, regex("\d?\d{7,10}"))
```

需要注意的是，**lazy quantifier**（惰性量词）匹配的是尽可能少的字符，因此这个字符只有在已经有10个字符被\(\d{7,10}\)匹配的情况下才会被匹配。（如果要贪婪地匹配第一个字符，我们需要写成：\(\d{0,1}\)。）**lazy quantifier**可以附加到任何其他量词上。

现在，我们如何让区号变成可选的并且只有在区号存在时才接受国家代码呢？

```
regex_match(input, regex("(?:\d{3,4})?\d{7}"))
```

在这个最终的正则表达式中，\(\d{7}\)要求7个数字。这7个数字前面可选地跟着3或4个数字。

注意我们没有附加**lazy quantifier**：`\(\d{3,4}\)?\(\d{7}\)`，`\(\d{3,4}\)?`会匹配3或4个字符，优先匹配3个。相反，我们让非捕获组最多匹配一次，优先不匹配。如果input不包含区号，比如："1234567"，则会导致不匹配。

关于量词的话题，最后我想提一下你可以使用的另一种附加量词，**possessive quantifier**（占有量词）。**惰性量词**或**占有量词**都可以附加到任何量词上。

possessive quantifier的唯一作用是帮助正则引擎告诉它，贪婪地匹配这些字符，

即使这会导致正则匹配失败，也绝不放弃它们。例如，这种用法没有多大意义：

`regex_match(input, regex("\d{3,4}+\d{7}))` 因为像"1234567890"这样的input不会被匹配，`\(\d{3,4}\)+`总是匹配4个字符，即使匹配3个字符会让正则表达式成功。

possessive quantifier最适合用于当被量化的元素限制了可匹配字符的数量。例如：

```
regex_match(input, regex(".*\d{3,4}+){3}))
```

可用于匹配input是否包含以下任意内容：

```
123 456 7890  
123-456-7890  
(123)456-7890  
(123) 456 - 7890
```

但当input包含非法输入时，这个正则表达式才真正发挥作用：

```
12345 - 67890
```

numeric string like: "123" Let's use a **n or more quantifier** to ensure that we're getting at least 7 digits:

```
regex_match(input, regex("\d{7,}"))
```

This will guarantee that we will get at least a phone number of digits, but input could also contain a numeric string that's too long like: "123456789012". So let's go with a **between n and m quantifier** so the input is at least 7 digits but not more than 11:

```
regex_match(input, regex("\d{7,11}"));
```

This gets us closer, but illegal numeric strings that are in the range of [7, 11] are still accepted, like: "123456789" So let's make the country code optional with a **lazy quantifier**:

```
regex_match(input, regex("\d?\d{7,10}"))
```

It's important to note that the **lazy quantifier** matches *as few characters as possible*, so the only way this character will be matched is if there are already 10 characters that have been matched by \(\d{7,10}\). (To match the first character greedily we would have had to do: \(\d{0,1}\).) The **lazy quantifier** can be appended to any other quantifier.

Now, how would we make the area code optional *and* only accept a country code if the area code was present?

```
regex_match(input, regex("(?:\d{3,4})?\d{7}))
```

In this final regex, the \(\d{7}\) requires 7 digits. These 7 digits are optionally preceded by either 3 or 4 digits.

Note that we did not append the **lazy quantifier**: `\(\d{3,4}\)?\(\d{7}\)`，the `\(\d{3,4}\)?` would have matched either 3 or 4 characters, preferring 3. Instead we're making the non-capturing group match at most once, preferring not to match. Causing a mismatch if input didn't include the area code like: "1234567".

In conclusion of the quantifier topic, I'd like to mention the other appending quantifier that you can use, the **possessive quantifier**. Either the **lazy quantifier** or the **possessive quantifier** can be appended to any quantifier.

The **possessive quantifier**'s only function is to assist the regex engine by telling it, greedily take these characters and don't ever give them up even if it causes the regex to fail. This for example doesn't make much sense:

`regex_match(input, regex("\d{3,4}+\d{7}))` Because an input like: "1234567890" wouldn't be matched as `\(\d{3,4}\)+` will always match 4 characters even if matching 3 would have allowed the regex to succeed.

The **possessive quantifier** is best used when the quantified token limits the number of matchable characters. For example:

```
regex_match(input, regex(".*\d{3,4}+){3}))
```

Can be used to match if input contained any of the following:

```
123 456 7890  
123-456-7890  
(123)456-7890  
(123) 456 - 7890
```

But when this regex really shines is when input contains an *illegal* input:

```
12345 - 67890
```

没有占有量词时，正则引擎必须回溯并测试.*和3或4个字符的每一种组合，看看是否能找到匹配的组合。使用占有量词后，正则从第二个占有量词停下的位置，即字符'0'开始，尝试调整.*以允许\ d{3,4}匹配；当无法匹配时，正则直接失败，不会回溯去尝试之前的.*调整是否能匹配。

第70.7节：拆分字符串

```
std::vector<std::string> split(const std::string &str, std::string regex)
{
    std::regex r{ regex };
    std::sregex_token_iterator start{ str.begin(), str.end(), r, -1 }, end;
    return std::vector<std::string>(start, end);
}

split("Some string with whitespace ", "\\s+"); // "Some", "string", "with", "whitespace"
```

Without the **possessive quantifier** the regex engine has to go back and test *every combination of .** and either 3 or 4 characters to see if it can find a matchable combination. With the **possessive quantifier** the regex starts where the 2nd **possessive quantifier** left off, the '0' character, and the regex engine tries to adjust the .* to allow \d{3,4} to match; when it can't the regex just fails, no back tracking is done to see if earlier .* adjustment could have allowed a match.

Section 70.7: Splitting a string

```
std::vector<std::string> split(const std::string &str, std::string regex)
{
    std::regex r{ regex };
    std::sregex_token_iterator start{ str.begin(), str.end(), r, -1 }, end;
    return std::vector<std::string>(start, end);
}

split("Some string\t with whitespace ", "\\s+"); // "Some", "string", "with", "whitespace"
```

第71章：实现定义的行为

第71.1节：整型大小

以下类型被定义为整型类型：

- 字符型 (char)
- 有符号整型
- 无符号整型
- `char16_t` 和 `char32_t`
- 布尔型 (bool)
- 宽字符型 (wchar_t)

除 `sizeof(char)` / `sizeof(有符号字符)` / `sizeof(无符号字符)` 外，这些大小在 §3.9.1.1 [basic.fundamental/1] 和 §5.3.3.1 [expr.sizeof] 之间分割定义，且 `sizeof(bool)` 完全由实现定义且无最小大小要求外，这些类型的最小大小要求在标准的 § 3.9.1 [basic.fundamental] 节中给出，具体如下。

char 的大小

C++ 标准的所有版本在 § 5.3.3.1 中规定，`sizeof` 对 `unsigned char`、`signed char` 和 `char` 的结果均为 1 (`char` 类型是有符号还是无符号由实现决定)。

版本 ≥ C++14

`char` 足够大，可以表示 256 个不同的值，适合存储 UTF-8 代码单元。

有符号和无符号整数类型的大小

标准在 § 3.9.1.2 中规定，标准有符号整数类型列表包括 `signed char`、`short int`、`int`、`long int` 和 `long long int`，每种类型至少具有不小于其前面类型的存储空间。此外，如 § 3.9.1.3 所述，这些类型各自对应一个标准无符号整数类型，分别是 `unsigned char`、`unsigned short int`、`unsigned int`、`unsigned long int` 和 `unsigned long long int`，且无符号类型与对应的有符号类型具有相同的大小和对齐要求。另外，§ 3.9.1.1 规定，`char` 的大小和对齐要求与 `signed char` 和 `unsigned char` 相同。

版本 < C++11

在 C++11 之前，`long long` 和 `unsigned long long` 并非 C++ 标准的正式部分。然而，在 C99 中引入后，许多编译器支持 `long long` 作为扩展的有符号整数类型，`unsigned long long` 作为扩展的无符号整数类型，规则与 C 类型相同。

因此标准保证：

```
1 == sizeof(char) == sizeof(signed char) == sizeof(unsigned char)
<= sizeof(short) == sizeof(unsigned short)
<= sizeof(int) == sizeof(unsigned int)
<= sizeof(long) == sizeof(unsigned long)
```

版本 ≥ C++11

```
<= sizeof(long long) == sizeof(unsigned long long)
```

标准未规定每种类型的具体最小尺寸。相反，每种类型都有一个最小范围

Chapter 71: Implementation-defined behavior

Section 71.1: Size of integral types

The following types are defined as *integral types*:

- `char`
- Signed integer types
- Unsigned integer types
- `char16_t` and `char32_t`
- `bool`
- `wchar_t`

With the exception of `sizeof(char)` / `sizeof(signed char)` / `sizeof(unsigned char)`, which is split between § 3.9.1.1 [basic.fundamental/1] and § 5.3.3.1 [expr.sizeof], and `sizeof(bool)`, which is entirely implementation-defined and has no minimum size, the minimum size requirements of these types are given in section § 3.9.1 [basic.fundamental] of the standard, and shall be detailed below.

Size of `char`

All versions of the C++ standard specify, in § 5.3.3.1, that `sizeof` yields 1 for `unsigned char`, `signed char`, and `char` (it is implementation defined whether the `char` type is `signed` or `unsigned`).

Version ≥ C++14

`char` is large enough to represent 256 different values, to be suitable for storing UTF-8 code units.

Size of signed and unsigned integer types

The standard specifies, in § 3.9.1.2, that in the list of *standard signed integer types*, consisting of `signed char`, `short int`, `int`, `long int`, and `long long int`, each type will provide at least as much storage as those preceding it in the list. Furthermore, as specified in § 3.9.1.3, each of these types has a corresponding *standard unsigned integer type*, `unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int`, and `unsigned long long int`, which has the same size and alignment as its corresponding signed type. Additionally, as specified in § 3.9.1.1, `char` has the same size and alignment requirements as both `signed char` and `unsigned char`.

Version < C++11

Prior to C++11, `long long` and `unsigned long long` were not officially part of the C++ standard. However, after their introduction to C, in C99, many compilers supported `long long` as an *extended signed integer type*, and `unsigned long long` as an *extended unsigned integer type*, with the same rules as the C types.

The standard thus guarantees that:

```
1 == sizeof(char) == sizeof(signed char) == sizeof(unsigned char)
<= sizeof(short) == sizeof(unsigned short)
<= sizeof(int) == sizeof(unsigned int)
<= sizeof(long) == sizeof(unsigned long)
```

Version ≥ C++11

```
<= sizeof(long long) == sizeof(unsigned long long)
```

Specific minimum sizes for each type are not given by the standard. Instead, each type has a minimum range of

它可以支持的值范围，如§3.9.1.3中所规定，继承自C标准，见§5.2.4.2.1。每种类型的最小大小可以大致从该范围推断，通过确定所需的最小位数；注意，对于任何给定平台，任何类型实际支持的范围可能大于最小值。注意，对于有符号类型，范围对应的是一补码，而非更常用的二补码；这是为了允许更多平台符合该标准。

类型	最小范围	所需最小位数
有符号字符型	-127 到 127 ($-(2^{7-1})$ 到 (2^{7-1}))	8
无符号字符型	0 到 255 (0 到 2^{8-1})	8
有符号短整型	-32,767 到 32,767 ($-(2^{15-1})$ 到 (2^{15-1}))	16
无符号短整型	0 到 65,535 (0 到 2^{16-1})	16
有符号整型	-32,767 到 32,767 ($-(2^{15-1})$ 到 (2^{15-1}))	16
无符号整型	0 到 65,535 (0 到 2^{16-1})	16
有符号长整型	-2,147,483,647 到 2,147,483,647 ($-(2^{31-1})$ 到 (2^{31-1}))	32
无符号长整型	0 到 4,294,967,295 (0 到 2^{32-1})	32

版本 ≥ C++11

类型	最小范围	最小所需位数
有符号长长整型	-9,223,372,036,854,775,807 到 9,223,372,036,854,775,807 ($-(2^{63}-1)$ 到 $(2^{63}-1)$)	64
无符号长长整型	0 到 18,446,744,073,709,551,615 (0 到 $2^{64}-1$)	64

由于每种类型允许大于其最小尺寸要求，类型大小可能因实现而异。最显著的例子是 64 位数据模型 LP64 和 LLP64，其中 LLP64 系统（如 64 位 Windows）具有 32 位的 int 和 long，而 LP64 系统（如 64 位 Linux）具有 32 位的 int 和 64 位的 long。因此，整数类型不能假定在所有平台上具有固定宽度。

版本 ≥ C++11

如果需要固定宽度的整数类型，请使用 `<cstdint>` 头文件中的类型，但请注意标准对实现支持精确宽度类型 `int8_t`、`int16_t`、`int32_t`、`int64_t`、`intptr_t`、`uint8_t`、`uint16_t`、`uint32_t`、`uint64_t` 和 `uintptr_t` 是可选的。

版本 ≥ C++11

char16_t 和 char32_t 的大小

`char16_t` 和 `char32_t` 的大小是由实现定义的，如 § 5.3.3.1 所规定，并遵循 § 3.9.1.5 中的规定：

- `char16_t` 足够大以表示任何 UTF-16 代码单元，并且具有与 `uint_least16_t` 相同的大小、有符号性和对齐方式；因此它的大小必须至少为 16 位。
- `char32_t` 足够大以表示任何 UTF-32 代码单元，并且具有与 `uint_least32_t` 相同的大小、有符号性和对齐方式；因此它的大小必须至少为 32 位。

bool 的大小

`bool` 的大小由实现定义，可能是 1，也可能不是。

wchar_t 的大小

`wchar_t`，如第 §3.9.1.5 节所规定，是一种独特的类型，其取值范围可以表示所有支持的区域设置中最大扩展字符集的每个不同代码单元。它的大小、有符号性和对齐方式与其他整型之一相同，该整型被称为其底层类型。该类型的大小由实现定义，正如

values it can support, which is, as specified in § 3.9.1.3, inherited from the C standard, in §5.2.4.2.1. The minimum size of each type can be roughly inferred from this range, by determining the minimum number of bits required; note that for any given platform, any type's actual supported range may be larger than the minimum. Note that for signed types, ranges correspond to one's complement, not the more commonly used two's complement; this is to allow a wider range of platforms to comply with the standard.

Type	Minimum range	Minimum bits required
<code>signed char</code>	-127 to 127 ($-(2^{7-1})$ to (2^{7-1}))	8
<code>unsigned char</code>	0 to 255 (0 to 2^{8-1})	8
<code>signed short</code>	-32,767 to 32,767 ($-(2^{15-1})$ to (2^{15-1}))	16
<code>unsigned short</code>	0 to 65,535 (0 to 2^{16-1})	16
<code>signed int</code>	-32,767 to 32,767 ($-(2^{15-1})$ to (2^{15-1}))	16
<code>unsigned int</code>	0 to 65,535 (0 to 2^{16-1})	16
<code>signed long</code>	-2,147,483,647 to 2,147,483,647 ($-(2^{31-1})$ to (2^{31-1}))	32
<code>unsigned long</code>	0 to 4,294,967,295 (0 to 2^{32-1})	32

Version ≥ C++11

Type	Minimum range	Minimum bits required
<code>signed long long</code>	-9,223,372,036,854,775,807 to 9,223,372,036,854,775,807 ($-(2^{63}-1)$ to $(2^{63}-1)$)	64
<code>unsigned long long</code>	0 to 18,446,744,073,709,551,615 (0 to $2^{64}-1$)	64

As each type is allowed to be greater than its minimum size requirement, types may differ in size between implementations. The most notable example of this is with the 64-bit data models LP64 and LLP64, where LLP64 systems (such as 64-bit Windows) have 32-bit `ints` and `longs`, and LP64 systems (such as 64-bit Linux) have 32-bit `ints` and 64-bit `longs`. Due to this, integer types cannot be assumed to have a fixed width across all platforms.

Version ≥ C++11

If integer types with fixed width are required, use types from the `<cstdint>` header, but note that the standard makes it optional for implementations to support the exact-width types `int8_t`、`int16_t`、`int32_t`、`int64_t`、`intptr_t`、`uint8_t`、`uint16_t`、`uint32_t`、`uint64_t` 和 `uintptr_t`.

Version ≥ C++11

Size of `char16_t` and `char32_t`

The sizes of `char16_t` and `char32_t` are implementation-defined, as specified in § 5.3.3.1, with the stipulations given in § 3.9.1.5:

- `char16_t` is large enough to represent any UTF-16 code unit, and has the same size, signedness, and alignment as `uint_least16_t`; it is thus required to be at least 16 bits in size.
- `char32_t` is large enough to represent any UTF-32 code unit, and has the same size, signedness, and alignment as `uint_least32_t`; it is thus required to be at least 32 bits in size.

Size of `bool`

The size of `bool` is implementation defined, and may or may not be 1.

Size of `wchar_t`

`wchar_t`，as specified in § 3.9.1.5, is a distinct type, whose range of values can represent every distinct code unit of the largest extended character set among the supported locales. It has the same size, signedness, and alignment as one of the other integral types, which is known as its *underlying type*. This type's size is implementation-defined, as

指定于§5.3.3.1，例如，可能至少为8、16或32位；如果系统支持Unicode，例如，`wchar_t`要求至少为32位（该规则的例外是Windows，在Windows中`wchar_t`为16位以兼容）。它继承自C90标准，ISO 9899:1990 § 4.1.5，仅做了少量措辞修改。

根据具体实现，`wchar_t` 的大小通常（但不总是）为8、16或32位。最常见的示例有：

- 在类Unix系统中，`wchar_t` 是32位，通常用于UTF-32。
- 在Windows中，`wchar_t` 是16位，用于UTF-16。
- 在仅支持8位的系统上，`wchar_t` 是8位。

版本 ≥ C++11

如果需要Unicode支持，建议使用`char`表示UTF-8，`char16_t`表示UTF-16，或`char32_t`表示UTF-32，而不是使用`wchar_t`。

数据模型

如上所述，整数类型的宽度在不同平台间可能不同。最常见的模型如下，大小以位为单位：

模型	<code>int</code>	<code>long</code>	<code>pointer</code>
LP32 (2/4/4)	16	32	32
ILP32 (4/4/4)	32	32	32
LLP64 (4/4/8)	32	32	64
LP64 (4/8/8)	32	64	64

在这些模型中：

- 16位Windows使用LP32。
- 32位*nix系统（Unix、Linux、Mac OSX及其他类Unix操作系统）和Windows使用ILP32。
- 64位Windows使用LLP64。
- 64位*nix系统使用LP64。

但请注意，这些模型并未在标准本身中具体提及。

第71.2节：Char可能是无符号或有符号

标准未指定`char`应为有符号还是无符号。不同编译器实现不同，或者可能允许通过命令行开关进行更改。

第71.3节：数值类型的范围

整数类型的范围由实现定义。头文件`<limits>`提供了`std::numeric_limits<T>`模板，提供所有基本类型的最小值和最大值。这些值满足C标准通过`<climits>`和（C++11及以上）`<cinttypes>`

头文件提供的保证。

- `std::numeric_limits<signed char>::min()`等于`SCHAR_MIN`，且小于或等于-127。
- `std::numeric_limits<有符号字符>::max()`等于`SCHAR_MAX`，且大于或等于127。
- `std::numeric_limits<unsigned char>::max()`等于`UCHAR_MAX`，且大于或等于 255。
- `std::numeric_limits<short>::min()`等于`SHRT_MIN`，且小于或等于 -32767。
- `std::numeric_limits<short>::max()`等于`SHRT_MAX`，且大于或等于 32767。

specified in § 5.3.3.1, and may be, for example, at least 8, 16, or 32 bits; if a system supports Unicode, for example, `wchar_t` is required to be at least 32 bits (an exception to this rule is Windows, where `wchar_t` is 16 bits for compatibility purposes). It is inherited from the C90 standard, ISO 9899:1990 § 4.1.5, with only minor rewording.

Depending on the implementation, the size of `wchar_t` is often, but not always, 8, 16, or 32 bits. The most common examples of these are:

- In Unix and Unix-like systems, `wchar_t` is 32-bit, and is usually used for UTF-32.
- In Windows, `wchar_t` is 16-bit, and is used for UTF-16.
- On a system which only has 8-bit support, `wchar_t` is 8 bit.

Version ≥ C++11

If Unicode support is desired, it is recommended to use `char` for UTF-8, `char16_t` for UTF-16, or `char32_t` for UTF-32, instead of using `wchar_t`.

Data Models

As mentioned above, the widths of integer types can differ between platforms. The most common models are as follows, with sizes specified in bits:

Model	<code>int</code>	<code>long</code>	<code>pointer</code>
LP32 (2/4/4)	16	32	32
ILP32 (4/4/4)	32	32	32
LLP64 (4/4/8)	32	32	64
LP64 (4/8/8)	32	64	64

Out of these models:

- 16-bit Windows used LP32.
- 32-bit *nix systems (Unix, Linux, Mac OSX, and other Unix-like OSes) and Windows use ILP32.
- 64-bit Windows uses LLP64.
- 64-bit *nix systems use LP64.

Note, however, that these models aren't specifically mentioned in the standard itself.

Section 71.2: Char might be unsigned or signed

The standard doesn't specify if `char` should be signed or unsigned. Different compilers implement it differently, or might allow to change it using a command line switch.

Section 71.3: Ranges of numeric types

The ranges of the integer types are implementation-defined. The header `<limits>` provides the `std::numeric_limits<T>` template which provides the minimum and maximum values of all fundamental types. The values satisfy guarantees provided by the C standard through the `<climits>` and (\geq C++11) `<cinttypes>` headers.

- `std::numeric_limits<signed char>::min()` equals `SCHAR_MIN`, which is less than or equal to -127.
- `std::numeric_limits<signed char>::max()` equals `SCHAR_MAX`, which is greater than or equal to 127.
- `std::numeric_limits<unsigned char>::max()` equals `UCHAR_MAX`, which is greater than or equal to 255.
- `std::numeric_limits<short>::min()` equals `SHRT_MIN`, which is less than or equal to -32767.
- `std::numeric_limits<short>::max()` equals `SHRT_MAX`, which is greater than or equal to 32767.

- `std::numeric_limits<unsigned short>::max()` 等于 `USHRT_MAX`, 且大于或等于 65535。
- `std::numeric_limits<int>::min()` 等于 `INT_MIN`, 且小于或等于 -32767。
- `std::numeric_limits<int>::max()` 等于 `INT_MAX`, 且大于或等于 32767。
- `std::numeric_limits<unsigned int>::max()` 等于 `UINT_MAX`, 且大于或等于 65535。
- `std::numeric_limits<long>::min()` 等于 `LONG_MIN`, 且小于或等于 -2147483647。
- `std::numeric_limits<long>::max()` 等于 `LONG_MAX`, 且大于或等于 2147483647。
- `std::numeric_limits<unsigned long>::max()` 等于 `ULONG_MAX`, 且大于或等于 4294967295。

版本 ≥ C++11

- `std::numeric_limits<long long>::min()` 等于 `LLONG_MIN`, 且小于或等于 -9223372036854775807。
- `std::numeric_limits<long long>::max()` 等于 `LLONG_MAX`, 且大于或等于 9223372036854775807。
- `std::numeric_limits<unsigned long long>::max()` 等于 `ULLONG_MAX`, 且大于或等于 18446744073709551615。

对于浮点类型 `T`, `max()` 是最大有限值, 而 `min()` 是最小的正归一化值。

为浮点类型提供了额外的成员, 这些成员也是实现定义的, 但满足通过 `<cfloat>` 头文件由 C 标准提供的某些保证。
◆

- 成员 `digits10` 表示十进制有效数字的位数。
 - `std::numeric_limits<float>::digits10` 等于 `FLT_DIG`, 至少为6。
 - `std::numeric_limits<double>::digits10` 等于 `DBL_DIG`, 至少为10。
 - `std::numeric_limits<long double>::digits10` 等于 `LDBL_DIG`, 至少为10。
- 成员 `min_exponent10` 是最小的负数 `E`, 使得 `10` 的 `E` 次方是正规数。
 - `std::numeric_limits<float>::min_exponent10` 等于 `FLT_MIN_10_EXP`, 最多为 -37。
 - `std::numeric_limits<double>::min_exponent10` 等于 `DBL_MIN_10_EXP`, 最多为 -37。
 - `std::numeric_limits<long double>::min_exponent10` 等于 `LDBL_MIN_10_EXP`, 最多为 -37。
- 成员 `max_exponent10` 是最大的 `E`, 使得 `10` 的 `E` 次方是有限的。
 - `std::numeric_limits<float>::max_exponent10` 等于 `FLT_MAX_10_EXP`, 至少为 37。
 - `std::numeric_limits<double>::max_exponent10` 等于 `DBL_MAX_10_EXP`, 至少为 37。
 - `std::numeric_limits<long double>::max_exponent10` 等于 `LDBL_MAX_10_EXP`, 至少为 37。
- 如果成员 `is_iec559` 为真, 则该类型符合 IEC 559 / IEEE 754 标准, 因此其范围由该标准确定。

第71.4节：浮点类型的值表示

该标准要求 `long double` 提供的精度至少与 `double` 相同, 而 `double` 提供的精度至少与 `float` 相同; 并且 `long double` 可以表示任何 `double` 能够表示的值, 而 `double` 可以表示任何 `float` 能够表示的值。然而, 表示的具体细节由实现定义。
◆

对于浮点类型 `T`, `std::numeric_limits<T>::radix` 指定了 `T` 表示所使用的基数。

如果 `std::numeric_limits<T>::is_iec559` 为真, 则 `T` 的表示形式符合 IEC 559 / IEEE 754 定义的格式之一。

第71.5节：从整数转换为有符号整数时的溢出

当有符号或无符号整数被转换为有符号整数类型时, 且其值无法表示于该类型中

- `std::numeric_limits<unsigned short>::max()` 等于 `USHRT_MAX`, 且大于或等于 65535。
- `std::numeric_limits<int>::min()` 等于 `INT_MIN`, 且小于或等于 -32767。
- `std::numeric_limits<int>::max()` 等于 `INT_MAX`, 且大于或等于 32767。
- `std::numeric_limits<unsigned int>::max()` 等于 `UINT_MAX`, 且大于或等于 65535。
- `std::numeric_limits<long>::min()` 等于 `LONG_MIN`, 且小于或等于 -2147483647。
- `std::numeric_limits<long>::max()` 等于 `LONG_MAX`, 且大于或等于 2147483647。
- `std::numeric_limits<unsigned long>::max()` 等于 `ULONG_MAX`, 且大于或等于 4294967295。

Version ≥ C++11

- `std::numeric_limits<long long>::min()` 等于 `LLONG_MIN`, 且小于或等于 -9223372036854775807。
- `std::numeric_limits<long long>::max()` 等于 `LLONG_MAX`, 且大于或等于 9223372036854775807。
- `std::numeric_limits<unsigned long long>::max()` 等于 `ULLONG_MAX`, 且大于或等于 18446744073709551615。

For floating-point types `T`, `max()` is the maximum finite value while `min()` is the minimum positive normalized value. Additional members are provided for floating-point types, which are also implementation-defined but satisfy certain guarantees provided by the C standard through the `<cfloat>` header.

- The member `digits10` gives the number of decimal digits of precision.
 - `std::numeric_limits<float>::digits10` equals `FLT_DIG`, which is at least 6.
 - `std::numeric_limits<double>::digits10` equals `DBL_DIG`, which is at least 10.
 - `std::numeric_limits<long double>::digits10` equals `LDBL_DIG`, which is at least 10.
- The member `min_exponent10` is the minimum negative `E` such that `10` to the power `E` is normal.
 - `std::numeric_limits<float>::min_exponent10` equals `FLT_MIN_10_EXP`, which is at most -37.
 - `std::numeric_limits<double>::min_exponent10` equals `DBL_MIN_10_EXP`, which is at most -37.
 - `std::numeric_limits<long double>::min_exponent10` equals `LDBL_MIN_10_EXP`, which is at most -37.
- The member `max_exponent10` is the maximum `E` such that `10` to the power `E` is finite.
 - `std::numeric_limits<float>::max_exponent10` equals `FLT_MAX_10_EXP`, which is at least 37.
 - `std::numeric_limits<double>::max_exponent10` equals `DBL_MAX_10_EXP`, which is at least 37.
 - `std::numeric_limits<long double>::max_exponent10` equals `LDBL_MAX_10_EXP`, which is at least 37.
- If the member `is_iec559` is true, the type conforms to IEC 559 / IEEE 754, and its range is therefore determined by that standard.

Section 71.4: Value representation of floating point types

The standard requires that `long double` provides at least as much precision as `double`, which provides at least as much precision as `float`; and that a `long double` can represent any value that a `double` can represent, while a `double` can represent any value that a `float` can represent. The details of the representation are, however, implementation-defined.

For a floating point type `T`, `std::numeric_limits<T>::radix` specifies the radix used by the representation of `T`.

If `std::numeric_limits<T>::is_iec559` is true, then the representation of `T` matches one of the formats defined by IEC 559 / IEEE 754.

Section 71.5: Overflow when converting from integer to signed integer

When either a signed or unsigned integer is converted to a signed integer type, and its value is not representable in

目标类型，产生的值由实现定义。例如：

```
// 假设在此实现中，带符号字符 (signed char) 的范围是 -128 到 +127，  
// 无符号字符 (unsigned char) 的范围是 0 到 255  
int x = 12345;  
signed char sc = x; // sc 具有实现定义的值  
unsigned char uc = x; // uc 初始化为 57 (即 12345 模 256)
```

第71.6节：枚举的底层类型（因此大小）

如果未显式指定无作用域枚举类型的底层类型，则该类型以实现定义的方式确定。

```
enum E {  
    RED,  
    GREEN,  
    BLUE,  
};  
using T = std::underlying_type<E>::type; // 实现定义
```

但是，标准确实要求枚举的底层类型不大于int，除非int和unsigned int都无法表示枚举的所有值。因此，在上述代码中，T可以是int、unsigned int或short，但不能是long long，仅举几例。

注意，枚举的大小（由sizeof返回）与其底层类型相同。

第71.7节：指针的数值

使用reinterpret_cast将指针转换为整数的结果是实现定义的，但“.....旨在对了解底层机器寻址结构的人来说不会令人惊讶。”

```
int x = 42;  
int* p = &x;  
long addr = reinterpret_cast<long>(p);  
std::cout << addr << ""; // 打印某个数值地址，// 可能是架构的本地地址  
格式
```

同样，由整数转换得到的指针也是实现定义的。

将指针存储为整数的正确方法是使用uintptr_t或intptr_t类型：

```
// `uintptr_t` 在 C++03 中不存在。它在 C99 的 <stdint.h> 中，作为可选类型  
#include <stdint.h>  
  
uintptr_t up;  
版本 ≥ C++11  
// C++11 中有可选的 `std::uintptr_t`  
#include <cstdint>  
  
std::uintptr_t up;
```

C++11 参考 C99 来定义 uintptr_t (C99 标准, 6.3.2.3)：

一种无符号整数类型，具有以下属性：任何指向void的有效指针都可以转换为该类型，然后再转换回指向void的指针，结果将与原始指针相等。

the destination type, the value produced is implementation-defined. Example:

```
// Suppose that on this implementation, the range of signed char is -128 to +127 and  
// the range of unsigned char is 0 to 255  
int x = 12345;  
signed char sc = x; // sc has an implementation-defined value  
unsigned char uc = x; // uc is initialized to 57 (i.e., 12345 modulo 256)
```

Section 71.6: Underlying type (and hence size) of an enum

If the underlying type is not explicitly specified for an unscoped enumeration type, it is determined in an implementation-defined manner.

```
enum E {  
    RED,  
    GREEN,  
    BLUE,  
};  
using T = std::underlying_type<E>::type; // implementation-defined
```

However, the standard does require the underlying type of an enumeration to be no larger than int unless both int and unsigned int are unable to represent all the values of the enumeration. Therefore, in the above code, T could be int, unsigned int, or short, but not long long, to give a few examples.

Note that an enum has the same size (as returned by sizeof) as its underlying type.

Section 71.7: Numeric value of a pointer

The result of casting a pointer to an integer using reinterpret_cast is implementation-defined, but "... is intended to be unsurprising to those who know the addressing structure of the underlying machine."

```
int x = 42;  
int* p = &x;  
long addr = reinterpret_cast<long>(p);  
std::cout << addr << "\n"; // prints some numeric address,  
// probably in the architecture's native address format
```

Likewise, the pointer obtained by conversion from an integer is also implementation-defined.

The right way to store a pointer as an integer is using the uintptr_t or intptr_t types:

```
// `uintptr_t` was not in C++03. It's in C99, in <stdint.h>, as an optional type  
#include <stdint.h>  
  
uintptr_t up;  
Version ≥ C++11  
// There is an optional `std::uintptr_t` in C++11  
#include <cstdint>  
  
std::uintptr_t up;
```

C++11 参考 C99 对于 uintptr_t (C99 标准, 6.3.2.3)：

一种无符号整数类型，具有以下属性：任何有效指针都可以转换为该类型，然后再转换回指向void的指针，结果将与原始指针相等。

对于大多数现代平台，你可以假设地址空间是平坦的，并且对uintptr_t的算术运算等同于对char *的算术运算，但实现完全可能在将void *转换为uintptr_t时执行任何转换，只要该转换在从uintptr_t转换回void *时可以被逆转。

技术细节

- 在符合XSI (X/Open系统接口) 标准的系统上，intptr_t和uintptr_t类型是必需的，否则它们是可选的。
- 根据C标准的含义，函数不是对象；C标准并不保证 uintptr_t可以存储函数指针。无论如何，POSIX (2.12.3) 符合性要求：

所有函数指针类型应具有与void指针类型相同的表示形式。

函数指针转换为void *时，不应改变其表示形式。由此转换得到的void *值可以通过显式转换转换回原始函数指针类型，且不会丢失信息。

- C99 §7.18.1：

当定义仅在是否带有前导u上不同的typedef名称时，它们应表示6.2.5节中描述的对应的有符号和无符号类型；提供其中一种对应类型的实现也应提供另一种。

uintptr_t 可能有意义，如果你想对指针的位进行操作，而这些操作用有符号整数无法合理完成。

While, for the majority of modern platforms, you can assume a flat address space and that arithmetic on `uintptr_t` is equivalent to arithmetic on `char *`, it's entirely possible for an implementation to perform any transformation when casting `void *` to `uintptr_t` as long the transformation can be reversed when casting back from `uintptr_t` to `void *`.

Technicalities

- On XSI-conformant (X/Open System Interfaces) systems, `intptr_t` and `uintptr_t` types are required, otherwise they are **optional**.
- Within the meaning of the C standard, functions aren't objects; it isn't guaranteed by the C standard that `uintptr_t` can hold a function pointer. Anyway POSIX (2.12.3) conformance requires that:

All function pointer types shall have the same representation as the type pointer to void. Conversion of a function pointer to void * shall not alter the representation. A void * value resulting from such a conversion can be converted back to the original function pointer type, using an explicit cast, without loss of information.

- C99 §7.18.1:

When typedef names differing only in the absence or presence of the initial u are defined, they shall denote corresponding signed and unsigned types as described in 6.2.5; an implementation providing one of these corresponding types shall also provide the other.

`uintptr_t` might make sense if you want to do things to the bits of the pointer that you can't do as sensibly with a signed integer.

第71.8节：字节中的位数

在C++中，byte 是char对象所占用的空间。字节中的位数由CHAR_BIT给出，该值定义在climits中，且要求至少为8。虽然大多数现代系统的字节是8位，且POSIX要求CHAR_BIT必须正好为8，但也有一些系统的CHAR_BIT大于8，即一个字节可能由8、16、32或64位组成。

Section 71.8: Number of bits in a byte

In C++, a *byte* is the space occupied by a `char` object. The number of bits in a byte is given by `CHAR_BIT`, which is defined in `climits` and required to be at least 8. While most modern systems have 8-bit bytes, and POSIX requires `CHAR_BIT` to be exactly 8, there are some systems where `CHAR_BIT` is greater than 8 i.e a single byte may be comprised of 8, 16, 32 or 64 bits.

第72章：异常

第72.1节：捕获异常

使用try/catch块来捕获异常。位于try部分的代码是可能抛出异常的代码，而catch子句中的代码则处理该异常。

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

int main() {
    std::string str("foo");

    try {
        str.at(10); // 访问元素，可能抛出 std::out_of_range
    } catch (const std::out_of_range& e) {
        // what() 继承自 std::exception 并包含解释性信息
        std::cout << e.what();
    }
}
```

可以使用多个catch子句来处理多种异常类型。如果存在多个catch子句，异常处理机制会按照它们在代码中出现的顺序尝试匹配：

```
std::string str("foo");

try {
    str.reserve(2); // 预留额外容量，可能抛出 std::length_error
    str.at(10); // 访问元素，可能抛出 std::out_of_range
} catch (const std::length_error& e) {
    std::cout << e.what();
} catch (const std::out_of_range& e) {
    std::cout << e.what();
}
```

派生自同一基类的异常类可以用一个针对该公共基类的catch子句捕获。上述示例可以用一个针对std::exception的catch子句替代针对std::length_error和std::out_of_range的两个catch子句：

```
std::string str("foo");

try {
    str.reserve(2); // 预留额外容量，可能抛出 std::length_error
    str.at(10); // 访问元素，可能抛出 std::out_of_range
} catch (const std::exception& e) {
    std::cout << e.what();
}
```

由于catch子句是按顺序尝试的，请确保先写更具体的catch子句，否则你的异常处理代码可能永远不会被调用：

```
try {
    /* 抛出异常的代码省略。 */
} catch (const std::exception& e) {
    /* 处理所有类型为 std::exception 的异常。 */
} catch (const std::runtime_error& e) {
```

Chapter 72: Exceptions

Section 72.1: Catching exceptions

A `try/catch` block is used to catch exceptions. The code in the `try` section is the code that may throw an exception, and the code in the `catch` clause(s) handles the exception.

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

int main() {
    std::string str("foo");

    try {
        str.at(10); // access element, may throw std::out_of_range
    } catch (const std::out_of_range& e) {
        // what() is inherited from std::exception and contains an explanatory message
        std::cout << e.what();
    }
}
```

Multiple `catch` clauses may be used to handle multiple exception types. If multiple `catch` clauses are present, the exception handling mechanism tries to match them **in order** of their appearance in the code:

```
std::string str("foo");

try {
    str.reserve(2); // reserve extra capacity, may throw std::length_error
    str.at(10); // access element, may throw std::out_of_range
} catch (const std::length_error& e) {
    std::cout << e.what();
} catch (const std::out_of_range& e) {
    std::cout << e.what();
}
```

Exception classes which are derived from a common base class can be caught with a single `catch` clause for the common base class. The above example can replace the two `catch` clauses for `std::length_error` and `std::out_of_range` with a single clause for `std::exception`:

```
std::string str("foo");

try {
    str.reserve(2); // reserve extra capacity, may throw std::length_error
    str.at(10); // access element, may throw std::out_of_range
} catch (const std::exception& e) {
    std::cout << e.what();
}
```

Because the `catch` clauses are tried in order, be sure to write more specific catch clauses first, otherwise your exception handling code might never get called:

```
try {
    /* Code throwing exceptions omitted. */
} catch (const std::exception& e) {
    /* Handle all exceptions of type std::exception. */
} catch (const std::runtime_error& e) {
```

```

/* 这段代码永远不会执行，因为 std::runtime_error 继承自 std::exception,
所有类型为 std::exception 的异常已经被前面的 catch 子句捕获。*/
}

```

另一种可能是捕获所有异常的处理器，它会捕获任何抛出的对象：

```

try {
    throw 10;
} 捕获 (...) {
std::cout << "捕获到异常";
}

```

第72.2节：重新抛出（传播）异常

有时你想对捕获的异常做一些处理（比如写日志或打印警告），然后让它继续向上传递到上层作用域进行处理。为此，你可以重新抛出你捕获的任何异常：

```

try {
... // 这里是一些代码
} catch (const SomeException& e) {
    std::cout << "捕获到异常";
    throw;
}

```

使用 `throw;` 不带参数将重新抛出当前捕获的异常。

版本 ≥ C++11

要重新抛出受管理的 `std::exception_ptr`，C++ 标准库提供了 `rethrow_exception` 函数，可以通过在程序中包含 `<exception>` 头文件来使用。

```

#include <iostream>
#include <string>
#include <exception>
#include <stdexcept>

void handle_eptr(std::exception_ptr eptr) // 传值方式是可以的
{
    try {
        if (eptr) {
            std::rethrow_exception(eptr);
        }
    } catch(const std::exception& e) {
        std::cout << "捕获异常 " << e.what() << "\n";
    }
}

std::exception_ptr eptr;
try {
    std::string().at(1); // 这会生成一个 std::out_of_range 异常
} catch(...) {
    eptr = std::current_exception(); // 捕获异常
}
handle_eptr(eptr);
} // 当 eptr 被销毁时，这里会调用 std::out_of_range 的析构函数

```

```

/* This block of code will never execute, because std::runtime_error inherits
from std::exception, and all exceptions of type std::exception were already
caught by the previous catch clause.*/
}

```

另一种可能性是捕获-all 处理器，它会捕获任何抛出的对象：

```

try {
    throw 10;
} catch (...) {
    std::cout << "caught an exception";
}

```

Section 72.2: Rethrow (propagate) exception

Sometimes you want to do something with the exception you catch (like write to log or print a warning) and let it bubble up to the upper scope to be handled. To do so, you can rethrow any exception you catch:

```

try {
    ... // some code here
} catch (const SomeException& e) {
    std::cout << "caught an exception";
    throw;
}

```

Using `throw;` without arguments will re-throw the currently caught exception.

Version ≥ C++11

To rethrow a managed `std::exception_ptr`, the C++ Standard Library has the `rethrow_exception` function that can be used by including the `<exception>` header in your program.

```

#include <iostream>
#include <string>
#include <exception>
#include <stdexcept>

void handle_eptr(std::exception_ptr eptr) // passing by value is ok
{
    try {
        if (eptr) {
            std::rethrow_exception(eptr);
        }
    } catch(const std::exception& e) {
        std::cout << "Caught exception " << e.what() << "\n";
    }
}

int main()
{
    std::exception_ptr eptr;
    try {
        std::string().at(1); // this generates an std::out_of_range
    } catch(...) {
        eptr = std::current_exception(); // capture
    }
    handle_eptr(eptr);
} // destructor for std::out_of_range called here, when the eptr is destructed

```

第72.3节：最佳实践：按值抛出，按常量引用捕获

一般来说，按值抛出（而非指针）被认为是良好实践，但捕获时应使用（const）引用。

```
try {
    // throw new std::runtime_error("Error!"); // 不要这样做!
    // 这会在堆上创建一个异常对象// 你需要捕获
    // 指针并自行管理内存。// 这可能导致内存泄漏！

    throw std::runtime_error("Error!");
} catch (const std::runtime_error& e) {
    std::cout << e.what() << std::endl;
}
```

捕获异常时使用引用的一个原因是，它消除了在传递到catch块（或传播到其他catch块）时重建对象的需要。通过引用捕获异常还允许多态地处理异常，避免对象切片。然而，如果你重新抛出异常（例如throw e；，见下面示例），仍然可能发生对象切片，因为throw e语句会根据声明的类型复制异常对象：

```
#include <iostream>

struct BaseException {
    virtual const char* what() const { return "BaseException"; }
};

struct DerivedException : BaseException {
    // 这里"virtual"关键字是可选的
    virtual const char* what() const { return "DerivedException"; }
};

int main(int argc, char** argv) {
    try {
        try {
            throw DerivedException();
        } catch (const BaseException& e) {
            std::cout << "第一个catch块: " << e.what() << std::endl;
            // 输出 ==> 第一个catch块: DerivedException

            throw e; // 这会将异常变为BaseException
                    // 而不是原始的DerivedException !
        }
    } catch (const BaseException& e) {
        std::cout << "第二个catch块: " << e.what() << std::endl;
        // 输出 ==> 第二个catch块: BaseException
    }
    return 0;
}
```

如果您确定不会对异常进行任何更改（例如添加信息或修改消息），通过常量引用捕获异常可以让编译器进行优化，从而提高性能。

但这仍然可能导致对象切片（如上例所示）。

警告：注意在catch块中抛出意外异常，尤其是与分配额外内存或资源相关的异常。例如，构造logic_error、runtime_error或它们的子类时，可能会因内存耗尽而抛出bad_alloc

Section 72.3: Best practice: throw by value, catch by const reference

In general, it is considered good practice to throw by value (rather than by pointer), but catch by (const) reference.

```
try {
    // throw new std::runtime_error("Error!"); // Don't do this!
    // This creates an exception object
    // on the heap and would require you to catch the
    // pointer and manage the memory yourself. This can
    // cause memory leaks!

    throw std::runtime_error("Error!");
} catch (const std::runtime_error& e) {
    std::cout << e.what() << std::endl;
}
```

One reason why catching by reference is a good practice is that it eliminates the need to reconstruct the object when being passed to the catch block (or when propagating through to other catch blocks). Catching by reference also allows the exceptions to be handled polymorphically and avoids object slicing. However, if you are rethrowing an exception (like `throw e;`, see example below), you can still get object slicing because the `throw e;` statement makes a copy of the exception as whatever type is declared:

```
#include <iostream>

struct BaseException {
    virtual const char* what() const { return "BaseException"; }
};

struct DerivedException : BaseException {
    // "virtual" keyword is optional here
    virtual const char* what() const { return "DerivedException"; }
};

int main(int argc, char** argv) {
    try {
        try {
            throw DerivedException();
        } catch (const BaseException& e) {
            std::cout << "First catch block: " << e.what() << std::endl;
            // Output ==> First catch block: DerivedException

            throw e; // This changes the exception to BaseException
                    // instead of the original DerivedException!
        }
    } catch (const BaseException& e) {
        std::cout << "Second catch block: " << e.what() << std::endl;
        // Output ==> Second catch block: BaseException
    }
    return 0;
}
```

If you are sure that you are not going to do anything to change the exception (like add information or modify the message), catching by const reference allows the compiler to make optimizations and can improve performance. But this can still cause object splicing (as seen in the example above).

Warning: Beware of throwing unintended exceptions in `catch` blocks, especially related to allocating extra memory or resources. For example, constructing `logic_error`, `runtime_error` or their subclasses might throw `bad_alloc`

在复制异常字符串时，I/O流在设置了相应异常掩码的情况下进行日志记录时可能会抛出异常，等等。

第72.4节：自定义异常

您不应该抛出原始值作为异常，而应使用标准异常类之一或自定义异常类。

继承自std::exception的自定义异常类是一个不错的选择。下面是一个直接继承自std::exception的自定义异常类：

```
#include <exception>

class Except: virtual public std::exception {

protected:

    int error_number;           ///<-- 错误编号
    int error_offset;          ///<-- 错误偏移
    std::string error_message;  ///<-- 错误信息

public:

    /** 构造函数 (C++ STL 字符串, 整数, 整数)。
     * @param msg 错误信息
     * @param err_num 错误编号
     * @param err_off 错误偏移量
     */
    explicit
    Except(const std::string& msg, int err_num, int err_off):
        error_number(err_num),
        error_offset(err_off),
        error_message(msg)
    {}

    /** 析构函数。
     * 虚函数以支持子类化。
     */
    virtual ~Except() throw () {}

    /** 返回指向 (常量) 错误描述的指针。
     * @return 指向 const char* 的指针。底层内存
     *         由 Except 对象持有。调用者不得
     *         试图释放该内存。
     */
    virtual const char* what() const throw () {
        return error_message.c_str();
    }

    /** 返回错误编号。
     * @return #error_number
     */
    virtual int getErrorNumber() const throw() {
        return error_number;
    }

    /** 返回错误偏移量。
     * @return #error_offset
     */
    virtual int getErrorOffset() const throw() {
```

due to memory running out when copying the exception string, I/O streams might throw during logging with respective exception masks set, etc.

Section 72.4: Custom exception

You shouldn't throw raw values as exceptions, instead use one of the standard exception classes or make your own.

Having your own exception class inherited from `std::exception` is a good way to go about it. Here's a custom exception class which directly inherits from `std::exception`:

```
#include <exception>

class Except: virtual public std::exception {

protected:

    int error_number;           ///<-- Error number
    int error_offset;          ///<-- Error offset
    std::string error_message;  ///<-- Error message

public:

    /** Constructor (C++ STL string, int, int).
     * @param msg The error message
     * @param err_num Error number
     * @param err_off Error offset
     */
    explicit
    Except(const std::string& msg, int err_num, int err_off):
        error_number(err_num),
        error_offset(err_off),
        error_message(msg)
    {}

    /** Destructor.
     * Virtual to allow for subclassing.
     */
    virtual ~Except() throw () {}

    /** Returns a pointer to the (constant) error description.
     * @return A pointer to a const char*. The underlying memory
     *         is in possession of the Except object. Callers must
     *         not attempt to free the memory.
     */
    virtual const char* what() const throw () {
        return error_message.c_str();
    }

    /** Returns error number.
     * @return #error_number
     */
    virtual int getErrorNumber() const throw() {
        return error_number;
    }

    /** Returns error offset.
     * @return #error_offset
     */
    virtual int getErrorOffset() const throw() {
```

```

    return error_offset;
}

};

```

一个示例的抛出捕获：

```

try {
    throw(Except("无法完成您期望的操作", -12, -34));
} catch (const Except& e) {
std::cout<<e.what()
    <<"错误编号: "<<e.getErrorNumber()<<"错误偏移: "
    <<e.getErrorOffset();}

```

由于你不仅仅是抛出一个简单的错误信息，还提供了一些其他值来表示错误的具体内容，你的错误处理变得更加高效且有意义。

有一个异常类可以让你很好地处理错误信息：`std::runtime_error`

你也可以从这个类继承：

```

#include <stdexcept>

class Except: virtual public std::runtime_error {

protected:

    int error_number;           ///< Error number
    int error_offset;          ///< Error offset

public:

    /** 构造函数 (C++ STL 字符串, 整数, 整数)。
    * @param msg 错误信息
    * @param err_num 错误编号
    * @param err_off 错误偏移量
    */
    explicit
    Except(const std::string& msg, int err_num, int err_off):
        std::runtime_error(msg)
    {
        error_number = err_num;
        error_offset = err_off;
    }

    /** 析构函数。
    * 虚函数以支持子类化。
    */
    virtual ~Except() throw () {}

    /** 返回错误编号。
    * @return #error_number
    */
    virtual int getErrorNumber() const throw() {
        return error_number;
    }

    /** 返回错误偏移量。
    */

```

```

    return error_offset;
}

};

```

An example throw catch:

```

try {
    throw(Except("Couldn't do what you were expecting", -12, -34));
} catch (const Except& e) {
    std::cout<<e.what()
        <<"\nError number: "<<e.getErrorNumber()
        <<"\nError offset: "<<e.getErrorOffset();
}

```

As you are not only just throwing a dumb error message, also some other values representing what the error exactly was, your error handling becomes much more efficient and meaningful.

There's an exception class that let's you handle error messages nicely :`std::runtime_error`

You can inherit from this class too:

```

#include <stdexcept>

class Except: virtual public std::runtime_error {

protected:

    int error_number;           ///< Error number
    int error_offset;          ///< Error offset

public:

    /** Constructor (C++ STL string, int, int).
    * @param msg The error message
    * @param err_num Error number
    * @param err_off Error offset
    */
    explicit
    Except(const std::string& msg, int err_num, int err_off):
        std::runtime_error(msg)
    {
        error_number = err_num;
        error_offset = err_off;
    }

    /** Destructor.
    * Virtual to allow for subclassing.
    */
    virtual ~Except() throw () {}

    /** Returns error number.
    * @return #error_number
    */
    virtual int getErrorNumber() const throw() {
        return error_number;
    }

    /** Returns error offset.
    */

```

```

* @return #error_offset
*/
virtual int getErrorOffset() const throw() {
    return error_offset;
}

};

```

请注意，我没有重写基类 (`std::runtime_error`) 中的`what()`函数，也就是说我们将使用基类版本的`what()`。如果你有进一步的需求，可以重写它。

第72.5节 : `std::uncaught_exceptions`

版本 ≥ C++17

C++17引入了`int std::uncaught_exceptions()` (替代了有限的`bool std::uncaught_exception()`)，用于了解当前有多少异常尚未被捕获。这使得类可以判断自己是否在栈展开期间被销毁。

```

#include <exception>
#include <string>
#include <iostream>

// 在销毁时应用更改：
// 异常（失败）时回滚
// 否则提交（成功）
类 Transaction
{
    公共:
        Transaction(const std::string & s) : message(s) {}
        Transaction(const Transaction&) = delete;
        Transaction& operator =(const Transaction&) = delete; void Commit()
        { std::cout << message << ": Commit"; } void RollBack() noexcept(true) { std::cout << message << ": Rollback"; }

    // ...

    ~Transaction() {
        if (uncaughtExceptionCount == std::uncaught_exceptions()) {
            Commit(); // 可能会抛出异常。
        } else { // 当前堆栈正在展开
            RollBack();
        }
    }

    私有:
        std::string message;
        int uncaughtExceptionCount = std::uncaught_exceptions();
};

class Foo
{
    公共:
        ~Foo() {
            try {
                Transaction transaction("In ~Foo"); // 提交,
                // 即使存在未捕获的异常
                // ...
            } catch (const std::exception& e) {
                std::cerr << "exception/~Foo:" << e.what() << std::endl;
            }
        }
};

```

```

* @return #error_offset
*/
virtual int getErrorOffset() const throw() {
    return error_offset;
}

};

```

Note that I haven't overridden the `what()` function from the base class (`std::runtime_error`) i.e we will be using the base class's version of `what()`. You can override it if you have further agenda.

Section 72.5: `std::uncaught_exceptions`

Version ≥ C++17

C++17 introduces `int std::uncaught_exceptions()` (to replace the limited `bool std::uncaught_exception()`) to know how many exceptions are currently uncaught. That allows for a class to determine if it is destroyed during a stack unwinding or not.

```

#include <exception>
#include <string>
#include <iostream>

// Apply change on destruction:
// Rollback in case of exception (failure)
// Else Commit (success)
class Transaction
{
    公共:
        Transaction(const std::string& s) : message(s) {}
        Transaction(const Transaction&) = delete;
        Transaction& operator =(const Transaction&) = delete;
        void Commit() { std::cout << message << ": Commit\n"; }
        void RollBack() noexcept(true) { std::cout << message << ": Rollback\n"; }

    // ...

    ~Transaction() {
        if (uncaughtExceptionCount == std::uncaught_exceptions()) {
            Commit(); // May throw.
        } else { // current stack unwinding
            RollBack();
        }
    }

    私有:
        std::string message;
        int uncaughtExceptionCount = std::uncaught_exceptions();
};

class Foo
{
    公共:
        ~Foo() {
            try {
                Transaction transaction("In ~Foo"); // Commit,
                // even if there is an uncaught exception
                // ...
            } catch (const std::exception& e) {
                std::cerr << "exception/~Foo:" << e.what() << std::endl;
            }
        }
};

```

```

    }
};

int main()
{
    try {
        事务 transaction("主函数"); // 回滚
        Foo foo; // ~Foo 提交其事务。
        //...
        throw std::runtime_error("错误");
    } catch (const std::exception& e) {
        std::cerr << "异常/主函数:" << e.what() << std::endl;
    }
}

```

输出：

```

在 ~Foo: 提交
在主函数: 回滚
异常/主函数: 错误

```

第72.6节：常规函数的函数尝试块

```

void 带尝试块的函数()
try
{
    // 尝试块主体
}
catch (...)
{
    // 捕获块主体
}

```

等价于

```

void function_with_try_block()
{
    尝试
    {
        // 尝试块主体
    }
    catch (...)
    {
        // 捕获块主体
    }
}

```

请注意，对于构造函数和析构函数，行为有所不同，因为catch块无论如何都会重新抛出异常（如果catch块体内没有其他throw，则抛出捕获的异常）。

函数main允许像其他函数一样拥有函数try块，但main的函数try块不会捕获在非局部静态变量构造期间或任何静态变量析构期间发生的异常。相反，会调用std::terminate。

第72.7节：嵌套异常

版本 ≥ C++11

```

    }
};

int main()
{
    try {
        Transaction transaction("In main"); // RollBack
        Foo foo; // ~Foo commit its transaction.
        //...
        throw std::runtime_error("Error");
    } catch (const std::exception& e) {
        std::cerr << "exception/main:" << e.what() << std::endl;
    }
}

```

Output:

```

In ~Foo: Commit
In main: Rollback
exception/main:Error

```

Section 72.6: Function Try Block for regular function

```

void function_with_try_block()
try
{
    // try block body
}
catch (...)
{
    // catch block body
}

```

Which is equivalent to

```

void function_with_try_block()
{
    尝试
    {
        // try block body
    }
    catch (...)
    {
        // catch block body
    }
}

```

Note that for constructors and destructors, the behavior is different as the catch block re-throws an exception anyway (the caught one if there is no other throw in the catch block body).

The function `main` is allowed to have a function try block like any other function, but `main`'s function try block will not catch exceptions that occur during the construction of a non-local static variable or the destruction of any static variable. Instead, `std::terminate` is called.

Section 72.7: Nested exception

Version ≥ C++11

在异常处理过程中，有一种常见用例是从低级函数（例如文件系统错误或数据传输错误）捕获通用异常，然后抛出更具体的高级异常，表示某些高级操作无法执行（例如无法在网络上发布照片）。

这允许异常处理针对高级操作的具体问题做出反应，同时仅凭错误和消息，程序员可以找到应用程序中异常发生的位置。该方案的缺点是异常调用栈被截断，原始异常丢失。这迫使开发者手动将原始异常的文本包含到新创建的异常中。

嵌套异常旨在通过将描述原因的低级异常附加到描述特定情况下含义的高级异常上来解决该问题。

std::nested_exception通过std::throw_with_nested实现异常嵌套：

```
#include <stdexcept>
#include <exception>
#include <string>
#include <fstream>
#include <iostream>

struct MyException
{
    MyException(const std::string& message) : message(message) {}
    std::string message;
};

void print_current_exception(int level)
{
    try {
        throw;
    } 捕获 (const std::exception& e) {
        std::cerr << std::string(level, ' ') << "exception: " << e.what() << ";";
    } catch (const MyException& e) {
        std::cerr << std::string(level, ' ') << "MyException: " << e.message << ";";
    } catch (...) {
        std::cerr << "Unknown exception";
    }
}

void print_current_exception_with_nested(int level = 0)
{
    try {
        throw;
    } 捕获 (...) {
        print_current_exception(level);
    }
    try {
        throw;
    } catch (const std::nested_exception& nested) {
        try {
            nested.rethrow_nested();
        } catch (...) {
            print_current_exception_with_nested(level + 1); // 递归
        }
    } 捕获 (...) {
        // 空 // 结束递归
    }
}

// 示例函数，捕获异常并将其包装在嵌套异常中
void 打开文件(const std::string& s)
```

During exception handling there is a common use case when you catch a generic exception from a low-level function (such as a filesystem error or data transfer error) and throw a more specific high-level exception which indicates that some high-level operation could not be performed (such as being unable to publish a photo on Web). This allows exception handling to react to specific problems with high level operations and also allows, having only one error message, the programmer to find a place in the application where an exception occurred. Downside of this solution is that exception callstack is truncated and original exception is lost. This forces developers to manually include text of original exception into a newly created one.

Nested exceptions aim to solve the problem by attaching low-level exception, which describes the cause, to a high-level exception, which describes what it means in this particular case.

std::nested_exception allows to nest exceptions thanks to std::throw_with_nested:

```
#include <stdexcept>
#include <exception>
#include <string>
#include <fstream>
#include <iostream>

struct MyException
{
    MyException(const std::string& message) : message(message) {}
    std::string message;
};

void print_current_exception(int level)
{
    try {
        throw;
    } catch (const std::exception& e) {
        std::cerr << std::string(level, ' ') << "exception: " << e.what() << '\n';
    } catch (const MyException& e) {
        std::cerr << std::string(level, ' ') << "MyException: " << e.message << '\n';
    } catch (...) {
        std::cerr << "Unknown exception\n";
    }
}

void print_current_exception_with_nested(int level = 0)
{
    try {
        throw;
    } catch (...) {
        print_current_exception(level);
    }
    try {
        throw;
    } catch (const std::nested_exception& nested) {
        try {
            nested.rethrow_nested();
        } catch (...) {
            print_current_exception_with_nested(level + 1); // recursion
        }
    } catch (...) {
        // Empty // End recursion
    }
}

// sample function that catches an exception and wraps it in a nested exception
void open_file(const std::string& s)
```

```

{
    try {
std::ifstream 文件(s);
    文件.异常(std::ios_base::failbit);
    } 捕获(...) {
std::throw_with_nested(MyException{"无法打开 " + s});
    }
}

```

```

{
    try {
        std::ifstream file(s);
        file.exceptions(std::ios_base::failbit);
    } catch(...) {
        std::throw_with_nested(MyException{"Couldn't open " + s});
    }
}

// sample function that catches an exception and wraps it in a nested exception
void run()
{
    try {
        open_file("nonexistent.file");
    } catch(...) {
        std::throw_with_nested( std::runtime_error("run() failed") );
    }
}

// runs the sample function above and prints the caught exception
int main()
{
    try {
        run();
    } catch(...) {
        print_current_exception_with_nested();
    }
}

```

Possible output:

```

exception: run() failed
MyException: Couldn't open nonexistent.file
exception: basic_ios::clear

```

If you work only with exceptions inherited from `std::exception`, code can even be simplified.

Section 72.8: Function Try Blocks In constructor

The only way to catch exception in initializer list:

```

struct A : public B
{
    A() try : B(), foo(1), bar(2)
    {
        // constructor body
    }
    catch (...)
    {
        // exceptions from the initializer list and constructor are caught here
        // if no exception is thrown here
        // then the caught exception is re-thrown.
    }

private:
    Foo foo;
    Bar bar;
};

```

第72.9节：析构函数中的函数尝试块

```
结构体 A
{
    ~A() noexcept(false) try
    {
        // 析构函数体
    }
    catch (...)
    {
        // 析构函数体中的异常在此捕获
        // 如果此处没有抛出异常
        // 则捕获的异常会被重新抛出。
    }
};
```

注意，虽然这是可能的，但在析构函数中抛出异常时需要非常小心，因为如果在栈展开期间调用的析构函数抛出异常，`std::terminate` 会被调用。

Section 72.9: Function Try Blocks In destructor

```
struct A
{
    ~A() noexcept(false) try
    {
        // destructor body
    }
    catch (...)
    {
        // exceptions of destructor body are caught here
        // if no exception is thrown here
        // then the caught exception is re-thrown.
    }
};
```

Note that, although this is possible, one needs to be very careful with throwing from destructor, as if a destructor called during stack unwinding throws an exception, `std::terminate` is called.

第73章：Lambda表达式

参数

默认捕获

指定所有未列出变量的捕获方式。可以是=（按值捕获）或&（按引用捕获）。如果省略，未列出的变量在lambda体内将无法访问。默认捕获（default-capture）必须位于捕获列表（capture-list）之前。

捕获列表

指定局部变量在lambda-body中的访问方式。没有前缀的变量按值捕获。带有&前缀的变量按引用捕获。在类方法中，可以使用this使其所有成员通过引用访问。未列出的变量不可访问，除非列表前有default-capture。

参数列表

指定 lambda 函数的参数。

可变的

（可选）通常按值捕获的变量是const的。指定mutable使它们变为非const。对这些变量的更改会在调用之间保留。

throw-specification (可选)

指定lambda函数的异常抛出行为。例如：noexcept 或 throw(std::exception)。

属性

（可选）lambda 函数的任何属性。例如，如果 *lambda-body* 总是抛出异常，则可以使用 [[noreturn]]。

-> 返回类型

（可选）指定lambda函数的返回类型。当编译器无法确定返回类型时，必须指定。

lambda主体

包含lambda函数实现的代码块。

第73.1节：什么是lambda表达式？

一个lambda表达式提供了一种简洁的方式来创建简单的函数对象。lambda表达式是一个纯右值，其结果对象称为闭包对象，表现得像一个函数对象。

“lambda表达式”这一名称来源于lambda演算，这是一种数学形式主义，由阿隆佐·丘奇于1930年代发明，用于研究逻辑和可计算性问题。lambda演算构成了函数式编程语言LISP的基础。与lambda演算和LISP相比，C++的lambda表达式具有匿名和捕获周围上下文变量的特性，但它们不具备操作和返回函数的能力。

lambda表达式通常用作接受可调用对象的函数的参数。这比创建一个仅作为参数传递使用的具名函数更简单。在这种情况下，通常更倾向于使用lambda表达式，因为它们允许内联定义函数对象。

lambda通常由三部分组成：捕获列表[]、可选的参数列表()和函数体{}，这三部分都可以为空：

```
[](){} // 一个空的lambda，既不执行任何操作也不返回任何值
```

捕获列表

[] 是捕获列表。默认情况下，lambda 无法访问外层作用域的变量。捕获一个变量使其在 lambda 内部可访问，可以是拷贝也可以是引用。被捕获的变量成为 lambda 的一部分；与函数参数不同，调用 lambda 时不必传递它们。

```
int a = 0; // 定义一个整型变量
auto f = []() { return a*9; }; // 错误：无法访问 'a'
auto f = [a]() { return a*9; }; // 正确，'a' 被按值“捕获”
auto f = [&a]() { return a++; }; // 正确，'a' 被按引用“捕获”
// 注意：程序员有责任
// 确保在调用 lambda 之前 a 不被销毁
```

Chapter 73: Lambdas

Parameter

default-capture

Specifies how all non-listed variables are captured. Can be = (capture by value) or & (capture by reference). If omitted, non-listed variables are inaccessible within the *lambda-body*. The *default-capture* must precede the *capture-list*.

capture-list

Specifies how local variables are made accessible within the *lambda-body*. Variables without prefix are captured by value. Variables prefixed with & are captured by reference. Within a class method, this can be used to make all its members accessible by reference. Non-listed variables are inaccessible, unless the list is preceded by a *default-capture*.

argument-list

Specifies the arguments of the lambda function.

mutable

(optional) Normally variables captured by value are const. Specifying mutable makes them non-const. Changes to those variables are retained between calls.

throw-specification

(optional) Specifies the exception throwing behavior of the lambda function. For example: noexcept or throw(std::exception).

attributes

(optional) Any attributes for the lambda function. For example, if the *lambda-body* always throws an exception then [[noreturn]] can be used.

-> return-type

(optional) Specifies the return type of the lambda function. Required when the return type cannot be determined by the compiler.

lambda-body

A code block containing the implementation of the lambda function.

Section 73.1: What is a lambda expression?

A **lambda expression** provides a concise way to create simple function objects. A lambda expression is a prvalue whose result object is called closure object, which behaves like a function object.

The name 'lambda expression' originates from lambda calculus, which is a mathematical formalism invented in the 1930s by Alonzo Church to investigate questions about logic and computability. Lambda calculus formed the basis of LISP, a functional programming language. Compared to lambda calculus and LISP, C++ lambda expressions share the properties of being unnamed, and to capture variables from the surrounding context, but they lack the ability to operate on and return functions.

A lambda expression is often used as an argument to functions that take a callable object. That can be simpler than creating a named function, which would be only used when passed as the argument. In such cases, lambda expressions are generally preferred because they allow defining the function objects inline.

A lambda consists typically of three parts: a capture list [] , an optional parameter list () and a body {}, all of which can be empty:

```
[](){} // An empty lambda, which does and returns nothing
```

Capture list

[] is the **capture list**. By default, variables of the enclosing scope cannot be accessed by a lambda. Capturing a variable makes it accessible inside the lambda, either as a copy or as a reference. Captured variables become a part of the lambda; in contrast to function arguments, they do not have to be passed when calling the lambda.

```
int a = 0; // Define an integer variable
auto f = []() { return a*9; }; // Error: 'a' cannot be accessed
auto f = [a]() { return a*9; }; // OK, 'a' is "captured" by value
auto f = [&a]() { return a++; }; // OK, 'a' is "captured" by reference
// Note: It is the responsibility of the programmer
// to ensure that a is not destroyed before the
```

```
// lambda 被调用之前, a 不被销毁。  
auto b = f();  
// 调用 lambda 函数。a 从捕获列表中获取,  
这里不需要传递。
```

参数列表

() 是参数列表，几乎与普通函数相同。如果 lambda 不接受参数，这对括号可以省略（除非需要声明 lambda 为 mutable）。以下两个 lambda 等价：

```
auto call_foo = [x](){ x.foo(); };  
auto call_foo2 = [x]{ x.foo(); };
```

版本 ≥ C++14

参数列表可以使用占位类型 auto 代替具体类型。这样，该参数表现得像函数模板的模板参数。以下 lambda 在泛型代码中对向量排序时等价：

```
auto sort_cpp11 = [](std::vector<T>::const_reference lhs, std::vector<T>::const_reference rhs) {  
    return lhs < rhs; };  
auto sort_cpp14 = [](const auto &lhs, const auto &rhs) { return lhs < rhs; };
```

函数体

{ } 是函数体，与普通函数中的相同。

调用 lambda

lambda 表达式的结果对象是一个闭包，可以使用 operator() 调用（与其他函数对象相同）：

```
int multiplier = 5;  
auto timesFive = [multiplier](int a) { return a * multiplier; };  
std::cout << timesFive(2); // 输出 10  
  
multiplier = 15;  
std::cout << timesFive(2); // 仍然输出 2*5 == 10
```

返回类型

默认情况下，lambda 表达式的返回类型是推断出来的。

```
[](){ return true; };
```

在这种情况下，返回类型是 bool。

你也可以使用以下语法手动指定返回类型：

```
[]() -> bool { return true; };
```

可变Lambda

在 Lambda 中按值捕获的对象默认是不可变的。这是因为生成的闭包对象的 operator() 默认是 const 的。

```
auto func = [c = 0](){++c; std::cout << c;}; // 编译失败, 因为 ++c  
// 试图修改Lambda的状态。
```

```
// lambda is called.  
auto b = f();  
not passed here.
```

Parameter list

() 是 **参数列表**，几乎与普通函数相同。如果 lambda 不接受参数，这对括号可以省略（除非需要声明 lambda 为 mutable）。这两个 lambda 是等价的：

```
auto call_foo = [x](){ x.foo(); };  
auto call_foo2 = [x]{ x.foo(); };
```

Version ≥ C++14

The parameter list can use the placeholder type auto instead of actual types. By doing so, this argument behaves like a template parameter of a function template. Following lambdas are equivalent when you want to sort a vector in generic code:

```
auto sort_cpp11 = [](std::vector<T>::const_reference lhs, std::vector<T>::const_reference rhs) {  
    return lhs < rhs; };  
auto sort_cpp14 = [](const auto &lhs, const auto &rhs) { return lhs < rhs; };
```

Function body

{ } 是 **body**，与普通函数中的相同。

Calling a lambda

A lambda expression's result object is a closure，which can be called using the operator() (as with other function objects):

```
int multiplier = 5;  
auto timesFive = [multiplier](int a) { return a * multiplier; };  
std::cout << timesFive(2); // Prints 10  
  
multiplier = 15;  
std::cout << timesFive(2); // Still prints 2*5 == 10
```

Return Type

By default, the return type of a lambda expression is deduced.

```
[](){ return true; };
```

In this case the return type is bool.

You can also manually specify the return type using the following syntax:

```
[]() -> bool { return true; };
```

Mutable Lambda

Objects captured by value in the lambda are by default immutable. This is because the operator() of the generated closure object is const by default.

```
auto func = [c = 0](){++c; std::cout << c;}; // fails to compile because ++c  
// tries to mutate the state of
```

// Lambda的状态。

可以通过使用关键字mutable来允许修改，这会使闭包对象的operator()变为非const：

```
auto func = [c = 0]() mutable {++c; std::cout << c;};
```

如果与返回类型一起使用，mutable 位于其前面。

```
auto func = [c = 0]() mutable -> int {++c; std::cout << c; return c;};
```

一个说明lambda表达式有用性的示例

C++11之前：

版本 < C++11

```
// 用于比较的通用函数对象
struct islessthan
{
    islessthan(int threshold) : _threshold(threshold) {}

    bool operator()(int value) const
    {
        return value < _threshold;
    }
private:
    int _threshold;
};

// 声明一个向量
const int arr[] = { 1, 2, 3, 4, 5 };
std::vector<int> vec(arr, arr+5);

// 查找一个小于给定输入的数字（假设这是函数输入）
int threshold = 10;
std::vector<int>::iterator it = std::find_if(vec.begin(), vec.end(), islessthan(threshold));
```

自 C++11 起：

版本 ≥ C++11

```
// 声明一个向量
std::vector<int> vec{ 1, 2, 3, 4, 5 };

// 查找一个小于给定输入的数字（假设这是函数输入）
int threshold = 10;
auto it = std::find_if(vec.begin(), vec.end(), [threshold](int value) { return value < threshold; });
```

第73.2节：指定返回类型

对于只有单个返回语句，或多个返回语句且表达式类型相同的lambda，编译器可以推断返回类型：

```
// 返回bool，因为"value > 10"是一个比较，结果为布尔值
auto l = [](int value) {
    return value > 10;
}
```

对于具有多个不同类型返回语句的lambda，编译器无法推断返回类型：

// the lambda.

Modifying can be allowed by using the keyword `mutable`, which make the closer object's operator() non-`const`:

```
auto func = [c = 0]() mutable {++c; std::cout << c;};
```

If used together with the return type, `mutable` comes before it.

```
auto func = [c = 0]() mutable -> int {++c; std::cout << c; return c;};
```

An example to illustrate the usefulness of lambdas

Before C++11:

Version < C++11

```
// Generic functor used for comparison
struct islessthan
{
    islessthan(int threshold) : _threshold(threshold) {}

    bool operator()(int value) const
    {
        return value < _threshold;
    }
private:
    int _threshold;
};

// Declare a vector
const int arr[] = { 1, 2, 3, 4, 5 };
std::vector<int> vec(arr, arr+5);

// Find a number that's less than a given input (assume this would have been function input)
int threshold = 10;
std::vector<int>::iterator it = std::find_if(vec.begin(), vec.end(), islessthan(threshold));
```

Since C++11:

Version ≥ C++11

```
// Declare a vector
std::vector<int> vec{ 1, 2, 3, 4, 5 };

// Find a number that's less than a given input (assume this would have been function input)
int threshold = 10;
auto it = std::find_if(vec.begin(), vec.end(), [threshold](int value) { return value < threshold; });
```

Section 73.2: Specifying the return type

For lambdas with a single return statement, or multiple return statements whose expressions are of the same type, the compiler can deduce the return type:

```
// Returns bool, because "value > 10" is a comparison which yields a Boolean result
auto l = [](int value) {
    return value > 10;
}
```

For lambdas with multiple return statements of *different* types, the compiler can't deduce the return type:

```
// 错误：如果lambda的返回类型未指定，返回类型必须匹配
auto l = [](int value) {
    if (value < 10) {
        return 1;
    } else {
        return 1.5;
    }
};
```

在这种情况下，必须显式指定返回类型：

```
// 返回类型显式指定为'double'
auto l = [](int value) -> double {
    if (value < 10) {
        return 1;
    } else {
        return 1.5;
    }
};
```

这些规则与 `auto` 类型推断规则相匹配。未显式指定返回类型的 lambda 从不返回引用，因此如果需要引用类型，也必须显式指定：

```
auto copy = [](X& x) { return x; }; // 'copy' 返回一个 X，复制其输入
auto ref = [](X& x) -> X& { return x; }; // 'ref' 返回一个 X&，不复制
```

第73.3节：按值捕获

如果在捕获列表中指定变量名，lambda 将按值捕获该变量。这意味着为 lambda 生成的闭包类型会存储该变量的副本。这也要求变量的类型必须是可拷贝构造的：

```
int a = 0;

[a]() {
    return a; // 可以，'a' 是按值捕获的
};
```

低于 C++14 版本

```
auto p = std::unique_ptr<T>(...);

[p]() { // 编译错误；`unique_ptr` 不是可拷贝构造的
    return p->createWidget();
};
```

从 C++14 开始，可以在现场初始化变量。这允许仅可移动类型被捕获到 lambda 中。

```
版本 ≥ C++14
auto p = std::make_unique<T>(...);

[p = std::move(p)]() {
    return p->createWidget();
};
```

尽管当通过名称捕获变量时，lambda 会按值捕获变量，但默认情况下这些变量不能在 lambda 函数体内被修改。这是因为闭包类型将 lambda 函数体放在一个 `operator() const` 的声明中。

```
// error: return types must match if lambda has unspecified return type
auto l = [](int value) {
    if (value < 10) {
        return 1;
    } else {
        return 1.5;
    }
};
```

In this case you have to specify the return type explicitly:

```
// The return type is specified explicitly as 'double'
auto l = [](int value) -> double {
    if (value < 10) {
        return 1;
    } else {
        return 1.5;
    }
};
```

The rules for this match the rules for `auto` type deduction. Lambdas without explicitly specified return types never return references, so if a reference type is desired it must be explicitly specified as well:

```
auto copy = [](X& x) { return x; }; // 'copy' returns an X, so copies its input
auto ref = [](X& x) -> X& { return x; }; // 'ref' returns an X&, no copy
```

Section 73.3: Capture by value

If you specify the variable's name in the capture list, the lambda will capture it by value. This means that the generated closure type for the lambda stores a copy of the variable. This also requires that the variable's type be *copy-constructible*:

```
int a = 0;

[a]() {
    return a; // Ok, 'a' is captured by value
};

Version < C++14
auto p = std::unique_ptr<T>(...);

[p]() { // Compile error; `unique_ptr` is not copy-constructible
    return p->createWidget();
};
```

From C++14 on, it is possible to initialize variables on the spot. This allows move only types to be captured in the lambda.

```
Version ≥ C++14
auto p = std::make_unique<T>(...);

[p = std::move(p)]() {
    return p->createWidget();
};
```

Even though a lambda captures variables by value when they are given by their name, such variables cannot be modified within the lambda body by default. This is because the closure type puts the lambda body in a declaration of `operator() const`.

`const` 适用于对闭包类型成员变量的访问，以及作为闭包成员的捕获变量（尽管表面上看起来不是）：

```
int a = 0;

[a]() {
    a = 2;      // 不合法, 'a' 是通过 const 访问的

    decltype(a) a1 = 1;
    a1 = 2; // 合法: 变量 'a1' 不是 const
};
```

要去除 `const`，必须在 lambda 上指定关键字 `mutable`：

```
int a = 0;

[a]() mutable {
    a = 2;      // 可以, 'a' 可以被修改
    return a;
};
```

因为 `a` 是按值捕获的，调用 lambda 所做的任何修改都不会影响外部的 `a`。构造 lambda 时，`a` 的值被复制到 lambda 中，所以 lambda 中的 `a` 副本与外部的 `a` 变量是分开的。

```
int a = 5;
auto plus5Val = [a] (void) { return a + 5; };
auto plus5Ref = [&a] (void) {return a + 5; };

a = 7;
std::cout << a << ", value " << plus5Val() << ", reference " << plus5Ref();
// The result will be "7, value 10, reference 12"
```

第73.4节：递归lambda

假设我们想将欧几里得的 `gcd()` 写成 lambda。作为函数，它是：

```
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a%b);
}
```

但是 lambda 不能递归，它没有办法调用自身。lambda 没有名字，在 lambda 体内使用 `this` 指的是捕获的 `this`（假设 lambda 是在成员函数体内创建的，否则这是错误）。那么我们如何解决这个问题？

使用 `std::function`

我们可以让 lambda 捕获一个尚未构造的 `std::function` 的引用：

```
std::function<int(int, int)> gcd = [&](int a, int b){
    return b == 0 ? a : gcd(b, a%b);
};
```

这可以工作，但应谨慎使用。它很慢（我们现在使用类型擦除而不是直接函数调用），它很脆弱（复制 `gcd` 或返回 `gcd` 会出错，因为 lambda 引用的是原始对象），并且它不适用于泛型 lambda。

The `const` applies to accesses to member variables of the closure type, and captured variables that are members of the closure (all appearances to the contrary):

```
int a = 0;

[a]() {
    a = 2;      // Illegal, 'a' is accessed via `const`

    decltype(a) a1 = 1;
    a1 = 2; // valid: variable 'a1' is not const
};
```

To remove the `const`, you have to specify the keyword `mutable` on the lambda:

```
int a = 0;

[a]() mutable {
    a = 2;      // OK, 'a' can be modified
    return a;
};
```

Because `a` was captured by value, any modifications done by calling the lambda will not affect `a`. The value of `a` was copied into the lambda when it was constructed, so the lambda's copy of `a` is separate from the external `a` variable.

```
int a = 5;
auto plus5Val = [a] (void) { return a + 5; };
auto plus5Ref = [&a] (void) {return a + 5; };

a = 7;
std::cout << a << ", value " << plus5Val() << ", reference " << plus5Ref();
// The result will be "7, value 10, reference 12"
```

Section 73.4: Recursive lambdas

Let's say we wish to write Euclid's `gcd()` as a lambda. As a function, it is:

```
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a%b);
}
```

But a lambda cannot be recursive, it has no way to invoke itself. A lambda has no name and using `this` within the body of a lambda refers to a captured `this` (assuming the lambda is created in the body of a member function, otherwise it is an error). So how do we solve this problem?

Use `std::function`

We can have a lambda capture a reference to a not-yet constructed `std::function`:

```
std::function<int(int, int)> gcd = [&](int a, int b){
    return b == 0 ? a : gcd(b, a%b);
};
```

This works, but should be used sparingly. It's slow (we're using type erasure now instead of a direct function call), it's fragile (copying `gcd` or returning `gcd` will break since the lambda refers to the original object), and it won't work with generic lambdas.

使用两个智能指针：

```
auto gcd_self = std::make_shared<std::unique_ptr< std::function<int(int, int)> >>();
*gcd_self = std::make_unique<std::function<int(int, int)>>(
    [gcd_self](int a, int b){
        return b == 0 ? a : (**gcd_self)(b, a%b);
    });
};
```

这增加了大量的间接调用（这是开销），但它可以被复制/返回，且所有副本共享状态。它确实允许你返回该lambda，并且相比上述方案更不易出错。

使用Y组合子

借助一个简短的工具结构，我们可以解决所有这些问题：

```
template <class F>
struct y_combinator {
    F f; // lambda将存储在这里

    // 一个转发操作符():
    template <class... Args>
    decltype(auto) operator()(Args&&... args) const {
        // 我们将自身传递给 f, 然后是参数。
        // lambda 应该将第一个参数作为 `auto&& recurse` 或类似的形式。
        return f(*this, std::forward<Args>(args)...);
    }

    // 辅助函数，用于推断 lambda 的类型：
    template <class F>
    y_combinator<std::decay_t<F>> make_y_combinator(F&& f) {
        return {std::forward<F>(f)};
    }
    // (注意，在 C++17 中我们可以做得比 `make_` 函数更好)
};
```

我们可以这样实现我们的 gcd：

```
auto gcd = make_y_combinator(
    [](auto&& gcd, int a, int b){
        return b == 0 ? a : gcd(b, a%b);
    });
};
```

y组合子是来自λ演算的一个概念，它允许你在未定义自身名称之前实现递归。这正是λ表达式所面临的问题。

你创建一个以“reurse”为第一个参数的λ表达式。当你想递归时，就将参数传递给reurse。

然后y组合子返回一个函数对象，该对象用它的参数调用该函数，但第一个参数是一个合适的“reurse”对象（即y组合子本身）。它还会将你调用y组合子时的其余参数转发给该λ表达式。

简而言之：

```
autofoo =make_y_combinator( [&](auto&& recurse, some arguments ) {
    // 编写处理某些参数的函数体
    // 当你想递归时，调用 recurse(其他参数)
});
```

Using two smart pointers:

```
auto gcd_self = std::make_shared<std::unique_ptr< std::function<int(int, int)> >>();
*gcd_self = std::make_unique<std::function<int(int, int)>>(
    [gcd_self](int a, int b){
        return b == 0 ? a : (**gcd_self)(b, a%b);
    });
};
```

This adds a lot of indirection (which is overhead), but it can be copied/returned, and all copies share state. It does let you return the lambda, and is otherwise less fragile than the above solution.

Use a Y-combinator

With the help of a short utility struct, we can solve all of these problems:

```
template <class F>
struct y_combinator {
    F f; // the lambda will be stored here

    // a forwarding operator():
    template <class... Args>
    decltype(auto) operator()(Args&&... args) const {
        // we pass ourselves to f, then the arguments.
        // the lambda should take the first argument as `auto&& recurse` or similar.
        return f(*this, std::forward<Args>(args)...);
    }

    // helper function that deduces the type of the lambda:
    template <class F>
    y_combinator<std::decay_t<F>> make_y_combinator(F&& f) {
        return {std::forward<F>(f)};
    }
    // (Be aware that in C++17 we can do better than a `make_` function)
};
```

we can implement our gcd as:

```
auto gcd = make_y_combinator(
    [](auto&& gcd, int a, int b){
        return b == 0 ? a : gcd(b, a%b);
    });
};
```

The y_combinator is a concept from the lambda calculus that lets you have recursion without being able to name yourself until you are defined. This is exactly the problem lambdas have.

You create a lambda that takes "reurse" as its first argument. When you want to recurse, you pass the arguments to recurse.

The y_combinator then returns a function object that calls that function with its arguments, but with a suitable "reurse" object (namely the y_combinator itself) as its first argument. It forwards the rest of the arguments you call the y_combinator with to the lambda as well.

In short:

```
auto foo = make_y_combinator( [&](auto&& recurse, some arguments) {
    // write body that processes some arguments
    // when you want to recurse, call recurse(some other arguments)
});
```

});

这样你就在lambda表达式中实现了递归，且没有严重的限制或显著的开销。

第73.5节：默认捕获

默认情况下，未在捕获列表中显式指定的局部变量，无法从lambda体内访问。然而，可以隐式捕获lambda体中使用的变量名：

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

// 默认按值捕获
[=]() { return a + b; }; // 正确；a 和 b 按值捕获

// 默认按引用捕获
[&]() { return a + b; }; // 正确；a 和 b 按引用捕获
```

显式捕获仍然可以与隐式默认捕获一起使用。显式捕获定义将覆盖默认捕获：

```
int a = 0;
int b = 1;

[=, &b]() {
    a = 2; // 非法；'a' 是按值捕获，且 lambda 不是 'mutable'
    b = 2; // 正确；'b' 是按引用捕获
};
```

第 73.6 节：类 lambda 和 this 的捕获

在类的成员函数中求值的 lambda 表达式隐式地是该类的友元：

```
类 Foo
{
private:
    int i;

公共:
    Foo(int val) : i(val) {}

    // 成员函数的定义
    void Test()
    {
        auto lamb = [](Foo &foo, int val)
        {
            // 修改私有成员变量
            foo.i = val;
        };

        // lamb 被允许访问私有成员，因为它是 Foo 的友元
        lamb(*this, 30);
    }
};
```

这样的 lambda 不仅是该类的友元，它拥有与声明它的类相同的访问权限。

Lambda 可以捕获this指针，表示外部函数被调用的对象实例。这

});

and you have recursion in a lambda with no serious restrictions or significant overhead.

Section 73.5: Default capture

By default, local variables that are not explicitly specified in the capture list, cannot be accessed from within the lambda body. However, it is possible to implicitly capture variables named by the lambda body:

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

// Default capture by value
[=]() { return a + b; }; // OK; a and b are captured by value

// Default capture by reference
[&]() { return a + b; }; // OK; a and b are captured by reference
```

Explicit capturing can still be done alongside implicit default capturing. The explicit capture definition will override the default capture:

```
int a = 0;
int b = 1;

[=, &b]() {
    a = 2; // Illegal; 'a' is capture by value, and lambda is not 'mutable'
    b = 2; // OK; 'b' is captured by reference
};
```

Section 73.6: Class lambdas and capture of this

A lambda expression evaluated in a class' member function is implicitly a friend of that class:

```
class Foo
{
private:
    int i;

public:
    Foo(int val) : i(val) {}

    // definition of a member function
    void Test()
    {
        auto lamb = [](Foo &foo, int val)
        {
            // modification of a private member variable
            foo.i = val;
        };

        // lamb is allowed to access a private member, because it is a friend of Foo
        lamb(*this, 30);
    }
};
```

Such a lambda is not only a friend of that class, it has the same access as the class it is declared within has.

Lambdas can capture the `this` pointer which represents the object instance the outer function was called on. This

通过将this添加到捕获列表中来完成：

```
类 Foo
{
private:
    int i;

公共:
Foo(int val) : i(val) {}

    void Test()
    {
        // 按值捕获 this 指针
        auto lamb = [this](int val)
        {
            i = val;
        };

        lamb(30);
    }
};
```

当this被捕获时，lambda 可以像在其包含类中一样使用包含类的成员名。因此，对这些成员会隐式应用this->操作符。

请注意，this是按值捕获的，但不是类型的值。它是按this的值捕获的，this是一个指针。因此，lambda 不拥有this。如果 lambda 的生命周期超过了创建它的对象的生命周期，lambda 可能会变得无效。

这也意味着 lambda 可以修改this而无需声明为mutable。被const修饰的是指针本身，而不是指针所指向的对象。也就是说，除非外部成员函数本身是const函数。

另外，请注意默认捕获子句，[=]和[&]都会隐式捕获this。它们都是按指针的值捕获的。实际上，当指定了默认捕获时，在捕获列表中显式指定this是错误的。

版本 ≥ C++17

lambda 可以捕获在创建 lambda 时创建的this对象的副本。这是通过添加 *this 到捕获列表中：

```
类 Foo
{
private:
    int i;

公共:
Foo(int val) : i(val) {}

    void Test()
    {
        // 捕获由 this 指针给出的对象的副本
        auto lamb = [*this](int val) mutable
        {
            i = val;
        };

        lamb(30); // 不会改变 this->i
    }
};
```

is done by adding `this` to the capture list:

```
class Foo
{
private:
    int i;

public:
    Foo(int val) : i(val) {}

    void Test()
    {
        // capture the this pointer by value
        auto lamb = [this](int val)
        {
            i = val;
        };

        lamb(30);
    }
};
```

When `this` is captured, the lambda can use member names of its containing class as though it were in its containing class. So an implicit `this->` is applied to such members.

Be aware that `this` is captured by value, but not the value of the type. It is captured by the value of `this`, which is a pointer. As such, the lambda does not own `this`. If the lambda outlives the lifetime of the object that created it, the lambda can become invalid.

This also means that the lambda can modify `this` without being declared `mutable`. It is the pointer which is `const`, not the object being pointed to. That is, unless the outer member function was itself a `const` function.

Also, be aware that the default capture clauses, both [=] and [&], will also capture `this` implicitly. And they both capture it by the value of the pointer. Indeed, it is an error to specify `this` in the capture list when a default is given.

Version ≥ C++17

Lambdas can capture a copy of the `this` object, created at the time the lambda is created. This is done by adding `*this` to the capture list:

```
class Foo
{
private:
    int i;

public:
    Foo(int val) : i(val) {}

    void Test()
    {
        // capture a copy of the object given by the this pointer
        auto lamb = [*this](int val) mutable
        {
            i = val;
        };

        lamb(30); // does not change this->i
    }
};
```

第73.7节：按引用捕获

如果在局部变量名前加上 `&`，则该变量将按引用捕获。概念上，这意味着 lambda 的闭包类型将有一个引用变量，初始化为对 lambda 作用域外对应变量的引用。lambda 函数体中对该变量的任何使用都将引用原始变量：

```
// 声明变量 'a'  
int a = 0;  
  
// 声明一个按引用捕获 'a' 的 lambda  
auto set = [&a]() {  
    a = 1;  
};  
  
set();  
assert(a == 1);
```

关键字 `mutable` 是不需要的，因为 `a` 本身不是 `const`。

当然，通过引用捕获意味着 lambda 必须不逃离它所捕获变量的作用域。所以你可以调用接受函数作为参数的函数，但你不能调用会将 lambda 存储超出你引用作用域的函数。并且你不能返回 lambda。

第73.8节：泛型lambda

版本 \geq C++14

lambda 函数可以接受任意类型的参数。这使得 lambda 更加通用：

```
auto twice = [](auto x){ return x+x; };  
  
int i = twice(2); // i == 4  
std::string s = twice("hello"); // s == "hellohello"
```

这在 C++ 中通过使闭包类型的 `operator()` 重载成为模板函数来实现。以下类型的行为与上述 lambda 闭包等效：

```
struct _unique_lambda_type  
{  
    template<typename T>  
    auto operator()(T x) const {return x + x;}  
};
```

并非所有泛型 lambda 中的参数都必须是泛型的：

```
[](auto x, int y) {return x + y;}
```

这里，`x` 是根据第一个函数参数推导的，而 `y` 始终是 `int` 类型。

泛型 lambda 也可以通过引用传递参数，使用 `auto` 和 `&` 的常规规则。如果泛型参数被声明为 `auto&&`，则这是对传入参数的 转发引用，而不是 `rvalue` 引用：

```
auto lamb1 = [](int &&x) {return x + 5;};  
auto lamb2 = [](auto &&x) {return x + 5;};  
int x = 10;
```

Section 73.7: Capture by reference

If you precede a local variable's name with an `&`, then the variable will be captured by reference. Conceptually, this means that the lambda's closure type will have a reference variable, initialized as a reference to the corresponding variable from outside of the lambda's scope. Any use of the variable in the lambda body will refer to the original variable:

```
// Declare variable 'a'  
int a = 0;  
  
// Declare a lambda which captures 'a' by reference  
auto set = [&a]() {  
    a = 1;  
};  
  
set();  
assert(a == 1);
```

The keyword `mutable` is not needed, because `a` itself is not `const`.

Of course, capturing by reference means that the lambda **must not** escape the scope of the variables it captures. So you could call functions that take a function, but you must not call a function that will *store* the lambda beyond the scope of your references. And you must not return the lambda.

Section 73.8: Generic lambdas

Version \geq C++14

Lambda functions can take arguments of arbitrary types. This allows a lambda to be more generic:

```
auto twice = [](auto x){ return x+x; };  
  
int i = twice(2); // i == 4  
std::string s = twice("hello"); // s == "hellohello"
```

This is implemented in C++ by making the closure type's `operator()` overload a template function. The following type has equivalent behavior to the above lambda closure:

```
struct _unique_lambda_type  
{  
    template<typename T>  
    auto operator()(T x) const {return x + x;}  
};
```

Not all parameters in a generic lambda need be generic:

```
[](auto x, int y) {return x + y;}
```

Here, `x` is deduced based on the first function argument, while `y` will always be `int`.

Generic lambdas can take arguments by reference as well, using the usual rules for `auto` and `&`. If a generic parameter is taken as `auto&&`, this is a *forwarding* reference to the passed in argument and not an *rvalue* reference:

```
auto lamb1 = [](int &&x) {return x + 5;};  
auto lamb2 = [](auto &&x) {return x + 5;};  
int x = 10;
```

```
lamb1(x); // 非法；对于 `int&&` 参数必须使用 `std::move(x)`。  
lamb2(x); // 合法；`x` 的类型被推导为 `int&`。
```

Lambda函数可以是可变参数的，并且可以完美转发它们的参数：

```
auto lam = [](auto&&... args){return f(std::forward<decltype(args)>(args)...);};
```

或者：

```
auto lam = [](auto&&... args){return f(decltype(args)(args)...);};
```

这仅对类型为 `auto&&` 的变量“正确”工作。

使用泛型 lambda 的一个重要原因是用于访问语法。

```
boost::variant<int, double> value;  
apply_visitor(value, [&](auto&& e){  
    std::cout << e;  
});
```

这里我们以多态的方式进行访问；但在其他上下文中，我们传递的类型名称并不重要：

```
mutex_wrapped<std::ostream&> os = std::cout;  
os.write([&](auto&& os){  
    os << "hello world";});
```

在这里重复写 `std::ostream&` 的类型是多余的；这就像每次使用变量时都必须提及它的类型一样。这里我们创建了一个访问器，但不是多态的；`auto` 的使用原因与在 `for(:)` 循环中使用 `auto` 相同。

```
lamb1(x); // Illegal; must use `std::move(x)` for `int&&` parameters.  
lamb2(x); // Legal; the type of `x` is deduced as `int&`.
```

Lambda functions can be variadic and perfectly forward their arguments:

```
auto lam = [](auto&&... args){return f(std::forward<decltype(args)>(args)...);};
```

or:

```
auto lam = [](auto&&... args){return f(decltype(args)(args)...);};
```

which only works "properly" with variables of type `auto&&`.

A strong reason to use generic lambdas is for visiting syntax.

```
boost::variant<int, double> value;  
apply_visitor(value, [&](auto&& e){  
    std::cout << e;  
});
```

Here we are visiting in a polymorphic manner; but in other contexts, the names of the type we are passing isn't interesting:

```
mutex_wrapped<std::ostream&> os = std::cout;  
os.write([&](auto&& os){  
    os << "hello world\n";  
});
```

Repeating the type of `std::ostream&` is noise here; it would be like having to mention the type of a variable every time you use it. Here we are creating a visitor, but no a polymorphic one; `auto` is used for the same reason you might use `auto` in a `for(:)` loop.

Section 73.9: Using lambdas for inline parameter pack unpacking

Version ≥ C++14

参数包展开传统上需要为每次想要展开时编写一个辅助函数。

在这个示例中：

```
template<std::size_t... Is>  
void print_indexes( std::index_sequence<Is...> ) {  
    using discard=int[];  
    (void)discard{0,((void)(  
        std::cout << Is << " " // 这里 Is 是编译时常量。  
    ),0)...};  
}  
template<std::size_t I>  
void print_indexes_upto() {  
    return print_indexes( std::make_index_sequence<I>{} );  
}
```

`print_indexes_upto` 想要创建并展开一个索引的参数包。为了做到这一点，它必须调用一个辅助函数。每次你想展开一个你创建的参数包时，最终都必须创建一个

```
template<std::size_t... Is>  
void print_indexes( std::index_sequence<Is...> ) {  
    using discard=int[];  
    (void)discard{0,((void)(  
        std::cout << Is << '\n' // 这里 Is 是一个编译时常量。  
    ),0)...};  
}  
template<std::size_t I>  
void print_indexes_upto() {  
    return print_indexes( std::make_index_sequence<I>{} );  
}
```

The `print_indexes_upto` 想要创建并展开一个参数包。为了做到这一点，它必须调用一个辅助函数。每次你想展开一个你创建的参数包时，最终都必须创建一个

自定义辅助函数来实现它。

这可以通过lambda避免。

你可以将参数包展开为一组lambda调用，像这样：

```
template<std::size_t I>
using index_t = std::integral_constant<std::size_t, I>;
template<std::size_t I>
constexpr index_t<I> index{};

template<class=void, std::size_t... Is>
auto index_over( std::index_sequence<Is...> ) {
    return [](auto&& f){
        using discard=int[];
        (void)discard{0,(void(
            f( index<Is> )
        ),0)...};
    };
}

template<std::size_t N>
auto index_over(index_t<N> = {}) {
    return index_over( std::make_index_sequence<N>{} );
}
```

版本 ≥ C++17

使用折叠表达式，index_over() 可以简化为：

```
template<class=void, std::size_t... Is>
auto index_over( std::index_sequence<Is...> ) {
    return [](auto&& f){
        ((void)(f(index<Is>)), ...);
    };
}
```

完成后，您可以使用此方法替代在其他代码中通过第二个重载手动展开参数包，从而实现“内联”展开参数包：

```
template<class Tup, class F>
void for_each_tuple_element(Tup&& tup, F&& f) {
    using T = std::remove_reference_t<Tup>;
    using std::tuple_size;
    auto from_zero_to_N = index_over< tuple_size<T>{} >();

    from_zero_to_N(
        [&](auto i){
            using std::get;
            f( get<i>( std::forward<Tup>(tup) ) );
        }
    );
}
```

由index_over传递给lambda的auto i是一个std::integral_constant<std::size_t, ???>。这具有一个constexpr转换为std::size_t的功能，该转换不依赖于this的状态，因此我们可以将其用作编译时常量，例如当我们将其传递给上面的std::get<i>时。

回到顶部的玩具示例，重写为：

custom helper function to do it.

This can be avoided with lambdas.

You can unpack parameter packs into a set of invocations of a lambda, like this:

```
template<std::size_t I>
using index_t = std::integral_constant<std::size_t, I>;
template<std::size_t I>
constexpr index_t<I> index{};

template<class=void, std::size_t... Is>
auto index_over( std::index_sequence<Is...> ) {
    return [](auto&& f){
        using discard=int[];
        (void)discard{0,(void(
            f( index<Is> )
        ),0)...};
    };
}

template<std::size_t N>
auto index_over(index_t<N> = {}) {
    return index_over( std::make_index_sequence<N>{} );
}
```

Version ≥ C++17

With fold expressions, index_over() can be simplified to:

```
template<class=void, std::size_t... Is>
auto index_over( std::index_sequence<Is...> ) {
    return [](auto&& f){
        ((void)(f(index<Is>)), ...);
    };
}
```

Once you have done that, you can use this to replace having to manually unpack parameter packs with a second overload in other code, letting you unpack parameter packs "inline":

```
template<class Tup, class F>
void for_each_tuple_element(Tup&& tup, F&& f) {
    using T = std::remove_reference_t<Tup>;
    using std::tuple_size;
    auto from_zero_to_N = index_over< tuple_size<T>{} >();

    from_zero_to_N(
        [&](auto i){
            using std::get;
            f( get<i>( std::forward<Tup>(tup) ) );
        }
    );
}
```

The auto i passed to the lambda by the index_over is a std::integral_constant<std::size_t, ???>. This has a constexpr conversion to std::size_t that does not depend on the state of this, so we can use it as a compile-time constant, such as when we pass it to std::get<i> above.

To go back to the toy example at the top, rewrite it as:

```
template<std::size_t I>
void print_indexes_upto() {
    index_over(index<I>) ([](auto i){ std::cout << i << "\n"; // 这里 i 是编译时常量);
}
```

这要短得多，并且将逻辑保留在使用它的代码中。

[可在线试玩的示例。](#)

第73.10节：广义捕获

版本 ≥ C++14

Lambda表达式可以捕获表达式，而不仅仅是变量。这允许Lambda存储仅能移动的类型：

```
auto p = std::make_unique<T>(...);

auto lamb = [p = std::move(p)]() //覆盖了对`p`的按值捕获。
{
    p->SomeFunc();
};
```

这将外部的p变量移动到lambda捕获变量中，也称为p。lamb现在拥有由make_unique分配的内存。由于闭包包含一个不可复制的类型，这意味着lamb本身也是不可复制的。但它可以被移动：

```
auto lamb_copy = lamb; //非法
auto lamb_move = std::move(lamb); //合法。
```

现在lamb_move拥有这块内存。

注意std::function要求存储的值必须是可复制的。你可以编写自己的只支持移动的std::function，或者你也可以将lambda放入一个shared_ptr包装器中：

```
auto shared_lambda = [](auto&& f){
    return [spf = std::make_shared<std::decay_t<decltype(f)>>(decltype(f)(f))]
        (auto&&...args)->decltype(auto) {
            return (*spf)(decltype(args)(args)...);
    };
};
auto lamb_shared = shared_lambda(std::move(lamb_move));
```

将我们的只支持移动的lambda及其状态放入一个共享指针中，然后返回一个可以被复制的lambda，这样就可以存储在std::function或类似的容器中。

泛化捕获使用auto类型推断变量的类型。默认情况下，它们会被声明为值捕获，但也可以是引用捕获：

```
int a = 0;

auto lamb = [&v = a](int add) //注意 `a` 和 `v` 名称不同
{
    v += add; //修改 `a`
};
```

```
template<std::size_t I>
void print_indexes_upto() {
    index_over(index<I>) ([](auto i){
        std::cout << i << '\n'; // here i is a compile-time constant
    });
}
```

which is much shorter, and keeps logic in the code that uses it.

[Live example](#) to play with.

Section 73.10: Generalized capture

Version ≥ C++14

Lambdas can capture expressions, rather than just variables. This permits lambdas to store move-only types:

```
auto p = std::make_unique<T>(...);

auto lamb = [p = std::move(p)]() //Overrides capture-by-value of `p`.
{
    p->SomeFunc();
};
```

This moves the outer p variable into the lambda capture variable, also called p. lamb now owns the memory allocated by make_unique. Because the closure contains a type that is non-copyable, this means that lamb is itself non-copyable. But it can be moved:

```
auto lamb_copy = lamb; //Illegal
auto lamb_move = std::move(lamb); //Legal.
```

Now lamb_move owns the memory.

Note that std::function requires that the values stored be copyable. You can write your own move-only-requring std::function, or you could just stuff the lambda into a shared_ptr wrapper:

```
auto shared_lambda = [](auto&& f){
    return [spf = std::make_shared<std::decay_t<decltype(f)>>(decltype(f)(f))]
        (auto&&...args)->decltype(auto) {
            return (*spf)(decltype(args)(args)...);
    };
};
auto lamb_shared = shared_lambda(std::move(lamb_move));
```

takes our move-only lambda and stuffs its state into a shared pointer then returns a lambda that *can* be copied, and then stored in a std::function or similar.

Generalized capture uses auto type deduction for the variable's type. It will declare these captures as values by default, but they can be references as well:

```
int a = 0;

auto lamb = [&v = a](int add) //Note that `a` and `v` have different names
{
    v += add; //Modifies `a`
};
```

```
lamb(20); //`a` 变为 20.
```

泛化捕获根本不需要捕获外部变量。它可以捕获任意表达式：

```
auto lamb = [p = std::make_unique<T>(...)]()  
{  
    p->SomeFunc();  
}
```

这对于给 lambda 赋予它们可以持有并可能修改的任意值非常有用，而无需在 lambda 外部声明它们。当然，只有在你不打算在 lambda 完成工作后访问这些变量时，这才有用。

第 73.11 节：转换为函数指针

如果 lambda 的捕获列表为空，则该 lambda 隐式转换为一个函数指针，该指针接受相同的参数并返回相同的返回类型：

```
auto sorter = [](int lhs, int rhs) -> bool {return lhs < rhs;};  
  
using func_ptr = bool(*)(int, int);  
func_ptr sorter_func = sorter; // 隐式转换
```

这种转换也可以通过一元加号运算符强制执行：

```
func_ptr sorter_func2 = +sorter; // 强制隐式转换
```

调用此函数指针的行为与调用lambda的operator()完全相同。此函数指针完全不依赖于源lambda闭包的存在。因此，它可能比lambda闭包存在时间更长。

此功能主要用于将lambda与处理函数指针的API一起使用，而不是C++函数对象。

版本 ≥ C++14

对于空捕获列表的泛型lambda，也可以转换为函数指针。如有必要，将使用模板参数推导来选择正确的特化。

```
auto sorter = [](auto lhs, auto rhs) { return lhs < rhs; };  
using func_ptr = bool(*)(int, int);  
func_ptr sorter_func = sorter; // 推导为 int, int  
// 但请注意，以下写法是歧义的  
// func_ptr sorter_func2 = +sorter;
```

第73.12节：使用仿函数将lambda函数移植到C++03

C++中的lambda函数是语法糖，提供了非常简洁的仿函数编写语法。因此，可以通过将lambda函数转换为仿函数，在C++03中获得等效功能（尽管冗长）：

```
// 一些示例类型：  
struct T1 {int dummy;};  
struct T2 {int dummy;};  
struct R {int dummy;};
```

```
lamb(20); //`a` becomes 20.
```

Generalize capture does not need to capture an external variable at all. It can capture an arbitrary expression:

```
auto lamb = [p = std::make_unique<T>(...)]()  
{  
    p->SomeFunc();  
}
```

This is useful for giving lambdas arbitrary values that they can hold and potentially modify, without having to declare them externally to the lambda. Of course, that is only useful if you do not intend to access those variables after the lambda has completed its work.

Section 73.11: Conversion to function pointer

If a lambda's capture list is empty, then the lambda has an implicit conversion to a function pointer that takes the same arguments and returns the same return type:

```
auto sorter = [](int lhs, int rhs) -> bool {return lhs < rhs;};  
  
using func_ptr = bool(*)(int, int);  
func_ptr sorter_func = sorter; // implicit conversion
```

Such a conversion may also be enforced using unary plus operator:

```
func_ptr sorter_func2 = +sorter; // enforce implicit conversion
```

Calling this function pointer behaves exactly like invoking operator() on the lambda. This function pointer is in no way reliant on the source lambda closure's existence. It therefore may outlive the lambda closure.

This feature is mainly useful for using lambdas with APIs that deal in function pointers, rather than C++ function objects.

Version ≥ C++14

Conversion to a function pointer is also possible for generic lambdas with an empty capture list. If necessary, template argument deduction will be used to select the correct specialization.

```
auto sorter = [](auto lhs, auto rhs) { return lhs < rhs; };  
using func_ptr = bool(*)(int, int);  
func_ptr sorter_func = sorter; // deduces int, int  
// note however that the following is ambiguous  
// func_ptr sorter_func2 = +sorter;
```

Section 73.12: Porting lambda functions to C++03 using functors

Lambda functions in C++ are syntactic sugar that provide a very concise syntax for writing functors. As such, equivalent functionality can be obtained in C++03 (albeit much more verbose) by converting the lambda function into a functor:

```
// Some dummy types:  
struct T1 {int dummy;};  
struct T2 {int dummy;};  
struct R {int dummy;};
```

```

// 使用lambda函数的代码 (需要C++11)
R use_lambda(T1 val, T2 ref) {
    // 使用auto, 因为lambda的类型未知。
    auto lambda = [val, &ref](int arg1, int arg2) -> R {
        /* lambda函数体 */
        return R();
    };
    return lambda(12, 27);
}

// 函数对象类 (适用于C++03)
// 类似于编译器为lambda函数生成的代码。
class Functor {
    // 捕获列表。
    T1 val;
    T2& ref;

public:
    // 构造函数
    inline Functor(T1 val, T2& ref) : val(val), ref(ref) {}

    // 函数体
    R operator()(int arg1, int arg2) const {
        /* lambda函数体 */
        return R();
    }
};

// 等同于 use_lambda, 但使用了函数对象 (有效的 C++03)。
R use_functor(T1 val, T2 ref) {
    Functor functor(val, ref);
    return functor(12, 27);
}

// 使其成为一个自包含的示例。
int main() {
    T1 t1;
    T2 t2;
    use_functor(t1,t2);
    use_lambda(t1,t2);
    return 0;
}

```

如果 lambda 函数是`mutable`, 则使仿函数的调用运算符为非常量, 即:

```

R operator()(int arg1, int arg2) /*非const*/ {
    /* lambda体 */
    return R();
}

```

```

// Code using a lambda function (requires C++11)
R use_lambda(T1 val, T2 ref) {
    // Use auto because the type of the lambda is unknown.
    auto lambda = [val, &ref](int arg1, int arg2) -> R {
        /* lambda-body */
        return R();
    };
    return lambda(12, 27);
}

// The functor class (valid C++03)
// Similar to what the compiler generates for the lambda function.
class Functor {
    // Capture list.
    T1 val;
    T2& ref;

public:
    // Constructor
    inline Functor(T1 val, T2& ref) : val(val), ref(ref) {}

    // Functor body
    R operator()(int arg1, int arg2) const {
        /* lambda-body */
        return R();
    }
};

// Equivalent to use_lambda, but uses a functor (valid C++03).
R use_functor(T1 val, T2 ref) {
    Functor functor(val, ref);
    return functor(12, 27);
}

// Make this a self-contained example.
int main() {
    T1 t1;
    T2 t2;
    use_functor(t1,t2);
    use_lambda(t1,t2);
    return 0;
}

```

If the lambda function is `mutable` then make the functor's call-operator non-const, i.e.:

```

R operator()(int arg1, int arg2) /*non-const*/ {
    /* lambda-body */
    return R();
}

```

第74章：值类别

第74.1节：值类别含义

C++中的表达式根据其结果被赋予特定的值类别。表达式的值类别会影响C++函数重载解析。

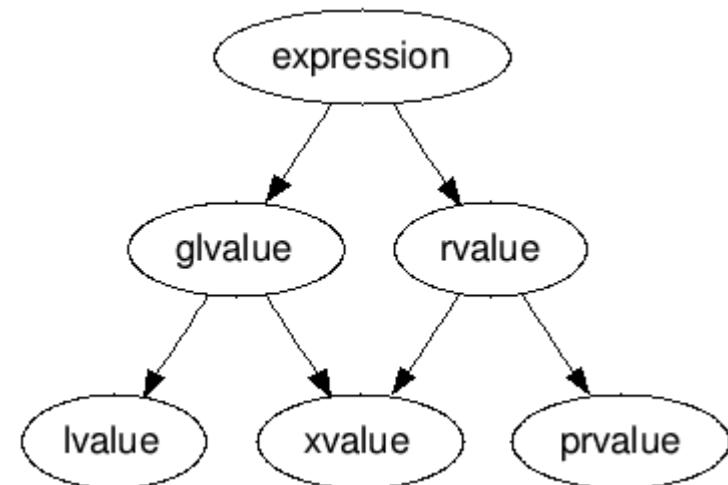
值类别决定表达式的两个重要但独立的属性。一个属性是表达式是否具有标识。表达式具有标识意味着它引用了一个具有变量名的对象。变量名可能未直接出现在表达式中，但该对象仍然可以有一个变量名。

另一个属性是是否允许从表达式的值隐式移动。更具体地说，是指表达式作为函数参数时，是否会绑定到右值参数类型。

C++定义了三种值类别，代表这些属性的有用组合：左值（具有标识但不可移动的表达式）、将亡值（具有标识且可移动的表达式）和纯右值（无标识且可移动的表达式）。C++中不存在既无标识又不可移动的表达式。

C++还定义了另外两种值类别，分别仅基于其中一个属性：通用左值（具有标识的表达式）和右值（可移动的表达式）。它们作为前述类别的有用组合。

该图作为示例：



第74.2节：右值 (rvalue)

右值表达式是指任何可以被隐式移动的表达式，无论其是否具有标识性。

更准确地说，右值表达式可以用作接受参数类型为`T&&`（其中`T`是`expr`的类型）的函数的实参。只有右值表达式才能作为此类函数参数的实参；如果使用了非右值表达式，则重载解析会选择任何不使用右值引用参数的函数。如果不存在这样的函数，则会产生错误。

右值表达式类别包括所有的`xvalue`和`prvalue`表达式，仅限于这些表达式。

标准库函数`std::move`用于显式地将非右值表达式转换为右值。更具体地说，它将表达式转换为`xvalue`，因为即使之前是无标识的`prvalue`表达式，通过将其作为参数传递给`std::move`，它获得了标识（函数的参数名）并变成了`xvalue`。

Chapter 74: Value Categories

Section 74.1: Value Category Meanings

Expressions in C++ are assigned a particular value category, based on the result of those expressions. Value categories for expressions can affect C++ function overload resolution.

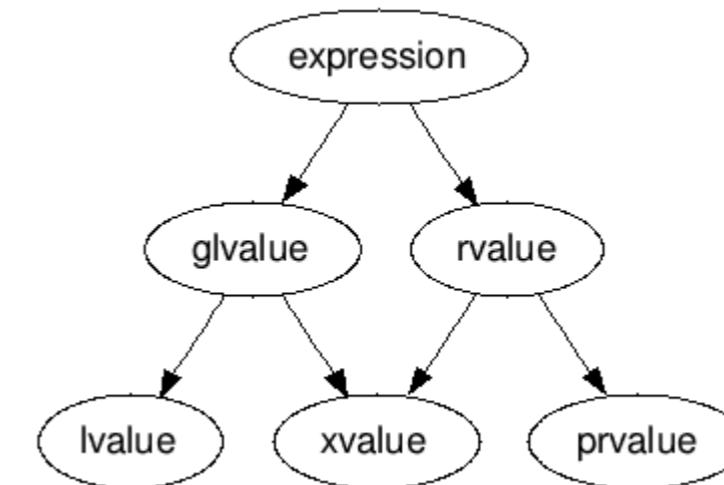
Value categories determines two important-but-separate properties about an expression. One property is whether the expression has identity. An expression has identity if it refers to an object that has a variable name. The variable name may not be involved in the expression, but the object can still have one.

The other property is whether it is legal to implicitly move from the expression's value. Or more specifically, whether the expression, when used as a function parameter, will bind to r-value parameter types or not.

C++ defines 3 value categories which represent the useful combination of these properties: lvalue (expressions with identity but not movable from), xvalue (expressions with identity that are moveable from), and prvalue (expressions without identity that are moveable from). C++ does not have expressions which have no identity and cannot be moved from.

C++ defines two other value categories, each based solely on one of these properties: glvalue (expressions with identity) and rvalue (expressions that can be moved from). These act as useful groupings of the prior categories.

This graph serves as an illustration:



Section 74.2: rvalue

An rvalue expression is any expression which can be implicitly moved from, regardless of whether it has identity.

More precisely, rvalue expressions may be used as the argument to a function that takes a parameter of type `T &&` (where `T` is the type of `expr`). Only rvalue expressions may be given as arguments to such function parameters; if a non-rvalue expression is used, then overload resolution will pick any function that does not use an rvalue reference parameter. And if none exist, then you get an error.

The category of rvalue expressions includes all xvalue and prvalue expressions, and only those expressions.

The standard library function `std::move` exists to explicitly transform a non-rvalue expression into an rvalue. More specifically, it turns the expression into an xvalue, since even if it was an identity-less prvalue expression before, by passing it as a parameter to `std::move`, it gains identity (the function's parameter name) and becomes an xvalue.

考虑以下情况：

```
std::string str("init");           //1
std::string test1(str);            //2
std::string test2(std::move(str));  //3

str = std::string("new value");    //4
std::string &str_ref = std::move(str); //5
std::string test3(str_ref);        //6
```

std::string有一个构造函数，接受单个参数类型为std::string&&，通常称为“移动构造函数”。然而，表达式str的值类别不是右值（具体来说是左值），因此不能调用该构造函数重载。相反，它调用了const std::string&重载，即拷贝构造函数。

第3行改变了情况。std::move的返回值是T&&，其中T是传入参数的基础类型。所以 std::move(str) 返回 std::string&&。函数调用的返回值是右值引用时，该调用是一个右值表达式（具体来说是一个xvalue），因此它可能调用 std::string 的移动构造函数。第3行之后，str 已被移动（其内容现在未定义）。

第4行将一个临时对象传递给std::string的赋值运算符。该运算符有一个重载，接受一个std::string&&。表达式 std::string("new value") 是一个右值表达式（具体来说是一个纯右值），因此可能调用该重载。因此，临时对象被移到str中，用特定内容替换了未定义的内容。

第5行创建了一个名为str_ref的右值引用，指向str。这就是值类别变得混淆的地方。

看，虽然str_ref是对std::string的右值引用，但表达式str_ref的值类别并不是一个右值。它是一个左值表达式。是的，真的。正因为如此，不能用表达式str_ref调用std::string的移动构造函数。因此第6行复制了str的值到test3中。

要移动它，我们必须再次使用std::move。

第74.3节：xvalue

xvalue（即“即将过期的值”）表达式是具有标识且表示可以被隐式移动的对象的表达式。xvalue表达式的一般概念是它们所代表的对象即将被销毁（因此称为“即将过期”），因此从它们隐式移动是可以接受的。

给定：

```
struct X { int n; };
extern X x;

4;          // 纯右值：没有标识
x;          // 左值
x.n;        // 左值
std::move(x); // x值
std::forward<X&>(x); // 左值
X{4};       // prvalue：没有身份
X{4}.n;     // xvalue：有身份且表示可重用资源
            // 这些资源可以被重复使用
```

第74.4节：prvalue

prvalue（纯右值）表达式是缺乏身份的表达式，其求值通常用于

Consider the following:

```
std::string str("init");           //1
std::string test1(str);            //2
std::string test2(std::move(str));  //3

str = std::string("new value");    //4
std::string &str_ref = std::move(str); //5
std::string test3(str_ref);        //6
```

std::string has a constructor which takes a single parameter of type std::string&&, commonly called a "move constructor". However, the value category of the expression str is not an rvalue (specifically it is an lvalue), so it cannot call that constructor overload. Instead, it calls the const std::string& overload, the copy constructor.

Line 3 changes things. The return value of std::move is a T&&, where T is the base type of the parameter passed in. So std::move(str) returns std::string&&. A function call who's return value is an rvalue expression (specifically an xvalue), so it may call the move constructor of std::string. After line 3, str has been moved from (who's contents are now undefined).

Line 4 passes a temporary to the assignment operator of std::string. This has an overload which takes a std::string&&. The expression std::string("new value") is an rvalue expression (specifically a prvalue), so it may call that overload. Thus, the temporary is moved into str, replacing the undefined contents with specific contents.

Line 5 creates a named rvalue reference called str_ref that refers to str. This is where value categories get confusing.

See, while str_ref is an rvalue reference to std::string, the value category of the expression str_ref is not an rvalue. It is an lvalue expression. Yes, really. Because of this, one cannot call the move constructor of std::string with the expression str_ref. Line 6 therefore copies the value of str into test3.

To move it, we would have to employ std::move again.

Section 74.3: xvalue

An xvalue (eXpiring value) expression is an expression which has identity and represents an object which can be implicitly moved from. The general idea with xvalue expressions is that the object they represent is going to be destroyed soon (hence the "eXpiring" part), and therefore implicitly moving from them is fine.

Given:

```
struct X { int n; };
extern X x;

4;          // prvalue: does not have an identity
x;          // lvalue
x.n;        // lvalue
std::move(x); // xvalue
std::forward<X&>(x); // lvalue
X{4};       // prvalue: does not have an identity
X{4}.n;     // xvalue: does have an identity and denotes resources
            // that can be reused
```

Section 74.4: prvalue

A prvalue (pure-rvalue) expression is an expression which lacks identity, whose evaluation is typically used to

初始化对象，并且可以被隐式移动。这些包括但不限于：

- 表示临时对象的表达式，例如 `std::string("123")`。
- 不返回引用的函数调用表达式
- 字面量（除字符串字面量外——字符串字面量是左值），如 `1`、`true`、`0.5f` 或 `'a'`
- `lambda` 表达式

内置的取地址操作符（`&`）不能应用于这些表达式。

第74.5节：左值

左值表达式是具有标识但不能被隐式移动的表达式。其中包括由变量名、函数名组成的表达式，内置解引用操作符使用的表达式，以及指向左值引用的表达式。

典型的左值只是一个名称，但左值也可以有其他形式：

```
struct X { ... };

X x;           // x 是一个左值
X* px = &x;   // px 是一个左值
*px = X{};    // *px 也是一个左值, X{} 是一个纯右值

X* foo_ptr(); // foo_ptr() 是一个纯右值
X& foo_ref(); // foo_ref() 是一个左值
```

此外，虽然大多数字面量（例如 `4`, `'x'` 等）是纯右值，但字符串字面量是左值。

第74.6节：glvalue

glvalue（“广义左值”）表达式是任何具有标识的表达式，无论它是否可以被移动。此类别包括左值（具有标识但不能被移动的表达式）和x值（具有标识且可以被移动的表达式），但不包括纯右值（无标识的表达式）。

如果一个表达式有一个名称，它就是一个glvalue：

```
struct X { int n; };
X foo();

X x;
x; // 有名字，所以它是一个glvalue
std::move(x); // 有名字（我们从"x"移动），所以它是一个glvalue
               // 可以被移动，所以它是一个xvalue而不是lvalue

foo(); // 没有名字，所以是prvalue，不是glvalue
X{};   // 临时对象没有名字，所以是prvalue，不是glvalue
X{}.n; // 有名字，所以是glvalue。可以被移动，所以是xvalue
```

initialize an object, and which can be implicitly moved from. These include, but are not limited to:

- Expressions that represent temporary objects, such as `std::string("123")`.
- A function call expression that does not return a reference
- A literal (except a string literal - those are lvalues), such has `1`, `true`, `0.5f`, or `'a'`
- A lambda expression

The built-in addressof operator (`&`) cannot be applied on these expressions.

Section 74.5: lvalue

An lvalue expression is an expression which has identity, but cannot be implicitly moved from. Among these are expressions that consist of a variable name, function name, expressions that are built-in dereference operator uses and expressions that refer to lvalue references.

The typical lvalue is simply a name, but lvalues can come in other flavors as well:

```
struct X { ... };

X x;           // x is an lvalue
X* px = &x;   // px is an lvalue
*px = X{};    // *px is also an lvalue, X{} is a prvalue

X* foo_ptr(); // foo_ptr() is a prvalue
X& foo_ref(); // foo_ref() is an lvalue
```

Additionally, while most literals (e.g. `4`, `'x'`, etc.) are prvalues, string literals are lvalues.

Section 74.6: glvalue

A glvalue (a "generalized lvalue") expression is any expression which has identity, regardless of whether it can be moved from or not. This category includes lvalues (expressions that have identity but can't be moved from) and xvalues (expressions that have identity, and can be moved from), but excludes prvalues (expressions without identity).

If an expression has a *name*, it's a glvalue:

```
struct X { int n; };
X foo();

X x;
x; // has a name, so it's a glvalue
std::move(x); // has a name (we're moving from "x"), so it's a glvalue
               // can be moved from, so it's an xvalue not an lvalue

foo(); // has no name, so is a prvalue, not a glvalue
X{};   // temporary has no name, so is a prvalue, not a glvalue
X{}.n; // HAS a name, so is a glvalue. can be moved from, so it's an xvalue
```

第75章：预处理器

C预处理器是一个简单的文本解析/替换器，在代码实际编译之前运行。用于扩展和简化C（以及后来的C++）语言的使用，它可用于：

- a. 使用#include包含其他文件
- b. 使用#define定义文本替换宏
- c. 使用#if #ifdef条件编译
- d. 平台/编译器特定逻辑（作为条件编译的扩展）

第75.1节：包含保护

一个头文件可能会被其他头文件包含。因此，一个包含多个头文件的源文件（编译单元）可能会间接地多次包含某些头文件。如果被多次包含的头文件中包含定义，编译器（预处理后）会检测到违反“一定义规则”（例如，2003年C++标准的第3.2节），从而发出诊断信息并导致编译失败。

通过使用“包含保护”（include guards）来防止多重包含，这些保护有时也称为头文件保护或宏保护。它们是通过预处理指令#define、#ifndef、#endif来实现的。

例如

```
// Foo.h
#ifndef FOO_H_INCLUDED
#define FOO_H_INCLUDED

class Foo // 一个类定义
{
};

#endif
```

使用包含保护的主要优点是它们能在所有符合标准的编译器和预处理器上正常工作。

然而，包含保护也给开发者带来一些问题，因为必须确保宏在项目中所有使用的头文件中是唯一的。具体来说，如果两个（或更多）头文件使用相同的FOO_H_INCLUDED作为包含保护，那么在一个编译单元中首先包含的那个头文件将有效地阻止其他头文件被包含。如果项目使用了多个第三方库且这些库的头文件恰好使用了相同的包含保护，则会带来特别的挑战。

还必须确保包含保护中使用的宏不会与头文件中定义的其他宏冲突。

大多数C++实现还支持#pragma once指令，该指令确保文件在单次编译中只被包含一次。这是一个事实上的标准指令，但并非任何ISO C++标准的一部分。例如：

```
// Foo.h
#pragma once

class Foo
```

Chapter 75: Preprocessor

The C preprocessor is a simple text parser/replacer that is run before the actual compilation of the code. Used to extend and ease the use of the C (and later C++) language, it can be used for:

- a. Including other files using #include
- b. Define a text-replacement macro using #define
- c. Conditional Compilation using #if #ifdef
- d. Platform/Compiler specific logic (as an extension of conditional compilation)

Section 75.1: Include Guards

A header file may be included by other header files. A source file (compilation unit) that includes multiple headers may therefore, indirectly, include some headers more than once. If such a header file that is included more than once contains definitions, the compiler (after preprocessing) detects a violation of the One Definition Rule (e.g. §3.2 of the 2003 C++ standard) and therefore issues a diagnostic and compilation fails.

Multiple inclusion is prevented using "include guards", which are sometimes also known as header guards or macro guards. These are implemented using the preprocessor #define, #ifndef, #endif directives.

e.g.

```
// Foo.h
#ifndef FOO_H_INCLUDED
#define FOO_H_INCLUDED

class Foo // a class definition
{
};

#endif
```

The key advantage of using include guards is that they will work with all standard-compliant compilers and preprocessors.

However, include guards also cause some problems for developers, as it is necessary to ensure the macros are unique within all headers used in a project. Specifically, if two (or more) headers use FOO_H_INCLUDED as their include guard, the first of those headers included in a compilation unit will effectively prevent the others from being included. Particular challenges are introduced if a project uses a number of third-party libraries with header files that happen to use include guards in common.

It is also necessary to ensure that the macros used in include guards do not conflict with any other macros defined in header files.

Most C++ implementations also support the #pragma once directive which ensures the file is only included once within a single compilation. This is a *de facto standard* directive, but it is not part of any ISO C++ standard. For example:

```
// Foo.h
#pragma once

class Foo
```

; 虽然

标准——本质上是特定于编译器的钩子，不支持该功能的编译器将会默默忽略它。
使用#pragma once的项目更难移植到不支持该指令的编译器。

许多针对C++的编码指南和保证标准明确反对除#include头文件或在头文件中放置包含保护之外的任何预处理器使用。

第75.2节：条件逻辑与跨平台处理

简而言之，条件预处理逻辑是通过宏定义使代码逻辑可用或不可用于编译。

三个主要的使用案例是：

- 不同的应用配置文件（例如调试、发布、测试、优化）可以是同一应用的候选项（例如带有额外的日志记录）。
- 跨平台编译 - 单一代码库，多平台编译。
- 利用通用代码库支持多个应用版本（例如软件的基础版、专业版和高级版） - 功能略有不同。

示例 a: 一个用于删除文件的跨平台方法（示意）：

```
#ifdef _WIN32
#include <windows.h> // 以及其他 Windows 系统文件
#endif
#include <cstdio>

bool remove_file(const std::string &path)
{
#ifdef _WIN32
    return DeleteFile(path.c_str());
#elif defined(_POSIX_VERSION) || defined(__unix__)
    return (0 == remove(path.c_str()));
#elif defined(__APPLE__)
    //TODO: 检查 NSAPI 是否有带权限对话框的更具体函数
    return (0 == remove(path.c_str()));
#else
#error "此平台不受支持"
#endif
}
```

宏如__WIN32、__APPLE__或__unix__通常由相应的实现预定义。

示例 b: 为调试版本启用额外日志记录：

```
void s_PrintAppStateOnUserPrompt()
{
std::cout << "-----开始转储-----" << AppState::Instance()->Settings().ToString() << "#if ( 1 == TESTING_MODE ) //隐私：我们只在
测试时需要用户详细信息<< ListToString(AppState::UndoStack()->GetActionNames())<< AppState
::Instance()->CrntDocument().Name()
<< AppState::Instance()->CrntDocument().SignatureSHA() << "#endif

```

{
};

While #pragma once avoids some problems associated with include guards, a #pragma - by definition in the standards - is inherently a compiler-specific hook, and will be silently ignored by compilers that don't support it. Projects which use #pragma once are more difficult to port to compilers that don't support it.

A number of coding guidelines and assurance standards for C++ specifically discourage any use of the preprocessor other than to #include header files or for the purposes of placing include guards in headers.

Section 75.2: Conditional logic and cross-platform handling

In a nutshell, conditional pre-processing logic is about making code-logic available or unavailable for compilation using macro definitions.

Three prominent use-cases are:

- different **app profiles** (e.g. debug, release, testing, optimised) that can be candidates of the same app (e.g. with extra logging).
- cross-platform compiles** - single code-base, multiple compilation platforms.
- utilising a common code-base for multiple **application versions** (e.g. Basic, Premium and Pro versions of a software) - with slightly different features.

Example a: A cross-platform approach for removing files (illustrative):

```
#ifdef _WIN32
#include <windows.h> // 以及其他 windows 系统文件
#endif
#include <cstdio>

bool remove_file(const std::string &path)
{
#ifdef _WIN32
    return DeleteFile(path.c_str());
#elif defined(_POSIX_VERSION) || defined(__unix__)
    return (0 == remove(path.c_str()));
#elif defined(__APPLE__)
    //TODO: check if NSAPI has a more specific function with permission dialog
    return (0 == remove(path.c_str()));
#else
#error "This platform is not supported"
#endif
}
```

Macros like __WIN32, __APPLE__ or __unix__ are normally predefined by corresponding implementations.

Example b: Enabling additional logging for a debug build:

```
void s_PrintAppStateOnUserPrompt()
{
    std::cout << "-----BEGIN-DUMP-----\n"
    << AppState::Instance()->Settings().ToString() << "\n"
#if ( 1 == TESTING_MODE ) //privacy: we want user details only when testing
    << ListToString(AppState::UndoStack()->GetActionNames())
    << AppState::Instance()->CrntDocument().Name()
    << AppState::Instance()->CrntDocument().SignatureSHA() << "\n"
#endif
}
```

```
<< "-----结束转储-----"\n}
```

示例 c: 在单独的产品版本中启用高级功能 (注：此为示例。通常更好的做法是允许功能解锁而无需重新安装应用程序)

```
void MainWindow::OnProcessButtonClick()
{
#ifndef _PREMIUM
    CreatePurchaseDialog("购买应用高级版", "此功能仅对我们的应用高级版用户开放。  
点击购买按钮，在我们的网站购买高级版");
    return;
#endif
    //...实际功能逻辑在此
}
```

一些常用技巧：

在调用时定义符号：

预处理器可以带有预定义符号（可选初始化）调用。例如此命令
(gcc -E 仅运行预处理器)

```
gcc -E -DOPTIMISE_FOR_OS_X -DTESTING_MODE=1 Sample.cpp
```

以与在 Sample.cpp 顶部添加 `#define OPTIMISE_FOR_OS_X` 和 `#define TESTING_MODE 1`
相同的方式处理 Sample.cpp。

确保宏已定义：

如果宏未定义且其值被比较或检查，预处理器几乎总是默默地将该值视为0。有几种方法可以处理这种情况。一种方法是假设默认设置表示为0，任何更改（例如应用程序构建配置）都需要显式完成（例如

默认情况下ENABLE_EXTRA_DEBUGGING=0，设置 -DENABLE_EXTRA_DEBUGGING=1 以覆盖）。另一种方法是使所有定义和默认值都显式化。这可以通过结合使用#ifndef和#error指令来实现：

```
#ifndef (ENABLE_EXTRA_DEBUGGING)
// 如果尚未包含，请包含 DefaultDefines.h。
#  error "ENABLE_EXTRA_DEBUGGING 未定义"
#else
#  if ( 1 == ENABLE_EXTRA_DEBUGGING )
    //代码
#  endif
#endif
```

第75.3节：X宏

一种在编译时生成重复代码结构的惯用技术。

X宏由两部分组成：列表和列表的执行。

示例：

```
#define LIST \
    X(dog) \
```

```
<< "-----END-DUMP-----"\n"
```

```
}
```

Example c: Enable a premium feature in a separate product build (note: this is illustrative. it is often a better idea to allow a feature to be unlocked without the need to reinstall an application)

```
void MainWindow::OnProcessButtonClick()
{
#ifndef _PREMIUM
    CreatePurchaseDialog("Buy App Premium", "This feature is available for our App Premium users.  
Click the Buy button to purchase the Premium version at our website");
    return;
#endif
    //...actual feature logic here
}
```

Some common tricks:

Defining symbols at invocation time:

The preprocessor can be called with predefined symbols (with optional initialisation). For example this command
(gcc -E 仅运行预处理器)

```
gcc -E -DOPTIMISE_FOR_OS_X -DTESTING_MODE=1 Sample.cpp
```

processes Sample.cpp in the same way as it would if `#define OPTIMISE_FOR_OS_X` and `#define TESTING_MODE 1`
were added to the top of Sample.cpp.

Ensuring a macro is defined:

If a macro isn't defined and its value is compared or checked, the preprocessor almost always silently assumes the
value to be 0. There are a few ways to work with this. One approach is to assume that the default settings are
represented as 0, and any changes (e.g. to the app build profile) needs to be explicitly done (e.g.
`ENABLE_EXTRA_DEBUGGING=0` by default, set `-DENABLE_EXTRA_DEBUGGING=1` to override). Another approach is
make all definitions and defaults explicit. This can be achieved using a combination of `#ifndef` and `#error`
directives:

```
#ifndef (ENABLE_EXTRA_DEBUGGING)
// please include DefaultDefines.h if not already included.
#  error "ENABLE_EXTRA_DEBUGGING is not defined"
#else
#  if ( 1 == ENABLE_EXTRA_DEBUGGING )
    //code
#  endif
#endif
```

Section 75.3: X-macros

An idiomatic technique for generating repeating code structures at compile time.

An X-macro consists of two parts: the list, and the execution of the list.

Example:

```
#define LIST \
    X(dog) \
```

```

X(cat)
X(raccoon)

// 类 Animal {
// 公共：
// void say();
// };

#define X(name) Animal name;
LIST
#undef X

int main() {
#define X(name) name.say();
    LIST
#undef X

    return 0;
}

```

这是预处理器展开后的内容：

```

Animal dog;
Animal cat;
Animal racoon;

int main() {
    dog.say();
    cat.say();
    racoon.say();

    return 0;
}

```

当列表变得更大（比如超过100个元素）时，这种技术有助于避免大量的复制粘贴。

来源：https://en.wikipedia.org/wiki/X_Macro

另见：X-macros

如果在使用LIST之前定义一个看似无关的X不符合你的喜好，你也可以将宏名称作为参数传递：

```

#define LIST(MACRO) \
    MACRO(dog) \
    MACRO(cat) \
    MACRO(raccoon)

```

现在，你可以明确指定在展开列表时应使用哪个宏，例如：

```

#define FORWARD_DECLARE_ANIMAL(name) Animal name;
LIST(FORWARD_DECLARE_ANIMAL)

```

如果每次调用MACRO都应接受额外的参数——相对于列表是常量，可以使用可变参数宏

```

// Visual Studio的一个解决方法
#define EXPAND(x) x

```

```

X(cat) \
X(raccoon)

// class Animal {
// public:
// void say();
// };

#define X(name) Animal name;
LIST
#undef X

int main() {
#define X(name) name.say();
    LIST
#undef X

    return 0;
}

```

which is expanded by the preprocessor into the following:

```

Animal dog;
Animal cat;
Animal racoon;

int main() {
    dog.say();
    cat.say();
    racoon.say();

    return 0;
}

```

As lists become bigger (let's say, more than 100 elements), this technique helps to avoid excessive copy-pasting.

Source: https://en.wikipedia.org/wiki/X_Macro

See also: X-macros

If defining a seemingly irrelevant X before using LIST is not to your liking, you can pass a macro name as an argument as well:

```

#define LIST(MACRO) \
    MACRO(dog) \
    MACRO(cat) \
    MACRO(raccoon)

```

Now, you explicitly specify which macro should be used when expanding the list, e.g.

```

#define FORWARD_DECLARE_ANIMAL(name) Animal name;
LIST(FORWARD_DECLARE_ANIMAL)

```

If each invocation of the MACRO should take additional parameters - constant with respect to the list, variadic macros can be used

```

// a workaround for Visual studio
#define EXPAND(x) x

```

```
#define LIST(MACRO, ...) \
    EXPAND(MACRO(dog, __VA_ARGS__)) \
    EXPAND(MACRO(cat, __VA_ARGS__)) \
    EXPAND(MACRO(raccoon, __VA_ARGS__))
```

第一个参数由LIST提供，其余参数由用户在调用LIST时提供。例如：

```
#define FORWARD_DECLARE(name, type, prefix) type prefix##name;
LIST(FORWARD_DECLARE,Animal,anim_)
LIST(FORWARD_DECLARE,Object,obj_)
```

将展开为

```
Animal anim_dog;
Animal anim_cat;
Animal anim_raccoon;
Object obj_dog;
Object obj_cat;
Object obj_raccoon;
```

第75.4节：宏

宏分为两大类：类似对象的宏和类似函数的宏。宏在编译过程的早期被视为一种标记替换。这意味着可以将大量（或重复）的代码段抽象为预处理器宏。

```
// 这是一个类似对象的宏
#define PI      3.14159265358979

// 这是一个类似函数的宏。
// 注意我们可以在其他宏定义（类似对象或类似函数）中使用之前
// 定义的宏 // 但要注意，如果你知道自己在做什么，这非常有用，// 但编译器不知
// 道该如何处理类型，因此推荐使用内联函数代替// （但例如对于最小值/最大值函数，这种宏还是很
// 有用的）
#define AREA(r)  (PI*(r)*(r))

// 它们可以这样使用：
double pi_macro = PI;
double area_macro = AREA(4.6);
```

Qt库利用这种技术通过让用户在继承QObject的自定义类头部声明Q_OBJECT宏来创建元对象系统。

宏名称通常用全大写字母书写，以便于与普通代码区分。这不是强制要求，但被许多程序员视为良好风格。

当遇到类似对象的宏时，它会被展开为简单的复制粘贴操作，宏名称被其定义替换。当遇到类似函数的宏时，其名称和参数都会被展开。

```
double pi_squared = PI * PI;
// 编译器看到的是：
double pi_squared = 3.14159265358979 * 3.14159265358979;

double area = AREA(5);
```

```
#define LIST(MACRO, ...) \
    EXPAND(MACRO(dog, __VA_ARGS__)) \
    EXPAND(MACRO(cat, __VA_ARGS__)) \
    EXPAND(MACRO(raccoon, __VA_ARGS__))
```

The first argument is supplied by the LIST, while the rest is provided by the user in the LIST invocation. For example:

```
#define FORWARD_DECLARE(name, type, prefix) type prefix##name;
LIST(FORWARD_DECLARE,Animal,anim_)
LIST(FORWARD_DECLARE,Object,obj_)
```

will expand to

```
Animal anim_dog;
Animal anim_cat;
Animal anim_raccoon;
Object obj_dog;
Object obj_cat;
Object obj_raccoon;
```

Section 75.4: Macros

Macros are categorized into two main groups: object-like macros and function-like macros. Macros are treated as a token substitution early in the compilation process. This means that large (or repeating) sections of code can be abstracted into a preprocessor macro.

```
// This is an object-like macro
#define PI      3.14159265358979

// This is a function-like macro.
// Note that we can use previously defined macros
// in other macro definitions (object-like or function-like)
// But watch out, its quite useful if you know what you're doing, but the
// Compiler doesn't know which type to handle, so using inline functions instead
// is quite recommended (But e.g. for Minimum/Maximum functions it is quite useful)
#define AREA(r)  (PI*(r)*(r))

// They can be used like this:
double pi_macro = PI;
double area_macro = AREA(4.6);
```

The Qt library makes use of this technique to create a meta-object system by having the user declare the Q_OBJECT macro at the head of the user-defined class extending QObject.

Macro names are usually written in all caps, to make them easier to differentiate from normal code. This isn't a requirement, but is merely considered good style by many programmers.

When an object-like macro is encountered, it's expanded as a simple copy-paste operation, with the macro's name being replaced with its definition. When a function-like macro is encountered, both its name and its parameters are expanded.

```
double pi_squared = PI * PI;
// Compiler sees:
double pi_squared = 3.14159265358979 * 3.14159265358979;

double area = AREA(5);
```

```
// 编译器看到的是：  
double area = (3.14159265358979*(5)*(5))
```

因此，类似函数的宏参数通常会用括号括起来，如上面的AREA()。这是为了防止宏展开过程中可能出现的错误，特别是由单个宏参数由多个实际值组成时引起的错误。

```
#define BAD_AREA(r) PI * r * r  
  
double bad_area = BAD_AREA(5 + 1.6);  
// 编译器看到的是：  
double bad_area = 3.14159265358979 * 5 + 1.6 * 5 + 1.6;  
  
double good_area = AREA(5 + 1.6);  
// 编译器看到的是：  
double good_area = (3.14159265358979*(5 + 1.6)*(5 + 1.6));
```

还要注意，由于这种简单的展开，传递给宏的参数必须小心，以防止出现意外的副作用。如果参数在求值过程中被修改，那么在宏展开后每次使用该参数时都会被修改，这通常不是我们想要的。即使宏将参数用括号括起来以防止展开破坏任何东西，这种情况仍然成立。

```
int oops = 5;  
double incremental_damage = AREA(oops++);  
// 编译器看到的是：  
double incremental_damage = (3.14159265358979*(oops++)*(oops++));
```

此外，宏不提供类型安全，导致难以理解的类型不匹配错误。

由于程序员通常用分号结束语句，设计为独立行使用的宏通常会设计成“吞掉”分号；这可以防止因多余的分号引起的意外错误。

```
#define IF_BREAKER(Func) Func();  
  
if (some_condition)  
    // 啊呀。  
IF_BREAKER(some_func);  
else  
std::cout << "我不小心成了孤儿。" << std::endl;
```

在此示例中，无意中出现的双分号破坏了if...else块，导致编译器无法将else与if匹配。为防止这种情况，宏定义中省略了分号，这将使宏“吞掉”紧跟其后的分号。

```
#define IF_FIXER(Func) Func()  
  
if (某个条件)  
IF_FIXER(某个函数);  
else  
std::cout << "太好了！我又能工作了！" << std::endl;
```

省略尾部分号还允许宏在不结束当前语句的情况下使用，这可能是有益的。

```
#define DO_SOMETHING(Func, Param) Func(Param, 2)
```

```
// Compiler sees:  
double area = (3.14159265358979*(5)*(5))
```

Due to this, function-like macro parameters are often enclosed within parentheses, as in AREA() above. This is to prevent any bugs that can occur during macro expansion, specifically bugs caused by a single macro parameter being composed of multiple actual values.

```
#define BAD_AREA(r) PI * r * r  
  
double bad_area = BAD_AREA(5 + 1.6);  
// Compiler sees:  
double bad_area = 3.14159265358979 * 5 + 1.6 * 5 + 1.6;  
  
double good_area = AREA(5 + 1.6);  
// Compiler sees:  
double good_area = (3.14159265358979*(5 + 1.6)*(5 + 1.6));
```

Also note that due to this simple expansion, care must be taken with the parameters passed to macros, to prevent unexpected side effects. If the parameter is modified during evaluation, it will be modified each time it is used in the expanded macro, which usually isn't what we want. This is true even if the macro encloses the parameters in parentheses to prevent expansion from breaking anything.

```
int oops = 5;  
double incremental_damage = AREA(oops++);  
// Compiler sees:  
double incremental_damage = (3.14159265358979*(oops++)*(oops++));
```

Additionally, macros provide no type-safety, leading to hard-to-understand errors about type mismatch.

As programmers normally terminate lines with a semicolon, macros that are intended to be used as standalone lines are often designed to "swallow" a semicolon; this prevents any unintended bugs from being caused by an extra semicolon.

```
#define IF_BREAKER(Func) Func();  
  
if (some_condition)  
    // Oops.  
IF_BREAKER(some_func);  
else  
    std::cout << "I am accidentally an orphan." << std::endl;
```

In this example, the inadvertent double semicolon breaks the `if...else` block, preventing the compiler from matching the `else` to the `if`. To prevent this, the semicolon is omitted from the macro definition, which will cause it to "swallow" the semicolon immediately following any usage of it.

```
#define IF_FIXER(Func) Func()  
  
if (some_condition)  
    IF_FIXER(some_func);  
else  
    std::cout << "Hooray! I work again!" << std::endl;
```

Leaving off the trailing semicolon also allows the macro to be used without ending the current statement, which can be beneficial.

```
#define DO_SOMETHING(Func, Param) Func(Param, 2)
```

```
// ...
```

```
some_function(DO_SOMETHING(某个函数, 3), DO_SOMETHING(某个函数, 42));
```

通常，宏定义在行尾结束。然而，如果宏需要跨多行，可以在行尾使用反斜杠来表示。该反斜杠必须是该行的最后一个字符，表示预处理器应将下一行与当前行连接，视为一行。这种用法可以连续多次。

```
#define TEXT "I \
am \
许多行
。

std::cout << TEXT << std::endl; // 输出：我是多行。
```

这在复杂的类似函数的宏中尤其有用，这些宏可能需要跨多行。

```
#define CREATE_OUTPUT_AND_DELETE(Str) \
    std::string* tmp = new std::string(Str); \
    std::cout << *tmp << std::endl; \
    delete tmp;

// ...

CREATE_OUTPUT_AND_DELETE("这实际上不需要使用 'new'。")
```

对于更复杂的类似函数的宏，给它们自己的作用域是有用的，以防止可能的名称冲突，或者使对象在宏结束时被销毁，类似于实际的函数。

一个常见的惯用法是 `do while 0`，其中宏被包裹在一个 `do-while` 块中。这个块通常 不以分号结尾，允许它吞掉一个分号。

```
#define DO_STUFF(Type, Param, ReturnVar) do { \
    Type temp(some_setup_values); \
    ReturnVar = temp.process(Param); \
} while (0)

int x;
DO_STUFF(MyClass, 41153.7, x);

// 编译器看到的是：

int x;
do {
    MyClass temp(some_setup_values);
    x = temp.process(41153.7);
} while (0);
```

还有可变参数宏；类似于可变参数函数，这些宏接受可变数量的参数，然后将它们全部展开替换特殊的“Varargs”参数，`__VA_ARGS__`。

```
#define VARIADIC(Param, ...) Param(__VA_ARGS__)

VARIADIC(sprintf, "%d", 8);
// 编译器看到的是：
```

```
// ...
```

```
some_function(DO_SOMETHING(some_func, 3), DO_SOMETHING(some_func, 42));
```

Normally, a macro definition ends at the end of the line. If a macro needs to cover multiple lines, however, a backslash can be used at the end of a line to indicate this. This backslash must be the last character in the line, which indicates to the preprocessor that the following line should be concatenated onto the current line, treating them as a single line. This can be used multiple times in a row.

```
#define TEXT "I \
am \
many \
lines.

// ...

std::cout << TEXT << std::endl; // Output: I am many lines.
```

This is especially useful in complex function-like macros, which may need to cover multiple lines.

```
#define CREATE_OUTPUT_AND_DELETE(Str) \
    std::string* tmp = new std::string(Str); \
    std::cout << *tmp << std::endl; \
    delete tmp;

// ...

CREATE_OUTPUT_AND_DELETE("There's no real need for this to use 'new'.")
```

In the case of more complex function-like macros, it can be useful to give them their own scope to prevent possible name collisions or to cause objects to be destroyed at the end of the macro, similar to an actual function. A common idiom for this is `do while 0`, where the macro is enclosed in a `do-while` block. This block is generally *not* followed with a semicolon, allowing it to swallow a semicolon.

```
#define DO_STUFF(Type, Param, ReturnVar) do { \
    Type temp(some_setup_values); \
    ReturnVar = temp.process(Param); \
} while (0)

int x;
DO_STUFF(MyClass, 41153.7, x);

// Compiler sees:

int x;
do {
    MyClass temp(some_setup_values);
    x = temp.process(41153.7);
} while (0);
```

There are also variadic macros; similarly to variadic functions, these take a variable number of arguments, and then expand them all in place of a special “Varargs” parameter, `__VA_ARGS__`.

```
#define VARIADIC(Param, ...) Param(__VA_ARGS__)

VARIADIC(sprintf, "%d", 8);
// Compiler sees:
```

```
printf("%d", 8);
```

注意，在展开过程中，`__VA_ARGS__`可以放在定义中的任何位置，并且会被正确展开。

```
#define VARIADIC2(POne, PTwo, PThree, ...) POne(PThree, __VA_ARGS__, PTwo)
VARIADIC2(some_func, 3, 8, 6, 9);
// 编译器看到的是：
some_func(8, 6, 9, 3);
```

在零参数可变参数的情况下，不同的编译器会对尾随逗号的处理方式不同。

一些编译器，如 Visual Studio，会在没有任何特殊语法的情况下默默忽略逗号。其他编译器，如 GCC，则要求你在`__VA_ARGS__`之前立即放置##。鉴于此，当关注可移植性时，明智的做法是有条件地定义可变参数宏。

```
// 在此示例中，COMPILER 是一个用户定义的宏，用于指定所使用的编译器。
```

```
#if      COMPILER == "VS"
#define VARIADIC3(Name, Param, ...) Name(Param, __VA_ARGS__)
#elif    COMPILER == "GCC"
#define VARIADIC3(Name, Param, ...) Name(Param, ##__VA_ARGS__)
#endif /* COMPILER */
```

第75.5节：预定义宏

预定义宏是由编译器定义的宏（与源文件中用户定义的宏相对）。这些宏不得被用户重新定义或取消定义。

以下宏是由 C++ 标准预定义的：

- `__LINE__` 包含该宏使用所在行的行号，并且可以通过`#line`指令更改。
- `__FILE__` 包含该宏使用所在文件的文件名，并且可以通过`#line`指令更改。
- `__DATE__` 包含文件编译的日期（格式为“Mmm dd yyyy”），其中Mmm的格式类似于调用`std::asctime()`所得。
- `__TIME__` 包含文件编译的时间（格式为“hh:mm:ss”）。
- `__cplusplus` 由符合标准的C++编译器在编译C++文件时定义。其值表示编译器完全符合的标准版本，即C++98 和C++03为199711L，C++11为201103L，C++14为201402L标准。

版本 ≥ c++11

- `__STDC_HOSTED__` 如果实现为hosted，则定义为1；如果为freestanding，则定义为0。

版本 ≥ c++17

- `__STDCPP_DEFAULT_NEW_ALIGNMENT__` 包含一个`size_t`字面量，表示调用不考虑对齐的`operator new`时使用的对齐方式。

此外，以下宏允许由实现预定义，可能存在也可能不存在：

- `__STDC__` 具有实现相关的含义，通常仅在以C语言编译文件时定义，用以表示完全符合C标准。（或者如果编译器决定不支持此宏，则永远不定义。）

版本 ≥ c++11

```
printf("%d", 8);
```

Note that during expansion, `__VA_ARGS__` can be placed anywhere in the definition, and will be expanded correctly.

```
#define VARIADIC2(POne, PTwo, PThree, ...) POne(PThree, __VA_ARGS__, PTwo)
VARIADIC2(some_func, 3, 8, 6, 9);
// Compiler sees:
some_func(8, 6, 9, 3);
```

In the case of a zero-argument variadic parameter, different compilers will handle the trailing comma differently. Some compilers, such as Visual Studio, will silently swallow the comma without any special syntax. Other compilers, such as GCC, require you to place ## immediately before `__VA_ARGS__`. Due to this, it is wise to conditionally define variadic macros when portability is a concern.

```
// In this example, COMPILER is a user-defined macro specifying the compiler being used.

#if      COMPILER == "VS"
#define VARIADIC3(Name, Param, ...) Name(Param, __VA_ARGS__)
#elif    COMPILER == "GCC"
#define VARIADIC3(Name, Param, ...) Name(Param, ##__VA_ARGS__)
#endif /* COMPILER */
```

Section 75.5: Predefined macros

Predefined macros are those that the compiler defines (in contrast to those user defines in the source file). Those macros must not be re-defined or undefined by user.

The following macros are predefined by the C++ standard:

- `__LINE__` contains the line number of the line this macro is used on, and can be changed by the `#line` directive.
- `__FILE__` contains the filename of the file this macro is used in, and can be changed by the `#line` directive.
- `__DATE__` contains date (in “Mmm dd yyyy” format) of the file compilation, where Mmm is formatted as if obtained by a call to `std::asctime()`.
- `__TIME__` contains time (in “hh:mm:ss” format) of the file compilation.
- `__cplusplus` is defined by (conformant) C++ compilers while compiling C++ files. Its value is the standard version the compiler is **fully** conformant with, i.e. 199711L for C++98 and C++03, 201103L for C++11 and 201402L for C++14 standard.

Version ≥ c++11

- `__STDC_HOSTED__` is defined to 1 if the implementation is *hosted*, or 0 if it is *freestanding*.

Version ≥ c++17

- `__STDCPP_DEFAULT_NEW_ALIGNMENT__` contains a `size_t` literal, which is the alignment used for a call to alignment-unaware operator `new`.

Additionally, the following macros are allowed to be predefined by implementations, and may or may not be present:

- `__STDC__` has implementation-dependent meaning, and is usually defined only when compiling a file as C, to signify full C standard compliance. (Or never, if the compiler decides not to support this macro.)

Version ≥ c++11

- `_STDC_VERSION_` 具有实现相关的含义，其值通常表示C语言版本，类似于`_cplusplus`表示C++版本。（或者如果编译器决定不支持此宏，则根本不定义。）
- `_STDC_MB_MIGHT_NEQ_WC_` 如果基本字符集的窄编码值可能不等于其宽编码对应值（例如`(uintmax_t)'x' != (uintmax_t)L'x'`），则定义为1。
- `_STDC_ISO_10646_` 如果 `wchar_t` 被编码为Unicode，则定义该宏，并扩展为一个整数常量，形式为`yyyymmL`，表示支持的最新Unicode版本。
- `_STDCPP_STRICT_POINTER_SAFETY_` 被定义为 1，如果实现具有 **严格指针安全**（否则为 **宽松指针安全**）
- `_STDCPP_THREADS_` 被定义为 1，如果程序可以有多个执行线程（适用于独立实现——托管实现总是可以有多个线程）

还值得一提的是 `_func_`，它不是宏，而是一个预定义的函数局部变量。它包含使用它的函数名，作为一个以实现定义格式存储的静态字符数组。

除了这些标准预定义宏外，编译器还可以有自己的一套预定义宏。必须参考编译器文档来了解这些宏。例如：

- [gcc](#)
- [Microsoft Visual C++](#)
- [clang](#)
- [Intel C++ 编译器](#)

有些宏只是用来查询某些特性的支持情况：

```
#ifdef __cplusplus // 如果由 C++ 编译器编译
extern "C" { // C 代码必须加修饰
    // C 库头文件声明在此
}
#endif
```

其他的对于调试非常有用：

```
版本 ≥ c++11
bool success = doSomething( /*一些参数*/ );
if( !success ){
    std::cerr << "错误: doSomething() 在第 " << __LINE__ - 2 << " 行失败" << "函数 " << __func__ << "() 中"
        << " 文件 " << __FILE__
        << std::endl;
}
```

还有一些用于简单的版本控制：

```
int main( int argc, char *argv[] ){
    if( argc == 2 && std::string( argv[1] ) == "-v" ){
        std::cout
        << "Hello World 程序"
            << "v 1.1" // 我必须手动记得更新这个<< "编译时间: " << __DATE__ << ' '
            << __TIME__ // 这个会自动更新<< std::endl;
    }
    else{
        std::cout << "Hello World!" ;
    }
}
```

- `_STDC_VERSION_` has implementation-dependent meaning, and its value is usually the C version, similarly to how `_cplusplus` is the C++ version. (Or is not even defined, if the compiler decides not to support this macro.)
- `_STDC_MB_MIGHT_NEQ_WC_` is defined to 1, if values of the narrow encoding of the basic character set might not be equal to the values of their wide counterparts (e.g. if `(uintmax_t)'x' != (uintmax_t)L'x'`)
- `_STDC_ISO_10646_` is defined if `wchar_t` is encoded as Unicode, and expands to an integer constant in the form `yyyymmL`, indicating the latest Unicode revision supported.
- `_STDCPP_STRICT_POINTER_SAFETY_` is defined to 1, if the implementation has *strict pointer safety* (otherwise it has *relaxed pointer safety*)
- `_STDCPP_THREADS_` is defined to 1, if the program can have more than one thread of execution (applicable to *freestanding implementation* — *hosted implementations* can always have more than one thread)

It is also worth mentioning `_func_`, which is not an macro, but a predefined function-local variable. It contains the name of the function it is used in, as a static character array in an implementation-defined format.

On top of those standard predefined macros, compilers can have their own set of predefined macros. One must refer to the compiler documentation to learn those. E.g.:

- [gcc](#)
- [Microsoft Visual C++](#)
- [clang](#)
- [Intel C++ Compiler](#)

Some of the macros are just to query support of some feature:

```
#ifdef __cplusplus // if compiled by C++ compiler
extern "C" { // C code has to be decorated
    // C library header declarations here
}
#endif
```

Others are very useful for debugging:

```
Version ≥ c++11
bool success = doSomething( /*some arguments*/ );
if( !success ){
    std::cerr << "ERROR: doSomething() failed on line " << __LINE__ - 2
        << " in function " << __func__ << "()"
        << " in file " << __FILE__
        << std::endl;
}
```

And others for trivial version control:

```
int main( int argc, char *argv[] ){
    if( argc == 2 && std::string( argv[1] ) == "-v" ){
        std::cout << "Hello World program\n"
            << "v 1.1\n" // I have to remember to update this manually
            << "compiled: " << __DATE__ << ' ' << __TIME__ // this updates automagically
            << std::endl;
    }
    else{
        std::cout << "Hello World!\n";
    }
}
```

第75.6节：预处理器运算符

#运算符或字符串化运算符用于将宏参数转换为字符串字面量。它只能用于带有参数的宏。

```
// 预处理器将参数x转换为字符串字面量#define PRINT(x) printf(#x "")
```

```
PRINT(这行将被预处理器转换为字符串);
```

```
// 编译器看到
```

```
printf("这行将被预处理器转换为字符串""");
```

编译器会连接两个字符串，最终的printf()参数将是一个以换行符结尾的字符串字面量。

预处理器会忽略宏参数前后的空格。因此下面的打印语句将给出相同的结果。

```
PRINT( 这行将被预处理器转换为字符串 );
```

如果字符串字面量的参数需要转义序列，比如双引号前的转义，预处理器会自动插入。

```
PRINT(这"行"将被预处理器转换为"字符串");
```

```
// 编译器看到
```

```
printf("这 \"行\" 将被预处理器转换为 \"字符串\"");
```

运算符或标记粘贴运算符用于连接宏的两个参数或标记。

```
// 预处理器将变量和 x 组合#define PRINT(x) printf("variable" #x " = %d", variable##x)
```

```
int variableY = 15;
```

```
PRINT(Y);
```

```
// 编译器看到
```

```
printf("variable""Y"" = %d", variableY);
```

最终输出将是

```
variableY = 15
```

第75.7节：#pragma once

大多数（但不是全部）C++ 实现支持 #pragma once 指令，该指令确保文件在单次编译中只被包含一次。它不是任何 ISO C++ 标准的一部分。例如：

```
// Foo.h
#pragma once
```

```
class Foo
{
```

虽然#pragma once避免了一些与包含保护相关的问题，但根据标准定义，#pragma本质上是一个特定于编译器的钩子，不支持它的编译器会默默忽略该指令。

Section 75.6: Preprocessor Operators

operator or stringizing operator is used to convert a Macro parameter to a string literal. It can only be used with the Macros having arguments.

```
// preprocessor will convert the parameter x to the string literal x
#define PRINT(x) printf(#x "\n")
```

```
PRINT(This line will be converted to string by preprocessor);
// Compiler sees
printf("This line will be converted to string by preprocessor"\n");
```

Compiler concatenate two strings and the final printf() argument will be a string literal with newline character at its end.

Preprocessor will ignore the spaces before or after the macro argument. So below print statement will give us the same result.

```
PRINT( This line will be converted to string by preprocessor );
```

If the parameter of the string literal requires an escape sequence like before a double quote() it will automatically be inserted by the preprocessor.

```
PRINT(This "line" will be converted to "string" by preprocessor);
// Compiler sees
printf("This \"line\" will be converted to \"string\" by preprocessor"\n");
```

operator or Token pasting operator is used to concatenate two parameters or tokens of a Macro.

```
// preprocessor will combine the variable and the x
#define PRINT(x) printf("variable" #x " = %d", variable##x)
```

```
int variableY = 15;
```

```
PRINT(Y);
```

```
// compiler sees
```

```
printf("variable""Y"" = %d", variableY);
```

and the final output will be

```
variableY = 15
```

Section 75.7: #pragma once

Most, but not all, C++ implementations support the #pragma once directive which ensures the file is only included once within a single compilation. It is not part of any ISO C++ standard. For example:

```
// Foo.h
#pragma once
```

```
class Foo
{
```

While #pragma once avoids some problems associated with include guards, a #pragma - by definition in the standards - is inherently a compiler-specific hook, and will be silently ignored by compilers that don't support it.

使用`#pragma once`的项目必须进行修改以符合标准。

对于某些编译器——尤其是那些使用预编译头文件的编译器——`#pragma once`可以显著加快编译过程。同样，一些预处理器通过跟踪哪些头文件使用了包含保护来加快编译速度。当同时使用`#pragma once`和包含保护时，整体效果取决于具体实现，可能会导致编译时间的增加或减少。

在Windows上编写基于MFC的应用程序时，`#pragma once`结合包含保护是头文件的推荐布局，这种布局由Visual Studio的add class、add dialog、add windows向导生成。

因此，在C++ Windows应用程序中常见两者结合使用。

第75.8节：预处理器错误信息

可以使用预处理器生成编译错误。这在多种情况下很有用，其中包括通知用户他们使用的是不支持的平台或编译器。

例如，如果gcc版本是3.0.0或更早版本，则返回错误。

```
#if __GNUC__ < 3
#error "此代码需要 gcc 版本大于 3.0.0"
#endif
```

例如：如果在苹果电脑上编译，则返回错误。

```
#ifdef __APPLE__
#error "本版本不支持苹果产品"
#endif
```

Projects which use `#pragma once` must be modified to be standard-compliant.

With some compilers - particularly those that employ [precompiled headers](#) - `#pragma once` can result in a considerable speedup of the compilation process. Similarly, some preprocessors achieve speedup of compilation by tracking which headers have employed include guards. The net benefit, when both `#pragma once` and include guards are employed, depends on the implementation and can be either an increase or decrease of compilation times.

`#pragma once` combined with include guards was the recommended layout for header files when writing MFC based applications on windows, and was generated by Visual Studio's add [class](#), add dialog, add windows wizards. Hence it is very common to find them combined in C++ Windows Applets.

Section 75.8: Preprocessor error messages

Compile errors can be generated using the preprocessor. This is useful for a number of reasons some of which include, notifying a user if they are on an unsupported platform or an unsupported compiler.

e.g. Return Error if gcc version is 3.0.0 or earlier.

```
#if __GNUC__ < 3
#error "This code requires gcc > 3.0.0"
#endif
```

e.g. Return Error if compiling on an Apple computer.

```
#ifdef __APPLE__
#error "Apple products are not supported in this release"
#endif
```

第76章：C++中的数据结构

第76.1节：C++中链表的实现

创建链表节点

```
class listNode
{
    公共:
        int data;
    listNode *next;
    listNode(int val):data(val),next(NULL){}
};
```

创建链表类

```
class List
{
    公共:
    listNode *head;
    List():head(NULL){}
    void insertAtBegin(int val);
    void insertAtEnd(int val);
    void insertAtPos(int val);
    void remove(int val);
    void print();
~List();
};
```

在链表开头插入新节点

```
void List::insertAtBegin(int val)//在链表前端插入
{
listNode *newnode = new listNode(val);
    newnode->next=this->head;
this->head=newnode;
}
```

在链表末尾插入新节点

```
void List::insertAtEnd(int val) //在链表末尾插入
{
    if(head==NULL)
    {
insertAtBegin(val);
    return;
    }
listNode *newnode = new listNode(val);
    listNode *ptr=this->head;
    while(ptr->next!=NULL)
    {
    ptr=ptr->next;
    }
    ptr->next=newnode;
}
```

在链表的特定位置插入

Chapter 76: Data Structures in C++

Section 76.1: Linked List implementation in C++

Creating a List Node

```
class listNode
{
    public:
        int data;
    listNode *next;
    listNode(int val):data(val),next(NULL){}
};
```

Creating List class

```
class List
{
    public:
    listNode *head;
    List():head(NULL){}
    void insertAtBegin(int val);
    void insertAtEnd(int val);
    void insertAtPos(int val);
    void remove(int val);
    void print();
~List();
};
```

Insert a new node at the beginning of the list

```
void List::insertAtBegin(int val)//inserting at front of list
{
    listNode *newnode = new listNode(val);
    newnode->next=this->head;
    this->head=newnode;
}
```

Insert a new node at the end of the list

```
void List::insertAtEnd(int val) //inserting at end of list
{
    if(head==NULL)
    {
        insertAtBegin(val);
        return;
    }
    listNode *newnode = new listNode(val);
    listNode *ptr=this->head;
    while(ptr->next!=NULL)
    {
        ptr=ptr->next;
    }
    ptr->next=newnode;
}
```

Insert at a particular position in list

```

void List::insertAtPos(int pos,int val)
{
listNode *newnode=new listNode(val);
if(pos==1)
{
    //作为头节点
newnode->next=this->head;
this->head=newnode;
return;
}
pos--;
listNode *ptr=this->head;
while(ptr!=NULL && --pos)
{
ptr=ptr->next;
}
if(ptr==NULL)
return;//元素不足
newnode->next=ptr->next;
ptr->next=newnode;
}

```

从链表中移除节点

```

void List::remove(int toBeRemoved)//移除元素
{
if(this->head==NULL)
return; //空链表
if(this->head->data==toBeRemoved)
{
    //第一个要删除的节点
listNode *temp=this->head;
this->head=this->head->next;
delete(temp);
return;
}
listNode *ptr=this->head;
while(ptr->next!=NULL && ptr->next->data!=toBeRemoved)
ptr=ptr->next;
if(ptr->next==NULL)
return;//未找到
listNode *temp=ptr->next;
ptr->next=ptr->next->next;
delete(temp);
}

```

打印链表

```

void List::print()//打印链表
{
listNode *ptr=this->head;
while(ptr!=NULL)
{
    cout<<ptr->data<<" ";
ptr=ptr->next;
}
cout<<endl;
}

```

链表的析构函数

```

void List::insertAtPos(int pos,int val)
{
listNode *newnode=new listNode(val);
if(pos==1)
{
    //as head
newnode->next=this->head;
this->head=newnode;
return;
}
pos--;
listNode *ptr=this->head;
while(ptr!=NULL && --pos)
{
    ptr=ptr->next;
}
if(ptr==NULL)
return;//not enough elements
newnode->next=ptr->next;
ptr->next=newnode;
}

```

Removing a node from the list

```

void List::remove(int toBeRemoved)//removing an element
{
if(this->head==NULL)
return; //empty
if(this->head->data==toBeRemoved)
{
    //first node to be removed
listNode *temp=this->head;
this->head=this->head->next;
delete(temp);
return;
}
listNode *ptr=this->head;
while(ptr->next!=NULL && ptr->next->data!=toBeRemoved)
ptr=ptr->next;
if(ptr->next==NULL)
return;//not found
listNode *temp=ptr->next;
ptr->next=ptr->next->next;
delete(temp);
}

```

Print the list

```

void List::print()//printing the list
{
listNode *ptr=this->head;
while(ptr!=NULL)
{
    cout<<ptr->data<<" ";
ptr=ptr->next;
}
cout<<endl;
}

```

Destructor for the list

```
List::~List()
{
listNode *ptr=this->head,*next=NULL;
    while(ptr!=NULL)
    {
next=ptr->next;
    delete(ptr);
    ptr=next;
    }
}
```

```
List::~List()
{
listNode *ptr=this->head,*next=NULL;
    while(ptr!=NULL)
    {
next=ptr->next;
    delete(ptr);
    ptr=next;
    }
}
```

第77章：模板

类、函数以及（自C++14起）变量都可以是模板。模板是一段带有一些自由参数的代码，当所有参数被指定后，它将成为一个具体的类、函数或变量。参数可以是类型、值，或者是模板本身。一个著名的模板是std::vector，当指定元素类型时，它就成为一个具体的容器类型，例如std::vector<int>。

第77.1节：基本类模板

类模板的基本思想是模板参数在编译时被替换为某种类型。结果是同一个类可以被多种类型重复使用。用户在声明类的变量时指定将使用哪种类型。下面在main()中展示了三个示例：

```
#include <iostream>
using std::cout;

template <typename T> // 一个简单的类，用于保存任意类型的一个数字
class Number {
public:
    void setNum(T n); // 将类字段设置为给定的数字
    T plus1() const; // 返回类字段的“后继”值
private:
    T num; // 类字段
};

template <typename T> // 将类字段设置为给定的数字
void Number<T>::setNum(T n) {
    num = n;
}

template <typename T> // 返回类字段的“后继”值
T Number<T>::plus1() const {
    return num + 1;
}

int main() {
    Number<int> anInt; // 使用整数测试 (int 替代类中的 T)
    anInt.setNum(1);
    cout << "我的整数 + 1 是 " << anInt.plus1() << ""; // 输出 2

    Number<double> aDouble; // 测试 double 类型
    aDouble.setNum(3.1415926535897);
    cout << "我的 double + 1 是 " << aDouble.plus1() << ""; // 输出 4.14159

    Number<float> aFloat; // 测试 float 类型
    aFloat.setNum(1.4);
    cout << "我的 float + 1 是 " << aFloat.plus1() << ""; // 输出 2.4
    return 0; // 成功
}
```

第 77.2 节：函数模板

模板也可以应用于函数（以及更传统的结构），效果相同。

```
// 'T' 代表未知类型
// 我们的两个参数将是相同类型。
```

Chapter 77: Templates

Classes, functions, and (since C++14) variables can be templated. A template is a piece of code with some free parameters that will become a concrete class, function, or variable when all parameters are specified. Parameters can be types, values, or themselves templates. A well-known template is std::vector, which becomes a concrete container type when the element type is specified, e.g., std::vector<int>.

Section 77.1: Basic Class Template

The basic idea of a class template is that the template parameter gets substituted by a type at compile time. The result is that the same class can be reused for multiple types. The user specifies which type will be used when a variable of the class is declared. Three examples of this are shown in main():

```
#include <iostream>
using std::cout;

template <typename T> // A simple class to hold one number of any type
class Number {
public:
    void setNum(T n); // Sets the class field to the given number
    T plus1() const; // returns class field's "follower"
private:
    T num; // Class field
};

template <typename T> // Set the class field to the given number
void Number<T>::setNum(T n) {
    num = n;
}

template <typename T> // returns class field's "follower"
T Number<T>::plus1() const {
    return num + 1;
}

int main() {
    Number<int> anInt; // Test with an integer (int replaces T in the class)
    anInt.setNum(1);
    cout << "My integer + 1 is " << anInt.plus1() << "\n"; // Prints 2

    Number<double> aDouble; // Test with a double
    aDouble.setNum(3.1415926535897);
    cout << "My double + 1 is " << aDouble.plus1() << "\n"; // Prints 4.14159

    Number<float> aFloat; // Test with a float
    aFloat.setNum(1.4);
    cout << "My float + 1 is " << aFloat.plus1() << "\n"; // Prints 2.4

    return 0; // Successful completion
}
```

Section 77.2: Function Templates

Templating can also be applied to functions (as well as the more traditional structures) with the same effect.

```
// 'T' stands for the unknown type
// Both of our arguments will be of the same type.
```

```
template<typename T>
void printSum(T add1, T add2)
{
    std::cout << (add1 + add2) << std::endl;
}
```

这随后可以像结构模板一样使用。

```
printSum<int>(4, 5);
printSum<float>(4.5f, 8.9f);
```

在这两种情况下，模板参数被用来替换参数的类型；结果就像普通的C++函数一样（如果参数类型与模板类型不匹配，编译器会应用标准转换）。

模板函数的一个额外特性（与模板类不同）是编译器可以根据传递给函数的参数推断模板参数。

```
printSum(4, 5); // 两个参数都是int类型。
                // 这允许编译器推断类型
                // T 也是int。
```

```
printSum(5.0, 4); // 在这种情况下，参数是两种不同的类型。
                  // 编译器无法推断T的类型
                  // 因为存在矛盾。因此
                  // 这是一个编译时错误。
```

此功能允许我们在结合模板结构和函数时简化代码。标准库中有一个常见模式，允许我们使用辅助函数make_X()来创建模板结构X。

```
// make_X 模式看起来是这样的。
// 1) 一个包含一个或多个模板类型的模板结构。
template<typename T1, typename T2>
struct MyPair
{
    T1 first;
    T2 second;
};

// 2) 一个 make 函数，其参数类型对应模板结构中的每个模板参数
。
template<typename T1, typename T2>
MyPair<T1, T2> make_MyPair(T1 t1, T2 t2)
{
    return MyPair<T1, T2>{t1, t2};
}
```

这有什么帮助？

```
auto val1 = MyPair<int, float>{5, 8.7}; // 显式定义类型创建对象
auto val2 = make_MyPair(5, 8.7); // 使用参数类型创建对象。
                                // 在这段代码中，val1 和 val2 类型相同。
```

注意：这并不是为了缩短代码而设计的。它旨在使代码更健壮。通过只需在一个地方修改代码即可改变类型，而不必在多个地方修改。

```
template<typename T>
void printSum(T add1, T add2)
{
    std::cout << (add1 + add2) << std::endl;
}
```

This can then be used in the same way as structure templates.

```
printSum<int>(4, 5);
printSum<float>(4.5f, 8.9f);
```

In both these case the template argument is used to replace the types of the parameters; the result works just like a normal C++ function (if the parameters don't match the template type the compiler applies the standard conversions).

One additional property of template functions (unlike template classes) is that the compiler can infer the template parameters based on the parameters passed to the function.

```
printSum(4, 5); // Both parameters are int.
                // This allows the compiler deduce that the type
                // T is also int.

printSum(5.0, 4); // In this case the parameters are two different types.
                  // The compiler is unable to deduce the type of T
                  // because there are contradictions. As a result
                  // this is a compile time error.
```

This feature allows us to simplify code when we combine template structures and functions. There is a common pattern in the standard library that allows us to make `template` structure X using a helper function `make_X()`.

```
// The make_X pattern looks like this.
// 1) A template structure with 1 or more template types.
template<typename T1, typename T2>
struct MyPair
{
    T1 first;
    T2 second;
};

// 2) A make function that has a parameter type for
// each template parameter in the template structure.
template<typename T1, typename T2>
MyPair<T1, T2> make_MyPair(T1 t1, T2 t2)
{
    return MyPair<T1, T2>{t1, t2};
}
```

How does this help?

```
auto val1 = MyPair<int, float>{5, 8.7}; // Create object explicitly defining the types
auto val2 = make_MyPair(5, 8.7); // Create object using the types of the parameters.
                                // In this code both val1 and val2 are the same
                                // type.
```

Note: This is not designed to shorten the code. This is designed to make the code more robust. It allows the types to be changed by changing the code in a single place rather than in multiple locations.

第77.3节：可变参数模板数据结构

版本 ≥ C++14

定义具有可变数量和类型的数据成员且在编译时确定的类或结构体通常非常有用。典型的例子是std::tuple，但有时需要定义自己的自定义结构体。下面是一个使用组合（而非继承，如std::tuple）定义结构体的示例。首先是通用（空）定义，它也作为后续特化中递归终止的基准情况：

```
template<typename ... T>
struct DataStructure {};
```

这已经允许我们定义一个空结构体，DataStructure<> data，尽管这目前还不是很有用。

接下来是递归特化情况：

```
template<typename T, typename ... Rest>
struct DataStructure<T, Rest ... >
{
    DataStructure(const T& first, const Rest& ... rest)
        : first(first)
        , rest(rest...)
    {}

    T first;
    DataStructure<Rest ... > rest;
};
```

这现在足以让我们创建任意数据结构，比如DataStructure<int, float, std::string> data(1, 2.1, "hello")。

那么发生了什么？首先，请注意这是一个特化，其要求至少存在一个可变参数模板参数（即上面的T），而不关心包Rest的具体组成。知道T存在允许定义其数据成员first。其余数据递归地打包为

DataStructure<Rest ... > rest。构造函数初始化这两个成员，包括对rest成员的递归构造函数调用。

为了更好地理解，我们可以通过一个例子来说明：假设你有一个声明DataStructure<int, float> data。该声明首先匹配该特化，生成一个结构体，包含int first和DataStructure<float> rest数据成员。该rest定义再次匹配此特化，创建其自身的float first和DataStructure<> rest成员。最后这个rest匹配基例定义，生成一个空结构体。

你可以将其可视化如下：

```
DataStructure<int, float>
-> int first
-> DataStructure<float> rest
    -> float first
    -> DataStructure<> rest
        -> (empty)
```

现在我们有了数据结构，但它还不是特别有用，因为我们无法轻松访问单个数据元素（例如，要访问DataStructure<int, float, std::string> data的最后一个成员，我们必须使用data.rest.rest.first，这并不太友好）。所以我们给它添加了一个get方法（只在特化中需要，因为基类结构没有数据可供get）：

Section 77.3: Variadic template data structures

Version ≥ C++14

It is often useful to define classes or structures that have a variable number and type of data members which are defined at compile time. The canonical example is std::tuple, but sometimes it is necessary to define your own custom structures. Here is an example that defines the structure using compounding (rather than inheritance) as with std::tuple. Start with the general (empty) definition, which also serves as the base-case for recursion termination in the later specialisation:

```
template<typename ... T>
struct DataStructure {};
```

This already allows us to define an empty structure, DataStructure<> data, albeit that isn't very useful yet.

Next comes the recursive case specialisation:

```
template<typename T, typename ... Rest>
struct DataStructure<T, Rest ... >
{
    DataStructure(const T& first, const Rest& ... rest)
        : first(first)
        , rest(rest...)
    {}

    T first;
    DataStructure<Rest ... > rest;
};
```

This is now sufficient for us to create arbitrary data structures, like DataStructure<int, float, std::string> data(1, 2.1, "hello").

So what's going on? First, note that this is a specialisation whose requirement is that at least one variadic template parameter (namely T above) exists, whilst not caring about the specific makeup of the pack Rest. Knowing that T exists allows the definition of its data member, first. The rest of the data is recursively packaged as DataStructure<Rest ... > rest. The constructor initiates both of those members, including a recursive constructor call to the rest member.

To understand this better, we can work through an example: suppose you have a declaration DataStructure<int, float> data. The declaration first matches against the specialisation, yielding a structure with int first and DataStructure<float> rest data members. The rest definition again matches this specialisation, creating its own float first and DataStructure<> rest members. Finally this last rest matches against the base-case definition, producing an empty structure.

You can visualise this as follows:

```
DataStructure<int, float>
-> int first
-> DataStructure<float> rest
    -> float first
    -> DataStructure<> rest
        -> (empty)
```

Now we have the data structure, but it's not terribly useful yet as we cannot easily access the individual data elements (for example to access the last member of DataStructure<int, float, std::string> data we would have to use data.rest.rest.first, which is not exactly user-friendly). So we add a get method to it (only needed

只在特化中需要，因为基类结构没有数据可供get) :

```
template<typename T, typename ... Rest>
struct DataStructure<T, Rest ... >
{
    ...
    template<size_t idx>
    auto get()
    {
        return GetHelper<idx, DataStructure<T, Rest...>>::get(*this);
    }
    ...
};
```

如你所见，这个get成员函数本身是模板化的一一这次是针对所需成员的索引（因此用法可以是data.get<1>()，类似于std::tuple）。实际工作由辅助类GetHelper中的静态函数完成。我们不能直接在DataStructure的get中定义所需功能的原因是（正如我们稍后将看到的）我们需要针对idx进行特化——但无法在不特化包含类模板的情况下特化模板成员函数。另请注意这里使用了C++14风格的auto，这大大简化了我们的工作，否则我们需要一个相当复杂的返回类型表达式。

接下来是辅助类。这次我们需要一个空的前置声明和两个特化。首先是声明：

```
template<size_t idx, typename T>
struct GetHelper;
```

现在是基例（当idx==0时）。在这种情况下，我们只返回first成员：

```
template<typename T, typename ... Rest>
struct GetHelper<0, DataStructure<T, Rest ... >>
{
    static T get(DataStructure<T, Rest...>& data)
    {
        return data.first;
    }
};
```

在递归情况下，我们将 idx 减一并调用 GetHelper 来处理 rest 成员：

```
template<size_t idx, typename T, typename ... Rest>
struct GetHelper<idx, DataStructure<T, Rest ... >>
{
    static auto get(DataStructure<T, Rest...>& data)
    {
        return GetHelper<idx-1, DataStructure<Rest ...>>::get(data.rest);
    }
};
```

举个例子，假设我们有 DataStructure<int, float> data，并且需要调用 data.get<1>()。

这会调用 GetHelper<1, DataStructure<int, float>>::get(data)（第二个特化版本），它又会调用 GetHelper<0, DataStructure<float>>::get(data.rest)，最终（因为此时 idx 为 0，使用第一个特化版本）返回 data.rest.first。

就是这样！下面是完整的可运行代码，并在 main 函数中给出了一些示例用法：

in the specialisation as the base-case structure has no data to get):

```
template<typename T, typename ... Rest>
struct DataStructure<T, Rest ... >
{
    ...
    template<size_t idx>
    auto get()
    {
        return GetHelper<idx, DataStructure<T, Rest...>>::get(*this);
    }
    ...
};
```

As you can see this get member function is itself templated - this time on the index of the member that is needed (so usage can be things like data.get<1>(), similar to std::tuple). The actual work is done by a static function in a helper class, GetHelper. The reason we can't define the required functionality directly in DataStructure's get is because (as we will shortly see) we would need to specialise on idx - but it isn't possible to specialise a template member function without specialising the containing class template. Note also the use of a C++14-style auto here makes our lives significantly simpler as otherwise we would need quite a complicated expression for the return type.

So on to the helper class. This time we will need an empty forward declaration and two specialisations. First the declaration:

```
template<size_t idx, typename T>
struct GetHelper;
```

Now the base-case (when idx==0). In this case we just return the first member:

```
template<typename T, typename ... Rest>
struct GetHelper<0, DataStructure<T, Rest ... >>
{
    static T get(DataStructure<T, Rest...>& data)
    {
        return data.first;
    }
};
```

In the recursive case, we decrement idx and invoke the GetHelper for the rest member:

```
template<size_t idx, typename T, typename ... Rest>
struct GetHelper<idx, DataStructure<T, Rest ... >>
{
    static auto get(DataStructure<T, Rest...>& data)
    {
        return GetHelper<idx-1, DataStructure<Rest ...>>::get(data.rest);
    }
};
```

To work through an example, suppose we have DataStructure<int, float> data and we need data.get<1>(). This invokes GetHelper<1, DataStructure<int, float>>::get(data) (the 2nd specialisation), which in turn invokes GetHelper<0, DataStructure<float>>::get(data.rest), which finally returns (by the 1st specialisation as now idx is 0) data.rest.first.

So that's it! Here is the whole functioning code, with some example use in the main function:

```

#include <iostream>

template<size_t idx, typename T>
struct GetHelper;

template<typename ... T>
struct DataStructure
{
};

template<typename T, typename ... Rest>
struct DataStructure<T, Rest ...>
{
    DataStructure(const T& first, const Rest& ... rest)
        : first(first)
        , rest(rest...)
    {}

    T first;
    DataStructure<Rest ...> rest;

    template<size_t idx>
    auto get()
    {
        return GetHelper<idx, DataStructure<T, Rest...>>::get(*this);
    }
};

template<typename T, typename ... Rest>
struct GetHelper<0, DataStructure<T, Rest ...>>
{
    static T get(DataStructure<T, Rest...>& data)
    {
        return data.first;
    }
};

template<size_t idx, typename T, typename ... Rest>
struct GetHelper<idx, DataStructure<T, Rest ...>>
{
    static auto get(DataStructure<T, Rest...>& data)
    {
        return GetHelper<idx-1, DataStructure<Rest ...>>::get(data.rest);
    }
};

int main()
{
    DataStructure<int, float, std::string> data(1, 2.1, "Hello");

    std::cout << data.get<0>() << std::endl;
    std::cout << data.get<1>() << std::endl;
    std::cout << data.get<2>() << std::endl;

    return 0;
}

```

第77.4节：参数转发

模板可以使用转发引用同时接受左值和右值引用：

```

#include <iostream>

template<size_t idx, typename T>
struct GetHelper;

template<typename ... T>
struct DataStructure
{
};

template<typename T, typename ... Rest>
struct DataStructure<T, Rest ...>
{
    DataStructure(const T& first, const Rest& ... rest)
        : first(first)
        , rest(rest...)
    {}

    T first;
    DataStructure<Rest ...> rest;

    template<size_t idx>
    auto get()
    {
        return GetHelper<idx, DataStructure<T, Rest...>>::get(*this);
    }
};

template<typename T, typename ... Rest>
struct GetHelper<0, DataStructure<T, Rest ...>>
{
    static T get(DataStructure<T, Rest...>& data)
    {
        return data.first;
    }
};

template<size_t idx, typename T, typename ... Rest>
struct GetHelper<idx, DataStructure<T, Rest ...>>
{
    static auto get(DataStructure<T, Rest...>& data)
    {
        return GetHelper<idx-1, DataStructure<Rest ...>>::get(data.rest);
    }
};

int main()
{
    DataStructure<int, float, std::string> data(1, 2.1, "Hello");

    std::cout << data.get<0>() << std::endl;
    std::cout << data.get<1>() << std::endl;
    std::cout << data.get<2>() << std::endl;

    return 0;
}

```

Section 77.4: Argument forwarding

Template may accept both lvalue and rvalue references using *forwarding reference*:

```
template <typename T>
void f(T &&t);
```

在这种情况下，t 的实际类型将根据上下文进行推导：

```
struct X { };

X x;
f(x); // 调用 f<X&>(x)
f(X()); // 调用 f<X>(x)
```

在第一种情况下，类型 T 被推导为 X 的引用 (X&)，t 的类型是 X 的左值引用，而在第二种情况下，T 的类型被推导为 X，t 的类型是 X 的右值引用 (X&&)。

注意：值得注意的是，在第一种情况下，`decltype(t)` 与 T 相同，但在第二种情况下则不相同。

为了将 t 完美转发到另一个函数，无论它是左值还是右值引用，都必须使用 `std::forward`：

```
template <typename T>
void f(T &&t) {
    g(std::forward<T>(t));
}
```

转发引用可以与可变参数模板一起使用：

```
template <typename... Args>
void f(Args&&... args) {
    g(std::forward<Args>(args)...);
}
```

注意：转发引用只能用于模板参数，例如，在以下代码中，v 是一个右值引用，而不是转发引用：

```
#include <vector>

template <typename T>
void f(std::vector<T> &&v);
```

第77.5节：部分模板特化

与完全模板特化相反，部分模板特化允许引入模板时固定现有模板的一部分参数。部分模板特化仅适用于模板类/结构体：

```
// 常见情况：
template<typename T, typename U>
struct S {
    T t_val;
    U u_val;
};

// 当第一个模板参数固定为int时的特殊情况
template<typename V>
struct S<int, V> {
    double another_value;
    int foo(double arg) { // Do something}
};
```

```
template <typename T>
void f(T &&t);
```

In this case, the real type of t will be deduced depending on the context:

```
struct X { };

X x;
f(x); // calls f<X&>(x)
f(X()); // calls f<X>(x)
```

In the first case, the type T is deduced as *reference to X (X&)*, and the type of t is *lvalue reference to X*, while in the second case the type of T is deduced as X and the type of t as *rvalue reference to X (X&&)*.

Note: It is worth noticing that in the first case, `decltype(t)` is the same as T, but not in the second.

In order to perfectly forward t to another function ,whether it is an lvalue or rvalue reference, one must use `std::forward`:

```
template <typename T>
void f(T &&t) {
    g(std::forward<T>(t));
}
```

Forwarding references may be used with variadic templates:

```
template <typename... Args>
void f(Args&&... args) {
    g(std::forward<Args>(args)...);
}
```

Note: Forwarding references can only be used for template parameters, for instance, in the following code, v is a rvalue reference, not a forwarding reference:

```
#include <vector>

template <typename T>
void f(std::vector<T> &&v);
```

Section 77.5: Partial template specialization

In contrast of a full template specialization partial template specialization allows to introduce template with some of the arguments of existing template fixed. Partial template specialization is only available for template class/structs:

```
// Common case:
template<typename T, typename U>
struct S {
    T t_val;
    U u_val;
};

// Special case when the first template argument is fixed to int
template<typename V>
struct S<int, V> {
    double another_value;
    int foo(double arg) { // Do something}
};
```

};

如上所示，部分模板特化可能引入完全不同的数据和函数成员。

当实例化部分特化模板时，会选择最合适的特化。例如，定义一个模板和两个部分特化：

```
template<typename T, typename U, typename V>
struct S {
    static void foo() {
        std::cout << "通用情况";
    }
};

template<typename U, typename V>
struct S<int, U, V> {
    static void foo() {
        std::cout << "T = int";
    }
};

template<typename V>
struct S<int, double, V> {
    static void foo() {
        std::cout << "T = int, U = double";
    }
};
```

现在以下调用：

```
S<std::string, int, double>::foo();
S<int, float, std::string>::foo();
S<int, double, std::string>::foo();
```

将打印

```
通用情况
T = int
T = int, U = double
```

函数模板只能完全特化：

```
template<typename T, typename U>
void foo(T t, U u) {
    std::cout << "通用情况: " << t << " " << u << std::endl;
}

// 可以。
template<int, int>
void foo<int, int>(int a1, int a2) {
    std::cout << "Two ints: " << a1 << " " << a2 << std::endl;
}

void invoke_foo() {
    foo(1, 2.1); // 输出 "General case: 1 2.1"
    foo(1, 2);   // 输出 "Two ints: 1 2"
}
```

};

As shown above, partial template specializations may introduce completely different sets of data and function members.

When a partially specialized template is instantiated, the most suitable specialization is selected. For example, let's define a template and two partial specializations:

```
template<typename T, typename U, typename V>
struct S {
    static void foo() {
        std::cout << "General case\n";
    }
};

template<typename U, typename V>
struct S<int, U, V> {
    static void foo() {
        std::cout << "T = int\n";
    }
};

template<typename V>
struct S<int, double, V> {
    static void foo() {
        std::cout << "T = int, U = double\n";
    }
};
```

Now the following calls:

```
S<std::string, int, double>::foo();
S<int, float, std::string>::foo();
S<int, double, std::string>::foo();
```

will print

```
General case
T = int
T = int, U = double
```

Function templates may only be fully specialized:

```
template<typename T, typename U>
void foo(T t, U u) {
    std::cout << "General case: " << t << " " << u << std::endl;
}

// OK.
template<int, int>
void foo<int, int>(int a1, int a2) {
    std::cout << "Two ints: " << a1 << " " << a2 << std::endl;
}

void invoke_foo() {
    foo(1, 2.1); // Prints "General case: 1 2.1"
    foo(1, 2);   // Prints "Two ints: 1 2"
}
```

```
// 编译错误：不允许部分函数特化。
template<typename U>
void foo<std::string, U>(std::string t, U u) {
    std::cout << "General case: " << t << " " << u << std::endl;
}
```

第77.6节：模板特化

你可以为模板类/方法的特定实例定义实现。

例如，如果你有：

```
template <typename T>
T sqrt(T t) { /* 一些通用实现 */ }
```

然后你可以这样写：

```
template<>
int sqrt<int>(int i) { /* 高度优化的整数实现 */ }
```

那么，写 `sqrt(4.0)` 的用户将获得通用实现，而 `sqrt(4)` 将获得专门化实现。

第77.7节：别名模板

版本 ≥ C++11

基本示例：

```
template<typename T> using pointer = T*;
```

此定义使得 `pointer<T>` 成为 `T*` 的别名。例如：

```
pointer<int> p = new int; // 等同于: int* p = new int;
```

别名模板不能被专门化。然而，可以通过让它们引用结构体中的嵌套类型间接实现该功能：

```
template<typename T>
struct nonconst_pointer_helper { typedef T* type; };

template<typename T>
struct nonconst_pointer_helper<T const> { typedef T* type; };

template<typename T> using nonconst_pointer = nonconst_pointer_helper<T>::type;
```

第77.8节：显式实例化

显式实例化定义从模板创建并声明一个具体的类、函数或变量，但暂时不使用它。显式实例化可以被其他翻译单元引用。这样可以避免在头文件中定义模板，如果模板只会用有限的参数集实例化。例如：

```
// print_string.h
template <class T>
void print_string(const T* str);
```

```
// Compilation error: partial function specialization is not allowed.
template<typename U>
void foo<std::string, U>(std::string t, U u) {
    std::cout << "General case: " << t << " " << u << std::endl;
}
```

Section 77.6: Template Specialization

You can define implementation for specific instantiations of a template class/method.

For example if you have:

```
template <typename T>
T sqrt(T t) { /* Some generic implementation */ }
```

You can then write:

```
template<>
int sqrt<int>(int i) { /* Highly optimized integer implementation */ }
```

Then a user that writes `sqrt(4.0)` will get the generic implementation whereas `sqrt(4)` will get the specialized implementation.

Section 77.7: Alias template

Version ≥ C++11

Basic example:

```
template<typename T> using pointer = T*;
```

This definition makes `pointer<T>` an alias of `T*`. For example:

```
pointer<int> p = new int; // equivalent to: int* p = new int;
```

Alias templates cannot be specialized. However, that functionality can be obtained indirectly by having them refer to a nested type in a struct:

```
template<typename T>
struct nonconst_pointer_helper { typedef T* type; };

template<typename T>
struct nonconst_pointer_helper<T const> { typedef T* type; };

template<typename T> using nonconst_pointer = nonconst_pointer_helper<T>::type;
```

Section 77.8: Explicit instantiation

An explicit instantiation definition creates and declares a concrete class, function, or variable from a template, without using it just yet. An explicit instantiation can be referenced from other translation units. This can be used to avoid defining a template in a header file, if it will only be instantiated with a finite set of arguments. For example:

```
// print_string.h
template <class T>
void print_string(const T* str);
```

```
// print_string.cpp
#include "print_string.h"
template void print_string(const char*);
template void print_string(const wchar_t*);
```

因为 `print_string<char>` 和 `print_string<wchar_t>` 在 `print_string.cpp` 中被显式实例化，链接器即使在头文件中没有定义 `print_string` 模板，也能找到它们。如果没有这些显式实例化声明，很可能发生链接错误。参见“为什么模板只能在头文件中实现？”

版本 ≥ C++11

如果显式实例化前加上 `extern` 关键字，它就变成显式实例化声明。对于给定特化的显式实例化声明的存在，会阻止当前翻译单元中该特化的隐式实例化。相反，原本会导致隐式实例化的对该特化的引用，可以引用同一或其他翻译单元中的显式实例化定义。

`foo.h`

```
#ifndef FOO_H
#define FOO_H
template <class T> void foo(T x) {
    // 复杂的实现
}
#endif
```

`foo.cpp`

```
#include "foo.h"
// 常见情况的显式实例化定义
template void foo(int);
template void foo(double);
```

`main.cpp`

```
#include "foo.h"
// 我们已经知道 foo.cpp 对这些有显式实例化定义
extern template void foo(double);
int main() {
    foo(42); // 在此实例化 foo<int>;
    // 浪费，因为 foo.cpp 已经提供了显式实例化！
    foo(3.14); // 此处不实例化 foo<double>;
    // 而是使用 foo.cpp 中的 foo<double> 实例化
}
```

```
// print_string.cpp
#include "print_string.h"
template void print_string(const char*);
template void print_string(const wchar_t*);
```

Because `print_string<char>` and `print_string<wchar_t>` are explicitly instantiated in `print_string.cpp`, the linker will be able to find them even though the `print_string` template is not defined in the header. If these explicit instantiation declarations were not present, a linker error would likely occur. See [Why can templates only be implemented in the header file?](#)

Version ≥ C++11

If an explicit instantiation definition is preceded by the `extern` keyword, it becomes an explicit instantiation *declaration* instead. The presence of an explicit instantiation declaration for a given specialization prevents the implicit instantiation of the given specialization within the current translation unit. Instead, a reference to that specialization that would otherwise cause an implicit instantiation can refer to an explicit instantiation definition in the same or another TU.

`foo.h`

```
#ifndef FOO_H
#define FOO_H
template <class T> void foo(T x) {
    // complicated implementation
}
#endif
```

`foo.cpp`

```
#include "foo.h"
// explicit instantiation definitions for common cases
template void foo(int);
template void foo(double);
```

`main.cpp`

```
#include "foo.h"
// we already know foo.cpp has explicit instantiation definitions for these
extern template void foo(double);
int main() {
    foo(42); // instantiates foo<int> here;
    // wasteful since foo.cpp provides an explicit instantiation already!
    foo(3.14); // does not instantiate foo<double> here;
    // uses instantiation of foo<double> in foo.cpp instead
}
```

第 77.9 节：非类型模板参数

除了类型作为模板参数外，我们还允许声明满足以下任一条件的常量表达式值作为模板参数：

- 整型或枚举类型，
- 指向对象的指针或指向函数的指针，
- 指向对象的左值引用或指向函数的左值引用，
- 指向成员的指针，
- `std::nullptr_t`。

Section 77.9: Non-type template parameter

Apart from types as a template parameter we are allowed to declare values of constant expressions meeting one of the following criteria:

- integral or enumeration type,
- pointer to object or pointer to function,
- lvalue reference to object or lvalue reference to function,
- pointer to member,
- `std::nullptr_t`.

像所有模板参数一样，非类型模板参数可以显式指定、使用默认值，或通过模板参数推导隐式推导。

非类型模板参数使用示例：

```
#include <iostream>

template<typename T, std::size_t size>
std::size_t size_of(T (&anArray)[size]) // 通过引用传递数组。要求。
{                                     // 一个精确的大小。我们允许所有大小
    return size;                   // 通过使用模板 "size"。
}

int main()
{
    char 一个字符数组[15];
    std::cout << "一个字符数组: " << size_of(一个字符数组) << "";

    int 一个数据数组[] = {1,2,3,4,5,6,7,8,9};   std::cout
    << "一个数据数组: " << size_of(一个数据数组) << "";
}
```

显式指定类型和非类型模板参数的示例：

```
#include <array>
int main ()
{
    std::array<int, 5> foo; // int 是类型参数，5 是非类型参数
}
```

非类型模板参数是实现模板递归的一种方式，并且使得元编程成为可能。

第77.10节：使用

auto声明非类型模板参数

在C++17之前，编写模板非类型参数时，必须先指定其类型。因此一个常见的写法是像这样写：

```
模板 <类 T, T N>
结构体 integral_constant {
    使用 类型 = T;
    static constexpr T value = N;
};

using five = integral_constant<int, 5>;
```

但是对于复杂的表达式，使用类似这样的写法需要在实例化模板时写出`decltype(expr)`, `expr`。解决方案是简化这种惯用法，直接允许使用`auto`：

```
版本 ≥ C++17
template <auto N>
struct integral_constant {
    using type = decltype(N);
    static constexpr type value = N;
};
```

Like all template parameters, non-type template parameters can be explicitly specified, defaulted, or derived implicitly via Template Argument Deduction.

Example of non-type template parameter usage:

```
#include <iostream>

template<typename T, std::size_t size>
std::size_t size_of(T (&anArray)[size]) // Pass array by reference. Requires.
{                                     // an exact size. We allow all sizes
    return size;                   // by using a template "size".
}

int main()
{
    char anArrayOfChar[15];
    std::cout << "anArrayOfChar: " << size_of(anArrayOfChar) << "\n";

    int anArrayOfData[] = {1,2,3,4,5,6,7,8,9};
    std::cout << "anArrayOfData: " << size_of(anArrayOfData) << "\n";
}
```

Example of explicitly specifying both type and non-type template parameters:

```
#include <array>
int main ()
{
    std::array<int, 5> foo; // int is a type parameter, 5 is non-type
}
```

Non-type template parameters are one of the ways to achieve template recurrence and enables to do Metaprogramming.

Section 77.10: Declaring non-type template arguments with auto

Prior to C++17, when writing a template non-type parameter, you had to specify its type first. So a common pattern became writing something like:

```
template <class T, T N>
struct integral_constant {
    using type = T;
    static constexpr T value = N;
};

using five = integral_constant<int, 5>;
```

But for complicated expressions, using something like this involves having to write `decltype(expr)`, `expr` when instantiating templates. The solution is to simplify this idiom and simply allow `auto`:

```
Version ≥ C++17
template <auto N>
struct integral_constant {
    using type = decltype(N);
    static constexpr type value = N;
};
```

```
using five = integral_constant<5>;  
unique_ptr的空自定义删除器
```

一个很好的激励示例可以来自尝试将空基类优化与unique_ptr的自定义删除器结合起来
unique_ptr。不同的C API删除器有不同的返回类型，但我们不关心——我们只想要一个能
适用于任何函数的东西：

```
template <auto DeleteFn>  
struct FunctionDeleter {  
    template <class T>  
    void operator()(T* ptr) const {  
        DeleteFn(ptr);  
    }  
};  
  
template <T, auto DeleteFn>  
using unique_ptr_deleter = std::unique_ptr<T, FunctionDeleter<DeleteFn>>;
```

现在你可以简单地使用任何可以接受类型为T参数的函数指针作为模板非类型参数，无论返回类型如何，并且获得一个无额外大小开销的unique_ptr：

```
unique_ptr_deleter<std::FILE, std::fclose> p;
```

第77.11节：模板模板参数

有时我们希望将一个模板类型传入模板中，而不固定其值。这就是模板模板参数的用途。非常简单的模板模板参数示例：

```
template <class T>  
struct Tag1 { };  
  
template <class T>  
struct Tag2 { };  
  
模板 <模板 <类> 类 标签>  
结构体 整数标签 {  
    typedef Tag<int> type;  
};  
  
int main() {  
    IntTag<Tag1>::type t;  
}  
版本 ≥ C++11  
  
#include <vector>  
#include <iostream>  
  
template <class T, template <class...> class C, class U>  
C<T> cast_all(const C<U> &c) {  
    C<T> result(c.begin(), c.end());  
    return result;  
}  
  
int main() {  
    std::vector<float> vf = {1.2, 2.6, 3.7};  
    auto vi = cast_all<int>(vf);  
    for(auto &&i: vi) {  
        std::cout << i << std::endl;  
    }
```

```
using five = integral_constant<5>;  
Empty custom deleter for unique_ptr
```

A nice motivating example can come from trying to combine the empty base optimization with a custom deleter for unique_ptr. Different C API deleters have different return types, but we don't care - we just want something to work for any function:

```
template <auto DeleteFn>  
struct FunctionDeleter {  
    template <class T>  
    void operator()(T* ptr) const {  
        DeleteFn(ptr);  
    }  
};  
  
template <T, auto DeleteFn>  
using unique_ptr_deleter = std::unique_ptr<T, FunctionDeleter<DeleteFn>>;
```

And now you can simply use any function pointer that can take an argument of type T as a template non-type parameter, regardless of return type, and get a no-size overhead unique_ptr out of it:

```
unique_ptr_deleter<std::FILE, std::fclose> p;
```

Section 77.11: Template template parameters

Sometimes we would like to pass into the template a template type without fixing its values. This is what template template parameters are created for. Very simple template template parameter examples:

```
template <class T>  
struct Tag1 { };  
  
template <class T>  
struct Tag2 { };  
  
template <template <class> class Tag>  
struct IntTag {  
    typedef Tag<int> type;  
};  
  
int main() {  
    IntTag<Tag1>::type t;  
}  
Version ≥ C++11  
  
#include <vector>  
#include <iostream>  
  
template <class T, template <class...> class C, class U>  
C<T> cast_all(const C<U> &c) {  
    C<T> result(c.begin(), c.end());  
    return result;  
}  
  
int main() {  
    std::vector<float> vf = {1.2, 2.6, 3.7};  
    auto vi = cast_all<int>(vf);  
    for(auto &&i: vi) {  
        std::cout << i << std::endl;  
    }
```

第77.12节：默认模板参数值

就像函数参数一样，模板参数也可以有默认值。所有带默认值的模板参数必须声明在模板参数列表的末尾。基本思想是，在模板实例化时，可以省略带默认值的模板参数。

默认模板参数值使用的简单示例：

```
template <class T, size_t N = 10>
struct my_array {
    T arr[N];
};

int main() {
    /* 默认参数被忽略, N = 5 */
    my_array<int, 5> a;

    /* 输出 a.arr 的长度:5 */
    std::cout << sizeof(a.arr) / sizeof(int) << std::endl;

    /* 最后一个参数被省略, N = 10 */
    my_array<int> b;

    /* 输出 b.arr 的长度:10 */
    std::cout << sizeof(b.arr) / sizeof(int) << std::endl;
}
```

Section 77.12: Default template parameter value

Just like in case of the function arguments, template parameters can have their default values. All template parameters with a default value have to be declared at the end of the template parameter list. The basic idea is that the template parameters with default value can be omitted while template instantiation.

Simple example of default template parameter value usage:

```
template <class T, size_t N = 10>
struct my_array {
    T arr[N];
};

int main() {
    /* Default parameter is ignored, N = 5 */
    my_array<int, 5> a;

    /* Print the length of a.arr: 5 */
    std::cout << sizeof(a.arr) / sizeof(int) << std::endl;

    /* Last parameter is omitted, N = 10 */
    my_array<int> b;

    /* Print the length of a.arr: 10 */
    std::cout << sizeof(b.arr) / sizeof(int) << std::endl;
}
```

第78章：表达式模板

第78.1节：表达式模板的基本示例

表达式模板是一种主要用于科学计算的编译时优化技术。它的主要目的是避免不必要的临时变量，并通过单次遍历（通常是在对数值聚合体进行操作时）优化循环计算。表达式模板最初是为了解决在实现数值数组（Array）或矩阵（Matrix）类型时，天真的运算符重载所带来的低效问题而设计的。Bjarne Stroustrup在其最新版本的著作《C++程序设计语言》中引入了表达式模板的等效术语，称之为“融合操作”。

在真正深入表达式模板之前，你应该先理解为什么你需要它们。为此，考虑下面给出的非常简单的矩阵类：

```
template <typename T, std::size_t COL, std::size_t ROW>
class Matrix {
public:
    using value_type = T;

    Matrix() : values(COL * ROW) {}

    static size_t cols() { return COL; }
    static size_t rows() { return ROW; }

    const T& operator()(size_t x, size_t y) const { return values[y * COL + x]; }
    T& operator()(size_t x, size_t y) { return values[y * COL + x]; }

private:
    std::vector<T> values;
};

template <typename T, std::size_t COL, std::size_t ROW>
Matrix<T, COL, ROW> operator+(const Matrix<T, COL, ROW>& lhs, const Matrix<T, COL, ROW>& rhs)
{
    Matrix<T, COL, ROW> result;

    for (size_t y = 0; y != lhs.rows(); ++y) {
        for (size_t x = 0; x != lhs.cols(); ++x) {
            result(x, y) = lhs(x, y) + rhs(x, y);
        }
    }
    return result;
}
```

根据之前的类定义，你现在可以编写如下的矩阵表达式：

```
const std::size_t cols = 2000;
const std::size_t rows = 1000;

Matrix<double, cols, rows> a, b, c;

// 初始化 a、b 和 c
for (std::size_t y = 0; y != rows; ++y) {
    for (std::size_t x = 0; x != cols; ++x) {
        a(x, y) = 1.0;
        b(x, y) = 2.0;
    }
}
```

Chapter 78: Expression templates

Section 78.1: A basic example illustrating expression templates

An expression template is a compile-time optimization technique used mostly in scientific computing. Its main purpose is to avoid unnecessary temporaries and optimize loop calculations using a single pass (typically when performing operations on numerical aggregates). Expression templates were initially devised in order to circumvent the inefficiencies of naïve operator overloading when implementing numerical Array or Matrix types. An equivalent terminology for expression templates has been introduced by Bjarne Stroustrup, who calls them "fused operations" in the latest version of his book, "The C++ Programming Language".

Before actually diving into expression templates, you should understand why you need them in the first place. To illustrate this, consider the very simple Matrix class given below:

```
template <typename T, std::size_t COL, std::size_t ROW>
class Matrix {
public:
    using value_type = T;

    Matrix() : values(COL * ROW) {}

    static size_t cols() { return COL; }
    static size_t rows() { return ROW; }

    const T& operator()(size_t x, size_t y) const { return values[y * COL + x]; }
    T& operator()(size_t x, size_t y) { return values[y * COL + x]; }

private:
    std::vector<T> values;
};

template <typename T, std::size_t COL, std::size_t ROW>
Matrix<T, COL, ROW> operator+(const Matrix<T, COL, ROW>& lhs, const Matrix<T, COL, ROW>& rhs)
{
    Matrix<T, COL, ROW> result;

    for (size_t y = 0; y != lhs.rows(); ++y) {
        for (size_t x = 0; x != lhs.cols(); ++x) {
            result(x, y) = lhs(x, y) + rhs(x, y);
        }
    }
    return result;
}
```

Given the previous class definition, you can now write Matrix expressions such as:

```
const std::size_t cols = 2000;
const std::size_t rows = 1000;

Matrix<double, cols, rows> a, b, c;

// initialize a, b & c
for (std::size_t y = 0; y != rows; ++y) {
    for (std::size_t x = 0; x != cols; ++x) {
        a(x, y) = 1.0;
        b(x, y) = 2.0;
    }
}
```

```

    c(x, y) = 3.0;
}

Matrix<double, 列, 行> d = a + b + c; // d(x, y) = 6

```

如上所示，能够重载operator+()为你提供了一种类似于矩阵自然数学符号的表示法。

不幸的是，与等效的“手工制作”版本相比，之前的实现效率也非常低下。

要理解原因，必须考虑当你写出诸如Matrix $d = a + b$ 这样的表达式时会发生什么 $+c$ 。实际上这展开为 $((a + b) + c)$ 或operator+(operator+(a, b), c)。换句话说，内部循环operator+()被执行了两次，而实际上可以很容易地在一次遍历中完成。这也导致创建了两个临时变量，进一步降低了性能。本质上，通过增加使用接近其数学对应符号的灵活性，你也使得Matrix类的效率大大降低。

例如，如果没有运算符重载，你可以使用单次遍历来实现更高效的矩阵求和：

```

template<typename T, std::size_t COL, std::size_t ROW>
Matrix<T, COL, ROW> add3(const Matrix<T, COL, ROW>& a,
                         const Matrix<T, COL, ROW>& b,
                         const Matrix<T, COL, ROW>& c)
{
    Matrix<T, COL, ROW> result;
    for (size_t y = 0; y != ROW; ++y) {
        for (size_t x = 0; x != COL; ++x) {
            result(x, y) = a(x, y) + b(x, y) + c(x, y);
        }
    }
    return result;
}

```

然而，上一个例子有其自身的缺点，因为它为Matrix类创建了一个更加复杂的接口（你必须考虑诸如Matrix::add2()、Matrix::AddMultiply()等方法）。

相反，让我们退一步，看看如何调整运算符重载以更高效地执行。问题的根源在于表达式Matrix $d = a + b + c$ 被过早地“急切”求值，而你还没有机会构建完整的表达式树。换句话说，你真正想实现的是一次性计算 $a + b + c$ ，并且只在实际需要将结果赋值给 d 时才进行计算。

这就是表达式模板的核心思想：运算符operator+()不会立即计算两个Matrix实例相加的结果，而是返回一个“表达式模板”，以便在整个表达式树构建完成后进行后续求值。

例如，下面是一个对应于两种类型求和的表达式模板的可能实现：

```

template <typename LHS, typename RHS>
class MatrixSum
{
    公共:
        using value_type = typename LHS::value_type;

```

```

    c(x, y) = 3.0;
}

Matrix<double, cols, rows> d = a + b + c; // d(x, y) = 6

```

As illustrated above, being able to overload operator+() provides you with a notation which mimics the natural mathematical notation for matrices.

Unfortunately, the previous implementation is also highly inefficient compared to an equivalent "hand-crafted" version.

To understand why, you have to consider what happens when you write an expression such as Matrix $d = a + b + c$. This in fact expands to $((a + b) + c)$ or operator+(operator+(a, b), c). In other words, the loop inside operator+() is executed twice, whereas it could have been easily performed in a single pass. This also results in 2 temporaries being created, which further degrades performance. In essence, by adding the flexibility to use a notation close to its mathematical counterpart, you have also made the Matrix class highly inefficient.

For example, without operator overloading, you could implement a far more efficient Matrix summation using a single pass:

```

template<typename T, std::size_t COL, std::size_t ROW>
Matrix<T, COL, ROW> add3(const Matrix<T, COL, ROW>& a,
                         const Matrix<T, COL, ROW>& b,
                         const Matrix<T, COL, ROW>& c)
{
    Matrix<T, COL, ROW> result;
    for (size_t y = 0; y != ROW; ++y) {
        for (size_t x = 0; x != COL; ++x) {
            result(x, y) = a(x, y) + b(x, y) + c(x, y);
        }
    }
    return result;
}

```

The previous example however has its own disadvantages because it creates a far more convoluted interface for the Matrix class (you would have to consider methods such as Matrix::add2(), Matrix::AddMultiply() and so on).

Instead let us take a step back and see how we can adapt operator overloading to perform in a more efficient way

The problem stems from the fact that the expression Matrix $d = a + b + c$ is evaluated too "eagerly" before you have had an opportunity to build the entire expression tree. In other words, what you really want to achieve is to evaluate $a + b + c$ in one pass and only once you actually need to assign the resulting expression to d .

This is the core idea behind expression templates: instead of having operator+() evaluate immediately the result of adding two Matrix instances, it will return an "expression template" for future evaluation once the entire expression tree has been built.

For example, here is a possible implementation for an expression template corresponding to the summation of 2 types:

```

template <typename LHS, typename RHS>
class MatrixSum
{
    public:
        using value_type = typename LHS::value_type;

```

```

MatrixSum(const LHS& lhs, const RHS& rhs) : rhs(rhs), lhs(lhs) {}

value_type operator() (int x, int y) const {
    return lhs(x, y) + rhs(x, y);
}

private:
    const LHS& lhs;
    const RHS& rhs;
};

```

这是更新后的operator+()版本

```

template <typename LHS, typename RHS>
MatrixSum<LHS, RHS> operator+(const LHS& lhs, const LHS& rhs) {
    return MatrixSum<LHS, RHS>(lhs, rhs);
}

```

如你所见，operator+()不再返回两个Matrix实例相加的“急切求值”结果（即另一个Matrix实例），而是返回表示加法操作的表达式模板。

最重要的一点是，该表达式尚未被求值。它仅仅保存了对操作数的引用。

事实上，没有任何东西阻止你像下面这样实例化MatrixSum<>表达式模板：

```
MatrixSum<Matrix<double>, Matrix<double> > SumAB(a, b);
```

不过，你可以在后续实际需要求和结果时，按如下方式对表达式 `d = a + b` 进行求值：

```

for (std::size_t y = 0; y != a.rows(); ++y) {
    for (std::size_t x = 0; x != a.cols(); ++x) {
        d(x, y) = SumAB(x, y);
    }
}

```

如你所见，使用表达式模板的另一个好处是，你基本上实现了在一次遍历中计算 `a` 和 `b` 的和并赋值给 `d`。

此外，没有什么能阻止你组合多个表达式模板。例如，`a + b + c` 会生成如下表达式模板：

```
MatrixSum<MatrixSum<Matrix<double>, Matrix<double> >, Matrix<double> > SumABC(SumAB, c);
```

这里你同样可以通过一次遍历来计算最终结果：

```

for (std::size_t y = 0; y != a.rows(); ++y) {
    for (std::size_t x = 0; x != a.cols(); ++x) {
        d(x, y) = SumABC(x, y);
    }
}

```

最后，拼图的最后一块是将你的表达式模板实际插入到Matrix类中。这本质上是通过为Matrix::operator=()提供一个实现来完成的，该操作符接受表达式模板作为参数，并像你之前“手动”做的那样一次遍历计算它：

```
template <typename T, std::size_t COL, std::size_t ROW>
```

```

MatrixSum(const LHS& lhs, const RHS& rhs) : rhs(rhs), lhs(lhs) {}

value_type operator() (int x, int y) const {
    return lhs(x, y) + rhs(x, y);
}

private:
    const LHS& lhs;
    const RHS& rhs;
};

```

And here is the updated version of operator+()

```

template <typename LHS, typename RHS>
MatrixSum<LHS, RHS> operator+(const LHS& lhs, const LHS& rhs) {
    return MatrixSum<LHS, RHS>(lhs, rhs);
}

```

As you can see, operator+() no longer returns an "eager evaluation" of the result of adding 2 Matrix instances (which would be another Matrix instance), but instead an expression template representing the addition operation. The most important point to keep in mind is that the expression has not been evaluated yet. It merely holds references to its operands.

In fact, nothing stops you from instantiating the MatrixSum<> expression template as follows:

```
MatrixSum<Matrix<double>, Matrix<double> > SumAB(a, b);
```

You can however at a later stage, when you actually need the result of the summation, evaluate the expression `d = a + b` as follows:

```

for (std::size_t y = 0; y != a.rows(); ++y) {
    for (std::size_t x = 0; x != a.cols(); ++x) {
        d(x, y) = SumAB(x, y);
    }
}

```

As you can see, another benefit of using an expression template, is that you have basically managed to evaluate the sum of `a` and `b` and assign it to `d` in a single pass.

Also, nothing stops you from combining multiple expression templates. For example, `a + b + c` would result in the following expression template:

```
MatrixSum<MatrixSum<Matrix<double>, Matrix<double> >, Matrix<double> > SumABC(SumAB, c);
```

And here again you can evaluate the final result using a single pass:

```

for (std::size_t y = 0; y != a.rows(); ++y) {
    for (std::size_t x = 0; x != a.cols(); ++x) {
        d(x, y) = SumABC(x, y);
    }
}

```

Finally, the last piece of the puzzle is to actually plug your expression template into the Matrix class. This is essentially achieved by providing an implementation for Matrix::operator=(), which takes the expression template as an argument and evaluates it in one pass, as you did "manually" before:

```
template <typename T, std::size_t COL, std::size_t ROW>
```

```

class Matrix {
public:
    using value_type = T;

Matrix() : values(COL * ROW) {}

static size_t cols() { return COL; }
static size_t rows() { return ROW; }

const T& operator()(size_t x, size_t y) const { return values[y * COL + x]; }
T& operator()(size_t x, size_t y) { return values[y * COL + x]; }

template <typename E>
矩阵<T, 行, 列>& 操作符=(const E& 表达式) {
    for (std::size_t y = 0; y != 行数(); ++y) {
        for (std::size_t x = 0; x != cols(); ++x) {
            values[y * COL + x] = expression(x, y);
        }
    }
    return *this;
}

private:
std::vector<T> values;
};

```

```

class Matrix {
public:
    using value_type = T;

Matrix() : values(COL * ROW) {}

static size_t cols() { return COL; }
static size_t rows() { return ROW; }

const T& operator()(size_t x, size_t y) const { return values[y * COL + x]; }
T& operator()(size_t x, size_t y) { return values[y * COL + x]; }

template <typename E>
Matrix<T, COL, ROW>& operator=(const E& expression) {
    for (std::size_t y = 0; y != rows(); ++y) {
        for (std::size_t x = 0; x != cols(); ++x) {
            values[y * COL + x] = expression(x, y);
        }
    }
    return *this;
}

private:
std::vector<T> values;
};

```

第79章：奇异递归模板模式 (CRTP)

一种类从一个以自身作为模板参数之一的类模板继承的模式。CRTP通常用于在C++中提供静态多态性。

第79.1节：奇异递归模板模式 (CRTP)

CRTP是一种强大的、静态的虚函数和传统继承的替代方案，可用于在编译时赋予类型属性。它通过让一个基类模板接受派生类作为其模板参数之一来实现。这使得它可以合法地对其this指针进行static_cast转换为派生类。

当然，这也意味着CRTP类必须始终作为某个其他类的基类使用。并且派生类必须将自身传递给基类。

版本 ≥ C++14

假设你有一组容器，它们都支持函数begin()和end()。标准库对容器的要求需要更多功能。我们可以设计一个CRTP基类，仅基于begin()和end()，提供这些功能：

```
#include <iterator>
template <typename Sub>
class Container {
private:
    // self() 返回派生类型的引用
    Sub& self() { return *static_cast<Sub*>(this); }
    Sub const& self() const { return *static_cast<Sub const*>(this); }

public:
    decltype(auto) front() {
        return *self().begin();
    }

    decltype(auto) back() {
        return *std::prev(self().end());
    }

    decltype(auto) size() const {
        return std::distance(self().begin(), self().end());
    }

    decltype(auto) operator[](std::size_t i) {
        return *std::next(self().begin(), i);
    }
};
```

上述类为任何提供了 begin() 和 end() 的子类提供了 front()、back()、size() 和 operator[] 函数。一个示例子类是一个简单的动态分配数组：

```
#include <memory>
// 一个动态分配数组
template <typename T>
class DynArray : public Container<DynArray<T>> {
public:
```

Chapter 79: Curiously Recurring Template Pattern (CRTP)

A pattern in which a class inherits from a class template with itself as one of its template parameters. CRTP is usually used to provide *static polymorphism* in C++.

Section 79.1: The Curiously Recurring Template Pattern (CRTP)

CRTP is a powerful, static alternative to virtual functions and traditional inheritance that can be used to give types properties at compile time. It works by having a base class template which takes, as one of its template parameters, the derived class. This permits it to legally perform a `static_cast` of its `this` pointer to the derived class.

Of course, this also means that a CRTP class must *always* be used as the base class of some other class. And the derived class must pass itself to the base class.

Version ≥ C++14

Let's say you have a set of containers that all support the functions begin() and end(). The standard library's requirements for containers require more functionality. We can design a CRTP base class that provides that functionality, based solely on begin() and end():

```
#include <iterator>
template <typename Sub>
class Container {
private:
    // self() yields a reference to the derived type
    Sub& self() { return *static_cast<Sub*>(this); }
    Sub const& self() const { return *static_cast<Sub const*>(this); }

public:
    decltype(auto) front() {
        return *self().begin();
    }

    decltype(auto) back() {
        return *std::prev(self().end());
    }

    decltype(auto) size() const {
        return std::distance(self().begin(), self().end());
    }

    decltype(auto) operator[](std::size_t i) {
        return *std::next(self().begin(), i);
    }
};
```

The above class provides the functions front(), back(), size(), and operator[] for any subclass which provides begin() and end(). An example subclass is a simple dynamically allocated array:

```
#include <memory>
// A dynamically allocated array
template <typename T>
class DynArray : public Container<DynArray<T>> {
public:
```

```

using Base = Container<DynArray<T>>;
DynArray(std::size_t size)
    : size_{size},
data_{std::make_unique<T[]>(size_)}
{ }

T* begin() { return data_.get(); }
const T* begin() const { return data_.get(); }
T* end() { return data_.get() + size_; }
const T* end() const { return data_.get() + size_; }

private:
std::size_t size_;
std::unique_ptr<T[]> data_;
};

```

DynArray类的用户可以轻松使用CRTP基类提供的接口，方法如下：

```

DynArray<int> arr(10);
arr.front() = 2;
arr[2] = 5;
assert(arr.size() == 10);

```

实用性：该模式特别避免了运行时发生的虚函数调用，这些调用通常用于遍历继承层次结构，而是简单依赖于静态转换：

```

DynArray<int> arr(10);
DynArray<int>::Base & base = arr;
base.begin(); // 无虚函数调用

```

基类Container<DynArray<int>>中函数begin()内唯一的静态转换使编译器能够大幅优化代码，运行时不会发生虚函数表查找。

限制：由于基类是模板化的，并且对于两个不同的DynArray来说基类不同，因此无法像通常使用普通继承时那样将它们基类的指针存储在同一类型的数组中，普通继承中基类不依赖于派生类型：

```

class A {};
class B: public A {};

A* a = new B;

```

第79.2节：使用CRTP避免代码重复

访问者模式中的示例为CRTP提供了一个有力的用例：

```

struct IShape
{
    virtual ~IShape() = default;

    virtual void accept(IShapeVisitor&) const = 0;
};

struct Circle : IShape
{
    // ...
    // 每个形状都必须以相同的方式实现此方法

```

```

using Base = Container<DynArray<T>>;
DynArray(std::size_t size)
    : size_{size},
data_{std::make_unique<T[]>(size_)}
{ }

T* begin() { return data_.get(); }
const T* begin() const { return data_.get(); }
T* end() { return data_.get() + size_; }
const T* end() const { return data_.get() + size_; }

private:
std::size_t size_;
std::unique_ptr<T[]> data_;
};

```

Users of the DynArray class can use the interfaces provided by the CRTP base class easily as follows:

```

DynArray<int> arr(10);
arr.front() = 2;
arr[2] = 5;
assert(arr.size() == 10);

```

Usefulness: This pattern particularly avoids virtual function calls at run-time which occur to traverse down the inheritance hierarchy and simply relies on static casts:

```

DynArray<int> arr(10);
DynArray<int>::Base & base = arr;
base.begin(); // no virtual calls

```

The only static cast inside the function begin() in the base class Container<DynArray<int>> allows the compiler to drastically optimize the code and no virtual table look up happens at runtime.

Limitations: Because the base class is templated and different for two different DynArrays it is not possible to store pointers to their base classes in an type-homogenous array as one could generally do with normal inheritance where the base class is not dependent on the derived type:

```

class A {};
class B: public A {};

A* a = new B;

```

Section 79.2: CRTP to avoid code duplication

The example in Visitor Pattern provides a compelling use-case for CRTP:

```

struct IShape
{
    virtual ~IShape() = default;

    virtual void accept(IShapeVisitor&) const = 0;
};

struct Circle : IShape
{
    // ...
    // Each shape has to implement this method the same way

```

```

void accept(IShapeVisitor& visitor) const override { visitor.visit(*this); }
// ...
};

struct Square : IShape
{
    // ...
    // 每个形状都必须以相同的方式实现此方法
    void accept(IShapeVisitor& visitor) const override { visitor.visit(*this); }
    // ...
};

```

IShape的每个子类型都需要以相同的方式实现相同的函数。这会导致大量额外的代码。相反，我们可以在继承体系中引入一个新类型来为我们完成这项工作：

```

template <class Derived>
struct IShapeAcceptor : IShape {
    void accept(IShapeVisitor& visitor) const override {
        // 使用我们确切的类型进行访问
        visitor.visit(*static_cast<Derived const*>(this));
    }
};

```

现在，每个形状只需继承自该接收器即可：

```

struct Circle : IShapeAcceptor<Circle>
{
    Circle(const Point& center, double radius) : center(center), radius(radius) {}
    Point center;
    double 半径;
};

struct 正方形 : IShapeAcceptor<正方形>
{
    正方形(const 点& 左上角, double 边长) : 左上角(左上角), 边长(边长) {}
    点 左上角;
    double 边长;
};

```

无需重复代码。

```

void accept(IShapeVisitor& visitor) const override { visitor.visit(*this); }
// ...
};

struct Square : IShape
{
    // ...
    // 每个形状都必须以相同的方式实现此方法
    void accept(IShapeVisitor& visitor) const override { visitor.visit(*this); }
    // ...
};

```

Each child type of IShape needs to implement the same function the same way. That's a lot of extra typing. Instead, we can introduce a new type in the hierarchy that does this for us:

```

template <class Derived>
struct IShapeAcceptor : IShape {
    void accept(IShapeVisitor& visitor) const override {
        // visit with our exact type
        visitor.visit(*static_cast<Derived const*>(this));
    }
};

```

And now, each shape simply needs to inherit from the acceptor:

```

struct Circle : IShapeAcceptor<Circle>
{
    Circle(const Point& center, double radius) : center(center), radius(radius) {}
    Point center;
    double radius;
};

struct Square : IShapeAcceptor<Square>
{
    Square(const Point& topLeft, double sideLength) : topLeft(topLeft), sideLength(sideLength) {}
    Point topLeft;
    double sideLength;
};

```

No duplicate code necessary.

第80章：线程

参数	详细信息
其他	接管其他的所有权，其他不再拥有该线程
函数	在单独线程中调用的函数
参数	函数的参数 Arguments for func

第80.1节：创建 std::thread

在 C++ 中，线程是通过 `std::thread` 类创建的。线程是一个独立的执行流；它类似于让一个助手执行一项任务，而你同时执行另一项任务。当线程中的所有代码执行完毕后，它会终止。创建线程时，需要传入要在线程上执行的内容。你可以传入的几种内容有：

- 普通函数
- 成员函数
- 函数对象
- Lambda 表达式

普通函数示例——在单独的线程上执行一个函数（实时示例）：

```
#include <iostream>
#include <thread>

void foo(int a)
{
    std::cout << a << ';'

int main()
{
    // 创建并执行线程
    std::thread thread(foo, 10); // foo 是要执行的函数，10 是传递给它的
                                // 参数

    // 继续执行；线程在单独执行

    // 等待线程完成；我们在这里等待直到完成
    thread.join();

    return 0;
}
```

成员函数示例 - 在单独线程上执行成员函数（实时示例）：

```
#include <iostream>
#include <thread>

class Bar
{
public:
    void foo(int a)
    {
        std::cout << a << ';'
    }
};
```

Chapter 80: Threading

Parameter	Details
other	Takes ownership of other, other doesn't own the thread anymore
func	Function to call in a separate thread
args	Arguments for func

Section 80.1: Creating a std::thread

In C++, threads are created using the `std::thread` class. A thread is a separate flow of execution; it is analogous to having a helper perform one task while you simultaneously perform another. When all the code in the thread is executed, it *terminates*. When creating a thread, you need to pass something to be executed on it. A few things that you can pass to a thread:

- Free functions
- Member functions
- Functor objects
- Lambda expressions

Free function example - executes a function on a separate thread ([Live Example](#)):

```
#include <iostream>
#include <thread>

void foo(int a)
{
    std::cout << a << '\n';
}

int main()
{
    // Create and execute the thread
    std::thread thread(foo, 10); // foo is the function to execute, 10 is the
                                // argument to pass to it

    // Keep going; the thread is executed separately

    // Wait for the thread to finish; we stay here until it is done
    thread.join();

    return 0;
}
```

Member function example - executes a member function on a separate thread ([Live Example](#)):

```
#include <iostream>
#include <thread>

class Bar
{
public:
    void foo(int a)
    {
        std::cout << a << '\n';
    }
};
```

```

int main()
{
    Bar bar;

    // 创建并执行线程
    std::thread thread(&Bar::foo, &bar, 10); // 向成员函数传递10

    // 成员函数将在单独线程中执行

    // 等待线程完成，这是一个阻塞操作
    thread.join();

    return 0;
}

```

函数对象示例（实时示例）：_____

```

#include <iostream>
#include <thread>

class Bar
{
public:
    void operator()(int a)
    {
        std::cout << a << ";"
    }
};

int main()
{
    Bar bar;

    // 创建并执行线程
    std::thread 线程(bar, 10); // 向函数对象传递10

    // 函数对象将在单独的线程中执行

    // 等待线程结束，这是一个阻塞操作
    线程.join();

    return 0;
}

```

Lambda表达式示例（实时示例）：_____

```

#include <iostream>
#include <thread>

int main()
{
    auto lambda = [] (int a) { std::cout << a << "; };

    // 创建并执行线程
    std::thread 线程(lambda, 10); // 向lambda表达式传递10

    // lambda表达式将在单独的线程中执行

    // 等待线程结束，这是一个阻塞操作
    线程.join();
}

```

```

int main()
{
    Bar bar;

    // Create and execute the thread
    std::thread thread(&Bar::foo, &bar, 10); // Pass 10 to member function

    // The member function will be executed in a separate thread

    // Wait for the thread to finish, this is a blocking operation
    thread.join();

    return 0;
}

```

Functor object example (Live Example):

```

#include <iostream>
#include <thread>

class Bar
{
public:
    void operator()(int a)
    {
        std::cout << a << '\n';
    }
};

int main()
{
    Bar bar;

    // Create and execute the thread
    std::thread thread(bar, 10); // Pass 10 to functor object

    // The functor object will be executed in a separate thread

    // Wait for the thread to finish, this is a blocking operation
    thread.join();

    return 0;
}

```

Lambda expression example (Live Example):

```

#include <iostream>
#include <thread>

int main()
{
    auto lambda = [] (int a) { std::cout << a << '\n'; };

    // Create and execute the thread
    std::thread thread(lambda, 10); // Pass 10 to the lambda expression

    // The lambda expression will be executed in a separate thread

    // Wait for the thread to finish, this is a blocking operation
    thread.join();
}

```

```
    return 0;  
}
```

第80.2节：向线程传递引用

不能直接向线程传递引用（或const引用），因为std::thread会复制/移动它们。
相反，使用std::reference_wrapper：

```
void foo(int& b)  
{  
    b = 10;  
}  
  
int a = 1;  
std::thread thread{ foo, std::ref(a) }; // 'a' 现在确实作为引用传递  
  
thread.join();  
std::cout << a << "\n"; // 输出 10
```

```
void bar(const ComplexObject& co)  
{  
    co.doCalculations();  
}  
  
ComplexObject object;  
std::thread thread{ bar, std::cref(object) }; // 'object' 作为 const& 传递  
  
thread.join();  
std::cout << object.getResult() << "\n"; // 输出结果
```

第80.3节：使用std::async代替std::thread

std::async 也能创建线程。与 std::thread 相比，它被认为功能较弱，但在你只想异步运行一个函数时更易于使用。

异步调用函数

```
#include <future>  
#include <iostream>  
  
unsigned int square(unsigned int i){  
    return i*i;  
}  
  
int main() {  
    auto f = std::async(std::launch::async, square, 8);  
    std::cout << "平方函数正在运行"; // 在平方函数运行时做些事情  
    std::cout << "结果是 " << f.get() << "\n"; // 获取平方  
    // 函数的结果}
```

常见陷阱

- std::async 返回一个 std::future，里面保存着函数将计算的返回值。当该 future 被销毁时，它会等待线程完成，从而使你的代码实际上变成单线程。
当你不需要返回值时，这一点很容易被忽视：

```
    return 0;  
}
```

Section 80.2: Passing a reference to a thread

You cannot pass a reference (or `const` reference) directly to a thread because `std::thread` will copy/move them.
Instead, use `std::reference_wrapper`:

```
void foo(int& b)  
{  
    b = 10;  
}  
  
int a = 1;  
std::thread thread{ foo, std::ref(a) }; // 'a' is now really passed as reference  
  
thread.join();  
std::cout << a << '\n'; // Outputs 10
```

```
void bar(const ComplexObject& co)  
{  
    co.doCalculations();  
}  
  
ComplexObject object;  
std::thread thread{ bar, std::cref(object) }; // 'object' is passed as const&  
  
thread.join();  
std::cout << object.getResult() << '\n'; // Outputs the result
```

Section 80.3: Using std::async instead of std::thread

`std::async` is also able to make threads. Compared to `std::thread` it is considered less powerful but easier to use when you just want to run a function asynchronously.

Asynchronously calling a function

```
#include <future>  
#include <iostream>  
  
unsigned int square(unsigned int i){  
    return i*i;  
}  
  
int main() {  
    auto f = std::async(std::launch::async, square, 8);  
    std::cout << "square currently running\n"; // do something while square is running  
    std::cout << "result is " << f.get() << '\n'; // getting the result from square  
}
```

Common Pitfalls

- `std::async` returns a `std::future` that holds the return value that will be calculated by the function. When that future gets destroyed it waits until the thread completes, making your code effectively single threaded. This is easily overlooked when you don't need the return value:

```
std::async(std::launch::async, square, 5);
//线程在此时已经完成，因为返回的 future 已被销毁
```

- std::async 在没有启动策略的情况下工作，因此 std::async(square, 5); 可以编译。当你这样做时，系统可以决定是否创建线程。其想法是系统会选择创建线程，除非它已经运行的线程数超过了它能高效运行的数量。不幸的是，实际实现中通常在这种情况下根本不会创建线程，因此你需要用 std::launch::async 来覆盖这种行为，强制系统创建线程。

- 注意竞态条件。

关于 Futures 和 Promises 的 async 更多内容

第80.4节：基本同步

线程同步可以通过互斥锁等同步原语来实现。标准库提供了几种互斥锁类型，但最简单的是 std::mutex。要锁定互斥锁，你需要构造一个锁对象。最简单的锁类型是 std::lock_guard：

```
std::mutex m;
void worker() {
    std::lock_guard<std::mutex> guard(m); // 获取互斥锁
    // 这里是同步代码
} // 当 guard 超出作用域时，互斥锁会自动释放
```

使用 std::lock_guard 时，互斥锁在锁对象的整个生命周期内被锁定。如果需要手动控制锁定区域，请改用 std::unique_lock：

```
std::mutex m;
void worker() {
    // 默认情况下，从互斥量构造 unique_lock 会锁定该互斥量
    // 通过传递 std::defer_lock 作为第二个参数，我们
    // 可以构造一个未锁定状态的保护对象，
    // 并在之后手动加锁。
    std::unique_lock<std::mutex> guard(m, std::defer_lock);
    // 互斥量尚未被锁定！
    guard.lock();
    // 临界区
    guard.unlock();
    // 互斥量再次被释放
}
```

更多线程同步结构

第80.5节：创建一个简单的线程池

C++11 的线程原语仍然相对底层。它们可以用来编写更高级的结构，比如线程池：

版本 ≥ C++14

```
struct 任务 {
    // 互斥量、条件变量和双端队列组成了一个
    // 线程安全的触发任务队列：
    std::mutex m;
    std::condition_variable v;
    // 注意 packaged_task<void> 可以存储 packaged_task<R>：
```

```
std::async(std::launch::async, square, 5);
//thread already completed at this point, because the returning future got destroyed
```

- std::async works without a launch policy, so std::async(square, 5); compiles. When you do that the system gets to decide if it wants to create a thread or not. The idea was that the system chooses to make a thread unless it is already running more threads than it can run efficiently. Unfortunately implementations commonly just choose not to create a thread in that situation, ever, so you need to override that behavior with std::launch::async which forces the system to create a thread.
- Beware of race conditions.

More on async on Futures and Promises

Section 80.4: Basic Synchronization

Thread synchronization can be accomplished using mutexes, among other synchronization primitives. There are several mutex types provided by the standard library, but the simplest is std::mutex. To lock a mutex, you construct a lock on it. The simplest lock type is std::lock_guard:

```
std::mutex m;
void worker() {
    std::lock_guard<std::mutex> guard(m); // Acquires a lock on the mutex
    // Synchronized code here
} // the mutex is automatically released when guard goes out of scope
```

With std::lock_guard the mutex is locked for the whole lifetime of the lock object. In cases where you need to manually control the regions for locking, use std::unique_lock instead:

```
std::mutex m;
void worker() {
    // by default, constructing a unique_lock from a mutex will lock the mutex
    // by passing the std::defer_lock as a second argument, we
    // can construct the guard in an unlocked state instead and
    // manually lock later.
    std::unique_lock<std::mutex> guard(m, std::defer_lock);
    // the mutex is not locked yet!
    guard.lock();
    // critical section
    guard.unlock();
    // mutex is again released
}
```

More Thread synchronization structures

Section 80.5: Create a simple thread pool

C++11 threading primitives are still relatively low level. They can be used to write a higher level construct, like a thread pool:

Version ≥ C++14

```
struct tasks {
    // the mutex, condition variable and deque form a single
    // thread-safe triggered queue of tasks:
    std::mutex m;
    std::condition_variable v;
    // note that a packaged_task<void> can store a packaged_task<R>:
```

```

std::deque<std::packaged_task<void()>> work;

// 这保存了表示工作线程完成的 futures：
std::vector<std::future<void()>> finished;

// queue( lambda ) 将 lambda 加入线程任务队列// 并返回一个 future，类型为 lambda
// 的返回类型，方便获取结果。
template<class F, class R=std::result_of_t<F&()>>
std::future<R> queue(F&& f) {
    // 将函数对象封装成 packaged_task，分离执行和返回值：
    std::packaged_task<R()> p(std::forward<F>(f));

    auto r=p.get_future(); // 在我们交出任务之前获取返回值
    {
        std::unique_lock<std::mutex> l(m);      work.em
        place_back(std::move(p)); // 将任务<R()>存储为任务<void()>
    }

    v.notify_one(); // 唤醒一个线程来处理任务

    return r; // 返回任务的future结果
}

// 在线程池中启动N个线程。
void start(std::size_t N=1){
    for (std::size_t i = 0; i < N; ++i)
    {
        // 每个线程都是一个std::async，运行this->thread_task()：
        finished.push_back(
            std::async(
                std::launch::async,
                [this]{ thread_task(); }
            )
        );
    }
}

// abort()取消所有未开始的任务，并通知所有工作线程
// 停止运行，并等待它们完成。
void abort() {
    cancel_pending();
    finish();
}

// cancel_pending() 仅取消所有未开始的任务：
void cancel_pending() {
    std::unique_lock<std::mutex> l(m);
    work.clear();
}

// finish 为每个线程排队一个“停止线程”的消息，然后等待它们完成：
void finish() {
    {
        std::unique_lock<std::mutex> l(m);
        for(auto&&unused:finished){
            work.push_back({});
        }
    }
    v.notify_all();
    finished.clear();
}
~tasks() {
    finish();
}

```

```

std::deque<std::packaged_task<void()>> work;

// this holds futures representing the worker threads being done:
std::vector<std::future<void()>> finished;

// queue( lambda ) will enqueue the lambda into the tasks for the threads
// to use. A future of the type the lambda returns is given to let you get
// the result out.
template<class F, class R=std::result_of_t<F&()>>
std::future<R> queue(F&& f) {
    // wrap the function object into a packaged task, splitting
    // execution from the return value:
    std::packaged_task<R()> p(std::forward<F>(f));

    auto r=p.get_future(); // get the return value before we hand off the task
    {
        std::unique_lock<std::mutex> l(m);
        work.emplace_back(std::move(p)); // store the task<R()> as a task<void()>
    }
    v.notify_one(); // wake a thread to work on the task

    return r; // return the future result of the task
}

// start N threads in the thread pool.
void start(std::size_t N=1){
    for (std::size_t i = 0; i < N; ++i)
    {
        // each thread is a std::async running this->thread_task():
        finished.push_back(
            std::async(
                std::launch::async,
                [this]{ thread_task(); }
            )
        );
    }
}

// abort() cancels all non-started tasks, and tells every working thread
// stop running, and waits for them to finish up.
void abort() {
    cancel_pending();
    finish();
}

// cancel_pending() merely cancels all non-started tasks:
void cancel_pending() {
    std::unique_lock<std::mutex> l(m);
    work.clear();
}

// finish enques a "stop the thread" message for every thread, then waits for them:
void finish() {
    {
        std::unique_lock<std::mutex> l(m);
        for(auto&&unused:finished){
            work.push_back({});
        }
    }
    v.notify_all();
    finished.clear();
}
~tasks() {
    finish();
}

```

```

private:
    // 工作线程执行的工作内容：
    void thread_task() {
        while(true){
            // 从队列中弹出一个任务：
            std::packaged_task<void()> f;
            {
                // 常见的线程安全队列代码：
                std::unique_lock<std::mutex> l(m);
                if (work.empty()){
                    v.wait(l,[&]{return !work.empty();});
                }
                f = std::move(work.front());
                work.pop_front();
            }
            // 如果任务无效，表示我们被要求中止：
            if (!f.valid()) return;
            // 否则，执行任务：
            f();
        }
    }
};

tasks.queue( []{ return "hello world"s; } ) 返回一个 std::future<std::string>, 当 tasks 对象开始执行时，该 future 会被赋值为 hello world。

```

你通过运行`tasks.start(10)`（这会启动10个线程）来创建线程。

使用`packaged_task<void()>`仅仅是因为没有存储只能移动类型的类型擦除`std::function`等价物。编写一个自定义的可能比使用`packaged_task<void()>`更快。

实时示例。

版本 = C++11

在C++11中，用`typename result_of<blah>::type`替换`result_of_t<blah>`。

关于互斥锁的更多内容。

第80.6节：确保线程总是被join

当调用`std::thread`的析构函数时，必须已经调用了`join()`或`detach()`。如果线程没有被`join`或`detach`，则默认会调用`std::terminate`。使用RAII，这通常很容易实现：

```

class thread_joiner
{
public:

    thread_joiner(std::thread t)
        : t_(std::move(t))
    {}

    ~thread_joiner()
    {
        if(t_.joinable()) {
            t_.join();
        }
    }
};

```

```

private:
    // the work that a worker thread does:
    void thread_task() {
        while(true){
            // pop a task off the queue:
            std::packaged_task<void()> f;
            {
                // usual thread-safe queue code:
                std::unique_lock<std::mutex> l(m);
                if (work.empty()){
                    v.wait(l,[&]{return !work.empty();});
                }
                f = std::move(work.front());
                work.pop_front();
            }
            // if the task is invalid, it means we are asked to abort:
            if (!f.valid()) return;
            // otherwise, run the task:
            f();
        }
    }
};

tasks.queue( []{ return "hello world"s; } ) returns a std::future<std::string>, which when the tasks object gets around to running it is populated with hello world.

```

You create threads by running `tasks.start(10)` (which starts 10 threads).

The use of `packaged_task<void()>` is merely because there is no type-erased `std::function` equivalent that stores move-only types. Writing a custom one of those would probably be faster than using `packaged_task<void()>`.

Live example.

Version = C++11

In C++11, replace `result_of_t<blah>` with `typename result_of<blah>::type`.

More on Mutexes.

Section 80.6: Ensuring a thread is always joined

When the destructor for `std::thread` is invoked, a call to either `join()` or `detach()` **must** have been made. If a thread has not been joined or detached, then by default `std::terminate` will be called. Using RAII, this is generally simple enough to accomplish:

```

class thread_joiner
{
public:

    thread_joiner(std::thread t)
        : t_(std::move(t))
    {}

    ~thread_joiner()
    {
        if(t_.joinable()) {
            t_.join();
        }
    }
};

```

```
}
```

```
private:
```

```
std::thread t_;
```

然后这样使用：

```
void perform_work()
{
    // 执行一些工作
}

void t()
{
    thread_joiner j{std::thread(perform_work)};
    // 在线程运行时执行其他计算
} // 线程在此处自动加入
```

这也提供了异常安全性；如果我们正常创建线程并且在 t() 中执行其他计算时抛出异常，join() 将永远不会被调用，我们的进程将被终止。

第80.7节：当前线程的操作

`std::this_thread` 是一个命名空间，包含从函数调用处对当前线程执行有趣操作的函数。

函数	描述
<code>get_id</code>	返回线程的ID
<code>sleep_for</code>	休眠指定的时间
<code>sleep_until</code>	休眠直到指定时间
<code>yield</code>	重新调度正在运行的线程，给予其他线程优先权

使用 `std::this_thread::get_id` 获取当前线程ID：

```
void foo()
{
    // 打印此线程的ID
    std::cout << std::this_thread::get_id() << ";"

    std::thread thread{ foo };
    thread.join(); // "线程" 的 ID 现已打印，应该类似于 12556

    foo(); // 主线程的 ID 已打印，应该类似于 2420
```

使用 `std::this_thread::sleep_for` 睡眠 3 秒：

```
void foo()
{
    std::this_thread::sleep_for(std::chrono::seconds(3));
}
```

```
}
```

```
private:
```

```
std::thread t_;
```

This is then used like so:

```
void perform_work()
{
    // Perform some work
}

void t()
{
    thread_joiner j{std::thread(perform_work)};
    // Do some other calculations while thread is running
} // Thread is automatically joined here
```

This also provides exception safety；if we had created our thread normally and the work done in t() performing other calculations had thrown an exception, join() would never have been called on our thread and our process would have been terminated.

Section 80.7: Operations on the current thread

`std::this_thread` 是一个 [namespace](#)，包含从函数调用处对当前线程执行有趣操作的函数。它是由谁调用的。

Function	Description
<code>get_id</code>	Returns the id of the thread
<code>sleep_for</code>	Sleeps for a specified amount of time
<code>sleep_until</code>	Sleeps until a specific time
<code>yield</code>	Reschedule running threads, giving other threads priority

Getting the current threads id using `std::this_thread::get_id`:

```
void foo()
{
    // Print this threads id
    std::cout << std::this_thread::get_id() << '\n';

    std::thread thread{ foo };
    thread.join(); // 'threads' id has now been printed, should be something like 12556

    foo(); // The id of the main thread is printed, should be something like 2420
```

Sleeping for 3 seconds using `std::this_thread::sleep_for`:

```
void foo()
{
    std::this_thread::sleep_for(std::chrono::seconds(3));
}
```

```
std::thread thread{ foo };
foo.join();

std::cout << "等待了3秒！";
```

使用`std::this_thread::sleep_until`睡眠直到未来3小时：

```
void foo()
{
    std::this_thread::sleep_until(std::chrono::system_clock::now() + std::chrono::hours(3));
}

std::thread 线程{ foo };
线程.join();

std::cout << "我们现在位于线程调用后3小时";
```

使用`std::this_thread::yield`让其他线程优先：

```
void foo(int a)
{
    for (int i = 0; i < a; ++i)
        std::this_thread::yield(); //现在其他线程优先，因为这个线程
                                //没有做任何重要的事情

    std::cout << "你好，世界！";}

std::thread 线程{ foo, 10 };
线程.join();
```

第80.8节：使用条件变量

条件变量是一种与互斥锁配合使用的原语，用于协调线程之间的通信。虽然它既不是唯一的也不是最高效的实现方式，但对于熟悉该模式的人来说，它可能是最简单的方式之一。

等待一个`std::condition_variable`，使用一个`std::unique_lock<std::mutex>`。这允许代码在决定是否继续获取之前安全地检查共享状态。

下面是一个生产者-消费者示例，使用了`std::thread`、`std::condition_variable`、`std::mutex`以及其他一些内容，使程序更有趣。

```
#include <condition_variable>
#include <cstdint>
#include <iostream>
#include <mutex>
#include <queue>
#include <random>
#include <thread>

int main()
{
    std::condition_variable cond;
    std::mutex mtx;
```

```
std::thread thread{ foo };
foo.join();

std::cout << "Waited for 3 seconds!\n";
```

Sleeping until 3 hours in the future using `std::this_thread::sleep_until`:

```
void foo()
{
    std::this_thread::sleep_until(std::chrono::system_clock::now() + std::chrono::hours(3));
}

std::thread thread{ foo };
thread.join();

std::cout << "We are now located 3 hours after the thread has been called\n";
```

Letting other threads take priority using `std::this_thread::yield`:

```
void foo(int a)
{
    for (int i = 0; i < a; ++i)
        std::this_thread::yield(); //Now other threads take priority, because this thread
                                //isn't doing anything important

    std::cout << "Hello World!\n";
}

std::thread thread{ foo, 10 };
thread.join();
```

Section 80.8: Using Condition Variables

A condition variable is a primitive used in conjunction with a mutex to orchestrate communication between threads. While it is neither the exclusive or most efficient way to accomplish this, it can be among the simplest to those familiar with the pattern.

One waits on a `std::condition_variable` with a `std::unique_lock<std::mutex>`. This allows the code to safely examine shared state before deciding whether or not to proceed with acquisition.

Below is a producer-consumer sketch that uses `std::thread`, `std::condition_variable`, `std::mutex`, and a few others to make things interesting.

```
#include <condition_variable>
#include <cstdint>
#include <iostream>
#include <mutex>
#include <queue>
#include <random>
#include <thread>

int main()
{
    std::condition_variable cond;
    std::mutex mtx;
```

```

std::queue<int> intq;
bool stopped = false;

std::thread producer{[&]()
{
    // 准备一个随机数生成器。
    // 我们的生产者将简单地向 intq 推送随机数。
    //
    std::default_random_engine gen{};
    std::uniform_int_distribution<int> dist{};

    std::size_t count = 4006;
    while(count--)
    {
        // 在更改之前始终加锁
        // 由互斥锁和
        // 条件变量（即“condvar”）保护的状态。
        std::lock_guard<std::mutex> L{mtx};

        // 向队列中推入一个随机整数
        intq.push(dist(gen));

        // 通知消费者队列中有整数
        cond.notify_one();
    }

    // 全部完成。
    // 获取锁，设置停止标志，
    // 然后通知消费者。
    std::lock_guard<std::mutex> L{mtx};

    std::cout << "Producer is done!" << std::endl;

    stopped = true;
    cond.notify_one();
};

std::thread consumer{[&]()
{
    do{
        std::unique_lock<std::mutex> L{mtx};
        cond.wait(L,[&]())
        {
            // 仅在
            // 已停止或队列
            // 非空时获取锁
            return stopped || !intq.empty();
        });

        // 此处我们拥有互斥锁；弹出队列
        // 直到队列为空。

        while( ! intq.empty() )
        {
            const auto val = intq.front();
            intq.pop();

            std::cout << "消费者弹出: " << val << std::endl;
        }
    }

    if(stopped){
        // 生产者已发出停止信号
    }
}}

```

```

std::queue<int> intq;
bool stopped = false;

std::thread producer{[&]()
{
    // Prepare a random number generator.
    // Our producer will simply push random numbers to intq.
    //
    std::default_random_engine gen{};
    std::uniform_int_distribution<int> dist{};

    std::size_t count = 4006;
    while(count--)
    {
        // Always lock before changing
        // state guarded by a mutex and
        // condition_variable (a.k.a. "condvar").
        std::lock_guard<std::mutex> L{mtx};

        // Push a random int into the queue
        intq.push(dist(gen));

        // Tell the consumer it has an int
        cond.notify_one();
    }

    // All done.
    // Acquire the lock, set the stopped flag,
    // then inform the consumer.
    std::lock_guard<std::mutex> L{mtx};

    std::cout << "Producer is done!" << std::endl;

    stopped = true;
    cond.notify_one();
};

std::thread consumer{[&]()
{
    do{
        std::unique_lock<std::mutex> L{mtx};
        cond.wait(L,[&]())
        {
            // Acquire the lock only if
            // we've stopped or the queue
            // isn't empty
            return stopped || !intq.empty();
        });

        // We own the mutex here; pop the queue
        // until it empties out.

        while( ! intq.empty() )
        {
            const auto val = intq.front();
            intq.pop();

            std::cout << "Consumer popped: " << val << std::endl;
        }
    }

    if(stopped){
        // producer has signaled a stop
    }
}}

```

```

    std::cout << "消费者已完成！" << std::endl;
    break;
}

}while(true);
};

consumer.join();
producer.join();

std::cout << "示例完成！" << std::endl;

return 0;
}

```

第80.9节：线程操作

当你启动一个线程时，它会一直执行直到完成。

通常，在某个时刻，你需要（可能线程已经完成）等待线程结束，因为你想使用结果，例如。

```

int n;
std::thread thread{ calculateSomething, std::ref(n) };

//执行其他操作

//我们现在需要'n'！
//等待线程完成——如果它还没有完成
thread.join();

//现在'n'拥有在独立线程中完成的计算结果std::cout << n << ";

```

你也可以detach线程，让它自由执行：

```

std::thread thread{ doSomething };

//分离线程，我们不再需要它（无论出于何种原因）
thread.detach();

//线程将在完成时终止，或者当主线程返回时终止

```

第80.10节：线程局部存储

线程局部存储可以使用`thread_local`关键字创建。用`thread_local`说明符声明的变量被称为具有**线程存储持续时间**。

- 程序中的每个线程都有自己的一份线程局部变量的副本。
- 具有函数（局部）作用域的线程局部变量将在控制流首次经过其定义时初始化。这样的变量隐式为静态，除非声明为`extern`。
- 具有命名空间或类（非局部）作用域的线程局部变量将在线程启动时初始化。
- 线程局部变量在线程终止时被销毁。
- 类的成员只有在静态时才能是线程局部的。因此，每个线程将有该变量的一份副本，而不是每个（线程，实例）对一份副本。

```

        std::cout << "Consumer is done!" << std::endl;
        break;
    }

}while(true);
};

consumer.join();
producer.join();

std::cout << "Example Completed!" << std::endl;

return 0;
}

```

Section 80.9: Thread operations

When you start a thread, it will execute until it is finished.

Often, at some point, you need to (possibly - the thread may already be done) wait for the thread to finish, because you want to use the result for example.

```

int n;
std::thread thread{ calculateSomething, std::ref(n) };

//Doing some other stuff

//We need 'n' now!
//Wait for the thread to finish - if it is not already done
thread.join();

//Now 'n' has the result of the calculation done in the separate thread
std::cout << n << '\n';

```

You can also detach the thread, letting it execute freely:

```

std::thread thread{ doSomething };

//Detaching the thread, we don't need it anymore (for whatever reason)
thread.detach();

//The thread will terminate when it is done, or when the main thread returns

```

Section 80.10: Thread-local storage

Thread-local storage can be created using the `thread_local` keyword. A variable declared with the `thread_local` specifier is said to have **thread storage duration**.

- Each thread in a program has its own copy of each thread-local variable.
- A thread-local variable with function (local) scope will be initialized the first time control passes through its definition. Such a variable is implicitly static, unless declared `extern`.
- A thread-local variable with namespace or class (non-local) scope will be initialized as part of thread startup.
- Thread-local variables are destroyed upon thread termination.
- A member of a class can only be thread-local if it is static. There will therefore be one copy of that variable per thread, rather than one copy per (thread, instance) pair.

示例：

```
void debug_counter() {
    thread_local int count = 0;
    Logger::log("该函数已被此线程调用 %d 次", ++count);
}
```

第80.11节：重新分配线程对象

我们可以创建空的线程对象，并在之后为它们分配任务。

如果我们将一个线程对象赋值给另一个处于活动状态、可joinable的线程，`std::terminate`将在替换线程之前自动被调用。

```
#include <thread>

void foo()
{
    std::this_thread::sleep_for(std::chrono::seconds(3));
}

// 创建100个不执行任何操作的线程对象
std::thread executors[100];

// 一些代码

// 我现在想创建一些线程

for (int i = 0; i < 100; i++)
{
    // 如果该对象没有分配线程
    if (!executors[i].joinable())
        executors[i] = std::thread(foo);
}
```

Example:

```
void debug_counter() {
    thread_local int count = 0;
    Logger::log("This function has been called %d times by this thread", ++count);
}
```

Section 80.11: Reassigning thread objects

We can create empty thread objects and assign work to them later.

If we assign a thread object to another active, joinable thread, `std::terminate` will automatically be called before the thread is replaced.

```
#include <thread>

void foo()
{
    std::this_thread::sleep_for(std::chrono::seconds(3));
}

// Create 100 thread objects that do nothing
std::thread executors[100];

// Some code

// I want to create some threads now

for (int i = 0; i < 100; i++)
{
    // If this object doesn't have a thread assigned
    if (!executors[i].joinable())
        executors[i] = std::thread(foo);
}
```

第81章：线程同步结构

当线程相互交互时，可能需要一些同步技术。在本主题中，您可以找到标准库提供的不同结构来解决这些问题。

第81.1节：std::condition_variable_any, std::cv_status

std::condition_variable的一个泛化，std::condition_variable_any可以与任何类型的BasicLockable结构一起使用。

std::cv_status作为条件变量的返回状态，有两种可能的返回代码：

- std::cv_status::no_timeout：没有超时，条件变量被通知
- std::cv_status::timeout：条件变量超时

第81.2节：std::shared_lock

shared_lock可以与unique_lock配合使用，以允许多个读者和独占写者。

```
#include <unordered_map>
#include <mutex>
#include <shared_mutex>
#include <thread>
#include <string>
#include <iostream>

class PhoneBook {
public:
    string getPhoneNo( const std::string & name )
    {
        shared_lock<shared_timed_mutex> r(_protect);
        auto it = _phonebook.find( name );
        if ( it == _phonebook.end() )
            return (*it).second;
        return "";
    }
    void addPhoneNo ( const std::string & name, const std::string & phone )
    {
        unique_lock<shared_timed_mutex> w(_protect);
        _phonebook[name] = phone;
    }

    shared_timed_mutex _protect;
    unordered_map<string,string> _phonebook;
};
```

第81.3节：std::call_once, std::once_flag

std::call_once 确保函数由竞争线程仅执行一次。如果无法完成任务，则抛出 std::system_error 异常。

与 std::once_flag 一起使用。

```
#include <mutex>
```

Chapter 81: Thread synchronization structures

Working with threads might need some synchronization techniques if the threads interact. In this topic, you can find the different structures which are provided by the standard library to solve these issues.

Section 81.1: std::condition_variable_any, std::cv_status

A generalization of std::condition_variable, std::condition_variable_any works with any type of BasicLockable structure.

std::cv_status as a return status for a condition variable has two possible return codes:

- std::cv_status::no_timeout: There was no timeout, condition variable was notified
- std::cv_status::no_timeout: Condition variable timed out

Section 81.2: std::shared_lock

A shared_lock can be used in conjunction with a unique lock to allow multiple readers and exclusive writers.

```
#include <unordered_map>
#include <mutex>
#include <shared_mutex>
#include <thread>
#include <string>
#include <iostream>

class PhoneBook {
public:
    string getPhoneNo( const std::string & name )
    {
        shared_lock<shared_timed_mutex> r(_protect);
        auto it = _phonebook.find( name );
        if ( it == _phonebook.end() )
            return (*it).second;
        return "";
    }
    void addPhoneNo ( const std::string & name, const std::string & phone )
    {
        unique_lock<shared_timed_mutex> w(_protect);
        _phonebook[name] = phone;
    }

    shared_timed_mutex _protect;
    unordered_map<string,string> _phonebook;
};
```

Section 81.3: std::call_once, std::once_flag

std::call_once ensures execution of a function exactly once by competing threads. It throws std::system_error in case it cannot complete its task.

Used in conjunction with std::once_flag.

```
#include <mutex>
```

```
#include <iostream>

std::once_flag flag;
void do_something(){
    std::call_once(flag, [](){std::cout << "只发生一次" << std::endl;});

    std::cout << "每次都会发生" << std::endl;
}
```

第81.4节：对象锁定以实现高效访问

通常，当你对对象执行多个操作时，想要锁定整个对象。例如，如果你需要使用迭代器检查或修改对象。每当你需要调用多个成员函数时，通常锁定整个对象比单独锁定各个成员函数更高效。

例如：

```
class text_buffer
{
    // 为了可读性/可维护性
    using mutex_type = std::shared_timed_mutex;
    using reading_lock = std::shared_lock<mutex_type>;
    using updates_lock = std::unique_lock<mutex_type>;

    public:
        // 这返回一个作用域锁，允许多个
        // 读者同时共享，同时排除任何写者
        [[nodiscard]]
        reading_lock lock_for_reading() const { return reading_lock(mtx); }

        // 这返回一个作用域锁，排他给一个
        // 写者，阻止任何读者
        [[nodiscard]]
        updates_lock lock_for_updates() { return updates_lock(mtx); }

        char* data() { return buf; }
        char const* data() const { return buf; }

        char* begin() { return buf; }
        char const* begin() const { return buf; }

        char* end() { return buf + sizeof(buf); }
        char const* end() const { return buf + sizeof(buf); }

        std::size_t size() const { return sizeof(buf); }

    private:
        char buf[1024];
        mutable mutex_type mtx; // mutable allows const objects to be locked
};
```

计算校验和时，对象被锁定以供读取，允许其他线程同时读取该对象。

```
std::size_t checksum(text_buffer const& buf)
{
    std::size_t sum = 0xA44944A4;

    // lock the object for reading
```

```
#include <iostream>

std::once_flag flag;
void do_something(){
    std::call_once(flag, [](){std::cout << "Happens once" << std::endl;});

    std::cout << "Happens every time" << std::endl;
}
```

Section 81.4: Object locking for efficient access

Often you want to lock the entire object while you perform multiple operations on it. For example, if you need to examine or modify the object using *iterators*. Whenever you need to call multiple member functions it is generally more efficient to lock the whole object rather than individual member functions.

For example:

```
class text_buffer
{
    // for readability/maintainability
    using mutex_type = std::shared_timed_mutex;
    using reading_lock = std::shared_lock<mutex_type>;
    using updates_lock = std::unique_lock<mutex_type>;

    public:
        // This returns a scoped lock that can be shared by multiple
        // readers at the same time while excluding any writers
        [[nodiscard]]
        reading_lock lock_for_reading() const { return reading_lock(mtx); }

        // This returns a scoped lock that is exclusive to one
        // writer preventing any readers
        [[nodiscard]]
        updates_lock lock_for_updates() { return updates_lock(mtx); }

        char* data() { return buf; }
        char const* data() const { return buf; }

        char* begin() { return buf; }
        char const* begin() const { return buf; }

        char* end() { return buf + sizeof(buf); }
        char const* end() const { return buf + sizeof(buf); }

        std::size_t size() const { return sizeof(buf); }

    private:
        char buf[1024];
        mutable mutex_type mtx; // mutable allows const objects to be locked
};
```

When calculating a checksum the object is locked for reading, allowing other threads that want to read from the object at the same time to do so.

```
std::size_t checksum(text_buffer const& buf)
{
    std::size_t sum = 0xA44944A4;

    // lock the object for reading
```

```

auto lock = buf.lock_for_reading();

for(auto c: buf)
sum = (sum << 8) | (((unsigned char) ((sum & 0xFF000000) >> 24)) ^ c);

return sum;
}

```

清空对象会更新其内部数据，因此必须使用排他锁进行操作。

```

void clear(text_buffer& buf)
{
    auto lock = buf.lock_for_updates(); // 排他锁
    std::fill(std::begin(buf), std::end(buf), '\0');
}

```

当获取多个锁时，应注意所有线程始终以相同顺序获取锁。

```

void transfer(text_buffer const& input, text_buffer& output)
{
    auto lock1 = input.lock_for_reading();
    auto lock2 = output.lock_for_updates();

std::copy(std::begin(input), std::end(input), std::begin(output));
}

```

注意：这最好使用 `std::deferred::lock` 并调用 `std::lock` 来完成

```

auto lock = buf.lock_for_reading();

for(auto c: buf)
    sum = (sum << 8) | (((unsigned char) ((sum & 0xFF000000) >> 24)) ^ c);

return sum;
}

```

Clearing the object updates its internal data so it must be done using an excluding lock.

```

void clear(text_buffer& buf)
{
    auto lock = buf.lock_for_updates(); // exclusive lock
    std::fill(std::begin(buf), std::end(buf), '\0');
}

```

When obtaining more than one lock care should be taken to always acquire the locks in the same order for all threads.

```

void transfer(text_buffer const& input, text_buffer& output)
{
    auto lock1 = input.lock_for_reading();
    auto lock2 = output.lock_for_updates();

    std::copy(std::begin(input), std::end(input), std::begin(output));
}

```

note: This is best done using `std::deferred::lock` and calling `std::lock`

第82章：三法则、五法则与零法则

第82.1节：零法则

版本 ≥ C++11

我们可以结合五法则和RAII的原则，得到一个更简洁的接口：零法则：任何需要管理的资源都应该放在它自己的类型中。该类型必须遵循五法则，但所有该资源的使用者都不需要编写任何五个特殊成员函数，可以简单地将它们全部设为默认。

使用在三法则示例中介绍的 Person 类，我们可以为 cstrings 创建一个资源管理对象：

```
class cstring {
private:
    char* p;

public:
    ~cstring() { delete [] p; }
    cstring(cstring const& );
    cstring(cstring&& );
    cstring& operator=(cstring const& );
    cstring& operator=(cstring&& );

    /* 其他成员视情况而定 */
};
```

一旦将其分离，我们的Person类就变得简单得多：

```
class Person {
    cstring name;
    int arg;

public:
    ~Person() = default;
    Person(Person const& ) = default;
    Person(Person&& ) = default;
    Person& operator=(Person const& ) = default;
    Person& operator=(Person&& ) = default;

    /* 其他成员视情况而定 */
};
```

Person中的特殊成员甚至不需要显式声明；编译器会根据Person的内容适当地默认或删除它们。因此，以下也是零规则的一个例子。

```
struct Person {
    cstring name;
    int arg;
};
```

如果cstring是一个仅支持移动的类型，且复制构造函数/赋值运算符被delete删除，那么Person也会自动变成仅支持移动的类型。

Chapter 82: The Rule of Three, Five, And Zero

Section 82.1: Rule of Zero

Version ≥ C++11

We can combine the principles of the Rule of Five and RAII to get a much leaner interface: the Rule of Zero: any resource that needs to be managed should be in its own type. That type would have to follow the Rule of Five, but all users of that resource do not need to write *any* of the five special member functions and can simply `default` all of them.

Using the Person class introduced in the Rule of Three example, we can create a resource-managing object for cstrings:

```
class cstring {
private:
    char* p;

public:
    ~cstring() { delete [] p; }
    cstring(cstring const& );
    cstring(cstring&& );
    cstring& operator=(cstring const& );
    cstring& operator=(cstring&& );

    /* other members as appropriate */
};
```

And once this is separate, our Person class becomes far simpler:

```
class Person {
    cstring name;
    int arg;

public:
    ~Person() = default;
    Person(Person const& ) = default;
    Person(Person&& ) = default;
    Person& operator=(Person const& ) = default;
    Person& operator=(Person&& ) = default;

    /* other members as appropriate */
};
```

The special members in Person do not even need to be declared explicitly; the compiler will default or delete them appropriately, based on the contents of Person. Therefore, the following is also an example of the rule of zero.

```
struct Person {
    cstring name;
    int arg;
};
```

If `cstring` were to be a move-only type, with a `deleted` copy constructor/assignment operator, then Person would automatically be move-only as well.

第82.2节：五规则

版本 ≥ C++11

C++11引入了两个新的特殊成员函数：移动构造函数和移动赋值运算符。

出于所有希望遵循 C++03 中三法则的相同原因，通常你也希望遵循 C++11 中的五法则：如果一个类需要五个特殊成员函数中的任意一个，并且希望使用移动语义，那么它很可能需要全部五个。

但是请注意，只要遵循三法则，未遵循五法则通常不被视为错误，而是错失了优化的机会。如果在编译器通常会使用移动构造函数或移动赋值运算符的情况下没有提供，编译器将尽可能使用拷贝语义，导致由于不必要的拷贝操作而效率较低。如果某个类不需要移动语义，则无需声明移动构造函数或赋值运算符。

与三法则的示例相同：

```
类 Person
{
    char* name;
    int age;

    公共:
        // 析构函数
        ~Person() { delete [] name; }

        // 实现拷贝语义
        Person(Person const& other)
            : name(new char[strlen(other.name) + 1])
            , age(other.age)
        {
            std::strcpy(name, other.name);
        }

        Person &operator=(Person const& other)
        {
            // 使用拷贝与交换惯用法实现赋值。
            Person 复制(其他);
            交换(*this, 复制);
            返回 *this;
        }

        // 实现移动语义
        // 注意：通常最好将移动操作符标记为 noexcept//      这允许标准库在类被用
        // 作容器时进行某些优化。
}

Person(Person&& 那个) noexcept
    : 名字(nullptr)
    , 年龄(0)
{
    交换(*this, 那个);
}

Person& 操作符=(Person&& 那个) noexcept
{
```

Section 82.2: Rule of Five

Version ≥ C++11

C++11 introduces two new special member functions: the move constructor and the move assignment operator. For all the same reasons that you want to follow the Rule of Three in C++03, you usually want to follow the Rule of Five in C++11: If a class requires ONE of five special member functions, and if move semantics are desired, then it most likely requires ALL FIVE of them.

Note, however, that failing to follow the Rule of Five is usually not considered an error, but a missed optimisation opportunity, as long as the Rule of Three is still followed. If no move constructor or move assignment operator is available when the compiler would normally use one, it will instead use copy semantics if possible, resulting in a less efficient operation due to unnecessary copy operations. If move semantics aren't desired for a class, then it has no need to declare a move constructor or assignment operator.

Same example as for the Rule of Three:

```
class Person
{
    char* name;
    int age;

    public:
        // Destructor
        ~Person() { delete [] name; }

        // Implement Copy Semantics
        Person(Person const& other)
            : name(new char[strlen(other.name) + 1])
            , age(other.age)
        {
            std::strcpy(name, other.name);
        }

        Person &operator=(Person const& other)
        {
            // Use copy and swap idiom to implement assignment.
            Person copy(other);
            swap(*this, copy);
            return *this;
        }

        // Implement Move Semantics
        // Note: It is usually best to mark move operators as noexcept
        //       This allows certain optimizations in the standard library
        //       when the class is used in a container.

        Person(Person&& that) noexcept
            : name(nullptr) // Set the state so we know it is undefined
            , age(0)
        {
            swap(*this, that);
        }

        Person& operator=(Person&& that) noexcept
{
```

```

交换(*this, 那个);
    返回 *this;
}

友元函数 void 交换(Person& 左值, Person& 右值) noexcept
{
    std::swap(lhs.name, rhs.name);
    std::swap(lhs.age, rhs.age);
}
};

```

或者，复制赋值运算符和移动赋值运算符都可以被一个单一的赋值运算符替代，该运算符通过值传递实例，而不是通过引用或右值引用，以便使用复制与交换（copy-and-swap）惯用法。

```

Person& operator=(Person copy)
{
    swap(*this, copy);
    return *this;
}

```

从三法则扩展到五法则对于性能来说很重要，但在大多数情况下并非严格必要。添加复制构造函数和赋值运算符可以确保移动该类型时不会发生内存泄漏（在那种情况下，移动构造会简单地退回到复制），但会执行调用者可能未预料到的复制操作。

第82.3节：三法则

版本 ≤ c++03

三法则指出，如果一个类型需要用户自定义的复制构造函数、复制赋值运算符或析构函数中的任何一个，那么它必须拥有全部三个。

该规则的原因是，任何需要这三者中任意一个的类都管理某种资源（文件句柄、动态分配的内存等），而这三者都需要用来一致地管理该资源。复制函数处理资源在对象间的复制方式，析构函数则会根据RAII原则销毁资源。

考虑一个管理字符串资源的类型：

```

类 Person
{
    char* name;
    int age;

    公共:
    Person(char const* new_name, int new_age)
        : name(new char[std::strlen(new_name) + 1])
        , age(new_age)
    {
        std::strcpy(name, new_name);
    }

    ~Person() {
        delete [] name;
    }
};

```

```

swap(*this, that);
    return *this;
}

friend void swap(Person& lhs, Person& rhs) noexcept
{
    std::swap(lhs.name, rhs.name);
    std::swap(lhs.age, rhs.age);
}
};

```

Alternatively, both the copy and move assignment operator can be replaced with a single assignment operator, which takes an instance by value instead of reference or rvalue reference to facilitate using the copy-and-swap idiom.

```

Person& operator=(Person copy)
{
    swap(*this, copy);
    return *this;
}

```

Extending from the Rule of Three to the Rule of Five is important for performance reasons, but is not strictly necessary in most cases. Adding the copy constructor and assignment operator ensures that moving the type will not leak memory (move-constructing will simply fall back to copying in that case), but will be performing copies that the caller probably did not anticipate.

Section 82.3: Rule of Three

Version ≤ c++03

The Rule of Three states that if a type ever needs to have a user-defined copy constructor, copy assignment operator, or destructor, then it must have *all three*.

The reason for the rule is that a class which needs any of the three manages some resource (file handles, dynamically allocated memory, etc), and all three are needed to manage that resource consistently. The copy functions deal with how the resource gets copied between objects, and the destructor would destroy the resource, in accord with RAII principles.

Consider a type that manages a string resource:

```

class Person
{
    char* name;
    int age;

    公共:
    Person(char const* new_name, int new_age)
        : name(new char[std::strlen(new_name) + 1])
        , age(new_age)
    {
        std::strcpy(name, new_name);
    }

    ~Person() {
        delete [] name;
    }
};

```

由于name是在构造函数中分配的，析构函数会释放它以避免内存泄漏。但如果这样的对象被复制，会发生什么呢？

```
int main()
{
    Person p1("foo", 11);
    Person p2 = p1;
}
```

首先，p1会被构造。然后p2会从p1复制。然而，C++自动生成的拷贝构造函数会按原样复制类型的每个成员。这意味着p1.name和p2.name都指向同一个字符串。

当main结束时，析构函数将被调用。首先调用p2的析构函数；它会删除该字符串。然后调用p1的析构函数。然而，该字符串已经被删除。对已删除的内存调用delete会导致未定义行为。

为避免这种情况，有必要提供合适的拷贝构造函数。一种方法是实现引用计数系统，不同的Person实例共享相同的字符串数据。每次复制时，共享的引用计数增加。析构函数则减少引用计数，只有当计数为零时才释放内存。

或者我们可以实现值语义和深拷贝行为：

```
Person(Person const& other)
: name(new char[strlen(other.name) + 1])
, age(other.age)
{
    std::strcpy(name, other.name);
}

Person &operator=(Person const& other)
{
    // 使用复制和交换惯用法来实现赋值
    Person copy(other);
    swap(copy);           // 假设 swap() 交换 *this 和 copy 的内容
    return *this;
}
```

复制赋值运算符的实现因需要释放现有缓冲区而变得复杂。复制和交换技术创建一个临时对象，该对象持有一个新的缓冲区。交换 *this 和 copy 的内容后，原缓冲区的所有权转移给 copy。随着函数返回，copy 的销毁释放了原本属于 *this 的缓冲区。

第82.4节：自我赋值保护

编写复制赋值运算符时，能够处理自我赋值是非常重要的。也就是说，它必须允许如下操作：

```
SomeType t = ...;
t = t;
```

自我赋值通常不会以如此明显的方式发生。它通常通过各种代码系统的迂回路径发生，赋值的位置仅有两个 Person 指针或引用，却不知道它们是同一个对象。

Since name was allocated in the constructor, the destructor deallocates it to avoid leaking memory. But what happens if such an object is copied?

```
int main()
{
    Person p1("foo", 11);
    Person p2 = p1;
}
```

First, p1 will be constructed. Then p2 will be copied from p1. However, the C++-generated copy constructor will copy each component of the type as-is. Which means that p1.name and p2.name both point to the **same** string.

When main ends, destructors will be called. First p2's destructor will be called; it will delete the string. Then p1's destructor will be called. However, the string is *already deleted*. Calling `delete` on memory that was already deleted yields undefined behavior.

To avoid this, it is necessary to provide a suitable copy constructor. One approach is to implement a reference counted system, where different Person instances share the same string data. Each time a copy is performed, the shared reference count is incremented. The destructor then decrements the reference count, only releasing the memory if the count is zero.

Or we could implement value semantics and deep copying behavior:

```
Person(Person const& other)
: name(new char[strlen(other.name) + 1])
, age(other.age)
{
    std::strcpy(name, other.name);
}

Person &operator=(Person const& other)
{
    // Use copy and swap idiom to implement assignment
    Person copy(other);
    swap(copy);           // assume swap() exchanges contents of *this and copy
    return *this;
}
```

Implementation of the copy assignment operator is complicated by the need to release an existing buffer. The copy and swap technique creates a temporary object which holds a new buffer. Swapping the contents of `*this` and `copy` gives ownership to `copy` of the original buffer. Destruction of `copy`, as the function returns, releases the buffer previously owned by `*this`.

Section 82.4: Self-assignment Protection

When writing a copy assignment operator, it is *very* important that it be able to work in the event of self-assignment. That is, it has to allow this:

```
SomeType t = ...;
t = t;
```

Self-assignment usually doesn't happen in such an obvious way. It typically happens via a circuitous route through various code systems, where the location of the assignment simply has two Person pointers or references and has no idea that they are the same object.

你编写的任何复制赋值运算符都必须能够考虑到这一点。

典型的做法是将所有赋值逻辑包裹在如下条件中：

```
SomeType &operator=(const SomeType &other)
{
    if(this != &other)
    {
        //执行赋值逻辑。
    }
    return *this;
}
```

注意：考虑自我赋值非常重要，并确保代码在发生自我赋值时行为正确。

然而，自我赋值是非常罕见的情况，针对它进行优化实际上可能会使正常情况变得更差。由于正常情况更为常见，针对自我赋值的优化可能会降低代码效率（因此使用时需谨慎）。

例如，实现赋值运算符的常用技术是复制和交换惯用法。该技术的正常实现通常不会测试自我赋值（尽管自我赋值代价较高，因为会进行复制）。原因是对正常情况的性能下降被证明代价更大（因为发生频率更高）。

版本 ≥ c++11

移动赋值运算符也必须防止自我赋值。然而，许多此类运算符的逻辑基于`std::swap`，它可以很好地处理同一内存的交换。因此，如果你的移动赋值逻辑仅仅是一系列交换操作，那么你不需要自我赋值保护。

如果情况不是这样，你必须采取与上述类似的措施。

Any copy assignment operator you write must be able to take this into account.

The typical way to do so is to wrap all of the assignment logic in a condition like this:

```
SomeType &operator=(const SomeType &other)
{
    if(this != &other)
    {
        //Do assignment logic.
    }
    return *this;
}
```

Note: It is important to think about self-assignment and ensure that your code behaves correctly when it happens. However, self-assignment is a very rare occurrence and optimizing to prevent it may actually pessimize the normal case. Since the normal case is much more common, pessimizing for self-assignment may well reduce your code efficiency (so be careful using it).

As an example, the normal technique for implementing the assignment operator is the [copy and swap idiom](#). The normal implementation of this technique does not bother to test for self-assignment (even though self-assignment is expensive because a copy is made). The reason is that pessimization of the normal case has been shown to be much more costly (as it happens more often).

Version ≥ c++11

Move assignment operators must also be protected against self-assignment. However, the logic for many such operators is based on `std::swap`, which can handle swapping from/to the same memory just fine. So if your move assignment logic is nothing more than a series of swap operations, then you do not need self-assignment protection.

If this is not the case, you *must* take similar measures as above.

第83章：RAII：资源获取即初始化

第83.1节：锁定

错误的锁定：

```
std::mutex mtx;

void 错误锁定示例() {
    mtx.lock();
    尝试
    {
        foo();
        bar();
        if (baz()) {
            mtx.unlock(); // 必须在每个退出点解锁。
            return;
        }
        quux();
        mtx.unlock(); // 正常解锁发生在这里。
    }
    catch(...) {
        mtx.unlock(); // 必须在出现异常时也强制解锁
        throw; // 并允许异常继续抛出。
    }
}
```

这种实现互斥锁加锁和解锁的方式是错误的。为了确保通过unlock()正确释放互斥锁，程序员必须确保所有导致函数退出的流程都调用了unlock()。正如上面所示，这是一种脆弱的过程，因为它要求维护者手动继续遵循该模式。

使用适当设计的类来实现RAII，这个问题就变得简单了：

```
std::mutex mtx;

void good_lock_example() {
    std::lock_guard<std::mutex> lk(mtx); // 构造函数加锁。
    // 析构函数解锁。析构函数调用
    // 由语言保证。
    foo();
    bar();
    if (baz()) {
        return;
    }
    quux();
}
```

lock_guard是一个非常简单的类模板，它在构造函数中调用参数的lock()，保持对该参数的引用，并在析构函数中调用参数的unlock()。也就是说，当lock_guard超出作用域时，mutex保证被解锁。无论超出作用域的原因是异常还是提前返回——所有情况都被处理；无论控制流如何，我们都保证正确解锁。

Chapter 83: RAII: Resource Acquisition Is Initialization

Section 83.1: Locking

Bad locking:

```
std::mutex mtx;

void bad_lock_example() {
    mtx.lock();
    try
    {
        foo();
        bar();
        if (baz()) {
            mtx.unlock(); // Have to unlock on each exit point.
            return;
        }
        quux();
        mtx.unlock(); // Normal unlock happens here.
    }
    catch(...) {
        mtx.unlock(); // Must also force unlock in the presence of
        // exceptions and allow the exception to continue.
    }
}
```

That is the wrong way to implement the locking and unlocking of the mutex. To ensure the correct release of the mutex with unlock() requires the programmer to make sure that all the flows resulting in the exiting of the function result in a call to unlock(). As shown above this is a brittle process as it requires any maintainers to continue following the pattern manually.

Using an appropriately crafted class to implement RAII, the problem is trivial:

```
std::mutex mtx;

void good_lock_example() {
    std::lock_guard<std::mutex> lk(mtx); // constructor locks.
    // destructor unlocks. destructor call
    // guaranteed by language.
    foo();
    bar();
    if (baz()) {
        return;
    }
    quux();
}
```

lock_guard is an extremely simple class template that simply calls lock() on its argument in its constructor, keeps a reference to the argument, and calls unlock() on the argument in its destructor. That is, when the lock_guard goes out of scope, the mutex is guaranteed to be unlocked. It doesn't matter if the reason it went out of scope is an exception or an early return - all cases are handled; regardless of the control flow, we have guaranteed that we will unlock correctly.

第83.2节 : ScopeSuccess (c++17)

版本 ≥ C++17

多亏了int std::uncaught_exceptions(), 我们可以实现仅在成功时（作用域内无抛出异常）执行的操作。之前bool std::uncaught_exception()仅允许检测是否有任何堆栈展开正在进行。

```
#include <exception>
#include <iostream>

template <typename F>
class ScopeSuccess
{
private:
    F f;
    int uncaughtExceptionCount = std::uncaught_exceptions();
public:
    explicit ScopeSuccess(const F& f) : f(f) {}
    ScopeSuccess(const ScopeSuccess&) = delete;
    ScopeSuccess& operator =(const ScopeSuccess&) = delete;

    // f() 可能会抛出异常, 因为它可以被正常捕获。
    ~ScopeSuccess() noexcept(noexcept(f())) {
        if (uncaughtExceptionCount == std::uncaught_exceptions()) {
            f();
        }
    }
};

struct Foo {
    ~Foo() {
        try {
            ScopeSuccess logSuccess{[](){std::cout << "Success 1";}};// 作用域成功,
            // 即使在栈展开期间 Foo 被销毁
            // (即当 0 < std::uncaught_exceptions() 时)
            // (或之前 std::uncaught_exception() == true)
            } catch (...) {
        }
        try {
            ScopeSuccess logSuccess{[](){std::cout << "成功 2";}};
            throw std::runtime_error("失败"); // std::uncaught_exceptions 的
                                            // 返回值增加} catch (...) { // std::uncaught_exception()
                                            // 的返回值减少
        }
    }
};

int main()
{
    try {
        Foo foo;
        throw std::runtime_error("失败"); // std::uncaught_exceptions() == 1
    } catch (...) { // std::uncaught_exceptions() == 0
    }
}
```

Section 83.2: ScopeSuccess (c++17)

Version ≥ C++17

Thanks to `int std::uncaught_exceptions()`, we can implement action which executes only on success (no thrown exception in scope). Previously `bool std::uncaught_exception()` just allows to detect if **any** stack unwinding is running.

```
#include <exception>
#include <iostream>

template <typename F>
class ScopeSuccess
{
private:
    F f;
    int uncaughtExceptionCount = std::uncaught_exceptions();
public:
    explicit ScopeSuccess(const F& f) : f(f) {}
    ScopeSuccess(const ScopeSuccess&) = delete;
    ScopeSuccess& operator =(const ScopeSuccess&) = delete;

    // f() might throw, as it can be caught normally.
    ~ScopeSuccess() noexcept(noexcept(f())) {
        if (uncaughtExceptionCount == std::uncaught_exceptions()) {
            f();
        }
    }
};

struct Foo {
    ~Foo() {
        try {
            ScopeSuccess logSuccess{[](){std::cout << "Success 1\n";}};
            // Scope succeeds,
            // even if Foo is destroyed during stack unwinding
            // (so when 0 < std::uncaught_exceptions())
            // (or previously std::uncaught_exception() == true)
            } catch (...) {
        }
        try {
            ScopeSuccess logSuccess{[](){std::cout << "Success 2\n";}};
            throw std::runtime_error("Failed"); // returned value
                                                // of std::uncaught_exceptions increases
        } catch (...) { // returned value of std::uncaught_exceptions decreases
        }
    }
};

int main()
{
    try {
        Foo foo;
        throw std::runtime_error("Failed"); // std::uncaught_exceptions() == 1
    } catch (...) { // std::uncaught_exceptions() == 0
    }
}
```

输出：

成功 1

第 83.3 节 : ScopeFail (C++17)

版本 ≥ C++17

多亏了 `int std::uncaught_exceptions()`, 我们可以实现仅在失败时 (作用域内抛出异常) 执行的操作。之前 `bool std::uncaught_exception()` 仅允许检测是否有 任何 栈展开正在进行。

```
#include <exception>
#include <iostream>

template <typename F>
class ScopeFail
{
private:
    F f;
    int uncaughtExceptionCount = std::uncaught_exceptions();
public:
    explicit ScopeFail(const F& f) : f(f) {}
    ScopeFail(const ScopeFail&) = delete;
    ScopeFail& operator =(const ScopeFail&) = delete;

    // f() 不应抛出异常, 否则将调用 std::terminate。
    ~ScopeFail() {
        if (uncaughtExceptionCount != std::uncaught_exceptions()) {
            f();
        }
    }
};

struct Foo {
    ~Foo() {
        try {
            ScopeFail logFailure{[](){std::cout << "失败 1";}}; // 范围成功,
            // 即使在栈展开期间 Foo 被销毁
            // (即当 0 < std::uncaught_exceptions() 时)
            // (或之前 std::uncaught_exception() == true)
        } catch (...) {
        }
        try {
            ScopeFail logFailure{[](){std::cout << "失败 2";}};
            throw std::runtime_error("失败"); // std::uncaught_exceptions 的
                                            // 返回值增加} catch (...) { // std::uncaught_exception
                                            // 的返回值减少
        }
    }
};

int main()
{
    try {
        Foo foo;
```

Output:

Success 1

Section 83.3: ScopeFail (c++17)

Version ≥ C++17

Thanks to `int std::uncaught_exceptions()`, we can implement action which executes only on failure (thrown exception in scope). Previously `bool std::uncaught_exception()` just allows to detect if **any** stack unwinding is running.

```
#include <exception>
#include <iostream>

template <typename F>
class ScopeFail
{
private:
    F f;
    int uncaughtExceptionCount = std::uncaught_exceptions();
public:
    explicit ScopeFail(const F& f) : f(f) {}
    ScopeFail(const ScopeFail&) = delete;
    ScopeFail& operator =(const ScopeFail&) = delete;

    // f() should not throw, else std::terminate is called.
    ~ScopeFail() {
        if (uncaughtExceptionCount != std::uncaught_exceptions()) {
            f();
        }
    }
};

struct Foo {
    ~Foo() {
        try {
            ScopeFail logFailure{[](){std::cout << "Fail 1\n";}};
            // Scope succeeds,
            // even if Foo is destroyed during stack unwinding
            // (so when 0 < std::uncaught_exceptions())
            // (or previously std::uncaught_exception() == true)
        } catch (...) {
        }
        try {
            ScopeFail logFailure{[](){std::cout << "Failure 2\n";}};
            throw std::runtime_error("Failed"); // returned value
                                                // of std::uncaught_exceptions increases
        } catch (...) { // returned value of std::uncaught_exceptions decreases
        }
    }
};

int main()
{
    try {
        Foo foo;
```

```

    throw std::runtime_error("失败"); // std::uncaught_exceptions() == 1
} catch (...) { // std::uncaught_exceptions() == 0
}
}

```

输出：

失败 2

第 83.4 节：Finally/ScopeExit

对于不想编写特殊类来处理某些资源的情况，我们可以编写一个通用类：

```

template<typename Function>
class Finally final
{
public:
    explicit Finally(Function f) : f(std::move(f)) {}
    ~Finally() { f(); } // (1) 见下文

    Finally(const Finally&) = delete;
    Finally(Finally&&) = default;
    Finally& operator =(const Finally&) = delete;
    Finally& operator =(Finally&&) = delete;

private:
    Function f;
};

// 当返回的对象超出作用域时执行函数 f。
template<typename Function>
auto onExit(Function &&f) { return Finally<std::decay_t<Function>>{std::forward<Function>(f)}; }

```

及其示例用法

```

void foo(std::vector<int>& v, int i)
{
    // ...

    v[i] += 42;
    auto autoRollBackChange = onExit([&]() { v[i] -= 42; });

    // ... 代码作为递归调用 `foo(v, i + 1)`
}

```

注 (1)：关于析构函数的定义，需要考虑一些处理异常的讨论：

- ~Finally() noexcept { f(); }: 在发生异常时调用 std::terminate~Finally() noexcept(noexcept)
- ept(f()) { f(); }: 仅在栈展开期间发生异常时调用 terminate()
- ~最终() 不抛出异常 { 尝试 { f(); } 捕获 (...) { /* 忽略异常 (可能会记录日志) */ } } 未调用 std::terminate，但我们无法处理错误（即使不是栈展开时）。

```

    throw std::runtime_error("Failed"); // std::uncaught_exceptions() == 1
} catch (...) { // std::uncaught_exceptions() == 0
}
}

```

Output:

Failure 2

Section 83.4: Finally/ScopeExit

For cases when we don't want to write special classes to handle some resource, we may write a generic class:

```

template<typename Function>
class Finally final
{
public:
    explicit Finally(Function f) : f(std::move(f)) {}
    ~Finally() { f(); } // (1) See below

    Finally(const Finally&) = delete;
    Finally(Finally&&) = default;
    Finally& operator =(const Finally&) = delete;
    Finally& operator =(Finally&&) = delete;

private:
    Function f;
};

// Execute the function f when the returned object goes out of scope.
template<typename Function>
auto onExit(Function &&f) { return Finally<std::decay_t<Function>>{std::forward<Function>(f)}; }

```

And its example usage

```

void foo(std::vector<int>& v, int i)
{
    // ...

    v[i] += 42;
    auto autoRollBackChange = onExit([&]() { v[i] -= 42; });

    // ... code as recursive call `foo(v, i + 1)`
}

```

Note (1): Some discussion about destructor definition has to be considered to handle exception:

- ~Finally() noexcept { f(); }: std::terminate is called in case of exception
- ~Finally() noexcept(noexcept(f())) { f(); }: terminate() is called only in case of exception during stack unwinding.
- ~Finally() noexcept { try { f(); } catch (...) { /* ignore exception (might log it) */ } } No std::terminate called, but we cannot handle error (even for non stack unwinding).

第84章：RTTI：运行时类型信息

第84.1节：dynamic_cast

将 `dynamic_cast<>()` 用作函数，它可以帮助你在继承层次结构中向下转换（主要描述）。

如果你必须对某些派生类B和C执行一些非多态操作，但收到的是基类A，那么
请这样写：

```
class A { public: virtual ~A(){} };  
  
class B: public A  
{ public: void work4B(){} };  
  
class C: public A  
{ public: void work4C(){} };  
  
void non_polymorphic_work(A* ap)  
{  
    if (B* bp =dynamic_cast<B*>(ap))  
        bp->work4B();  
    if (C* cp =dynamic_cast<C*>(ap))  
        cp->work4C();  
}
```

第84.2节：typeid关键字

`typeid`关键字是一个一元运算符，如果操作数的类型是多态类类型，则返回其运行时类型信息。它返回一个类型为`const std::type_info`的左值。顶层的cv限定符会被忽略。

```
struct Base {  
    virtual ~Base() = default;  
};  
struct Derived : Base {};  
Base* b = new Derived;  
assert(typeid(*b) == typeid(Derived{})); // 正确
```

`typeid`也可以直接应用于类型。在这种情况下，首先会去除顶层引用，然后忽略顶层的cv限定符。因此，上述例子也可以写成 `typeid(Derived)` 而不是 `typeid(Derived{})`：

```
assert(typeid(*b) == typeid(Derived{})); // 正确
```

如果 `typeid` 应用于任何 不是 多态类类型的表达式，则操作数不会被求值，返回的类型信息是静态类型的。

```
struct Base {  
    // 注意：无虚析构函数  
};  
结构体 Derived : Base {};  
Derived d;  
Base& b = d;
```

Chapter 84: RTTI: Run-Time Type Information

Section 84.1: dynamic_cast

Use `dynamic_cast<>()` as a function, which helps you to cast down through an inheritance hierarchy (main description).

If you must do some non-polymorphic work on some derived classes B and C, but received the base `class A`, then write like this:

```
class A { public: virtual ~A(){} };  
  
class B: public A  
{ public: void work4B(){} };  
  
class C: public A  
{ public: void work4C(){} };  
  
void non_polymorphic_work(A* ap)  
{  
    if (B* bp =dynamic_cast<B*>(ap))  
        bp->work4B();  
    if (C* cp =dynamic_cast<C*>(ap))  
        cp->work4C();  
}
```

Section 84.2: The typeid keyword

The `typeid` keyword is a unary operator that yields run-time type information about its operand if the operand's type is a polymorphic class type. It returns an lvalue of type `const std::type_info`. Top-level cv-qualification are ignored.

```
struct Base {  
    virtual ~Base() = default;  
};  
struct Derived : Base {};  
Base* b = new Derived;  
assert(typeid(*b) == typeid(Derived{})); // OK
```

`typeid` can also be applied to a type directly. In this case, first top-level references are stripped, then top-level cv-qualification is ignored. Thus, the above example could have been written with `typeid(Derived)` instead of `typeid(Derived{})`:

```
assert(typeid(*b) == typeid(Derived{})); // OK
```

If `typeid` is applied to any expression that is *not* of polymorphic class type, the operand is not evaluated, and the type info returned is for the static type.

```
struct Base {  
    // note: no virtual destructor  
};  
struct Derived : Base {};  
Derived d;  
Base& b = d;
```

```
断言(typeid(b) == typeid(Base)); // 不是 Derived  
断言(std::declval<Base>() == typeid(Base)); // 因为未求值所以正确
```

第 84.3 节：类型名称

你可以通过使用 `std::type_info` 对象的成员函数 `.name()` 来在运行时获取类型的实现定义名称，该对象由 `typeid` 返回。

```
#include <iostream>  
#include <typeinfo>  
  
int main()  
{  
    int speed = 110;  
  
    std::cout << typeid(speed).name() << ";"}
```

输出（实现定义）：

```
int
```

第84.4节：在C++中何时使用哪种类型转换

使用 `dynamic_cast` 在继承层次结构中转换指针/引用。

使用 `static_cast` 进行普通类型转换。

使用 `reinterpret_cast` 进行位模式的低级重新解释。请极其谨慎使用。

使用 `const_cast` 去除`const/volatile`。除非不得不使用`const`不正确的API，否则应避免使用。

```
assert(typeid(b) == typeid(Base)); // not Derived  
assert(typeid(std::declval<Base>()) == typeid(Base)); // OK because unevaluated
```

Section 84.3: Name of a type

You can retrieve the implementation defined name of a type in runtime by using the `.name()` member function of the `std::type_info` object returned by `typeid`.

```
#include <iostream>  
#include <typeinfo>  
  
int main()  
{  
    int speed = 110;  
  
    std::cout << typeid(speed).name() << '\n';  
}
```

Output (implementation-defined):

```
int
```

Section 84.4: When to use which cast in c++

Use **dynamic_cast** for converting pointers/references within an inheritance hierarchy.

Use **static_cast** for ordinary type conversions.

Use **reinterpret_cast** for low-level reinterpreting of bit patterns. Use with extreme caution.

Use **const_cast** for casting away `const/volatile`. Avoid this unless you are stuck using a const-incorrect API.

第85章：互斥锁

第85.1节：互斥类型

C++1x 提供了多种互斥量类：

- std::mutex - 提供简单的锁定功能。
- std::timed_mutex - 提供 try_to_lock 功能
- std::recursive_mutex - 允许同一线程递归锁定。
- std::shared_mutex, std::shared_timed_mutex - 提供共享锁和独占锁功能。

第85.2节：std::lock

std::lock 使用死锁避免算法来锁定一个或多个互斥量。如果在调用过程中抛出异常以锁定多个对象，std::lock 会在重新抛出异常之前解锁已成功锁定的对象。

```
std::lock(_mutex1, _mutex2);
```

第85.3节：std::unique_lock, std::shared_lock, std::lock_guard

用于 RAII 风格获取尝试锁、定时尝试锁和递归锁。

std::unique_lock 允许对互斥量的独占所有权。

std::shared_lock 允许对互斥量进行共享所有权。多个线程可以在 std::shared_mutex 上持有 std::shared_lock。C++14 起可用。

std::lock_guard 是 std::unique_lock 和 std::shared_lock 的轻量级替代方案。

```
#include <unordered_map>
#include <mutex>
#include <shared_mutex>
#include <thread>
#include <string>
#include <iostream>

class PhoneBook {
public:
    std::string getPhoneNo( const std::string & name )
    {
        std::shared_lock<std::shared_timed_mutex> l(_protect);
        auto it = _phonebook.find( name );
        if ( it != _phonebook.end() )
            return (*it).second;
        return "";
    }
    void addPhoneNo ( const std::string & name, const std::string & phone )
    {
        std::unique_lock<std::shared_timed_mutex> l(_protect);
        _phonebook[name] = phone;
    }

    std::shared_timed_mutex _protect;
    std::unordered_map<std::string, std::string> _phonebook;
}
```

Chapter 85: Mutexes

Section 85.1: Mutex Types

C++1x offers a selection of mutex classes:

- std::mutex - offers simple locking functionality.
- std::timed_mutex - offers try_to_lock functionality
- std::recursive_mutex - allows recursive locking by the same thread.
- std::shared_mutex, std::shared_timed_mutex - offers shared and unique lock functionality.

Section 85.2: std::lock

std::lock uses deadlock avoidance algorithms to lock one or more mutexes. If an exception is thrown during a call to lock multiple objects, std::lock unlocks the successfully locked objects before re-throwing the exception.

```
std::lock(_mutex1, _mutex2);
```

Section 85.3: std::unique_lock, std::shared_lock, std::lock_guard

Used for the RAII style acquiring of try locks, timed try locks and recursive locks.

std::unique_lock allows for exclusive ownership of mutexes.

std::shared_lock allows for shared ownership of mutexes. Several threads can hold std::shared_locks on a std::shared_mutex. Available from C++ 14.

std::lock_guard is a lightweight alternative to std::unique_lock and std::shared_lock.

```
#include <unordered_map>
#include <mutex>
#include <shared_mutex>
#include <thread>
#include <string>
#include <iostream>

class PhoneBook {
public:
    std::string getPhoneNo( const std::string & name )
    {
        std::shared_lock<std::shared_timed_mutex> l(_protect);
        auto it = _phonebook.find( name );
        if ( it != _phonebook.end() )
            return (*it).second;
        return "";
    }
    void addPhoneNo ( const std::string & name, const std::string & phone )
    {
        std::unique_lock<std::shared_timed_mutex> l(_protect);
        _phonebook[name] = phone;
    }

    std::shared_timed_mutex _protect;
    std::unordered_map<std::string, std::string> _phonebook;
}
```

第85.4节：锁类策略：std::try_to_lock, std::adopt_lock, std::defer_lock

创建 std::unique_lock 时，有三种不同的锁定策略可供选择：std::try_to_lock, std::defer_lock 和 std::adopt_lock

1. std::try_to_lock 允许尝试获取锁而不阻塞：

```
{
std::atomic_int temp {0};
std::mutex _mutex;

std::thread t( [&](){

    while( temp!= -1){
std::this_thread::sleep_for(std::chrono::seconds(5));
    std::unique_lock<std::mutex> lock( _mutex, std::try_to_lock);

    if(lock.owns_lock()){
        //执行某些操作
temp=0;
    }
});

    while ( true )
{
std::this_thread::sleep_for(std::chrono::seconds(1));
    std::unique_lock<std::mutex> lock( _mutex, std::try_to_lock);
    if(lock.owns_lock()){
        if (temp < INT_MAX){
            ++temp;
        }
std::cout << temp << std::endl;
    }
}
}
```

2. std::defer_lock 允许创建一个锁结构而不立即获取锁。当锁定多个互斥量时，如果两个函数调用者试图同时获取锁，就存在死锁的可能性：

```
{
std::unique_lock<std::mutex> lock1(_mutex1, std::defer_lock);
std::unique_lock<std::mutex> lock2(_mutex2, std::defer_lock);
lock1.lock()
lock2.lock(); // 此处发生死锁
std::cout << "Locked! << std::endl;
//...
}
```

使用以下代码，无论函数中发生什么，锁都会以适当的顺序被获取和释放：

```
{
std::unique_lock<std::mutex> lock1(_mutex1, std::defer_lock);
std::unique_lock<std::mutex> lock2(_mutex2, std::defer_lock);
```

Section 85.4: Strategies for lock classes: std::try_to_lock, std::adopt_lock, std::defer_lock

When creating a std::unique_lock, there are three different locking strategies to choose from: std::try_to_lock, std::defer_lock and std::adopt_lock

1. std::try_to_lock 允许尝试获取锁而不阻塞：

```
{
std::atomic_int temp {0};
std::mutex _mutex;

std::thread t( [&](){

    while( temp!= -1){
std::this_thread::sleep_for(std::chrono::seconds(5));
    std::unique_lock<std::mutex> lock( _mutex, std::try_to_lock);

    if(lock.owns_lock()){
        //do something
temp=0;
    }
});

    while ( true )
{
std::this_thread::sleep_for(std::chrono::seconds(1));
    std::unique_lock<std::mutex> lock( _mutex, std::try_to_lock);
    if(lock.owns_lock()){
        if (temp < INT_MAX){
            ++temp;
        }
std::cout << temp << std::endl;
    }
}
}
```

2. std::defer_lock 允许创建一个锁结构而不立即获取锁。当锁定多个互斥量时，如果两个函数调用者试图同时获取锁，就存在死锁的可能性：

```
{
std::unique_lock<std::mutex> lock1(_mutex1, std::defer_lock);
std::unique_lock<std::mutex> lock2(_mutex2, std::defer_lock);
lock1.lock()
lock2.lock(); // deadlock here
std::cout << "Locked! << std::endl;
//...
}
```

With the following code, whatever happens in the function, the locks are acquired and released in appropriate order:

```
{
std::unique_lock<std::mutex> lock1(_mutex1, std::defer_lock);
std::unique_lock<std::mutex> lock2(_mutex2, std::defer_lock);
```

```

std::lock(lock1,lock2); // 不会发生死锁
std::cout << "Locked! << std::endl;
//...
}

```

3. `std::adopt_lock` 不会尝试第二次加锁，如果调用线程当前已经拥有该锁。

```

{
std::unique_lock<std::mutex> lock1(_mutex1, std::adopt_lock);
std::unique_lock<std::mutex> lock2(_mutex2, std::adopt_lock);
std::cout << "Locked! << std::endl;
//...
}

```

需要注意的是，`std::adopt_lock` 并不能替代递归互斥锁的使用。当锁超出作用域时，互斥锁将被释放。

第85.5节：`std::mutex`

`std::mutex` 是一种简单的、非递归的同步结构，用于保护被多个线程访问的数据。

```

std::atomic_int temp{0};
std::mutex _mutex;

std::thread t( [&](){
    while( temp!= -1){
        std::this_thread::sleep_for(std::chrono::seconds(5));
        std::unique_lock<std::mutex> lock( _mutex);

        temp=0;
    }
});

while ( true )
{
    std::this_thread::sleep_for(std::chrono::milliseconds(1));
    std::unique_lock<std::mutex> lock( _mutex, std::try_to_lock);
    if ( temp < INT_MAX )
        temp++;
    cout << temp << endl;
}

```

第85.6节：`std::scoped_lock` (C++ 17)

`std::scoped_lock` 提供了RAII风格的语义，用于拥有一个或多个互斥量，结合了 `std::lock` 使用的避免死锁算法。当 `std::scoped_lock` 被销毁时，互斥量会按照获取的相反顺序释放。

```

{
std::scoped_lock lock{_mutex1,_mutex2};
//执行某些操作
}

```

```

std::lock(lock1,lock2); // no deadlock possible
std::cout << "Locked! << std::endl;
//...
}

```

3. `std::adopt_lock` does not attempt to lock a second time if the calling thread currently owns the lock.

```

{
std::unique_lock<std::mutex> lock1(_mutex1, std::adopt_lock);
std::unique_lock<std::mutex> lock2(_mutex2, std::adopt_lock);
std::cout << "Locked! << std::endl;
//...
}

```

Something to keep in mind is that `std::adopt_lock` is not a substitute for recursive mutex usage. When the lock goes out of scope the mutex is **released**.

Section 85.5: `std::mutex`

`std::mutex` is a simple, non-recursive synchronization structure that is used to protect data which is accessed by multiple threads.

```

std::atomic_int temp{0};
std::mutex _mutex;

std::thread t( [&]{
    while( temp!= -1){
        std::this_thread::sleep_for(std::chrono::seconds(5));
        std::unique_lock<std::mutex> lock( _mutex);

        temp=0;
    }
});

while ( true )
{
    std::this_thread::sleep_for(std::chrono::milliseconds(1));
    std::unique_lock<std::mutex> lock( _mutex, std::try_to_lock);
    if ( temp < INT_MAX )
        temp++;
    cout << temp << endl;
}

```

Section 85.6: `std::scoped_lock` (C++ 17)

`std::scoped_lock` provides RAII style semantics for owning one or more mutexes, combined with the lock avoidance algorithms used by `std::lock`. When `std::scoped_lock` is destroyed, mutexes are released in the reverse order from which they were acquired.

```

{
std::scoped_lock lock{_mutex1,_mutex2};
//do something
}

```

第86章：递归互斥量

第86.1节：std::recursive_mutex

递归互斥量允许同一线程递归地锁定资源——一直到未指定的限制。

这在现实中几乎没有正当理由。某些复杂的实现可能需要在不释放锁的情况下调用函数的重载副本。

```
std::atomic_int temp{0};  
std::recursive_mutex _mutex;  
  
//launch_deferred 在相同线程ID上启动异步任务  
  
auto future1 = std::async(  
    std::launch::deferred,  
    [&]()  
    {  
        std::cout << std::this_thread::get_id() << std::endl;  
  
        std::this_thread::sleep_for(std::chrono::seconds(3));  
        std::unique_lock<std::recursive_mutex> lock(_mutex);  
        temp=0;  
    });  
  
auto future2 = std::async(  
    std::launch::deferred,  
    [&]()  
    {  
        std::cout << std::this_thread::get_id() << std::endl;  
        while (true)  
        {  
            std::this_thread::sleep_for(std::chrono::milliseconds(1));  
            std::unique_lock<std::recursive_mutex> lock(_mutex, std::try_to_lock);  
            if (temp < INT_MAX)  
                temp++;  
  
            cout << temp << endl;  
        }  
    });  
future1.get();  
future2.get();
```

Chapter 86: Recursive Mutex

Section 86.1: std::recursive_mutex

Recursive mutex allows the same thread to recursively lock a resource - up to an unspecified limit.

There are very few real-word justifications for this. Certain complex implementations might need to call an overloaded copy of a function without releasing the lock.

```
std::atomic_int temp{0};  
std::recursive_mutex _mutex;  
  
//launch_deferred launches asynchronous tasks on the same thread id  
  
auto future1 = std::async(  
    std::launch::deferred,  
    [&]()  
    {  
        std::cout << std::this_thread::get_id() << std::endl;  
  
        std::this_thread::sleep_for(std::chrono::seconds(3));  
        std::unique_lock<std::recursive_mutex> lock(_mutex);  
        temp=0;  
    });  
  
auto future2 = std::async(  
    std::launch::deferred,  
    [&]()  
    {  
        std::cout << std::this_thread::get_id() << std::endl;  
        while (true)  
        {  
            std::this_thread::sleep_for(std::chrono::milliseconds(1));  
            std::unique_lock<std::recursive_mutex> lock(_mutex, std::try_to_lock);  
            if (temp < INT_MAX)  
                temp++;  
  
            cout << temp << endl;  
        }  
    });  
future1.get();  
future2.get();
```

第87章：信号量

截至目前，C++中没有信号量，但可以通过互斥锁和条件变量轻松实现。

此示例摘自：

[C++0x没有信号量？如何同步线程？](#)

第87.1节：C++11信号量

```
#include <mutex>
#include <condition_variable>

class Semaphore {
public:
Semaphore (int count_ = 0)
: count(count_)
{
}

inline void notify( int tid ) {
    std::unique_lock<std::mutex> lock(mtx);
    count++;
    cout << "线程 " << tid << " 通知" << endl;
    //通知等待的线程
cv.notify_one();
}
inline void wait( int tid ) {
    std::unique_lock<std::mutex> lock(mtx);
    while(count == 0) {
        cout << "线程 " << tid << " 等待" << endl;
        //等待互斥锁直到调用通知
cv.wait(lock);
        cout << "线程 " << tid << " 运行" << endl;
    }
    count--;
}
private:
std::mutex mtx;
std::condition_variable cv;
int count;
};
```

第87.2节：信号量类的实际应用

下面的函数添加了四个线程。三个线程竞争信号量，信号量的计数设置为1。一个较慢的线程调用 `notify_one()`，允许一个等待的线程继续执行。

结果是 `s1` 立即开始自旋，导致信号量的使用计数 `count` 保持在1以下。其他线程依次在条件变量上等待，直到调用 `notify()`。

```
int main()
{
Semaphore sem(1);

thread s1([&]() {
```

Chapter 87: Semaphore

Semaphores are not available in C++ as of now, but can easily be implemented with a mutex and a condition variable.

This example was taken from:

[C++0x has no semaphores? How to synchronize threads?](#)

Section 87.1: Semaphore C++ 11

```
#include <mutex>
#include <condition_variable>

class Semaphore {
public:
Semaphore (int count_ = 0)
: count(count_)
{
}

inline void notify( int tid ) {
    std::unique_lock<std::mutex> lock(mtx);
    count++;
    cout << "thread " << tid << " notify" << endl;
    //notify the waiting thread
cv.notify_one();
}
inline void wait( int tid ) {
    std::unique_lock<std::mutex> lock(mtx);
    while(count == 0) {
        cout << "thread " << tid << " wait" << endl;
        //wait on the mutex until notify is called
cv.wait(lock);
        cout << "thread " << tid << " run" << endl;
    }
    count--;
}
private:
std::mutex mtx;
std::condition_variable cv;
int count;
};
```

Section 87.2: Semaphore class in action

The following function adds four threads. Three threads compete for the semaphore, which is set to a count of one. A slower thread calls `notify_one()`, allowing one of the waiting threads to proceed.

The result is that `s1` immediately starts spinning, causing the Semaphore's usage count to remain below 1. The other threads wait in turn on the condition variable until `notify()` is called.

```
int main()
{
Semaphore sem(1);

thread s1([&]() {
```

```

        while(true) {
this_thread::sleep_for(std::chrono::seconds(5));
    sem.wait( 1 );
}
});
thread s2([&]() {
    while(true){
sem.wait( 2 );
}
});
thread s3([&]() {
    while(true) {
this_thread::sleep_for(std::chrono::milliseconds(600));
    sem.wait( 3 );
}
});
thread s4([&]() {
    while(true) {
this_thread::sleep_for(std::chrono::seconds(5));
    sem.notify( 4 );
}
});

s1.join();
s2.join();
s3.join();
s4.join();

...
}

```

```

        while(true) {
this_thread::sleep_for(std::chrono::seconds(5));
    sem.wait( 1 );
}
});
thread s2([&]() {
    while(true){
sem.wait( 2 );
}
});
thread s3([&]() {
    while(true) {
this_thread::sleep_for(std::chrono::milliseconds(600));
    sem.wait( 3 );
}
});
thread s4([&]() {
    while(true) {
this_thread::sleep_for(std::chrono::seconds(5));
    sem.notify( 4 );
}
});

s1.join();
s2.join();
s3.join();
s4.join();

...
}

```

第88章：期货与承诺

承诺（Promises）和期货（Futures）用于在不同线程之间传递单个对象。

一个std::promise对象由生成结果的线程设置。

一个std::future对象可用于获取值、测试值是否可用，或暂停执行直到值可用。

第88.1节：异步操作类

- std::async：执行异步操作。
- std::future：提供对异步操作结果的访问。
- std::promise：封装异步操作的结果。
- std::packaged_task：将函数及其返回类型的相关承诺打包在一起。

第88.2节：std::future 和 std::promise

以下示例设置了一个 promise，由另一个线程使用：

```
{  
    auto promise = std::promise<std::string>();  
  
    auto producer = std::thread([&]  
    {  
        promise.set_value("Hello World");  
    });  
  
    auto future = promise.get_future();  
  
    auto consumer = std::thread([&]  
    {  
        std::cout << future.get();  
    });  
  
    producer.join();  
    consumer.join();  
}
```

第88.3节：延迟异步示例

此代码实现了一个 std::async 的版本，但其行为就像 async 总是以 deferredLaunch 策略调用一样。此函数也没有 async 的特殊 future 行为；返回的 future 可以在从未获取其值的情况下被销毁。

```
模板<类型名 F>  
auto async_deferred(F&& func) -> std::future<decltype(func())>  
{  
    using result_type = decltype(func());  
  
    auto promise = std::promise<result_type>();  
    auto future = promise.get_future();  
  
    std::thread(std::bind([=](std::promise<result_type>& promise)  
    {  
        尝试  
    }, promise))  
    .join();  
}  
else  
    return future;
```

Chapter 88: Futures and Promises

Promises and Futures are used to ferry a single object from one thread to another.

A std::promise object is set by the thread which generates the result.

A std::future object can be used to retrieve a value, to test to see if a value is available, or to halt execution until the value is available.

Section 88.1: Async operation classes

- std::async: performs an asynchronous operation.
- std::future: provides access to the result of an asynchronous operation.
- std::promise: packages the result of an asynchronous operation.
- std::packaged_task: bundles a function and the associated promise for its return type.

Section 88.2: std::future and std::promise

The following example sets a promise to be consumed by another thread:

```
{  
    auto promise = std::promise<std::string>();  
  
    auto producer = std::thread([&]  
    {  
        promise.set_value("Hello World");  
    });  
  
    auto future = promise.get_future();  
  
    auto consumer = std::thread([&]  
    {  
        std::cout << future.get();  
    });  
  
    producer.join();  
    consumer.join();  
}
```

Section 88.3: Deferred async example

This code implements a version of std::async, but it behaves as if async were always called with the deferred launch policy. This function also does not have async's special future behavior; the returned future can be destroyed without ever acquiring its value.

```
template<typename F>  
auto async_deferred(F&& func) -> std::future<decltype(func())>  
{  
    using result_type = decltype(func());  
  
    auto promise = std::promise<result_type>();  
    auto future = promise.get_future();  
  
    std::thread(std::bind([=](std::promise<result_type>& promise)  
    {  
        尝试  
    }, promise))  
    .join();  
}  
else  
    return future;
```

```

{
promise.set_value(func());
    // 注意：不适用于 std::promise<void>。需要一些元模板编程，超出本示例范围。
}
catch(...)

promise.set_exception(std::current_exception());
}

}, std::move(promise)).detach();

return future;
}

```

第88.4节：std::packaged_task 和 std::future

std::packaged_task 将函数及其返回类型相关的 promise 绑定在一起：

```

模板<类型名 F>
auto async_deferred(F&& func) -> std::future<decltype(func())>
{
    auto task = std::packaged_task<decltype(func())>(std::forward<F>(func));
    auto future = task.get_future();

    std::thread(std::move(task)).detach();

    return std::move(future);
}

```

线程会立即开始运行。我们可以选择将其分离，或者在作用域结束时将其加入。当调用 `std::thread` 的函数结束时，结果就已经准备好了。

注意，这与 `std::async` 略有不同，后者返回的 `std::future` 在析构时实际上会阻塞，直到线程结束。

第 88.5 节：std::future_error 和 std::future_errc

如果不满足 `std::promise` 和 `std::future` 的约束条件，将抛出类型为 `std::future_error` 的异常。

异常中的错误代码成员类型为 `std::future_errc`，取值如下，并附带一些测试用例：

```

enum class future_errc {
broken_promise      = /* 任务不再共享 */,
future_already_retrieved = /* 答案已被检索 */,
promise_already_satisfied = /* 答案已被存储 */,
no_state            = /* 访问处于非共享状态的承诺 */
};

```

非活动的 `promise`：

```

int test()
{
std::promise<int> pr;
    return 0; // 返回成功
}

```

```

{
promise.set_value(func());
    // Note: Will not work with std::promise<void>. Needs some meta-template programming
which is out of scope for this example.
}
catch(...)

promise.set_exception(std::current_exception());
}

}, std::move(promise)).detach();

return future;
}

```

Section 88.4: std::packaged_task and std::future

`std::packaged_task` bundles a function and the associated promise for its return type:

```

template<typename F>
auto async_deferred(F&& func) -> std::future<decltype(func())>
{
    auto task = std::packaged_task<decltype(func())>(std::forward<F>(func));
    auto future = task.get_future();

    std::thread(std::move(task)).detach();

    return std::move(future);
}

```

The thread starts running immediately. We can either detach it, or have join it at the end of the scope. When the function call to `std::thread` finishes, the result is ready.

Note that this is slightly different from `std::async` where the returned `std::future` when destructed will actually **block** until the thread is finished.

Section 88.5: std::future_error and std::future_errc

If constraints for `std::promise` and `std::future` are not met an exception of type `std::future_error` is thrown.

The error code member in the exception is of type `std::future_errc` and values are as below, along with some test cases:

```

enum class future_errc {
broken_promise      = /* the task is no longer shared */,
future_already_retrieved = /* the answer was already retrieved */,
promise_already_satisfied = /* the answer was stored already */,
no_state            = /* access to a promise in non-shared state */
};

```

Inactive promise:

```

int test()
{
std::promise<int> pr;
    return 0; // returns ok
}

```

活动的 promise，未使用：

```
int test()
{
std::promise<int> pr;
    auto fut = pr.get_future(); // 无限期阻塞！
    return 0;
}
```

重复获取：

```
int test()
{
std::promise<int> pr;
    auto fut1 = pr.get_future();

try{
    auto fut2 = pr.get_future(); // 第二次尝试获取 future
    return 0;
}
catch(const std::future_error& e)
{
    cout << e.what() << endl; // 错误：“future 已经从 promise 或 packaged_task 中被获取。”
    return -1;
}
return fut2.get();
}
```

设置 std::promise 的值两次：

```
int test()
{
std::promise<int> pr;
    auto fut = pr.get_future();
    try{
std::promise<int> pr2(std::move(pr));
    pr2.set_value(10);
pr2.set_value(10); // 第二次尝试设置 promise 会抛出异常
    }
catch(const std::future_error& e)
{
    cout << e.what() << endl; // 错误：“promise 的状态已经被设置。”
    return -1;
}
return fut.get();
}
```

第88.6节：std::future 和 std::async

在下面的简单并行归并排序示例中，std::async 用于启动多个并行的 merge_sort 任务。std::future 用于等待结果并进行同步：

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

void merge(int low,int mid,int high, vector<int>&num)
```

Active promise, unused:

```
int test()
{
    std::promise<int> pr;
    auto fut = pr.get_future(); //blocks indefinitely!
    return 0;
}
```

Double retrieval:

```
int test()
{
    std::promise<int> pr;
    auto fut1 = pr.get_future();

try{
    auto fut2 = pr.get_future(); // second attempt to get future
    return 0;
}
catch(const std::future_error& e)
{
    cout << e.what() << endl; // Error: "The future has already been retrieved from the
promise or packaged_task."
    return -1;
}
return fut2.get();
}
```

Setting std::promise value twice:

```
int test()
{
    std::promise<int> pr;
    auto fut = pr.get_future();
    try{
        std::promise<int> pr2(std::move(pr));
        pr2.set_value(10);
        pr2.set_value(10); // second attempt to set promise throws exception
    }
    catch(const std::future_error& e)
    {
        cout << e.what() << endl; // Error: "The state of the promise has already been
set."
        return -1;
    }
    return fut.get();
}
```

Section 88.6: std::future and std::async

In the following naive parallel merge sort example, std::async is used to launch multiple parallel merge_sort tasks. std::future is used to wait for the results and synchronize them:

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

void merge(int low,int mid,int high, vector<int>&num)
```

```

{
vector<int> copy(num.size());
    int h,i,j,k;
h=low;
    i=low;
    j=mid+1;

    while((h<=mid)&&(j<=high))
    {
        if(num[h]<=num[j])
        {
copy[i]=num[h];
            h++;
        }
        否则
        {
copy[i]=num[j];
            j++;
        }
i++;
    }
    if(h>mid)
    {
        for(k=j;k<=high;k++)
copy[i]=num[k];
            i++;
    }
    否则
    {
        for(k=h;k<=mid;k++)
copy[i]=num[k];
            i++;
    }
    for(k=low;k<=high;k++)
        swap(num[k],copy[k]);
}
}

void merge_sort(int low,int high,vector<int>& num)
{
    int mid;
    if(low>high)
    {
mid = low + (high-low)/2;
        auto future1 = std::async(std::launch::deferred,[&]()
merge_sort(low,mid,num);
                    );
        auto future2 = std::async(std::launch::deferred, [&]()
merge_sort(mid+1,high,num) ;
                    );
future1.get();
        future2.get();
        merge(low,mid,high,num);
    }
}

```

```

{
vector<int> copy(num.size());
    int h,i,j,k;
h=low;
    i=low;
    j=mid+1;

    while((h<=mid)&&(j<=high))
    {
        if(num[h]<=num[j])
        {
copy[i]=num[h];
            h++;
        }
        else
        {
copy[i]=num[j];
            j++;
        }
i++;
    }
    if(h>mid)
    {
        for(k=j;k<=high;k++)
copy[i]=num[k];
            i++;
    }
    else
    {
        for(k=h;k<=mid;k++)
copy[i]=num[k];
            i++;
    }
    for(k=low;k<=high;k++)
        swap(num[k],copy[k]);
}
}

void merge_sort(int low,int high,vector<int>& num)
{
    int mid;
    if(low>high)
    {
mid = low + (high-low)/2;
        auto future1 = std::async(std::launch::deferred,[&]()
                    {
merge_sort(low,mid,num);
                    });
        auto future2 = std::async(std::launch::deferred, [&]()
                    {
merge_sort(mid+1,high,num) ;
                    });
future1.get();
        future2.get();
        merge(low,mid,high,num);
    }
}

```

注意：在示例中，`std::async` 使用了策略 `std::launch_deferred` 启动。这是为了避免每次调用时创建新线程。在我们的示例中，`std::async` 的调用是无序的，它们在调用 `std::future::get()` 时同步。

`std::launch_async` 强制每次调用都创建一个新线程。

默认策略是 `std::launch::deferred | std::launch::async`，意味着具体实现决定创建新线程的策略。

Note: In the example `std::async` is launched with policy `std::launch_deferred`. This is to avoid a new thread being created in every call. In the case of our example, the calls to `std::async` are made out of order, they synchronize at the calls for `std::future::get()`.

`std::launch_async` forces a new thread to be created in every call.

The default policy is `std::launch::deferred | std::launch::async`, meaning the implementation determines the policy for creating new threads.

第89章：原子类型

第89.1节：多线程访问

原子类型可以安全地读取和写入两个线程共享的内存位置。

一个可能导致数据竞争的错误示例：

```
#include <thread>
#include <iostream>

//函数将把从'a'到'b'（包括两者）之间的所有值加到'result'中
void add(int a, int b, int * result) {
    for (int i = a; i <= b; i++) {
        *result += i;
    }
}

int main() {
    //原始数据类型没有线程安全性
    int shared = 0;

    //创建一个可能与'main'线程并行运行的线程
    //该线程将运行上面定义的函数'add'，参数为a = 1, b = 100, result =
    &shared
    //相当于'add(1,100, &shared);'
    std::thread addingThread(add, 1, 100, &shared);

    //尝试将'shared'的值打印到控制台
    //main将持续重复此操作，直到addingThread变为可连接状态
    while (!addingThread.joinable()) {
        //这可能导致未定义行为或打印出损坏的值
        //如果addingThread尝试写入'shared'，而主线程正在读取它
        std::cout << shared << std::endl;
    }

    //在执行结束时重新加入线程以进行清理
    addingThread.join();

    return 0;
}
```

上述示例可能导致读取损坏并引发未定义行为。

一个具有线程安全性的示例：

```
#include <atomic>
#include <thread>
#include <iostream>

//函数将把从'a'到'b'（包括两者）之间的所有值加到'result'中
void add(int a, int b, std::atomic<int> * result) {
    for (int i = a; i <= b; i++) {
        //以原子方式将'i'加到'result'中
        result->fetch_add(i);
    }
}
```

Chapter 89: Atomic Types

Section 89.1: Multi-threaded Access

An atomic type can be used to safely read and write to a memory location shared between two threads.

A Bad example that is likely to cause a data race:

```
#include <thread>
#include <iostream>

//function will add all values including and between 'a' and 'b' to 'result'
void add(int a, int b, int * result) {
    for (int i = a; i <= b; i++) {
        *result += i;
    }
}

int main() {
    //a primitive data type has no thread safety
    int shared = 0;

    //create a thread that may run parallel to the 'main' thread
    //the thread will run the function 'add' defined above with parameters a = 1, b = 100, result =
    &shared
    //analogous to 'add(1,100, &shared);'
    std::thread addingThread(add, 1, 100, &shared);

    //attempt to print the value of 'shared' to console
    //main will keep repeating this until the addingThread becomes joinable
    while (!addingThread.joinable()) {
        //this may cause undefined behavior or print a corrupted value
        //if the addingThread tries to write to 'shared' while the main thread is reading it
        std::cout << shared << std::endl;
    }

    //rejoin the thread at the end of execution for cleaning purposes
    addingThread.join();

    return 0;
}
```

The above example may cause a corrupted read and can lead to undefined behavior.

An example with thread safety:

```
#include <atomic>
#include <thread>
#include <iostream>

//function will add all values including and between 'a' and 'b' to 'result'
void add(int a, int b, std::atomic<int> * result) {
    for (int i = a; i <= b; i++) {
        //atomically add 'i' to result
        result->fetch_add(i);
    }
}
```

```

}

int main() {
    //使用atomic模板存储非原子对象
    std::atomic<int> shared = 0;

    //创建一个可能与'main'线程并行运行的线程
    //该线程将运行上面定义的函数'add'，参数为a = 1, b = 100, result =
    &shared
    //相当于'add(1,100, &shared);'
    std::thread addingThread(add, 1, 10000, &shared);

    //将'shared'的值打印到控制台
    //main将持续重复此操作，直到addingThread变为可join状态
    while (!addingThread.joinable()) {
        //以线程安全的方式原子读取shared的值
        std::cout << shared.load() << std::endl;
    }

    //在执行结束时重新加入线程以进行清理
    addingThread.join();

    return 0;
}

```

上述示例是安全的，因为所有store()和load()操作的atomic数据类型都保护了封装的int免受同时访问。

```

}

int main() {
    //atomic template used to store non-atomic objects
    std::atomic<int> shared = 0;

    //create a thread that may run parallel to the 'main' thread
    //the thread will run the function 'add' defined above with parameters a = 1, b = 100, result =
    &shared
    //analogous to 'add(1,100, &shared);'
    std::thread addingThread(add, 1, 10000, &shared);

    //print the value of 'shared' to console
    //main will keep repeating this until the addingThread becomes joinable
    while (!addingThread.joinable()) {
        //safe way to read the value of shared atomically for thread safe read
        std::cout << shared.load() << std::endl;
    }

    //rejoin the thread at the end of execution for cleaning purposes
    addingThread.join();

    return 0;
}

```

The above example is safe because all store() and load() operations of the atomic data type protect the encapsulated int from simultaneous access.

第90章：类型擦除

类型擦除是一组技术，用于创建一种类型，该类型可以为各种底层类型提供统一接口，同时向客户端隐藏底层类型信息。`std::function<R(A...)>`，能够持有各种类型的可调用对象，可能是C++中类型擦除最著名的例子。

第90.1节：仅可移动的`std::function`

`std::function`类型擦除归结为少数几个操作。它要求存储的值必须是可复制的。

这在某些情况下会引发问题，比如lambda存储`unique_ptr`。如果你在不需要复制的上下文中使用`std::function`，比如线程池中分发任务给线程，这个要求可能会增加开销。

特别是，`std::packaged_task<Sig>`是一个仅可移动的可调用对象。你可以将`std::packaged_task<R(Args...)>`存储在`std::packaged_task<void(Args...)>`中，但这是一种相当重量级且晦涩的方式来创建仅可移动的可调用类型擦除类。

因此这个任务。这展示了如何编写一个简单的`std::function`类型。我省略了拷贝构造函数（这将涉及向`details::task_pimpl<...>`中添加一个`clone`方法）。

```
template<class Sig>
struct task;

// 将其放入命名空间允许我们针对void返回值进行良好的特化：
namespace details {
    template<class R, class...Args>
    struct task_pimpl {
        virtual R invoke(Args&&...args) const = 0;
        virtual ~task_pimpl() {};
        virtual const std::type_info& target_type() const = 0;
    };

    // 存储一个F。 invoke(Args&&...) 调用f
    template<class F, class R, class...Args>
    struct task_pimpl_impl:task_pimpl<R,Args...> {
        F f;
        template<class Fin>
        task_pimplImpl( Fin&& fin ):f(std::forward<Fin>(fin)) {}
        virtual R invoke(Args&&...args) const final override {
            return f(std::forward<Args>(args)...);
        }
        virtual const std::type_info& target_type() const final override {
            return typeid(F);
        }
    };

    // void 版本丢弃了 f 的返回值：
    template<class F, class...Args>
    struct task_pimpl_impl<F,void,Args...>:task_pimpl<void,Args...> {
        F f;
        template<class Fin>
        task_pimplImpl( Fin&& fin ):f(std::forward<Fin>(fin)) {}
        virtual void invoke(Args&&...args) const final override {
            f(std::forward<Args>(args)...);
        }
    };
}
```

Chapter 90: Type Erasure

Type erasure is a set of techniques for creating a type that can provide a uniform interface to various underlying types, while hiding the underlying type information from the client. `std::function<R(A...)>`, which has the ability to hold callable objects of various types, is perhaps the best known example of type erasure in C++.

Section 90.1: A move-only `std::function`

`std::function` type erases down to a few operations. One of the things it requires is that the stored value be copyable.

This causes problems in a few contexts, like lambdas storing unique ptrs. If you are using the `std::function` in a context where copying doesn't matter, like a thread pool where you dispatch tasks to threads, this requirement can add overhead.

In particular, `std::packaged_task<Sig>` is a callable object that is move-only. You can store a `std::packaged_task<R(Args...)>` in a `std::packaged_task<void(Args...)>`, but that is a pretty heavy-weight and obscure way to create a move-only callable type-erasure class.

Thus the task. This demonstrates how you could write a simple `std::function` type. I omitted the copy constructor (which would involve adding a `clone` method to `details::task_pimpl<...>` as well).

```
template<class Sig>
struct task;

// putting it in a namespace allows us to specialize it nicely for void return value:
namespace details {
    template<class R, class...Args>
    struct task_pimpl {
        virtual R invoke(Args&&...args) const = 0;
        virtual ~task_pimpl() {};
        virtual const std::type_info& target_type() const = 0;
    };

    // store an F. invoke(Args&&...) calls the f
    template<class F, class R, class...Args>
    struct task_pimpl_impl:task_pimpl<R,Args...> {
        F f;
        template<class Fin>
        task_pimplImpl( Fin&& fin ):f(std::forward<Fin>(fin)) {}
        virtual R invoke(Args&&...args) const final override {
            return f(std::forward<Args>(args)...);
        }
        virtual const std::type_info& target_type() const final override {
            return typeid(F);
        }
    };

    // the void version discards the return value of f:
    template<class F, class...Args>
    struct task_pimpl_impl<F,void,Args...>:task_pimpl<void,Args...> {
        F f;
        template<class Fin>
        task_pimplImpl( Fin&& fin ):f(std::forward<Fin>(fin)) {}
        virtual void invoke(Args&&...args) const final override {
            f(std::forward<Args>(args)...);
        }
    };
}
```

```

virtual const std::type_info& target_type() const final override {
    return typeid(F);
}

};

template<class R, class...Args>
struct task<R(Args...)> {
    // 半正规：
    task()=default;
    task(task&&)=default;
    // 不允许拷贝

private:
    // 别名，用于使下面的一些 SFINAE 代码更简洁：
    template<class F>
    using call_r = std::result_of_t<F const&(Args...)>;
    template<class F>
    using is_task = std::is_same<std::decay_t<F>, task>;
public:
    // 可以从可调用对象 F 构造
    template<class F,
        // 可以用 Args... 调用并转换为 R 的类型：
        class= decltype( (R)(std::declval<call_r<F>>() ) ),
        // 并且不是同一类型：
        std::enable_if_t<!is_task<F>{}, int>* = nullptr
    >
    task(F&& f):
        m_pImpl( make_pimpl(std::forward<F>(f)) )
    {}

    // 核心：调用运算符
    R operator()(Args... args)const {
        return m_pImpl->invoke( std::forward<Args>(args)... );
    }
    explicit operator bool() const {
        return (bool)m_pImpl;
    }
    void swap( 任务& o ) {
        std::swap( m_pImpl, o.m_pImpl );
    }
    template<class F>
    void assign( F&& f ) {
        m_pImpl = make_pimpl(std::forward<F>(f));
    }
    // std::function 接口的一部分：
    const std::type_info& target_type() const {
        if (!*this) return typeid(void);
        return m_pImpl->target_type();
    }
    template< class T >
    T* target() {
        return target_impl<T>();
    }
    template< class T >
    const T* target() const {
        return target_impl<T>();
    }
    // 与 nullptr 比较 ：
    friend bool operator==( std::nullptr_t, task const& self ) { return !self; }
    friend bool operator==( task const& self, std::nullptr_t ) { return !self; }
    friend bool operator!=( std::nullptr_t, task const& self ) { return !!self; }

```

```

virtual const std::type_info& target_type() const final override {
    return typeid(F);
}
};

template<class R, class...Args>
struct task<R(Args...)> {
    // semi-regular:
    task()=default;
    task(task&&)=default;
    // no copy

private:
    // aliases to make some SFINAE code below less ugly:
    template<class F>
    using call_r = std::result_of_t<F const&(Args...)>;
    template<class F>
    using is_task = std::is_same<std::decay_t<F>, task>;
public:
    // can be constructed from a callable F
    template<class F,
        // that can be invoked with Args... and converted-to-R:
        class= decltype( (R)(std::declval<call_r<F>>() ) ),
        // and is not this same type:
        std::enable_if_t<!is_task<F>{}, int>* = nullptr
    >
    task(F&& f):
        m_pImpl( make_pimpl(std::forward<F>(f)) )
    {}

    // the meat: the call operator
    R operator()(Args... args)const {
        return m_pImpl->invoke( std::forward<Args>(args)... );
    }
    explicit operator bool() const {
        return (bool)m_pImpl;
    }
    void swap( task& o ) {
        std::swap( m_pImpl, o.m_pImpl );
    }
    template<class F>
    void assign( F&& f ) {
        m_pImpl = make_pimpl(std::forward<F>(f));
    }
    // Part of the std::function interface:
    const std::type_info& target_type() const {
        if (!*this) return typeid(void);
        return m_pImpl->target_type();
    }
    template< class T >
    T* target() {
        return target_impl<T>();
    }
    template< class T >
    const T* target() const {
        return target_impl<T>();
    }
    // compare with nullptr ：
    friend bool operator==( std::nullptr_t, task const& self ) { return !self; }
    friend bool operator==( task const& self, std::nullptr_t ) { return !self; }
    friend bool operator!=( std::nullptr_t, task const& self ) { return !!self; }

```

```

friend bool operator!=( task const& self, std::nullptr_t ) { return !!self; }
private:
    template<class T>
    using pimpl_t = details::task_pimpl<T, R, Args...>;
    template<class F>
    static auto make_pimpl( F&& f ) {
        using dF=std::decay_t<F>;
        using pImpl_t = pimpl_t<dF>;
        return std::make_unique<pImpl_t>(std::forward<F>(f));
    }
    std::unique_ptr<details::task_pimpl<R,Args...>> m_pImpl;
    template< class T >
    T* target_impl() const {
        return dynamic_cast<pimpl_t<T*>>(m_pImpl.get());
    }
};

```

为了使这个库更实用，你需要添加一个小型缓冲优化，这样它就不会将每个可调用对象都存储在堆上。

添加SBO需要一个非默认的ask (task&&)，类中包含一些std::aligned_storage_t，一个带有可设置为仅销毁（而不将内存返回堆）的删除器的m_pImpl unique_ptr，以及ask_pimpl中的
emplace_move_to(void*) = 0。

[上述代码的实际示例（无SBO）。](#)

第90.2节：使用手动vtable擦除为常规类型

C++依赖于所谓的常规类型（或至少是伪常规类型）。

常规类型是指可以通过拷贝或移动构造、赋值和销毁，并且可以进行相等比较的类型。它也可以无参数构造。最后，它还支持一些在各种std算法和容器中非常有用的操作。

[这是根本论文，但在C++11中会想要添加std::hash支持。](#)

我将在这里使用手动vtable方法进行类型擦除。

```

using dtor_unique_ptr = std::unique_ptr<void, void(*)(void*)>;
template<class T, class...Args>
dtor_unique_ptr make_dtor_unique_ptr( Args&&... args ) {
    return {new T(std::forward<Args>(args)...), [](void* self){ delete static_cast<T*>(self); }};
}
struct regular_vtable {
    void(*copy_assign)(void* 目标, void const* 源); // T&=(T const&)
    void(*move_assign)(void* 目标, void* 源); // T&=(T&&)
    bool(*equals)(void const* lhs, void const* rhs); // T const&==T const&
    bool(*order)(void const* lhs, void const* rhs); // std::less<T>{}(T const&, T const&)
    std::size_t(*hash)(void const* self); // std::hash<T>{}(T const&)
    std::type_info const&(*type)(); // typeid(T)
    dtor_unique_ptr(*clone)(void const* self); // T(T const&)
};

template<class T>
regular_vtable make_regular_vtable() noexcept {
    return {

```

```

friend bool operator!=( task const& self, std::nullptr_t ) { return !!self; }
private:
    template<class T>
    using pimpl_t = details::task_pimpl<T, R, Args...>;
    template<class F>
    static auto make_pimpl( F&& f ) {
        using dF=std::decay_t<F>;
        using pImpl_t = pimpl_t<dF>;
        return std::make_unique<pImpl_t>(std::forward<F>(f));
    }
    std::unique_ptr<details::task_pimpl<R,Args...>> m_pImpl;
    template< class T >
    T* target_impl() const {
        return dynamic_cast<pimpl_t<T*>>(m_pImpl.get());
    }
};

```

To make this library-worthy, you'd want to add in a small buffer optimization, so it does not store every callable on the heap.

Adding SBO would require a non-default task(task&&), some std::aligned_storage_t within the class, a m_pImpl unique_ptr with a deleter that can be set to destroy-only (and not return the memory to the heap), and a
emplace_move_to(void*) = 0 in the task_pimpl.

[live example of the above code \(with no SBO\).](#)

Section 90.2: Erasing down to a Regular type with manual vtable

C++ thrives on what is known as a Regular type (or at least Pseudo-Regular).

A Regular type is a type that can be constructed and assigned-to and assigned-from via copy or move, can be destroyed, and can be compared equal-to. It can also be constructed from no arguments. Finally, it also has support for a few other operations that are highly useful in various std algorithms and containers.

[This is the root paper, but in C++11 would want to add std::hash support.](#)

I will use the manual vtable approach to type erasure here.

```

using dtor_unique_ptr = std::unique_ptr<void, void(*)(void*)>;
template<class T, class...Args>
dtor_unique_ptr make_dtor_unique_ptr( Args&&... args ) {
    return {new T(std::forward<Args>(args)...), [](void* self){ delete static_cast<T*>(self); }};
}
struct regular_vtable {
    void(*copy_assign)(void* dest, void const* src); // T&=(T const&)
    void(*move_assign)(void* dest, void* src); // T&=(T&&)
    bool(*equals)(void const* lhs, void const* rhs); // T const&==T const&
    bool(*order)(void const* lhs, void const* rhs); // std::less<T>{}(T const&, T const&)
    std::size_t(*hash)(void const* self); // std::hash<T>{}(T const&)
    std::type_info const&(*type)(); // typeid(T)
    dtor_unique_ptr(*clone)(void const* self); // T(T const&)
};

template<class T>
regular_vtable make_regular_vtable() noexcept {
    return {

```

```

[](void* dest, void const* src){ *static_cast<T*>(dest) = *static_cast<T const*>(src); },
[](void* dest, void* src){ *static_cast<T*>(dest) = std::move(*static_cast<T*>(src)); },
[](void const* lhs, void const* rhs){ return *static_cast<T const*>(lhs) == *static_cast<T const*>(rhs); },
[](void const* lhs, void const* rhs) { return std::less<T>{}(*static_cast<T const*>(lhs),*static_cast<T const*>(rhs)); },
[](void const* self){ return std::hash<T>{}(*static_cast<T const*>(self)); },
[]()> decltype(auto){ return typeid(T); },
[](void const* self){ return make_dtor_unique_ptr<T>(*static_cast<T const*>(self)); }
};

template<class T>
regular_vtable const* get_regular_vtable() noexcept {
    static const regular_vtable vtable=make_regular_vtable<T>();
    return &vtable;
}

struct regular_type {
    using self=regular_type;
    regular_vtable const* vtable = 0;
    dtor_unique_ptr ptr{nullptr, [](void*){}};

    bool empty() const { return !vtable; }

    template<class T, class...Args>
    void emplace( Args&&... args ) {
        ptr = make_dtor_unique_ptr<T>(std::forward<Args>(args)...);
        if (ptr)
            vtable = get_regular_vtable<T>();
        else
            vtable = nullptr;
    }
    friend bool operator==(regular_type const& lhs, regular_type const& rhs) {
        if (lhs.vtable != rhs.vtable) return false;
        return lhs.vtable->equals( lhs.ptr.get(), rhs.ptr.get() );
    }
    bool before(regular_type const& rhs) const {
        auto const& lhs = *this;
        if (!lhs.vtable || !rhs.vtable)
            return std::less<regular_vtable const*>{}(lhs.vtable,rhs.vtable);
        if (lhs.vtable != rhs.vtable)
            return lhs.vtable->type().before(rhs.vtable->type());
        return lhs.vtable->order( lhs.ptr.get(), rhs.ptr.get() );
    }
    // 技术上还需要调用 before 的友元 bool operator<

    std::type_info const* type() const {
        if (!vtable) return nullptr;
        return &vtable->type();
    }
    regular_type(regular_type&& o):
        vtable(o.vtable),
        ptr(std::move(o.ptr))
    {
        o.vtable = nullptr;
    }
    friend void swap(regular_type& lhs, regular_type& rhs){
        std::swap(lhs.ptr, rhs.ptr);
        std::swap(lhs.vtable, rhs.vtable);
    }
    regular_type& operator=(regular_type&& o) {
        if (o.vtable == vtable) {

```

```

        [](void* dest, void const* src){ *static_cast<T const*>(dest) = *static_cast<T const*>(src); },
        [](void* dest, void* src){ *static_cast<T*>(dest) = std::move(*static_cast<T*>(src)); },
        [](void const* lhs, void const* rhs){ return *static_cast<T const*>(lhs) == *static_cast<T const*>(rhs); },
        [](void const* lhs, void const* rhs) { return std::less<T>{}(*static_cast<T const*>(lhs),*static_cast<T const*>(rhs)); },
        [](void const* self){ return std::hash<T>{}(*static_cast<T const*>(self)); },
        []()> decltype(auto){ return typeid(T); },
        [](void const* self){ return make_dtor_unique_ptr<T>(*static_cast<T const*>(self)); }
    }

    template<class T>
    regular_vtable const* get_regular_vtable() noexcept {
        static const regular_vtable vtable=make_regular_vtable<T>();
        return &vtable;
    }

    struct regular_type {
        using self=regular_type;
        regular_vtable const* vtable = 0;
        dtor_unique_ptr ptr{nullptr, [](void*){}};

        bool empty() const { return !vtable; }

        template<class T, class...Args>
        void emplace( Args&&... args ) {
            ptr = make_dtor_unique_ptr<T>(std::forward<Args>(args)...);
            if (ptr)
                vtable = get_regular_vtable<T>();
            else
                vtable = nullptr;
        }
        friend bool operator==(regular_type const& lhs, regular_type const& rhs) {
            if (lhs.vtable != rhs.vtable) return false;
            return lhs.vtable->equals( lhs.ptr.get(), rhs.ptr.get() );
        }
        bool before(regular_type const& rhs) const {
            auto const& lhs = *this;
            if (!lhs.vtable || !rhs.vtable)
                return std::less<regular_vtable const*>{}(lhs.vtable,rhs.vtable);
            if (lhs.vtable != rhs.vtable)
                return lhs.vtable->type().before(rhs.vtable->type());
            return lhs.vtable->order( lhs.ptr.get(), rhs.ptr.get() );
        }
        // technically friend bool operator< that calls before is also required

        std::type_info const* type() const {
            if (!vtable) return nullptr;
            return &vtable->type();
        }
        regular_type(regular_type&& o):
            vtable(o.vtable),
            ptr(std::move(o.ptr))
        {
            o.vtable = nullptr;
        }
        friend void swap(regular_type& lhs, regular_type& rhs){
            std::swap(lhs.ptr, rhs.ptr);
            std::swap(lhs.vtable, rhs.vtable);
        }
        regular_type& operator=(regular_type&& o) {
            if (o.vtable == vtable) {

```

```

vtable->move_assign(ptr.get(), o.ptr.get());
    return *this;
}
auto tmp = std::move(o);
swap(*this, tmp);
return *this;
}
regular_type(regular_type const& o):
vtable(o.vtable),
ptr(o.vtable?o.vtable->clone(o.ptr.get()):dtor_unique_ptr<nullptr, [](void*){}})
{
    if (!ptr && vtable) vtable = nullptr;
}
regular_type& operator=(regular_type const& o) {
    if (o.vtable == vtable) {
        vtable->copy_assign(ptr.get(), o.ptr.get());
        return *this;
    }
    auto tmp = o;
    swap(*this, tmp);
    return *this;
}
std::size_t hash() const {
    if (!vtable) return 0;
    return vtable->hash(ptr.get());
}
模板<类 T,
std::enable_if_t< !std::is_same<std::decay_t<T>, regular_type>{}, int>* =nullptr
>
regular_type(T&& t) {
emplace<std::decay_t<T>>(std::forward<T>(t));
}
};

namespace std {
    template<>
    struct hash<regular_type> {
        std::size_t operator()( regular_type const& r )const {
            return r.hash();
        }
    };
    template<>
    struct less<regular_type> {
        bool operator()( regular_type const& lhs, regular_type const& rhs ) const {
            return lhs.before(rhs);
        }
    };
}

```

[live example.](#)

这种常规类型可以用作接受“任何常规类型”作为键的std::map或std::unordered_map的键，例如：

```
std::map<regular_type, std::any>
```

基本上这将是一个从任何常规类型映射到任何可复制类型的映射。

与any不同，我的regular_type不进行小对象优化，也不支持取回原始数据。
取回原始类型并不难。

```

vtable->move_assign(ptr.get(), o.ptr.get());
    return *this;
}
auto tmp = std::move(o);
swap(*this, tmp);
return *this;
}
regular_type(regular_type const& o):
vtable(o.vtable),
ptr(o.vtable?o.vtable->clone(o.ptr.get()):dtor_unique_ptr<nullptr, [](void*){}))
{
    if (!ptr && vtable) vtable = nullptr;
}
regular_type& operator=(regular_type const& o) {
    if (o.vtable == vtable) {
        vtable->copy_assign(ptr.get(), o.ptr.get());
        return *this;
    }
    auto tmp = o;
    swap(*this, tmp);
    return *this;
}
std::size_t hash() const {
    if (!vtable) return 0;
    return vtable->hash(ptr.get());
}
template<class T,
std::enable_if_t< !std::is_same<std::decay_t<T>, regular_type>{}, int>* =nullptr
>
regular_type(T&& t) {
emplace<std::decay_t<T>>(std::forward<T>(t));
}
};

namespace std {
    template<>
    struct hash<regular_type> {
        std::size_t operator()( regular_type const& r )const {
            return r.hash();
        }
    };
    template<>
    struct less<regular_type> {
        bool operator()( regular_type const& lhs, regular_type const& rhs ) const {
            return lhs.before(rhs);
        }
    };
}

```

[live example.](#)

Such a regular type can be used as a key for a std::map or a std::unordered_map that accepts *anything regular* for a key, like:

```
std::map<regular_type, std::any>
```

would be basically a map from anything regular, to anything copyable.

Unlike any, my regular_type does no small object optimization nor does it support getting the original data back.
Getting the original type back isn't hard.

小对象优化要求我们在regular_type中存储一个对齐的存储缓冲区，并且小心调整ptr的删除器，只销毁对象而不删除它。

我会从make_dtor_unique_ptr开始，教它如何有时将数据存储在缓冲区中，如果缓冲区没有空间则存储在堆中。这可能就足够了。

第90.3节：基本机制

类型擦除是一种隐藏对象类型的方式，即使该对象不是从公共基类派生的。通过这样做，它在静态多态（模板；在使用处必须在编译时知道确切类型，但定义时不必声明符合接口）和动态多态（继承和虚函数；在使用处不必在编译时知道确切类型，但定义时必须声明符合接口）之间架起了一座桥梁。

以下代码展示了类型擦除的基本机制。

```
#include <iostream>

class Printable
{
public:
    template <typename T>
    Printable(T value) : pValue(new Value<T>(value)) {}
    ~Printable() { delete pValue; }
    void print(std::ostream &os) const { pValue->print(os); }

private:
    Printable(Printable const &) /* in C++1x: =delete */; // not implemented
    void operator = (Printable const &) /* in C++1x: =delete */; // not implemented
    struct ValueBase
    {
        virtual ~ValueBase() = default;
        virtual void print(std::ostream &) const = 0;
    };
    template <typename T>
    struct Value : ValueBase
    {
        Value(T const &t) : v(t) {}
        virtual void print(std::ostream &os) const { os << v; }
        T v;
    };
    ValueBase *pValue;
};
```

在使用处，只需可见上述定义，就像具有虚函数的基类一样。例如：

```
#include <iostream>

void print_value(Printable const &p)
{
    p.print(std::cout);
}
```

请注意，这不是模板，而是一个普通函数，只需在头文件中声明，且可以在实现文件中定义（与模板不同，模板的定义必须在使用处可见）。

在具体类型的定义中，不需要了解Printable，只需符合一个

Small object optimization requires that we store an aligned storage buffer within the regular_type, and carefully tweak the deleter of the ptr to only destroy the object and not delete it.

I would start at make_dtor_unique_ptr and teach it how to sometimes store the data in a buffer, and then in the heap if no room in the buffer. That may be sufficient.

Section 90.3: Basic mechanism

Type erasure is a way to hide the type of an object from code using it, even though it is not derived from a common base class. In doing so, it provides a bridge between the worlds of static polymorphism (templates; at the place of use, the exact type must be known at compile time, but it need not be declared to conform to an interface at definition) and dynamic polymorphism (inheritance and virtual functions; at the place of use, the exact type need not be known at compile time, but must be declared to conform to an interface at definition).

The following code shows the basic mechanism of type erasure.

```
#include <iostream>

class Printable
{
public:
    template <typename T>
    Printable(T value) : pValue(new Value<T>(value)) {}
    ~Printable() { delete pValue; }
    void print(std::ostream &os) const { pValue->print(os); }

private:
    Printable(Printable const &) /* in C++1x: =delete */; // not implemented
    void operator = (Printable const &) /* in C++1x: =delete */; // not implemented
    struct ValueBase
    {
        virtual ~ValueBase() = default;
        virtual void print(std::ostream &) const = 0;
    };
    template <typename T>
    struct Value : ValueBase
    {
        Value(T const &t) : v(t) {}
        virtual void print(std::ostream &os) const { os << v; }
        T v;
    };
    ValueBase *pValue;
};
```

At the use site, only the above definition need to be visible, just as with base classes with virtual functions. For example:

```
#include <iostream>

void print_value(Printable const &p)
{
    p.print(std::cout);
}
```

Note that this is *not* a template, but a normal function that only needs to be declared in a header file, and can be defined in an implementation file (unlike templates, whose definition must be visible at the place of use).

At the definition of the concrete type, nothing needs to be known about Printable, it just needs to conform to an

接口，就像模板一样：

```
struct MyType { int i; };
ostream& operator << (ostream &os, MyType const &mc)
{
    return os << "MyType {" << mc.i << "}";
}
```

现在我们可以将该类的对象传递给上面定义的函数：

```
MyType foo = { 42 };
print_value(foo);
```

第90.4节：擦除至连续的T缓冲区

并非所有类型擦除都涉及虚拟继承、内存分配、定位 new，甚至函数指针。

类型擦除之所以称为类型擦除，是因为它描述了一组行为，并接受任何支持该行为的类型并将其封装起来。所有不在该行为集合中的信息都会被“遗忘”或“擦除”。

一个array_view接受其传入的范围或容器类型，并擦除除它是一个连续的T缓冲区这一事实之外的所有内容。

```
// 用于 SFINAE 的辅助特性：
template<class T>
using data_t = decltype( std::declval<T>().data() );

template<class Src, class T>
using compatible_data = std::integral_constant<bool, std::is_same< data_t<Src>, T* >{} || std::is_same< data_t<Src>, std::remove_const_t<T>*>{}>;

template<class T>
struct array_view {
    // 类的核心：
    T* b=nullptr;
    T* e=nullptr;
    T* begin() const { return b; }
    T* end() const { return e; }

    // 提供良好连续区间的预期方法：
    T* data() const { return begin(); }
    bool empty() const { return begin()==end(); }
    std::size_t size() const { return end()-begin(); }

    T& operator[](std::size_t i) const{ return begin()[i]; }
    T& front() const{ return *begin(); }
    T& back() const{ return *(end()-1); }

    // 有用的辅助函数，允许你快速且安全地从此范围生成其他范围：
    array_view without_front( std::size_t i=1 ) const {
        i = (std::min)(i, size());
        return {begin()+i, end()};
    }
    array_view without_back( std::size_t i=1 ) const {
        i = (std::min)(i, size());
        return {begin(), end()-i};
    }
}
```

// 有用的辅助函数，允许你快速且安全地从此范围生成其他范围：

```
array_view without_front( std::size_t i=1 ) const {
    i = (std::min)(i, size());
    return {begin()+i, end()};
}
array_view without_back( std::size_t i=1 ) const {
    i = (std::min)(i, size());
    return {begin(), end()-i};
}
```

interface, as with templates:

```
struct MyType { int i; };
ostream& operator << (ostream &os, MyType const &mc)
{
    return os << "MyType {" << mc.i << "}";
}
```

We can now pass an object of this class to the function defined above:

```
MyType foo = { 42 };
print_value(foo);
```

Section 90.4: Erasing down to a contiguous buffer of T

Not all type erasure involves virtual inheritance, allocations, placement new, or even function pointers.

What makes type erasure type erasure is that it describes a (set of) behavior(s), and takes any type that supports that behavior and wraps it up. All information that isn't in that set of behaviors is "forgotten" or "erased".

An array_view takes its incoming range or container type and erases everything except the fact it is a contiguous buffer of T.

```
// helper traits for SFINAE:
template<class T>
using data_t = decltype( std::declval<T>().data() );

template<class Src, class T>
using compatible_data = std::integral_constant<bool, std::is_same< data_t<Src>, T* >{} || std::is_same< data_t<Src>, std::remove_const_t<T>*>{}>

template<class T>
struct array_view {
    // the core of the class:
    T* b=nullptr;
    T* e=nullptr;
    T* begin() const { return b; }
    T* end() const { return e; }

    // provide the expected methods of a good contiguous range:
    T* data() const { return begin(); }
    bool empty() const { return begin()==end(); }
    std::size_t size() const { return end()-begin(); }

    T& operator[](std::size_t i) const{ return begin()[i]; }
    T& front() const{ return *begin(); }
    T& back() const{ return *(end()-1); }

    // useful helpers that let you generate other ranges from this one
    // quickly and safely:
    array_view without_front( std::size_t i=1 ) const {
        i = (std::min)(i, size());
        return {begin()+i, end()};
    }
    array_view without_back( std::size_t i=1 ) const {
        i = (std::min)(i, size());
        return {begin(), end()-i};
    }
}
```

```

// array_view 是普通旧数据，因此默认拷贝：
array_view(array_view const&)=default;
// 生成一个空的、空范围：
array_view()=default;

// 最终构造函数：
array_view(T* s, T* f):b(s),e(f) {}
// 在我的经验中，起始和长度是有用的：
array_view(T* s, std::size_t length):array_view(s, s+length) {}

// SFINAЕ 构造函数，一次性接受任何支持 .data() 的容器
// 或其他范围：
template<class Src,
std::enable_if_t< compatible_data<std::remove_reference_t<Src>&, T >{}, int>* =nullptr,
std::enable_if_t< !std::is_same<std::decay_t<Src>, array_view >{}, int>* =nullptr
>
array_view( Src&& src ):
array_view( src.data(), src.size() )
{}

// 数组构造函数：
template<std::size_t N>
array_view( T(&arr)[N] ):array_view(arr, N) {}

// 初始化列表，支持基于 {} 的初始化：
template<class U,
std::enable_if_t< std::is_same<const U, T>{}, int>* =nullptr
>
array_view( std::initializer_list<U> il ):array_view(il.begin(), il.end()) {}

```

array_view 接受任何支持 .data() 返回指向 T 的指针且有 .size() 方法的容器，或者数组，并将其简化为对连续 T 元素的随机访问范围。

它可以接受 std::vector<T>、std::string<T>、std::array<T, N>、T[37]、初始化列表（包括基于 {} 的），或者你自定义的支持该接口的类型（通过 T* x.data() 和 size_t x.size()）。

在这种情况下，我们可以从被简化的对象中提取数据，加上我们的“视图”非拥有状态，意味着我们不必分配内存或编写自定义的类型相关函数。

实时示例。

一个改进是使用非成员的 data 和非成员的 size，在支持 ADL 的上下文中调用。

第90.5节：使用 std::any 进行类型擦除

此示例使用了 C++14 和 boost::any。在 C++17 中，你可以改用 std::any。

我们最终得到的语法是：

```

const auto print =
make_any_method<void(std::ostream&)>([](auto&& p, std::ostream& t){ t << p << ""; });
super_any<decltype(p
rint)> a = 7;
(a->*print)(std::cout);

```

这几乎是最优的。

```

// array_view is plain old data, so default copy:
array_view(array_view const&)=default;
// generates a null, empty range:
array_view()=default;

// final constructor:
array_view(T* s, T* f):b(s),e(f) {}
// start and length is useful in my experience:
array_view(T* s, std::size_t length):array_view(s, s+length) {}

// SFINAЕ constructor that takes any .data() supporting container
// or other range in one fell swoop:
template<class Src,
std::enable_if_t< compatible_data<std::remove_reference_t<Src>&, T >{}, int>* =nullptr,
std::enable_if_t< !std::is_same<std::decay_t<Src>, array_view >{}, int>* =nullptr
>
array_view( Src&& src ):
array_view( src.data(), src.size() )
{};

// array constructor:
template<std::size_t N>
array_view( T(&arr)[N] ):array_view(arr, N) {}

// initializer list, allowing {} based:
template<class U,
std::enable_if_t< std::is_same<const U, T>{}, int>* =nullptr
>
array_view( std::initializer_list<U> il ):array_view(il.begin(), il.end()) {};

```

an array_view takes any container that supports .data() returning a pointer to T and a .size() method, or an array, and erases it down to being a random-access range over contiguous Ts.

It can take a std::vector<T>, a std::string<T> a std::array<T, N> a T[37], an initializer list (including {} based ones), or something else you make up that supports it (via T* x.data() and size_t x.size()).

In this case, the data we can extract from the thing we are erasing, together with our "view" non-owning state, means we don't have to allocate memory or write custom type-dependent functions.

Live example.

An improvement would be to use a non-member data and a non-member size in an ADL-enabled context.

Section 90.5: Type erasing type erasure with std::any

This example uses C++14 and boost::any. In C++17 you can swap in std::any instead.

The syntax we end up with is:

```

const auto print =
make_any_method<void(std::ostream&)>([](auto&& p, std::ostream& t){ t << p << "\n"; });
super_any<decltype(print)> a = 7;
(a->*print)(std::cout);

```

which is almost optimal.

此示例基于 @dyp 和 @cpplearner 的工作以及我自己的工作。

首先我们使用一个标签来传递类型：

```
template<class T>struct tag_t{constexpr tag_t(){};};
template<class T>constexpr tag_t<T> tag{};
```

该特征类获取与any_method存储的签名：

这会创建一个函数指针类型，以及给定any_method的该函数指针的工厂：

```
template<class any_method>
using any_sig_from_method = typename any_method::signature;

template<class any_method, class Sig=any_sig_from_method<any_method>>
struct any_method_function;

template<class any_method, class R, class...Args>
struct any_method_function<any_method, R(Args...)>
{
    template<class T>
    using decorate = std::conditional_t< any_method::is_const, T const, T >;
    
    using any = decorate<boost::any>;
    
    using type = R(*)(any&, any_method const*, Args&&...);
    template<class T>
    type operator()( tag_t<T> )const{
        return +[](any& self, any_method const* method, Args&&...args) {
            return (*method)( boost::any_cast<decorate<T>&>(self), decltype(args)(args)... );
        };
    }
};
```

any_method_function::type 是我们将与实例一起存储的函数指针类型。

any_method_function::operator() 接受一个 tag_t<T> 并写入一个自定义的 any_method_function::type，该类型假设 any& 将是一个 T。

我们希望能够一次性进行多种方法的类型擦除。因此，我们将它们打包成一个元组，并编写一个辅助包装器，将该元组按类型存储在静态存储中，并维护一个指向它们的指针。

```
template<class...any_methods>
using any_method_tuple = std::tuple< typename any_method_function<any_methods>::type... >;

template<class...any_methods, class T>
any_method_tuple<any_methods...> make_vtable( tag_t<T> ) {
    return std::make_tuple(
        any_method_function<any_methods>{}(tag<T>)... );
}

template<class...methods>
struct any_methods {
private:
    any_method_tuple<methods...> const* vtable = 0;
    template<class T>
    static any_method_tuple<methods...> const* get_vtable( tag_t<T> ) {
        static const auto table = make_vtable<methods...>(tag<T>);
        return &table;
    }
}
```

This example is based off of work by [@dyp](#) and [@cpplearner](#) as well as my own.

First we use a tag to pass around types:

```
template<class T>struct tag_t{constexpr tag_t(){};};
template<class T>constexpr tag_t<T> tag{};
```

This trait class gets the signature stored with an any_method:

```
template<class any_method>
using any_sig_from_method = typename any_method::signature;

template<class any_method, class Sig=any_sig_from_method<any_method>>
struct any_method_function;

template<class any_method, class R, class...Args>
struct any_method_function<any_method, R(Args...)>
{
    template<class T>
    using decorate = std::conditional_t< any_method::is_const, T const, T >;
    
    using any = decorate<boost::any>;
    
    using type = R(*)(any&, any_method const*, Args&&...);
    template<class T>
    type operator()( tag_t<T> )const{
        return +[](any& self, any_method const* method, Args&&...args) {
            return (*method)( boost::any_cast<decorate<T>&>(self), decltype(args)(args)... );
        };
    }
};
```

any_method_function::type 是我们将与实例一起存储的函数指针类型。

any_method_function::operator() 接受一个 tag_t<T> 并写入一个自定义的 any_method_function::type，该类型假设 any& 将是一个 T。

We want to be able to type-erase more than one method at a time. So we bundle them up in a tuple, and write a helper wrapper to stick the tuple into static storage on a per-type basis and maintain a pointer to them.

```
template<class...any_methods>
using any_method_tuple = std::tuple< typename any_method_function<any_methods>::type... >;

template<class...any_methods, class T>
any_method_tuple<any_methods...> make_vtable( tag_t<T> ) {
    return std::make_tuple(
        any_method_function<any_methods>{}(tag<T>)... );
}

template<class...methods>
struct any_methods {
private:
    any_method_tuple<methods...> const* vtable = 0;
    template<class T>
    static any_method_tuple<methods...> const* get_vtable( tag_t<T> ) {
        static const auto table = make_vtable<methods...>(tag<T>);
        return &table;
    }
}
```

```

    }
公共:
any_methods() = default;
template<class T>
any_methods( tag_t<T> ): vtable(get_vtable(tag<T>)) {}
any_methods& operator=(any_methods const&)=default;
template<class T>
void change_type( tag_t<T> ={} ) { vtable = get_vtable(tag<T>); }

template<class any_method>
auto get_invoker( tag_t<any_method> ={} ) const {
    return std::get<typename any_method_function<any_method>::type>( *vtable );
}
};


```

我们可以针对 vtable 较小的情况（例如，只有 1 个条目）进行特化，并在这些情况下使用存储在类内的直接指针以提高效率。

现在我们开始介绍 super_any。我使用 super_any_t 来使 super_any 的声明更简单一些。

```
template<class...methods>
struct super_any_t;
```

这会搜索 super_any 支持的方法，以便进行 SFINAE 和提供更好的错误信息：

```

template<class super_any, class method>
struct super_method_applies_helper : std::false_type {};

template<class M0, class...Methods, class method>
struct super_method_applies_helper<super_any_t<M0, Methods...>, method> :
std::integral_constant<bool, std::is_same<M0, method>{} || super_method_applies_helper<super_any_t<Methods...>, method>{}> {};

template<class...methods, class method>
auto super_method_test( super_any_t<methods...> const&, tag_t<method> )
{
    return std::integral_constant<bool, super_method_applies_helper<super_any_t<methods...>, method>{} && method::is_const >{};
}
template<class...methods, class method>
auto super_method_test( super_any_t<methods...>&, tag_t<method> )
{
    return std::integral_constant<bool, super_method_applies_helper<super_any_t<methods...>, method>{} >{};
}

template<class super_any, class method>
struct super_method_applies:
    decltype( super_method_test( std::declval<super_any>(), tag<method> ) )
{};


```

接下来我们创建 any_method 类型。any_method 是一种伪方法指针。我们全局创建它，并且 const 地使用如下语法：

```
const auto print=make_any_method( [](auto&&self, auto&&os){ os << self; } );
```

或者在 C++17 中：

```

    }
public:
any_methods() = default;
template<class T>
any_methods( tag_t<T> ): vtable(get_vtable(tag<T>)) {}
any_methods& operator=(any_methods const&)=default;
template<class T>
void change_type( tag_t<T> ={} ) { vtable = get_vtable(tag<T>); }

template<class any_method>
auto get_invoker( tag_t<any_method> ={} ) const {
    return std::get<typename any_method_function<any_method>::type>( *vtable );
}
};


```

We could specialize this for cases where the vtable is small (for example, 1 item), and use direct pointers stored in-class in those cases for efficiency.

Now we start the super_any. I use super_any_t to make the declaration of super_any a bit easier.

```
template<class...methods>
struct super_any_t;
```

This searches the methods that the super any supports for SFINAE and better error messages:

```

template<class super_any, class method>
struct super_method_applies_helper : std::false_type {};

template<class M0, class...Methods, class method>
struct super_method_applies_helper<super_any_t<M0, Methods...>, method> :
std::integral_constant<bool, std::is_same<M0, method>{} || super_method_applies_helper<super_any_t<Methods...>, method>{}> {};

template<class...methods, class method>
auto super_method_test( super_any_t<methods...> const&, tag_t<method> )
{
    return std::integral_constant<bool, super_method_applies_helper<super_any_t<methods...>, method>{} && method::is_const >{};
}
template<class...methods, class method>
auto super_method_test( super_any_t<methods...>&, tag_t<method> )
{
    return std::integral_constant<bool, super_method_applies_helper<super_any_t<methods...>, method>{} >{};
}

template<class super_any, class method>
struct super_method_applies:
    decltype( super_method_test( std::declval<super_any>(), tag<method> ) )
{};


```

Next we create the any_method type. An any_method is a pseudo-method-pointer. We create it globally and `constly` using syntax like:

```
const auto print=make_any_method( [](auto&&self, auto&&os){ os << self; } );
```

or in C++17:

```
const any_method print=[](auto&&self, auto&&os){ os << self; };
```

请注意，使用非lambda可能会使事情变得复杂，因为我们使用该类型进行查找步骤。这可以修复，但会使这个示例比现在更长。因此，始终从lambda初始化any方法，或者从以lambda为模板参数的类型初始化。

```
template<class Sig, bool const_method, class F>
struct any_method {
    using signature=Sig;
    enum{is_const=const_method};
private:
    F f;
public:

    template<class Any,
             // SFINAE测试其中一个Any是否匹配此类型：
std::enable_if_t< super_method_applies< Any&&, any_method >{}, int>* =nullptr
    >
    friend auto operator->*( Any&& self, any_method const& m ) {
        // 我们不使用any_method的值，因为每个any_method都有
        // 唯一的类型(!)，并且我们检查super_any中的某个auto*已经
        // 拥有指向我们的指针。然后我们调度到对应的
        // any_method_data...
        return [&self, invoke = self.get_invoker(tag<any_method>), m](auto&&...args)->decltype(auto)
    {
        return invoke( decltype(self)(self), &m, decltype(args)(args)... );
    };
}
any_method( F fin ):f(std::move(fin)) {}

template<class...Args>
decltype(auto) operator()(Args&&...args)const {
    return f(std::forward<Args>(args)...);
}
};
```

我认为这是一个工厂方法，在C++17中不需要：

```
template<class Sig, bool is_const=false, class F>
any_method<Sig, is_const, std::decay_t<F>>
make_any_method( F&& f ) {
    return {std::forward<F>(f)};
}
```

这是增强版的any。它既是一个any，同时携带一组类型擦除函数指针，这些指针会随着所包含的any的变化而变化：

```
template<class... methods>
struct super_any_t:boost::any, any_methods<methods...> {
    using vtable=any_methods<methods...>;
public:
    模板<类 T,
std::enable_if_t< !std::is_base_of<super_any_t, std::decay_t<T>>{}, int> =0
    >
super_any_t( T&& t ):
boost::any( std::forward<T>(t) )
{
    using dT=std::decay_t<T>;
```

```
const any_method print=[](auto&&self, auto&&os){ os << self; };
```

Note that using a non-lambda can make things hairy, as we use the type for a lookup step. This can be fixed, but would make this example longer than it already is. So always initialize an any method from a lambda, or from a type parameterized on a lambda.

```
template<class Sig, bool const_method, class F>
struct any_method {
    using signature=Sig;
    enum{is_const=const_method};
private:
    F f;
public:

    template<class Any,
             // SFINAE testing that one of the Anys's matches this type:
std::enable_if_t< super_method_applies< Any&&, any_method >{}, int>* =nullptr
    >
    friend auto operator->*( Any&& self, any_method const& m ) {
        // we don't use the value of the any_method, because each any_method has
        // a unique type (!) and we check that one of the auto*'s in the super_any
        // already has a pointer to us. We then dispatch to the corresponding
        // any_method_data...
        return [&self, invoke = self.get_invoker(tag<any_method>), m](auto&&...args)->decltype(auto)
    {
        return invoke( decltype(self)(self), &m, decltype(args)(args)... );
    };
}
any_method( F fin ):f(std::move(fin)) {}

template<class...Args>
decltype(auto) operator()(Args&&...args)const {
    return f(std::forward<Args>(args)...);
}
};
```

A factory method, not needed in C++17 I believe:

```
template<class Sig, bool is_const=false, class F>
any_method<Sig, is_const, std::decay_t<F>>
make_any_method( F&& f ) {
    return {std::forward<F>(f)};
}
```

This is the augmented any. It is both an any, and it carries around a bundle of type-erasure function pointers that change whenever the contained any does:

```
template<class... methods>
struct super_any_t:boost::any, any_methods<methods...> {
    using vtable=any_methods<methods...>;
public:
    template<class T,
std::enable_if_t< !std::is_base_of<super_any_t, std::decay_t<T>>{}, int> =0
    >
super_any_t( T&& t ):
boost::any( std::forward<T>(t) )
{
    using dT=std::decay_t<T>;
```

```

this->change_type( tag<dT> );
}

boost::any& as_any()&{return *this;}
boost::any&& as_any()&&{return std::move(*this);}
boost::any const& as_any()const&{return *this;}
super_any_t()=default;
super_any_t(super_any_t&& o):
boost::any( std::move( o.as_any() ) ),
vtable(o)
{}
super_any_t(super_any_t const& o):
boost::any( o.as_any() ),
vtable(o)
{}
template<class S,
std::enable_if_t< std::is_same<std::decay_t<S>, super_any_t>{}, int> =0
>
super_any_t( S&& o ):
boost::any( std::forward<S>(o).as_any() ),
vtable(o)
{}
super_any_t& operator=(super_any_t&&)=default;
super_any_t& operator=(super_any_t const&)=default;

模板<类 T,
std::enable_if_t< !std::is_same<std::decay_t<T>, super_any_t>{}, int>*& =nullptr
>
super_any_t& operator=( T&& t ) {
((boost::any*)&this) = std::forward<T>(t);
using dT=std::decay_t<T>;
this->change_type( tag<dT> );
return *this;
}
};

```

因为我们将any_method存储为const对象，这使得创建一个super_any变得更容易：

```
template<class...Ts>
using super_any = super_any_t< std::remove_cv_t<Ts>... >;
```

测试代码：

```

const auto print = make_any_method<void(std::ostream&)>([](auto&& p, std::ostream& t){ t << p << "" });
const auto wprint = make_any_method<void(std::wostream&)>([](auto&& p, std::wostream& os ){ os << p << L"" });

int main()
{
super_any<decltype(print), decltype(wprint)> a = 7;
super_any<decltype(print), decltype(wprint)> a2 = 7;

(a->*print)(std::cout);
(a->*wprint)(std::wcout);
}
```

[live example](#).

最初发布于Stack Overflow的自问自答（上述人员帮助实现）。

```

this->change_type( tag<dT> );
}

boost::any& as_any()&{return *this;}
boost::any&& as_any()&&{return std::move(*this);}
boost::any const& as_any()const&{return *this;}
super_any_t()=default;
super_any_t(super_any_t&& o):
boost::any( std::move( o.as_any() ) ),
vtable(o)
{}
super_any_t(super_any_t const& o):
boost::any( o.as_any() ),
vtable(o)
{}
template<class S,
std::enable_if_t< std::is_same<std::decay_t<S>, super_any_t>{}, int> =0
>
super_any_t( S&& o ):
boost::any( std::forward<S>(o).as_any() ),
vtable(o)
{}
super_any_t& operator=(super_any_t&&)=default;
super_any_t& operator=(super_any_t const&)=default;

template<class T,
std::enable_if_t< !std::is_same<std::decay_t<T>, super_any_t>{}, int>*& =nullptr
>
super_any_t& operator=( T&& t ) {
((boost::any*)&this) = std::forward<T>(t);
using dT=std::decay_t<T>;
this->change_type( tag<dT> );
return *this;
}
};
```

Because we store the any_methods as `const` objects, this makes making a `super_any` a bit easier:

```
template<class...Ts>
using super_any = super_any_t< std::remove_cv_t<Ts>... >;
```

Test code:

```

const auto print = make_any_method<void(std::ostream&)>([](auto&& p, std::ostream& t){ t << p <<
"\n"; });
const auto wprint = make_any_method<void(std::wostream&)>([](auto&& p, std::wostream& os ){ os << p
<< L"\n"; });

int main()
{
super_any<decltype(print), decltype(wprint)> a = 7;
super_any<decltype(print), decltype(wprint)> a2 = 7;

(a->*print)(std::cout);
(a->*wprint)(std::wcout);
}
```

[live example](#).

Originally posted [here](#) in a SO self question & answer (and people noted above helped with the implementation).

第91章：显式类型转换

表达式可以使用 `dynamic_cast< T >`、`static_cast< T >`、`reinterpret_cast< T >` 或 `const_cast< T >` 显式转换或强制转换为类型 `T`，具体取决于所需的转换类型。

C++ 也支持函数风格的类型转换表示法，`T(expr)`，以及 C 风格的类型转换表示法，`(T)expr`。

第 91.1 节：C 风格类型转换

C 风格类型转换可以被视为“尽力而为”的转换，之所以如此命名，是因为它是唯一可以在 C 语言中使用的转换。该转换的语法是 `(NewType)variable`。

每当使用此转换时，它会依次使用以下 C++ 转换之一：

- `const_cast<NewType>(variable)`
- `static_cast<NewType>(variable)`
- `const_cast<NewType>(static_cast<const NewType>(variable))`
- `reinterpret_cast<const NewType>(variable)`
- `const_cast<NewType>(reinterpret_cast<const NewType>(variable))`

函数风格的类型转换非常相似，但由于其语法的限制：`NewType(expression)`，因此只能转换没有空格的类型。

最好使用新的 C++ 类型转换，因为它更易读，可以在 C++ 源代码中的任何地方轻松识别，并且错误会在编译时被检测，而不是在运行时。

由于此转换可能导致意外的 `reinterpret_cast`，因此通常被认为是危险的。

第 91.2 节：去除 const 属性的类型转换

指向常量对象的指针可以使用 `const_cast` 关键字转换为指向非常量对象的指针。这里我们使用 `const_cast` 来调用一个不是 `const` 正确的函数。该函数只接受一个非常量的 `char*` 参数，尽管它从不通过该指针写入数据：

```
void bad_strlen(char*);  
const char* s = "hello, world!";  
bad_strlen(s); // 编译错误  
bad_strlen(const_cast<char*>(s)); // 可以，但最好让bad_strlen接受const char*
```

`const_cast` 用于引用类型时，可以将 `const` 限定的左值转换为非 `const` 限定的值。

`const_cast` 很危险，因为它使 C++ 类型系统无法阻止你尝试修改 `const` 对象。这样做会导致未定义行为。

```
const int x = 123;  
int& mutable_x = const_cast<int&>(x);  
mutable_x = 456; // 可能编译通过，但会产生*未定义行为*
```

第91.3节：基类到派生类的转换

指向基类的指针可以使用 `static_cast` 转换为指向派生类的指针。`static_cast` 不会进行任何运行时检查，当指针实际上并不指向所需类型时，可能导致未定义行为。

Chapter 91: Explicit type conversions

An expression can be *explicitly converted* or *cast* to type `T` using `dynamic_cast<T>`, `static_cast<T>`, `reinterpret_cast<T>`, or `const_cast<T>`, depending on what type of cast is intended.

C++ also supports function-style cast notation, `T(expr)`, and C-style cast notation, `(T)expr`.

Section 91.1: C-style casting

C-Style casting can be considered 'Best effort' casting and is named so as it is the only cast which could be used in C. The syntax for this cast is `(NewType)variable`.

Whenever this cast is used, it uses one of the following c++ casts (in order):

- `const_cast<NewType>(variable)`
- `static_cast<NewType>(variable)`
- `const_cast<NewType>(static_cast<const NewType>(variable))`
- `reinterpret_cast<const NewType>(variable)`
- `const_cast<NewType>(reinterpret_cast<const NewType>(variable))`

Functional casting is very similar, though as a few restrictions as the result of its syntax: `NewType(expression)`. As a result, only types without spaces can be cast to.

It's better to use new c++ cast, because it's more readable and can be spotted easily anywhere inside a C++ source code and errors will be detected in compile-time, instead in run-time.

As this cast can result in unintended `reinterpret_cast`, it is often considered dangerous.

Section 91.2: Casting away constness

A pointer to a `const` object can be converted to a pointer to non-`const` object using the `const_cast` keyword. Here we use `const_cast` to call a function that is not `const`-correct. It only accepts a non-`const` `char*` argument even though it never writes through the pointer:

```
void bad_strlen(char*);  
const char* s = "hello, world!";  
bad_strlen(s); // compile error  
bad_strlen(const_cast<char*>(s)); // OK, but it's better to make bad_strlen accept const char*
```

`const_cast` to reference type can be used to convert a `const`-qualified lvalue into a non-`const`-qualified value.

`const_cast` is dangerous because it makes it impossible for the C++ type system to prevent you from trying to modify a `const` object. Doing so results in undefined behavior.

```
const int x = 123;  
int& mutable_x = const_cast<int&>(x);  
mutable_x = 456; // may compile, but produces *undefined behavior*
```

Section 91.3: Base to derived conversion

A pointer to base class can be converted to a pointer to derived class using `static_cast`. `static_cast` does not do any run-time checking and can lead to undefined behaviour when the pointer does not actually point to the desired type.

```

结构体 Base {};
结构体 Derived : Base {};
Derived d;
Base* p1 = &d;
Derived* p2 = p1; // 错误；需要强制转换
Derived* p3 = static_cast<Derived*>(p1); // 正确；p2 现在指向 Derived 对象
Base b;
Base* p4 = &b;
Derived* p5 = static_cast<Derived*>(p4); // 未定义行为，因为 p4 并不
// 指向 Derived 对象

```

同样，基类的引用可以使用 `static_cast` 转换为派生类的引用。

```

结构体 Base {};
结构体 Derived : Base {};
Derived d;
Base& r1 = d;
Derived& r2 = r1; // 错误；需要强制转换
Derived& r3 = static_cast<Derived&>(r1); // 正确；r3 现在引用 Derived 对象

```

如果源类型是多态的，可以使用 `dynamic_cast` 来执行基类到派生类的转换。它执行运行时检查，失败时可恢复，而不是产生未定义行为。在指针情况下，失败时返回空指针。在引用情况下，失败时抛出类型为 `std::bad_cast`（或继承自 `std::bad_cast` 的类）的异常。

```

结构体 Base { virtual ~Base(); }; // Base 是多态的
结构体 Derived : Base {};
Base* b1 = new Derived;
Derived* d1 = dynamic_cast<Derived*>(b1); // 正确；d1 指向 Derived 对象
Base* b2 = new Base;
Derived* d2 = dynamic_cast<Derived*>(b2); // d2 是空指针

```

第 91.4 节：指针与整数之间的转换

对象指针（包括 `void*`）或函数指针可以使用 `reinterpret_cast` 转换为整数类型。只有当目标类型足够长时，才会编译通过。结果是实现定义的，通常返回指针所指向内存字节的数值地址。

通常，`long` 或 `unsigned long` 足够容纳任何指针值，但标准并不保证这一点。

版本 ≥ C++11

如果存在类型 `std::intptr_t` 和 `std::uintptr_t`，则保证它们足够长以容纳 `void*`（因此也能容纳任何对象类型的指针）。然而，它们不保证足够长以容纳函数指针。

同样，`reinterpret_cast` 可用于将整数类型转换为指针类型。结果同样是实现定义的，但保证指针值经过整数类型的往返转换后保持不变。标准不保证数值零会被转换为空指针。

```

void register_callback(void (*fp)(void*), void* arg); // 可能是 C API
void my_callback(void* x) {
    std::cout << "the value is: " << reinterpret_cast<long>(x); // 可能会编译通过
}
long x;
std::cin >> x;

```

```

struct Base {};
struct Derived : Base {};
Derived d;
Base* p1 = &d;
Derived* p2 = p1; // error; cast required
Derived* p3 = static_cast<Derived*>(p1); // OK; p2 now points to Derived object
Base b;
Base* p4 = &b;
Derived* p5 = static_cast<Derived*>(p4); // undefined behaviour since p4 does not
// point to a Derived object

```

Likewise, a reference to base class can be converted to a reference to derived class using `static_cast`.

```

struct Base {};
struct Derived : Base {};
Derived d;
Base& r1 = d;
Derived& r2 = r1; // error; cast required
Derived& r3 = static_cast<Derived&>(r1); // OK; r3 now refers to Derived object

```

If the source type is polymorphic, `dynamic_cast` can be used to perform a base to derived conversion. It performs a run-time check and failure is recoverable instead of producing undefined behaviour. In the pointer case, a null pointer is returned upon failure. In the reference case, an exception is thrown upon failure of type `std::bad_cast` (or a class derived from `std::bad_cast`).

```

struct Base { virtual ~Base(); }; // Base is polymorphic
struct Derived : Base {};
Base* b1 = new Derived;
Derived* d1 = dynamic_cast<Derived*>(b1); // OK; d1 points to Derived object
Base* b2 = new Base;
Derived* d2 = dynamic_cast<Derived*>(b2); // d2 is a null pointer

```

Section 91.4: Conversion between pointer and integer

An object pointer (including `void*`) or function pointer can be converted to an integer type using `reinterpret_cast`. This will only compile if the destination type is long enough. The result is implementation-defined and typically yields the numeric address of the byte in memory that the pointer points to.

Typically, `long` or `unsigned long` is long enough to hold any pointer value, but this is not guaranteed by the standard.

Version ≥ C++11

If the types `std::intptr_t` and `std::uintptr_t` exist, they are guaranteed to be long enough to hold a `void*` (and hence any pointer to object type). However, they are not guaranteed to be long enough to hold a function pointer.

Similarly, `reinterpret_cast` can be used to convert an integer type into a pointer type. Again the result is implementation-defined, but a pointer value is guaranteed to be unchanged by a round trip through an integer type. The standard does not guarantee that the value zero is converted to a null pointer.

```

void register_callback(void (*fp)(void*), void* arg); // probably a C API
void my_callback(void* x) {
    std::cout << "the value is: " << reinterpret_cast<long>(x); // will probably compile
}
long x;
std::cin >> x;

```

```
register_callback(my_callback,  
    reinterpret_cast<void*>(x)); // 希望这不会丢失信息...
```

第91.5节：通过显式构造函数或显式转换函数进行转换

涉及调用显式构造函数或转换函数的转换不能隐式完成。我们可以使用`static_cast`请求显式完成转换。其含义与直接初始化相同，只是结果是一个临时对象。

```
class C {  
std::unique_ptr<int> p;  
public:  
    explicit C(int* p) : p(p) {}  
};  
void f(C c);  
void g(int* p) {  
    f(p); // 错误：C::C(int*) 是 explicit  
    f(static_cast<C>(p)); // 正确  
    f(C(p)); // 等同于上一行  
    C c(p); f(c); // 错误：C 不可复制  
}
```

```
register_callback(my_callback,  
    reinterpret_cast<void*>(x)); // hopefully this doesn't lose information...
```

Section 91.5: Conversion by explicit constructor or explicit conversion function

A conversion that involves calling an explicit constructor or conversion function can't be done implicitly. We can request that the conversion be done explicitly using `static_cast`. The meaning is the same as that of a direct initialization, except that the result is a temporary.

```
class C {  
    std::unique_ptr<int> p;  
public:  
    explicit C(int* p) : p(p) {}  
};  
void f(C c);  
void g(int* p) {  
    f(p); // error: C::C(int*) is explicit  
    f(static_cast<C>(p)); // ok  
    f(C(p)); // equivalent to previous line  
    C c(p); f(c); // error: C is not copyable  
}
```

第91.6节：隐式转换

`static_cast` 可以执行任何隐式转换。此处使用`static_cast` 有时非常有用，例如以下示例：

- 当传递参数给省略号时，“预期”的参数类型在编译时未知，因此不会发生隐式转换。

```
const double x = 3.14;  
printf("%d", static_cast<int>(x)); // 输出3 // printf("%d", x); /  
// 未定义行为；printf此处期望的是int类型// 替代方案：  
  
// const int y = x; printf("%d", y);
```

如果没有显式类型转换，`double` 对象将被传递给省略号，导致未定义行为。

- 派生类的赋值运算符可以像这样调用基类的赋值运算符：

```
struct Base { /* ... */ };  
struct Derived : Base {  
    Derived& operator=(const Derived& other) {  
        static_cast<Base&>(*this) = other;  
        // alternative:  
        // Base& this_base_ref = *this; this_base_ref = other;  
    }  
};
```

Section 91.6: Implicit conversion

`static_cast` 可以执行任何隐式转换。此处使用`static_cast` 有时非常有用，例如以下示例：

- When passing arguments to an ellipsis, the "expected" argument type is not statically known, so no implicit conversion will occur.

```
const double x = 3.14;  
printf("%d\n", static_cast<int>(x)); // prints 3  
// printf("%d\n", x); // undefined behaviour; printf is expecting an int here  
// alternative:  
// const int y = x; printf("%d\n", y);
```

Without the explicit type conversion, a `double` object would be passed to the ellipsis, and undefined behaviour would occur.

- A derived class assignment operator can call a base class assignment operator like so:

```
struct Base { /* ... */ };  
struct Derived : Base {  
    Derived& operator=(const Derived& other) {  
        static_cast<Base&>(*this) = other;  
        // alternative:  
        // Base& this_base_ref = *this; this_base_ref = other;  
    }  
};
```

第91.7节：枚举转换

`static_cast` 可以将整数或浮点类型转换为枚举类型（无论是作用域枚举还是

Section 91.7: Enum conversions

`static_cast` can convert from an integer or floating point type to an enumeration type (whether scoped or

无作用域枚举），反之亦然。它也可以在枚举类型之间进行转换。

- 从无作用域枚举类型转换为算术类型是隐式转换；可以使用static_cast，但不是必须的。

版本 ≥ C++11

- 当作用域枚举类型转换为算术类型时：

- 如果枚举的值可以在目标类型中被精确表示，结果就是该值。
- 否则，如果目标类型是整数类型，结果是不确定的。
- 否则，如果目标类型是浮点类型，结果与先转换到底层类型再转换为浮点类型的结果相同。

示例：

```
枚举类 Format {
    TEXT = 0,
    PDF = 1000,
    OTHER = 2000,
};

Format f = Format::PDF;
int a = f; // 错误
int b = static_cast<int>(f); // 正确；b 是 1000
char c = static_cast<char>(f); // 未指定，如果 1000 不适合 char
double d = static_cast<double>(f); // d 是 1000.0... 可能是
```

- 当整数或枚举类型转换为枚举类型时：

- 如果原始值在目标枚举的范围内，结果就是该值。注意该值可能与所有枚举量都不相等。
- 否则，结果是未指定的（<= C++14）或未定义的（>= C++17）。

示例：

```
枚举 Scale {
    SINGLE = 1,
    DOUBLE = 2,
    QUAD = 4
};
Scale s1 = 1; // 错误
Scale s2 = static_cast<Scale>(2); // s2 是 DOUBLE
Scale s3 = static_cast<Scale>(3); // s3 的值为 3，且不等于任何枚举量
Scale s9 = static_cast<Scale>(9); // C++14 中未指定的值；C++17 中为未定义行为
```

版本 ≥ C++11

- 当浮点类型转换为枚举类型时，结果与先转换为枚举的底层类型再转换为枚举类型相同。

```
enum Direction {
    UP = 0,
    LEFT = 1,
    DOWN = 2,
    RIGHT = 3,
};
```

unscoped), and vice versa. It can also convert between enumeration types.

- The conversion from an unscoped enumeration type to an arithmetic type is an implicit conversion; it is possible, but not necessary, to use `static_cast`.

Version ≥ C++11

- When a scoped enumeration type is converted to an arithmetic type:

- If the enum's value can be represented exactly in the destination type, the result is that value.
- Otherwise, if the destination type is an integer type, the result is unspecified.
- Otherwise, if the destination type is a floating point type, the result is the same as that of converting to the underlying type and then to the floating point type.

Example:

```
enum class Format {
    TEXT = 0,
    PDF = 1000,
    OTHER = 2000,
};

Format f = Format::PDF;
int a = f; // error
int b = static_cast<int>(f); // ok; b is 1000
char c = static_cast<char>(f); // unspecified, if 1000 doesn't fit into char
double d = static_cast<double>(f); // d is 1000.0... probably
```

- When an integer or enumeration type is converted to an enumeration type:

- If the original value is within the destination enum's range, the result is that value. Note that this value might be unequal to all enumerators.
- Otherwise, the result is unspecified (<= C++14) or undefined (>= C++17).

Example:

```
enum Scale {
    SINGLE = 1,
    DOUBLE = 2,
    QUAD = 4
};
Scale s1 = 1; // error
Scale s2 = static_cast<Scale>(2); // s2 is DOUBLE
Scale s3 = static_cast<Scale>(3); // s3 has value 3, and is not equal to any enumerator
Scale s9 = static_cast<Scale>(9); // unspecified value in C++14; UB in C++17
```

Version ≥ C++11

- When a floating point type is converted to an enumeration type, the result is the same as converting to the enum's underlying type and then to the enum type.

```
enum Direction {
    UP = 0,
    LEFT = 1,
    DOWN = 2,
    RIGHT = 3,
};
```

```
Direction d = static_cast<Direction>(3.14); // d 是 RIGHT
```

```
Direction d = static_cast<Direction>(3.14); // d is RIGHT
```

第 91.8 节：派生类到基类的成员指针转换

派生类的成员指针可以使用 `static_cast` 转换为基类的成员指针。
所指向的类型必须匹配。

如果操作数是成员指针的空指针值，结果也是成员指针的空指针值。

否则，只有当操作数所指向的成员实际存在于目标类中，或者目标类是包含操作数所指向成员的类的基类或派生类时，
转换才有效。`static_cast` 不会检查有效性。如果转换无效，行为未定义。

```
struct A {};
struct B { int x; };
struct C : A, B { int y; double z; };
int B::*p1 = &B::x;
int C::*p2 = p1; // 正确；隐式转换
int B::*p3 = p2; // 错误
int B::*p4 = static_cast<int B::*>(p2); // 正确；p4 等于 p1
int A::*p5 = static_cast<int A::*>(p2); // 未定义；p2 指向 x，它是无关类 B 的成员
// 
double C::*p6 = &C::z;
double A::*p7 = static_cast<double A::*>(p6); // 可以，尽管 A 不包含 z
int A::*p8 = static_cast<int A::*>(p6); // 错误：类型不匹配
```

第91.9节：void* 转换为 T*

在 C++ 中，`void*` 不能隐式转换为 `T*`，其中 `T` 是一个对象类型。相反，应该使用 `static_cast` 来显式执行转换。如果操作数实际上指向一个 `T` 对象，结果将指向该对象。

否则，结果未指定。

版本 ≥ C++11

即使操作数不指向 `T` 对象，只要操作数指向的字节地址对于类型 `T` 是正确对齐的，转换结果就指向同一个字节。

```
// 分配一个包含100个整数的数组，较为复杂的方式
int* a = malloc(100*sizeof(*a)); // 错误；malloc返回void*
int* a = static_cast<int*>(malloc(100*sizeof(*a))); // 正确
// int* a = new int[100]; // 不需要强制类型转换
// std::vector<int> a(100); // 更好

const char c = '!';
const void* p1 = &c;
const char* p2 = p1; // 错误const char*
p3 = static_cast<const char*>(p1); // 正确；p3 指向 c
const int* p4 = static_cast<const int*>(p1); // 如果 alignof(int) > alignof(char)，C++11 中可能未指定
char* p5 = static_cast<char*>(p1); // 错误：去除 const 属性的转换
```

Section 91.8: Derived to base conversion for pointers to members

A pointer to member of derived class can be converted to a pointer to member of base class using `static_cast`.
The types pointed to must match.

If the operand is a null pointer to member value, the result is also a null pointer to member value.

Otherwise, the conversion is only valid if the member pointed to by the operand actually exists in the destination class, or if the destination class is a base or derived class of the class containing the member pointed to by the operand. `static_cast` does not check for validity. If the conversion is not valid, the behaviour is undefined.

```
struct A {};
struct B { int x; };
struct C : A, B { int y; double z; };
int B::*p1 = &B::x;
int C::*p2 = p1; // ok; implicit conversion
int B::*p3 = p2; // error
int B::*p4 = static_cast<int B::*>(p2); // ok; p4 is equal to p1
int A::*p5 = static_cast<int A::*>(p2); // undefined; p2 points to x, which is a member
// of the unrelated class B
double C::*p6 = &C::z;
double A::*p7 = static_cast<double A::*>(p6); // ok, even though A doesn't contain z
int A::*p8 = static_cast<int A::*>(p6); // error: types don't match
```

Section 91.9: void* to T*

In C++, `void*` cannot be implicitly converted to `T*` where `T` is an object type. Instead, `static_cast` should be used to perform the conversion explicitly. If the operand actually points to a `T` object, the result points to that object.
Otherwise, the result is unspecified.

Version ≥ C++11

Even if the operand does not point to a `T` object, as long as the operand points to a byte whose address is properly aligned for the type `T`, the result of the conversion points to the same byte.

```
// allocating an array of 100 ints, the hard way
int* a = malloc(100*sizeof(*a)); // error; malloc returns void*
int* a = static_cast<int*>(malloc(100*sizeof(*a))); // ok
// int* a = new int[100]; // no cast needed
// std::vector<int> a(100); // better

const char c = '!';
const void* p1 = &c;
const char* p2 = p1; // error
const char* p3 = static_cast<const char*>(p1); // ok; p3 points to c
const int* p4 = static_cast<const int*>(p1); // unspecified in C++03;
// possibly unspecified in C++11 if
// alignof(int) > alignof(char)
char* p5 = static_cast<char*>(p1); // error: casting away constness
```

第 91.10 节：类型惩罚转换

可以使用 `reinterpret_cast` 将指向某对象类型的指针（或引用）转换为指向任何其他对象类型的指针（或引用）。这不会调用任何构造函数或转换函数。

```
int x = 42;
char* p = static_cast<char*>(&x);      // 错误：static_cast 无法执行此转换
char* p = reinterpret_cast<char*>(&x); // 正确
*p = 'z';                                // 可能修改 x (见下文)
```

版本 ≥ C++11

`reinterpret_cast` 的结果表示与操作数相同的地址，前提是该地址对于目标类型是适当对齐的。否则，结果未指定。

```
int x = 42;
char& r = reinterpret_cast<char&>(x);
const void* px = &x;
const void* pr = &r;
assert(px == pr); // 应该永远不会触发
```

版本 < C++11

`reinterpret_cast` 的结果是不确定的，除了指针（或引用）在从源类型到目标类型再回到源类型的往返过程中会保持不变，只要目标类型的对齐要求不比源类型更严格。

```
int x = 123;
unsigned int& r1 = reinterpret_cast<unsigned int&>(x);
int& r2 = reinterpret_cast<int&>(r1);
r2 = 456; // 将 x 设置为 456
```

在大多数实现中，`reinterpret_cast` 不会改变地址，但这一要求直到 C++11 才被标准化。

`reinterpret_cast` 也可以用于将一种指向数据成员的指针类型转换为另一种，或将一种指向成员函数的指针类型转换为另一种。

`reinterpret_cast` 的使用被认为是危险的，因为通过 `reinterpret_cast` 获得的指针或引用进行读写操作时，如果源类型和目标类型无关，可能会触发未定义行为。

Section 91.10: Type punning conversion

A pointer (resp. reference) to an object type can be converted to a pointer (resp. reference) to any other object type using `reinterpret_cast`. This does not call any constructors or conversion functions.

```
int x = 42;
char* p = static_cast<char*>(&x);      // error: static_cast cannot perform this conversion
char* p = reinterpret_cast<char*>(&x); // OK
*p = 'z';                                // maybe this modifies x (see below)
```

Version ≥ C++11

The result of `reinterpret_cast` represents the same address as the operand, provided that the address is appropriately aligned for the destination type. Otherwise, the result is unspecified.

```
int x = 42;
char& r = reinterpret_cast<char&>(x);
const void* px = &x;
const void* pr = &r;
assert(px == pr); // should never fire
```

Version < C++11

The result of `reinterpret_cast` is unspecified, except that a pointer (resp. reference) will survive a round trip from the source type to the destination type and back, as long as the destination type's alignment requirement is not stricter than that of the source type.

```
int x = 123;
unsigned int& r1 = reinterpret_cast<unsigned int&>(x);
int& r2 = reinterpret_cast<int&>(r1);
r2 = 456; // sets x to 456
```

On most implementations, `reinterpret_cast` does not change the address, but this requirement was not standardized until C++11.

`reinterpret_cast` can also be used to convert from one pointer-to-data-member type to another, or one pointer-to-member-function type to another.

Use of `reinterpret_cast` is considered dangerous because reading or writing through a pointer or reference obtained using `reinterpret_cast` may trigger undefined behaviour when the source and destination types are unrelated.

第92章：无名类型

第92.1节：未命名类

与具名类或结构体不同，匿名类和结构体必须在定义处实例化，且不能有构造函数或析构函数。

```
结构体 {
    int foo;
    double bar;
} foobar;

foobar.foo = 5;
foobar.bar = 4.0;

类 {
    int baz;
public:
    int buzz;

    void setBaz(int v) {
        baz = v;
    }
} barbar;

barbar.setBaz(15);
barbar.buzz = 2;
```

第92.2节：作为类型别名

匿名类类型也可以用于创建类型别名，即通过 `typedef` 和 `using`：

```
版本 < C++11
using vec2d = struct {
    float x;
    float y;
};

typedef struct {
    float x;
    float y;
} vec2d;

vec2d pt;
pt.x = 4.f;
pt.y = 3.f;
```

第92.3节：匿名成员

作为C++的非标准扩展，常见编译器允许使用类作为匿名成员。

```
struct 示例 {
    struct {
        int 内部_b;
    };

    int 外部_b;
```

Chapter 92: Unnamed types

Section 92.1: Unnamed classes

Unlike a named class or struct, unnamed classes and structs must be instantiated where they are defined, and cannot have constructors or destructors.

```
struct {
    int foo;
    double bar;
} foobar;

foobar.foo = 5;
foobar.bar = 4.0;

class {
    int baz;
public:
    int buzz;

    void setBaz(int v) {
        baz = v;
    }
} barbar;

barbar.setBaz(15);
barbar.buzz = 2;
```

Section 92.2: As a type alias

Unnamed class types may also be used when creating type aliases, i.e. via `typedef` and `using`:

```
Version < C++11
using vec2d = struct {
    float x;
    float y;
};

typedef struct {
    float x;
    float y;
} vec2d;

vec2d pt;
pt.x = 4.f;
pt.y = 3.f;
```

Section 92.3: Anonymous members

As a non-standard extension to C++, common compilers allow the use of classes as anonymous members.

```
struct Example {
    struct {
        int inner_b;
    };

    int outer_b;
```

```
//匿名结构体的成员可以像父结构体的成员一样访问
示例() : inner_b(2), outer_b(4) {
inner_b = outer_b + 2;
}
};

示例 ex;

// 对于外部代码引用该结构体同样适用
ex.inner_b -= ex.outer_b;
```

第92.4节：匿名联合体

匿名联合体的成员名属于联合体声明的作用域，且必须与该作用域内的所有其他名称不同。这里的示例结构与使用“struct”的匿名成员示例结构相同，但符合标准。

```
struct Sample {
    union {
        int a;
        int b;
    };
    int c;
};
int main()
{
    Sample sa;
    sa.a = 3;
    sa.b = 4;
    sa.c = 5;
}
```

```
//The anonymous struct's members are accessed as if members of the parent struct
Example() : inner_b(2), outer_b(4) {
    inner_b = outer_b + 2;
}
};

Example ex;

//The same holds true for external code referencing the struct
ex.inner_b -= ex.outer_b;
```

Section 92.4: Anonymous Union

Member names of an anonymous union belong to the scope of the union declaration and must be distinct to all other names of this scope. The example here has the same construction as example Anonymous Members using "struct" but is standard conform.

```
struct Sample {
    union {
        int a;
        int b;
    };
    int c;
};
int main()
{
    Sample sa;
    sa.a = 3;
    sa.b = 4;
    sa.c = 5;
}
```

第93章：类型特性

第93.1节：类型属性

版本 ≥ C++11

类型属性比较可以施加于不同变量的修饰符。这些类型的用处并不总是显而易见的。

注意：下面的示例仅在非优化编译器上提供改进。这是一个简单的概念验证，而非复杂示例。

例如，快速除以四。

```
template<typename T>
inline T FastDivideByFour(const T &var) {
    // 如果输入类型不是无符号整型，将会报错。
    static_assert(std::is_unsigned<T>::value && std::is_integral<T>::value,
        "此函数仅设计用于无符号整型。");
    return (var >> 2);
}
```

是否为常量：

当类型是常量时，这将被评估为真。

```
std::cout << std::is_const<const int>::value << ""; // 输出 true。
std::cout << std::is_const<int>::value << ""; // 输出 false。
```

是否为易变类型：

当类型是易变类型时，这将被评估为真。

```
std::cout << std::is_volatile<static volatile int>::value << ""; // 输出 true。
std::cout << std::is_const<const int>::value << ""; // 输出 false。
```

是否为有符号类型：

对于所有有符号类型，这将被评估为真。

```
std::cout << std::is_signed<int>::value << ""; // 输出 true。
std::cout << std::is_signed<float>::value << ""; // 输出 true。
std::cout << std::is_signed<unsigned int>::value << ""; // 输出 false。
std::cout << std::is_signed<uint8_t>::value << ""; // 输出 false。
```

是否为无符号：

对所有无符号类型均评估为 true。

```
std::cout << std::is_unsigned<unsigned int>::value << ""; // 输出 true。
std::cout << std::is_signed<uint8_t>::value << ""; // 输出 true。
std::cout << std::is_unsigned<int>::value << ""; // 输出 false。
std::cout << std::is_signed<float>::value << ""; // 输出 false。
```

Chapter 93: Type Traits

Section 93.1: Type Properties

Version ≥ C++11

Type properties compare the modifiers that can be placed upon different variables. The usefulness of these type traits is not always obvious.

Note: The example below would only offer an improvement on a non-optimizing compiler. It is a simple proof of concept, rather than complex example.

e.g. Fast divide by four.

```
template<typename T>
inline T FastDivideByFour(const T &var) {
    // Will give an error if the inputted type is not an unsigned integral type.
    static_assert(std::is_unsigned<T>::value && std::is_integral<T>::value,
        "This function is only designed for unsigned integral types.");
    return (var >> 2);
}
```

Is Constant:

This will evaluate as true when type is constant.

```
std::cout << std::is_const<const int>::value << "\n"; // Prints true.
std::cout << std::is_const<int>::value << "\n"; // Prints false.
```

Is Volatile:

This will evaluate as true when the type is volatile.

```
std::cout << std::is_volatile<static volatile int>::value << "\n"; // Prints true.
std::cout << std::is_const<const int>::value << "\n"; // Prints false.
```

Is signed:

This will evaluate as true for all signed types.

```
std::cout << std::is_signed<int>::value << "\n"; // Prints true.
std::cout << std::is_signed<float>::value << "\n"; // Prints true.
std::cout << std::is_signed<unsigned int>::value << "\n"; // Prints false.
std::cout << std::is_signed<uint8_t>::value << "\n"; // Prints false.
```

Is Unsigned:

Will evaluate as true for all unsigned types.

```
std::cout << std::is_unsigned<unsigned int>::value << "\n"; // Prints true.
std::cout << std::is_signed<uint8_t>::value << "\n"; // Prints true.
std::cout << std::is_unsigned<int>::value << "\n"; // Prints false.
std::cout << std::is_signed<float>::value << "\n"; // Prints false.
```

第93.2节：标准类型特征

版本 ≥ C++11

类型特征 (type_traits) 头文件包含一组模板类和辅助工具，用于在编译时转换和检查类型的属性。

这些特征通常用于模板中检查用户错误，支持泛型编程，并允许进行优化。

大多数类型特征用于检查某个类型是否满足某些条件。它们具有以下形式：

```
template <class T> struct is_foo;
```

如果模板类以满足某些条件 foo 的类型实例化，则 `is_foo<T>` 继承自 `std::integral_constant<bool, true>` (也称为 `std::true_type`)，否则继承自 `std::integral_constant<bool, false>` (也称为 `std::false_type`)。这使得该特征具有以下成员：

常量

`static constexpr bool value`

如果 T 满足条件 foo，则为 `true`，否则为 `false`

函数

`operator bool`

返回 `value`

版本 ≥ C++14

`bool operator()`

返回值

类型

名称 定义

`value_type` `bool`

类型 `std::integral_constant<bool, value>`

该特性随后可以用于诸如 `static_assert` 或 `std::enable_if` 等结构。以下是使用 `std::is_pointer` 的示例：

```
template <typename T>
void i_require_a_pointer (T t) {
    static_assert(std::is_pointer<T>::value, "T 必须是指针类型");
}

//当 T 不是指针类型时的重载
template <typename T>
typename std::enable_if::type
does_something_special_with_pointer (T t) {
    //做一些无聊的事情
}

//当 T 是指针类型时的重载
template <typename T>
```

Section 93.2: Standard type traits

Version ≥ C++11

The `type_traits` header contains a set of template classes and helpers to transform and check properties of types at compile-time.

These traits are typically used in templates to check for user errors, support generic programming, and allow for optimizations.

Most type traits are used to check if a type fulfills some criteria. These have the following form:

```
template <class T> struct is_foo;
```

If the template class is instantiated with a type which fulfills some criteria `foo`, then `is_foo<T>` inherits from `std::integral_constant<bool, true>` (a.k.a. `std::true_type`), otherwise it inherits from `std::integral_constant<bool, false>` (a.k.a. `std::false_type`). This gives the trait the following members:

Constants

`static constexpr bool value`

`true` if T fulfills the criteria `foo`, `false` otherwise

Functions

`operator bool`

Returns `value`

Version ≥ C++14

`bool operator()`

Returns `value`

Types

Name Definition

`value_type` `bool`

type `std::integral_constant<bool, value>`

The trait can then be used in constructs such as `static_assert` or `std::enable_if`. An example with `std::is_pointer`:

```
template <typename T>
void i_require_a_pointer (T t) {
    static_assert(std::is_pointer<T>::value, "T must be a pointer type");
}

//Overload for when T is not a pointer type
template <typename T>
typename std::enable_if::type
does_something_special_with_pointer (T t) {
    //Do something boring
}

//Overload for when T is a pointer type
template <typename T>
```

```

typename std::enable_if<std::is_pointer<T>::value>::type
does_something_special_with_pointer (T t) {
    //执行特殊操作
}

```

还有各种转换类型的traits，例如`std::add_pointer`和`std::underlying_type`。这些traits通常暴露一个包含转换后类型的单一type成员类型。例如，`std::add_pointer<int>::type`是`int*`。

第93.3节：使用`std::is_same<T, T>`的类型关系

版本 ≥ C++11

`std::is_same<T, T>`类型关系用于比较两个类型。如果类型相同则返回布尔值`true`，否则为`false`。

例如

```

// 在大多数x86和x86_64编译器上打印true。
std::cout << std::is_same<int, int32_t>::value << "";// 在所有编译器
上打印false。
std::cout << std::is_same<float, int>::value << "";// 在所有编译器
上打印 false。
std::cout << std::is_same<unsigned int, int>::value << "";

```

`std::is_same` 类型关系也适用于 `typedef`，无论 `typedef` 如何。这实际上在第一个示例中比较 `int == int32_t` 时有所体现，但这并不完全明显。

例如

```

// 在所有编译器上打印 true。
typedef int MyType
std::cout << std::is_same<int, MyType>::value << "";

```

使用 `std::is_same` 来警告不正确使用模板类或函数的情况。

当结合 `static_assert` 使用时，`std::is_same` 模板是强制正确使用模板类和函数的有力工具。

例如，一个函数只允许输入 `int` 类型和两个结构体中的一个。

```

#include <type_traits>
struct foo {
    int member;
    // 其他变量
};

struct bar {
    char member;
};

template<typename T>
int AddStructMember(T var1, int var2) {
    // 如果类型 T != foo 或 T != bar 则显示错误信息。
    static_assert(std::is_same<T, foo>::value ||
        std::is_same<T, bar>::value,
        "此函数不支持指定的类型。");
}

```

```

typename std::enable_if<std::is_pointer<T>::value>::type
does_something_special_with_pointer (T t) {
    //Do something special
}

```

There are also various traits which transform types, such as `std::add_pointer` and `std::underlying_type`. These traits generally expose a single type member type which contains the transformed type. For example, `std::add_pointer<int>::type` is `int*`.

Section 93.3: Type relations with `std::is_same<T, T>`

Version ≥ C++11

The `std::is_same<T, T>` type relation is used to compare two types. It will evaluate as boolean, true if the types are the same and false if otherwise.

e.g.

```

// Prints true on most x86 and x86_64 compilers.
std::cout << std::is_same<int, int32_t>::value << "\n";
// Prints false on all compilers.
std::cout << std::is_same<float, int>::value << "\n";
// Prints false on all compilers.
std::cout << std::is_same<unsigned int, int>::value << "\n";

```

The `std::is_same` type relation will also work regardless of typedefs. This is actually demonstrated in the first example when comparing `int == int32_t` however this is not entirely clear.

e.g.

```

// Prints true on all compilers.
typedef int MyType
std::cout << std::is_same<int, MyType>::value << "\n";

```

Using `std::is_same` to warn when improperly using a templated class or function.

When combined with a static assert the `std::is_same` template can be valuable tool in enforcing proper usage of templated classes and functions.

e.g. A function that only allows input from an `int` and a choice of two structs.

```

#include <type_traits>
struct foo {
    int member;
    // Other variables
};

struct bar {
    char member;
};

template<typename T>
int AddStructMember(T var1, int var2) {
    // If type T != foo || T != bar then show error message.
    static_assert(std::is_same<T, foo>::value ||
        std::is_same<T, bar>::value,
        "This function does not support the specified type.");
}

```

```
    return var1.member + var2;
}
```

第 93.4 节：基本类型特征

版本 ≥ C++11

有许多不同的类型特征用于比较更通用的类型。

是否为整型：

对所有整型类型（如int、char、long、unsigned int等）评估为真。

```
std::cout << std::is_integral<int>::value << ""; // 输出 true.
std::cout << std::is_integral<char>::value << ""; // 输出 true.
std::cout << std::is_integral<float>::value << ""; // 输出 false.
```

是否为浮点数：

对所有浮点类型（如float、double、long double等）均返回 true。

```
std::cout << std::is_floating_point<float>::value << ""; // 输出 true.
std::cout << std::is_floating_point<double>::value << ""; // 输出 true.
std::cout << std::is_floating_point<char>::value << ""; // 输出 false.
```

是否为枚举类型：

对所有枚举类型（包括enum class）均返回 true。

```
enum fruit {apple, pair, banana};enum
class vegetable {carrot, spinach, leek};std::cout << std:
::is_enum<fruit>::value << ""; // 输出 true.
std::cout << std::is_enum<vegetable>::value << ""; // 输出 true.
std::cout << std::is_enum<int>::value << ""; // 输出 false.
```

是否为指针：

对所有指针类型的判断结果为 true。

```
std::cout << std::is_pointer<int *>::value << ""; // 输出 true.
typedef int* MyPTR;
std::cout << std::is_pointer<MyPTR>::value << ""; // 输出 true.
std::cout << std::is_pointer<int>::value << ""; // 输出 false.
```

是否为类：

对所有类和结构体判断为 true，枚举类（enum class）除外。

```
结构体 FOO {整型 x, y;};
类 BAR {
    公共:
        整型 x, y;
};
枚举类 fruit {苹果, 梨, 香蕉};标准::输出流 << 标准
::是类<FOO>::值 << ""; // 输出 true.
标准::输出流 << 标准::是类<BAR>::值 << ""; // 输出 true.
标准::输出流 << 标准::是类<fruit>::值 << ""; // 输出 false.
```

```
    return var1.member + var2;
}
```

Section 93.4: Fundamental type traits

Version ≥ C++11

There are a number of different type traits that compare more general types.

Is Integral:

Evaluates as true for all integer types int, char, long, unsigned int etc.

```
std::cout << std::is_integral<int>::value << "\n"; // Prints true.
std::cout << std::is_integral<char>::value << "\n"; // Prints true.
std::cout << std::is_integral<float>::value << "\n"; // Prints false.
```

Is Floating Point:

Evaluates as true for all floating point types. float, double, long double etc.

```
std::cout << std::is_floating_point<float>::value << "\n"; // Prints true.
std::cout << std::is_floating_point<double>::value << "\n"; // Prints true.
std::cout << std::is_floating_point<char>::value << "\n"; // Prints false.
```

Is Enum:

Evaluates as true for all enumerated types, including enum class.

```
enum fruit {apple, pair, banana};
enum class vegetable {carrot, spinach, leek};
std::cout << std::is_enum<fruit>::value << "\n"; // Prints true.
std::cout << std::is_enum<vegetable>::value << "\n"; // Prints true.
std::cout << std::is_enum<int>::value << "\n"; // Prints false.
```

Is Pointer:

Evaluates as true for all pointers.

```
std::cout << std::is_pointer<int *>::value << "\n"; // Prints true.
typedef int* MyPTR;
std::cout << std::is_pointer<MyPTR>::value << "\n"; // Prints true.
std::cout << std::is_pointer<int>::value << "\n"; // Prints false.
```

Is Class:

Evaluates as true for all classes and struct, with the exception of enum class.

```
struct FOO {int x, y;};
class BAR {
    public:
        int x, y;
};
enum class fruit {apple, pair, banana};
std::cout << std::is_class<FOO>::value << "\n"; // Prints true.
std::cout << std::is_class<BAR>::value << "\n"; // Prints true.
std::cout << std::is_class<fruit>::value << "\n"; // Prints false.
```

```
标准::输出流 << 标准::是类<整型>::值 << ""; // 输出 false。
```

```
std::cout << std::is_class<int>::value << "\n"; // Prints false.
```

第94章：返回类型协变

第94.1节：基例的协变结果版本，静态类型检查

```
// 2. 基础示例的协变返回类型版本，静态类型检查。

class Top
{
public:
    virtual Top* clone() const = 0;
    virtual ~Top() = default;      // 通过 Top* 删除时必需。
};

class D : public Top
{
public:
    D* /* ← 协变返回类型 */ clone() const override
    { return new D( *this ); }
};

class DD : public D
{
private:
    int answer_ = 42;

public:
    int answer() const
    { return answer_; }

DD* /* ← 协变返回类型 */ clone() const override
{ return new DD( *this ); }
};

#include <iostream>
using namespace std;

int main()
{
    DD* p1 = new DD();
    DD* p2 = p1->clone();
    // 通过静态类型检查保证了 *p2 的动态类型为 DD。

    cout << p2->answer() << endl;           // "42"
    delete p2;
    delete p1;
}
```

第94.2节：协变智能指针结果（自动清理）

```
// 3. 协变智能指针结果（自动清理）。

#include <memory>
using std::unique_ptr;

template< class Type >
auto up( Type* p ) { return unique_ptr<Type>( p ); }
```

Chapter 94: Return Type Covariance

Section 94.1: Covariant result version of the base example, static type checking

```
// 2. Covariant result version of the base example, static type checking.

class Top
{
public:
    virtual Top* clone() const = 0;
    virtual ~Top() = default;      // Necessary for `delete` via Top*.
};

class D : public Top
{
public:
    D* /* ← Covariant return */ clone() const override
    { return new D( *this ); }
};

class DD : public D
{
private:
    int answer_ = 42;

public:
    int answer() const
    { return answer_; }

DD* /* ← Covariant return */ clone() const override
{ return new DD( *this ); }
};

#include <iostream>
using namespace std;

int main()
{
    DD* p1 = new DD();
    DD* p2 = p1->clone();
    // Correct dynamic type DD for *p2 is guaranteed by the static type checking.

    cout << p2->answer() << endl;           // "42"
    delete p2;
    delete p1;
}
```

Section 94.2: Covariant smart pointer result (automated cleanup)

```
// 3. Covariant smart pointer result (automated cleanup).

#include <memory>
using std::unique_ptr;

template< class Type >
auto up( Type* p ) { return unique_ptr<Type>( p ); }
```

```

class Top
{
private:
    virtual Top* virtual_clone() const = 0;

公共:
unique_ptr<Top> clone() const
{ return up( virtual_clone() ); }

    virtual ~Top() = default;      // 通过 Top* 删除时必需。
};

class D : public Top
{
private:
D* /* ← 协变返回 */ virtual_clone() const override
{ return new D( *this ); }

公共:
unique_ptr<D> /* ← 表面协变返回 */ clone() const
{ return up( virtual_clone() ); }

class DD : public D
{
private:
    int answer_ = 42;

DD* /* ← 协变返回 */ virtual_clone() const override
{ return new DD( *this ); }

公共:
    int answer() const
{ return answer_; }

unique_ptr<DD> /* ← 表面协变返回 */ clone() const
{ return up( virtual_clone() ); }

#include <iostream>
using namespace std;

int main()
{
    auto p1 = unique_ptr<DD>(new DD());
    auto p2 = p1->clone();
    // 通过静态类型检查保证了 *p2 的动态类型为 DD。

    cout << p2->answer() << endl;      // "42"
    // 清理通过 unique_ptr 自动完成。
}

```

```

class Top
{
private:
    virtual Top* virtual_clone() const = 0;

public:
    unique_ptr<Top> clone() const
{ return up( virtual_clone() ); }

    virtual ~Top() = default;      // Necessary for `delete` via Top*.
};

class D : public Top
{
private:
D* /* ← Covariant return */ virtual_clone() const override
{ return new D( *this ); }

public:
    unique_ptr<D> /* ← Apparent covariant return */ clone() const
{ return up( virtual_clone() ); }

class DD : public D
{
private:
    int answer_ = 42;

DD* /* ← Covariant return */ virtual_clone() const override
{ return new DD( *this ); }

public:
    int answer() const
{ return answer_; }

unique_ptr<DD> /* ← Apparent covariant return */ clone() const
{ return up( virtual_clone() ); }

#include <iostream>
using namespace std;

int main()
{
    auto p1 = unique_ptr<DD>(new DD());
    auto p2 = p1->clone();
    // Correct dynamic type DD for *p2 is guaranteed by the static type checking.

    cout << p2->answer() << endl;      // "42"
    // Cleanup is automated via unique_ptr.
}

```

第95章：对象类型的布局

第95.1节：类类型

这里所说的“类”，指的是使用 `class` 或 `struct` 关键字定义的类型（但不包括 `enum struct` 或 `enum class`）。

- 即使是空类也至少占用一个字节的存储空间；因此它将纯粹由填充组成。这确保了如果 `p` 指向一个空类对象，那么 `p + 1` 是一个不同的地址，并指向一个不同的对象。然而，当空类用作基类时，其大小可能为 0。详见空基类优化。

```
class Empty_1 {}; // sizeof(Empty_1) == 1
class Empty_2 {}; // sizeof(Empty_2) == 1
class Derived : Empty_1 {}; // sizeof(Derived) == 1
class DoubleDerived : Empty_1, Empty_2 {}; // sizeof(DoubleDerived) == 1
class Holder { Empty_1 e; }; // sizeof(Holder) == 1
class DoubleHolder { Empty_1 e1; Empty_2 e2; }; // sizeof(DoubleHolder) == 2
class DerivedHolder : Empty_1 { Empty_1 e; }; // sizeof(DerivedHolder) == 2
```

- 类类型的对象表示包含基类和非静态成员类型的对象表示。因此，例如，在以下类中：

```
struct S {
    int x;
    char* y;
};
```

在 `S` 对象中，有一段连续的 `sizeof(int)` 字节序列，称为子对象，包含了 `x` 的值，另有一段 `sizeof(char*)` 字节的子对象包含了 `y` 的值。这两者不能交错存放。

- 如果一个类类型具有类型为 `t1, t2, ..., tN` 的成员和/或基类，则根据上述内容，其大小必须至少为 `sizeof(t1) + sizeof(t2) + ... + sizeof(tN)`。然而，根据成员和基类的对齐要求，编译器可能被迫在子对象之间，或在整个对象的开头或结尾插入填充。

```
struct AnInt { int i; };
// sizeof(AnInt) == sizeof(int)
// 假设一个典型的32位或64位系统, sizeof(AnInt) == 4 (4).
struct TwoInts { int i, j; };
// sizeof(TwoInts) >= 2 * sizeof(int)
// 假设典型的32位或64位系统, sizeof(TwoInts) == 8 (4 + 4).
struct IntAndChar { int i; char c; };
// sizeof(IntAndChar) >= sizeof(int) + sizeof(char)
// 假设典型的32位或64位系统, sizeof(IntAndChar) == 8 (4 + 1 + 填充).
struct AnIntDerived : AnInt { long long l; };
// sizeof(AnIntDerived) >= sizeof(AnInt) + sizeof(long long)
// 假设典型的32位或64位系统, sizeof(AnIntDerived) == 16 (4 + 填充 + 8).
```

- 如果由于对齐要求在对象中插入了填充，则大小将大于成员和基类大小之和。在 `n` 字节对齐的情况下，大小通常是大于所有成员和基类大小的最小 `n` 的倍数。每个成员 `memN` 通常会放置在一个地址上，该地址是 `alignof(memN)` 的倍数，且 `n` 通常是所有成员中最大的 `alignof`。

Chapter 95: Layout of object types

Section 95.1: Class types

By "class", we mean a type that was defined using the `class` or `struct` keyword (but not `enum struct` or `enum class`).

- Even an empty class still occupies at least one byte of storage; it will therefore consist purely of padding. This ensures that if `p` points to an object of an empty class, then `p + 1` is a distinct address and points to a distinct object. However, it is possible for an empty class to have a size of 0 when used as a base class. See [empty base optimisation](#).

```
class Empty_1 {}; // sizeof(Empty_1) == 1
class Empty_2 {}; // sizeof(Empty_2) == 1
class Derived : Empty_1 {}; // sizeof(Derived) == 1
class DoubleDerived : Empty_1, Empty_2 {}; // sizeof(DoubleDerived) == 1
class Holder { Empty_1 e; }; // sizeof(Holder) == 1
class DoubleHolder { Empty_1 e1; Empty_2 e2; }; // sizeof(DoubleHolder) == 2
class DerivedHolder : Empty_1 { Empty_1 e; }; // sizeof(DerivedHolder) == 2
```

- The object representation of a class type contains the object representations of the base class and non-static member types. Therefore, for example, in the following class:

```
struct S {
    int x;
    char* y;
};
```

there is a consecutive sequence of `sizeof(int)` bytes within an `S` object, called a *subobject*, that contain the value of `x`, and another subobject with `sizeof(char*)` bytes that contains the value of `y`. The two cannot be interleaved.

- If a class type has members and/or base classes with types `t1, t2, ..., tN`, the size must be at least `sizeof(t1) + sizeof(t2) + ... + sizeof(tN)` given the preceding points. However, depending on the alignment requirements of the members and base classes, the compiler may be forced to insert padding between subobjects, or at the beginning or end of the complete object.

```
struct AnInt { int i; };
// sizeof(AnInt) == sizeof(int)
// Assuming a typical 32- or 64-bit system, sizeof(AnInt) == 4 (4).
struct TwoInts { int i, j; };
// sizeof(TwoInts) >= 2 * sizeof(int)
// Assuming a typical 32- or 64-bit system, sizeof(TwoInts) == 8 (4 + 4).
struct IntAndChar { int i; char c; };
// sizeof(IntAndChar) >= sizeof(int) + sizeof(char)
// Assuming a typical 32- or 64-bit system, sizeof(IntAndChar) == 8 (4 + 1 + padding).
struct AnIntDerived : AnInt { long long l; };
// sizeof(AnIntDerived) >= sizeof(AnInt) + sizeof(long long)
// Assuming a typical 32- or 64-bit system, sizeof(AnIntDerived) == 16 (4 + padding + 8).
```

- If padding is inserted in an object due to alignment requirements, the size will be greater than the sum of the sizes of the members and base classes. With `n`-byte alignment, size will typically be the smallest multiple of `n` which is larger than the size of all members & base classes. Each member `memN` will typically be placed at an address which is a multiple of `alignof(memN)`, and `n` will typically be the largest `alignof` out of all members'.

alignof。因此，如果一个具有较小 alignof的成员后面跟着一个具有较大 alignof的成员，则如果将后者紧跟前者放置，后者可能无法正确对齐。在这种情况下，会在两个成员之间插入填充（也称为对齐成员），以便后者成员能够获得所需的对齐。相反，如果一个具有较大 alignof的成员后面跟着一个具有较小 alignof的成员，通常不需要填充。这个过程也称为“打包”。

由于类通常共享其成员中最大 alignof的成员的 alignof，类通常会对齐到它们直接或间接包含的最大内置类型的 alignof。

```
// 假设 sizeof(short) == 2, sizeof(int) == 4, sizeof(long long) == 8.
// 假设编译器指定了 4 字节对齐。
struct Char { char c; };
// sizeof(Char)          == 1 (sizeof(char))
struct Int { int i; };
// sizeof(Int)          == 4 (sizeof(int))
struct CharInt { char c; int i; };
// sizeof(CharInt)       == 8 (1 (char) + 3 (填充) + 4 (int))
struct ShortIntCharInt { short s; int i; char c; int j; };
// sizeof(ShortIntCharInt) == 16 (2 (short) + 2 (填充) + 4 (int) + 1 (char) +
//                               3 (填充) + 4 (int))
struct ShortIntCharCharInt { short s; int i; char c; char d; int j; };
// sizeof(ShortIntCharCharInt) == 16 (2 (short) + 2 (填充) + 4 (int) + 1 (char) +
//                                   1 (char) + 2 (填充) + 4 (int))
struct ShortCharShortInt { short s; char c; short t; int i; };
// sizeof(ShortCharShortInt) == 12 (2 (short) + 1 (char) + 1 (填充) + 2 (short) +
//                                 2 (填充) + 4 (int))
struct IntLLInt { int i; long long l; int j; };
// sizeof(IntLLInt)      == 16 (4 (int) + 8 (long long) + 4 (int))
// 如果没有显式指定打包，大多数编译器会按
// 8 字节对齐进行打包，具体如下：
// sizeof(IntLLInt)      == 24 (4 (int) + 4 (填充) + 8 (long long) +
//                           4 (int) + 4 (填充))

// 假设 sizeof(bool) == 1, sizeof(ShortIntCharInt) == 16, sizeof(IntLLInt) == 24.
// 假设默认对齐方式：alignof(ShortIntCharInt) == 4, alignof(IntLLInt) == 8.
struct ShortChar3ArrShortInt {
    short s;
    char c3[3];
    short t;
    int i;
};

// ShortChar3ArrShortInt 具有4字节对齐：alignof(int) >= alignof(char) &&
//                                         alignof(int) >= alignof(short)
// sizeof(ShortChar3ArrShortInt) == 12 (2 (short) + 3 (char[3]) + 1 (填充) +
//                                     2 (short) + 4 (int))
// 注意 t 放置在2字节对齐处，而非4字节。alignof(short) == 2.

struct Large_1 {
    ShortIntCharInt sici;
    bool b;
    ShortIntCharInt tjdj;
};

// Large_1 具有4字节对齐。
// alignof(ShortIntCharInt) == alignof(int) == 4
// alignof(b) == 1
// 因此，alignof(Large_1) == 4.
// sizeof(Large_1) == 36 (16 (ShortIntCharInt) + 1 (bool) + 3 (填充) +
//                      16 (ShortIntCharInt))

struct Large_2 {
    IntLLInt illi;
};
```

alignofs. Due to this, if a member with a smaller alignof is followed by a member with a larger alignof, there is a possibility that the latter member will not be aligned properly if placed immediately after the former. In this case, padding (also known as an *alignment member*) will be placed between the two members, such that the latter member can have its desired alignment. Conversely, if a member with a larger alignof is followed by a member with a smaller alignof, no padding will usually be necessary. This process is also known as "packing".

Due to classes typically sharing the alignof of their member with the largest alignof, classes will typically be aligned to the alignof of the largest built-in type they directly or indirectly contain.

```
// Assume sizeof(short) == 2, sizeof(int) == 4, and sizeof(long long) == 8.
// Assume 4-byte alignment is specified to the compiler.
struct Char { char c; };
// sizeof(Char)          == 1 (sizeof(char))
struct Int { int i; };
// sizeof(Int)          == 4 (sizeof(int))
struct CharInt { char c; int i; };
// sizeof(CharInt)       == 8 (1 (char) + 3 (padding) + 4 (int))
struct ShortIntCharInt { short s; int i; char c; int j; };
// sizeof(ShortIntCharInt) == 16 (2 (short) + 2 (padding) + 4 (int) + 1 (char) +
//                                3 (padding) + 4 (int))
struct ShortIntCharCharInt { short s; int i; char c; char d; int j; };
// sizeof(ShortIntCharCharInt) == 16 (2 (short) + 2 (padding) + 4 (int) + 1 (char) +
//                                   1 (char) + 2 (padding) + 4 (int))
struct ShortCharShortInt { short s; char c; short t; int i; };
// sizeof(ShortCharShortInt) == 12 (2 (short) + 1 (char) + 1 (padding) + 2 (short) +
//                                 2 (padding) + 4 (int))
struct IntLLInt { int i; long long l; int j; };
// sizeof(IntLLInt)      == 16 (4 (int) + 8 (long long) + 4 (int))
// If packing isn't explicitly specified, most compilers will pack this as
// 8-byte alignment, such that:
// sizeof(IntLLInt)      == 24 (4 (int) + 4 (padding) + 8 (long long) +
//                           4 (int) + 4 (padding))

// Assume sizeof(bool) == 1, sizeof(ShortIntCharInt) == 16, and sizeof(IntLLInt) == 24.
// Assume default alignment: alignof(ShortIntCharInt) == 4, alignof(IntLLInt) == 8.
struct ShortChar3ArrShortInt {
    short s;
    char c3[3];
    short t;
    int i;
};

// ShortChar3ArrShortInt has 4-byte alignment: alignof(int) >= alignof(char) &&
//                                         alignof(int) >= alignof(short)
// sizeof(ShortChar3ArrShortInt) == 12 (2 (short) + 3 (char[3]) + 1 (padding) +
//                                     2 (short) + 4 (int))
// Note that t is placed at alignment of 2, not 4. alignof(short) == 2.

struct Large_1 {
    ShortIntCharInt sici;
    bool b;
    ShortIntCharInt tjdj;
};

// Large_1 has 4-byte alignment.
// alignof(ShortIntCharInt) == alignof(int) == 4
// alignof(b) == 1
// Therefore, alignof(Large_1) == 4.
// sizeof(Large_1) == 36 (16 (ShortIntCharInt) + 1 (bool) + 3 (padding) +
//                      16 (ShortIntCharInt))

struct Large_2 {
    IntLLInt illi;
};
```

```

    float f;
    IntLLInt jmmj;
};

// Large_2 具有8字节对齐。
// alignof(IntLLInt) == alignof(long long) == 8
// alignof(float) == 4
// 因此, alignof(Large_2) == 8。
// sizeof(Large_2) == 56 (24 (IntLLInt) + 4 (float) + 4 (填充) + 24 (IntLLInt))

```

版本 ≥ C++11

- 如果使用`alignas`强制严格对齐，即使类型本身较小，也会使用填充字节来满足指定的对齐要求。例如，以下定义中，`Chars<5>`将在末尾插入三个（或更多）填充字节，使其总大小为8。一个对齐为4的类不可能大小为5，因为无法构造该类的数组，因此必须通过插入填充字节将大小“向上取整”为4的倍数。

```

// 该类型应始终对齐到4的倍数。根据需要插入填充。

// Chars<1>..Chars<4>占4字节, Chars<5>..Chars<8>占8字节, 依此类推。
template<size_t SZ>
struct alignas(4) Chars { char arr[SZ]; }; static_assert
t(sizeof(Chars<1>) == sizeof(Chars<4>), "对齐要求严格。");

```

- 如果一个类的两个非静态成员具有相同的访问说明符，则声明顺序靠后的成员保证在对象表示中也靠后。但如果两个非静态成员具有不同的访问说明符，它们在对象中的相对顺序是不确定的。
- 基类子对象在对象中的出现顺序是不确定的，是否连续出现，以及它们是在成员子对象之前、之后还是之间也不确定。

第95.2节：算术类型

窄字符类型

`unsigned char`类型使用所有位来表示二进制数。因此，例如，如果`unsigned char`是8位长，则`char`对象的256种可能的位模式表示256个不同的值{0, 1, ..., 255}。数字42保证由位模式00101010表示。

`signed char`类型没有填充位，即如果`signed char`是8位长，则它有8位容量来表示一个数字。

请注意，这些保证不适用于除窄字符类型以外的其他类型。

整数类型

无符号整数类型使用纯二进制系统，但可能包含填充位。例如，（虽然不太可能）`unsigned int`可能是64位长，但只能存储介于0和 $2^{32} - 1$ 之间的整数，包括端点。其余的32位将是填充位，不应直接写入。

有符号整数类型使用带符号位且可能带有填充位的二进制系统。属于有符号整数类型和对应无符号整数类型的公共范围的值具有相同的表示。例如，如果`unsigned short`对象的位模式0001010010101011表示值5291，那么当将其解释为`short`对象时，也表示值5291。

```

    float f;
    IntLLInt jmmj;
};

// Large_2 has 8-byte alignment.
// alignof(IntLLInt) == alignof(long long) == 8
// alignof(float) == 4
// Therefore, alignof(Large_2) == 8.
// sizeof(Large_2) == 56 (24 (IntLLInt) + 4 (float) + 4 (padding) + 24 (IntLLInt))

```

Version ≥ C++11

- If strict alignment is forced with `alignas`, padding will be used to force the type to meet the specified alignment, even when it would otherwise be smaller. For example, with the definition below, `Chars<5>` will have three (or possibly more) padding bytes inserted at the end so that its total size is 8. It is not possible for a class with an alignment of 4 to have a size of 5 because it would be impossible to make an array of that class, so the size must be "rounded up" to a multiple of 4 by inserting padding bytes.

```

// This type shall always be aligned to a multiple of 4. Padding shall be inserted as
// needed.
// Chars<1>..Chars<4> are 4 bytes, Chars<5>..Chars<8> are 8 bytes, etc.
template<size_t SZ>
struct alignas(4) Chars { char arr[SZ]; };

static_assert(sizeof(Chars<1>) == sizeof(Chars<4>), "Alignment is strict.\n");

```

- If two non-static members of a class have the same access specifier, then the one that comes later in declaration order is guaranteed to come later in the object representation. But if two non-static members have different access specifiers, their relative order within the object is unspecified.
- It is unspecified what order the base class subobjects appear in within an object, whether they occur consecutively, and whether they appear before, after, or between member subobjects.

Section 95.2: Arithmetic types

Narrow character types

The `unsigned char` type uses all bits to represent a binary number. Therefore, for example, if `unsigned char` is 8 bits long, then the 256 possible bit patterns of a `char` object represent the 256 different values {0, 1, ..., 255}. The number 42 is guaranteed to be represented by the bit pattern 00101010.

The `signed char` type has no padding bits, i.e., if `signed char` is 8 bits long, then it has 8 bits of capacity to represent a number.

Note that these guarantees do not apply to types other than narrow character types.

Integer types

The unsigned integer types use a pure binary system, but may contain padding bits. For example, it is possible (though unlikely) for `unsigned int` to be 64 bits long but only be capable of storing integers between 0 and $2^{32} - 1$ inclusive. The other 32 bits would be padding bits, which should not be written to directly.

The signed integer types use a binary system with a sign bit and possibly padding bits. Values that belong to the common range of a signed integer type and the corresponding unsigned integer type have the same representation. For example, if the bit pattern 0001010010101011 of an `unsigned short` object represents the value 5291, then it also represents the value 5291 when interpreted as a `short` object.

是否使用二补码、反码或符号-幅度表示是实现定义的，因为这三种系统都满足前一段中的要求。

浮点类型

浮点类型的值表示是实现定义的。最常见的是，`float`和`double`类型符合IEEE 754标准，分别为32位和64位长（例如，`float`具有23位精度，紧跟8位指数和1位符号位）。然而，标准并不保证任何内容。浮点类型通常具有“陷阱表示”，在计算中使用时会导致错误。

第95.3节：数组

数组类型的元素之间没有填充。因此，元素类型为T的数组只是一个序列按顺序排列在内存中的对象。

多维数组是数组的数组，上述规则递归适用。例如，如果我们有如下声明

```
int a[5][3];
```

那么 `a` 是一个包含5个3个int元素数组的数组。因此，`a[0]`由三个元素 `a[0][0]`、`a[0][1]`、`a[0][2]`组成，它们在内存中的排列顺序在 `a[1]`之前，后者由 `a[1][0]`、`a[1][1]`和 `a[1][2]`组成。这称为行优先顺序。

It is implementation-defined whether a two's complement, one's complement, or sign-magnitude representation is used, since all three systems satisfy the requirement in the previous paragraph.

Floating point types

The value representation of floating point types is implementation-defined. Most commonly, the `float` and `double` types conform to IEEE 754 and are 32 and 64 bits long (so, for example, `float` would have 23 bits of precision which would follow 8 exponent bits and 1 sign bit). However, the standard does not guarantee anything. Floating point types often have "trap representations", which cause errors when they are used in calculations.

Section 95.3: Arrays

An array type has no padding in between its elements. Therefore, an array with element type T is just a sequence of T objects laid out in memory, in order.

A multidimensional array is an array of arrays, and the above applies recursively. For example, if we have the declaration

```
int a[5][3];
```

then `a` is an array of 5 arrays of 3 `ints`. Therefore, `a[0]`, which consists of the three elements `a[0][0]`, `a[0][1]`, `a[0][2]`, is laid out in memory before `a[1]`, which consists of `a[1][0]`, `a[1][1]`, and `a[1][2]`. This is called *row major* order.

第96章：类型推断

本章讨论涉及C++11中可用的关键字auto类型的类型推断。

第96.1节：数据类型：Auto

此示例展示了编译器可以执行的基本类型推断。

```
auto a = 1;          // a = int
auto b = 2u;          // b = unsigned int
auto c = &a;          // c = int*
const auto d = c; // d = const int*
const auto& e = b; // e = const unsigned int&

auto x = a + b      // x = int, #compiler warning unsigned and signed

auto v = std::vector<int>; // v = std::vector<int>
```

然而，auto关键字并不总是在没有额外提示的情况下对&或

const或constexpr执行预期的类型推断

```
// y = 无符号整数,
// 注意y并不推断为const unsigned int&
// 编译器会生成一个副本，而不是对e或b的引用值
auto y = e;
```

第96.2节：Lambda自动调节

数据类型auto关键字是程序员声明lambda函数的一种便捷方式。它通过缩短程序员声明函数指针时需要输入的文本量来提供帮助。

```
auto DoThis = [](int a, int b) { return a + b; };
// DoThis的类型是(int)(*DoThis)(int, int)
// 否则我们就得写得很长
int(*pDoThis)(int, int)= [](int a, int b) { return a + b; };

auto c = Dothis(1, 2); // c = int
auto d = pDothis(1, 2); // d = int

// 使用'auto'缩短了lambda函数的定义
```

默认情况下，如果未定义lambda函数的返回类型，则会根据返回表达式的类型自动推断。

这三者基本上是同一件事

```
[](int a, int b) -> int { return a + b; };
[](int a, int b) -> auto { return a + b; };
[](int a, int b) { return a + b; };
```

第96.3节：循环和auto

此示例展示了如何使用auto来简化for循环中的类型声明

```
std::map<int, std::string> Map;
```

Chapter 96: Type Inference

This topic discusses about type inferencing that involves the keyword `auto` type that is available from C++11.

Section 96.1: Data Type: Auto

This example shows the basic type inferences the compiler can perform.

```
auto a = 1;          // a = int
auto b = 2u;          // b = unsigned int
auto c = &a;          // c = int*
const auto d = c; // d = const int*
const auto& e = b; // e = const unsigned int&

auto x = a + b      // x = int, #compiler warning unsigned and signed

auto v = std::vector<int>; // v = std::vector<int>
```

However, the `auto` keyword does not always perform the expected type inference without additional hints for `&` or `const` or `constexpr`

```
// y = unsigned int,
// note that y does not infer as const unsigned int&
// The compiler would have generated a copy instead of a reference value to e or b
auto y = e;
```

Section 96.2: Lambda auto

The data type `auto` keyword is a convenient way for programmers to declare lambda functions. It helps by shortening the amount of text programmers need to type to declare a function pointer.

```
auto DoThis = [](int a, int b) { return a + b; };
// Do this is of type (int)(*DoThis)(int, int)
// else we would have to write this long
int(*pDoThis)(int, int)= [](int a, int b) { return a + b; };

auto c = Dothis(1, 2); // c = int
auto d = pDothis(1, 2); // d = int

// using 'auto' shortens the definition for lambda functions
```

By default, if the return type of lambda functions is not defined, it will be automatically inferred from the return expression types.

These 3 is basically the same thing

```
[](int a, int b) -> int { return a + b; };
[](int a, int b) -> auto { return a + b; };
[](int a, int b) { return a + b; };
```

Section 96.3: Loops and auto

This example shows how `auto` can be used to shorten type declaration for for loops

```
std::map<int, std::string> Map;
```

```
for (auto pair : Map)           //  pair = std::pair<int, std::string>
for (const auto pair : Map)     //  pair = const std::pair<int, std::string>
for (const auto& pair : Map)    //  pair = const std::pair<int, std::string>&
for (auto i = 0; i < 1000; ++i) //  i = int
for (auto i = 0; i < Map.size(); ++i) //  注意i是int类型, 而不是size_t
for (auto i = Map.size(); i > 0; --i) //  i是size_t类型
```

```
for (auto pair : Map)           //  pair = std::pair<int, std::string>
for (const auto pair : Map)     //  pair = const std::pair<int, std::string>
for (const auto& pair : Map)    //  pair = const std::pair<int, std::string>&
for (auto i = 0; i < 1000; ++i) //  i = int
for (auto i = 0; i < Map.size(); ++i) //  Note that i = int and not size_t
for (auto i = Map.size(); i > 0; --i) //  i = size_t
```

第97章：typedef和类型别名

typedef和（自C++11起）using关键字可用于为已有类型指定新名称。

第97.1节：基本typedef语法

typedef声明的语法与变量或函数声明相同，但包含关键字typedef。

typedef的存在使声明成为类型声明，而非变量或函数声明。

```
int T;           // T的类型是int
typedef int T; // T是int的别名
```

```
int A[100];      // A的类型是“包含100个int的数组”
typedef int A[100]; // A是类型“包含100个int的数组”的别名
```

一旦定义了类型别名，就可以与原始类型名称互换使用。

```
typedef int A[100];
// S是一个包含100个int数组的结构体
struct S {
A data;
};
```

typedef从不创建独立类型。它只是为已有类型提供另一种引用方式。

```
struct S {
    int f(int);
};
typedef int I;
// ok: 定义了 int S::f(int)
I S::f(I x) { return x; }
```

第97.2节：typedef的更复杂用法

typedef 声明与普通变量和函数声明具有相同的语法规则，这可以用来读取和编写更复杂的声明。

```
void (*f)(int);          // f 的类型是“指向参数为int，返回void的函数的指针”
typedef void (*f)(int); // f 是“指向参数为int，返回void的函数的指针”的别名
```

这对于语法复杂的结构特别有用，比如指向非静态成员的指针。

```
void (Foo::*pmf)(int);      // pmf 的类型是“指向Foo的成员函数的指针，该函数参数为int，返回void”
typedef void (Foo::*pmf)(int); // pmf 是“指向Foo的成员函数的指针，该函数参数为int，返回void”的别名
```

即使是有经验的程序员，也很难记住以下函数声明的语法：

```
void (Foo::*f(const char*))(int);
int (&g())[100];
```

typedef 可以用来使它们更易读写：

```
typedef void (Foo::pmf)(int); // pmf 是指向成员函数类型的指针
```

Chapter 97: Typedef and type aliases

The `typedef` and (since C++11) `using` keywords can be used to give a new name to an existing type.

Section 97.1: Basic typedef syntax

A `typedef` declaration has the same syntax as a variable or function declaration, but it contains the word `typedef`. The presence of `typedef` causes the declaration to declare a type instead of a variable or function.

```
int T;           // T has type int
typedef int T; // T is an alias for int

int A[100];      // A has type "array of 100 ints"
typedef int A[100]; // A is an alias for the type "array of 100 ints"
```

Once a type alias has been defined, it can be used interchangeably with the original name of the type.

```
typedef int A[100];
// S is a struct containing an array of 100 ints
struct S {
    A data;
};
```

`typedef` never creates a distinct type. It only gives another way of referring to an existing type.

```
struct S {
    int f(int);
};
typedef int I;
// ok: defines int S::f(int)
I S::f(I x) { return x; }
```

Section 97.2: More complex uses of typedef

The rule that `typedef` declarations have the same syntax as ordinary variable and function declarations can be used to read and write more complex declarations.

```
void (*f)(int);          // f has type "pointer to function of int returning void"
typedef void (*f)(int); // f is an alias for "pointer to function of int returning void"
```

This is especially useful for constructs with confusing syntax, such as pointers to non-static members.

```
void (Foo::*pmf)(int);      // pmf has type "pointer to member function of Foo taking int
                            // and returning void"
typedef void (Foo::*pmf)(int); // pmf is an alias for "pointer to member function of Foo
                            // taking int and returning void"
```

It is hard to remember the syntax of the following function declarations, even for experienced programmers:

```
void (Foo::*f(const char*))(int);
int (&g())[100];
```

`typedef` can be used to make them easier to read and write:

```
typedef void (Foo::pmf)(int); // pmf is a pointer to member function type
```

```
pmf Foo::f(const char*); // f 是 Foo 的成员函数  
typedef int (&ra)[100]; // ra 表示“引用100个int的数组”  
ra g(); // g 返回对100个int数组的引用
```

第97.3节：使用typedef声明多种类型

typedef 关键字是一个说明符，因此它分别应用于每个声明符。因此，每个声明的名称指的是该名称在没有typedef时所具有的类型。

```
int *x, (*p)(); // x 的类型是 int*, p 的类型是 int(*)()  
typedef int *x, (*p)(); // x 是 int* 的别名，而 p 是 int(*)() 的别名
```

第97.4节：“using”别名声明

版本 ≥ C++11

using 的语法非常简单：要定义的名称写在左侧，定义写在右侧。无需扫描名称所在位置。

```
using I = int;  
using A = int[100]; // 100个整数的数组  
using FP = void(*)(int); // 返回void的int参数函数指针  
using MP = void (Foo::*)(int); // Foo类中返回void的int参数成员函数指针
```

使用 using 创建类型别名的效果与使用 typedef 完全相同。这只是实现相同功能的另一种语法。

与 typedef 不同， using 可以模板化。用 using 创建的“模板typedef”称为别名模板。

```
pmf Foo::f(const char*); // f is a member function of Foo  
typedef int (&ra)[100]; // ra means "reference to array of 100 ints"  
ra g(); // g returns reference to array of 100 ints
```

Section 97.3: Declaring multiple types with typedef

The `typedef` keyword is a specifier, so it applies separately to each declarator. Therefore, each name declared refers to the type that that name would have in the absence of `typedef`.

```
int *x, (*p)(); // x has type int*, and p has type int(*)()  
typedef int *x, (*p)(); // x is an alias for int*, while p is an alias for int(*)()
```

Section 97.4: Alias declaration with "using"

Version ≥ C++11

The syntax of `using` is very simple: the name to be defined goes on the left hand side, and the definition goes on the right hand side. No need to scan to see where the name is.

```
using I = int;  
using A = int[100]; // array of 100 ints  
using FP = void(*)(int); // pointer to function of int returning void  
using MP = void (Foo::*)(int); // pointer to member function of Foo of int returning void
```

Creating a type alias with `using` has exactly the same effect as creating a type alias with `typedef`. It is simply an alternative syntax for accomplishing the same thing.

Unlike `typedef`, `using` can be templated. A "template typedef" created with `using` is called an alias template.

第98章：类型推断

第98.1节：构造函数的模板参数推断

在C++17之前，模板推断无法为构造函数推断类类型，必须显式指定。然而，有时这些类型非常繁琐，或者（在lambda的情况下）无法命名，因此出现了大量类型工厂（如 `make_pair()`、`make_tuple()`、`back_inserter()`等）。

版本 ≥ C++17

现在这已不再必要：

```
std::pair p(2, 4.5); // std::pair<int, double>
std::tuple t(4, 3, 2.5); // std::tuple<int, int, double>
std::copy_n(vi1.begin(), 3,
std::back_insert_iterator(vi2)); // 构造一个back_insert_iterator<std::vector<int>>
std::lock_guard lk(mtx); // std::lock_guard<decltype(mtx)>
```

构造函数被认为用于推导类模板参数，但在某些情况下这不足以完成推导，我们可以提供显式的推导指南：

```
template <class 迭代器>
vector(迭代器, 迭代器) -> vector<typename iterator_traits<迭代器>::value_type>

int 数组[] = {1, 2, 3};
std::vector v(std::begin(数组), std::end(数组)); // 推导为 std::vector<int>
```

第98.2节：auto类型推导

版本 ≥ C++11

使用 `auto` 关键字的类型推导几乎与模板类型推导相同。以下是一些示例：

```
auto x = 27; // (x既不是指针也不是引用)，x的类型是intconst auto cx = x; // (cx既不是指
针也不是引用)，cx的类型是const intconst auto& rx = x; // (rx是非通用引用)，rx的类型是对const int的引用

auto&& uref1 = x; // x是int且为左值，所以uref1的类型是int&
auto&& uref2 = cx; // cx是const int且为左值，所以uref2的类型是const int&
auto&& uref3 = 27; // 27是int且为右值，所以uref3的类型是int&&
```

差异总结如下：

```
auto x1 = 27; // 类型是int，值为27
// 类型是int，值为27
auto x2(27); // 类型是st
d::initializer_list<int>, 值为{ 27 }auto x3 = { 27 }; // 类型是std::initializer_list<int>, 值为{
27 }// 在某些编译器中，类型可能被推导为int，值为27。详见备注。
auto x5 = { 1, 2.0 } // 错误！无法推断 std::initializer_list<t> 的类型 T
```

正如你所见，如果使用大括号初始化器，`auto` 会被强制创建为类型为 `std::initializer_list<T>` 的变量。如果无法推断出 `T` 的类型，代码将被拒绝。

Chapter 98: type deduction

Section 98.1: Template parameter deduction for constructors

Prior to C++17, template deduction cannot deduce the class type for you in a constructor. It must be explicitly specified. Sometimes, however, these types can be very cumbersome or (in the case of lambdas) impossible to name, so we got a proliferation of type factories (like `make_pair()`, `make_tuple()`, `back_inserter()`, etc.).

Version ≥ C++17

This is no longer necessary:

```
std::pair p(2, 4.5); // std::pair<int, double>
std::tuple t(4, 3, 2.5); // std::tuple<int, int, double>
std::copy_n(vi1.begin(), 3,
std::back_insert_iterator(vi2)); // constructs a back_insert_iterator<std::vector<int>>
std::lock_guard lk(mtx); // std::lock_guard<decltype(mtx)>
```

Constructors are considered to deduce the class template parameters, but in some cases this is insufficient and we can provide explicit deduction guides:

```
template <class Iter>
vector(Iter, Iter) -> vector<typename iterator_traits<Iter>::value_type>

int array[] = {1, 2, 3};
std::vector v(std::begin(array), std::end(array)); // deduces std::vector<int>
```

Section 98.2: Auto Type Deduction

Version ≥ C++11

Type deduction using the `auto` keyword works almost the same as Template Type Deduction. Below are a few examples:

```
auto x = 27; // (x is neither a pointer nor a reference), x's type is int
const auto cx = x; // (cx is neither a pointer nor a reference), cx's type is const int
const auto& rx = x; // (rx is a non-universal reference), rx's type is a reference to a const
int

auto&& uref1 = x; // x is int and lvalue, so uref1's type is int&
auto&& uref2 = cx; // cx is const int and lvalue, so uref2's type is const int &
auto&& uref3 = 27; // 27 is an int and rvalue, so uref3's type is int&&
```

The differences are outlined below:

```
auto x1 = 27; // type is int, value is 27
auto x2(27); // type is int, value is 27
auto x3 = { 27 }; // type is std::initializer_list<int>, value is { 27 }
auto x4{ 27 }; // type is std::initializer_list<int>, value is { 27 }
// in some compilers type may be deduced as an int with a
// value of 27. See remarks for more information.
auto x5 = { 1, 2.0 } // error! can't deduce T for std::initializer_list<t>
```

As you can see if you use braced initializers, `auto` is forced into creating a variable of type `std::initializer_list<T>`. If it can't deduce the type of `T`, the code is rejected.

当auto用作函数的返回类型时，表示该函数具有尾随返回类型。

```
auto f() -> int {  
    return 42;  
}
```

版本 ≥ C++14

C++14 除了允许 C++11 中的 auto 用法外，还允许以下用法：

1. 当用作没有尾随返回类型的函数的返回类型时，表示函数的返回类型应从函数体中的 return 语句推断（如果有的话）。

```
// f 返回 int:  
auto f() { return 42; }  
// g 返回 void:  
auto g() { std::cout << "hello, world!"; }
```

2. 当用作 lambda 的参数类型时，定义该 lambda 为泛型 lambda。

```
auto triple = [](auto x) { return 3*x; };  
const auto x = triple(42); // x 是值为 126 的 const int
```

特殊形式 decltype(auto) 使用 decltype 的类型推导规则来推导类型，而不是 auto 的规则。

```
int* p = new int(42);  
auto x = *p; // x 的类型是 int  
decltype(auto) y = *p; // y 是对 *p 的引用
```

在 C++03 及更早版本中，auto 关键字作为存储类说明符，其含义与现在完全不同，这是从 C 语言继承而来。

第 98.3 节：模板类型推导

模板泛型语法

```
template<typename T>  
void f(ParamType param);  
  
f(expr);
```

情况 1：ParamType 是引用或指针，但不是通用引用或转发引用。在这种情况下，类型推导按以下方式进行。编译器忽略 expr 中存在的引用部分。然后编译器将 expr 的类型与 ParamType 进行模式匹配，以确定 T。

```
template<typename T>  
void f(T& param); //param 是一个引用  
  
int x = 27; // x 是一个整数  
const int cx = x; // cx 是一个常量整型  
const int& rx = x; // rx 是 x 的常量整型引用  
  
f(x); // T 是 int, 参数类型是 int&  
f(cx); // T 是常量整型, 参数类型是常量整型&  
f(rx); // T 是常量整型, 参数类型是常量整型&
```

When auto is used as the return type of a function, it specifies that the function has a trailing return type.

```
auto f() -> int {  
    return 42;  
}
```

Version ≥ C++14

C++14 允许，除了在 C++11 中允许的 auto 之外，还允许以下用法：

1. When used as the return type of a function without a trailing return type, specifies that the function's return type should be deduced from the return statements in the function's body, if any.

```
// f returns int:  
auto f() { return 42; }  
// g returns void:  
auto g() { std::cout << "hello, world!\n"; }
```

2. When used in the parameter type of a lambda, defines the lambda to be a generic lambda.

```
auto triple = [](auto x) { return 3*x; };  
const auto x = triple(42); // x is a const int with value 126
```

The special form decltype(auto) deduces a type using the type deduction rules of decltype rather than those of auto.

```
int* p = new int(42);  
auto x = *p; // x has type int  
decltype(auto) y = *p; // y is a reference to *p
```

In C++03 和更早版本中，the auto keyword had a completely different meaning as a storage class specifier that was inherited from C.

Section 98.3: Template Type Deduction

Template Generic Syntax

```
template<typename T>  
void f(ParamType param);  
  
f(expr);
```

Case 1: ParamType is a Reference or Pointer, but not a Universal or Forward Reference. In this case type deduction works this way. The compiler ignores the reference part if it exists in expr. The compiler then pattern-matches expr's type against ParamType to determine T.

```
template<typename T>  
void f(T& param); //param is a reference  
  
int x = 27; // x is an int  
const int cx = x; // cx is a const int  
const int& rx = x; // rx is a reference to x as a const int  
  
f(x); // T is int, param's type is int&  
f(cx); // T is const int, param's type is const int&  
f(rx); // T is const int, param's type is const int&
```

情况 2 : ParamType 是通用引用或转发引用。在这种情况下, 如果 expr 是右值, 类型推导与情况 1 相同。如果 expr 是左值, 则 T 和 ParamType 都被推导为左值引用。

```
template<typename T>
void f(T&& param); // param 是通用引用

int x = 27; // x 是一个整型
const int cx = x; // cx 是一个常量整型
const int& rx = x; // rx 是 x 的常量整型引用

f(x); // x 是左值, 所以 T 是 int&, 参数类型也是 int&
f(cx); // cx 是左值, 所以 T 是常量整型&, 参数类型也是常量整型&
f(rx); // rx 是左值, 所以 T 是常量整型&, 参数类型也是常量整型&
f(27); // 27 是右值, 所以 T 是 int, 参数类型因此是 int&&
```

情况 3 : ParamType 既不是指针也不是引用。如果 expr 是引用, 则忽略引用部分。如果 expr 是常量, 也会被忽略。如果是 volatile, 在推导 T 类型时也会被忽略。

```
template<typename T>
void f(T param); // param 现在是按值传递

int x = 27; // x 是一个整型
const int cx = x; // cx 是一个常量整型
const int& rx = x; // rx 是 x 的常量整型引用

f(x); // T 和参数类型都是 int
f(cx); // T 和参数类型仍然都是 int
f(rx); // T 和参数类型依然都是 int
```

Case 2: ParamType is a Universal Reference or Forward Reference. In this case type deduction is the same as in case 1 if the expr is an rvalue. If expr is an lvalue, both T and ParamType are deduced to be lvalue references.

```
template<typename T>
void f(T&& param); // param is a universal reference

int x = 27; // x is an int
const int cx = x; // cx is a const int
const int& rx = x; // rx is a reference to x as a const int

f(x); // x is lvalue, so T is int&, param's type is also int&
f(cx); // cx is lvalue, so T is const int&, param's type is also const int&
f(rx); // rx is lvalue, so T is const int&, param's type is also const int&
f(27); // 27 is rvalue, so T is int, param's type is therefore int&&
```

Case 3: ParamType is Neither a Pointer nor a Reference. If expr is a reference the reference part is ignored. If expr is const that is ignored as well. If it is volatile that is also ignored when deducing T's type.

```
template<typename T>
void f(T param); // param is now passed by value

int x = 27; // x is an int
const int cx = x; // cx is a const int
const int& rx = x; // rx is a reference to x as a const int

f(x); // T's and param's types are both int
f(cx); // T's and param's types are again both int
f(rx); // T's and param's types are still both int
```

第99章：尾随返回类型

第99.1节：避免限定嵌套类型名

```
类 名称非常长的类 {  
    公有：  
        类 迭代器 { /* ... */ };  
        迭代器 end();  
};
```

使用尾随返回类型定义成员end：

```
auto 名称非常长的类::end() -> 迭代器 { 返回 迭代器(); }
```

不使用尾随返回类型定义成员end：

```
名称非常长的类::迭代器 名称非常长的类::end() { 返回 迭代器(); }
```

尾随返回类型在类的作用域中查找，而前置返回类型在封闭的命名空间作用域中查找，因此可能需要“冗余”的限定。

第99.2节：Lambda表达式

lambda 只能有尾置返回类型；前置返回类型语法不适用于 lambda。请注意，在许多情况下，根本不需要为 lambda 指定返回类型。

```
struct Base {};  
struct Derived1 : Base {};  
struct Derived2 : Base {};  
auto lambda = [](bool b) -> Base* { if (b) return new Derived1; else return new Derived2; };  
// ill-formed: auto lambda = Base* [](bool b) { ... };
```

Chapter 99: Trailing return type

Section 99.1: Avoid qualifying a nested type name

```
class ClassWithAReallyLongName {  
public:  
    class Iterator { /* ... */ };  
    Iterator end();  
};
```

Defining the member end with a trailing return type:

```
auto ClassWithAReallyLongName::end() -> Iterator { return Iterator(); }
```

Defining the member end without a trailing return type:

```
ClassWithAReallyLongName::Iterator ClassWithAReallyLongName::end() { return Iterator(); }
```

The trailing return type is looked up in the scope of the class, while a leading return type is looked up in the enclosing namespace scope and can therefore require "redundant" qualification.

Section 99.2: Lambda expressions

A lambda can *only* have a trailing return type; the leading return type syntax is not applicable to lambdas. Note that in many cases it is not necessary to specify a return type for a lambda at all.

```
struct Base {};  
struct Derived1 : Base {};  
struct Derived2 : Base {};  
auto lambda = [](bool b) -> Base* { if (b) return new Derived1; else return new Derived2; };  
// ill-formed: auto lambda = Base* [](bool b) { ... };
```

第100章：对齐

C++ 中所有类型都有对齐要求。这是对该类型对象可创建的内存地址的限制。如果将该地址除以对象的对齐值结果为整数，则该内存地址对对象的创建是有效的。

类型的对齐总是2的幂（包括1）。

第100.1节：控制对齐

版本 ≥ C++11

关键字alignas可以用来强制变量、类数据成员、类的声明或定义，或者枚举的声明或定义具有特定的对齐方式（如果支持）。它有两种形式：

- alignas(x)，其中x是一个常量表达式，赋予实体对齐方式x（如果支持）。
- alignas(T)，其中T是一个类型，赋予实体与T的对齐要求相同的对齐方式，即alignof(T)（如果支持）。

如果对同一实体应用多个alignas说明符，则以最严格的对齐方式为准。

在此示例中，缓冲区buf保证适当对齐以存放一个int对象，尽管其元素类型是unsigned char，该类型的对齐要求可能较弱。

```
alignas(int) unsigned char buf[sizeof(int)];  
new (buf) int(42);
```

alignas不能用来赋予类型比该类型本身更小的对齐方式：

```
alignas(1) int i; //格式错误，除非该平台上的`int`对齐方式为1字节。  
alignas(char) int j; //格式错误，除非`int`的对齐方式与`char`相同或更小。
```

当alignas给出一个整数常量表达式时，必须是有效的对齐方式。有效的对齐方式总是2的幂，且必须大于零。编译器必须支持所有有效对齐方式，直到类型std::max_align_t的对齐要求。它们可能支持比这更大的对齐方式，但对这类对象的内存分配支持有限。对齐方式的上限依赖于具体实现。

C++17在operator new中直接支持为超对齐类型分配内存。

第100.2节：查询类型的对齐方式

版本 ≥ C++11

可以使用alignof关键字作为一元运算符来查询类型的对齐要求。结果是一个类型为std::size_t的常量表达式，即可以在编译时求值。

```
#include <iostream>  
int main() {  
    std::cout << "int的对齐要求是: " << alignof(int) << ";\n";
```

可能的输出

Chapter 100: Alignment

All types in C++ have an alignment. This is a restriction on the memory address that objects of that type can be created within. A memory address is valid for an object's creation if dividing that address by the object's alignment is a whole number.

Type alignments are always a power of two (including 1).

Section 100.1: Controlling alignment

Version ≥ C++11

The `alignas` keyword can be used to force a variable, class data member, declaration or definition of a class, or declaration or definition of an enum, to have a particular alignment, if supported. It comes in two forms:

- `alignas(x)`, where x is a constant expression, gives the entity the alignment x, if supported.
- `alignas(T)`, where T is a type, gives the entity an alignment equal to the alignment requirement of T, that is, `alignof(T)`, if supported.

If multiple `alignas` specifiers are applied to the same entity, the strictest one applies.

In this example, the buffer buf is guaranteed to be appropriately aligned to hold an `int` object, even though its element type is `unsigned char`, which may have a weaker alignment requirement.

```
alignas(int) unsigned char buf[sizeof(int)];  
new (buf) int(42);
```

`alignas` cannot be used to give a type a smaller alignment than the type would have without this declaration:

```
alignas(1) int i; //Il-formed, unless `int` on this platform is aligned to 1 byte.  
alignas(char) int j; //Il-formed, unless `int` has the same or smaller alignment than `char`.
```

`alignas`, when given an integer constant expression, must be given a valid alignment. Valid alignments are always powers of two, and must be greater than zero. Compilers are required to support all valid alignments up to the alignment of the type `std::max_align_t`. They *may* support larger alignments than this, but support for allocating memory for such objects is limited. The upper limit on alignments is implementation dependent.

C++17 features direct support in operator `new` for allocating memory for over-aligned types.

Section 100.2: Querying the alignment of a type

Version ≥ C++11

The alignment requirement of a type can be queried using the `alignof` keyword as a unary operator. The result is a constant expression of type `std::size_t`, i.e., it can be evaluated at compile time.

```
#include <iostream>  
int main() {  
    std::cout << "The alignment requirement of int is: " << alignof(int) << '\n';  
}
```

Possible output

int的对齐要求是: 4

如果应用于数组，则返回元素类型的对齐要求。如果应用于引用类型，则返回被引用类型的对齐要求。（引用本身没有对齐要求，因为它们不是对象。）

The alignment requirement of int is: 4

If applied to an array, it yields the alignment requirement of the element type. If applied to a reference type, it yields the alignment requirement of the referenced type. (References themselves have no alignment, since they are not objects.)

第101章：完美转发

第101.1节：工厂函数

假设我们想编写一个工厂函数，该函数接受任意参数列表，并将这些参数原样传递给另一个函数。这样函数的一个例子是make_unique，它用于安全地构造一个新的T实例并返回一个拥有该实例的unique_ptr<T>。

关于可变模板和右值引用的语言规则允许我们编写这样的函数。

```
template<class T, class... A>
unique_ptr<T> make_unique(A&&... args)
{
    return unique_ptr<T>(new T(std::forward<A>(args)...));
}
```

省略号...表示参数包，代表任意数量的类型。编译器将在调用处将该参数包展开为正确数量的参数。然后，这些参数通过std::forward传递给T的构造函数。该函数需要保持参数的引用限定符。

```
struct foo
{
    foo() {}
    foo(const foo&) {}           // 复制构造函数
    foo(foo&&) {}              // 移动构造函数
    foo(int, int, int) {};
};

foo f;
auto p1 = make_unique<foo>(f);          // 调用 foo::foo(const foo&)
auto p2 = make_unique<foo>(std::move(f)); // 调用 foo::foo(foo&&)
auto p3 = make_unique<foo>(1, 2, 3);
```

Chapter 101: Perfect Forwarding

Section 101.1: Factory functions

Suppose we want to write a factory function that accepts an arbitrary list of arguments and passes those arguments unmodified to another function. An example of such a function is make_unique, which is used to safely construct a new instance of T and return a unique_ptr<T> that owns the instance.

The language rules regarding variadic templates and rvalue references allows us to write such a function.

```
template<class T, class... A>
unique_ptr<T> make_unique(A&&... args)
{
    return unique_ptr<T>(new T(std::forward<A>(args)...));
}
```

The use of ellipses ... indicate a parameter pack, which represents an arbitrary number of types. The compiler will expand this parameter pack to the correct number of arguments at the call site. These arguments are then passed to T's constructor using std::forward. This function is required to preserve the ref-qualifiers of the arguments.

```
struct foo
{
    foo() {}
    foo(const foo&) {}           // copy constructor
    foo(foo&&) {}              // copy constructor
    foo(int, int, int) {};
};

foo f;
auto p1 = make_unique<foo>(f);          // calls foo::foo(const foo&)
auto p2 = make_unique<foo>(std::move(f)); // calls foo::foo(foo&&)
auto p3 = make_unique<foo>(1, 2, 3);
```

第102章 : decltype

关键字 decltype 可以用来获取变量、函数或表达式的类型。

第102.1节：基本示例

此示例仅说明如何使用该关键字。

```
int a = 10;  
  
// 假设此处变量'a'的类型未知，或者可能被程序员更改（例如从int改为long long）。  
  
// 因此我们使用decltype关键字声明另一个相同类型的变量'b'。  
  
decltype(a) b; // 'decltype(a)'的结果是'int'
```

例如，如果有人将'a'的类型更改为：

```
float a=99.0f;
```

那么变量 b 的类型现在会自动变为 float。

第102.2节：另一个例子

假设我们有一个向量：

```
std::vector<int> intVector;
```

我们想为这个向量声明一个迭代器。一个显而易见的想法是使用 auto。然而，有时可能只需要声明一个迭代器变量（而不赋值）。我们会这样做：

```
vector<int>::iterator iter;
```

但是，使用 decltype 会更简单且减少错误（如果 intVector 的类型发生变化）。

```
decltype(intVector)::iterator iter;
```

或者：

```
decltype(intVector.begin()) iter;
```

在第二个例子中，begin 的返回类型用于确定实际类型，即 vector<int>::iterator。

如果我们需要一个 const_iterator，只需使用 cbegin：

```
decltype(intVector.cbegin()) iter; // vector<int>::const_iterator
```

Chapter 102: decltype

The keyword `decltype` can be used to get the type of a variable, function or an expression.

Section 102.1: Basic Example

This example just illustrates how this keyword can be used.

```
int a = 10;  
  
// Assume that type of variable 'a' is not known here, or it may  
// be changed by programmer (from int to long long, for example).  
// Hence we declare another variable, 'b' of the same type using  
// decltype keyword.  
decltype(a) b; // 'decltype(a)' evaluates to 'int'
```

If, for example, someone changes, type of 'a' to:

```
float a=99.0f;
```

Then the type of variable b now automatically becomes `float`.

Section 102.2: Another example

Let's say we have vector:

```
std::vector<int> intVector;
```

And we want to declare an iterator for this vector. An obvious idea is to use `auto`. However, it may be needed just declare an iterator variable (and not to assign it to anything). We would do:

```
vector<int>::iterator iter;
```

However, with `decltype` it becomes easy and less error prone (if type of `intVector` changes).

```
decltype(intVector)::iterator iter;
```

Alternatively:

```
decltype(intVector.begin()) iter;
```

In second example, the return type of begin is used to determine the actual type, which is `vector<int>::iterator`.

If we need a `const_iterator`, we just need to use `cbegin`:

```
decltype(intVector.cbegin()) iter; // vector<int>::const_iterator
```

第103章：SFINAE（替换失败不是错误）

第103.1节：什么是SFINAE

SFINAE代表Substitution Failure Is Not An Error。由于替换类型（或值）来实例化函数模板或类模板而导致的格式错误代码不是严重的编译错误，它仅被视为推导失败。

在实例化函数模板或类模板特化时的推导失败会将该候选项从考虑集合中移除——就好像该失败的候选项从未存在过一样。

```
template <class T>
auto begin(T& c) -> decltype(c.begin()) { return c.begin(); }

template <class T, size_t N>
T* begin(T (&arr)[N]) { return arr; }

int vals[10];
begin(vals); // 好的。第一个函数模板替换失败，因为
             // vals.begin() 是格式错误的。这不是错误！该函数
             // 只是被移除为可行的重载候选，
             // 剩下的是数组重载。
```

只有在立即上下文中的替换失败才被视为推导失败，所有其他情况都被视为严重错误。

```
template <class T>
void add_one(T& val) { val += 1; }

int i = 4;
add_one(i); // 正确

std::string msg = "Hello";
add_one(msg); // 错误。msg += 1 对于 std::string 是格式错误的，但此
              // 失败不在替换 T 的立即上下文中
```

第103.2节：void_t

版本 ≥ C++11

void_t 是一个元函数，将任意（数量的）类型映射为类型 void。void_t 的主要目的是便于编写类型特征。

std::void_t 将成为 C++17 的一部分，但在此之前，实现非常简单：

```
template <class...> using void_t = void;
```

某些编译器需要稍微不同的实现：

```
template <class...>
struct make_void { using type = void; };

template <typename... T>
using void_t = typename make_void<T...>::type;
```

Chapter 103: SFINAE (Substitution Failure Is Not An Error)

Section 103.1: What is SFINAE

SFINAE stands for Substitution Failure Is Not An Error. Ill-formed code that results from substituting types (or values) to instantiate a function template or a class template is **not** a hard compile error, it is only treated as a deduction failure.

Deduction failures on instantiating function templates or class template specializations remove that candidate from the set of consideration - as if that failed candidate did not exist to begin with.

```
template <class T>
auto begin(T& c) -> decltype(c.begin()) { return c.begin(); }

template <class T, size_t N>
T* begin(T (&arr)[N]) { return arr; }

int vals[10];
begin(vals); // OK. The first function template substitution fails because
             // vals.begin() is ill-formed. This is not an error! That function
             // is just removed from consideration as a viable overload candidate,
             // leaving us with the array overload.
```

Only substitution failures in the **immediate context** are considered deduction failures, all others are considered hard errors.

```
template <class T>
void add_one(T& val) { val += 1; }

int i = 4;
add_one(i); // ok

std::string msg = "Hello";
add_one(msg); // error. msg += 1 is ill-formed for std::string, but this
              // failure is NOT in the immediate context of substituting T
```

Section 103.2: void_t

Version ≥ C++11

void_t is a meta-function that maps any (number of) types to type void. The primary purpose of void_t is to facilitate writing of type traits.

std::void_t will be part of C++17, but until then, it is extremely straightforward to implement:

```
template <class...> using void_t = void;
```

Some compilers require a slightly different implementation:

```
template <class...>
struct make_void { using type = void; };

template <typename... T>
using void_t = typename make_void<T...>::type;
```

void_t的主要应用是编写检查语句有效性的类型特征。例如，检查一个类型是否有一个无参数的成员函数foo()：

```
template <class T, class=void>
struct has_foo : std::false_type {};

template <class T>
struct has_foo<T, void_t<decltype(std::declval<T&>().foo())>> : std::true_type {};
```

这是如何工作的？当我尝试实例化has_foo<T>::value时，编译器会尝试寻找has_foo<T, void>的最佳特化。我们有两个选项：主模板和这个涉及实例化底层表达式的次模板：

- 如果T确实有成员函数foo()，那么该函数返回的任意类型都会被转换为void，根据模板部分排序规则，该特化优先于主模板。因此
has_foo<T>::value将为true
- 如果T没有这样的成员函数（或者它需要多个参数），那么该特化的替换失败，我们只能回退到主模板。因此，has_foo<T>::value为false。

一个更简单的例子：

```
template<class T, class=void>
struct can_reference : std::false_type {};

template<class T>
struct can_reference<T, std::void_t<T>> : std::true_type {};
```

这段代码没有使用std::declval或decltype。

你可能注意到一个常见的void参数模式。我们可以将其提取出来：

```
struct details {
    template<template<class...>class Z, class=void, class...Ts>
    struct can_apply : std::false_type
    {};
    template<template<class...>class Z, class...Ts>
    struct can_apply<Z, std::void_t<Z<Ts...>>, Ts...> : std::true_type
    {};
};

template<template<class...>class Z, class...Ts>
using can_apply = details::can_apply<Z, void, Ts...>;
```

它隐藏了std::void_t的使用，使得can_apply表现得像一个指示器，判断作为第一个模板参数提供的类型在将其他类型替换进去后是否是良构的。之前的示例现在可以使用can_apply重写为：

```
template<class T>
using ref_t = T&;

template<class T>
using can_reference = can_apply<ref_t, T>; // T& 对于 T 是否良构？
```

以及：

The primary application of void_t is writing type traits that check validity of a statement. For example, let's check if a type has a member function foo() that takes no arguments:

```
template <class T, class=void>
struct has_foo : std::false_type {};

template <class T>
struct has_foo<T, void_t<decltype(std::declval<T&>().foo())>> : std::true_type {};
```

How does this work? When I try to instantiate has_foo<T>::value, that will cause the compiler to try to look for the best specialization for has_foo<T, void>. We have two options: the primary, and this secondary one which involves having to instantiate that underlying expression:

- If T *does* have a member function foo(), then whatever type that returns gets converted to void, and the specialization is preferred to the primary based on the template partial ordering rules. So has_foo<T>::value will be true
- If T *doesn't* have such a member function (or it requires more than one argument), then substitution fails for the specialization and we only have the primary template to fallback on. Hence, has_foo<T>::value is false.

A simpler case:

```
template<class T, class=void>
struct can_reference : std::false_type {};

template<class T>
struct can_reference<T, std::void_t<T>> : std::true_type {};
```

this doesn't use std::declval or decltype.

You may notice a common pattern of a void argument. We can factor this out:

```
struct details {
    template<template<class...>class Z, class=void, class...Ts>
    struct can_apply : std::false_type
    {};
    template<template<class...>class Z, class...Ts>
    struct can_apply<Z, std::void_t<Z<Ts...>>, Ts...> : std::true_type
    {};
};

template<template<class...>class Z, class...Ts>
using can_apply = details::can_apply<Z, void, Ts...>;
```

which hides the use of std::void_t and makes can_apply act like an indicator whether the type supplied as the first template argument is well-formed after substituting the other types into it. The previous examples may now be rewritten using can_apply as:

```
template<class T>
using ref_t = T&;

template<class T>
using can_reference = can_apply<ref_t, T>; // Is T& well formed for T?
```

and:

```

template<class T>
using dot_foo_r = decltype(std::declval<T&>().foo());

template<class T>
using can_dot_foo = can_apply<dot_foo_r, T>; // T.foo() 对于 T 是否良构?

```

这看起来比原始版本更简单。

有一些后C++17的提案，提出了类似于can_apply的std特性。

void_t的实用性是由沃尔特·布朗（Walter Brown）发现的。他在2016年CppCon大会上对此做了精彩的演讲。

第103.3节：enable_if

std::enable_if 是一个方便的工具，用于使用布尔条件触发SFINAE。它定义为：

```

template <bool Cond, typename Result=void>
struct enable_if { };

template <typename Result>
struct enable_if<true, Result> {
    using type = Result;
};

```

也就是说，enable_if<true, R>::type 是 R 的别名，而 enable_if<false, T>::type 是错误的，因为该 enable_if 特化没有 type 成员类型。

std::enable_if 可以用来约束模板：

```

int negate(int i) { return -i; }

template <class F>
auto negate(F f) { return -f(); }

```

这里，调用 negate(1) 会因歧义而失败。但第二个重载并不打算用于整数类型，因此我们可以添加：

```

int negate(int i) { return -i; }

template <class F, class = typename std::enable_if<!std::is_arithmetic<F>::value>::type>
auto negate(F f) { return -f(); }

```

现在，实例化 negate<int> 会导致替换失败，因为 !std::is_arithmetic<int>::value 为 false。由于 SFINAE，这不是一个严重错误，该候选函数仅仅从重载集合中被移除。因此，negate(1) 只有一个可行的候选函数——然后调用该函数。

何时使用它

值得记住的是，std::enable_if 是基于 SFINAE 的一个辅助工具，但它并不是使 SFINAE 起作用的根本原因。让我们考虑以下两种实现类似于 std::size 功能的替代方案，即一个重载集合 size(arg) 用于获取容器或数组的大小：

```

// 用于容器
template<typename Cont>
auto size1(Cont const& cont) -> decltype(cont.size());

```

```

template<class T>
using dot_foo_r = decltype(std::declval<T&>().foo());

template<class T>
using can_dot_foo = can_apply<dot_foo_r, T>; // Is T.foo() well formed for T?

```

which seems simpler than the original versions.

There are post-C++17 proposals for std traits similar to can_apply.

The utility of void_t was discovered by Walter Brown. He gave a wonderful presentation on it at CppCon 2016.

Section 103.3: enable_if

std::enable_if 是一个方便的工具来使用布尔条件来触发 SFINAE。它被定义为：

```

template <bool Cond, typename Result=void>
struct enable_if { };

template <typename Result>
struct enable_if<true, Result> {
    using type = Result;
};

```

也就是说，enable_if<true, R>::type 是 R 的别名，而 enable_if<false, T>::type 是非法的，因为该 enable_if 特化没有 type 成员类型。

std::enable_if 可以用来约束模板：

```

int negate(int i) { return -i; }

template <class F>
auto negate(F f) { return -f(); }

```

在这里，调用 negate(1) 会因歧义而失败。但第二个重载并不打算用于整数类型，因此我们可以添加：

```

int negate(int i) { return -i; }

template <class F, class = typename std::enable_if<!std::is_arithmetic<F>::value>::type>
auto negate(F f) { return -f(); }

```

现在，实例化 negate<int> 会导致替换失败，因为 !std::is_arithmetic<int>::value 为 false。由于 SFINAE，这不是一个严重错误，该候选函数仅仅从重载集合中被移除。因此，negate(1) 只有一个可行的候选函数——然后调用该函数。

When to use it

值得记住的是，std::enable_if 是一个辅助工具，但它并不是使 SFINAE 起作用的根本原因。让我们考虑以下两种实现类似于 std::size 功能的替代方案，即一个重载集合 size(arg) 用于获取容器或数组的大小：

```

// 用于容器
template<typename Cont>
auto size1(Cont const& cont) -> decltype(cont.size());

```

```

// 用于数组
template<typename Elt, std::size_t Size>
std::size_t size1(Elt const(&arr)[Size]);

// 实现省略
template<typename Cont>
struct is_sizeable;

// 用于容器
template<typename 容器, std::enable_if_t<std::is_sizeable<容器>::value, int> = 0>
auto size2(容器 const& 容器变量);

// 用于数组
template<typename 元素, std::size_t 大小>
std::size_t size2(元素 const(&数组)[大小]);

```

假设 `is_sizeable` 已被正确编写，这两个声明在SFINAE方面应该完全等价。哪一个写起来更简单，哪一个更容易一眼看懂和审查？

现在让我们考虑如何实现避免有符号整数溢出的算术辅助函数，转而采用环绕或模运算行为。也就是说，例如 `incr(i, 3)` 应该等同于 `i += 3`，只是结果总是定义良好的，即使 `i` 是一个值为 `INT_MAX` 的 `int`。这是两种可能的实现方案：

```

// 处理有符号类型
template<typename 整数类型>
auto incr1(整数类型& 目标, 整数类型 数量)
-> std::void_t<int[static_cast<整数类型>(-1) < static_cast<整数类型>(0)]>;

// 处理无符号类型，直接执行 target += amount
// 因为无符号算术已经符合预期行为
template<typename 整数类型>
auto incr1(整数类型& 目标, 整数类型 数量)
-> std::void_t<int[static_cast<整数类型>(0) < static_cast<整数类型>(-1)]>;

template<typename 整数类型, std::enable_if_t<std::is_signed<整数类型>::value, int> = 0>
void incr2(整数类型& 目标, 整数类型 数量);

template<typename 整数类型, std::enable_if_t<std::is_unsigned<整数类型>::value, int> = 0>
void incr2(整数类型& 目标, 整数类型 数量);

```

哪一个写起来最简单，哪一个最容易一目了然地审查和理解？

`std::enable_if` 的一个优势在于它如何与重构和API设计配合。如果 `is_sizeable<Cont>::value` 是用来反映 `cont.size()` 是否有效，那么直接使用表达式本身作为 `size1` 可能更简洁，尽管这可能取决于 `is_sizeable` 是否会在多个地方使用。与此形成对比的是 `std::is_signed`，它比其实现泄露到 `incr1` 声明中更清楚地反映了其意图。

第103.4节：`is_detected`

为了泛化 `type_trait` 的创建：基于 SFINAE，有实验性质的 traits `detected_or`、`detected_t`、`is_detected`。

模板参数为 `typename Default, template <typename...> Op` 和 `typename ... Args`：

- `is_detected`：根据 `Op<Args...>` 的有效性，别名为 `std::true_type` 或 `std::false_type`。
- `detected_t`：根据 `Op<Args...>` 的有效性，别名为 `Op<Args...>` 或 `nonesuch`。

```

// for arrays
template<typename Elt, std::size_t Size>
std::size_t size1(Elt const(&arr)[Size]);

// implementation omitted
template<typename Cont>
struct is_sizeable;

// for containers
template<typename Cont, std::enable_if_t<std::is_sizeable<Cont>::value, int> = 0>
auto size2(Cont const& cont);

// for arrays
template<typename Elt, std::size_t Size>
std::size_t size2(Elt const(&arr)[Size]);

```

Assuming that `is_sizeable` is written appropriately, these two declarations should be exactly equivalent with respect to SFINAE. Which is the easiest to write, and which is the easiest to review and understand at a glance?

Now let's consider how we might want to implement arithmetic helpers that avoid signed integer overflow in favour of wrap around or modular behaviour. Which is to say that e.g. `incr(i, 3)` would be the same as `i += 3` save for the fact that the result would always be defined even if `i` is an `int` with value `INT_MAX`. These are two possible alternatives:

```

// handle signed types
template<typename Int>
auto incr1(Int& target, Int amount)
-> std::void_t<int[static_cast<Int>(-1) < static_cast<Int>(0)]>;

// handle unsigned types by just doing target += amount
// since unsigned arithmetic already behaves as intended
template<typename Int>
auto incr1(Int& target, Int amount)
-> std::void_t<int[static_cast<Int>(0) < static_cast<Int>(-1)]>;

template<typename Int, std::enable_if_t<std::is_signed<Int>::value, int> = 0>
void incr2(Int& target, Int amount);

template<typename Int, std::enable_if_t<std::is_unsigned<Int>::value, int> = 0>
void incr2(Int& target, Int amount);

```

Once again which is the easiest to write, and which is the easiest to review and understand at a glance?

A strength of `std::enable_if` is how it plays with refactoring and API design. If `is_sizeable<Cont>::value` is meant to reflect whether `cont.size()` is valid then just using the expression as it appears for `size1` can be more concise, although that could depend on whether `is_sizeable` would be used in several places or not. Contrast that with `std::is_signed` which reflects its intention much more clearly than when its implementation leaks into the declaration of `incr1`.

Section 103.4: `is_detected`

To generalize `type_trait` creation:based on SFINAE there are experimental traits `detected_or`, `detected_t`, `is_detected`.

With template parameters `typename Default, template <typename...> Op` and `typename ... Args`:

- `is_detected`: alias of `std::true_type` or `std::false_type` depending of the validity of `Op<Args...>`
- `detected_t`: alias of `Op<Args...>` or `nonesuch` depending of validity of `Op<Args...>`.

- detected_or : 别名为一个结构体，包含 value_t (即 is_detected) 和 type (即 Op<Args...> 或 Default) , 取决于 Op<Args...> 的有效性。

可以使用 std::void_t 结合 SFINAE 实现如下：

版本 ≥ C++17

```
namespace detail {
    template <class Default, class AlwaysVoid,
              template<class...> class Op, class... Args>
    struct detector
    {
        using value_t = std::false_type;
        using type = Default;
    };

    template <class Default, template<class...> class Op, class... Args>
    struct detector<Default, std::void_t<Op<Args...>>, Op, Args...>
    {
        using value_t = std::true_type;
        using type = Op<Args...>;
    };
}

} // namespace detail

// special type to indicate detection failure
struct nosuch {
nonesuch() = delete;
~nonesuch() = delete;
nonesuch(nonesuch const&) = delete;
    void operator=(nonesuch const&) = delete;
};

template <template<class...> class Op, class... Args>
using is_detected =
    typename detail::detector<nonesuch, void, Op, Args...>::value_t;

template <template<class...> class Op, class... Args>
using detected_t = typename detail::detector<nonesuch, void, Op, Args...>::type;

template <class Default, template<class...> class Op, class... Args>
using detected_or = detail::detector<Default, void, Op, Args...>;
```

可以简单实现检测方法存在性的特征：

```
typename <typename T, typename ...Ts>
using foo_type = decltype(std::declval<T>().foo(std::declval<Ts>()...));

struct C1 {};

struct C2 {
    int foo(char) const;
};

template <typename T>
using has_foo_char = is_detected<foo_type, T, char>;

static_assert(!has_foo_char<C1>::value, "Unexpected");
static_assert(has_foo_char<C2>::value, "Unexpected");

static_assert(std::is_same<int, detected_t<foo_type, C2, char>>::value,
             "Unexpected");
```

- detected_or: alias of a struct with value_t which is is_detected, and type which is Op<Args...> or Default depending of validity of Op<Args...>

which can be implemented using std::void_t for SFINAE as following:

Version ≥ C++17

```
namespace detail {
    template <class Default, class AlwaysVoid,
              template<class...> class Op, class... Args>
    struct detector
    {
        using value_t = std::false_type;
        using type = Default;
    };

    template <class Default, template<class...> class Op, class... Args>
    struct detector<Default, std::void_t<Op<Args...>>, Op, Args...>
    {
        using value_t = std::true_type;
        using type = Op<Args...>;
    };
}

} // namespace detail

// special type to indicate detection failure
struct nosuch {
nonesuch() = delete;
~nonesuch() = delete;
nonesuch(nonesuch const&) = delete;
    void operator=(nonesuch const&) = delete;
};

template <template<class...> class Op, class... Args>
using is_detected =
    typename detail::detector<nonesuch, void, Op, Args...>::value_t;

template <template<class...> class Op, class... Args>
using detected_t = typename detail::detector<nonesuch, void, Op, Args...>::type;

template <class Default, template<class...> class Op, class... Args>
using detected_or = detail::detector<Default, void, Op, Args...>;
```

Traits to detect presence of method can then be simply implemented:

```
typename <typename T, typename ...Ts>
using foo_type = decltype(std::declval<T>().foo(std::declval<Ts>()...));

struct C1 {};

struct C2 {
    int foo(char) const;
};

template <typename T>
using has_foo_char = is_detected<foo_type, T, char>;

static_assert(!has_foo_char<C1>::value, "Unexpected");
static_assert(has_foo_char<C2>::value, "Unexpected");

static_assert(std::is_same<int, detected_t<foo_type, C2, char>>::value,
             "Unexpected");
```

```

static_assert(std::is_same<void, // Default
             detected_or<void, foo_type, C1, char>>::value,
             "Unexpected");
static_assert(std::is_same<int, detected_or<void, foo_type, C2, char>>::value,
             "Unexpected");

```

第103.5节：具有大量选项的重载解析

如果需要在多个选项之间进行选择，通过 `enable_if<>` 启用其中一个可能相当繁琐，因为还需要对多个条件进行取反。

重载之间的顺序可以通过继承来选择，即标签分发（tag dispatch）。

我们不是测试需要良构的内容，也不是测试所有其他版本条件的取反，而是仅测试我们需要的内容，最好是在尾置返回类型中的 `decltype` 中进行测试。

这可能会使多个选项都良构，我们通过使用“标签”来区分它们，类似于迭代器特征标签（`random_access_tag` 等）。这是可行的，因为直接匹配优于基类，基类又优于基类的基类，依此类推。

```

#include <algorithm>
#include <iterator>

namespace detail
{
    // 这给我们提供了无限的类型，它们彼此继承
    template<std::size_t N>
    struct pick : pick<N-1> {};
    template<>
    struct pick<0> {};

    // 我们希望优先选择的重载在 pick<N> 中具有更高的 N
    // 这是第一个辅助模板函数
    template<typename T>
    auto stable_sort(T& t, pick<2>)
        -> decltype( t.stable_sort(), void() )
    {
        // 如果容器有成员函数 stable_sort，则使用该函数
        t.stable_sort();
    }

    // 该辅助函数将作为第二优先匹配
    template <typename T>;
    auto stable_sort(T & t, pick<1>)
        -> decltype( t.sort(), void() )
    {
        // 如果容器有成员函数 sort，但没有成员函数 stable_sort
        // 通常认为 sort 成员函数是稳定排序
        t.sort();
    }

    // 该辅助函数将作为最后选择
    template <typename T>;
    auto stable_sort(T & t, pick<0>)
        -> decltype(std::stable_sort(std::begin(t), std::end(t)), void())
    {
        // 容器既没有成员函数 sort，也没有成员函数 stable_sort
        std::stable_sort(std::begin(t), std::end(t));
    }
}

```

```

static_assert(std::is_same<void, // Default
             detected_or<void, foo_type, C1, char>>::value,
             "Unexpected");
static_assert(std::is_same<int, detected_or<void, foo_type, C2, char>>::value,
             "Unexpected");

```

Section 103.5: Overload resolution with a large number of options

If you need to select between several options, enabling just one via `enable_if<>` can be quite cumbersome, since several conditions needs to be negated too.

The ordering between overloads can instead be selected using inheritance, i.e. tag dispatch.

Instead of testing for the thing that needs to be well-formed, and also testing the negation of all the other versions conditions, we instead test just for what we need, preferably in a `decltype` in a trailing return.

This might leave several option well formed, we differentiate between those using 'tags', similar to iterator-trait tags (`random_access_tag` et al). This works because a direct match is better than a base class, which is better than a base class of a base class, etc.

```

#include <algorithm>
#include <iterator>

namespace detail
{
    // this gives us infinite types, that inherit from each other
    template<std::size_t N>
    struct pick : pick<N-1> {};
    template<>
    struct pick<0> {};

    // the overload we want to be preferred have a higher N in pick<N>
    // this is the first helper template function
    template<typename T>
    auto stable_sort(T& t, pick<2>)
        -> decltype( t.stable_sort(), void() )
    {
        // if the container have a member stable_sort, use that
        t.stable_sort();
    }

    // this helper will be second best match
    template<typename T>
    auto stable_sort(T& t, pick<1>)
        -> decltype( t.sort(), void() )
    {
        // if the container have a member sort, but no member stable_sort
        // it's customary that the sort member is stable
        t.sort();
    }

    // this helper will be picked last
    template<typename T>
    auto stable_sort(T& t, pick<0>)
        -> decltype( std::stable_sort(std::begin(t), std::end(t)), void() )
    {
        // the container have neither a member sort, nor member stable_sort
        std::stable_sort(std::begin(t), std::end(t));
    }
}

```

```
}
```

```
// 这是用户调用的函数。它将通过“标签”帮助分发调用  
// 到正确的实现。  
template<typename T>  
void stable_sort(T & t)  
{  
    // 使用一个比上面任何使用过的都大的 N。  
    // 这将选择最高的且格式正确的重载。  
    detail::stable_sort(t, detail::pick<10>{});  
}
```

还有其他常用的方法来区分重载，比如精确匹配优于类型转换，类型转换优于省略号。

然而，标签分发可以扩展到任意数量的选择，并且意图更为清晰。

第103.6节：函数模板中的尾随 decltype

版本 ≥ C++11

约束函数之一是使用尾随 decltype 来指定返回类型：

```
命名空间 details {  
    使用 std::to_string;  
  
    // 这个函数的约束是能够调用 to_string(T)  
    模板 <类 T>  
    自动 convert_to_string(T 常量& val, 整数 )  
        -> decltype(to_string(val))  
    {  
        返回 to_string(val);  
    }  
  
    // 这个函数没有约束，但由于省略号参数而不太优先选择  
    模板 <类 T>  
    std::string convert_to_string(T 常量& val, ... )  
    {  
        std::ostringstream oss;  
        oss << val;  
        返回 oss.str();  
    }  
  
    template <class T>  
    std::string convert_to_string(T const& val)  
    {  
        return details::convert_to_string(val, 0);  
    }  
}
```

如果我调用 convert_to_string()，传入一个可以调用 to_string() 的参数，那么 details::convert_to_string() 有两个可行的函数。第一个函数优先，因为从 0 到 int 的转换是比从 0 到 ... 更好的隐式转换序列。

如果我调用 convert_to_string()，传入一个无法调用 to_string() 的参数，那么第一个函数模板实例化会导致替换失败（不存在 decltype(to_string(val)))。因此，该候选函数会从重载集合中移除。第二个函数模板没有约束，因此会被选中，我们将通过 operator<<(std::ostream&, T) 进行处理。如果该操作符未定义，则会在 oss << val 这一行产生编译错误，并显示模板调用栈。

```
}
```

```
// this is the function the user calls. it will dispatch the call  
// to the correct implementation with the help of 'tags'.  
template<typename T>  
void stable_sort(T & t)  
{  
    // use an N that is higher than any used above.  
    // this will pick the highest overload that is well formed.  
    detail::stable_sort(t, detail::pick<10>{});  
}
```

There are other methods commonly used to differentiate between overloads, such as exact match being better than conversion, being better than ellipsis.

However, tag-dispatch can extend to any number of choices, and is a bit more clear in intent.

Section 103.6: trailing decltype in function templates

Version ≥ C++11

One of constraining function is to use trailing decltype to specify the return type:

```
namespace details {  
    using std::to_string;  
  
    // this one is constrained on being able to call to_string(T)  
    模板 <类 T>  
    auto convert_to_string(T const& val, int )  
        -> decltype(to_string(val))  
    {  
        返回 to_string(val);  
    }  
  
    // this one is unconstrained, but less preferred due to the ellipsis argument  
    模板 <类 T>  
    std::string convert_to_string(T const& val, ... )  
    {  
        std::ostringstream oss;  
        oss << val;  
        返回 oss.str();  
    }  
  
    template <class T>  
    std::string convert_to_string(T const& val)  
    {  
        返回 details::convert_to_string(val, 0);  
    }  
}
```

If I call convert_to_string() with an argument with which I can invoke to_string(), then I have two viable functions for details::convert_to_string(). The first is preferred since the conversion from 0 to int is a better implicit conversion sequence than the conversion from 0 to ...

If I call convert_to_string() with an argument from which I cannot invoke to_string(), then the first function template instantiation leads to substitution failure (there is no decltype(to_string(val))). As a result, that candidate is removed from the overload set. The second function template is unconstrained, so it is selected and we instead go through operator<<(std::ostream&, T). If that one is undefined, then we have a hard compile error with a template stack on the line oss << val.

第103.7节 : enable_if_all / enable_if_any

版本 ≥ C++11

动机示例

当你在模板参数列表中有一个可变参数包时，如以下代码片段：

```
template<typename ...Args> void func(Args &&...args) { //... };
```

标准库（C++17之前）没有直接的方法编写enable_if来对Args中的所有参数或任意参数施加SFINAE约束。C++17引入了std::conjunction和std::disjunction解决了这个问题。例如：

```
/// C++17: 对 Args 中所有参数的 SFINAE 约束。  
template<typename ...Args,  
std::enable_if_t<std::conjunction_v<custom_conditions_v<Args>...>>* = nullptr>  
void func(Args &&...args) { //... };
```

```
/// C++17: 对 Args 中任意参数的 SFINAE 约束。  
template<typename ...Args,  
std::enable_if_t<std::disjunction_v<custom_conditions_v<Args>...>>* = nullptr>  
void func(Args &&...args) { //... };
```

如果您没有可用的 C++17，有几种解决方案可以实现这些功能。其中之一是使用基类和部分特化，如本问题的回答中所示。

或者，也可以手动实现std::conjunction和std::disjunction的行为，方法相当直接。下面的示例将演示这些实现，并将它们与std::enable_if结合，生成两个别名：enable_if_all和enable_if_any，它们的语义完全符合预期。这可能提供一个更具扩展性的解决方案。

enable_if_all和enable_if_any的实现

首先让我们使用自定义的seq_and和seq_or分别模拟std::conjunction和std::disjunction：

```
/// 适用于 C++14 之前的辅助工具。  
template<bool B, class T, class F >  
using conditional_t = typename std::conditional<B,T,F>::type;  
  
/// 模拟 C++17 的 std::conjunction.  
template<bool...> struct seq_or: std::false_type {};  
template<bool...> struct seq_and: std::true_type {};  
  
template<bool B1, bool... Bs>  
struct seq_or<B1,Bs...>;  
conditional_t<B1, std::true_type, seq_or<Bs...>> {};  
  
template<bool B1, bool... Bs>  
struct seq_and<B1,Bs...>;  
conditional_t<B1, seq_and<Bs...>, std::false_type> {};
```

然后实现非常直接：

```
template<bool... Bs>  
using enable_if_any = std::enable_if<seq_or<Bs...>::value>;  
  
template<bool... Bs>
```

Section 103.7: enable_if_all / enable_if_any

Version ≥ C++11

Motivational example

When you have a variadic template pack in the template parameters list, like in the following code snippet:

```
template<typename ...Args> void func(Args &&...args) { //... };
```

The standard library (prior to C++17) offers no direct way to write **enable_if** to impose SFINAE constraints on **all of the parameters** in Args or **any of the parameters** in Args. C++17 offers **std::conjunction** and **std::disjunction** which solve this problem. For example:

```
/// C++17: SFINAE constraints on all of the parameters in Args.
```

```
template<typename ...Args,  
std::enable_if_t<std::conjunction_v<custom_conditions_v<Args>...>>* = nullptr>  
void func(Args &&...args) { //... };
```

```
/// C++17: SFINAE constraints on any of the parameters in Args.
```

```
template<typename ...Args,  
std::enable_if_t<std::disjunction_v<custom_conditions_v<Args>...>>* = nullptr>  
void func(Args &&...args) { //... };
```

If you do not have C++17 available, there are several solutions to achieve these. One of them is to use a base-case class and **partial specializations**, as demonstrated in answers of this [question](#).

Alternatively, one may also implement by hand the behavior of std::conjunction and std::disjunction in a rather straight-forward way. In the following example I'll demonstrate the implementations and combine them with std::enable_if to produce two alias: enable_if_all and enable_if_any, which do exactly what they are supposed to semantically. This may provide a more scalable solution.

Implementation of enable_if_all and enable_if_any

First let's emulate std::conjunction and std::disjunction using customized seq_and and seq_or respectively:

```
/// Helper for prior to C++14.  
template<bool B, class T, class F >  
using conditional_t = typename std::conditional<B,T,F>::type;  
  
/// Emulate C++17 std::conjunction.  
template<bool...> struct seq_or: std::false_type {};  
template<bool...> struct seq_and: std::true_type {};  
  
template<bool B1, bool... Bs>  
struct seq_or<B1,Bs...>;  
conditional_t<B1, std::true_type, seq_or<Bs...>> {};  
  
template<bool B1, bool... Bs>  
struct seq_and<B1,Bs...>;  
conditional_t<B1, seq_and<Bs...>, std::false_type> {};
```

Then the implementation is quite straight-forward:

```
template<bool... Bs>  
using enable_if_any = std::enable_if<seq_or<Bs...>::value>;  
  
template<bool... Bs>
```

```
using enable_if_all = std::enable_if<seq_and<Bs...>::value>;
```

最后一些辅助工具：

```
template<bool... Bs>
using enable_if_any_t = typename enable_if_any<Bs...>::type;

template<bool... Bs>
using enable_if_all_t = typename enable_if_all<Bs...>::type;
```

用法

用法也很直接：

```
/// 对 Args 中所有参数的 SFINAE 约束。
template<typename ...Args,
enable_if_all_t<custom_conditions_v<Args>...*>* = nullptr>
void func(Args &&...args) { //... };

/// 对 Args 中任一参数的 SFINAE 约束。
template<typename ...Args,
enable_if_any_t<custom_conditions_v<Args>...*>* = nullptr>
void func(Args &&...args) { //... };
```

```
using enable_if_all = std::enable_if<seq_and<Bs...>::value>;
```

Eventually some helpers:

```
template<bool... Bs>
using enable_if_any_t = typename enable_if_any<Bs...>::type;

template<bool... Bs>
using enable_if_all_t = typename enable_if_all<Bs...>::type;
```

Usage

The usage is also straight-forward:

```
/// SFINAE constraints on all of the parameters in Args.
template<typename ...Args,
enable_if_all_t<custom_conditions_v<Args>...*>* = nullptr>
void func(Args &&...args) { //... };

/// SFINAE constraints on any of the parameters in Args.
template<typename ...Args,
enable_if_any_t<custom_conditions_v<Args>...*>* = nullptr>
void func(Args &&...args) { //... };
```

第104章：未定义行为

什么是未定义行为（UB）？根据 ISO C++ 标准（§1.3.24, N4296），它是“本国际标准不对其施加任何要求的行为”。

这意味着当程序遇到未定义行为（UB）时，它可以做任何它想做的事情。这通常意味着程序崩溃，但它也可能什么都不做，让恶魔从你的鼻子里飞出来，甚至看起来正常工作！

不用说，你应该避免编写会触发未定义行为的代码。

第104.1节：通过空指针读取或写入

```
int *ptr = nullptr;  
*ptr = 1; // 未定义行为
```

这是未定义行为，因为空指针不指向任何有效对象，所以在*ptr处没有对象可写入。

虽然这通常会导致段错误，但它是未定义的，任何情况都有可能发生。

第104.2节：使用未初始化的局部变量

```
int a;  
std::cout << a; // 未定义行为！
```

这会导致未定义行为，因为 a 未初始化。

人们常常错误地声称这是因为值是“未确定的”，或者是“之前该内存位置中的任意值”。然而，正是访问上述示例中 a 的值这一行为导致了未定义行为。实际上，打印“垃圾值”是这种情况下常见的症状，但这只是未定义行为的众多可能表现之一。

虽然在实际中极不可能（因为它依赖于特定的硬件支持），但编译器同样可能在编译上述代码示例时电击程序员。拥有这样的编译器和硬件支持，对未定义行为的这种响应将显著提高程序员对未定义行为真实含义的平均（存活）理解——即标准对结果行为不作任何约束。

版本 ≥ C++14

使用不确定值的 `unsigned char` 类型时，如果该值用作以下情况，不会产生未定义行为：

- 三元条件运算符的第二或第三个操作数；
- 内置逗号运算符的右操作数；
- 转换为 `unsigned char` 的操作数；
- 赋值运算符的右操作数（如果左操作数也是 `unsigned char` 类型）；
- `unsigned char` 对象的初始化值；

或者该值被丢弃。在这些情况下，不确定值仅在表达式结果中传播（如果适用）。

注意，`static` 变量总是被零初始化（如果可能）：

Chapter 104: Undefined Behavior

What is undefined behavior (UB)? According to the ISO C++ Standard (§1.3.24, N4296), it is "behavior for which this International Standard imposes no requirements."

This means that when a program encounters UB, it is allowed to do whatever it wants. This often means a crash, but it may simply do nothing, make demons fly out of your nose, or even appear to work properly!

Needless to say, you should avoid writing code that invokes UB.

Section 104.1: Reading or writing through a null pointer

```
int *ptr = nullptr;  
*ptr = 1; // Undefined behavior
```

This is **undefined behavior**, because a null pointer does not point to any valid object, so there is no object at *ptr to write to.

Although this most often causes a segmentation fault, it is undefined and anything can happen.

Section 104.2: Using an uninitialized local variable

```
int a;  
std::cout << a; // Undefined behavior!
```

This results in **undefined behavior**, because a is uninitialized.

It is often, incorrectly, claimed that this is because the value is "indeterminate", or "whatever value was in that memory location before". However, it is the act of accessing the value of a in the above example that gives undefined behaviour. In practice, printing a "garbage value" is a common symptom in this case, but that is only one possible form of undefined behaviour.

Although highly unlikely in practice (since it is reliant on specific hardware support) the compiler could equally well electrocute the programmer when compiling the code sample above. With such a compiler and hardware support, such a response to undefined behaviour would markedly increase average (living) programmer understanding of the true meaning of undefined behaviour - which is that the standard places no constraint on the resultant behaviour.

Version ≥ C++14

Using an indeterminate value of `unsigned char` type does not produce undefined behavior if the value is used as:

- the second or third operand of the ternary conditional operator;
- the right operand of the built-in comma operator;
- the operand of a conversion to `unsigned char`;
- the right operand of the assignment operator, if the left operand is also of type `unsigned char`;
- the initializer for an `unsigned char` object;

or if the value is discarded. In such cases, the indeterminate value simply propagates to the result of the expression, if applicable.

Note that a `static` variable is **always** zero-initialized (if possible):

```
static int a;  
std::cout << a; // 定义良好的行为, 'a' 为 0
```

第104.3节：访问越界索引

访问数组（或标准库容器，因为它们都是使用原始数组实现的）中越界的索引是未定义行为：

```
int array[] = {1, 2, 3, 4, 5};  
array[5] = 0; // 未定义行为
```

允许指针指向数组末尾（在此例中为`array + 5`），但不能解引用它，因为它不是有效元素。

```
const int *end = array + 5; // 指向最后一个索引之后的位置的指针  
for (int *p = array; p != end; ++p)  
    // 对 `p` 进行操作
```

一般来说，不允许创建越界指针。指针必须指向数组内的元素，或指向数组末尾之后的一个位置。

第104.4节：通过指向没有虚析构函数的基类的指针删除派生对象

```
class base {};  
class derived: public base {};  
  
int main() {  
    base* p = new derived();  
    delete p; // 这是未定义行为！  
}
```

在章节 [expr.delete] §5.3.5/3 中，标准指出如果对一个静态类型没有virtual析构函数的对象调用`delete`：

如果要删除的对象的静态类型与其动态类型不同，则静态类型应为该对象动态类型的基类，并且静态类型应具有虚析构函数，否则行为未定义。

无论派生类是否向基类添加了任何数据成员，情况均如此。

第104.5节：扩展 `std` 或 `posix` 命名空间

标准 (17.6.4.2.1/1) 通常禁止扩展 std 命名空间：

如果 C++ 程序向命名空间 std 或 std 命名空间内的某个命名空间添加声明或定义，除非另有说明，否则其行为未定义。

对于 posix 也是同样的规定 (17.6.4.2.2/1)：

```
static int a;  
std::cout << a; // Defined behavior, 'a' is 0
```

Section 104.3: Accessing an out-of-bounds index

It is **undefined behavior** to access an index that is out of bounds for an array (or standard library container for that matter, as they are all implemented using a *raw array*):

```
int array[] = {1, 2, 3, 4, 5};  
array[5] = 0; // Undefined behavior
```

It is *allowed* to have a pointer pointing to the end of the array (in this case `array + 5`), you just can't dereference it, as it is not a valid element.

```
const int *end = array + 5; // Pointer to one past the last index  
for (int *p = array; p != end; ++p)  
    // Do something with `p`
```

In general, you're not allowed to create an out-of-bounds pointer. A pointer must point to an element within the array, or one past the end.

Section 104.4: Deleting a derived object via a pointer to a base class that doesn't have a virtual destructor

```
class base {};  
class derived: public base {};  
  
int main() {  
    base* p = new derived();  
    delete p; // The is undefined behavior!  
}
```

In section [expr.delete] §5.3.5/3 the standard says that if `delete` is called on an object whose static type does not have a `virtual` destructor:

if the static type of the object to be deleted is different from its dynamic type, the static type shall be a base class of the dynamic type of the object to be deleted and the static type shall have a virtual destructor or the behavior is undefined.

This is the case regardless of the question whether the derived class added any data members to the base class.

Section 104.5: Extending the `std` or `posix` Namespace

The standard (17.6.4.2.1/1) generally forbids extending the std namespace:

The behavior of a C++ program is undefined if it adds declarations or definitions to namespace std or to a namespace within namespace std unless otherwise specified.

The same goes for posix (17.6.4.2.2/1):

如果 C++ 程序向命名空间 `posix` 或 `posix` 命名空间内的某个命名空间添加声明或定义，除非另有说明，否则其行为未定义。

The behavior of a C++ program is undefined if it adds declarations or definitions to namespace `posix` or to a namespace within namespace `posix` unless otherwise specified.

考虑以下情况：

```
#include <algorithm>

namespace std
{
    int foo(){}}
}
```

标准中没有禁止算法（或其包含的某个头文件）定义相同的定义，因此这段代码将违反“一定义规则”。

因此，通常这是被禁止的。不过有一些特定的例外允许。也许最有用的是，允许为用户定义类型添加特化。例如，假设你的代码有

```
类 foo
{
    // 内容
};
```

那么下面的写法是可以的

```
命名空间 std
{
    模板<>
    结构体 hash<foo>
    {
        公共:
            size_t operator()(const foo &f) const;
    };
}
```

第104.6节：无效的指针运算

以下指针运算的使用会导致未定义行为：

- 整数的加减运算，如果结果不属于与指针操作数相同的数组对象。（这里，数组末尾之后的一个元素仍被视为属于该数组。）

```
int a[10];
int* p1 = &a[5];
int* p2 = p1 + 4; // 正确；p2 指向 a[9]
int* p3 = p1 + 5; // 正确；p3 指向数组末尾之后的位置
int* p4 = p1 + 6; // 未定义行为
int* p5 = p1 - 5; // 正确；p5 指向 a[0]
int* p6 = p1 - 6; // 未定义行为
int* p7 = p3 - 5; // 正确；p7 指向 a[5]
```

- 两个指针相减，前提是它们必须都属于同一个数组对象。（同样，数组末尾之后的元素也被视为属于该数组。）例外情况是两个空指针可以相减，结果为 0。

Consider the following:

```
#include <algorithm>

namespace std
{
    int foo(){}}
}
```

Nothing in the standard forbids `algorithm` (or one of the headers it includes) defining the same definition, and so this code would violate the [One Definition Rule](#).

So, in general, this is forbidden. There are [specific exceptions allowed](#), though. Perhaps most usefully, it is allowed to add specializations for user defined types. So, for example, suppose your code has

```
class foo
{
    // Stuff
};
```

Then the following is fine

```
namespace std
{
    模板<>
    结构体 hash<foo>
    {
        公共:
            size_t operator()(const foo &f) const;
    };
}
```

Section 104.6: Invalid pointer arithmetic

The following uses of pointer arithmetic cause undefined behavior:

- Addition or subtraction of an integer, if the result does not belong to the same array object as the pointer operand. (Here, the element one past the end is considered to still belong to the array.)

```
int a[10];
int* p1 = &a[5];
int* p2 = p1 + 4; // ok; p2 points to a[9]
int* p3 = p1 + 5; // ok; p2 points to one past the end of a
int* p4 = p1 + 6; // UB
int* p5 = p1 - 5; // ok; p2 points to a[0]
int* p6 = p1 - 6; // UB
int* p7 = p3 - 5; // ok; p7 points to a[5]
```

- Subtraction of two pointers if they do not both belong to the same array object. (Again, the element one past the end is considered to belong to the array.) The exception is that two null pointers may be subtracted, yielding 0.

```

int a[10];
int b[10];
int *p1 = &a[8], *p2 = &a[3];
int d1 = p1 - p2; // 结果为 5
int *p3 = p1 + 2; // 正确; p3 指向数组末尾之后的一个位置
int d2 = p3 - p2; // 结果为 7
int *p4 = &b[0];
int d3 = p4 - p1; // 未定义行为 (UB)

```

- 两个指针相减如果结果溢出，返回类型为 `std::ptrdiff_t`。
- 任何指针算术运算中，如果任一操作数所指向的类型与其所指对象的动态类型不匹配（忽略 `cv` 限定符），则不符合标准。根据标准，“特别地，当数组包含派生类类型的对象时，不能对指向基类的指针进行指针算术运算。”

```

struct Base { int x; };
struct Derived : Base { int y; };
Derived a[10];
Base* p1 = &a[1];           // 正确
Base* p2 = p1 + 1;          // 未定义行为 (UB); p1 指向 Derived
Base* p3 = p1 - 1;          // 同上
Base* p4 = &a[2];           // 正确
auto p5 = p4 - p1;          // 未定义行为 (UB); p4 和 p1 指向 Derived
const Derived* p6 = &a[1];
const Derived* p7 = p6 + 1; // 正确; cv 限定符无关紧要

```

```

int a[10];
int b[10];
int *p1 = &a[8], *p2 = &a[3];
int d1 = p1 - p2; // yields 5
int *p3 = p1 + 2; // ok; p3 points to one past the end of a
int d2 = p3 - p2; // yields 7
int *p4 = &b[0];
int d3 = p4 - p1; // UB

```

- Subtraction of two pointers if the result overflows `std::ptrdiff_t`.
- Any pointer arithmetic where either operand's pointee type does not match the dynamic type of the object pointed to (ignoring `cv`-qualification). According to the standard, "[in] particular, a pointer to a base class cannot be used for pointer arithmetic when the array contains objects of a derived class type."

```

struct Base { int x; };
struct Derived : Base { int y; };
Derived a[10];
Base* p1 = &a[1];           // ok
Base* p2 = p1 + 1;          // UB; p1 points to Derived
Base* p3 = p1 - 1;          // likewise
Base* p4 = &a[2];           // ok
auto p5 = p4 - p1;          // UB; p4 and p1 point to Derived
const Derived* p6 = &a[1];
const Derived* p7 = p6 + 1; // ok; cv-qualifiers don't matter

```

第 104.7 节：非 `void` 返回类型的函数缺少返回语句

在返回类型非 `void` 的函数中省略 `return` 语句是 未定义行为。

```

int function() {
    // 缺少返回语句
}

int main() {
function(); // 未定义行为
}

```

大多数现代编译器在编译时会对这种未定义行为发出警告。

注意：`main` 是唯一的例外。如果`main`没有`return`语句，编译器会自动插入`return 0;`，因此可以安全省略。

第104.8节：访问悬空引用

访问已超出作用域或已被销毁对象的引用是非法的。这样的引用称为 **悬空引用**，因为它不再指向有效对象。

```

#include <iostream>
int& getX() {
    int x = 42;
    return x;
}

```

Section 104.7: No return statement for a function with a non-void return type

Omitting the `return` statement in a function which has a return type that is not `void` is **undefined behavior**.

```

int function() {
    // Missing return statement
}

int main() {
    function(); // Undefined Behavior
}

```

Most modern day compilers emit a warning at compile time for this kind of undefined behavior.

Note: `main` is the only exception to the rule. If `main` doesn't have a `return` statement, the compiler automatically inserts `return 0;` for you, so it can be safely left out.

Section 104.8: Accessing a dangling reference

It is illegal to access a reference to an object that has gone out of scope or been otherwise destroyed. Such a reference is said to be **dangling** since it no longer refers to a valid object.

```

#include <iostream>
int& getX() {
    int x = 42;
    return x;
}

```

```
int main() {
    int& r = getX();
    std::cout << r << "";
}
```

在此示例中，局部变量 `x` 在 `getX` 返回时超出作用域。（注意，生命周期延长不能将局部变量的生命周期延长到其定义块的作用域之外。）因此，`r` 是一个悬空引用。该程序具有未定义行为，尽管在某些情况下它可能看起来能正常工作并打印42。

第104.9节：整数除以零

```
int x = 5 / 0; // 未定义行为
```

除以0在数学上是未定义的，因此这是一种未定义行为是合理的。

但是：

```
float x = 5.0f / 0.0f; // x 是 +无穷大
```

大多数实现采用IEEE-754标准，该标准定义浮点数除以零时返回NaN（如果分子是0.0f）、无穷大（如果分子为正）或-无穷大（如果分子为负）。

第104.10节：按无效位数移位

对于内置的移位运算符，右操作数必须是非负且严格小于提升后的左操作数的位宽。否则，行为是未定义的。

```
const int a = 42;
const int b = a << -1; // UB
const int c = a << 0; // 正确
const int d = a << 32; // 如果 int 是 32 位或更少则未定义行为
const int e = a >> 32; // 如果 int 是 32 位或更少也未定义行为
const signed char f = 'x';
const int g = f << 10; // 即使 signed char 是 10 位或更少也正确;
                      // int 必须至少是 16 位
```

第 104.11 节：内存分配和释放的不正确配对

只有当对象是由 `new` 分配且不是数组时，才能用 `delete` 释放。如果传给 `delete` 的参数不是由 `new` 返回的或者是数组，则行为未定义。

只有当对象是由 `new` 分配且是数组时，才能用 `delete[]` 释放。如果传给 `delete[]` 的参数不是由 `new` 返回的或者不是数组，则行为未定义。

如果传给 `free` 的参数不是由 `malloc` 返回的，则行为未定义。

```
int* p1 = new int;
delete p1; // 正确
// delete[] p1; // 未定义
// free(p1); // 未定义

int* p2 = new int[10];
delete[] p2; // 正确
// delete p2; // 未定义
// free(p2); // 未定义
```

```
int main() {
    int& r = getX();
    std::cout << r << "\n";
}
```

In this example, the local variable `x` goes out of scope when `getX` returns. (Note that *lifetime extension* cannot extend the lifetime of a local variable past the scope of the block in which it is defined.) Therefore `r` is a dangling reference. This program has undefined behavior, although it may appear to work and print 42 in some cases.

Section 104.9: Integer division by zero

```
int x = 5 / 0; // Undefined behavior
```

Division by 0 is mathematically undefined, and as such it makes sense that this is undefined behavior.

However:

```
float x = 5.0f / 0.0f; // x is +infinity
```

Most implementation implement IEEE-754, which defines floating point division by zero to return NaN (if numerator is 0.0f), infinity (if numerator is positive) or -infinity (if numerator is negative).

Section 104.10: Shifting by an invalid number of positions

For the built-in shift operator, the right operand must be nonnegative and strictly less than the bit width of the promoted left operand. Otherwise, the behavior is undefined.

```
const int a = 42;
const int b = a << -1; // UB
const int c = a << 0; // ok
const int d = a << 32; // UB if int is 32 bits or less
const int e = a >> 32; // also UB if int is 32 bits or less
const signed char f = 'x';
const int g = f << 10; // ok even if signed char is 10 bits or less;
                      // int must be at least 16 bits
```

Section 104.11: Incorrect pairing of memory allocation and deallocation

An object can only be deallocated by `delete` if it was allocated by `new` and is not an array. If the argument to `delete` was not returned by `new` or is an array, the behavior is undefined.

An object can only be deallocated by `delete[]` if it was allocated by `new` and is an array. If the argument to `delete[]` was not returned by `new` or is not an array, the behavior is undefined.

If the argument to `free` was not returned by `malloc`, the behavior is undefined.

```
int* p1 = new int;
delete p1; // correct
// delete[] p1; // undefined
// free(p1); // undefined

int* p2 = new int[10];
delete[] p2; // correct
// delete p2; // undefined
// free(p2); // undefined
```

```
int* p3 = static_cast<int*>(malloc(sizeof(int)));
free(p3); // 正确
// delete p3; // 未定义
// delete[] p3; // 未定义
```

通过在 C++ 程序中完全避免使用 malloc 和 free，可以避免此类问题，优先使用标准库的智能指针代替原始的 new 和 delete，优先使用 std::vector 和 std::string 代替原始的 new 和 delete[]。

第 104.12 节：有符号整数溢出

```
int x = INT_MAX + 1;
// x 可以是任何值 -> 未定义行为
```

如果在表达式求值过程中，结果在数学上未定义或不在其类型可表示的值范围内，则行为是未定义的。

(C++11 标准 第 5 章第 4 节)

这是比较棘手的问题之一，因为它通常会产生可重现的、不会崩溃的行为，因此开发者可能会倾向于过度依赖观察到的行为。

另一方面：

```
unsigned int x = UINT_MAX + 1;
// x 是 0
```

定义良好，因为：

声明为无符号的无符号整数，应遵守模 2^n 的算术法则，其中 n 是该特定大小整数值表示中的位数。

(C++11 标准第3.9.1/4段)

有时编译器可能利用未定义行为进行优化

```
有符号整数 x ;
如果(x > x + 1)
{
    //执行某些操作
}
```

这里由于有符号整数溢出未定义，编译器可以假设它永远不会发生，因此可以优化掉“if”代码块

第104.13节：多个非相同定义（唯一定义规则）

如果一个类、枚举、内联函数、模板或模板成员具有外部链接并在多个翻译单元中定义，则所有定义必须相同，否则根据唯一定义规则行为未定义

```
int* p3 = static_cast<int*>(malloc(sizeof(int)));
free(p3); // correct
// delete p3; // undefined
// delete[] p3; // undefined
```

Such issues can be avoided by completely avoiding `malloc` and `free` in C++ programs, preferring the standard library smart pointers over raw `new` and `delete`, and preferring `std::vector` and `std::string` over raw `new` and `delete[]`.

Section 104.12: Signed Integer Overflow

```
int x = INT_MAX + 1;
// x can be anything -> Undefined behavior
```

If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type, the behavior is undefined.

(C++11 Standard paragraph 5/4)

This is one of the more nasty ones, as it usually yields reproducible, non-crashing behavior so developers may be tempted to rely heavily on the observed behavior.

On the other hand:

```
unsigned int x = UINT_MAX + 1;
// x is 0
```

is well defined since:

Unsigned integers, declared unsigned, shall obey the laws of arithmetic modulo 2^n where n is the number of bits in the value representation of that particular size of integer.

(C++11 Standard paragraph 3.9.1/4)

Sometimes compilers may exploit an undefined behavior and optimize

```
signed int x ;
if(x > x + 1)
{
    //do something
}
```

Here since a signed integer overflow is not defined, compiler is free to assume that it may never happen and hence it can optimize away the "if" block

Section 104.13: Multiple non-identical definitions (the One Definition Rule)

If a class, enum, inline function, template, or member of a template has external linkage and is defined in multiple translation units, all definitions must be identical or the behavior is undefined according to the [One Definition Rule](#)

foo.h:

```
class Foo {
public:
    double x;
private:
    int y;
};
```

Foo get_foo()

foo.cpp:

```
#include "foo.h"
Foo get_foo() /* 实现 */
```

main.cpp:

```
// 我想访问私有成员，所以我打算用我自己的类型替换 Foo
class Foo {
public:
    double x;
    int y;
};
Foo get_foo(); // 自己声明这个函数，因为我们没有包含 foo.h
int main() {
    Foo foo = get_foo();
    // 对 foo.y 进行操作
}
```

上述程序表现出未定义行为，因为它在不同的翻译单元中包含了类::Foo的两个定义，该类具有外部链接性，但这两个定义并不相同。与在同一翻译单元内重新定义类不同，编译器不要求诊断此问题。

第104.14节：修改const对象

任何试图修改const对象的行为都是未定义的。这适用于const变量、const对象的成员以及声明为const的类成员。（但是，const对象的mutable成员不是const。）

这种尝试可以通过const_cast实现：

```
const int x = 123; const_cast<int&>(x) = 456; std::cout
<< x << ";
```

编译器通常会内联const int对象的值，因此这段代码可能编译通过并打印123。

编译器也可能将const对象的值放在只读内存中，因此可能会发生段错误。无论如何，行为都是未定义的，程序可能表现出任何行为。

下面的程序隐藏了一个更为微妙的错误：

```
#include <iostream>

class Foo* instance;
```

foo.h:

```
class Foo {
public:
    double x;
private:
    int y;
};
```

Foo get_foo();

foo.cpp:

```
#include "foo.h"
Foo get_foo() /* implementation */
```

main.cpp:

```
// I want access to the private member, so I am going to replace Foo with my own type
class Foo {
public:
    double x;
    int y;
};
Foo get_foo(); // declare this function ourselves since we aren't including foo.h
int main() {
    Foo foo = get_foo();
    // do something with foo.y
}
```

The above program exhibits undefined behavior because it contains two definitions of the class ::Foo, which has external linkage, in different translation units, but the two definitions are not identical. Unlike redefinition of a class within the same translation unit, this problem is not required to be diagnosed by the compiler.

Section 104.14: Modifying a const object

Any attempt to modify a const object results in undefined behavior. This applies to const variables, members of const objects, and class members declared const. (However, a mutable member of a const object is not const.)

Such an attempt can be made through const_cast:

```
const int x = 123;
const_cast<int&>(x) = 456;
std::cout << x << '\n';
```

A compiler will usually inline the value of a const int object, so it's possible that this code compiles and prints 123. Compilers can also place const objects' values in read-only memory, so a segmentation fault may occur. In any case, the behavior is undefined and the program might do anything.

The following program conceals a far more subtle error:

```
#include <iostream>

class Foo* instance;
```

```

class Foo {
public:
    int get_x() const { return m_x; }
    void set_x(int x) { m_x = x; }
private:
Foo(int x, Foo*& this_ref): m_x(x) {
    this_ref = this;
}
int m_x;
friend const Foo& getFoo();
};

const Foo& getFoo() {
    static const Foo foo(123, instance);
    return foo;
}

void do_evil(int x) {
    instance->set_x(x);
}

int main() {
    const Foo& foo = getFoo();
    do_evil(456);
std::cout << foo.get_x() << ";"
}

```

在这段代码中，getFoo 创建了一个类型为 `const Foo` 的单例，其成员 `m_x` 被初始化为 123。然后调用了 `do_evil`，`foo.m_x` 的值显然被改为 456。哪里出了问题？

尽管名字叫 `do_evil`，但它并没有做什么特别恶意的事情；它所做的只是通过一个 `Foo*` 调用一个设置函数。但该指针指向一个 `const Foo` 对象，尽管没有使用 `const_cast`。这个指针是通过 `Foo` 的构造函数获得的。一个 `const` 对象在初始化完成之前并不是真正的 `const`，因此在构造函数内，`this` 的类型是 `Foo*`，而不是 `const Foo*`。

因此，即使程序中没有明显危险的构造，也会发生未定义行为。

第104.15节：从[[noreturn]]函数返回

版本 ≥ C++11

标准示例，[dcl.attr.noreturn]：

```

[[ noreturn ]] void f() {
    throw "error"; // 正确
}
[[ noreturn ]] void q(int i) { // 如果参数 <= 0, 行为未定义
    if (i > 0)
        throw "positive";
}

```

第104.16节：无限模板递归

标准示例，[temp.inst]/17：

```

template<class T> class X {
    X<T*> p; // 正确
    X<T*> a; // 隐式生成 X<T> 需要
                // 隐式实例化 X<T*>, 这又需要
}

```

```

class Foo {
public:
    int get_x() const { return m_x; }
    void set_x(int x) { m_x = x; }
private:
    Foo(int x, Foo*& this_ref): m_x(x) {
        this_ref = this;
    }
    int m_x;
    friend const Foo& getFoo();
};

const Foo& getFoo() {
    static const Foo foo(123, instance);
    return foo;
}

void do_evil(int x) {
    instance->set_x(x);
}

int main() {
    const Foo& foo = getFoo();
    do_evil(456);
    std::cout << foo.get_x() << '\n';
}

```

In this code, `getFoo` creates a singleton of type `const Foo` and its member `m_x` is initialized to 123. Then `do_evil` is called and the value of `foo.m_x` is apparently changed to 456. What went wrong?

Despite its name, `do_evil` does nothing particularly evil; all it does is call a setter through a `Foo*`. But that pointer points to a `const Foo` object even though `const_cast` was not used. This pointer was obtained through `Foo`'s constructor. A `const` object does not become `const` until its initialization is complete, so `this` has type `Foo*`, not `const Foo*`, within the constructor.

Therefore, undefined behavior occurs even though there are no obviously dangerous constructs in this program.

Section 104.15: Returning from a [[noreturn]] function

Version ≥ C++11

Example from the Standard, [dcl.attr.noreturn]:

```

[[ noreturn ]] void f() {
    throw "error"; // OK
}
[[ noreturn ]] void q(int i) { // behavior is undefined if called with an argument <= 0
    if (i > 0)
        throw "positive";
}

```

Section 104.16: Infinite template recursion

Example from the Standard, [temp.inst]/17:

```

template<class T> class X {
    X<T*> p; // OK
    X<T*> a; // implicit generation of X<T> requires
                // the implicit instantiation of X<T*> which requires
}

```

```
// 隐式实例化 X<T**>, 依此类推...
```

```
}
```

第104.17节：转换为或从浮点类型时的溢出

如果在以下转换过程中：

- 整数类型转换为浮点类型，
- 浮点类型转换为整数类型，或
- 浮点类型转换为更短的浮点类型，

源值超出目标类型可表示的值范围，则结果为未定义行为。示例：

```
double x = 1e100;  
int y = x; // int 可能无法容纳如此大的数值，因此这是未定义行为
```

第104.18节：修改字符串字面量

版本 < C++11

```
char *str = "hello world";  
str[0] = 'H';
```

"hello world" 是字符串字面量，因此修改它会导致未定义行为。

上述示例中对 str 的初始化在 C++03 中被正式弃用（计划在未来标准版本中移除）。2003 年之前的许多编译器可能会对此发出警告（例如，怀疑的转换）。2003 年之后，编译器通常会警告这是一个弃用的转换。

版本 ≥ C++11

上述示例是非法的，并且在 C++11 及更高版本中会导致编译器诊断。可以构造一个类似的示例，通过显式允许类型转换来表现未定义行为，例如：

```
char *str = const_cast<char*>("hello world");  
str[0] = 'H';
```

第104.19节：以错误类型访问对象

在大多数情况下，将一种类型的对象当作另一种类型访问是非法的（忽略cv限定符）。例如：

```
float x = 42;  
int y = reinterpret_cast<int&>(x);
```

结果是未定义行为。

对此严格别名规则有一些例外：

- 类类型的对象可以被当作其实际类类型的基类类型访问。
- 任何类型都可以被当作char或unsigned char访问，但反之不成立：不能将char数组当作任意类型访问。
- 有符号整数类型可以被当作对应的无符号类型访问，反之亦然。

```
// the implicit instantiation of X<T**> which ...
```

```
}
```

Section 104.17: Overflow during conversion to or from floating point type

If, during the conversion of:

- an integer type to a floating point type,
- a floating point type to an integer type, or
- a floating point type to a shorter floating point type,

the source value is outside the range of values that can be represented in the destination type, the result is undefined behavior. Example:

```
double x = 1e100;  
int y = x; // int probably cannot hold numbers that large, so this is UB
```

Section 104.18: Modifying a string literal

Version < C++11

```
char *str = "hello world";  
str[0] = 'H';
```

"hello world" is a string literal, so modifying it gives undefined behaviour.

The initialisation of str in the above example was formally deprecated (scheduled for removal from a future version of the standard) in C++03. A number of compilers before 2003 might issue a warning about this (e.g. a suspicious conversion). After 2003, compilers typically warn about a deprecated conversion.

Version ≥ C++11

The above example is illegal, and results in a compiler diagnostic, in C++11 and later. A similar example may be constructed to exhibit undefined behaviour by explicitly permitting the type conversion, such as:

```
char *str = const_cast<char*>("hello world");  
str[0] = 'H';
```

Section 104.19: Accessing an object as the wrong type

In most cases, it is illegal to access an object of one type as though it were a different type (disregarding cv-qualifiers). Example:

```
float x = 42;  
int y = reinterpret_cast<int&>(x);
```

The result is undefined behavior.

There are some exceptions to this *strict aliasing* rule:

- An object of class type can be accessed as though it were of a type that is a base class of the actual class type.
- Any type can be accessed as a `char` or `unsigned char`, but the reverse is not true: a `char` array cannot be accessed as though it were an arbitrary type.
- A signed integer type can be accessed as the corresponding unsigned type and *vice versa*.

相关规则是，如果在一个对象上调用非静态成员函数，而该对象实际上并不具有与该函数定义类相同的类型，或其派生类类型，则会发生未定义行为。即使该函数不访问该对象，这条规则依然成立。

```
struct Base {  
};  
struct Derived : Base {  
    void f() {}  
};  
struct Unrelated {};  
Unrelated u;  
Derived& r1 = reinterpret_cast<Derived&>(u); // 正确  
r1.f(); // 未定义行为  
Base b;  
Derived& r2 = reinterpret_cast<Derived&>(b); // 正确  
r2.f(); // 未定义行为
```

A related rule is that if a non-static member function is called on an object that does not actually have the same type as the defining class of the function, or a derived class, then undefined behavior occurs. This is true even if the function does not access the object.

```
struct Base {  
};  
struct Derived : Base {  
    void f() {}  
};  
struct Unrelated {};  
Unrelated u;  
Derived& r1 = reinterpret_cast<Derived&>(u); // ok  
r1.f(); // UB  
Base b;  
Derived& r2 = reinterpret_cast<Derived&>(b); // ok  
r2.f(); // UB
```

第104.20节：指向成员的指针的无效派生到基类转换

当使用 `static_cast` 将 `T D::*` 转换为 `T B::*` 时，被指向的成员必须属于 B 的基类或派生类。否则行为未定义。参见指向成员的指针的派生到基类转换

第104.21节：销毁已被销毁的对象

在此示例中，显式调用了一个对象的析构函数，该对象稍后将被自动销毁。

```
struct S {  
    ~S() { std::cout << "销毁 S"; }  
}  
  
int main() {  
    S s;  
    s.~S();  
} // 未定义行为：s 在此处被第二次销毁
```

当一个 `std::unique_ptr<T>` 指向一个具有自动或静态存储期的 T 时，会发生类似的问题。

```
void f(std::unique_ptr<S> p);  
int main() {  
    S s;  
    std::unique_ptr<S> p(&s);  
    f(std::move(p)); // s 在从 f 返回时被销毁  
} // 未定义行为：s 被销毁
```

另一种导致对象被销毁两次的方式是两个 `shared_ptr` 都管理该对象，但彼此之间不共享所有权。

```
void f(std::shared_ptr<S> p1, std::shared_ptr<S> p2);  
int main() {  
    S* p = new S;  
    // 我想传递同一个对象两次...  
    std::shared_ptr<S> sp1(p);  
    std::shared_ptr<S> sp2(p);  
    f(sp1, sp2);  
} // 未定义行为：sp1 和 sp2 会分别销毁 s
```

Section 104.20: Invalid derived-to-base conversion for pointers to members

When `static_cast` is used to convert `T D::*` to `T B::*`, the member pointed to must belong to a class that is a base class or derived class of B. Otherwise the behavior is undefined. See Derived to base conversion for pointers to members

Section 104.21: Destroying an object that has already been destroyed

In this example, a destructor is explicitly invoked for an object that will later be automatically destroyed.

```
struct S {  
    ~S() { std::cout << "destroying S\n"; }  
};  
int main() {  
    S s;  
    s.~S();  
} // UB: s destroyed a second time here
```

A similar issue occurs when a `std::unique_ptr<T>` is made to point at a T with automatic or static storage duration.

```
void f(std::unique_ptr<S> p);  
int main() {  
    S s;  
    std::unique_ptr<S> p(&s);  
    f(std::move(p)); // s destroyed upon return from f  
} // UB: s destroyed
```

Another way to destroy an object twice is by having two `shared_ptr`s both manage the object without sharing ownership with each other.

```
void f(std::shared_ptr<S> p1, std::shared_ptr<S> p2);  
int main() {  
    S* p = new S;  
    // I want to pass the same object twice...  
    std::shared_ptr<S> sp1(p);  
    std::shared_ptr<S> sp2(p);  
    f(sp1, sp2);  
} // UB: both sp1 and sp2 will destroy s separately
```

```
// 注意：这是正确的：  
// std::shared_ptr<S> sp(p);  
// f(sp, sp);
```

第104.22节：通过成员指针访问不存在的成员

当通过成员指针访问对象的非静态成员时，如果对象实际上不包含该成员指针所指示的成员，则行为未定义。（这样的成员指针可以通过 `static_cast` 获得。）

```
struct Base { int x; };  
struct Derived : Base { int y; };  
int Derived::*pdy = &Derived::y;  
int Base::*pby = static_cast<int Base::*>(pdy);  
  
Base* b1 = new Derived;  
b1->pby = 42; // 好的；将派生对象中的 y 设置为 42  
Base* b2 = new Base;  
b2->pby = 42; // 未定义；Base中没有y成员
```

第104.23节：无效的基类到派生类的静态转换

如果使用 `static_cast` 将指向基类的指针（或引用）转换为指向派生类的指针（或引用），但操作数并未指向（或引用）派生类类型的对象，则行为未定义。

参见基类到派生类的转换。

第104.24节：浮点溢出

如果产生浮点类型的算术运算结果超出结果类型可表示的范围，根据C++标准，行为未定义，但可能由机器所遵循的其他标准定义，如IEEE 754。

```
float x = 1.0;  
for (int i = 0; i < 10000; i++) {  
    x *= 10.0; // 最终可能会溢出；行为未定义  
}
```

第104.25节：在构造函数或析构函数中调用（纯）虚函数成员

标准（10.4）规定：

成员函数可以从抽象类的构造函数（或析构函数）中调用；对于从该构造函数（或析构函数）创建（或销毁）的对象，直接或间接对纯虚函数进行虚函数调用（10.3）的效果是未定义的。

更一般地，一些C++权威人士，例如斯科特·迈耶斯，建议永远不要从构造函数和析构函数中调用虚函数（即使是纯虚函数）。

考虑以下示例，修改自上述链接：

类 transaction

```
// NB: this is correct:  
// std::shared_ptr<S> sp(p);  
// f(sp, sp);
```

Section 104.22: Access to nonexistent member through pointer to member

When accessing a non-static member of an object through a pointer to member, if the object does not actually contain the member denoted by the pointer, the behavior is undefined. (Such a pointer to member can be obtained through `static_cast`.)

```
struct Base { int x; };  
struct Derived : Base { int y; };  
int Derived::*pdy = &Derived::y;  
int Base::*pby = static_cast<int Base::*>(pdy);  
  
Base* b1 = new Derived;  
b1->pby = 42; // ok; sets y in Derived object to 42  
Base* b2 = new Base;  
b2->pby = 42; // undefined; there is no y member in Base
```

Section 104.23: Invalid base-to-derived static cast

If `static_cast` is used to convert a pointer (resp. reference) to base class to a pointer (resp. reference) to derived class, but the operand does not point (resp. refer) to an object of the derived class type, the behavior is undefined. See Base to derived conversion.

Section 104.24: Floating point overflow

If an arithmetic operation that yields a floating point type produces a value that is not in the range of representable values of the result type, the behavior is undefined according to the C++ standard, but may be defined by other standards the machine might conform to, such as IEEE 754.

```
float x = 1.0;  
for (int i = 0; i < 10000; i++) {  
    x *= 10.0; // will probably overflow eventually; undefined behavior  
}
```

Section 104.25: Calling (Pure) Virtual Members From Constructor Or Destructor

The Standard (10.4) states:

Member functions can be called from a constructor (or destructor) of an abstract class; the effect of making a virtual call (10.3) to a pure virtual function directly or indirectly for the object being created (or destroyed) from such a constructor (or destructor) is undefined.

More generally, some C++ authorities, e.g. Scott Meyers, suggest never calling virtual functions (even non-pure ones) from constructors and destructors.

Consider the following example, modified from the above link:

```
class transaction
```

```

{
    公共:
    transaction() { log_it(); }
        virtual void log_it() const = 0;
    };

类 sell_transaction : public transaction
{
    公共:
        virtual void log_it() const { /* 执行某些操作 */ }
};

```

假设我们创建一个 sell_transaction 对象：

```
sell_transaction s;
```

这隐式调用了卖出交易(sell_transaction)的构造函数，该构造函数首先调用交易(transaction)的构造函数。然而，当调用交易(transaction)的构造函数时，对象尚未成为卖出交易(sell_transaction)类型，而仅仅是交易(transaction)类型。

因此，在交易(transaction)::交易(transaction)()中调用的log_it，不会执行看似直观的操作——即调用卖出交易(sell_transaction)::log_it。

- 如果log_it是纯虚函数，如本例所示，则行为是未定义的。
- 如果log_it是非纯虚函数，则会调用交易(transaction)::log_it。

第104.26节：通过不匹配的函数指针类型调用函数

为了通过函数指针调用函数，函数指针的类型必须与函数的类型完全匹配。否则，行为是未定义的。示例：

```

int f();
void (*p)() = reinterpret_cast<void(*)()>(f);
p(); // 未定义

```

```

{
    public:
        transaction() { log_it(); }
        virtual void log_it() const = 0;
};

class sell_transaction : public transaction
{
    public:
        virtual void log_it() const { /* Do something */ }
};

```

Suppose we create a sell_transaction object:

```
sell_transaction s;
```

This implicitly calls the constructor of sell_transaction, which first calls the constructor of transaction. When the constructor of transaction is called though, the object is not yet of the type sell_transaction, but rather only of the type transaction.

Consequently, the call in transaction::transaction() to log_it, won't do what might seem to be the intuitive thing - namely call sell_transaction::log_it.

- If log_it is pure virtual, as in this example, the behaviour is undefined.
- If log_it is non-pure virtual, transaction::log_it will be called.

Section 104.26: Function call through mismatched function pointer type

In order to call a function through a function pointer, the function pointer's type must exactly match the function's type. Otherwise, the behaviour is undefined. Example:

```

int f();
void (*p)() = reinterpret_cast<void(*)()>(f);
p(); // undefined

```

第105章：重载解析

第105.1节：参数传递的实参成本分类

重载解析将传递实参给参数的成本划分为四种不同的类别，称为“序列”。每个序列可能包含零个、一个或多个转换

- 标准转换序列

```
void f(int a); f(42);
```

- 用户定义的转换序列

```
void f(std::string s); f("hello");
```

- 省略号转换序列

```
void f(...); f(42);
```

- 列表初始化序列

```
void f(std::vector<int> v); f({1, 2, 3});
```

一般原则是，标准转换序列是最便宜的，其次是用户定义的转换序列，最后是省略号转换序列。

一种特殊情况是列表初始化序列，它不构成转换（初始化列表不是具有类型的表达式）。其成本通过将其定义为等同于其他三种转换序列之一来确定，具体取决于参数类型和初始化列表的形式。

第105.2节：算术提升和转换

将整数类型转换为相应的提升类型优于将其转换为其他整数类型。

```
void f(int x);
void f(short x);
有符号字符 c = 42;
f(c); // 调用 f(int); 提升为 int 优于转换为 short
short s = 42;
f(s); // 调用 f(short); 精确匹配优于提升为 int
```

将float提升为double优于将其转换为其他浮点类型。

```
void f(double x);
void f(long double x);
f(3.14f); // 调用 f(double); 升级为 double 比转换为 long double 更好
```

除提升外的算术转换彼此既不优于也不劣于对方。

```
void f(float x);
void f(long double x);
f(3.14); // 二义性
```

Chapter 105: Overload resolution

Section 105.1: Categorization of argument to parameter cost

Overload resolution partitions the cost of passing an argument to a parameter into one of four different categories, called "sequences". Each sequence may include zero, one or several conversions

- Standard conversion sequence

```
void f(int a); f(42);
```

- User defined conversion sequence

```
void f(std::string s); f("hello");
```

- Ellipsis conversion sequence

```
void f(...); f(42);
```

- List initialization sequence

```
void f(std::vector<int> v); f({1, 2, 3});
```

The general principle is that Standard conversion sequences are the cheapest, followed by user defined conversion sequences, followed by ellipsis conversion sequences.

A special case is the list initialization sequence, which does not constitute a conversion (an initializer list is not an expression with a type). Its cost is determined by defining it to be equivalent to one of the other three conversion sequences, depending on the parameter type and form of initializer list.

Section 105.2: Arithmetic promotions and conversions

Converting an integer type to the corresponding promoted type is better than converting it to some other integer type.

```
void f(int x);
void f(short x);
signed char c = 42;
f(c); // calls f(int); promotion to int is better than conversion to short
short s = 42;
f(s); // calls f(short); exact match is better than promotion to int
```

Promoting a `float` to `double` is better than converting it to some other floating point type.

```
void f(double x);
void f(long double x);
f(3.14f); // calls f(double); promotion to double is better than conversion to long double
```

Arithmetic conversions other than promotions are neither better nor worse than each other.

```
void f(float x);
void f(long double x);
f(3.14); // ambiguous
```

```
void g(long x);
void g(long double x);
g(42); // 二义性
g(3.14); // 二义性
```

因此，为了确保调用函数 f 时，无论是整型还是任何标准类型的浮点参数，都不会产生二义性，需要总共八个重载，这样对于每种可能的参数类型，要么有一个重载完全匹配，要么会选择带有提升参数类型的唯一重载。

```
void f(int x);
void f(unsigned int x);
void f(long x);
void f(unsigned long x);
void f(long long x);
void f(unsigned long long x);
void f(double x);
void f(long double x);
```

第105.3节：转发引用上的重载

在提供转发引用重载时，必须非常小心，因为它可能匹配得过于完美：

```
结构体 A {
A() = default;           // #1
A(A const&) = default; // #2

template <class T>
A(T&);                  // #3
};
```

这里的意图是 A 是可拷贝的，并且我们有另一个构造函数可能初始化其他成员。然而：

```
A a; // 调用 #1
A b(a); // 调用 #3！
```

构造调用有两个可行的匹配：

```
A(A const&); // #2
A(A&); // #3, T = A&
```

两者都是精确匹配，但 #3 接受对比 #2 更少 cv 限定的对象的引用，因此它具有更好的标准转换序列，是最佳可行函数。

这里的解决方案是始终约束这些构造函数（例如使用 SFINAE）：

```
template <class T,
    class = std::enable_if_t<!std::is_convertible<std::decay_t<T>*, A*>::value>
>
A(T&);
```

这里的类型特征是排除任何 A 或公开且明确继承自 A 的类参与考虑，这会使得前面示例中该构造函数形式不正确（因此从重载集合中移除）。结果，调用了拷贝构造函数——这正是我们想要的。

```
void g(long x);
void g(long double x);
g(42); // ambiguous
g(3.14); // ambiguous
```

Therefore, in order to ensure that there will be no ambiguity when calling a function f with either integral or floating-point arguments of any standard type, a total of eight overloads are needed, so that for each possible argument type, either an overload matches exactly or the unique overload with the promoted argument type will be selected.

```
void f(int x);
void f(unsigned int x);
void f(long x);
void f(unsigned long x);
void f(long long x);
void f(unsigned long long x);
void f(double x);
void f(long double x);
```

Section 105.3: Overloading on Forwarding Reference

You must be very careful when providing a forwarding reference overload as it may match too well:

```
struct A {
A() = default;           // #1
A(A const&) = default; // #2

template <class T>
A(T&);                  // #3
};
```

The intent here was that A is copyable, and that we have this other constructor that might initialize some other member. However:

```
A a; // calls #1
A b(a); // calls #3!
```

There are two viable matches for the construction call:

```
A(A const&); // #2
A(A&); // #3, with T = A&
```

Both are Exact Matches, but #3 takes a reference to a less cv-qualified object than #2 does, so it has the better standard conversion sequence and is the best viable function.

The solution here is to always constrain these constructors (e.g. using SFINAE):

```
template <class T,
    class = std::enable_if_t<!std::is_convertible<std::decay_t<T>*, A*>::value>
>
A(T&);
```

The type trait here is to exclude any A or class publicly and unambiguously derived from A from consideration, which would make this constructor ill-formed in the example described earlier (and hence removed from the overload set). As a result, the copy constructor is invoked - which is what we wanted.

第105.4节：精确匹配

不需要参数类型转换或仅需要在仍被视为精确匹配的类型之间转换的重载，优先于需要其他转换才能调用的重载。

```
void f(int x);
void f(double x);
f(42); // 调用 f(int)
```

当实参绑定到相同类型的引用时，即使引用具有更多的cv限定，匹配也被视为不需要转换。

```
void f(int& x);
void f(double x);
int x = 42;
f(x); // 参数类型是 int；与 int& 精确匹配

void g(const int& x);
void g(int x);
g(x); // 模糊；两个重载都完全匹配
```

为了重载解析的目的，类型“数组 of T”被视为与类型“指针 to T”完全匹配，函数类型 T 被视为与函数指针类型 T* 完全匹配，尽管两者都需要转换。

```
void f(int* p);
void f(void* p);

void g(int* p);
void g(int (&p)[100]);

int a[100];
f(a); // 调用 f(int*); 与数组到指针的转换完全匹配
g(a); // 模糊调用；两个重载都给出完全匹配
```

第105.5节：基于常量性和易变性的重载

如果可能，将指针参数传递给 T* 参数比传递给 const T* 参数更好。

```
struct Base {};
struct Derived : Base {};
void f(Base* pb);
void f(const Base* pb);
void f(const Derived* pd);
void f(bool b);

Base b;
f(&b); // f(Base*) 比 f(const Base*) 更好
Derived d;
f(&d); // f(const Derived*) 比 f(Base*) 更好；
        // 常量性只是“决胜规则”
```

同样地，如果可能的话，将参数传递给 T&类型的参数，比传递给const T&类型的参数更好，即使两者的匹配等级完全相同。

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

Section 105.4: Exact match

An overload without conversions needed for parameter types or only conversions needed between types that are still considered exact matches is preferred over an overload that requires other conversions in order to call.

```
void f(int x);
void f(double x);
f(42); // calls f(int)
```

When an argument binds to a reference to the same type, the match is considered to not require a conversion even if the reference is more cv-qualified.

```
void f(int& x);
void f(double x);
int x = 42;
f(x); // argument type is int; exact match with int&

void g(const int& x);
void g(int x);
g(x); // ambiguous; both overloads give exact match
```

For the purposes of overload resolution, the type "array of T" is considered to match exactly with the type "pointer to T", and the function type T is considered to match exactly with the function pointer type T*, even though both require conversions.

```
void f(int* p);
void f(void* p);

void g(int* p);
void g(int (&p)[100]);

int a[100];
f(a); // calls f(int*); exact match with array-to-pointer conversion
g(a); // ambiguous; both overloads give exact match
```

Section 105.5: Overloading on constness and volatility

Passing a pointer argument to a T* parameter, if possible, is better than passing it to a const T* parameter.

```
struct Base {};
struct Derived : Base {};
void f(Base* pb);
void f(const Base* pb);
void f(const Derived* pd);
void f(bool b);

Base b;
f(&b); // f(Base*) is better than f(const Base*)
Derived d;
f(&d); // f(const Derived*) is better than f(Base*) though;
        // constness is only a "tie-breaker" rule
```

Likewise, passing an argument to a T& parameter, if possible, is better than passing it to a const T& parameter, even if both have exact match rank.

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

```

int x;
f(x); // 两个重载都完全匹配，但 f(int&) 仍然更优
const int y = 42;
f(y); // 只有 f(const int&) 可用

```

该规则同样适用于 `const` 限定的成员函数，这对于允许对非 `const` 对象进行可变访问以及对 `const` 对象进行不可变访问非常重要。

```

class IntVector {
public:
    // ...
    int* data() { return m_data; }
    const int* data() const { return m_data; }
private:
    // ...
    int* m_data;
};

IntVector v1;
int* data1 = v1.data();           // Vector::data() 比 Vector::data() const 更好；
                                // data1 可用于修改向量的数据

const IntVector v2;
const int* data2 = v2.data();     // 只有 Vector::data() const 可用；
                                // data2 不能用于修改向量的数据

```

同样，`volatile` 重载的优先级会低于非 `volatile` 重载。

```

class AtomicInt {
public:
    // ...
    int load();
    int load() volatile;
private:
    // ...
};

AtomicInt a1;
a1.load(); // 优先选择非 volatile 重载；无副作用
volatile AtomicInt a2;
a2.load(); // 只有 volatile 重载可用；有副作用
static_cast<volatile AtomicInt&>(a1).load(); // 强制 a1 使用 volatile 语义

```

第 105.6 节：名称查找和访问检查

重载解析发生在名称查找之后。这意味着如果名称查找失败，重载解析不会选择匹配更好的函数：

```

void f(int x);
struct S {
    void f(double x);
    void g() { f(42); } // 调用的是 S::f，因为全局的 f 在这里不可见，
                        // 尽管它可能是更合适的匹配
};

```

重载解析发生在访问检查之前。即使不可访问的函数匹配度更高，重载解析也可能选择它，而不是可访问的函数。

```

class C {
public:
    static void f(double x);
}

```

```

int x;
f(x); // both overloads match exactly, but f(int&) is still better
const int y = 42;
f(y); // only f(const int&) is viable

```

This rule applies to `const`-qualified member functions as well, where it is important for allowing mutable access to non-`const` objects and immutable access to `const` objects.

```

class IntVector {
public:
    // ...
    int* data() { return m_data; }
    const int* data() const { return m_data; }
private:
    // ...
    int* m_data;
};

IntVector v1;
int* data1 = v1.data();           // Vector::data() is better than Vector::data() const;
                                // data1 can be used to modify the vector's data

const IntVector v2;
const int* data2 = v2.data();     // only Vector::data() const is viable;
                                // data2 can't be used to modify the vector's data

```

In the same way, a `volatile` overload will be less preferred than a non-`volatile` overload.

```

class AtomicInt {
public:
    // ...
    int load();
    int load() volatile;
private:
    // ...
};

AtomicInt a1;
a1.load(); // non-volatile overload preferred; no side effect
volatile AtomicInt a2;
a2.load(); // only volatile overload is viable; side effect
static_cast<volatile AtomicInt&>(a1).load(); // force volatile semantics for a1

```

Section 105.6: Name lookup and access checking

Overload resolution occurs *after* name lookup. This means that a better-matching function will not be selected by overload resolution if it loses name lookup:

```

void f(int x);
struct S {
    void f(double x);
    void g() { f(42); } // calls S::f because global f is not visible here,
                        // even though it would be a better match
};

```

Overload resolution occurs *before* access checking. An inaccessible function might be selected by overload resolution if it is a better match than an accessible function.

```

class C {
public:
    static void f(double x);
}

```

```

private:
    static void f(int x);
};

C::f(42); // 错误！调用了私有的 C::f(int)，尽管公共的 C::f(double) 是可行的。

```

同样，重载解析发生时不会检查结果调用是否符合 [explicit](#):

```

struct X {
    explicit X(int );
    X(char );
};

void foo(X );
foo({4}); // X(int) 更优，但表达式格式错误，因为选中的构造函数是 explicit

```

第 105.7 节：类层次结构内的重载

以下示例将使用此类层次结构：

```

struct A { int m; };
struct B : A {};
struct C : B {};

```

从派生类类型到基类类型的转换优先于用户定义的转换。这适用于按值传递或按引用传递，以及将指向派生类的指针转换为指向基类的指针时。

```

struct Unrelated {
    Unrelated(B b);
};

void f(A a);
void f(Unrelated u);
B b;
f(b); // 调用 f(A)

```

从派生类到基类的指针转换也优于转换为 `void*`。

```

void f(A* p);
void f(void* p);
B b;
f(&b); // 调用 f(A*)

```

如果在同一继承链中存在多个重载，则优先选择最派生的基类重载。这基于与虚函数调用类似的原则：选择“最专门化”的实现。

然而，重载解析总是在编译时发生，且永远不会隐式向下转换。

```

void f(const A& a);
void f(const B& b);
C c;
f(c); // 调用 f(const B&)
B b;
A& r = b;
f(r); // 调用 f(const A&); f(const B&) 重载不可用

```

对于成员指针，由于其相对于类是逆变的，类似的规则适用于相反方向：优先选择最不派生的派生类。

```

private:
    static void f(int x);
};

C::f(42); // Error! Calls private C::f(int) even though public C::f(double) is viable.

```

Similarly, overload resolution happens without checking whether the resulting call is well-formed with regards to [explicit](#):

```

struct X {
    explicit X(int );
    X(char );
};

void foo(X );
foo({4}); // X(int) is better much, but expression is
          // ill-formed because selected constructor is explicit

```

Section 105.7: Overloading within a class hierarchy

The following examples will use this class hierarchy:

```

struct A { int m; };
struct B : A {};
struct C : B {};

```

The conversion from derived class type to base class type is preferred to user-defined conversions. This applies when passing by value or by reference, as well as when converting pointer-to-derived to pointer-to-base.

```

struct Unrelated {
    Unrelated(B b);
};

void f(A a);
void f(Unrelated u);
B b;
f(b); // calls f(A)

```

A pointer conversion from derived class to base class is also better than conversion to `void*`.

```

void f(A* p);
void f(void* p);
B b;
f(&b); // calls f(A*)

```

If there are multiple overloads within the same chain of inheritance, the most derived base class overload is preferred. This is based on a similar principle as virtual dispatch: the “most specialized” implementation is chosen. However, overload resolution always occurs at compile time and will never implicitly down-cast.

```

void f(const A& a);
void f(const B& b);
C c;
f(c); // calls f(const B&)
B b;
A& r = b;
f(r); // calls f(const A&); the f(const B&) overload is not viable

```

For pointers to members, which are contravariant with respect to the class, a similar rule applies in the opposite direction: the least derived derived class is preferred.

```
void f(int B::*p);
void f(int C::*p);
int A::*p = &A::m;
f(p); // 调用 f(int B::*)
```

第105.8节：重载解析的步骤

重载解析的步骤如下：

1. 通过名称查找找到候选函数。非限定调用将执行常规的非限定查找以及基于参数的查找（如果适用）。
2. 将候选函数集合筛选为一组可行函数。对于可行函数，存在一个在调用函数时传入的参数与函数参数之间的隐式转换序列。

```
void f(char);           // (1)
void f(int ) = delete; // (2)
void f();               // (3)
void f(int& );        // (4)

f(4); // 1,2 是可行的（尽管2被删除了！）
// 3 不可行，因为参数列表不匹配
// 4 不可行，因为我们不能将临时对象绑定到
//     非const左值引用
```

3. 选择最佳可行候选函数。一个可行函数F1比另一个可行函数F2更优，当且仅当F1中每个参数的隐式转换序列不比F2中对应的隐式转换序列差，且...：

3.1. 对于某个参数，F1中该参数的隐式转换序列优于F2中该参数的转换序列，或者

```
void f(int ); // (1)
void f(char ); // (2)

f(4); // 调用(1)，转换序列更优
```

3.2. 在用户定义的转换中，F1返回值到目标类型的标准转换序列优于F2返回类型的转换序列，或者

```
结构体 A
{
operator int();
operator double();
} a;

int i = a; // a.operator int() 优于 a.operator double() 的转换
float f = a; // 二义性
```

3.3. 在直接引用绑定中，F1具有某种引用类型而F2没有，或者

```
结构体 A
{
运算符 X&(); // #1
运算符 X&&(); // #2
```

```
void f(int B::*p);
void f(int C::*p);
int A::*p = &A::m;
f(p); // calls f(int B::*)
```

Section 105.8: Steps of Overload Resolution

The steps of overload resolution are:

1. Find candidate functions via name lookup. Unqualified calls will perform both regular unqualified lookup as well as argument-dependent lookup (if applicable).
2. Filter the set of candidate functions to a set of *viable* functions. A viable function for which there exists an implicit conversion sequence between the arguments the function is called with and the parameters the function takes.

```
void f(char);           // (1)
void f(int ) = delete; // (2)
void f();               // (3)
void f(int& );        // (4)

f(4); // 1,2 are viable (even though 2 is deleted!)
// 3 is not viable because the argument lists don't match
// 4 is not viable because we cannot bind a temporary to
//     a non-const lvalue reference
```

3. Pick the best viable candidate. A viable function F1 is a better function than another viable function F2 if the implicit conversion sequence for each argument in F1 is not worse than the corresponding implicit conversion sequence in F2, and...:

3.1. For some argument, the implicit conversion sequence for that argument in F1 is a better conversion sequence than for that argument in F2, or

```
void f(int ); // (1)
void f(char ); // (2)

f(4); // call (1), better conversion sequence
```

3.2. In a user-defined conversion, the standard conversion sequence from the return of F1 to the destination type is a better conversion sequence than that of the return type of F2, or

```
struct A
{
operator int();
operator double();
} a;

int i = a; // a.operator int() is better than a.operator double() and a conversion
float f = a; // ambiguous
```

3.3. In a direct reference binding, F1 has the same kind of reference by F2 is not, or

```
struct A
{
operator X&(); // #1
operator X&&(); // #2
```

```
};  
A a;  
X& lx = a; // 调用 #1  
X&& rx = a; // 调用 #2
```

3.4. F1 不是函数模板特化，但 F2 是，或者

```
template <class T> void f(T); // #1  
void f(int); // #2  
  
f(42); // 调用 #2, 非模板版本
```

3.5. F1 和 F2 都是函数模板特化，但 F1 比 F2 更特化。

```
template <class T> void f(T); // #1  
template <class T> void f(T*); // #2  
  
int* p;  
f(p); // 调用 #2, 更特化版本
```

这里的顺序很重要。更好的转换序列检查发生在模板与非模板检查之前。这导致了一个关于转发引用重载的常见错误：

```
结构体 A {  
A(A const&); // #1  
  
template <class T>  
A(T&&); // #2, 未受约束  
};  
  
A a;  
A b(a); // 调用 #2!  
// #1 不是模板，但 #2 解析为  
// A(A&), 这是比 #1 更少 cv 限定的引用  
// 这使得它成为更好的隐式转换序列
```

如果最终没有单一最佳可行候选，则调用是二义性的：

```
void f(double) {}  
void f(float) {}  
  
f(42); // 错误：二义性
```

```
};  
A a;  
X& lx = a; // calls #1  
X&& rx = a; // calls #2
```

3.4. F1 is not a function template specialization, but F2 is, or

```
template <class T> void f(T); // #1  
void f(int); // #2  
  
f(42); // calls #2, the non-template
```

3.5. F1 and F2 are both function template specializations, but F1 is more specialized than F2.

```
template <class T> void f(T); // #1  
template <class T> void f(T*); // #2  
  
int* p;  
f(p); // calls #2, more specialized
```

The ordering here is significant. The better conversion sequence check happens before the template vs non-template check. This leads to a common error with overloading on forwarding reference:

```
struct A {  
A(A const&); // #1  
  
template <class T>  
A(T&&); // #2, not constrained  
};  
  
A a;  
A b(a); // calls #2!  
// #1 is not a template but #2 resolves to  
// A(A&), which is a less cv-qualified reference than #1  
// which makes it a better implicit conversion sequence
```

If there's no single best viable candidate at the end, the call is ambiguous:

```
void f(double) {}  
void f(float) {}  
  
f(42); // error: ambiguous
```

第106章：移动语义

第106.1节：移动语义

移动语义是在C++中将一个对象移动到另一个对象的一种方式。为此，我们清空旧对象，并将其所有内容放入新对象中。

为此，我们必须理解什么是右值引用。右值引用（ $T\&&$ ，其中 T 是对象类型）与普通引用（ $T\&$ ，现在称为左值引用）没有太大区别。但它们作为两种不同的类型存在，因此，我们可以编写构造函数或函数来接受其中一种类型，这在处理移动语义时是必要的。

我们需要两种不同类型的原因是为了指定两种不同的行为。左值引用构造函数与复制相关，而右值引用构造函数与移动相关。

要移动一个对象，我们将使用`std::move(obj)`。该函数返回对象的右值引用，因此我们可以将该对象的数据“窃取”到一个新的对象中。下面讨论了几种实现方法。

需要注意的是，使用`std::move`仅仅创建了一个右值引用。换句话说，语句

`std::move(obj)`并不会改变`obj`的内容，而`auto obj2 = std::move(obj)`（可能）会改变。

第106.2节：使用`std::move`将复杂度从 $O(n^2)$ 降低到 $O(n)$

C++11引入了对移动对象的核心语言和标准库支持。其思想是，当一个对象 o 是临时对象且需要逻辑上的复制时，可以安全地直接“窃取” o 的资源，比如动态分配的缓冲区，使得 o 逻辑上变为空，但仍可析构和复制。

核心语言支持主要是

- rvalue reference类型构造器 $\&\&$ ，例如，`std::string&&`是对`std::string`的右值引用，表示被引用的对象是一个临时对象，其资源可以直接被“窃取”（即移动）
- 对移动构造函数 $T(T\&&)$ 的特殊支持，该构造函数应高效地从指定的另一个对象移动资源，而不是实际复制资源，
- 对移动赋值运算符`auto operator=(T&&) -> T&`的特殊支持，该运算符也应从源对象移动资源。

标准库支持主要是来自`<utility>`头文件的`std::move`函数模板。该函数生成指定对象的右值引用，表示该对象可以被移动，就像它是一个临时对象一样。

对于容器，实际复制的复杂度通常是 $O(n)$ ，其中 n 是容器中元素的数量，而移动的复杂度是 $O(1)$ ，即常数时间。对于一个逻辑上复制该容器 n 次的算法，这可以将复杂度从通常不切实际的 $O(n^2)$ 降低到线性 $O(n)$ 。

在他2013年9月19日发表于《Dr. Dobbs Journal》上的文章“永不改变的容器”中，安德鲁·科尼格（Andrew Koenig）提出了一个有趣的算法低效示例，该示例涉及使用一种变量初始化后不可变的编程风格。在这种风格中，循环通常用递归来表达。对于某些算法，比如生成柯拉兹序列，递归需要逻辑上复制一个容器：

```
// 基于安德鲁·科尼格在其《Dr. Dobbs Journal》文章中的示例
```

Chapter 106: Move Semantics

Section 106.1: Move semantics

Move semantics are a way of moving one object to another in C++. For this, we empty the old object and place everything it had in the new object.

For this, we must understand what an rvalue reference ($T\&&$ where T is the object type) is not much different than a normal reference ($T\&$, now called lvalue references). But they act as 2 different types, and so, we can make constructors or functions that take one type or the other, which will be necessary when dealing with move semantics.

The reason why we need two different types is to specify two different behaviors. Lvalue reference constructors are related to copying, while rvalue reference constructors are related to moving.

To move an object, we will use `std::move(obj)`. This function returns an rvalue reference to the object, so that we can steal the data from that object into a new one. There are several ways of doing this which are discussed below.

Important to note is that the use of `std::move` creates just an rvalue reference. In other words the statement `std::move(obj)` does not change the content of `obj`, while `auto obj2 = std::move(obj)` (possibly) does.

Section 106.2: Using `std::move` to reduce complexity from $O(n^2)$ to $O(n)$

C++11 introduced core language and standard library support for **moving** an object. The idea is that when an object o is a temporary and one wants a logical copy, then its safe to just pilfer o 's resources, such as a dynamically allocated buffer, leaving o logically empty but still destructible and copyable.

The core language support is mainly

- the **rvalue reference** type builder $\&\&$, e.g., `std::string&&` is an rvalue reference to a `std::string`, indicating that that referred to object is a temporary whose resources can just be pilfered (i.e. moved)
- special support for a **move constructor** $T(T\&&)$, which is supposed to efficiently move resources from the specified other object, instead of actually copying those resources, and
- special support for a **move assignment operator** `auto operator=(T&&) -> T&`, which also is supposed to move from the source.

The standard library support is mainly the `std::move` function template from the `<utility>` header. This function produces an rvalue reference to the specified object, indicating that it can be moved from, just as if it were a temporary.

For a container actual copying is typically of $O(n)$ complexity, where n is the number of items in the container, while moving is $O(1)$, constant time. And for an algorithm that logically copies that container n times, this can reduce the complexity from the usually impractical $O(n^2)$ to just linear $O(n)$.

In his article “Containers That Never Change” in Dr. Dobbs Journal in September 19 2013, Andrew Koenig presented an interesting example of algorithmic inefficiency when using a style of programming where variables are immutable after initialization. With this style loops are generally expressed using recursion. And for some algorithms such as generating a Collatz sequence, the recursion requires logically copying a container:

```
// Based on an example by Andrew Koenig in his Dr. Dobbs Journal article
```

```

// “永不改变的容器” 2013年9月19日，链接地址
// <url: http://www.drdobbs.com/cpp/containers-that-never-change/240161543>

// 此处包含，例如 <vector>

命名空间 my {
    模板< 类 Item >
    使用 Vector_ = /* 例如 std::vector<Item> */;

    自动 concat( Vector_<int> 常量引用& v, int 常量 x )
        -> Vector_<int>
    {
        自动 result{ v };
        result.push_back( x );
        返回 result;
    }

    auto collatz_aux( int const n, Vector_<int> const& result )
        -> Vector_<int>
    {
        if( n == 1 )
        {
            返回 result;
        }
        auto const new_result = concat( result, n );
        if( n % 2 == 0 )
        {
            返回 collatz_aux( n/2, new_result );
        }
        否则
        {
            返回 collatz_aux( 3*n + 1, new_result );
        }
    }

    auto collatz( int const n )
        -> Vector_<int>
    {
        assert( n != 0 );
        返回 collatz_aux( n, Vector_<int>() );
    }
} // namespace my

#include <iostream>
using namespace std;
auto main() -> int
{
    for( int const x : my::collatz( 42 ) )
    {
        cout << x << ' ';
    }
    cout << "\n";
}

```

输出：

42 21 64 32 16 8 4 2

由于向量的复制，项目复制操作的数量大致为 $O(n^2)$ ，因为这是 $1 + 2 + 3$

+ ... n 的总和。

```

// “Containers That Never Change” September 19, 2013, available at
// <url: http://www.drdobbs.com/cpp/containers-that-never-change/240161543>

// Includes here, e.g. <vector>

namespace my {
    模板< class Item >
    using Vector_ = /* E.g. std::vector<Item> */;

    auto concat( Vector_<int> const& v, int const x )
        -> Vector_<int>
    {
        auto result{ v };
        result.push_back( x );
        返回 result;
    }

    auto collatz_aux( int const n, Vector_<int> const& result )
        -> Vector_<int>
    {
        if( n == 1 )
        {
            返回 result;
        }
        auto const new_result = concat( result, n );
        if( n % 2 == 0 )
        {
            返回 collatz_aux( n/2, new_result );
        }
        else
        {
            返回 collatz_aux( 3*n + 1, new_result );
        }
    }

    auto collatz( int const n )
        -> Vector_<int>
    {
        assert( n != 0 );
        返回 collatz_aux( n, Vector_<int>() );
    }
} // namespace my

#include <iostream>
using namespace std;
auto main() -> int
{
    for( int const x : my::collatz( 42 ) )
    {
        cout << x << ' ';
    }
    cout << '\n';
}

```

Output:

42 21 64 32 16 8 4 2

The number of item copy operations due to copying of the vectors is here roughly $O(n^2)$, since it's the sum $1 + 2 + 3 + \dots + n$.

具体数字上，使用 g++ 和 Visual C++ 编译器，上述调用 `collatz(42)` 产生了一个包含 8 个元素的 Collatz 序列，以及 36 次项目复制操作 ($8*7/2 = 28$ ，加上一些额外操作)，这些操作发生在向量复制构造函数调用中。

所有这些项目复制操作都可以通过简单地移动不再需要其值的向量来消除。

为此，需要去除向量类型参数的 `const` 和引用，改为按值传递向量。

函数返回已经自动优化。对于传递向量且函数后续不再使用的调用，只需对这些缓冲区应用 `std::move` 来移动它们，而不是实际复制：

```
using std::move;

auto concat( Vector<int> v, int const x )
    -> Vector<int>
{
    v.push_back( x );
    // 警告：在返回语句中移动局部对象会阻止拷贝省略 [-Wpessimizing-move]
    // 参见 https://stackoverflow.com/documentation/c%2b%2b/2489/copy-elision
    // return move( v );
    return v;
}

auto collatz_aux( int const n, Vector<int> result )
    -> Vector<int>
{
    if( n == 1 )
    {
        return result;
    }
    auto new_result = concat( move( result ), n );
    struct result; // 确保之后绝对不使用 `result`。
    if( n % 2 == 0 )
    {
        return collatz_aux( n/2, move( new_result ) );
    }
    否则
    {
        return collatz_aux( 3*n + 1, move( new_result ) );
    }
}

auto collatz( int const n )
    -> Vector<int>
{
    assert( n != 0 );
    return collatz_aux( n, Vector<int>() );
}
```

这里，使用 g++ 和 Visual C++ 编译器时，由于调用 vector 拷贝构造函数导致的元素拷贝操作次数恰好为 0。

该算法在生成的 Collatz 序列长度上仍然是 $O(n)$ 复杂度，但这是一个相当显著的改进： $O(n^2) \rightarrow O(n)$ 。

借助一些语言支持，或许可以使用移动语义，同时表达并强制变量在初始化和最终移动之间的不可变性，之后对该变量的任何使用都应视为错误。可惜的是，截至 C++14，C++ 并不支持这一点。对于无循环代码，可以通过将相关名称重新声明为不完整的 `struct`（如上文的 `struct result;`）来强制执行移动后不再使用，但这很难看且不易被其他程序员理解；此外，诊断信息可能会非常误导。

In concrete numbers, with g++ and Visual C++ compilers the above invocation of `collatz(42)` resulted in a Collatz sequence of 8 items and 36 item copy operations ($8*7/2 = 28$, plus some) in vector copy constructor calls.

All of these item copy operations can be removed by simply moving vectors whose values are not needed anymore. To do this it's necessary to remove `const` and reference for the vector type arguments, passing the vectors *by value*. The function returns are already automatically optimized. For the calls where vectors are passed, and not used again further on in the function, just apply `std::move` to *move* those buffers rather than actually copying them:

```
using std::move;

auto concat( Vector<int> v, int const x )
    -> Vector<int>
{
    v.push_back( x );
    // warning: moving a local object in a return statement prevents copy elision [-Wpessimizing-move]
    // See https://stackoverflow.com/documentation/c%2b%2b/2489/copy-elision
    // return move( v );
    return v;
}

auto collatz_aux( int const n, Vector<int> result )
    -> Vector<int>
{
    if( n == 1 )
    {
        return result;
    }
    auto new_result = concat( move( result ), n );
    struct result; // Make absolutely sure no use of `result` after this.
    if( n % 2 == 0 )
    {
        return collatz_aux( n/2, move( new_result ) );
    }
    else
    {
        return collatz_aux( 3*n + 1, move( new_result ) );
    }
}

auto collatz( int const n )
    -> Vector<int>
{
    assert( n != 0 );
    return collatz_aux( n, Vector<int>() );
}
```

Here, with g++ and Visual C++ compilers, the number of item copy operations due to vector copy constructor invocations, was exactly 0.

The algorithm is necessarily still $O(n)$ in the length of the Collatz sequence produced, but this is a quite dramatic improvement: $O(n^2) \rightarrow O(n)$.

With some language support one could perhaps use moving and still express and enforce the immutability of a variable *between its initialization and final move*, after which any use of that variable should be an error. Alas, as of C++14 C++ does not support that. For loop-free code the no use after move can be enforced via a re-declaration of the relevant name as an incomplete `struct`, as with `struct result;` above, but this is ugly and not likely to be understood by other programmers; also the diagnostics can be quite misleading.

总结来说，C++ 语言和库对移动语义的支持允许算法复杂度大幅度改进，但由于支持不完整，代价是放弃了 `const` 所能提供的代码正确性保证和代码清晰度。

为完整起见，用于测量因拷贝构造函数调用导致的元素拷贝操作次数的带有检测功能的 `vector` 类：

```
template< class Item >
class Copy_tracking_vector
{
private:
    static auto n_copy_ops()
        -> int&
    {
        static int value;
        return value;
    }

vector<Item> items_;

public:
    static auto n() -> int { return n_copy_ops(); }

    void push_back( Item const& o ) { items_.push_back( o ); }
    auto begin() const { return items_.begin(); }
    auto end() const { return items_.end(); }

Copy_tracking_vector(){}

Copy_tracking_vector( Copy_tracking_vector const& other )
    : items_( other.items_ )
{ n_copy_ops() += items_.size(); }

Copy_tracking_vector( Copy_tracking_vector&& other )
    : items_( move( other.items_ ) )
{ }
};
```

第106.3节：移动构造函数

假设我们有以下代码片段。

```
class A {
public:
    int a;
    int b;

A(const A &other) {
    this->a = other.a;
    this->b = other.b;
}
};
```

要创建一个拷贝构造函数，也就是说，编写一个函数来复制一个对象并创建一个新的对象，我们通常会选择上面显示的语法，我们会有一个接受另一个类型为A的对象引用的构造函数，并在方法内部手动复制该对象。

或者，我们也可以写成`A(const A &) = default;`，这会自动复制所有成员，

Summing up, the C++ language and library support for moving allows drastic improvements in algorithm complexity, but due to the support's incompleteness, at the cost of forsaking the code correctness guarantees and code clarity that `const` can provide.

For completeness, the instrumented vector class used to measure the number of item copy operations due to copy constructor invocations:

```
template< class Item >
class Copy_tracking_vector
{
private:
    static auto n_copy_ops()
        -> int&
    {
        static int value;
        return value;
    }

vector<Item> items_;

public:
    static auto n() -> int { return n_copy_ops(); }

    void push_back( Item const& o ) { items_.push_back( o ); }
    auto begin() const { return items_.begin(); }
    auto end() const { return items_.end(); }

Copy_tracking_vector(){}

Copy_tracking_vector( Copy_tracking_vector const& other )
    : items_( other.items_ )
{ n_copy_ops() += items_.size(); }

Copy_tracking_vector( Copy_tracking_vector&& other )
    : items_( move( other.items_ ) )
{ }
};
```

Section 106.3: Move constructor

Say we have this code snippet.

```
class A {
public:
    int a;
    int b;

A(const A &other) {
    this->a = other.a;
    this->b = other.b;
}
};
```

To create a copy constructor, that is, to make a function that copies an object and creates a new one, we normally would choose the syntax shown above, we would have a constructor for A that takes a reference to another object of type A, and we would copy the object manually inside the method.

Alternatively, we could have written `A(const A &) = default;` which automatically copies over all members,

利用其拷贝构造函数。

然而，要创建一个移动构造函数，我们将接受一个右值引用而不是左值引用，就像这里一样。

```
class Wallet {  
public:  
    int nrOfDollars;  
  
    Wallet() = default; //默认构造函数  
  
    Wallet(Wallet &&other) {  
        this->美元数量 = other.美元数量;  
        other.美元数量 = 0;  
    }  
};
```

请注意，我们将旧值设置为零。默认的移动构造函数 (`Wallet(Wallet&&) = default;`) 会复制 `nrOfDollars` 的值，因为它是一个POD类型。

由于移动语义旨在允许从原始实例“窃取”状态，因此需要考虑在这种窃取之后原始实例应呈现的状态。在本例中，如果我们不将值改为零，就会导致美元数量被重复计算。

```
Wallet a;  
a.nrOfDollars = 1;  
Wallet b (std::move(a)); //调用 B(B&& other);  
std::cout << a.nrOfDollars << std::endl; //0  
std::cout << b.nrOfDollars << std::endl; //1
```

因此，我们已经从旧对象移动构造了一个新对象。

虽然上述是一个简单的例子，但它展示了移动构造函数的目的。在更复杂的情况下，比如涉及资源管理时，它会变得更加有用。

```
// 管理涉及指定类型的操作。  
// 拥有一个堆上的辅助对象和一个内存中的辅助对象（可能在栈上）。  
// 两个辅助对象都是可默认构造、可拷贝构造和可移动构造的。  
template<typename T,  
         template<typename> typename HeapHelper,  
         template<typename> typename StackHelper>  
class OperationsManager {  
    using MyType = OperationsManager<T, HeapHelper, StackHelper>;  
  
    HeapHelper<T>* h_helper;  
    StackHelper<T> s_helper;  
    // ...  
  
    公共:  
        // Default constructor & Rule of Five.  
    OperationsManager() : h_helper(new HeapHelper<T>) {}  
    OperationsManager(const MyType& other)  
        : h_helper(new HeapHelper<T>(*other.h_helper)), s_helper(other.s_helper) {}  
    MyType& operator=(MyType copy) {  
        swap(*this, copy);  
        return *this;  
    }  
    ~OperationsManager() {  
        if (h_helper) { delete h_helper; }  
    }
```

making use of its copy constructor.

To create a move constructor, however, we will be taking an rvalue reference instead of an lvalue reference, like here.

```
class Wallet {  
public:  
    int nrOfDollars;  
  
    Wallet() = default; //default ctor  
  
    Wallet(Wallet &&other) {  
        this->nrOfDollars = other.nrOfDollars;  
        other.nrOfDollars = 0;  
    }  
};
```

Please notice that we set the old values to zero. The default move constructor (`Wallet(Wallet&&) = default;`) copies the value of `nrOfDollars`, as it is a POD.

As move semantics are designed to allow 'stealing' state from the original instance, it is important to consider how the original instance should look like after this stealing. In this case, if we would not change the value to zero we would have doubled the amount of dollars into play.

```
Wallet a;  
a.nrOfDollars = 1;  
Wallet b (std::move(a)); //calling B(B&& other);  
std::cout << a.nrOfDollars << std::endl; //0  
std::cout << b.nrOfDollars << std::endl; //1
```

Thus we have move constructed an object from an old one.

While the above is a simple example, it shows what the move constructor is intended to do. It becomes more useful in more complex cases, such as when resource management is involved.

```
// Manages operations involving a specified type.  
// Owns a helper on the heap, and one in its memory (presumably on the stack).  
// Both helpers are DefaultConstructible, CopyConstructible, and MoveConstructible.  
template<typename T,  
         template<typename> typename HeapHelper,  
         template<typename> typename StackHelper>  
class OperationsManager {  
    using MyType = OperationsManager<T, HeapHelper, StackHelper>;  
  
    HeapHelper<T>* h_helper;  
    StackHelper<T> s_helper;  
    // ...  
  
    公共:  
        // Default constructor & Rule of Five.  
    OperationsManager() : h_helper(new HeapHelper<T>) {}  
    OperationsManager(const MyType& other)  
        : h_helper(new HeapHelper<T>(*other.h_helper)), s_helper(other.s_helper) {}  
    MyType& operator=(MyType copy) {  
        swap(*this, copy);  
        return *this;  
    }  
    ~OperationsManager() {  
        if (h_helper) { delete h_helper; }  
    }
```

```

}

// 移动构造函数 (不使用 swap())。
// 接收另一个对象的 HeapHelper<T>*。
// 通过强制使用 StackHelper<T> 的移动构造函数，接收另一个对象的 StackHelper<T>。
// 将另一个对象的 HeapHelper<T>* 替换为 nullptr，防止另一个对象销毁时删除我们新创建的 helper。

OperationsManager(MyType&& other) noexcept
    : h_helper(other.h_helper),
s_helper(std::move(other.s_helper)) {
    other.h_helper = nullptr;
}

// 移动构造函数 (使用 swap())。
// 将我们的成员置于希望另一个对象处于的状态，然后与另一个对象交换成员。

// OperationsManager(MyType&& other) noexcept : h_helper(nullptr) {
//     swap(*this, other);
// }

// 复制/移动辅助函数。
friend void swap(MyType& left, MyType& right) noexcept {
    std::swap(left.h_helper, right.h_helper);
    std::swap(left.s_helper, right.s_helper);
}
};

}

// Move constructor (without swap()).
// Takes other's HeapHelper<T>*.
// Takes other's StackHelper<T>, by forcing the use of StackHelper<T>'s move constructor.
// Replaces other's HeapHelper<T>* with nullptr, to keep other from deleting our shiny
// new helper when it's destroyed.
OperationsManager(MyType&& other) noexcept
    : h_helper(other.h_helper),
s_helper(std::move(other.s_helper)) {
    other.h_helper = nullptr;
}

// Move constructor (with swap()).
// Places our members in the condition we want other's to be in, then switches members
// with other.
// OperationsManager(MyType&& other) noexcept : h_helper(nullptr) {
//     swap(*this, other);
// }

// Copy/move helper.
friend void swap(MyType& left, MyType& right) noexcept {
    std::swap(left.h_helper, right.h_helper);
    std::swap(left.s_helper, right.s_helper);
}
};

```

第106.4节：重新使用已移动的对象

你可以重新使用已移动的对象：

```

void consumingFunction(std::vector<int> vec) {
    // 一些操作
}

int main() {
    // 使用1, 2, 3, 4初始化vec
    std::vector<int> vec{1, 2, 3, 4};

    // 通过移动发送向量
    consumingFunction(std::move(vec));

    // 此时vec对象处于不确定状态。
    // 由于对象未被销毁，我们可以为其赋予新内容。
    // 在这种情况下，我们将为向量赋予一个空值，
    // 使其实际上变为空
    vec = {};

    // 由于向量已经获得了确定的值，我们可以正常使用它。
    vec.push_back(42);

    // 再次通过移动发送向量。
    consumingFunction(std::move(vec));
}

```

第106.5节：移动赋值

类似于我们可以通过左值引用给对象赋值并复制它，我们也可以将值从一个对象移动到另一个对象，而不构造新的对象。我们称之为移动赋值。我们将值从

```

}

// Move constructor (without swap()).
// Takes other's HeapHelper<T>*.
// Takes other's StackHelper<T>, by forcing the use of StackHelper<T>'s move constructor.
// Replaces other's HeapHelper<T>* with nullptr, to keep other from deleting our shiny
// new helper when it's destroyed.
OperationsManager(MyType&& other) noexcept
    : h_helper(other.h_helper),
s_helper(std::move(other.s_helper)) {
    other.h_helper = nullptr;
}

// Move constructor (with swap()).
// Places our members in the condition we want other's to be in, then switches members
// with other.
// OperationsManager(MyType&& other) noexcept : h_helper(nullptr) {
//     swap(*this, other);
// }

// Copy/move helper.
friend void swap(MyType& left, MyType& right) noexcept {
    std::swap(left.h_helper, right.h_helper);
    std::swap(left.s_helper, right.s_helper);
}
};


```

Section 106.4: Re-use a moved object

You can re-use a moved object:

```

void consumingFunction(std::vector<int> vec) {
    // Some operations
}

int main() {
    // initialize vec with 1, 2, 3, 4
    std::vector<int> vec{1, 2, 3, 4};

    // Send the vector by move
    consumingFunction(std::move(vec));

    // Here the vec object is in an indeterminate state.
    // Since the object is not destroyed, we can assign it a new content.
    // We will, in this case, assign an empty value to the vector,
    // making it effectively empty
    vec = {};

    // Since the vector has gained a determinate value, we can use it normally.
    vec.push_back(42);

    // Send the vector by move again.
    consumingFunction(std::move(vec));
}

```

Section 106.5: Move assignment

Similarly to how we can assign a value to an object with an lvalue reference, copying it, we can also move the values from an object to another without constructing a new one. We call this move assignment. We move the values from

一个对象移动到另一个已存在的对象。

为此，我们需要重载operator=，不是让它接受左值引用（如复制赋值中），而是让它接受右值引用。

```
class A {  
    int a;  
A& operator=(A&& other) {  
    this->a = other.a;  
    other.a = 0;  
    return *this;  
}  
};
```

这是定义移动赋值的典型语法。我们重载了operator =，使其可以接受一个右值引用，并将其赋值给另一个对象。

```
A a;  
a.a = 1;  
A b;  
b = std::move(a); //调用 A& operator=(A&& other)  
std::cout << a.a << std::endl; //0  
std::cout << b.a << std::endl; //1
```

因此，我们可以将一个对象移动赋值给另一个对象。

第106.6节：在容器上使用移动语义

你可以移动一个容器，而不是复制它：

```
void print(const std::vector<int>& vec) {  
    for (auto&& val : vec) {  
        std::cout << val << ", ";  
    }  
    std::cout << std::endl;  
}  
  
int main() {  
    // 使用1, 2, 3, 4初始化vec1, vec2为空向量  
    std::vector<int> vec1{1, 2, 3, 4};  
    std::vector<int> vec2;  
  
    // 以下代码将打印1, 2, 3, 4  
    print(vec1);  
  
    // 以下代码将打印一个换行符  
    print(vec2);  
  
    // 向量vec2通过移动赋值进行赋值。  
    // 这将“窃取”vec1的值而不进行复制。  
    vec2 = std::move(vec1);  
  
    // 此时vec1对象处于不确定状态，但仍然有效。  
    // 对象vec1未被销毁，  
    // 但无法保证其内容。  
  
    // 以下代码将打印1, 2, 3, 4  
    print(vec2);
```

one object to another existing object.

For this, we will have to overload operator =, not so that it takes an lvalue reference, like in copy assignment, but so that it takes an rvalue reference.

```
class A {  
    int a;  
A& operator=(A&& other) {  
    this->a = other.a;  
    other.a = 0;  
    return *this;  
}  
};
```

This is the typical syntax to define move assignment. We overload operator = so that we can feed it an rvalue reference and it can assign it to another object.

```
A a;  
a.a = 1;  
A b;  
b = std::move(a); //calling A& operator=(A&& other)  
std::cout << a.a << std::endl; //0  
std::cout << b.a << std::endl; //1
```

Thus, we can move assign an object to another one.

Section 106.6: Using move semantics on containers

You can move a container instead of copying it:

```
void print(const std::vector<int>& vec) {  
    for (auto&& val : vec) {  
        std::cout << val << ", ";  
    }  
    std::cout << std::endl;  
}  
  
int main() {  
    // initialize vec1 with 1, 2, 3, 4 and vec2 as an empty vector  
    std::vector<int> vec1{1, 2, 3, 4};  
    std::vector<int> vec2;  
  
    // The following line will print 1, 2, 3, 4  
    print(vec1);  
  
    // The following line will print a new line  
    print(vec2);  
  
    // The vector vec2 is assigned with move assingment.  
    // This will "steal" the value of vec1 without copying it.  
    vec2 = std::move(vec1);  
  
    // Here the vec1 object is in an indeterminate state, but still valid.  
    // The object vec1 is not destroyed,  
    // but there's is no guarantees about what it contains.  
  
    // The following line will print 1, 2, 3, 4  
    print(vec2);
```


第107章：Pimpl惯用法

第107.1节：基本的Pimpl惯用法

版本 ≥ C++11

在头文件中：

```
// widget.h

#include <memory> // std::unique_ptr
#include <experimental/propagate_const>

class Widget
{
    公共:
    Widget();
    ~Widget();
    void DoSomething();

    私有:
        // pImpl惯用法的名称来源于典型的变量名
        // 即pImpl：
        struct Impl; // 前置声明      std::experimental::propagate_const<std::unique_ptr<Impl>> pImpl; // 指向实际实现的指针
};


```

在实现文件中：

```
// widget.cpp

#include "widget.h"
#include "reallycomplextype.h" // 不需要在 widget.h 中包含此头文件

struct Widget::Impl
{
    // 这里是 Widget 需要的属性
    ReallyComplexType rct;
};

Widget::Widget() :
pImpl(std::make_unique<Impl>())
{}

Widget::~Widget() = default;

void Widget::DoSomething()
{
    // 在这里使用 pImpl 执行操作
}
```

pImpl包含了Widget的状态（或其中一部分/大部分）。与其在头文件中暴露Widget的状态描述，不如只在实现中暴露。

pImpl代表“指向实现的指针”。Widget的“真实”实现就在pImpl中。

危险：注意，为了使其与unique_ptr一起工作，~Widget()必须在文件中Impl完全可见的位置实现。

Chapter 107: Pimpl Idiom

Section 107.1: Basic Pimpl idiom

Version ≥ C++11

In the header file:

```
// widget.h

#include <memory> // std::unique_ptr
#include <experimental/propagate_const>

class Widget
{
    公共:
    Widget();
    ~Widget();
    void DoSomething();

    私有:
        // the pImpl idiom is named after the typical variable name used
        // ie, pImpl:
        struct Impl; // forward declaration
        std::experimental::propagate_const<std::unique_ptr<Impl>> pImpl; // ptr to actual
        implementation
};


```

In the implementation file:

```
// widget.cpp

#include "widget.h"
#include "reallycomplextype.h" // no need to include this header inside widget.h

struct Widget::Impl
{
    // the attributes needed from Widget go here
    ReallyComplexType rct;
};

Widget::Widget() :
pImpl(std::make_unique<Impl>())
{}

Widget::~Widget() = default;

void Widget::DoSomething()
{
    // do the stuff here with pImpl
}
```

The pImpl contains the Widget state (or some/most of it). Instead of the Widget description of state being exposed in the header file, it can be only exposed within the implementation.

pImpl stands for "pointer to implementation". The "real" implementation of Widget is in the pImpl.

Danger: Note that for this to work with unique_ptr, ~Widget() must be implemented at a point in a file where the

Impl完全可见的位置实现。你可以在那里`=default`它，但如果在Impl未定义的地方`=default`，程序可能很容易
变得不合法，且不需要诊断。

Impl is fully visible. You can `=default` it there, but if `=default` where Impl is undefined, the program may easily
become ill-formed, no diagnostic required.

第108章：auto

第108.1节：基本的auto示例

关键字auto提供变量类型的自动推导。

当处理冗长的类型名称时，它尤其方便：

```
std::map< std::string, std::shared_ptr< Widget > > table;
// C++98
std::map< std::string, std::shared_ptr< Widget > >::iterator i = table.find( "42" );
// C++11/14/17
auto j = table.find( "42" );
```

使用基于范围的 for 循环：

```
vector<int> v = {0, 1, 2, 3, 4, 5};
for(auto n: v)
    std::cout << n << ' ';
```

使用 lambda 表达式：

```
auto f = [](){ std::cout << "lambda"; };
```

为了避免重复写类型：

```
auto w = std::make_shared< Widget >();
```

为了避免意外且不必要的拷贝：

```
auto myMap = std::map<int, float>();
myMap.emplace(1, 3.14);

std::pair<int, float> const& firstPair2 = *myMap.begin(); // 拷贝！
auto const& firstPair = *myMap.begin(); // 无拷贝！
```

拷贝的原因是返回的类型实际上是std::pair<const int, float>！

第108.2节：泛型lambda (C++14)

版本 ≥ C++14

C++14允许在lambda参数中使用auto

```
auto print = [](const auto& arg) { std::cout << arg << std::endl; };

print(42);
print("hello world");
```

该lambda大致等同于

```
struct lambda {
    template <typename T>
    auto operator ()(const T& arg) const {
```

Chapter 108: auto

Section 108.1: Basic auto sample

The keyword `auto` provides the auto-deduction of type of a variable.

It is especially convenient when dealing with long type names:

```
std::map< std::string, std::shared_ptr< Widget > > table;
// C++98
std::map< std::string, std::shared_ptr< Widget > >::iterator i = table.find( "42" );
// C++11/14/17
auto j = table.find( "42" );
```

with range-based for loops:

```
vector<int> v = {0, 1, 2, 3, 4, 5};
for(auto n: v)
    std::cout << n << ' ';
```

with lambdas:

```
auto f = [](){ std::cout << "lambda\n"; };
f();
```

to avoid the repetition of the type:

```
auto w = std::make_shared< Widget >();
```

to avoid surprising and unnecessary copies:

```
auto myMap = std::map<int, float>();
myMap.emplace(1, 3.14);

std::pair<int, float> const& firstPair2 = *myMap.begin(); // copy!
auto const& firstPair = *myMap.begin(); // no copy!
```

The reason for the copy is that the returned type is actually `std::pair<const int, float>`!

Section 108.2: Generic lambda (C++14)

Version ≥ C++14

C++14 allows to use `auto` in lambda argument

```
auto print = [](const auto& arg) { std::cout << arg << std::endl; };

print(42);
print("hello world");
```

That lambda is mostly equivalent to

```
struct lambda {
    template <typename T>
    auto operator ()(const T& arg) const {
```

```
    std::cout << arg << std::endl;
}
};
```

然后

```
lambda print;

print(42);
print("hello world");
```

第108.3节：auto和代理对象

有时auto的行为可能与程序员预期的不完全一致。它会对表达式进行类型推断，即使类型推断并非正确的做法。

举个例子，当代码中使用代理对象时：

```
std::vector<bool> flags{true, true, false};
auto flag = flags[0];
flags.push_back(true);
```

这里flag的类型不是bool，而是std::vector<bool>::reference，因为对于bool特化的模板vector，operator []返回的是带有转换操作符operator bool的代理对象。

当flags.push_back(true)修改容器时，这个伪引用可能会变成悬空引用，指向一个已不存在的元素。

这也使得下面的情况成为可能：

```
void foo(bool b);

std::vector<bool> getFlags();

auto flag = getFlags()[5];
foo(flag);
```

该vector会被立即丢弃，因此flag是对已被丢弃元素的伪引用。调用foo会导致未定义行为。

在这种情况下，你可以使用auto声明变量，并通过强制转换为你希望推断的类型来初始化它：

```
auto flag = static_cast<bool>(getFlags()[5]);
```

但在那种情况下，直接用bool替换auto更为合理。

代理对象可能引发问题的另一个情况是表达式模板。在这种情况下，为了效率，模板有时并不设计为在当前完整表达式之外继续存在，下一次使用代理对象会导致未定义行为。

第108.4节：auto与表达式模板

当表达式模板涉及时，auto也可能引发问题：

```
auto mult(int c) {
    return c * std::valarray<int>{1};
```

```
    std::cout << arg << std::endl;
}
};
```

and then

```
lambda print;

print(42);
print("hello world");
```

Section 108.3: auto and proxy objects

Sometimes `auto` may behave not quite as was expected by a programmer. It type deduces the expression, even when type deduction is not the right thing to do.

As an example, when proxy objects are used in the code:

```
std::vector<bool> flags{true, true, false};
auto flag = flags[0];
flags.push_back(true);
```

Here `flag` would be not `bool`, but `std::vector<bool>::reference`, since for `bool` specialization of template vector the operator `[]` returns a proxy object with conversion operator `operator bool` defined.

When `flags.push_back(true)` modifies the container, this pseudo-reference could end up dangling, referring to an element that no longer exists.

It also makes the next situation possible:

```
void foo(bool b);

std::vector<bool> getFlags();

auto flag = getFlags()[5];
foo(flag);
```

The vector is discarded immediately, so `flag` is a pseudo-reference to an element that has been discarded. The call to `foo` invokes undefined behavior.

In cases like this you can declare a variable with `auto` and initialize it by casting to the type you want to be deduced:

```
auto flag = static_cast<bool>(getFlags()[5]);
```

but at that point, simply replacing `auto` with `bool` makes more sense.

Another case where proxy objects can cause problems is expression templates. In that case, the templates are sometimes not designed to last beyond the current full-expression for efficiency sake, and using the proxy object on the next causes undefined behavior.

Section 108.4: auto and Expression Templates

`auto` can also cause problems where expression templates come into play:

```
auto mult(int c) {
    return c * std::valarray<int>{1};
```

```
}
```

```
auto v = mult(3);
std::cout << v[0]; // 某个值，可能是，但几乎肯定不是，3。
```

原因是 operator* 在 valarray 上返回一个代理对象，该对象引用 valarray 以实现惰性计算。通过使用 auto，你创建了一个悬空引用。若不是这样，mult 会返回一个 std::valarray<int>，那么代码肯定会打印3。

第108.5节：auto、const和引用

auto 关键字本身表示一个值类型，类似于 int 或 char。它可以通过 const 关键字和 & 符号修饰，分别表示常量类型或引用类型。这些修饰符可以组合使用。

在这个例子中，s 是一个值类型（其类型将被推断为 std::string），因此 for 循环的每次迭代都会将向量中的字符串复制到 s 中。

```
std::vector<std::string> strings = { "stuff", "things", "misc" };
for(auto s : strings) {
    std::cout << s << std::endl;
}
```

如果循环体修改了 s（例如调用 s.append(" and stuff")），只有这个副本会被修改，strings 中的原始成员不会被改变。

另一方面，如果 s 用 auto& 声明，它将是一个引用类型（推断为 std::string&），因此在循环的每次迭代中，它将被赋值为向量中字符串的引用：

```
for(auto& s : strings) {
    std::cout << s << std::endl;
}
```

在这个循环体内，对 s 的修改将直接影响它所引用的 strings 中的元素。

最后，如果 s 被声明为 const auto&，则它将是一个常量引用类型，意味着在循环的每次迭代中，它将被赋值为向量中字符串的 const 引用：

```
for(const auto& s : strings) {
    std::cout << s << std::endl;
}
```

在这个循环体内， s 不能被修改（即不能调用非const方法）。

在使用带范围的 for 循环时，如果循环体不会修改被遍历的结构，通常建议使用 const auto&，因为这样可以避免不必要的拷贝。

第108.6节：尾置返回类型

auto 用于尾置返回类型的语法中：

```
auto main() -> int {}
```

这等价于

```
}
```

```
auto v = mult(3);
std::cout << v[0]; // some value that could be, but almost certainly is not, 3.
```

The reason is that operator* on valarray gives you a proxy object that refers to the valarray as a means of lazy evaluation. By using auto, you're creating a dangling reference. Instead of mult had returned a std::valarray<int>, then the code would definitely print 3.

Section 108.5: auto, const, and references

The auto keyword by itself represents a value type, similar to int or char. It can be modified with the const keyword and the & symbol to represent a const type or a reference type, respectively. These modifiers can be combined.

In this example, s is a value type (its type will be inferred as std::string), so each iteration of the for loop copies a string from the vector into s.

```
std::vector<std::string> strings = { "stuff", "things", "misc" };
for(auto s : strings) {
    std::cout << s << std::endl;
}
```

If the body of the loop modifies s (such as by calling s.append(" and stuff")), only this copy will be modified, not the original member of strings.

On the other hand, if s is declared with auto& it will be a reference type (inferred to be std::string&), so on each iteration of the loop it will be assigned a reference to a string in the vector:

```
for(auto& s : strings) {
    std::cout << s << std::endl;
}
```

In the body of this loop, modifications to s will directly affect the element of strings that it references.

Finally, if s is declared const auto&, it will be a const reference type, meaning that on each iteration of the loop it will be assigned a const reference to a string in the vector:

```
for(const auto& s : strings) {
    std::cout << s << std::endl;
}
```

Within the body of this loop, s cannot be modified (i.e. no non-const methods can be called on it).

When using auto with range-based for loops, it is generally good practice to use const auto& if the loop body will not modify the structure being looped over, since this avoids unnecessary copies.

Section 108.6: Trailing return type

auto is used in the syntax for trailing return type:

```
auto main() -> int {}
```

which is equivalent to

```
int main() {}
```

主要与 decltype 结合使用，以便使用参数替代 std::declval<T>：

```
template <typename T1, typename T2>
auto Add(const T1& lhs, const T2& rhs) -> decltype(lhs + rhs) { return lhs + rhs; }
```

```
int main() {}
```

Mostly useful combined with decltype to use parameters instead of std::declval<T>:

```
template <typename T1, typename T2>
auto Add(const T1& lhs, const T2& rhs) -> decltype(lhs + rhs) { return lhs + rhs; }
```

第109章：拷贝省略

第109.1节：拷贝省略的目的

标准中有些地方为了初始化对象会进行拷贝或移动。拷贝省略（有时称为返回值优化）是一种优化，在某些特定情况下，编译器被允许避免拷贝或移动，尽管标准规定必须发生。

考虑以下函数：

```
std::string get_string()
{
    return std::string("我是一个字符串。");
}
```

根据标准的严格措辞，该函数将初始化一个临时的std::string，然后将其复制/移动到返回值对象中，随后销毁该临时对象。标准非常明确地说明了代码的这种解释方式。

复制省略是一条规则，允许C++编译器忽略临时对象的创建及其后续的复制/销毁。也就是说，编译器可以直接用初始化表达式来初始化函数的返回值，而不必先创建临时对象。这显然提高了性能。

然而，这对用户有两个可见的影响：

1. 类型必须具有本应被调用的复制/移动构造函数。即使编译器省略了复制/移动，该类型仍然必须能够被复制/移动。
2. 在可能发生省略的情况下，复制/移动构造函数的副作用不保证会发生。

考虑以下情况：

```
版本 ≥ C++11
struct my_type
{
    my_type() = default;
    my_type(const my_type &) {std::cout << "复制中";}  my_type(my_type &&) {std::cout << "移动中";}
};

my_type func()
{
    return my_type();
}
```

调用func会做什么？嗯，它永远不会打印“Copying”，因为临时对象是右值，且my_type是可移动类型。那么它会打印“Moving”吗？

如果没有拷贝省略规则，这将要求始终打印“Moving”。但由于拷贝省略规则的存在，移动构造函数可能会被调用，也可能不会被调用；这取决于具体实现。

因此，在可能发生省略的上下文中，你不能依赖拷贝/移动构造函数的调用。

因为省略是一种优化，编译器可能并不支持所有情况下的省略。无论编译器是否对特定情况进行省略，类型仍然必须支持被省略的操作。所以如果拷贝

Chapter 109: Copy Elision

Section 109.1: Purpose of copy elision

There are places in the standard where an object is copied or moved in order to initialize an object. Copy elision (sometimes called return value optimization) is an optimization whereby, under certain specific circumstances, a compiler is permitted to avoid the copy or move even though the standard says that it must happen.

Consider the following function:

```
std::string get_string()
{
    return std::string("I am a string.");
}
```

According to the strict wording of the standard, this function will initialize a temporary std::string, then copy/move that into the return value object, then destroy the temporary. The standard is very clear that this is how the code is interpreted.

Copy elision is a rule that permits a C++ compiler to *ignore* the creation of the temporary and its subsequent copy/destruction. That is, the compiler can take the initializing expression for the temporary and initialize the function's return value from it directly. This obviously saves performance.

However, it does have two visible effects on the user:

1. The type must have the copy/move constructor that would have been called. Even if the compiler elides the copy/move, the type must still be able to have been copied/moved.
2. Side-effects of copy/move constructors are not guaranteed in circumstances where elision can happen.

Consider the following:

```
Version ≥ C++11
struct my_type
{
    my_type() = default;
    my_type(const my_type &) {std::cout << "Copying\n";}
    my_type(my_type &&) {std::cout << "Moving\n";}
};

my_type func()
{
    return my_type();
}
```

What will calling func do? Well, it will never print "Copying", since the temporary is an rvalue and my_type is a moveable type. So will it print "Moving"?

Without the copy elision rule, this would be required to always print "Moving". But because the copy elision rule exists, the move constructor may or may not be called; it is implementation-dependent.

And therefore, you cannot depend on the calling of copy/move constructors in contexts where elision is possible.

Because elision is an optimization, your compiler may not support elision in all cases. And regardless of whether the compiler elides a particular case or not, the type must still support the operation being elided. So if a copy

构造被省略，类型仍然必须有拷贝构造函数，尽管它不会被调用。

第109.2节：保证拷贝省略

版本 ≥ C++17

通常，省略是一种优化。虽然几乎所有编译器都支持最简单情况下的拷贝省略，但省略仍然给用户带来了特定的负担。也就是说，被省略拷贝/移动的类型必须仍然拥有被省略的拷贝/移动操作。

例如：

```
std::mutex a_mutex;
std::lock_guard<std::mutex> get_lock()
{
    return std::lock_guard<std::mutex>(a_mutex);
}
```

这在某些情况下可能有用，比如 `a_mutex` 是某个系统私有的互斥锁，但外部用户可能想对其进行作用域锁定。

这也是不合法的，因为 `std::lock_guard` 不能被复制或移动。尽管几乎所有的C++编译器都会省略复制/移动操作，但标准仍然要求该类型必须具备该操作。

直到C++17。

C++17通过有效地重新定义某些表达式的含义，规定必须进行省略，从而不进行任何复制/移动。考虑上述代码。

根据C++17之前的规定，该代码表示创建一个临时对象，然后使用该临时对象复制/移动到返回值，但临时复制可以被省略。根据C++17的规定，根本不会创建临时对象。

在C++17中，任何prvalue表达式在用于初始化与表达式类型相同的对象时，不会生成临时对象。该表达式直接初始化该对象。如果返回的prvalue与返回值类型相同，则该类型不需要复制/移动构造函数。因此，根据C++17规则，上述代码可以正常工作。

C++17的规定适用于prvalue类型与被初始化类型匹配的情况。因此，给定上述 `get_lock`，这也不需要复制/移动：

```
std::lock_guard the_lock = get_lock();
```

由于 `get_lock` 的结果是一个prvalue表达式，用于初始化与其类型相同的对象，因此不会发生复制或移动。该表达式从不创建临时对象；它直接用于初始化 `the_lock`。不存在省略，因为没有复制/移动需要被省略。

“保证复制省略”一词因此有些误导，但这是该特性在C++17标准化提案中的名称。它并不保证省略；它消除了复制/移动，重新定义了C++，使得根本不存在需要被省略的复制/移动。

此功能仅适用于涉及 prvalue 表达式的情况。因此，它使用通常的省略规则：

```
std::mutex a_mutex;
std::lock_guard<std::mutex> get_lock()
{
    std::lock_guard<std::mutex> my_lock(a_mutex);
```

construction is elided, the type must still have a copy constructor, even though it will not be called.

Section 109.2: Guaranteed copy elision

Version ≥ C++17

Normally, elision is an optimization. While virtually every compiler support copy elision in the simplest of cases, having elision still places a particular burden on users. Namely, the type who's copy/move is being elided *must* still have the copy/move operation that was elided.

For example:

```
std::mutex a_mutex;
std::lock_guard<std::mutex> get_lock()
{
    return std::lock_guard<std::mutex>(a_mutex);
}
```

This might be useful in cases where `a_mutex` is a mutex that is privately held by some system, yet an external user might want to have a scoped lock to it.

This is also not legal, because `std::lock_guard` cannot be copied or moved. Even though virtually every C++ compiler will elide the copy/move, the standard still *requires* the type to have that operation available.

Until C++17.

C++17 mandates elision by effectively redefining the very meaning of certain expressions so that no copy/moving takes place. Consider the above code.

Under pre-C++17 wording, that code says to create a temporary and then use the temporary to copy/move into the return value, but the temporary copy can be elided. Under C++17 wording, that does not create a temporary at all.

In C++17, any prvalue expression, when used to initialize an object of the same type as the expression, does not generate a temporary. The expression directly initializes that object. If you return a prvalue of the same type as the return value, then the type need not have a copy/move constructor. And therefore, under C++17 rules, the above code can work.

The C++17 wording works in cases where the prvalue's type matches the type being initialized. So given `get_lock` above, this will also not require a copy/move:

```
std::lock_guard the_lock = get_lock();
```

Since the result of `get_lock` is a prvalue expression being used to initialize an object of the same type, no copying or moving will happen. That expression never creates a temporary; it is used to directly initialize `the_lock`. There is no elision because there is no copy/move to be elided.

The term "guaranteed copy elision" is therefore something of a misnomer, but *that is the name of the feature as it is proposed for C++17 standardization*. It does not guarantee elision at all; it *eliminates* the copy/move altogether, redefining C++ so that there never was a copy/move to be elided.

This feature only works in cases involving a prvalue expression. As such, this uses the usual elision rules:

```
std::mutex a_mutex;
std::lock_guard<std::mutex> get_lock()
{
    std::lock_guard<std::mutex> my_lock(a_mutex);
```

```
//执行操作  
return my_lock;  
}
```

虽然这是拷贝省略的有效情况，但C++17规则并不消除此情况下的拷贝/移动。因此，类型仍然必须有拷贝/移动构造函数来初始化返回值。由于lock_guard没有，这仍然是编译错误。实现可以拒绝在传递或返回可平凡拷贝类型对象时省略拷贝。这是为了允许在寄存器中移动此类对象，一些ABI可能在其调用约定中要求这样做。

```
struct trivially_copyable {  
    int a;  
};  
  
void foo (trivially_copyable a) {}  
  
foo(trivially_copyable{}); //不强制拷贝省略
```

第109.3节：参数省略

当你将一个实参传递给函数，且该实参是函数参数类型的纯右值表达式，且该类型不是引用时，则可以省略该纯右值的构造。

```
void func(std::string str) { ... }  
  
func(std::string("foo"));
```

这表示创建一个临时的string，然后将其移动到函数参数str中。拷贝省略允许该表达式直接在str中创建对象，而不是先创建临时对象再移动。

这是一个对声明为explicit的构造函数非常有用的优化。例如，我们本可以将上述写成func("foo")，但这是因为string有一个隐式构造函数，可以将constchar*转换为string。如果该构造函数是explicit的，我们将被迫使用临时变量来调用该explicit构造函数。复制省略使我们避免了不必要的复制/移动。

第109.4节：返回值省略

如果你从函数返回一个prvalue表达式，且该prvalue表达式的类型与函数的返回类型相同，那么可以省略从prvalue临时对象的复制：

```
std::string func()  
{  
    return std::string("foo");  
}
```

几乎所有编译器在这种情况下都会省略临时对象的构造。

第109.5节：命名返回值省略

如果你从函数返回一个lvalue表达式，且该lvalue：

- 表示该函数的一个自动变量，该变量将在return后被销毁，且该自动变量不是函数参数
- 并且该变量的类型与函数的返回类型相同

```
//Do stuff  
return my_lock;  
}
```

While this is a valid case for copy elision, C++17 rules do not *eliminate* the copy/move in this case. As such, the type must still have a copy/move constructor to use to initialize the return value. And since lock_guard does not, this is still a compile error. Implementations are allowed to refuse to elide copies when passing or returning an object of trivially-copyable type. This is to allow moving such objects around in registers, which some ABIs might mandate in their calling conventions.

```
struct trivially_copyable {  
    int a;  
};  
  
void foo (trivially_copyable a) {}  
  
foo(trivially_copyable{}); //copy elision not mandated
```

Section 109.3: Parameter elision

When you pass an argument to a function, and the argument is a prvalue expression of the function's parameter type, and this type is not a reference, then the prvalue's construction can be elided.

```
void func(std::string str) { ... }  
  
func(std::string("foo"));
```

This says to create a temporary string, then move it into the function parameter str. Copy elision permits this expression to directly create the object in str, rather than using a temporary+move.

This is a useful optimization for cases where a constructor is declared `explicit`. For example, we could have written the above as `func("foo")`, but only because `string` has an implicit constructor that converts from a `const char*` to a `string`. If that constructor was `explicit`, we would be forced to use a temporary to call the `explicit` constructor. Copy elision saves us from having to do a needless copy/move.

Section 109.4: Return value elision

If you return a prvalue expression from a function, and the prvalue expression has the same type as the function's return type, then the copy from the prvalue temporary can be elided:

```
std::string func()  
{  
    return std::string("foo");  
}
```

Pretty much all compilers will elide the temporary construction in this case.

Section 109.5: Named return value elision

If you return an lvalue expression from a function, and this lvalue:

- represents an automatic variable local to that function, which will be destroyed after the `return`
- the automatic variable is not a function parameter
- and the type of the variable is the same type as the function's return type

如果以上情况都成立，那么可以省略从左值的复制/移动：

```
std::string func()
{
    std::string str("foo");
    //执行操作
    return str;
}
```

更复杂的情况也有资格进行省略，但情况越复杂，编译器实际省略的可能性越小：

```
std::string func()
{
    std::string ret("foo");
    if(some_condition)
    {
        return "bar";
    }
    return ret;
}
```

编译器仍然可以省略ret，但这样做的可能性降低了。

如前所述，值参数不允许进行省略。

```
std::string func(std::string str)
{
    str.assign("foo");
    //执行操作
    return str; //无法省略
}
```

第109.6节：拷贝初始化省略

如果使用prvalue表达式来拷贝初始化一个变量，且该变量与prvalue表达式类型相同，则可以省略拷贝。

```
std::string str = std::string("foo");
```

拷贝初始化实际上将其转换为std::string str("foo"); (存在细微差别)。

这对返回值同样适用：

```
std::string func()
{
    return std::string("foo");
}

std::string str = func();
```

如果不进行拷贝省略，这将导致调用两次std::string的移动构造函数。拷贝省略允许调用移动构造函数1次或0次，大多数编译器会选择后者。

If all of these are the case, then the copy/move from the lvalue can be elided:

```
std::string func()
{
    std::string str("foo");
    //Do stuff
    return str;
}
```

More complex cases are eligible for elision, but the more complex the case, the less likely the compiler will be to actually elide it:

```
std::string func()
{
    std::string ret("foo");
    if(some_condition)
    {
        return "bar";
    }
    return ret;
}
```

The compiler could still elide ret, but the chances of them doing so go down.

As noted earlier, elision is not permitted for value *parameters*.

```
std::string func(std::string str)
{
    str.assign("foo");
    //Do stuff
    return str; //No elision possible
}
```

Section 109.6: Copy initialization elision

If you use a prvalue expression to copy initialize a variable, and that variable has the same type as the prvalue expression, then the copying can be elided.

```
std::string str = std::string("foo");
```

Copy initialization effectively transforms this into std::string str("foo"); (there are minor differences).

This also works with return values:

```
std::string func()
{
    return std::string("foo");
}

std::string str = func();
```

Without copy elision, this would provoke 2 calls to std::string's move constructor. Copy elision permits this to call the move constructor 1 or zero times, and most compilers will opt for the latter.

第110章：折叠表达式

第110.1节：一元折叠

一元折叠用于对参数包进行特定运算符的折叠。有两种一元折叠：

- 一元左折叠(... op pack)，展开如下：

```
((Pack1 op Pack2) op ...) op PackN
```

- 一元右折叠(pack op ...)，展开如下：

```
Pack1 op (... (Pack(N-1) op PackN))
```

下面是一个示例

```
template<typename... Ts>
int sum(Ts... args)
{
    return (... + args); //一元左折叠
    //return (args + ...); //一元右折叠

    // 如果运算符是结合律的，则两者等价。
    // 对于 +, ((1+2)+3) (左折叠) == (1+(2+3)) (右折叠)
    // 对于 -, ((1-2)-3) (左折叠) != (1-(2-3)) (右折叠)
}

int result = sum(1, 2, 3); // 6
```

第110.2节：二元折叠

二元折叠基本上是一元折叠，只是多了一个参数。

二元折叠有两种类型：

- 二元左折叠 - (value op ... op pack) - 展开如下：

```
((Value op Pack1) op Pack2) op ... op PackN
```

- 二元右折叠(pack op ... op value) - 展开如下：

```
Pack1 op (... op (Pack(N-1) op (PackN op Value)))
```

下面是一个示例：

```
template<typename... Ts>
int removeFrom(int num, Ts... args)
{
    return (num - ... - args); //二元左折叠// 注意由于减法运
    算符不满足结合律，不能使用二元右折叠
}
```

Chapter 110: Fold Expressions

Section 110.1: Unary Folds

Unary folds are used to *fold* parameter packs over a specific operator. There are 2 kinds of unary folds:

- Unary **Left** Fold (... op pack) which expands as follows:

```
((Pack1 op Pack2) op ...) op PackN
```

- Unary **Right** Fold (pack op ...) which expands as follows:

```
Pack1 op (... (Pack(N-1) op PackN))
```

Here is an example

```
template<typename... Ts>
int sum(Ts... args)
{
    return (... + args); //Unary left fold
    //return (args + ...); //Unary right fold

    // The two are equivalent if the operator is associative.
    // For +, ((1+2)+3) (left fold) == (1+(2+3)) (right fold)
    // For -, ((1-2)-3) (left fold) != (1-(2-3)) (right fold)
}

int result = sum(1, 2, 3); // 6
```

Section 110.2: Binary Folds

Binary folds are basically unary folds, with an extra argument.

There are 2 kinds of binary folds:

- Binary **Left** Fold - (value op ... op pack) - Expands as follows:

```
((Value op Pack1) op Pack2) op ... op PackN
```

- Binary **Right** Fold (pack op ... op value) - Expands as follows:

```
Pack1 op (... op (Pack(N-1) op (PackN op Value)))
```

Here is an example:

```
template<typename... Ts>
int removeFrom(int num, Ts... args)
{
    return (num - ... - args); //Binary left fold
    // Note that a binary right fold cannot be used
    // due to the lack of associativity of operator-
}
```

```
int result = removeFrom(1000, 5, 10, 15); //'result' is 1000 - 5 - 10 - 15 = 970
```

第110.3节：逗号折叠

在参数包的每个元素上执行特定函数是一个常见操作。在 C++11 中，我们能做的最好方式是：

```
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    using expander = int[];
    (void)expander{0,
        (void(os << args), 0)...};
}
```

但使用折叠表达式，上述代码可以简化为：

```
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    (void(os << args), ...);
}
```

无需晦涩的模板样板代码。

```
int result = removeFrom(1000, 5, 10, 15); //'result' is 1000 - 5 - 10 - 15 = 970
```

Section 110.3: Folding over a comma

It is a common operation to need to perform a particular function over each element in a parameter pack. With C++11, the best we can do is:

```
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    using expander = int[];
    (void)expander{0,
        (void(os << args), 0)...};
}
```

But with a fold expression, the above simplifies nicely to:

```
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    (void(os << args), ...);
}
```

No cryptic boilerplate required.

第111章：联合体

第111.1节：未定义行为

```
union U {  
    int a;  
    short b;  
    float c;  
};  
U u;  
  
u.a = 10;  
if (u.b == 10) {  
    // 这是未定义行为，因为“a”是最后一个被写入的成员。// 许多编译器会允许这样做并  
    // 可能发出警告，// 但结果将是“如预期”；这是编译器的扩展，// 不能保证在所有编译器中  
    // 都适用（即这不是符合标准/可移植的代码）。  
}
```

第111.2节：联合体的基本特性

联合体是一种特殊的结构体，其中所有成员共享重叠的内存空间。

```
union U {  
    int a;  
    short b;  
    float c;  
};  
U u;  
  
// a 和 b 的地址将相等  
(void*)&u.a == (void*)&u.b;  
(void*)&u.a == (void*)&u.c;  
  
// 给联合体的任一成员赋值会改变所有成员共享的内存  
u.c = 4.f;  
u.a = 5;  
u.c != 4.f;
```

第111.3节：典型用法

联合体对于最小化专用数据的内存使用非常有用，例如在实现混合数据类型时。

```
结构体 AnyType {  
    枚举 {  
        IS_INT,  
        IS_FLOAT  
    } 类型;  
  
    联合体 数据 {  
        整数 作为整数;  
        浮点数 作为浮点数;  
    } 值;  
  
    AnyType(整数 i) : 类型(IS_INT) { 值.作为整数 = i; }  
    AnyType(浮点数 f) : 类型(IS_FLOAT) { 值.作为浮点数 = f; }  
};
```

Chapter 111: Unions

Section 111.1: Undefined Behavior

```
union U {  
    int a;  
    short b;  
    float c;  
};  
U u;  
  
u.a = 10;  
if (u.b == 10) {  
    // this is undefined behavior since 'a' was the last member to be  
    // written to. A lot of compilers will allow this and might issue a  
    // warning, but the result will be "as expected"; this is a compiler  
    // extension and cannot be guaranteed across compilers (i.e. this is  
    // not compliant/portable code).  
}
```

Section 111.2: Basic Union Features

Unions are a specialized struct within which all members occupy overlapping memory.

```
union U {  
    int a;  
    short b;  
    float c;  
};  
U u;  
  
// Address of a and b will be equal  
(void*)&u.a == (void*)&u.b;  
(void*)&u.a == (void*)&u.c;  
  
// Assigning to any union member changes the shared memory of all members  
u.c = 4.f;  
u.a = 5;  
u.c != 4.f;
```

Section 111.3: Typical Use

Unions are useful for minimizing memory usage for exclusive data, such as when implementing mixed data types.

```
struct AnyType {  
    enum {  
        IS_INT,  
        IS_FLOAT  
    } type;  
  
    union Data {  
        int as_int;  
        float as_float;  
    } value;  
  
    AnyType(int i) : type(IS_INT) { value.as_int = i; }  
    AnyType(float f) : type(IS_FLOAT) { value.as_float = f; }  
};
```

```
整数 获取整数() 常量 {
    如果(类型 == IS_INT)
        返回 值.作为整数;
    否则
        返回 (整数)值.作为浮点数;
}
```

```
浮点数 获取浮点数() 常量 {
    如果(类型 == IS_FLOAT)
        返回 值.作为浮点数;
    否则
        返回 (浮点数)值.作为整数;
}
};
```

```
int get_int() const {
    if(type == IS_INT)
        return value.as_int;
    else
        return (int)value.as_float;
}

float get_float() const {
    if(type == IS_FLOAT)
        return value.as_float;
    else
        return (float)value.as_int;
}
};
```

第112章：设计模式在C++中的实现

在本页中，您可以找到设计模式在C++中的实现示例。有关这些模式的详细信息，您可以查看设计模式文档。

第112.1节：适配器模式

将一个类的接口转换成客户期望的另一个接口。适配器（或包装器）使得由于接口不兼容而无法一起工作的类能够协同工作。适配器模式的动机是，如果我们能修改接口，就可以重用现有的软件。

1. 适配器模式依赖于对象组合。
2. 客户端调用适配器对象上的操作。
3. 适配器调用被适配者以执行操作。
4. 在STL中，stack是从vector适配而来：当stack执行push()时，底层的vector执行vector::push_back()。

示例：

```
#include <iostream>

// 期望的接口 (目标)
class Rectangle
{
public:
    virtual void draw() = 0;
};

// 旧版组件 (适配者)
class 旧版矩形
{
public:
    旧版矩形(int x1, int y1, int x2, int y2) {
        x1_ = x1;
        y1_ = y1;
        x2_ = x2;
        y2_ = y2;
        std::cout << "旧版矩形(x1,y1,x2,y2)";}

    void oldDraw() {
        std::cout << "旧版矩形: oldDraw(). ";
    }

private:
    int x1_;
    int y1_;
    int x2_;
    int y2_;
};

// 适配器包装器
class RectangleAdapter: public Rectangle, private LegacyRectangle
{
public:
    RectangleAdapter(int x, int y, int w, int h):
        LegacyRectangle(x, y, x + w, y + h) {
```

Chapter 112: Design pattern implementation in C++

On this page, you can find examples of how design patterns are implemented in C++. For the details on these patterns, you can check out the design patterns documentation.

Section 112.1: Adapter Pattern

Convert the interface of a class into another interface clients expect. Adapter (or Wrapper) lets classes work together that couldn't otherwise because of incompatible interfaces. Adapter pattern's motivation is that we can reuse existing software if we can modify the interface.

1. Adapter pattern relies on object composition.
2. Client calls operation on Adapter object.
3. Adapter calls Adaptee to carry out the operation.
4. In STL, stack adapted from vector: When stack executes push(), underlying vector does vector::push_back().

Example:

```
#include <iostream>

// Desired interface (Target)
class Rectangle
{
public:
    virtual void draw() = 0;
};

// Legacy component (Adaptee)
class LegacyRectangle
{
public:
    LegacyRectangle(int x1, int y1, int x2, int y2) {
        x1_ = x1;
        y1_ = y1;
        x2_ = x2;
        y2_ = y2;
        std::cout << "LegacyRectangle(x1,y1,x2,y2)\n";
    }

    void oldDraw() {
        std::cout << "LegacyRectangle: oldDraw(). \n";
    }

private:
    int x1_;
    int y1_;
    int x2_;
    int y2_;
};

// Adapter wrapper
class RectangleAdapter: public Rectangle, private LegacyRectangle
{
public:
    RectangleAdapter(int x, int y, int w, int h):
        LegacyRectangle(x, y, x + w, y + h) {
```

```

std::cout << "RectangleAdapter(x,y,x+w,x+h)";}

void draw() {
    std::cout << "RectangleAdapter: draw()." : oldDraw();
}

int main()
{
    int x = 20, y = 50, w = 300, h = 200;
    Rectangle *r = new RectangleAdapter(x,y,w,h);
    r->draw();
}

//输出：
//LegacyRectangle(x1,y1,x2,y2)
//RectangleAdapter(x,y,x+w,x+h)

```

代码总结：

1. 客户端认为他正在与一个矩形交谈
2. 目标是矩形类。这是客户端调用方法的对象。

```

矩形*r=new矩形适配器(x,y,w,h);
r->绘制();

```

3. 注意适配器类使用了多重继承。

```

类矩形适配器: 公有继承矩形, 私有继承LegacyRectangle {
    ...
}

```

4. 适配器矩形适配器使得LegacyRectangle能够响应请求（在矩形上调用绘制()）通过继承两个类。
5. LegacyRectangle类没有与矩形相同的方法（绘制()），但是适配器（RectangleAdapter）可以接收Rectangle的方法调用，并转而调用LegacyRectangle上的方法oldDraw()。

```

class RectangleAdapter: public Rectangle, private LegacyRectangle {
public:
    RectangleAdapter(int x, int y, int w, int h):
        LegacyRectangle(x, y, x + w, y + h) {
            std::cout << "RectangleAdapter(x,y,x+w,y+h)";}

    void draw() {
        std::cout << "RectangleAdapter: draw()." : oldDraw();
    }
}

```

适配器设计模式将一个类的接口转换成兼容但不同的接口。因此，这与代理模式类似，因为它是单组件的包装器。但适配器类和原始类的接口可能不同。

```

std::cout << "RectangleAdapter(x,y,x+w,x+h)\n";
}

void draw() {
    std::cout << "RectangleAdapter: draw().\n";
    oldDraw();
}
};

int main()
{
    int x = 20, y = 50, w = 300, h = 200;
    Rectangle *r = new RectangleAdapter(x,y,w,h);
    r->draw();
}

//Output:
//LegacyRectangle(x1,y1,x2,y2)
//RectangleAdapter(x,y,x+w,x+h)

```

Summary of the code:

1. The client thinks he is talking to a Rectangle
2. The target is the Rectangle class. This is what the client invokes method on.

```

Rectangle *r = new RectangleAdapter(x,y,w,h);
r->draw();

```

3. Note that the adapter class uses multiple inheritance.

```

class RectangleAdapter: public Rectangle, private LegacyRectangle {
    ...
}

```

4. The Adapter RectangleAdapter lets the LegacyRectangle responds to request (draw()) on a Rectangle by inheriting BOTH classes.
5. The LegacyRectangle class does not have the same methods (draw()) as Rectangle, but the Adapter (RectangleAdapter) can take the Rectangle method calls and turn around and invoke method on the LegacyRectangle, oldDraw().

```

class RectangleAdapter: public Rectangle, private LegacyRectangle {
public:
    RectangleAdapter(int x, int y, int w, int h):
        LegacyRectangle(x, y, x + w, y + h) {
            std::cout << "RectangleAdapter(x,y,x+w,y+h)\n";
        }

    void draw() {
        std::cout << "RectangleAdapter: draw().\n";
        oldDraw();
    }
}

```

Adapter design pattern translates the interface for one class into a compatible but different interface. So, this is similar to the **proxy** pattern in that it's a single-component wrapper. But the interface for the adapter class and the

原始类的接口可能不同。

正如上面示例所示，这种适配器模式对于为现有API暴露不同接口以使其能与其他代码协作非常有用。此外，通过使用适配器模式，我们可以将异构接口转换为提供一致的API。

桥接模式的结构类似于对象适配器，但桥接的意图不同：它旨在将接口与其实现分离，以便它们可以轻松且独立地变化。适配器的目的是改变现有对象的接口。

第112.2节：观察者模式

观察者模式的意图是定义对象之间的一对多依赖关系，以便当一个对象的状态发生变化时，所有依赖它的对象都会被自动通知并更新。

主题和观察者定义了一对多的关系。观察者依赖于主题，当主题的状态发生变化时，观察者会收到通知。根据通知，观察者也可能会被更新为新的值。

以下是《设计模式》（Design Patterns）一书中由伽玛（Gamma）提供的示例。

```
#include <iostream>
#include <vector>

class Subject;

class Observer
{
public:
    virtual ~Observer() = default;
    virtual void Update(Subject&) = 0;
};

class Subject
{
public:
    virtual ~Subject() = default;
    void Attach(Observer& o) { observers.push_back(&o); }
    void Detach(Observer& o)
    {
        observers.erase(std::remove(observers.begin(), observers.end(), &o));
    }
    void Notify()
    {
        for (auto* o : observers) {
            o->Update(*this);
        }
    }
private:
    std::vector<Observer*> 观察者;
};

class ClockTimer : public Subject
{
public:
    void SetTime(int hour, int minute, int second)
    {
        this->hour = hour;
    }
}
```

original class may be different.

As we've seen in the example above, this **adapter** pattern is useful to expose a different interface for an existing API to allow it to work with other code. Also, by using adapter pattern, we can take heterogeneous interfaces, and transform them to provide consistent API.

Bridge pattern has a structure similar to an object adapter, but Bridge has a different intent: It is meant to **separate** an interface from its implementation so that they can be varied easily and independently. An **adapter** is meant to **change the interface** of an **existing** object.

Section 112.2: Observer pattern

Observer Pattern's intent is to define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

The subject and observers define the one-to-many relationship. The observers are dependent on the subject such that when the subject's state changes, the observers get notified. Depending on the notification, the observers may also be updated with new values.

Here is the example from the book "Design Patterns" by Gamma.

```
#include <iostream>
#include <vector>

class Subject;

class Observer
{
public:
    virtual ~Observer() = default;
    virtual void Update(Subject&) = 0;
};

class Subject
{
public:
    virtual ~Subject() = default;
    void Attach(Observer& o) { observers.push_back(&o); }
    void Detach(Observer& o)
    {
        observers.erase(std::remove(observers.begin(), observers.end(), &o));
    }
    void Notify()
    {
        for (auto* o : observers) {
            o->Update(*this);
        }
    }
private:
    std::vector<Observer*> observers;
};

class ClockTimer : public Subject
{
public:
    void SetTime(int hour, int minute, int second)
    {
        this->hour = hour;
    }
}
```

```

this->minute = minute;
this->second = second;

    Notify();
}

int GetHour() const { return hour; }
int GetMinute() const { return minute; }
int GetSecond() const { return second; }

private:
    int hour;
    int minute;
    int second;
};

class DigitalClock: public Observer
{
public:
    explicit DigitalClock(ClockTimer& s) : subject(s) { subject.Attach(*this); }
    ~DigitalClock() { subjectDetach(*this); }
    void Update(Subject& theChangedSubject) override
    {
        if (&theChangedSubject == &subject) {
            Draw();
        }
    }

    void Draw()
    {
        int hour = subject.GetHour();
        int minute = subject.GetMinute();
        int second = subject.GetSecond();

        std::cout << "数字时间是 " << hour << ":"
              << minute << ":"
              << second << std::endl;
    }
}

private:
ClockTimer& subject;
};

class 模拟时钟: public 观察者
{
public:
    explicit 模拟时钟(ClockTimer& s) : subject(s) { subject.Attach(*this); }
    ~模拟时钟() { subjectDetach(*this); }
    void Update(Subject& theChangedSubject) override
    {
        if (&theChangedSubject == &subject) {
            Draw();
        }
    }

    void Draw()
    {
        int hour = subject.GetHour();
        int minute = subject.GetMinute();
        int second = subject.GetSecond();

        std::cout << "模拟时间是 " << hour << ":"
              << minute << ":";
    }
}

```

```

this->minute = minute;
this->second = second;

    Notify();
}

int GetHour() const { return hour; }
int GetMinute() const { return minute; }
int GetSecond() const { return second; }

private:
    int hour;
    int minute;
    int second;
};

class DigitalClock: public Observer
{
public:
    explicit DigitalClock(ClockTimer& s) : subject(s) { subject.Attach(*this); }
    ~DigitalClock() { subjectDetach(*this); }
    void Update(Subject& theChangedSubject) override
    {
        if (&theChangedSubject == &subject) {
            Draw();
        }
    }

    void Draw()
    {
        int hour = subject.GetHour();
        int minute = subject.GetMinute();
        int second = subject.GetSecond();

        std::cout << "Digital time is " << hour << ":"
              << minute << ":"
              << second << std::endl;
    }
}

private:
ClockTimer& subject;
};

class AnalogClock: public Observer
{
public:
    explicit AnalogClock(ClockTimer& s) : subject(s) { subject.Attach(*this); }
    ~AnalogClock() { subjectDetach(*this); }
    void Update(Subject& theChangedSubject) override
    {
        if (&theChangedSubject == &subject) {
            Draw();
        }
    }

    void Draw()
    {
        int hour = subject.GetHour();
        int minute = subject.GetMinute();
        int second = subject.GetSecond();

        std::cout << "Analog time is " << hour << ":"
              << minute << ":";
    }
}

```

```

    << second << std::endl;
}

private:
ClockTimer& subject;
};

int main()
{
ClockTimer 计时器;

DigitalClock 数字时钟(计时器);
AnalogClock 模拟时钟(计时器);

计时器.SetTime(14, 41, 36);
}

```

输出：

```

数字时间是14:41:36
模拟时间是14:41:36

```

以下是模式的总结：

1. 对象（数字时钟DigitalClock或模拟时钟AnalogClock对象）使用主题接口（Attach()或Detach()）来订阅（注册）为观察者或取消订阅（移除）自身作为观察者（subject.Attach(*this);, subjectDetach(*this);）。
2. 每个主题可以有多个观察者（vector<Observer*> observers;）。
3. 所有观察者都需要实现观察者接口。该接口只有一个方法，Update()，当主题状态改变时会被调用（Update(Subject &)）。
4. 除了Attach()和Detach()方法外，具体主题还实现了Notify()方法，用于在状态改变时更新所有当前观察者。但在本例中，这些都在父类Subject中完成（Subject::Attach(Observer&), void Subject::Detach(Observer&)和void Subject::Notify()）。

5. 具体对象还可以有设置和获取其状态的方法。

6. 具体观察者可以是任何实现观察者接口的类。每个观察者通过具体主题订阅（注册）以接收更新（subject.Attach(*this);）。

7. 观察者模式的两个对象是松散耦合的，它们可以交互但彼此了解很少彼此。

变体：

信号与槽

信号与槽是Qt引入的一种语言结构，它使实现观察者模式变得简单，同时避免了样板代码。其概念是控件（也称为小部件）可以发送包含事件信息的信号，其他控件可以使用称为槽的特殊函数接收这些信号。Qt中的槽必须是作为类成员声明的。信号/槽系统非常适合图形用户界面的设计。同样，信号/槽系统也可以用于异步I/O（包括套接字、管道、串行设备等）事件通知，或将超时事件与相应的对象实例和方法或函数关联。无需编写注册/注销/调用代码，因为Qt的元对象编译器（MOC）会自动生成所需的基础设施。

```

    << second << std::endl;
}

private:
ClockTimer& subject;
};

int main()
{
ClockTimer timer;

DigitalClock digitalClock(timer);
AnalogClock analogClock(timer);

timer.SetTime(14, 41, 36);
}

```

Output:

```

Digital time is 14:41:36
Analog time is 14:41:36

```

Here are the summary of the pattern:

1. Objects (DigitalClock or AnalogClock object) use the Subject interfaces (Attach() or Detach()) either to subscribe (register) as observers or unsubscribe (remove) themselves from being observers (subject.Attach(*this);, subjectDetach(*this);).
2. Each subject can have many observers (vector<Observer*> observers;).
3. All observers need to implement the Observer interface. This interface just has one method, Update(), that gets called when the Subject's state changes (Update(Subject &))
4. In addition to the Attach() and Detach() methods, the concrete subject implements a Notify() method that is used to update all the current observers whenever state changes. But in this case, all of them are done in the parent class, Subject (Subject::Attach (Observer&), void Subject::Detach(Observer&) and void Subject::Notify() .
5. The Concrete object may also have methods for setting and getting its state.
6. Concrete observers can be any class that implements the Observer interface. Each observer subscribe (register) with a concrete subject to receive update (subject.Attach(*this);).
7. The two objects of Observer Pattern are **loosely coupled**, they can interact but with little knowledge of each other.

Variation:

Signal and Slots

Signals and slots is a language construct introduced in Qt, which makes it easy to implement the Observer pattern while avoiding boilerplate code. The concept is that controls (also known as widgets) can send signals containing event information which can be received by other controls using special functions known as slots. The slot in Qt must be a class member declared as such. The signal/slot system fits well with the way Graphical User Interfaces are designed. Similarly, the signal/slot system can be used for asynchronous I/O (including sockets, pipes, serial devices, etc.) event notification or to associate timeout events with appropriate object instances and methods or functions. No registration/deregistration/invocation code need be written, because Qt's Meta Object Compiler (MOC) automatically generates the needed infrastructure.

C#语言也支持类似的结构，尽管术语和语法不同：事件扮演信号的角色，委托则是槽。此外，委托可以是局部变量，类似于函数指针，而Qt中的槽必须是作为类成员声明的。

第112.3节：工厂模式

工厂模式解耦对象创建，并允许通过通用接口按名称创建对象：

```
class Animal{
public:
    virtual std::shared_ptr<Animal> clone() const = 0;
    virtual std::string getname() const = 0;
};

class Bear: public Animal{
public:
    virtual std::shared_ptr<Animal> clone() const override
    {
        return std::make_shared<Bear>(*this);
    }
    virtual std::string getname() const override
    {
        return "bear";
    }
};

class Cat: public Animal{
public:
    virtual std::shared_ptr<Animal> clone() const override
    {
        return std::make_shared<Cat>(*this);
    }
    virtual std::string getname() const override
    {
        return "cat";
    }
};

class AnimalFactory{
public:
    static std::shared_ptr<Animal> getAnimal( const std::string& name )
    {
        if ( name == "bear" )
            return std::make_shared<Bear>();
        if ( name == "cat" )
            return std::shared_ptr<Cat>();

        return nullptr;
    }
};
```

第112.4节：带流式API的建造者模式

建造者模式将对象的创建与对象本身解耦。其主要思想是对象不必负责自身的创建。复杂对象的正确且有效的组装可能本身就是一项复杂的任务，因此这项任务可以委托给另一个类来完成。

The C# language also supports a similar construct although with a different terminology and syntax: events play the role of signals, and delegates are the slots. Additionally, a delegate can be a local variable, much like a function pointer, while a slot in Qt must be a class member declared as such.

Section 112.3: Factory Pattern

Factory pattern decouples object creation and allows creation by name using a common interface:

```
class Animal{
public:
    virtual std::shared_ptr<Animal> clone() const = 0;
    virtual std::string getname() const = 0;
};

class Bear: public Animal{
public:
    virtual std::shared_ptr<Animal> clone() const override
    {
        return std::make_shared<Bear>(*this);
    }
    virtual std::string getname() const override
    {
        return "bear";
    }
};

class Cat: public Animal{
public:
    virtual std::shared_ptr<Animal> clone() const override
    {
        return std::make_shared<Cat>(*this);
    }
    virtual std::string getname() const override
    {
        return "cat";
    }
};

class AnimalFactory{
public:
    static std::shared_ptr<Animal> getAnimal( const std::string& name )
    {
        if ( name == "bear" )
            return std::make_shared<Bear>();
        if ( name == "cat" )
            return std::shared_ptr<Cat>();

        return nullptr;
    }
};
```

Section 112.4: Builder Pattern with Fluent API

The Builder Pattern decouples the creation of the object from the object itself. The main idea behind is that **an object does not have to be responsible for its own creation**. The correct and valid assembly of a complex object may be a complicated task in itself, so this task can be delegated to another class.

受C#中电子邮件建造者的启发，我决定在这里做一个C++版本。电子邮件对象不一定是一个非常复杂的对象，但它可以演示该模式。

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

using namespace std;

// 前置声明建造者
class EmailBuilder;

class Email
{
    公共:
        friend class EmailBuilder; // 构建器可以访问 Email 的私有成员

    static EmailBuilder make();

    string to_string() const {
        stringstream stream;      strea
        m << "from: " << m_from << "to: " << m_to
            << "subject: " << m_subject << "bo
            dy: " << m_body; return strea
        m.str();
    }

    私有:
        Email() = default; // 限制构造函数为构建器

        string m_from;
        string m_to;
        string m_subject;
        string m_body;
    };
}

class EmailBuilder
{
    公共:
        EmailBuilder& from(const string &from) {
            m_email.m_from = from;
            return *this;
        }

        EmailBuilder& to(const string &to) {
            m_email.m_to = to;
            return *this;
        }

        EmailBuilder& subject(const string &subject) {
            m_email.m_subject = subject;
            return *this;
        }

        EmailBuilder& body(const string &body) {
            m_email.m_body = body;
            return *this;
        }

    operator Email&&() {

```

Inspired by the Email Builder in C#, I've decided to make a C++ version here. An Email object is not necessarily a *very complex object*, but it can demonstrate the pattern.

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

using namespace std;

// Forward declaring the builder
class EmailBuilder;

class Email
{
    公共:
        friend class EmailBuilder; // the builder can access Email's privates

    static EmailBuilder make();

    string to_string() const {
        stringstream stream;
        stream << "from: " << m_from
            << "\nto: " << m_to
            << "\nsubject: " << m_subject
            << "\nbody: " << m_body;
        return stream.str();
    }

    私有:
        Email() = default; // restrict construction to builder

        string m_from;
        string m_to;
        string m_subject;
        string m_body;
    };

    class EmailBuilder
    {
        公共:
            EmailBuilder& from(const string &from) {
                m_email.m_from = from;
                return *this;
            }

            EmailBuilder& to(const string &to) {
                m_email.m_to = to;
                return *this;
            }

            EmailBuilder& subject(const string &subject) {
                m_email.m_subject = subject;
                return *this;
            }

            EmailBuilder& body(const string &body) {
                m_email.m_body = body;
                return *this;
            }

        operator Email&&() {

```

```

        return std::move(m_email); // 注意这里使用了 move
    }

private:
Email m_email;
};

EmailBuilder Email::make()
{
    return EmailBuilder();
}

// 额外示例！
std::ostream& operator <<(std::ostream& stream, const Email& email)
{
    stream << email.to_string();
    return stream;
}

int main()
{
    Email mail = Email::make().from("me@mail.com")
        .to("you@mail.com")
        .subject("C++ builders")
        .body("我喜欢这个 API, 你呢？");

    cout << mail << endl;
}

```

对于较旧版本的C++, 可以忽略`std::move`操作, 并去掉转换运算符中的`&&` (尽管这会创建一个临时副本)。

构建器在通过`operator Email&&()`释放构建好的电子邮件时完成其工作。在此示例中, 构建器是一个临时对象, 并在被销毁前返回电子邮件。你也可以使用显式的操作, 如`Email EmailBuilder::build() { ... }`, 代替转换运算符。

传递构建器

构建者模式提供的一个很棒的功能是能够让多个参与者共同构建一个对象。这是通过将构建器传递给其他参与者来实现的, 每个参与者都会为构建的对象提供更多信息。

当你构建某种查询, 添加过滤器和其他规格时, 这尤其强大。

```

void add_addresses(EmailBuilder& builder)
{
    builder.from("me@mail.com")
        .to("you@mail.com");
}

void compose_mail(EmailBuilder& builder)
{
    builder.subject("我知道主题")
        .body("还有正文。地址由别人知道。");
}

int main()
{
    EmailBuilder builder;
    add_addresses(builder);
    compose_mail(builder);
}

```

```

        return std::move(m_email); // notice the move
    }

private:
Email m_email;
};

EmailBuilder Email::make()
{
    return EmailBuilder();
}

// Bonus example!
std::ostream& operator <<(std::ostream& stream, const Email& email)
{
    stream << email.to_string();
    return stream;
}

int main()
{
    Email mail = Email::make().from("me@mail.com")
        .to("you@mail.com")
        .subject("C++ builders")
        .body("I like this API, don't you?");

    cout << mail << endl;
}

```

For older versions of C++, one may just ignore the `std::move` operation and remove the `&&` from the conversion operator (although this will create a temporary copy).

The builder finishes its work when it releases the built email by the operator `Email&&()`. In this example, the builder is a temporary object and returns the email before being destroyed. You could also use an explicit operation like `Email EmailBuilder::build() { ... }` instead of the conversion operator.

Pass the builder around

A great feature the Builder Pattern provides is the ability to **use several actors to build an object together**. This is done by passing the builder to the other actors that will each one give some more information to the built object. This is specially powerful when you are building some sort of query, adding filters and other specifications.

```

void add_addresses(EmailBuilder& builder)
{
    builder.from("me@mail.com")
        .to("you@mail.com");
}

void compose_mail(EmailBuilder& builder)
{
    builder.subject("I know the subject")
        .body("And the body. Someone else knows the addresses.");
}

int main()
{
    EmailBuilder builder;
    add_addresses(builder);
    compose_mail(builder);
}

```

```
Email mail = builder;
cout << mail << endl;
}
```

设计变体：可变对象

您可以根据需要更改此模式的设计。我将给出一个变体。

在给定的示例中，Email 对象是不可变的，即其属性不能被修改，因为无法访问它们。这是一个预期的特性。如果您需要在创建对象后修改它，则必须为其提供一些设置方法。由于这些设置方法会在构建器中重复，您可以考虑将所有功能合并到一个类中（不再需要构建器类）。不过，我会首先考虑是否真的需要使构建的对象可变。

```
Email mail = builder;
cout << mail << endl;
}
```

Design variant : Mutable object

You can change the design of this pattern to fit your needs. I'll give one variant.

In the given example the Email object is immutable, i.e., its properties can't be modified because there is no access to them. This was a desired feature. If you need to modify the object after its creation you have to provide some setters to it. Since those setters would be duplicated in the builder, you may consider to do it all in one class (no builder class needed anymore). Nevertheless, I would consider the need to make the built object mutable in the first place.

第113章：单例设计模式

第113.1节：延迟初始化

此示例摘自此处的Q & A部分：<http://stackoverflow.com/a/1008289/3807729>

请参阅这篇文章，了解一个简单的延迟求值且保证销毁的单例设计：

[有人能给我一个C++中单例模式的示例吗？](#)

经典的惰性求值且正确销毁的单例模式。

```
类 S
{
    公共:
        静态 S& 获取实例()
        {
            静态 S 实例; // 保证被销毁。
                        // 首次使用时实例化。
            返回 实例;
        }
    私有:
        S() {};
                    // 构造函数？这里需要{}括号。

        // C++ 03
        // ======
        // 不要忘记声明这两个。你要确保它们
        // 不可接受，否则你可能会意外得到
        // 单例的副本。
        S(S const&);           // 不实现
        无效 操作符=(S const&); // 不实现

        // C++ 11
        // =====
        // 我们可以使用更好的技术来删除我们不想要的方法

    公共:
        S(S const&) = delete;
        void operator=(S const&) = delete;

        // 注意：Scott Meyers 在他的《Effective Modern C++》一
        // 书中提到，删除的函数通常应该是公有的，// 因为这会产生更好的错误
        // 信息，// 这是由于编译器在检查可访问性时会优先于删除状态
};


```

请参阅这篇关于何时使用单例的文章：（不常用）

[单例：应该如何使用](#)

请参阅这两篇关于初始化顺序及如何应对的文章：

[静态变量初始化顺序](#)

[查找 C++ 静态初始化顺序问题](#)

请参阅这篇描述生命周期的文章：

[C++ 函数中静态变量的生命周期是什么？](#)

请参阅这篇讨论单例线程相关影响的文章：

[Singleton 实例声明为 GetInstance 方法的静态变量](#)

Chapter 113: Singleton Design Pattern

Section 113.1: Lazy Initialization

This example has been lifted from the Q & A section here:<http://stackoverflow.com/a/1008289/3807729>

See this article for a simple design for a lazy evaluated with guaranteed destruction singleton:
[Can any one provide me a sample of Singleton in c++?](#)

The classic lazy evaluated and correctly destroyed singleton.

```
class S
{
    public:
        static S& getInstance()
        {
            static S instance; // Guaranteed to be destroyed.
                                // Instantiated on first use.
            return instance;
        }
    private:
        S() {};           // Constructor? (the {} brackets) are needed here.

        // C++ 03
        // =====
        // Don't forget to declare these two. You want to make sure they
        // are unacceptable otherwise you may accidentally get copies of
        // your singleton appearing.
        S(S const&);           // Don't Implement
        void operator=(S const&); // Don't implement

        // C++ 11
        // =====
        // We can use the better technique of deleting the methods
        // we don't want.
    public:
        S(S const&) = delete;
        void operator=(S const&) = delete;

        // Note: Scott Meyers mentions in his Effective Modern
        // C++ book, that deleted functions should generally
        // be public as it results in better error messages
        // due to the compilers behavior to check accessibility
        // before deleted status
};


```

See this article about when to use a singleton: (not often)

[Singleton: How should it be used](#)

See this two article about initialization order and how to cope:

[Static variables initialisation order](#)

[Finding C++ static initialization order problems](#)

See this article describing lifetimes:

[What is the lifetime of a static variable in a C++ function?](#)

See this article that discusses some threading implications to singletons:

[Singleton instance declared as static variable of GetInstance method](#)

第113.2节：静态析构安全的单例模式

在存在多个静态对象的情况下，有时需要保证单例不会被销毁，直到所有使用该单例的静态对象都不再需要它为止。

在这种情况下，可以使用std::shared_ptr来保持单例对所有用户的存活，即使程序结束时静态析构函数正在被调用：

```
class Singleton
{
public:
Singleton(Singleton const&) = delete;
Singleton& operator=(Singleton const&) = delete;

static std::shared_ptr<Singleton> instance()
{
    static std::shared_ptr<Singleton> s{new Singleton};
    return s;
}

private:
Singleton() {}
};
```

注意：此示例作为问答部分的答案出现。

第113.3节：线程安全的单例模式

版本 ≥ C++11

C++11标准保证函数作用域对象的初始化是以同步方式进行的。这可以用来实现带有延迟初始化的线程安全单例模式。

```
类 Foo
{
public:
    static Foo& instance()
    {
        static Foo inst;
        return inst;
    }
private:
    Foo() {}
    Foo(const Foo&) = delete;
    Foo& operator =(const Foo&) = delete;
};
```

第113.4节：子类

```
class API
{
public:
    static API& instance();

    virtual ~API() {}
```

Section 113.2: Static deinitialization-safe singleton

There are times with multiple static objects where you need to be able to guarantee that the *singleton* will not be destroyed until all the static objects that use the *singleton* no longer need it.

In this case std::shared_ptr can be used to keep the *singleton* alive for all users even when the static destructors are being called at the end of the program:

```
class Singleton
{
public:
    Singleton(Singleton const&) = delete;
    Singleton& operator=(Singleton const&) = delete;

    static std::shared_ptr<Singleton> instance()
    {
        static std::shared_ptr<Singleton> s{new Singleton};
        return s;
    }

private:
    Singleton() {}
};
```

NOTE: [This example appears as an answer in the Q&A section here.](#)

Section 113.3: Thread-safe Singleton

Version ≥ C++11

The C++11 standards guarantees that the initialization of function scope objects are initialized in a synchronized manner. This can be used to implement a thread-safe singleton with lazy initialization.

```
class Foo
{
public:
    static Foo& instance()
    {
        static Foo inst;
        return inst;
    }
private:
    Foo() {}
    Foo(const Foo&) = delete;
    Foo& operator =(const Foo&) = delete;
};
```

Section 113.4: Subclasses

```
class API
{
public:
    static API& instance();

    virtual ~API() {}
```

```

virtual const char* func1() = 0;
virtual void func2() = 0;

protected:
API() {}
API(const API&) = delete;
API& operator=(const API&) = delete;
};

class WindowsAPI : public API
{
public:
    virtual const char* func1() override { /* Windows 代码 */ }
    virtual void func2() override { /* Windows 代码 */ }
};

class LinuxAPI : public API
{
public:
    virtual const char* func1() override { /* Linux 代码 */ }
    virtual void func2() override { /* Linux 代码 */ }
};

API& API::instance() {
#if PLATFORM == WIN32
    static WindowsAPI 实例;
#elif PLATFORM == LINUX
    static LinuxAPI 实例;
#endif
    return 实例;
}

```

在此示例中，一个简单的编译器开关将API类绑定到相应的子类。通过这种方式，可以访问API而无需与特定平台代码耦合。

```

virtual const char* func1() = 0;
virtual void func2() = 0;

protected:
API() {}
API(const API&) = delete;
API& operator=(const API&) = delete;
};

class WindowsAPI : public API
{
public:
    virtual const char* func1() override { /* Windows code */ }
    virtual void func2() override { /* Windows code */ }
};

class LinuxAPI : public API
{
public:
    virtual const char* func1() override { /* Linux code */ }
    virtual void func2() override { /* Linux code */ }
};

API& API::instance() {
#if PLATFORM == WIN32
    static WindowsAPI instance;
#elif PLATFORM == LINUX
    static LinuxAPI instance;
#endif
    return instance;
}

```

In this example, a simple compiler switch binds the API class to the appropriate subclass. In this way, API can be accessed without being coupled to platform-specific code.

第114章：用户自定义字面量

第114.1节：自制的二进制用户自定义字面量

尽管你可以在C++14中这样写二进制数字：

```
int number =0b0001'0101; // ==21
```

这里有一个著名的自制二进制数实现示例：

注意：整个模板展开程序在编译时运行。

```
template<char FIRST,char... REST>struct binary
{
    static_assert( FIRST == '0' || FIRST == '1', "无效的二进制数字" );
    enum { value = ( ( FIRST - '0' ) << sizeof...(REST) ) + binary<REST...>::value } ;
};

template<> struct binary<'0'> { enum { value = 0 } ; };
template<> struct binary<'1'> { enum { value = 1 } ; };

// 原始字面量操作符
template<char... LITERAL> inline
constexpr unsigned int operator "" _b() { return binary<LITERAL...>::value; }

// 原始字面量操作符
template<char... LITERAL> inline
constexpr unsigned int operator "" _B() { return binary<LITERAL...>::value; }

#include <iostream>

int main()
{
    std::cout << 10101_B << ", " << 011011000111_b << " ; // 输出 21, 1735}
```

第114.2节：用于持续时间的标准用户自定义字面量

版本 ≥ C++14

以下持续时间用户字面量声明在命名空间 `std::literals::chrono_literals` 中，其中 `literals` 和 `chrono_literals` 都是内联命名空间。可以通过 `using namespace std::literals`、`using namespace std::chrono_literals` 和 `using namespace std::literals::chrono_literals` 来访问这些操作符。

```
#include <chrono>
#include <iostream>

int main()
{
    using namespace std::literals::chrono_literals;

    std::chrono::nanoseconds t1 = 600ns;
    std::chrono::microseconds t2 = 42us;
    std::chrono::milliseconds t3 = 51ms;
    std::chrono::seconds t4 = 61s;
    std::chrono::minutes t5 = 88min;
```

Chapter 114: User-Defined Literals

Section 114.1: Self-made user-defined literal for binary

Despite you can write a binary number in C++14 like:

```
int number =0b0001'0101; // ==21
```

here comes a famous example with a self-made implementation for binary numbers:

Note: The whole template expanding program is running at compile time.

```
template< char FIRST, char... REST > struct binary
{
    static_assert( FIRST == '0' || FIRST == '1', "invalid binary digit" );
    enum { value = ( ( FIRST - '0' ) << sizeof...(REST) ) + binary<REST...>::value } ;
};

template<> struct binary<'0'> { enum { value = 0 } ; };
template<> struct binary<'1'> { enum { value = 1 } ; };

// raw literal operator
template< char... LITERAL > inline
constexpr unsigned int operator "" _b() { return binary<LITERAL...>::value; }

// raw literal operator
template< char... LITERAL > inline
constexpr unsigned int operator "" _B() { return binary<LITERAL...>::value; }

#include <iostream>

int main()
{
    std::cout << 10101_B << ", " << 011011000111_b << '\n' ; // prints 21, 1735
}
```

Section 114.2: Standard user-defined literals for duration

Version ≥ C++14

Those following duration user literals are declared in the `namespace std::literals::chrono_literals`, where both `literals` and `chrono_literals` are inline namespaces. Access to these operators can be gained with `using namespace std::literals`, `using namespace std::chrono_literals`, and `using namespace std::literals::chrono_literals`.

```
#include <chrono>
#include <iostream>

int main()
{
    using namespace std::literals::chrono_literals;

    std::chrono::nanoseconds t1 = 600ns;
    std::chrono::microseconds t2 = 42us;
    std::chrono::milliseconds t3 = 51ms;
    std::chrono::seconds t4 = 61s;
    std::chrono::minutes t5 = 88min;
```

```

auto t6 = 2 * 0.5h;
auto total = t1 + t2 + t3 + t4 + t5 + t6;

std::cout.precision(13);
std::cout << total.count() << " 纳秒" << std::endl; // 8941051042600 纳秒
    std::cout << std::chrono::duration_cast<std::chrono::hours>(total).count()
        << " 小时" << std::endl; // 2 小时
}

```

第114.3节：带有长双精度值的用户自定义字面量

```

#include <iostream>

long double operator"" _km(long double val)
{
    return val * 1000.0;
}

long double operator"" _mi(long double val)
{
    return val * 1609.344;
}

int main()
{
    std::cout << "3 公里 = " << 3.0_km << " 米";    std::cout
    << "3 英里 = " << 3.0_mi << " 米";return 0;
}

```

该程序的输出如下：

```

3公里 = 3000 米
3mi = 4828.03 m

```

第114.4节：字符串的标准用户自定义字面量

版本 ≥ C++14

以下字符串用户字面量声明在 `namespace std::literals::string_literals` 中，其中 `literals` 和 `string_literals` 都是内联命名空间。可以通过 `using namespace std::literals`、`using namespace std::string_literals` 和 `using namespace std::literals::string_literals` 来访问这些操作符。

```

#include <codecvt>
#include <iostream>
#include <locale>
#include <string>

int main()
{
    using namespace std::literals::string_literals;

    std::string s = "hello world"s;
    std::u16string s16 = u"hello world"s;
    std::u32string s32 = U"hello world"s;
    std::wstring ws = L"hello world"s;
}

```

```

auto t6 = 2 * 0.5h;
auto total = t1 + t2 + t3 + t4 + t5 + t6;

std::cout.precision(13);
std::cout << total.count() << " nanoseconds" << std::endl; // 8941051042600 nanoseconds
    std::cout << std::chrono::duration_cast<std::chrono::hours>(total).count()
        << " hours" << std::endl; // 2 hours
}

```

Section 114.3: User-defined literals with long double values

```

#include <iostream>

long double operator"" _km(long double val)
{
    return val * 1000.0;
}

long double operator"" _mi(long double val)
{
    return val * 1609.344;
}

int main()
{
    std::cout << "3 km = " << 3.0_km << " m\n";
    std::cout << "3 mi = " << 3.0_mi << " m\n";
    return 0;
}

```

The output of this program is the following:

```

3 km = 3000 m
3 mi = 4828.03 m

```

Section 114.4: Standard user-defined literals for strings

Version ≥ C++14

Those following string user literals are declared in the `namespace std::literals::string_literals`, where both `literals` and `string_literals` are inline namespaces. Access to these operators can be gained with `using namespace std::literals`, `using namespace std::string_literals`, and `using namespace std::literals::string_literals`.

```

#include <codecvt>
#include <iostream>
#include <locale>
#include <string>

int main()
{
    using namespace std::literals::string_literals;

    std::string s = "hello world"s;
    std::u16string s16 = u"hello world"s;
    std::u32string s32 = U"hello world"s;
    std::wstring ws = L"hello world"s;
}

```

```

std::cout << s << std::endl;

std::wstring_convert<std::codecvt_utf8_utf16<char16_t>, char16_t> utf16conv;
std::cout << utf16conv.to_bytes(s16) << std::endl;

std::wstring_convert<std::codecvt_utf8_utf16<char32_t>, char32_t> utf32conv;
std::cout << utf32conv.to_bytes(s32) << std::endl;

std::wcout << ws << std::endl;
}

```

注意：

字面字符串可能包含 \0

```

std::string s1 = "foo\0\0bar"; // 从 C 字符串构造：结果为 "foo"s
std::string s2 = "foo\0\0bar"s; // 该字符串中间包含两个 '\0'

```

第114.5节：复数的标准用户自定义字面量

版本 ≥ C++14

以下复数用户字面量声明在 `namespace std::literals::complex_literals` 中，其中 `literals` 和 `complex_literals` 都是内联命名空间。可以通过 `using namespace std::literals`, `using namespace std::complex_literals` 和 `using namespace std::literals::complex_literals` 来访问这些操作符。

```

#include <complex>
#include <iostream>

int main()
{
    using namespace std::literals::complex_literals;

    std::complex<double> c = 2.0 + 1i;      // {2.0, 1.}
    std::complex<float> cf = 2.0f + 1if;     // {2.0f, 1.f}
    std::complex<long double> cl = 2.0L + 1il; // {2.0L, 1.L}

    std::cout << "abs" << c << " = " << abs(c) << std::endl; // abs(2,1) = 2.23607
    std::cout << "abs" << cf << " = " << abs(cf) << std::endl; // abs(2,1) = 2.23607
    std::cout << "abs" << cl << " = " << abs(cl) << std::endl; // abs(2,1) = 2.23607
}

```

```

std::cout << s << std::endl;

std::wstring_convert<std::codecvt_utf8_utf16<char16_t>, char16_t> utf16conv;
std::cout << utf16conv.to_bytes(s16) << std::endl;

std::wstring_convert<std::codecvt_utf8_utf16<char32_t>, char32_t> utf32conv;
std::cout << utf32conv.to_bytes(s32) << std::endl;

std::wcout << ws << std::endl;
}

```

Note:

Literal string may containing \0

```

std::string s1 = "foo\0\0bar"; // constructor from C-string: results in "foo"s
std::string s2 = "foo\0\0bar"s; // That string contains 2 '\0' in its middle

```

Section 114.5: Standard user-defined literals for complex

Version ≥ C++14

Those following complex user literals are declared in the `namespace std::literals::complex_literals`, where both `literals` and `complex_literals` are inline namespaces. Access to these operators can be gained with `using namespace std::literals`, `using namespace std::complex_literals`, and `using namespace std::literals::complex_literals`.

```

#include <complex>
#include <iostream>

int main()
{
    using namespace std::literals::complex_literals;

    std::complex<double> c = 2.0 + 1i;      // {2.0, 1.}
    std::complex<float> cf = 2.0f + 1if;     // {2.0f, 1.f}
    std::complex<long double> cl = 2.0L + 1il; // {2.0L, 1.L}

    std::cout << "abs" << c << " = " << abs(c) << std::endl; // abs(2,1) = 2.23607
    std::cout << "abs" << cf << " = " << abs(cf) << std::endl; // abs(2,1) = 2.23607
    std::cout << "abs" << cl << " = " << abs(cl) << std::endl; // abs(2,1) = 2.23607
}

```

第115章：内存管理

第115.1节：自由存储（堆，动态分配...）

术语“堆”是一个通用的计算机术语，指的是一块内存区域，可以从中独立于栈分配和释放内存。

在C++中，标准将这块区域称为自由存储区（Free Store），认为这是一个更准确的术语。

从自由存储区分配的内存区域的生命周期可能比其最初分配时的作用域更长。

过大而无法存放在栈上的数据也可以从自由存储区分配。

原始内存可以通过new和delete关键字进行分配和释放。

```
float *foo = nullptr;
{
    *foo = new float; // 为一个float分配内存
    float bar;        // 栈上分配
} // bar的生命周期结束，而foo仍然存活 delete foo;

// 释放foo指向的float内存，使指针失效 foo = nullptr; // delete后将指针设为nullptr通常被认为是良好习惯
```

也可以使用new和delete分配固定大小的数组，语法略有不同。数组分配与非数组分配不兼容，混用会导致堆损坏。分配数组时，还会以实现定义的方式分配内存来跟踪数组大小，以便后续删除。

```
// 为一个包含256个整数的数组分配内存
int *foo = new int[256];
// 删除 foo 处的一个包含 256 个整数的数组
delete[] foo;
```

当使用new和delete代替malloc和free时，构造函数和析构函数会被执行（类似于基于栈的对象）。这就是为什么new和delete优于malloc和free的原因。

```
struct ComplexType {
    int a = 0;

ComplexType() { std::cout << "Ctor" << std::endl; }
~ComplexType() { std::cout << "Dtor" << std::endl; }
};

// 为 ComplexType 分配内存，并调用其构造函数
ComplexType *foo = new ComplexType();
// 调用 ComplexType() 的析构函数并删除 pC 处的 ComplexType 内存
delete foo;
```

从 C++11 开始，推荐使用智能指针来表示所有权。

版本 ≥ C++14

C++14 向 STL 添加了std::make_unique，推荐使用std::make_unique或std::make_shared代替直接使用裸指针的new和delete。

Chapter 115: Memory management

Section 115.1: Free Storage (Heap, Dynamic Allocation ...)

The term '**heap**' is a general computing term meaning an area of memory from which portions can be allocated and deallocated independently of the memory provided by the **stack**.

In C++ the Standard refers to this area as the **Free Store** which is considered a more accurate term.

Areas of memory allocated from the **Free Store** may live longer than the original scope in which it was allocated. Data too large to be stored on the stack may also be allocated from the **Free Store**.

Raw memory can be allocated and deallocated by the *new* and *delete* keywords.

```
float *foo = nullptr;
{
    *foo = new float; // Allocates memory for a float
    float bar;        // Stack allocated
} // End lifetime of bar, while foo still alive

delete foo;           // Deletes the memory for the float at pF, invalidating the pointer
foo = nullptr;         // Setting the pointer to nullptr after delete is often considered good
practice
```

It's also possible to allocate fixed size arrays with *new* and *delete*, with a slightly different syntax. Array allocation is not compatible with non-array allocation, and mixing the two will lead to heap corruption. Allocating an array also allocates memory to track the size of the array for later deletion in an implementation-defined way.

```
// Allocates memory for an array of 256 ints
int *foo = new int[256];
// Deletes an array of 256 ints at foo
delete[] foo;
```

When using *new* and *delete* instead *malloc* and *free*, the constructor and destructor will get executed (Similar to stack based objects). This is why *new* and *delete* are preferred over *malloc* and *free*.

```
struct ComplexType {
    int a = 0;

ComplexType() { std::cout << "Ctor" << std::endl; }
~ComplexType() { std::cout << "Dtor" << std::endl; }
};

// Allocates memory for a ComplexType, and calls its constructor
ComplexType *foo = new ComplexType();
// Calls the destructor for ComplexType() and deletes memory for a ComplexType at pC
delete foo;
Version ≥ C++11
```

From C++11 on, the use of smart pointers is recommended for indicating ownership.

Version ≥ C++14

C++14 added std::make_unique to the STL, changing the recommendation to favor std::make_unique or std::make_shared instead of using naked new and delete.

第 115.2 节：定位 new

有些情况下，我们不想依赖自由存储区（Free Store）来分配内存，而是想使用自定义的内存分配方式，使用`new`操作符。

针对这些情况，我们可以使用定位`new`（Placement New），告诉`new`操作符从预先分配的内存位置分配内存。

例如

```
int a4字节整数;  
char *a4字节字符 = new (&a4字节整数) char[4];
```

在这个例子中，`a4字节字符`指向的内存是通过整数变量`a4字节整数`在“栈”上分配的4字节内存。

这种内存分配方式的好处是程序员可以控制分配过程。在上面的例子中，由于`a4字节整数`是在栈上分配的，我们不需要显式调用`delete a4字节字符`。

同样的行为也可以应用于动态分配的内存。例如

```
int *a8字节动态整数 = new int[2];  
char *a8字节字符 = new (a8字节动态整数) char[8];
```

在这种情况下，`a8字节字符`指向的内存是由`a8字节动态整数`动态分配的内存。但在这种情况下，我们需要显式调用`delete a8字节动态整数`来释放内存。

C++ 类的另一个示例

```
struct ComplexType {  
    int a;  
  
    ComplexType() : a(0) {}  
    ~ComplexType() {}  
};  
  
int main() {  
    char* dynArray = new char[256];  
  
    //调用 ComplexType 的构造函数，将内存初始化为 ComplexType  
    new((void*)dynArray) ComplexType();  
  
    //完成后清理内存  
    reinterpret_cast<ComplexType*>(dynArray)->~ComplexType();  
    delete[] dynArray;  
  
    //栈内存也可以与定位 new 一起使用  
    alignas(ComplexType) char localArray[256]; //alignas() 自 C++11 起可用  
  
    new((void*)localArray) ComplexType();  
  
    //栈内存只需调用析构函数  
    reinterpret_cast<ComplexType*>(localArray)->~ComplexType();  
  
    return 0;  
}
```

Section 115.2: Placement new

There are situations when we don't want to rely upon Free Store for allocating memory and we want to use custom memory allocations using `new`.

For these situations we can use Placement `New`, where we can tell `new` operator to allocate memory from a pre-allocated memory location

For example

```
int a4byteInteger;  
char *a4byteChar = new (&a4byteInteger) char[4];
```

In this example, the memory pointed by `a4byteChar` is 4 byte allocated to 'stack' via integer variable `a4byteInteger`.

The benefit of this kind of memory allocation is the fact that programmers control the allocation. In the example above, since `a4byteInteger` is allocated on stack, we don't need to make an explicit call to 'delete `a4byteChar`'.

Same behavior can be achieved for dynamic allocated memory also. For example

```
int *a8byteDynamicInteger = new int[2];  
char *a8byteChar = new (a8byteDynamicInteger) char[8];
```

In this case, the memory pointer by `a8byteChar` will be referring to dynamic memory allocated by `a8byteDynamicInteger`. In this case however, we need to explicitly call `delete a8byteDynamicInteger` to release the memory

Another example for C++ Class

```
struct ComplexType {  
    int a;  
  
    ComplexType() : a(0) {}  
    ~ComplexType() {}  
};  
  
int main() {  
    char* dynArray = new char[256];  
  
    //Calls ComplexType's constructor to initialize memory as a ComplexType  
    new((void*)dynArray) ComplexType();  
  
    //Clean up memory once we're done  
    reinterpret_cast<ComplexType*>(dynArray)->~ComplexType();  
    delete[] dynArray;  
  
    //Stack memory can also be used with placement new  
    alignas(ComplexType) char localArray[256]; //alignas() available since C++11  
  
    new((void*)localArray) ComplexType();  
  
    //Only need to call the destructor for stack memory  
    reinterpret_cast<ComplexType*>(localArray)->~ComplexType();  
  
    return 0;  
}
```

第115.3节：栈

栈是一个小的内存区域，执行期间临时值会被放置在这里。与堆分配相比，分配到栈上的数据非常快，因为所有内存已经为此目的分配好了。

```
int main() {
    int a = 0; //存储在栈上
    return a;
}
```

之所以称为栈，是因为函数调用链会将它们的临时内存“堆叠”在彼此之上，每个调用使用一小段独立的内存。

```
float bar() {
    //f 会在其他变量之后被放置到栈上
    float f = 2;
    return f;
}

double foo() {
    //d 会紧跟在 main() 内的其他变量之后被放置
    double d = bar();
    return d;
}

int main() {
    //在调用 foo() 之前，栈中没有存储用户变量
    return (int)foo();
}
```

存储在栈上的数据仅在分配该变量的作用域仍然有效时才有效。

```
int* pA = nullptr;

void foo() {
    int b = *pA;
    pA = &b;
}

int main() {
    int a = 5;
    pA = &a;
    foo();
    //未定义行为，pA 指向的值已不在作用域内
    a = *pA;
}
```

Section 115.3: Stack

The stack is a small region of memory into which temporary values are placed during execution. Allocating data into the stack is very fast compared to heap allocation, as all the memory has already been assigned for this purpose.

```
int main() {
    int a = 0; //Stored on the stack
    return a;
}
```

The stack is named because chains of function calls will have their temporary memory 'stacked' on top of each other, each one using a separate small section of memory.

```
float bar() {
    //f will be placed on the stack after anything else
    float f = 2;
    return f;
}

double foo() {
    //d will be placed just after anything within main()
    double d = bar();
    return d;
}

int main() {
    //The stack has no user variables stored in it until foo() is called
    return (int)foo();
}
```

Data stored on the stack is only valid so long as the scope that allocated the variable is still active.

```
int* pA = nullptr;

void foo() {
    int b = *pA;
    pA = &b;
}

int main() {
    int a = 5;
    pA = &a;
    foo();
    //Undefined behavior, the value pointed to by pA is no longer in scope
    a = *pA;
}
```

第116章：C++11内存模型

不同线程尝试访问同一内存位置时，如果至少有一个操作是修改操作（也称为存储操作），则会发生数据竞争。这些数据竞争会导致未定义行为。为避免这种情况，需要防止这些线程同时执行此类冲突操作。

同步原语（互斥锁、临界区等）可以保护此类访问。C++11引入的内存模型定义了两种新的可移植方式，用于在多线程环境中同步访问内存：原子操作和屏障。

原子操作

现在可以通过使用原子加载和原子存储操作来读取和写入指定的内存位置。

为了方便，这些操作被封装在std::atomic<t>模板类中。该类封装了类型为t的值，但这次对对象的加载和存储是原子的。

该模板并非对所有类型都可用。可用的类型取决于具体实现，但通常包括大多数（或全部）可用的整型类型以及指针类型。因此，std::atomic<unsigned>和std::atomic<std::vector<foo> *>应该是可用的，而std::atomic<std::pair<bool,char>>很可能不可用。

原子操作具有以下属性：

- 所有原子操作都可以从多个线程并发执行，而不会导致未定义行为。
- 一次原子加载将看到原子对象构造时的初始值，或者通过某个原子存储操作写入的值。
- 对同一原子对象的原子存储在所有线程中顺序一致。如果某线程已经看到某个原子存储操作的值，后续的原子加载操作将看到相同的值，或者后续原子存储操作存储的值。
- 原子读-改-写操作允许原子加载和原子存储之间没有其他原子存储操作。例如，可以从多个线程原子地递增计数器，无论线程间的竞争如何，都不会丢失任何递增。
- 原子操作接受一个可选的std::memory_order参数，该参数定义了该操作相对于其他内存位置的附加属性
 - 无额外限制
 - 如果load-acquire看到由store-release存储的值，则在store-release之前顺序执行的存储发生在load acquire之后顺序执行的加载之前
 - 类似于memory_order_acquire但仅适用于依赖加载
 - 结合了load-acquire和store-release
 - 顺序一致性

std::memory_order	含义
std::memory_order_relaxed	无额外限制
std::memory_order_release → std::memory_order_acquire	如果load-acquire看到由store-release存储的值，则在store-release之前顺序执行的存储发生在load acquire之后顺序执行的加载之前
std::memory_order_consume	类似于memory_order_acquire但仅适用于依赖加载
std::memory_order_acq_rel	结合了load-acquire和store-release
std::memory_order_seq_cst	顺序一致性

这些内存顺序标签允许三种不同的内存排序规则：顺序一致性、放松，以及释放-获取及其对应的释放-消费。

顺序一致性

如果未为原子操作指定内存顺序，则默认顺序为顺序一致性。该模式也可以通过给操作加上std::memory_order_seq_cst标签来显式选择。

Chapter 116: C++11 Memory Model

Different threads trying to access the same memory location participate in a *data race* if at least one of the operations is a modification (also known as *store operation*). These *data races* cause *undefined behavior*. To avoid them one needs to prevent these threads from concurrently executing such conflicting operations.

Synchronization primitives (mutex, critical section and the like) can guard such accesses. The Memory Model introduced in C++11 defines two new portable ways to synchronize access to memory in multi-threaded environment: atomic operations and fences.

Atomic Operations

It is now possible to read and write to given memory location by the use of *atomic load* and *atomic store* operations. For convenience these are wrapped in the std::atomic<t> template class. This class wraps a value of type t but this time *loads* and *stores* to the object are atomic.

The template is not available for all types. Which types are available is implementation specific, but this usually includes most (or all) available integral types as well as pointer types. So that std::atomic<unsigned> and std::atomic<std::vector<foo> *> should be available, while std::atomic<std::pair<bool,char>> most probably wont be.

Atomic operations have the following properties:

- All atomic operations can be performed concurrently from multiple threads without causing undefined behavior.
- An *atomic load* will see either the initial value which the atomic object was constructed with, or the value written to it via some *atomic store* operation.
- Atomic stores* to the same atomic object are ordered the same in all threads. If a thread has already seen the value of some *atomic store* operation, subsequent *atomic load* operations will see either the same value, or the value stored by subsequent *atomic store* operation.
- Atomic read-modify-write* operations allow *atomic load* and *atomic store* to happen without other *atomic store* in between. For example one can atomically increment a counter from multiple threads, and no increment will be lost regardless of the contention between the threads.
- Atomic operations receive an optional std::memory_order parameter which defines what additional properties the operation has regarding other memory locations.

std::memory_order	Meaning
std::memory_order_relaxed	no additional restrictions
std::memory_order_release → std::memory_order_acquire	if load-acquire sees the value stored by store-release then stores sequenced before the store-release happen before loads sequenced after the load acquire
std::memory_order_consume	like memory_order_acquire but only for dependent loads
std::memory_order_acq_rel	combines load-acquire and store-release
std::memory_order_seq_cst	sequential consistency

These memory order tags allow three different memory ordering disciplines: *sequential consistency*, *relaxed*, and *release-acquire* with its sibling *release-consume*.

Sequential Consistency

If no memory order is specified for an atomic operation, the order defaults to *sequential consistency*. This mode can also be explicitly selected by tagging the operation with std::memory_order_seq_cst.

在此顺序下，没有任何内存操作可以跨越原子操作。所有在原子操作之前排序的内存操作都发生在原子操作之前，且原子操作发生在所有在其之后排序的内存操作之前。该模式可能是最容易理解的，但也会带来最大的性能损失。它还阻止了所有可能试图将操作重排序到原子操作之后的编译器优化。

放松排序

与顺序一致性相反的是放松内存排序。它通过 `std::memory_order_relaxed` 标签选择。放松的原子操作不会对其他内存操作施加任何限制。唯一保留的效果是该操作本身仍然是原子的。

释放-获取排序

一个原子存储操作可以加上 `std::memory_order_release` 标签，一个原子加载操作可以加上 `std::memory_order_acquire` 标签。第一个操作称为（原子）存储释放，第二个操作称为（原子）加载获取。

当 `load-acquire` 看到由 `store-release` 写入的值时，发生以下情况：所有在 `store-release` 之前顺序执行的存储操作对在 `load-acquire` 之后顺序执行的加载操作变得可见（发生在之前）。

原子读-改写-写入操作也可以接收累积标签 `std::memory_order_acq_rel`。这使得操作中的原子加载部分成为原子加载-获取，而原子存储部分则变为原子存储-释放。

编译器不允许将存储操作移动到原子存储-释放操作之后，也不允许将加载操作移动到原子加载-获取（或加载-消费）之前。

还要注意，没有原子加载-释放或原子存储-获取。尝试创建此类操作会使它们成为松散(relaxed)操作。

释放-消费顺序

这种组合类似于释放-获取，但这次原子加载被标记为 `std::memory_order_consume`，变为（原子）加载-消费操作。此模式与释放-获取相同，唯一的区别是，在加载-消费之后排序的加载操作中，只有依赖于加载-消费加载值的操作被排序。

栅栏

栅栏也允许线程间的内存操作排序。栅栏可以是释放栅栏或获取栅栏。

如果释放栅栏发生在获取栅栏之前，那么在释放栅栏之前排序的存储对在获取栅栏之后排序的加载是可见的。为了保证释放栅栏发生在获取栅栏之前，可以使用包括松散原子操作在内的其他同步原语。

第116.1节：内存模型的必要性

```
int x, y;
bool ready = false;

void init()
{
    x = 2;
```

With this order no memory operation can cross the atomic operation. All memory operations sequenced before the atomic operation happen before the atomic operation and the atomic operation happens before all memory operations that are sequenced after it. This mode is probably the easiest one to reason about but it also leads to the greatest penalty to performance. It also prevents all compiler optimizations that might otherwise try to reorder operations past the atomic operation.

Relaxed Ordering

The opposite to *sequential consistency* is the *relaxed* memory ordering. It is selected with the `std::memory_order_relaxed` tag. Relaxed atomic operation will impose no restrictions on other memory operations. The only effect that remains, is that the operation is itself still atomic.

Release-Acquire Ordering

An *atomic store* operation can be tagged with `std::memory_order_release` and an *atomic load* operation can be tagged with `std::memory_order_acquire`. The first operation is called (*atomic*) *store-release* while the second is called (*atomic*) *load-acquire*.

When *load-acquire* sees the value written by a *store-release* the following happens: all store operations sequenced before the *store-release* become visible to (*happen before*) load operations that are sequenced after the *load-acquire*.

Atomic read-modify-write operations can also receive the cumulative tag `std::memory_order_acq_rel`. This makes the *atomic load* portion of the operation an *atomic load-acquire* while the *atomic store* portion becomes *atomic store-release*.

The compiler is not allowed to move store operations after an *atomic store-release* operation. It is also not allowed to move load operations before *atomic load-acquire* (or *load-consume*).

Also note that there is no *atomic load-release* or *atomic store-acquire*. Attempting to create such operations makes them *relaxed* operations.

Release-Consume Ordering

This combination is similar to *release-acquire*, but this time the *atomic load* is tagged with `std::memory_order_consume` and becomes (*atomic*) *load-consume* operation. This mode is the same as *release-acquire* with the only difference that among the load operations sequenced after the *load-consume* only those depending on the value loaded by the *load-consume* are ordered.

Fences

Fences also allow memory operations to be ordered between threads. A fence is either a release fence or acquire fence.

If a release fence happens before an acquire fence, then stores sequenced before the release fence are visible to loads sequenced after the acquire fence. To guarantee that the release fence happens before the acquire fence one may use other synchronization primitives including relaxed atomic operations.

Section 116.1: Need for Memory Model

```
int x, y;
bool ready = false;

void init()
{
    x = 2;
```

```

y = 3;
ready = true;
}
void use()
{
    if (ready)
        std::cout << x + y;
}

```

一个线程调用init()函数，而另一个线程（或信号处理器）调用use()函数。人们可能会期望use()函数要么打印5，要么什么也不做。但由于多种原因，情况可能并非总是如此：

- CPU 可能会重新排序init()中发生的写操作，使得实际执行的代码看起来像是：

```

void init()
{
    ready = true;
    x = 2;
    y = 3;
}

```

- CPU 可能会重新排序在 use() 中发生的读取操作，因此实际执行的代码可能变为：

```

void use()
{
    int local_x = x;
    int local_y = y;
    if (ready)
        std::cout << local_x + local_y;
}

```

- 一个优化的 C++ 编译器可能会以类似的方式决定重新排序程序。

这种重新排序不会改变单线程程序的行为，因为线程不能交错调用 init() 和 use()。另一方面，在多线程环境中，一个线程可能会看到另一个线程执行的部分写操作，这可能导致 use() 看到 ready==true 但 x 或 y 或两者都是垃圾值。

C++ 内存模型允许程序员指定允许和不允许的重新排序操作，从而使多线程程序也能按预期运行。上述示例可以这样改写为线程安全的方式：

```

int x, y;
std::atomic<bool> ready{false};

void init()
{
    x = 2;
    y = 3;
    ready.store(true, std::memory_order_release);
}
void use()
{
    if (ready.load(std::memory_order_acquire))
        std::cout << x + y;
}

```

```

y = 3;
ready = true;
}
void use()
{
    if (ready)
        std::cout << x + y;
}

```

One thread calls the init() function while another thread (or signal handler) calls the use() function. One might expect that the use() function will either print 5 or do nothing. This may not always be the case for several reasons:

- The CPU may reorder the writes that happen in init() so that the code that actually executes might look like:

```

void init()
{
    ready = true;
    x = 2;
    y = 3;
}

```

- The CPU may reorder the reads that happen in use() so that the actually executed code might become:

```

void use()
{
    int local_x = x;
    int local_y = y;
    if (ready)
        std::cout << local_x + local_y;
}

```

- An optimizing C++ compiler may decide to reorder the program in similar way.

Such reordering cannot change the behavior of a program running in single thread because a thread cannot interleave the calls to init() and use(). On the other hand in a multi-threaded setting one thread may see part of the writes performed by the other thread where it may happen that use() may see ready==true and garbage in x or y or both.

The C++ Memory Model allows the programmer to specify which reordering operations are permitted and which are not, so that a multi-threaded program would also be able to behave as expected. The example above can be rewritten in thread-safe way like this:

```

int x, y;
std::atomic<bool> ready{false};

void init()
{
    x = 2;
    y = 3;
    ready.store(true, std::memory_order_release);
}
void use()
{
    if (ready.load(std::memory_order_acquire))
        std::cout << x + y;
}

```

这里 init() 执行了 原子存储释放 (atomic store-release) 操作。这不仅将值 true 存储到 ready 中，还告诉编译器不能将此操作移动到在它之前的写操作之前。

use() 函数执行了 原子加载获取 (atomic load-acquire) 操作。它读取 ready 的当前值，同时禁止编译器将序列在它之后的读操作提前到 原子加载获取 之前执行。

这些原子操作还会促使编译器插入必要的硬件指令，通知CPU避免不希望的重排序。

由于 原子存储释放 和 原子加载获取 操作针对的是同一内存位置，内存模型规定：如果 加载获取 操作看到的是 存储释放 操作写入的值，那么 init() 线程在该 存储释放 之前执行的所有写操作，对 use() 线程在其 加载获取 之后执行的加载操作都是可见的。也就是说，如果 use() 看到 ready==true，则保证它能看到 x==2 和 y==3。

注意，编译器和CPU仍然允许先写入 y 再写入 x，同样 use() 中对这些变量的读取也可以以任意顺序发生。

第116.2节：栅栏示例

上述示例也可以通过栅栏和松散原子操作来实现：

```
int x, y;
std::atomic<bool> ready{false};

void init()
{
    x = 2;
    y = 3;
    atomic_thread_fence(std::memory_order_release);
    ready.store(true, std::memory_order_relaxed);
}
void use()
{
    if (ready.load(std::memory_order_relaxed))
    {
        atomic_thread_fence(std::memory_order_acquire);
        std::cout << x + y;
    }
}
```

如果原子加载操作看到的是原子存储写入的值，那么存储操作发生在加载操作之前，栅栏也是如此：释放栅栏发生在获取栅栏之前，使得在释放栅栏之前对 x 和 y 的写入对紧随获取栅栏之后的 std::cout 语句可见。

如果栅栏能够减少整体的获取、释放或其他同步操作的数量，则栅栏可能是有益的。例如：

```
void block_and_use()
{
    while (!ready.load(std::memory_order_relaxed))
        ;
    atomic_thread_fence(std::memory_order_acquire);
    std::cout << x + y;
}
```

block_and_use() 函数通过松散原子加载不断轮询直到 ready 标志被设置。然后使用单个获取栅栏来提供所需的内存顺序。

Here init() performs *atomic store-release* operation. This not only stores the value `true` into `ready`, but also tells the compiler that it cannot move this operation before write operations that are *sequenced before* it.

The `use()` function does an *atomic load-acquire* operation. It reads the current value of `ready` and also forbids the compiler from placing read operations that are *sequenced after* it to *happen before* the *atomic load-acquire*.

These atomic operations also cause the compiler to put whatever hardware instructions are needed to inform the CPU to refrain from the unwanted reorderings.

Because the *atomic store-release* is to the same memory location as the *atomic load-acquire*, the memory model stipulates that if the *load-acquire* operation sees the value written by the *store-release* operation, then all writes performed by `init()`'s thread prior to that *store-release* will be visible to loads that `use()`'s thread executes after its *load-acquire*. That is if `use()` sees `ready==true`, then it is guaranteed to see `x==2` and `y==3`.

Note that the compiler and the CPU are still allowed to write to `y` before writing to `x`, and similarly the reads from these variables in `use()` can happen in any order.

Section 116.2: Fence example

The example above can also be implemented with fences and relaxed atomic operations:

```
int x, y;
std::atomic<bool> ready{false};

void init()
{
    x = 2;
    y = 3;
    atomic_thread_fence(std::memory_order_release);
    ready.store(true, std::memory_order_relaxed);
}
void use()
{
    if (ready.load(std::memory_order_relaxed))
    {
        atomic_thread_fence(std::memory_order_acquire);
        std::cout << x + y;
    }
}
```

If the atomic load operation sees the value written by the atomic store then the store happens before the load, and so do the fences: the release fence happens before the acquire fence making the writes to `x` and `y` that precede the release fence to become visible to the `std::cout` statement that follows the acquire fence.

A fence might be beneficial if it can reduce the overall number of acquire, release or other synchronization operations. For example:

```
void block_and_use()
{
    while (!ready.load(std::memory_order_relaxed))
        ;
    atomic_thread_fence(std::memory_order_acquire);
    std::cout << x + y;
}
```

The `block_and_use()` function spins until the `ready` flag is set with the help of relaxed atomic load. Then a single acquire fence is used to provide the needed memory ordering.

第117章：作用域

第117.1节：全局变量

为了声明一个在不同源文件中都可以访问的变量实例，可以使用关键字extern将其声明在全局作用域中。该关键字告诉编译器代码的某处有该变量的定义，因此它可以在任何地方使用，所有的读写操作都将在同一块内存中进行。

```
// 文件 my_globals.h :  
  
#ifndef __MY_GLOBALS_H__  
#define __MY_GLOBALS_H__  
  
extern int circle_radius; // 向编译器承诺 circle_radius  
                         // 将在某处被定义
```

```
#endif
```

```
// 文件 foo1.cpp :  
  
#include "my_globals.h"  
  
int circle_radius = 123; // 定义 extern 变量
```

```
// 文件 main.cpp :  
  
#include "my_globals.h"  
#include <iostream>  
  
int main()  
{  
    std::cout << "半径是: " << circle_radius << "";  
    return 0;  
}
```

输出：

```
半径是: 123
```

第117.2节：简单块作用域

块中变量的作用域 { ... }，始于声明之后，止于块的末尾。如果存在嵌套块，内层块可以隐藏在外层块中声明的变量的作用域。

```
{  
    int x = 100;  
    // ^  
    // `x` 的作用域从这里开始  
    //  
} // <- `x` 的作用域在这里结束
```

如果嵌套块在外层块内开始，之前在

Chapter 117: Scopes

Section 117.1: Global variables

To declare a single instance of a variable which is accessible in different source files, it is possible to make it in the global scope with keyword `extern`. This keyword says the compiler that somewhere in the code there is a definition for this variable, so it can be used everywhere and all write/read will be done in one place of memory.

```
// File my_globals.h:  
  
#ifndef __MY_GLOBALS_H__  
#define __MY_GLOBALS_H__  
  
extern int circle_radius; // Promise to the compiler that circle_radius  
                         // will be defined somewhere  
  
#endif
```

```
// File foo1.cpp:  
  
#include "my_globals.h"  
  
int circle_radius = 123; // Defining the extern variable
```

```
// File main.cpp:  
  
#include "my_globals.h"  
#include <iostream>  
  
int main()  
{  
    std::cout << "The radius is: " << circle_radius << "\n";  
    return 0;  
}
```

Output:

```
The radius is: 123
```

Section 117.2: Simple block scope

The scope of a variable in a block { ... }, begins after declaration and ends at the end of the block. If there is nested block, the inner block can hide the scope of a variable which is declared in the outer block.

```
{  
    int x = 100;  
    // ^  
    // Scope of `x` begins here  
    //  
} // <- Scope of `x` ends here
```

If a nested block starts within an outer block, a new declared variable with the same name which is before in the

外层类中声明的同名新变量会隐藏第一个变量。

```
{  
    int x = 100;  
  
    {  
        int x = 200;  
  
        std::cout << x; // <- 输出是 200  
    }  
  
    std::cout << x; // <- 输出是 100  
}
```

outer class, hides the first one.

```
{  
    int x = 100;  
  
    {  
        int x = 200;  
  
        std::cout << x; // <- Output is 200  
    }  
  
    std::cout << x; // <- Output is 100  
}
```

第118章 : static_assert

参数	详情
<code>bool constexpr</code>	要检查的表达式
消息	当 <code>bool constexpr</code> 为 <code>false</code> 时打印的消息

第118.1节 : static_assert

断言意味着应该检查一个条件，如果为假，则表示错误。对于 `static_assert()`，这在编译时完成。

```
template<typename T>
T mul10(const T t)
{
    static_assert( std::is_integral<T>::value, "mul10() 仅适用于整型类型" );
    return (t << 3) + (t << 1);
}
```

A `static_assert()` 有一个必需的第一个参数，即条件，必须是一个 `bool constexpr`。它可能有第二个参数，消息，是一个字符串字面量。从 C++17 开始，第二个参数是可选的；之前是必需的。

版本 \geq C++17

```
template<typename T>
T mul10(const T t)
{
    static_assert(std::is_integral<T>::value);
    return (t << 3) + (t << 1);
}
```

它用于以下情况：

- 通常，需要在编译时对某些类型或 `constexpr` 值进行验证
- 模板函数需要验证传入类型的某些属性
- 想要为以下内容编写测试用例：
 - 模板元函数
 - `constexpr` 函数
 - 宏元编程
- 需要某些宏定义（例如，C++ 版本）
- 移植遗留代码，对 `sizeof(T)`（例如，32位整数）的断言
- 程序运行需要某些编译器特性（打包、空基类优化等）

注意 `static_assert()` 不参与 SFINAE：因此，当可能存在额外的重载/特化时，不应使用它来替代模板元编程技术（如 `std::enable_if<>`）。它可以在模板代码中使用，当预期的重载/特化已经找到，但需要进一步验证时。在这种情况下，它可能比依赖 SFINAE 提供更具体的错误信息。

Chapter 118: static_assert

Parameter	Details
<code>bool constexpr</code>	Expression to check
<code>message</code>	Message to print when <code>bool constexpr</code> is <code>false</code>

Section 118.1: static_assert

Assertions mean that a condition should be checked and if it's false, it's an error. For `static_assert()`, this is done compile-time.

```
template<typename T>
T mul10(const T t)
{
    static_assert( std::is_integral<T>::value, "mul10() only works for integral types" );
    return (t << 3) + (t << 1);
}
```

A `static_assert()` has a mandatory first parameter, the condition, that is a `bool constexpr`. It *might* have a second parameter, the message, that is a string literal. From C++17, the second parameter is optional; before that, it's mandatory.

Version \geq C++17

```
template<typename T>
T mul10(const T t)
{
    static_assert(std::is_integral<T>::value);
    return (t << 3) + (t << 1);
}
```

It is used when:

- In general, a verification at compile-time is required on some type or `constexpr` value
- A template function needs to verify certain properties of a type passed to it
- One wants to write test cases for:
 - template metafunctions
 - `constexpr` functions
 - macro metaprogramming
- Certain defines are required (for ex., C++ version)
- Porting legacy code, assertions on `sizeof(T)` (e.g., 32-bit int)
- Certain compiler features are required for the program to work (packing, empty base class optimization, etc.)

Note that `static_assert()` does not participate in SFINAE: thus, when additional overloads / specializations are possible, one should not use it instead of template metaprogramming techniques (like `std::enable_if<>`). It might be used in template code when the expected overload / specialization is already found, but further verifications are required. In such cases, it might provide more concrete error message(s) than relying on SFINAE for this.

第119章：constexpr

`constexpr` 是一个关键字，可用于将变量的值标记为常量表达式，将函数标记为可能用于常量表达式，或者（自 C++17 起）将 if 语句标记为仅编译其一个分支。

第119.1节：constexpr变量

声明为 `constexpr` 的变量隐式为 `const`，其值可用作常量表达式。

与#define的比较

`constexpr` 是基于类型安全的替代`#define`的编译时表达式。使用 `constexpr` 时，编译时求值的表达式会被结果替换。例如：

版本 ≥ C++11

```
int main()
{
    constexpr int N = 10 + 2;
    cout << N;
}
```

将生成以下代码：

```
cout << 12;
```

基于预处理器的编译时宏会有所不同。考虑：

```
#define N 10 + 2

int main()
{
    cout << N;
}
```

将产生：

```
cout << 10 + 2;
```

这显然会被转换为 `cout << 10 + 2;`。然而，编译器需要做更多的工作。此外，如果使用不当，会产生问题。

例如（使用`#define`）：

```
cout << N * 2;
```

形成：

```
cout << 10 + 2 * 2; // 14
```

但是预先计算的`constexpr`会正确地给出24。

与`const`的比较

Chapter 119: constexpr

`constexpr` is a keyword that can be used to mark a variable's value as a constant expression, a function as potentially usable in constant expressions, or (since C++17) an if statement as having only one of its branches selected to be compiled.

Section 119.1: constexpr variables

A variable declared `constexpr` is implicitly `const` and its value may be used as a constant expression.

Comparison with `#define`

A `constexpr` is type-safe replacement for `#define` based compile-time expressions. With `constexpr` the compile-time evaluated expression is replaced with the result. For example:

Version ≥ C++11

```
int main()
{
    constexpr int N = 10 + 2;
    cout << N;
}
```

will produce the following code:

```
cout << 12;
```

A pre-processor based compile-time macro would be different. Consider:

```
#define N 10 + 2

int main()
{
    cout << N;
}
```

will produce:

```
cout << 10 + 2;
```

which will obviously be converted to `cout << 10 + 2;`. However, the compiler would have to do more work. Also, it creates a problem if not used correctly.

For example (with `#define`):

```
cout << N * 2;
```

forms:

```
cout << 10 + 2 * 2; // 14
```

But a pre-evaluated `constexpr` would correctly give 24.

Comparison with `const`

一个const变量是一个需要内存存储的变量。一个constexpr则不需要。一个constexpr产生编译时常量，且不可更改。你可能会说const也不可更改。但请考虑：

```
int main()
{
    const int size1 = 10;
    const int size2 = abs(10);

    int arr_one[size1];
    int arr_two[size2];
}
```

在大多数编译器中，第二条语句会失败（例如GCC可能可以）。如你所知，任何数组的大小必须是常量表达式（即编译时值）。第二个变量size2被赋予了一个在运行时决定的值（即使你知道它是10，但对编译器来说它不是编译时常量）。

这意味着const可能是也可能不是一个真正的编译时常量。你无法保证或强制某个特定的const值绝对是编译时常量。你可以使用#define，但它也有自己的缺陷。

因此，简单地使用：

```
版本 ≥ C++11
int main()
{
    constexpr int size = 10;

    int arr[size];
}
```

一个constexpr表达式必须在编译时求值。因此，不能使用：

```
版本 ≥ C++11
constexpr int size = abs(10);
```

除非函数(abs)本身返回一个constexpr值。

所有基本类型都可以用constexpr初始化。

```
版本 ≥ C++11
constexpr bool FailFatal = true;
constexpr float PI = 3.14f;
constexpr char* site= "StackOverflow";
```

有趣且方便的是，你也可以使用auto：

```
版本 ≥ C++11
constexpr auto domain = ".COM"; // const char * const domain = ".COM"
constexpr auto PI = 3.14; // constexpr double
```

第119.2节：静态if语句

版本 ≥ C++17

if constexpr语句可用于有条件地编译代码。条件必须是常量表达式。未被选择的分支将被丢弃。模板中的丢弃语句不会被实例化。对于

A `const` variable is a **variable** which needs memory for its storage. A `constexpr` does not. A `constexpr` produces compile time constant, which cannot be changed. You may argue that `const` may also not be changed. But consider:

```
int main()
{
    const int size1 = 10;
    const int size2 = abs(10);

    int arr_one[size1];
    int arr_two[size2];
}
```

With most compilers the second statement will fail (may work with GCC, for example). The size of any array, as you might know, has to be a constant expression (i.e. results in compile-time value). The second variable `size2` is assigned some value that is decided at runtime (even though you know it is 10, for the compiler it is not compile-time).

This means that a `const` may or may not be a true compile-time constant. You cannot guarantee or enforce that a particular `const` value is absolutely compile-time. You may use `#define` but it has its own pitfalls.

Therefore simply use:

```
Version ≥ C++11
int main()
{
    constexpr int size = 10;

    int arr[size];
}
```

A `constexpr` expression must evaluate to a compile-time value. Thus, you cannot use:

```
Version ≥ C++11
constexpr int size = abs(10);
```

Unless the function (`abs`) is itself returning a `constexpr`.

All basic types can be initialized with `constexpr`.

```
Version ≥ C++11
constexpr bool FailFatal = true;
constexpr float PI = 3.14f;
constexpr char* site= "StackOverflow";
```

Interestingly, and conveniently, you may also use `auto`:

```
Version ≥ C++11
constexpr auto domain = ".COM"; // const char * const domain = ".COM"
constexpr auto PI = 3.14; // constexpr double
```

Section 119.2: Static if statement

Version ≥ C++17

The `if constexpr` statement can be used to conditionally compile code. The condition must be a constant expression. The branch not selected is *discarded*. A discarded statement inside a template is not instantiated. For

示例：

```
template<class T, class ... Rest>
void g(T &&p, Rest &&...rs)
{
    // ... 处理 p
    if constexpr (sizeof...(rs) > 0)
        g(rs...); // 从不以空参数列表实例化
}
```

此外，仅在被丢弃语句中 odr 使用的变量和函数不要求被定义，且被丢弃的 return 语句不用于函数返回类型推导。

if constexpr 与 #ifdef 不同。#ifdef 有条件地编译代码，但仅基于预处理时可求值的条件。例如，#ifdef 不能用于根据模板参数的值有条件地编译代码。另一方面，if constexpr 不能用于丢弃语法上无效的代码，而 #ifdef 可以。

```
if constexpr(false) {
    foobar; // 错误；foobar 未声明
    std::vector<int> v("hello, world"); // 错误；无匹配的构造函数
}
```

第119.3节：constexpr 函数

声明为 constexpr 的函数隐式为内联函数，对此类函数的调用可能产生常量表达式。例如，以下函数如果以常量表达式参数调用，也会产生常量表达式：

```
版本 ≥ C++11
constexpr int Sum(int a, int b)
{
    return a + b;
}
```

因此，函数调用的结果可以用作数组边界或模板参数，或者用于初始化一个 constexpr 变量：

```
版本 ≥ C++11
int main()
{
    constexpr int S = Sum(10, 20);

    int Array[S];
    int Array2[Sum(20, 30)]; // 50 数组大小，编译时确定
}
```

注意，如果你从函数的返回类型说明中去掉 constexpr，赋值给 S 将无法工作，因为 S 是一个 constexpr 变量，必须被赋值为编译时常量。同样，如果函数 Sum 不是 constexpr，数组的大小也不会是常量表达式。

关于 constexpr 函数的有趣之处在于，你也可以像使用普通函数一样使用它：

```
版本 ≥ C++11
int a = 20;
auto sum = Sum(a, abs(-20));
```

example:

```
template<class T, class ... Rest>
void g(T &&p, Rest &&...rs)
{
    // ... handle p
    if constexpr (sizeof...(rs) > 0)
        g(rs...); // never instantiated with an empty argument list
}
```

In addition, variables and functions that are odr-used only inside discarded statements are not required to be defined, and discarded return statements are not used for function return type deduction.

if constexpr 是 distinct from #ifdef. #ifdef 有条件地编译代码，但仅基于预处理时可求值的条件。例如，#ifdef 不能用于根据模板参数的值有条件地编译代码。另一方面，if constexpr 不能用于丢弃语法上无效的代码，而 #ifdef 可以。

```
if constexpr(false) {
    foobar; // error; foobar has not been declared
    std::vector<int> v("hello, world"); // error; no matching constructor
}
```

Section 119.3: constexpr functions

A function that is declared constexpr is implicitly inline and calls to such a function potentially yield constant expressions. For example, the following function, if called with constant expression arguments, yields a constant expression too:

```
Version ≥ C++11
constexpr int Sum(int a, int b)
{
    return a + b;
}
```

Thus, the result of the function call may be used as an array bound or a template argument, or to initialize a constexpr variable:

```
Version ≥ C++11
int main()
{
    constexpr int S = Sum(10, 20);

    int Array[S];
    int Array2[Sum(20, 30)]; // 50 array size, compile time
}
```

Note that if you remove constexpr from function's return type specification, assignment to S will not work, as S is a constexpr variable, and must be assigned a compile-time const. Similarly, size of array will also not be a constant-expression, if function Sum is not constexpr.

Interesting thing about constexpr functions is that you may also use it like ordinary functions:

```
Version ≥ C++11
int a = 20;
auto sum = Sum(a, abs(-20));
```

Sum现在将不再是`constexpr`函数，它将被编译为普通函数，接受变量（非常量）参数，并返回非常量值。你不需要写两个函数。

这也意味着如果你尝试将这样的调用赋值给非常量变量，代码将无法编译：

版本 ≥ C++11

```
int a = 20;
constexpr auto sum = Sum(a, abs(-20));
```

原因很简单：`constexpr`只能被赋值为编译时常量。然而，上述函数调用使得Sum变成了非`constexpr`（右值是非常量，但左值声明自身为`constexpr`）。

`constexpr`函数必须也返回编译时常量。以下代码将无法编译：

版本 ≥ C++11

```
constexpr int Sum(int a, int b)
{
    int a1 = a;      // 错误
    return a + b;
}
```

因为 `a1` 是非`constexpr`变量，阻止了函数成为真正的`constexpr`函数。将其改为`constexpr`并赋值为 `a` 也不行——因为 `a`（传入参数）的值仍然未知：

版本 ≥ C++11

```
constexpr int Sum(int a, int b)
{
    constexpr int a1 = a;      // 错误
    ...
}
```

此外，以下代码也无法编译：

版本 ≥ C++11

```
constexpr int Sum(int a, int b)
{
    return abs(a) + b; // 或 abs(a) + abs(b)
}
```

由于 `abs(a)` 不是一个常量表达式（即使是 `abs(10)` 也不行，因为 `abs` 并没有返回一个 `constexpr int`！

这怎么样？

版本 ≥ C++11

```
constexpr int Abs(int v)
{
    return v >= 0 ? v : -v;
}

constexpr int Sum(int a, int b)
{
    return Abs(a) + b;
}
```

我们自定义了一个Abs函数，它是一个`constexpr`，且Abs的函数体也没有违反任何规则。此外，在调用处（在Sum内部），表达式被求值为一个`constexpr`。因此，调用`Sum(-10, 20)`将是一个编译时常量表达式，结果为30。

Sum will not be a `constexpr` function now, it will be compiled as an ordinary function, taking variable (non-constant) arguments, and returning non-constant value. You need not to write two functions.

It also means that if you try to assign such call to a non-const variable, it won't compile:

Version ≥ C++11

```
int a = 20;
constexpr auto sum = Sum(a, abs(-20));
```

The reason is simple: `constexpr` must only be assigned a compile-time constant. However, the above function call makes Sum a non-`constexpr` (R-value is non-const, but L-value is declaring itself to be `constexpr`).

The `constexpr` function **must** also return a compile-time constant. Following will not compile:

Version ≥ C++11

```
constexpr int Sum(int a, int b)
{
    int a1 = a;      // ERROR
    return a + b;
}
```

Because `a1` is a non-`constexpr` variable, and prohibits the function from being a true `constexpr` function. Making it `constexpr` and assigning it a will also not work - since value of a (incoming parameter) is still not yet known:

Version ≥ C++11

```
constexpr int Sum(int a, int b)
{
    constexpr int a1 = a;      // ERROR
    ...
}
```

Furthermore, following will also not compile:

Version ≥ C++11

```
constexpr int Sum(int a, int b)
{
    return abs(a) + b; // or abs(a) + abs(b)
}
```

Since `abs(a)` is not a constant expression (even `abs(10)` will not work, since `abs` is not returning a `constexpr int`！

What about this?

Version ≥ C++11

```
constexpr int Abs(int v)
{
    return v >= 0 ? v : -v;
}

constexpr int Sum(int a, int b)
{
    return Abs(a) + b;
}
```

We crafted our own Abs function which is a `constexpr`, and the body of Abs also doesn't break any rule. Also, at the call site (inside Sum), the expression evaluates to a `constexpr`. Hence, the call to `Sum(-10, 20)` will be a compile-time constant expression resulting to 30.

第120章：一义性规则 (ODR)

第120.1节：通过重载解析违反ODR

即使内联函数的标记完全相同，如果名称查找不指向同一实体，也可能违反ODR。让我们考虑以下中的func：

- header.h

```
void 重载(int);
inline void 函数() { 重载('*'); }
```

- foo.cpp

```
#include "header.h"

void foo()
{
    函数(); // `重载` 指的是 `void 重载(int)`
}
```

- bar.cpp

```
void 重载(char); // 可能来自其他包含文件
#include "header.h"

void bar()
{
    函数(); // `重载` 指的是 `void 重载(char)`
}
```

我们有一个ODR违规，因为重载根据翻译单元的不同指向不同的实体。

第120.2节：多重定义函数

一体化定义规则 (One Definition Rule) 最重要的结果是，具有外部链接的非内联函数在程序中应只定义一次，尽管它们可以被多次声明。因此，这类函数不应定义在头文件中，因为头文件可能会被不同的翻译单元多次包含。

foo.h:

```
#ifndef FOO_H
#define FOO_H
#include <iostream>
void foo() { std::cout << "foo"; }
void bar();
#endif
```

foo.cpp:

```
#include "foo.h"
void bar() { std::cout << "bar"; }
```

main.cpp:

Chapter 120: One Definition Rule (ODR)

Section 120.1: ODR violation via overload resolution

Even with identical tokens for inline functions, ODR can be violated if lookup of names doesn't refer to the same entity. let's consider func in following:

- header.h

```
void overloaded(int);
inline void func() { overloaded('*'); }
```

- foo.cpp

```
#include "header.h"

void foo()
{
    func(); // `overloaded` refers to `void overloaded(int)`
}
```

- bar.cpp

```
void overloaded(char); // can come from other include
#include "header.h"

void bar()
{
    func(); // `overloaded` refers to `void overloaded(char)`
}
```

We have an ODR violation as overloaded refers to different entities depending of the translation unit.

Section 120.2: Multiply defined function

The most important consequence of the One Definition Rule is that non-inline functions with external linkage should only be defined once in a program, although they can be declared multiple times. Therefore, such functions should not be defined in headers, since a header can be included multiple times from different translation units.

foo.h:

```
#ifndef FOO_H
#define FOO_H
#include <iostream>
void foo() { std::cout << "foo"; }
void bar();
#endif
```

foo.cpp:

```
#include "foo.h"
void bar() { std::cout << "bar"; }
```

main.cpp:

```
#include "foo.h"
int main() {
    foo();
    bar();
}
```

在这个程序中，函数foo定义在头文件foo.h中，该头文件被包含了两次：一次来自foo.cpp，另一次来自main.cpp。因此，每个翻译单元都包含了foo的定义。请注意，foo.h中的包含保护并不能阻止这种情况发生，因为foo.cpp和main.cpp都分别包含了foo.h。尝试构建该程序最可能的结果是链接时错误，指出foo被多重定义。

为了避免此类错误，应当在头文件中声明函数，并在对应的.cpp文件中定义它们，但有一些例外（参见其他示例）。

第120.3节：内联函数

声明为inline的函数可以在多个翻译单元中定义，前提是所有定义都完全相同。并且它必须在每个使用它的翻译单元中定义。因此，内联函数应当定义在头文件中，且无需在实现文件中声明。

程序的行为将如同该函数只有一个定义。

foo.h:

```
#ifndef FOO_H
#define FOO_H
#include <iostream>
inline void foo() { std::cout << "foo"; }
void bar()
#endif
```

foo.cpp:

```
#include "foo.h"
void bar() {
    // 更复杂的定义
}
```

main.cpp:

```
#include "foo.h"
int main() {
    foo();
    bar();
}
```

在此示例中，更简单的函数foo在头文件中以内联方式定义，而更复杂的函数bar不是内联的，定义在实现文件中。foo.cpp和main.cpp两个翻译单元都包含了foo的定义，但由于foo是内联函数，该程序是合法的。

在类定义内定义的函数（可以是成员函数或友元函数）是隐式内联的。因此，如果类定义在头文件中，类的成员函数可以在类定义内定义，即使这些定义可能被包含在多个翻译单元中：

```
// 在 foo.h 中
class Foo {
```

```
#include "foo.h"
int main() {
    foo();
    bar();
}
```

In this program, the function foo is defined in the header foo.h, which is included twice: once from foo.cpp and once from main.cpp. Each translation unit therefore contains its own definition of foo. Note that the include guards in foo.h do not prevent this from happening, since foo.cpp and main.cpp both separately include foo.h. The most likely result of trying to build this program is a link-time error identifying foo as having been multiply defined.

To avoid such errors, one should declare functions in headers and define them in the corresponding .cpp files, with some exceptions (see other examples).

Section 120.3: Inline functions

A function declared `inline` may be defined in multiple translation units, provided that all definitions are identical. It also must be defined in every translation unit in which it is used. Therefore, inline functions *should* be defined in headers and there is no need to mention them in the implementation file.

The program will behave as though there is a single definition of the function.

foo.h:

```
#ifndef FOO_H
#define FOO_H
#include <iostream>
inline void foo() { std::cout << "foo"; }
void bar();
#endif
```

foo.cpp:

```
#include "foo.h"
void bar() {
    // more complicated definition
}
```

main.cpp:

```
#include "foo.h"
int main() {
    foo();
    bar();
}
```

In this example, the simpler function foo is defined inline in the header file while the more complicated function bar is not inline and is defined in the implementation file. Both the foo.cpp and main.cpp translation units contain definitions of foo, but this program is well-formed since foo is inline.

A function defined within a class definition (which may be a member function or a friend function) is *implicitly* inline. Therefore, if a class is defined in a header, member functions of the class may be defined within the class definition, even though the definitions may be included in multiple translation units:

```
// in foo.h
class Foo {
```

```
void bar() { std::cout << "bar"; }
void baz();
};
```

```
// 在 foo.cpp 中
void Foo::baz() {
    // 定义
}
```

函数Foo::baz是在类外定义的，因此它不是内联函数，且不得在头文件中定义。

```
void bar() { std::cout << "bar"; }
void baz();
};
```

```
// in foo.cpp
void Foo::baz() {
    // definition
}
```

The function `Foo::baz` is defined out-of-line, so it is *not* an inline function, and must not be defined in the header.

第121章：未指定行为

第121.1节：超出范围的枚举值

如果将作用域枚举转换为无法容纳其值的较小整型，则结果值为未指定。例如：

```
enum class E {
    X = 1,
    Y = 1000,
};

// 假设1000无法存入char类型
char c1 = static_cast<char>(E::X); // c1为1
char c2 = static_cast<char>(E::Y); // c2的值未指定
```

此外，如果将整数转换为枚举且该整数值超出枚举值范围，则结果值为未指定。例如：

```
enum Color {
    RED = 1,
    GREEN = 2,
    BLUE = 3,
};

Color c = static_cast<Color>(4);
```

然而，在下一个例子中，行为不是未指定的，因为源值在枚举的范围内，尽管它不等于任何枚举成员：

```
enum Scale {
    ONE = 1,
    TWO = 2,
    FOUR = 4,
};

Scale s = static_cast<Scale>(3);
```

这里 s 的值将是 3，并且不等于 ONE、TWO 和 FOUR。

第121.2节：函数参数的求值顺序

如果一个函数有多个参数，它们的求值顺序是不确定的。以下代码可能会打印 `x = 1, y = 2` 或 `x = 2, y = 1`，但具体哪种情况是不确定的。

```
int f(int x, int y) {printf(
    "x = %d, y = %d", x, y);}

int get_val() {
    static int x = 0;
    return ++x;
}

int main() {
    f(get_val(), get_val());
}
```

版本 ≥ C++17

在 C++17 中，函数参数的求值顺序仍然是不确定的。

Chapter 121: Unspecified behavior

Section 121.1: Value of an out-of-range enum

If a scoped enum is converted to an integral type that is too small to hold its value, the resulting value is unspecified. Example:

```
enum class E {
    X = 1,
    Y = 1000,
};

// assume 1000 does not fit into a char
char c1 = static_cast<char>(E::X); // c1 is 1
char c2 = static_cast<char>(E::Y); // c2 has an unspecified value
```

Also, if an integer is converted to an enum and the integer's value is outside the range of the enum's values, the resulting value is unspecified. Example:

```
enum Color {
    RED = 1,
    GREEN = 2,
    BLUE = 3,
};

Color c = static_cast<Color>(4);
```

However, in the next example, the behavior is *not* unspecified, since the source value is within the *range* of the enum, although it is unequal to all enumerators:

```
enum Scale {
    ONE = 1,
    TWO = 2,
    FOUR = 4,
};

Scale s = static_cast<Scale>(3);
```

Here s will have the value 3, and be unequal to ONE, TWO, and FOUR.

Section 121.2: Evaluation order of function arguments

If a function has multiple arguments, it is unspecified what order they are evaluated in. The following code could print `x = 1, y = 2` or `x = 2, y = 1` but it is unspecified which.

```
int f(int x, int y) {
    printf("x = %d, y = %d\n", x, y);
}

int get_val() {
    static int x = 0;
    return ++x;
}

int main() {
    f(get_val(), get_val());
}
```

Version ≥ C++17

In C++17, the order of evaluation of function arguments remains unspecified.

但是，每个函数参数都会被完全求值，并且调用对象保证在任何函数参数之前被求值。

```
结构体 from_int {  
from_int(int x) { std::cout << "from_int (" << x << ")"; };  
  
int make_int(int x){ std::cout << "make_int (" << x << ")"; return x; }  
  
void foo(from_int a, from_int b) {  
}  
void bar(from_int a, from_int b) {  
}  
  
auto which_func(bool b){ std::cout << b?"foo":"bar" << "";return b?foo:bar;  
}  
  
int main(int argc, char const*const* argv) {  
    which_func( true )( make_int(1), make_int(2) );  
}
```

这必须打印：

```
bar  
make_int(1)  
from_int(1)  
make_int(2)  
from_int(2)
```

或者

```
bar  
make_int(2)  
from_int(2)  
make_int(1)  
from_int(1)
```

它可能不会在任何make或from之后打印bar，并且它可能不会打印：

```
bar  
make_int(2)  
make_int(1)  
from_int(2)  
from_int(1)
```

或类似内容。在C++17之前，在make_int之后打印bar是合法的，且在执行任何

from_int之后执行所有make_int也是合法的。

第121.3节：某些reinterpret_cast转换的结果

从一种函数指针类型到另一种函数指针类型，或从一种函数引用类型到另一种函数引用类型的reinterpret_cast结果是未指定的。例如：

```
int f();
```

However, each function argument is completely evaluated, and the calling object is guaranteed evaluated before any function arguments are.

```
struct from_int {  
    from_int(int x) { std::cout << "from_int (" << x << ")\\n"; }  
};  
int make_int(int x){ std::cout << "make_int (" << x << ")\\n"; return x; }  
  
void foo(from_int a, from_int b) {  
}  
void bar(from_int a, from_int b) {  
}  
  
auto which_func(bool b){  
    std::cout << b?"foo":"bar" << "\\n";  
    return b?foo:bar;  
}  
  
int main(int argc, char const*const* argv) {  
    which_func( true )( make_int(1), make_int(2) );  
}
```

this must print:

```
bar  
make_int(1)  
from_int(1)  
make_int(2)  
from_int(2)
```

or

```
bar  
make_int(2)  
from_int(2)  
make_int(1)  
from_int(1)
```

it may not print bar after any of the make or from's, and it may not print:

```
bar  
make_int(2)  
make_int(1)  
from_int(2)  
from_int(1)
```

or similar. Prior to C++17 printing bar after make_ints was legal, as was doing both make_ints prior to doing any from_ints.

Section 121.3: Result of some reinterpret_cast conversions

The result of a `reinterpret_cast` from one function pointer type to another, or one function reference type to another, is unspecified. Example:

```
int f();
```

```
auto fp = reinterpret_cast<int*>(int)&f; // fp的值未指定  
版本 ≤ C++03
```

从一种对象指针类型到另一种对象指针类型，或从一种对象引用类型到另一种对象引用类型的`reinterpret_cast`结果是不确定的。示例：

```
int x = 42;  
char* p = reinterpret_cast<char*>(&x); // p 的值是不确定的
```

然而，在大多数编译器中，这等同于`static_cast<char*>(static_cast<void*>(&x))`，因此得到的指针p指向了x的第一个字节。这在C++11中被规定为标准行为。更多细节请参见类型惩罚转换（type punning conversion）。

第121.4节：引用所占空间

引用不是对象，与对象不同，它不保证占用连续的内存字节。
标准未规定引用是否需要任何存储空间。语言的多个特性使得无法以可移植的方式检查引用可能占用的任何存储空间：

- 如果对引用使用`sizeof`，返回的是被引用类型的大小，因此无法得知引用本身是否占用存储空间。
- 引用数组是非法的，因此无法通过检查假设的引用数组中两个连续元素的地址来确定引用的大小。
- 如果取引用的地址，结果是被引用对象的地址，因此无法获得指向引用本身的指针。
- 如果类中有引用成员，尝试使用`offsetof`提取该成员的地址会导致未定义行为，因为这样的类不是标准布局类。
- 如果一个类有引用成员，则该类不再是标准布局，因此尝试访问用于存储该引用的任何数据将导致未定义或未指定的行为。

实际上，在某些情况下，引用变量的实现可能类似于指针变量，因此占用与指针相同的存储空间，而在其他情况下，引用可能根本不占用空间，因为它可以被优化掉。例如，在：

```
void f() {  
    int x;  
    int& r = x;  
    // 对 r 进行某些操作  
}
```

编译器可以自由地将r视为x的别名，并将函数f中其余部分所有出现的r替换为x，且不为r分配任何存储空间。

第121.5节：大多数标准库类的移动后状态

版本 ≥ C++11

所有标准库容器在被移动后都处于“有效但未指定”状态。例如，在以下代码中，移动后v2将包含{1, 2, 3, 4}，但不保证v1为空。

```
int main() {  
    std::vector<int> v1{1, 2, 3, 4};  
    std::vector<int> v2 = std::move(v1);
```

```
auto fp = reinterpret_cast<int*>(int)&f; // fp has unspecified value  
Version ≤ C++03
```

The result of a `reinterpret_cast` from one object pointer type to another, or one object reference type to another, is unspecified. Example:

```
int x = 42;  
char* p = reinterpret_cast<char*>(&x); // p has unspecified value
```

However, with most compilers, this was equivalent to `static_cast<char*>(static_cast<void*>(&x))` so the resulting pointer p pointed to the first byte of x. This was made the standard behavior in C++11. See type punning conversion for more details.

Section 121.4: Space occupied by a reference

A reference is not an object, and unlike an object, it is not guaranteed to occupy some contiguous bytes of memory. The standard leaves it unspecified whether a reference requires any storage at all. A number of features of the language conspire to make it impossible to portably examine any storage the reference might occupy:

- If `sizeof` is applied to a reference, it returns the size of the referenced type, thereby giving no information about whether the reference occupies any storage.
- Arrays of references are illegal, so it is not possible to examine the addresses of two consecutive elements of a hypothetical reference of arrays in order to determine the size of a reference.
- If the address of a reference is taken, the result is the address of the referent, so we cannot get a pointer to the reference itself.
- If a class has a reference member, attempting to extract the address of that member using `-offsetof` yields undefined behavior since such a class is not a standard-layout class.
- If a class has a reference member, the class is no longer standard layout, so attempts to access any data used to store the reference results in undefined or unspecified behavior.

In practice, in some cases a reference variable may be implemented similarly to a pointer variable and hence occupy the same amount of storage as a pointer, while in other cases a reference may occupy no space at all since it can be optimized out. For example, in:

```
void f() {  
    int x;  
    int& r = x;  
    // do something with r  
}
```

the compiler is free to simply treat r as an alias for x and replace all occurrences of r in the rest of the function f with x, and not allocate any storage to hold r.

Section 121.5: Moved-from state of most standard library classes

Version ≥ C++11

All standard library containers are left in a *valid but unspecified* state after being moved from. For example, in the following code, v2 will contain {1, 2, 3, 4} after the move, but v1 is not guaranteed to be empty.

```
int main() {  
    std::vector<int> v1{1, 2, 3, 4};  
    std::vector<int> v2 = std::move(v1);
```

}

某些类确实具有明确定义的已移动状态。最重要的例子是
std::unique_ptr<T>, 移动后保证为null。

第121.6节：某些指针比较的结果

如果两个指针使用<、>、<=或>=进行比较，以下情况结果未指定：

- 指针指向不同的数组。（非数组对象视为大小为1的数组。）

```
int x;
int y;
const bool b1 = &x < &y;           // 未指定
int a[10];
const bool b2 = &a[0] < &a[1];    // true
const bool b3 = &a[0] < &x;      // 未指定
const bool b4 = (a + 9) < (a + 10); // true
                                // 注意：a+10 指向数组末尾之后的位置
```

- 指针指向同一对象，但指向不同访问控制的成员。

```
class A {
public:
    int x;
    int y;
    bool f1() { return &x < &y; } // true; x 在 y 之前
    bool f2() { return &x < &z; } // 未指定
private:
    int z;
};
```

第121.7节：从无效的void*值进行静态转换

如果将void*值转换为指向对象类型T*的指针，但该值未针对T正确对齐，则得到的指针值未指定。示例：

```
// 假设int的对齐方式为4
int x = 42;
void* p1 = &x;
// 进行一些指针运算...
void* p2 = static_cast<char*>(p1) + 2;
int* p3 = static_cast<int*>(p2);
```

p3的值未指定，因为p2不能指向类型为int的对象；其值不是一个正确对齐的地址。

第121.8节：跨翻译单元的全局变量初始化顺序

虽然在一个翻译单元内，全局变量的初始化顺序是指定的，但跨翻译单元的初始化顺序是未指定的。

因此，具有以下文件的程序

}

Some classes do have a precisely defined moved-from state. The most important case is that of std::unique_ptr<T>, which is guaranteed to be null after being moved from.

Section 121.6: Result of some pointer comparisons

If two pointers are compared using <, >, <=, or >=, the result is unspecified in the following cases:

- The pointers point into different arrays. (A non-array object is considered an array of size 1.)

```
int x;
int y;
const bool b1 = &x < &y;           // unspecified
int a[10];
const bool b2 = &a[0] < &a[1];    // true
const bool b3 = &a[0] < &x;      // unspecified
const bool b4 = (a + 9) < (a + 10); // true
                                // note: a+10 points past the end of the array
```

- The pointers point into the same object, but to members with different access control.

```
class A {
public:
    int x;
    int y;
    bool f1() { return &x < &y; } // true; x comes before y
    bool f2() { return &x < &z; } // unspecified
private:
    int z;
};
```

Section 121.7: Static cast from bogus void* value

If a void* value is converted to a pointer to object type, T*, but is not properly aligned for T, the resulting pointer value is unspecified. Example:

```
// Suppose that alignof(int) is 4
int x = 42;
void* p1 = &x;
// Do some pointer arithmetic...
void* p2 = static_cast<char*>(p1) + 2;
int* p3 = static_cast<int*>(p2);
```

The value of p3 is unspecified because p2 cannot point to an object of type int; its value is not a properly aligned address.

Section 121.8: Order of initialization of globals across TU

Whereas inside a Translation Unit, order of initialization of global variables is specified, order of initialization across Translation Units is unspecified.

So program with following files

- foo.cpp

```
#include <iostream>

int dummyFoo = ((std::cout << "foo"), 0);
```

- bar.cpp

```
#include <iostream>

int dummyBar = ((std::cout << "bar"), 0);
```

- main.cpp

```
int main() {}
```

可能产生的输出：

foobar

或者

barfoo

这可能导致静态初始化顺序灾难。

- foo.cpp

```
#include <iostream>

int dummyFoo = ((std::cout << "foo"), 0);
```

- bar.cpp

```
#include <iostream>

int dummyBar = ((std::cout << "bar"), 0);
```

- main.cpp

```
int main() {}
```

might produce as output:

foobar

or

barfoo

That may lead to *Static Initialization Order Fiasco*.

第122章：依赖参数的名称查找

第122.1节：找到哪些函数

函数的查找首先通过收集一组“关联类”和“关联命名空间”来完成，这些类和命名空间包含一个或多个以下内容，具体取决于参数类型T。首先，我们展示类、枚举和类模板特化名称的规则。

- 如果T是嵌套类、成员枚举，则为其所在的外部类。
- 如果T是枚举（它也可能是类成员！），则为其最内层的命名空间。
- 如果T是类（它也可能是嵌套的！），则包括其所有基类和类本身。所有关联类的最内层命名空间。
- 如果T是ClassTemplate<TemplateArguments>（这也是一个类！），则包括与模板类型参数关联的类和命名空间，任何模板参数的命名空间，以及任何模板参数的外部类（如果模板参数是成员模板）。

现在还有一些针对内置类型的规则。

- 如果T是指向U的指针或U的数组，则包括与U关联的类和命名空间。例如：`void (*fptr)(A); f(fptr);`，包含与`void(A)`关联的命名空间和类（见下一条规则）。
- 如果T是函数类型，则包括与参数和返回类型关联的类和命名空间。例如：`void(A)`将包括与A关联的命名空间和类。
- 如果T是成员指针，则包括与成员类型关联的类和命名空间（可能适用于成员函数指针和数据成员指针！）。例如：`B A::*p; void (A::*pf)(B); f(p); f(pf);` 包含与A、B、`void(B)`关联的命名空间和类（上述函数类型规则适用）。

所有相关命名空间中的所有函数和模板都通过参数依赖查找 (*argument dependent lookup*) 找到。此外，在关联类中声明的命名空间作用域友元函数也会被找到，这些函数通常是不可见的。然而，使用指令 (*using directives*) 会被忽略。

以下所有示例调用都是有效的，无需在调用中通过命名空间名称限定f。

```
命名空间 A {
    结构体 Z { };
    命名空间 I { void g(Z); }
    使用命名空间 I;

    结构体 X { 结构体 Y { }; 友元函数 void f(Y) { } ;
        void f(X p) { }
        void f(std::shared_ptr<X> p) { }
    }

    // 示例调用
    f(A::X());
    f(A::X::Y());
    f(std::make_shared<A::X>());
}

g(A::Z()); // 无效："using namespace I;" 被忽略!
```

Chapter 122: Argument Dependent Name Lookup

Section 122.1: What functions are found

Functions are found by first collecting a set of “associated classes” and “associated namespaces” that include one or more of the following, depending on the argument type T. First, let us show the rules for classes, enumeration and class template specialization names.

- If T is a nested class, member enumeration, then the surrounding class of it.
- If T is an enumeration (it may also be a class member!), the innermost namespace of it.
- If T is a class (it may also be nested!), all its base classes and the class itself. The innermost namespace of all associated classes.
- If T is a ClassTemplate<TemplateArguments> (this is also a class!), the classes and namespaces associated with the template type arguments, the namespace of any template template argument and the surrounding class of any template template argument, if a template argument is a member template.

Now there are a few rules for builtin types as well

- If T is a pointer to U or array of U, the classes and namespaces associated with U. Example: `void (*fptr)(A); f(fptr);`, includes the namespaces and classes associated with `void(A)` (see next rule).
- If T is a function type, the classes and namespaces associated with parameter and return types. Example: `void(A)` would include the namespaces and classes associated with A.
- If T is a pointer to member, the classes and namespaces associated with the member type (may apply to both pointer to member functions and pointer to data member!). Example: `B A::*p; void (A::*pf)(B); f(p); f(pf);` includes the namespaces and classes associated with A, B, `void(B)` (which applies bullet above for function types).

All functions and templates within all associated namespaces are found by argument dependent lookup. In addition, namespace-scope friend functions declared in associated classes are found, which are normally not visible. Using directives are ignored, however.

All of the following example calls are valid, without qualifying f by the namespace name in the call.

```
namespace A {
    struct Z { };
    namespace I { void g(Z); }
    using namespace I;

    struct X { struct Y { }; friend void f(Y) { } ;
        void f(X p) { }
        void f(std::shared_ptr<X> p) { }

    }

    // example calls
    f(A::X());
    f(A::X::Y());
    f(std::make_shared<A::X>());

    g(A::Z()); // invalid: "using namespace I;" is ignored!
```

第123章：属性

第123.1节：[[fallthrough]]

版本 ≥ C++17

每当一个case在switch中结束时，下一个case的代码将被执行。可以通过使用`break`语句来阻止这种情况。由于这种所谓的贯穿行为在非预期时可能引入错误，多个编译器和静态分析工具会对此发出警告。

从C++17开始，引入了一个标准属性，用于指示当代码意图贯穿时不需要警告。编译器可以安全地在一个case没有以break或[[fallthrough]]结束且至少有一条语句时发出警告。

```
switch(input) {
    case 2011:
    case 2014:
    case 2017:
        std::cout << "使用现代C++" << std::endl;
        [[fallthrough]]; // > 无警告
    case 1998:
    case 2003:
        standard = input;
}
```

请参阅[提案](#)，了解有关如何使用[[fallthrough]]的更详细示例。

第123.2节：[[nodiscard]]

版本 ≥ C++17

[[nodiscard]]属性可用于指示函数调用时不应忽略函数的返回值。如果忽略返回值，编译器应发出警告。该属性可以添加到：

- 函数定义
- 类型

将该属性添加到类型的行为与将该属性添加到返回此类型的每个函数相同。

```
template<typename Function>
[[nodiscard]]最终<std::decay_t<Function>> onExit(Function &&f);

void f(int &i) {
    assert(i == 0); // 仅为使注释清晰！
    ++i; // i == 1
    auto exit1 = onExit([&i]{ --i; }); // 在f()退出时减1
    ++i; // i == 2
    onExit([&i]{ --i; }); // 错误：直接减1
    // 预期编译器警告
    std::cout << i << std::endl; // 预期：2，实际：1
}
```

请参阅[提案](#)，了解有关如何使用[[nodiscard]]的更详细示例。

Chapter 123: Attributes

Section 123.1: [[fallthrough]]

Version ≥ C++17

Whenever a `case` is ended in a `switch`, the code of the next case will get executed. This last one can be prevented by using the `'break'` statement. As this so-called fallthrough behavior can introduce bugs when not intended, several compilers and static analyzers give a warning on this.

From C++17 on, a standard attribute was introduced to indicate that the warning is not needed when the code is meant to fall through. Compilers can safely give warnings when a case is ended without `break` or `[[fallthrough]]` and has at least one statement.

```
switch(input) {
    case 2011:
    case 2014:
    case 2017:
        std::cout << "Using modern C++" << std::endl;
        [[fallthrough]]; // > No warning
    case 1998:
    case 2003:
        standard = input;
}
```

See [the proposal](#) for more detailed examples on how `[[fallthrough]]` can be used.

Section 123.2: [[nodiscard]]

Version ≥ C++17

The `[[nodiscard]]` attribute can be used to indicate that the return value of a function shouldn't be ignored when you do a function call. If the return value is ignored, the compiler should give a warning on this. The attribute can be added to:

- A function definition
- A type

Adding the attribute to a type has the same behaviour as adding the attribute to every single function which returns this type.

```
template<typename Function>
[[nodiscard]] Finally<std::decay_t<Function>> onExit(Function &&f);

void f(int &i) {
    assert(i == 0); // Just to make comments clear!
    ++i; // i == 1
    auto exit1 = onExit([&i]{ --i; }); // Reduce by 1 on exiting f()
    ++i; // i == 2
    onExit([&i]{ --i; }); // BUG: Reducing by 1 directly
    // Compiler warning expected
    std::cout << i << std::endl; // Expected: 2, Real: 1
}
```

See [the proposal](#) for more detailed examples on how `[[nodiscard]]` can be used.

注意：示例中省略了 Finally/onExit 的实现细节，详见 Finally/ScopeExit。

第123.3节：[[deprecated]] 和 [[deprecated("reason")]]

版本 ≥ C++14

C++14 引入了通过属性弃用函数的标准方式。[[deprecated]] 可用于表示函数已被弃用。[[deprecated("reason")]] 允许添加具体原因，编译器可显示该原因。

```
void function(std::unique_ptr<A> &&a);

// 提供具体信息，帮助其他程序员修正代码
[[deprecated("请改用带 unique_ptr 的版本，此函数将在下一个
版本中移除")]]
void function(std::auto_ptr<A> a);

// 无信息，调用时会产生通用警告。
[[deprecated]]
void function(A *a);
```

该属性可应用于：

- 类的声明
- typedef 名称
- 一个变量
- 非静态数据成员
- 函数
- 枚举
- 模板特化

(参考 [C++14 标准草案：7.6.5 弃用属性](#))

第 123.4 节：[[maybe_unused]]

[[maybe_unused]] 属性用于在代码中指示某些逻辑可能未被使用。这通常与预处理器条件相关，可能会使用也可能不会使用。由于编译器可能会对未使用的变量发出警告，这是一种通过表明意图来抑制警告的方法。

一个典型的例子是调试版本中需要而生产版本中不需要的变量，比如表示成功的返回值。在调试版本中，应对条件进行断言，而在生产版本中这些断言已被移除。

```
[[maybe_unused]] auto mapInsertResult = configuration.emplace("LicenseInfo",
stringifiedLicenseInfo);
assert(mapInsertResult.second); // 我们只在启动期间调用，所以不可能已存在于映射中
```

一个更复杂的例子是位于匿名命名空间中的各种辅助函数。如果这些函数在编译期间未被使用，编译器可能会发出警告。理想情况下，你希望用与调用者相同的预处理器标签来保护它们，但由于这可能变得复杂，[[maybe_unused]] 属性是一个更易维护的替代方案。

```
namespace {
[[maybe_unused]] std::string createWindowsConfigFilePath(const std::string &relativePath);
```

Note: The implementation details of Finally/onExit are omitted in the example, see Finally/ScopeExit.

Section 123.3: [[deprecated]] and [[deprecated("reason")]]

Version ≥ C++14

C++14 introduced a standard way of deprecating functions via attributes. [[deprecated]] can be used to indicate that a function is deprecated. [[deprecated("reason")]] allows adding a specific reason which can be shown by the compiler.

```
void function(std::unique_ptr<A> &&a);

// Provides specific message which helps other programmers fixing there code
[[deprecated("Use the variant with unique_ptr instead, this function will be removed in the next
release")]]
void function(std::auto_ptr<A> a);

// No message, will result in generic warning if called.
[[deprecated]]
void function(A *a);
```

This attribute may be applied to:

- the declaration of a class
- a typedef-name
- a variable
- a non-static data member
- a function
- an enumeration
- a template specialization

(ref. [c++14 standard draft: 7.6.5 Deprecated attribute](#))

Section 123.4: [[maybe_unused]]

The [[maybe_unused]] attribute is created for indicating in code that certain logic might not be used. This is often linked to preprocessor conditions where this might be used or might not be used. As compilers can give warnings on unused variables, this is a way of suppressing them by indicating intent.

A typical example of variables which are needed in debug builds while unneeded in production are return values indicating success. In the debug builds, the condition should be asserted, though in production these asserts have been removed.

```
[[maybe_unused]] auto mapInsertResult = configuration.emplace("LicenseInfo",
stringifiedLicenseInfo);
assert(mapInsertResult.second); // We only get called during startup, so we can't be in the map
```

A more complex example are different kind of helper functions which are in an unnamed namespace. If these functions aren't used during compilation, a compiler might give a warning on them. Ideally you would like to guard them with the same preprocessor tags as the caller, though as this might become complex the [[maybe_unused]] attribute is a more maintainable alternative.

```
namespace {
[[maybe_unused]] std::string createWindowsConfigFilePath(const std::string &relativePath);
```

```
// TODO: 在 BSD、MAC 等系统上重用此代码
[[maybe_unused]] std::string createLinuxConfigFilePath(const std::string &relativePath);
}

std::string createConfigFilePath(const std::string &relativePath) {
#ifndef OS == "WINDOWS"
    return createWindowsConfigFilePath(relativePath);
#elif OS == "LINUX"
    return createLinuxConfigFilePath(relativePath);
#else
#error "操作系统尚不支持"
#endif
}
```

有关如何使用`[[maybe_unused]]`的更详细示例，请参见提案。

第123.5节：[[noreturn]]

版本 ≥ C++11

C++11 引入了`[[noreturn]]`属性。它可用于函数，表示该函数不会通过执行`return`语句或到达函数体末尾而返回调用者（需要注意的是，这不适用于`void`函数，因为它们确实会返回给调用者，只是没有返回值）。这样的函数可能通过调用`std::terminate`或`std::exit`，或者抛出异常来结束。还值得注意的是，这样的函数可以通过执行`longjmp`来返回。

例如，下面的函数要么总是抛出异常，要么调用`std::terminate`，因此它是`[[noreturn]]`的良好候选：

```
[[noreturn]] void ownAssertFailureHandler(std::string message) {
    std::cerr << message << std::endl;
    if (THROW_EXCEPTION_ON_ASSERT)
        throw AssertException(std::move(message));
    std::terminate();
}
```

这种功能允许编译器在知道代码永远不会被执行的情况下结束一个没有返回语句的函数。这里，因为下面代码中调用的`ownAssertFailureHandler`（如上所定义）永远不会返回，编译器不需要在该调用之后添加代码：

```
std::vector<int> createSequence(int end) {
    if (end > 0) {
        std::vector<int> sequence;
        sequence.reserve(end+1);
        for (int i = 0; i <= end; ++i)
            sequence.push_back(i);
        return sequence;
    }
    ownAssertFailureHandler("传递给 createSequence() 的数字为负数"s);
    // return std::vector<int>{}; //< 由于 [[noreturn]], 不需要返回
}
```

如果函数实际上会返回，则行为未定义，因此以下写法是不允许的：

```
[[noreturn]] void assertPositive(int number) {
    if (number >= 0)
        return;
    else
```

```
// TODO: Reuse this on BSD, MAC ...
[[maybe_unused]] std::string createLinuxConfigFilePath(const std::string &relativePath);
}

std::string createConfigFilePath(const std::string &relativePath) {
#ifndef OS == "WINDOWS"
    return createWindowsConfigFilePath(relativePath);
#elif OS == "LINUX"
    return createLinuxConfigFilePath(relativePath);
#else
#error "OS is not yet supported"
#endif
}
```

See [the proposal](#) for more detailed examples on how `[[maybe_unused]]` can be used.

Section 123.5: [[noreturn]]

Version ≥ C++11

C++11 introduced the `[[noreturn]]` attribute. It can be used for a function to indicate that the function does not return to the caller by either executing a `return` statement, or by reaching the end if its body (it is important to note that this does not apply to `void` functions, since they do return to the caller, they just do not return any value). Such a function may end by calling `std::terminate` or `std::exit`, or by throwing an exception. It is also worth noting that such a function can return by executing `longjmp`.

For instance, the function below will always either throw an exception or call `std::terminate`, so it is a good candidate for `[[noreturn]]`:

```
[[noreturn]] void ownAssertFailureHandler(std::string message) {
    std::cerr << message << std::endl;
    if (THROW_EXCEPTION_ON_ASSERT)
        throw AssertException(std::move(message));
    std::terminate();
}
```

This kind of functionality allows the compiler to end a function without a `return` statement if it knows the code will never be executed. Here, because the call to `ownAssertFailureHandler` (defined above) in the code below will never return, the compiler does not need to add code below that call:

```
std::vector<int> createSequence(int end) {
    if (end > 0) {
        std::vector<int> sequence;
        sequence.reserve(end+1);
        for (int i = 0; i <= end; ++i)
            sequence.push_back(i);
        return sequence;
    }
    ownAssertFailureHandler("Negative number passed to createSequence()"s);
    // return std::vector<int>{}; //< Not needed because of [[noreturn]]
}
```

It is undefined behavior if the function will actually return, so the following is not allowed:

```
[[noreturn]] void assertPositive(int number) {
    if (number >= 0)
        return;
    else
```

```
ownAssertFailureHandler("预期为正数"s); //< [[noreturn]]  
}
```

请注意，`[[noreturn]]`主要用于void函数。然而，这并非强制要求，允许这些函数用于泛型编程：

```
template<class 不一致处理器>  
double fortyTwoDivideBy(int i) {  
    if (i == 0)  
        i = 不一致处理器::correct(i);  
    return 42. / i;  
}  
  
struct 不一致抛出器 {  
    static [[noreturn]] int correct(int i) { ownAssertFailureHandler("未知不一致"s); }  
}  
  
struct 不一致改为一 {  
    static int correct(int i) { return 1; }  
}  
  
double fortyTwo = fortyTwoDivideBy<不一致改为一>(0);  
double unreachable = fortyTwoDivideBy<不一致抛出器>(0);
```

以下标准库函数具有此属性：

- std::abort
- std::exit
- std::quick_exit
- std::unexpected
- std::terminate
- std::rethrow_exception
- std::throw_with_nested
- std::nested_exception::rethrow_nested

```
ownAssertFailureHandler("Positive number expected"s); //< [[noreturn]]  
}
```

Note that the `[[noreturn]]` is mostly used in void functions. However, this is not a requirement, allowing the functions to be used in generic programming:

```
template<class InconsistencyHandler>  
double fortyTwoDivideBy(int i) {  
    if (i == 0)  
        i = InconsistencyHandler::correct(i);  
    return 42. / i;  
}  
  
struct InconsistencyThrower {  
    static [[noreturn]] int correct(int i) { ownAssertFailureHandler("Unknown inconsistency"s); }  
}  
  
struct InconsistencyChangeToOne {  
    static int correct(int i) { return 1; }  
}  
  
double fortyTwo = fortyTwoDivideBy<InconsistencyChangeToOne>(0);  
double unreachable = fortyTwoDivideBy<InconsistencyThrower>(0);
```

The following standard library functions have this attribute:

- std::abort
- std::exit
- std::quick_exit
- std::unexpected
- std::terminate
- std::rethrow_exception
- std::throw_with_nested
- std::nested_exception::rethrow_nested

第124章：C++中的递归

第124.1节：使用尾递归和斐波那契式递归来求解斐波那契数列

使用递归获取斐波那契数列第N项的最简单且最明显的方法是这个

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

但是，这个算法无法扩展到更高的项：随着 n 变得越来越大，所需的函数调用次数呈指数增长。这个可以用简单的尾递归来替代。

```
int get_term_fib(int n, int prev = 0, int curr = 1)
{
    if (n == 0)
        return prev;
    if (n == 1)
        return curr;
    return get_term_fib(n - 1, curr, prev + curr);
}
```

每次调用该函数现在都会立即计算斐波那契数列的下一个项，因此函数调用次数与 n 呈线性增长。

第124.2节：带备忘录的递归

递归函数可能非常耗费资源。如果它们是纯函数（即对于相同参数调用总是返回相同值，且不依赖或修改外部状态），则可以通过存储已计算的值以牺牲内存为代价显著加快速度。

下面是带备忘录的斐波那契数列实现：

```
#include <map>

int fibonacci(int n)
{
    static std::map<int, int> values;
    if (n==0 || n==1)
        return n;
    std::map<int,int>::iterator iter = values.find(n);
    if (iter == values.end())
    {
        return values[n] = fibonacci(n-1) + fibonacci(n-2);
    }
    否则
    {
        return iter->second;
    }
}
```

Chapter 124: Recursion in C++

Section 124.1: Using tail recursion and Fibonacci-style recursion to solve the Fibonacci sequence

The simple and most obvious way to use recursion to get the Nth term of the Fibonacci sequence is this

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

However, this algorithm does not scale for higher terms: for bigger and bigger n , the number of function calls that you need to make grows exponentially. This can be replaced with a simple tail recursion.

```
int get_term_fib(int n, int prev = 0, int curr = 1)
{
    if (n == 0)
        return prev;
    if (n == 1)
        return curr;
    return get_term_fib(n - 1, curr, prev + curr);
}
```

Each call to the function now immediately calculates the next term in the Fibonacci sequence, so the number of function calls scales linearly with n .

Section 124.2: Recursion with memoization

Recursive functions can get quite expensive. If they are pure functions (functions that always return the same value when called with the same arguments, and that neither depend on nor modify external state), they can be made considerably faster at the expense of memory by storing the values already calculated.

The following is an implementation of the Fibonacci sequence with memoization:

```
#include <map>

int fibonacci(int n)
{
    static std::map<int, int> values;
    if (n==0 || n==1)
        return n;
    std::map<int,int>::iterator iter = values.find(n);
    if (iter == values.end())
    {
        return values[n] = fibonacci(n-1) + fibonacci(n-2);
    }
    else
    {
        return iter->second;
    }
}
```

请注意，尽管使用了简单的递归公式，但在第一次调用时该函数的时间复杂度为 $O(n)$ 。在随后的相同值调用中，时间复杂度当然是 $O(1)$ 。

但请注意，这个实现不是可重入的。此外，它不允许清除存储的值。另一种实现方式是允许将映射作为额外参数传入：

```
#include <map>

int fibonacci(int n, std::map<int, int> values)
{
    if (n==0 || n==1)
        return n;
    std::map<int,int>::iterator iter = values.find(n);
    if (iter == values.end())
    {
        return values[n] = fibonacci(n-1) + fibonacci(n-2);
    }
    否则
    {
        return iter->second;
    }
}
```

对于这个版本，调用者需要维护存储值的映射。这有一个优点是函数现在是可重入的，且调用者可以移除不再需要的值，从而节省内存。缺点是它破坏了封装性；调用者可以通过向映射中填入错误的值来改变输出。

Note that despite using the simple recursion formula, on first call this function is $O(n)$. On subsequent calls with the same value, it is of course $O(1)$.

Note however that this implementation is not reentrant. Also, it doesn't allow to get rid of stored values. An alternative implementation would be to allow the map to be passed as additional argument:

```
#include <map>

int fibonacci(int n, std::map<int, int> values)
{
    if (n==0 || n==1)
        return n;
    std::map<int,int>::iterator iter = values.find(n);
    if (iter == values.end())
    {
        return values[n] = fibonacci(n-1) + fibonacci(n-2);
    }
    else
    {
        return iter->second;
    }
}
```

For this version, the caller is required to maintain the map with the stored values. This has the advantage that the function is now reentrant, and that the caller can remove values that are no longer needed, saving memory. It has the disadvantage that it breaks encapsulation; the caller can change the output by populating the map with incorrect values.

第125章：算术元编程

这些是使用C++模板元编程在编译时处理算术运算的示例。

第125.1节：以 $O(\log n)$ 计算幂

此示例展示了使用模板元编程计算幂的高效方法。

```
template <int base, unsigned int exponent>
struct power
{
    static const int halfvalue = power<base, exponent / 2>::value;
    static const int value = halfvalue * halfvalue * power<base, exponent % 2>::value;
};

template <int base>
struct power<base, 0>
{
    static const int value = 1;
    static_assert(base != 0, "power<0, 0> is not allowed");
};

template <int base>
struct power<base, 1>
{
    static const int value = base;
};
```

示例用法：

```
std::cout << power<2, 9>::value;
```

版本 ≥ C++14

这个也处理负指数：

```
template <int base, int exponent>
struct powerDouble
{
    static const int exponentAbs = exponent < 0 ? (-exponent) : exponent;
    static const int halfvalue = powerDouble<base, exponentAbs / 2>::intermediateValue;
    static const int intermediateValue = halfvalue * halfvalue * powerDouble<base, exponentAbs % 2>::intermediateValue;

    constexpr static double value = exponent < 0 ? (1.0 / intermediateValue) : intermediateValue;
};

template <int base>
struct powerDouble<base, 0>
{
    static const int intermediateValue = 1;
    constexpr static double value = 1;
    static_assert(base != 0, "powerDouble<0, 0> is not allowed");
};

template <int base>
```

Chapter 125: Arithmetic Metaprogramming

These are examples of using C++ template metaprogramming in processing arithmetic operations at compile time.

Section 125.1: Calculating power in $O(\log n)$

This example shows an efficient way of calculating power using template metaprogramming.

```
template <int base, unsigned int exponent>
struct power
{
    static const int halfvalue = power<base, exponent / 2>::value;
    static const int value = halfvalue * halfvalue * power<base, exponent % 2>::value;
};

template <int base>
struct power<base, 0>
{
    static const int value = 1;
    static_assert(base != 0, "power<0, 0> is not allowed");
};

template <int base>
struct power<base, 1>
{
    static const int value = base;
};
```

Example Usage:

```
std::cout << power<2, 9>::value;
```

Version ≥ C++14

This one also handles negative exponents:

```
template <int base, int exponent>
struct powerDouble
{
    static const int exponentAbs = exponent < 0 ? (-exponent) : exponent;
    static const int halfvalue = powerDouble<base, exponentAbs / 2>::intermediateValue;
    static const int intermediateValue = halfvalue * halfvalue * powerDouble<base, exponentAbs % 2>::intermediateValue;

    constexpr static double value = exponent < 0 ? (1.0 / intermediateValue) : intermediateValue;
};

template <int base>
struct powerDouble<base, 0>
{
    static const int intermediateValue = 1;
    constexpr static double value = 1;
    static_assert(base != 0, "powerDouble<0, 0> is not allowed");
};

template <int base>
```

```
struct powerDouble<base, 1>
{
    static const int intermediateValue = base;
    constexpr static double value = base;
};

int main()
{
    std::cout << powerDouble<2,-3>::value;
}
```

```
struct powerDouble<base, 1>
{
    static const int intermediateValue = base;
    constexpr static double value = base;
};

int main()
{
    std::cout << powerDouble<2,-3>::value;
}
```

第126章：可调用对象

可调用对象是所有可以用作函数的C++结构的集合。实际上，这包括所有你可以传递给C++17标准库函数`invoke()`的东西，或者可以用在`std::function`构造函数中的对象，这些包括：函数指针、带有`operator()`的类、带有隐式转换的类、函数引用、成员函数指针、成员数据指针、lambda表达式。可调用对象在许多STL算法中用作谓词。

第126.1节：函数指针

函数指针是传递函数的最基本方式，也可以在C语言中使用。（详见C语言文档）

就可调用对象而言，函数指针可以定义为：

```
typedef 返回类型(*名称)(参数); // 所有
using 名称 = 返回类型(*)(参数); // <= C++11
using 名称 = std::add_pointer<返回类型(参数)>::type; // <= C++11
using 名称 = std::add_pointer_t<返回类型(参数)>; // <= C++14
```

如果我们使用函数指针来编写自己的vector排序函数，它看起来会是：

```
using LessThanFunctionPtr = std::add_pointer_t<bool(int, int)>;
void sortVectorInt(std::vector<int>&v, LessThanFunctionPtr lessThan) {
    if (v.size() < 2)
        return;
    if (v.size() == 2) {
        if (!lessThan(v.front(), v.back())) // 调用函数指针
            std::swap(v.front(), v.back());
        return;
    }
    std::sort(v, lessThan);
}

bool lessThanInt(int lhs, int rhs) { return lhs < rhs; }
sortVectorInt(vectorOfInt, lessThanInt); // 传递指向普通函数的指针

struct GreaterThanInt {
    static bool cmp(int lhs, int rhs) { return lhs > rhs; }
};
sortVectorInt(vectorOfInt, &GreaterThanInt::cmp); // 传递指向静态成员函数的指针
```

或者，我们也可以通过以下方式调用函数指针：

- `(*lessThan)(v.front(), v.back())` // 全部
- `std::invoke(lessThan, v.front(), v.back())` // C++17 及以前

第126.2节：带有 `operator()` 的类（函数对象）

所有重载了 `operator()` 的类都可以用作函数对象。这些类可以手动编写（通常称为函数对象），也可以通过从 C++11 开始编写 Lambda 由编译器自动生成。

```
struct Person {
    std::string name;
    unsigned int age;
};
```

Chapter 126: Callable Objects

Callable objects are the collection of all C++ structures which can be used as a function. In practice, this are all things you can pass to the C++17 STL function `invoke()` or which can be used in the constructor of `std::function`, this includes: Function pointers, Classes with `operator()`, Classes with implicit conversions, References to functions, Pointers to member functions, Pointers to member data, lambdas. The callable objects are used in many STL algorithms as predicate.

Section 126.1: Function Pointers

Function pointers are the most basic way of passing functions around, which can also be used in C. (See the C documentation for more details).

For the purpose of callable objects, a function pointer can be defined as:

```
typedef returnType(*name)(arguments); // All
using name = returnType(*)(arguments); // <= C++11
using name = std::add_pointer<returnType(arguments)>::type; // <= C++11
using name = std::add_pointer_t<returnType(arguments)>; // <= C++14
```

If we would be using a function pointer for writing our own vector sort, it would look like:

```
using LessThanFunctionPtr = std::add_pointer_t<bool(int, int)>;
void sortVectorInt(std::vector<int>&v, LessThanFunctionPtr lessThan) {
    if (v.size() < 2)
        return;
    if (v.size() == 2) {
        if (!lessThan(v.front(), v.back())) // Invoke the function pointer
            std::swap(v.front(), v.back());
        return;
    }
    std::sort(v, lessThan);
}

bool lessThanInt(int lhs, int rhs) { return lhs < rhs; }
sortVectorInt(vectorOfInt, lessThanInt); // Passes the pointer to a free function

struct GreaterThanInt {
    static bool cmp(int lhs, int rhs) { return lhs > rhs; }
};
sortVectorInt(vectorOfInt, &GreaterThanInt::cmp); // Passes the pointer to a static member function
```

Alternatively, we could have invoked the function pointer one of following ways:

- `(*lessThan)(v.front(), v.back())` // All
- `std::invoke(lessThan, v.front(), v.back())` // <= C++17

Section 126.2: Classes with `operator()` (Functors)

Every class which overloads the `operator()` can be used as a function object. These classes can be written by hand (often referred to as functors) or automatically generated by the compiler by writing Lambdas from C++11 on.

```
struct Person {
    std::string name;
    unsigned int age;
};
```

```
// 用于按姓名查找人的函数对象
FindPersonByName(const std::string &name) : _name(name) {}

// 重载的方法，将被调用
bool operator()(const Person &person) const {
    return person.name == _name;
}

private:
std::string _name;
};

std::vector<Person> v; // 假设这里包含数据
std::vector<Person>::iterator iFind =
std::find_if(v.begin(), v.end(), FindPersonByName("Foobar"));
// ...

```

由于函数对象有自己的身份，不能放入typedef中，必须通过模板参数接受。std::find_if的定义可以如下：

```
template<typename Iterator, typename Predicate>
Iterator find_if(Iterator begin, Iterator end, Predicate &predicate) {
    for (Iterator i = begin, i != end, ++i)
        if (predicate(*i))
            return i;
    return end;
}
```

从 C++17 开始，可以使用 invoke 来调用谓词：std::invoke(predicate, *i)。

```
// Functor which find a person by name
struct FindPersonByName {
    FindPersonByName(const std::string &name) : _name(name) {}

    // Overloaded method which will get called
    bool operator()(const Person &person) const {
        return person.name == _name;
    }

private:
    std::string _name;
};

std::vector<Person> v; // Assume this contains data
std::vector<Person>::iterator iFind =
    std::find_if(v.begin(), v.end(), FindPersonByName("Foobar"));
// ...

```

As functors have their own identity, they cannot be put in a typedef and these have to be accepted via template argument. The definition of std::find_if can look like:

```
template<typename Iterator, typename Predicate>
Iterator find_if(Iterator begin, Iterator end, Predicate &predicate) {
    for (Iterator i = begin, i != end, ++i)
        if (predicate(*i))
            return i;
    return end;
}
```

From C++17 on, the calling of the predicate can be done with invoke: std::invoke(predicate, *i).

第127章：客户端服务器示例

第127.1节：Hello TCP 客户端

该程序是 Hello TCP 服务器程序的配套程序，你可以运行其中一个来验证彼此的有效性。程序流程与 Hello TCP 服务器非常相似，因此也请务必查看该程序。

代码如下 -

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

#include <arpa/inet.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    // 现在我们将 IP 地址和端口号作为程序的参数传入
    if (argc != 3) {
        std::cerr << "请按 'program <ipaddress> <port>' 格式运行程序";return -1;
    }

    auto &ipAddress = argv[1];
    auto &portNum = argv[2];

    addrinfo hints, *p;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;

    int gAddRes = getaddrinfo(ipAddress, portNum, &hints, &p);
    if (gAddRes != 0) {
        std::cerr << gai_strerror(gAddRes) << "";return -2;
    }

    if (p == NULL) {
        std::cerr << "未找到地址";return -3;
    }

    // socket() 调用创建一个新的套接字并返回其描述符
    int sockFD = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
    if (sockFD == -1) {
        std::cerr << "创建套接字时出错";return -4;
    }

    // 注意：这里没有像 Hello TCP Server 中那样调用 bind()
    // 为什么？你当然可以调用，但这不是必须的
    // 因为客户端不一定需要固定端口号
    // 所以下一次调用会将其绑定到一个随机可用端口号
```

Chapter 127: Client server examples

Section 127.1: Hello TCP Client

This program is complimentary to Hello TCP Server program, you can run either of them to check the validity of each other. The program flow is quite common with Hello TCP server, so make sure to take a look at that too.

Here's the code -

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

#include <arpa/inet.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    // Now we're taking an ipaddress and a port number as arguments to our program
    if (argc != 3) {
        std::cerr << "Run program as 'program <ipaddress> <port>'\n";
        return -1;
    }

    auto &ipAddress = argv[1];
    auto &portNum = argv[2];

    addrinfo hints, *p;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;

    int gAddRes = getaddrinfo(ipAddress, portNum, &hints, &p);
    if (gAddRes != 0) {
        std::cerr << gai_strerror(gAddRes) << "\n";
        return -2;
    }

    if (p == NULL) {
        std::cerr << "No addresses found\n";
        return -3;
    }

    // socket() call creates a new socket and returns its descriptor
    int sockFD = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
    if (sockFD == -1) {
        std::cerr << "Error while creating socket\n";
        return -4;
    }

    // Note: there is no bind() call as there was in Hello TCP Server
    // why? well you could call it though it's not necessary
    // because client doesn't necessarily have a fixed port number
    // so next call will bind it to a random available port number
```

```

// connect() 调用尝试与指定服务器建立 TCP 连接
int connectR = connect(sockFD, p->ai_addr, p->ai_addrlen);
if (connectR == -1) {
close(sockFD);
std::cerr << "连接套接字时出错";return -5;
}

std::string reply(15, ' ');

// recv() 调用尝试从服务器获取响应
// 但这里有个问题，响应可能需要多次调用
// recv() 才能完全接收
// 这将在另一个示例中演示，为了保持简洁
auto bytes_recv = recv(sockFD, &reply.front(), reply.size(), 0);
if (bytes_recv == -1) {
std::cerr << "接收字节时出错";return -6;
}

std::cout << "客户端接收: " << reply << std::endl; close(sockFD);

freeaddrinfo(p);

return 0;
}

```

第127.2节：Hello TCP服务器

首先让我再说，你应该先访问Beej的网络编程指南并快速阅读一遍，该指南对大部分内容有更详细的解释。我们将在这里创建一个简单的TCP服务器，它会对所有传入连接发送“Hello World”，然后关闭连接。另一个需要注意的是，服务器将以迭代方式与客户端通信，也就是说一次处理一个客户端。务必查看相关的手册页，因为它们可能包含关于每个函数调用和套接字结构的重要信息。

我们将使用端口运行服务器，因此也会接受一个端口号参数。让我们开始编写代码——

```

#include <cstring> // sizeof()
#include <iostream>
#include <string>

// 用于 socket()、getaddrinfo() 等函数的头文件
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/types.h>

#include <unistd.h> // close()

int main(int argc, char *argv[])
{
    // 检查是否提供了端口号..
    if (argc != 2) {
std::cerr << "请按 'program <port>' 格式运行程序"";return -1;
    }

    auto &portNum = argv[1];
    const unsigned int backLog = 8; // 允许进入队列的连接数
}

```

```

// connect() call tries to establish a TCP connection to the specified server
int connectR = connect(sockFD, p->ai_addr, p->ai_addrlen);
if (connectR == -1) {
close(sockFD);
std::cerr << "Error while connecting socket\n";
return -5;
}

std::string reply(15, ' ');

// recv() call tries to get the response from server
// BUT there's a catch here, the response might take multiple calls
// to recv() before it is completely received
// will be demonstrated in another example to keep this minimal
auto bytes_recv = recv(sockFD, &reply.front(), reply.size(), 0);
if (bytes_recv == -1) {
std::cerr << "Error while receiving bytes\n";
return -6;
}

std::cout << "\nClient received: " << reply << std::endl;
close(sockFD);
freeaddrinfo(p);

return 0;
}

```

Section 127.2: Hello TCP Server

Let me start by saying you should first visit [Beej's Guide to Network Programming](#) and give it a quick read, which explains most of this stuff a bit more verbosely. We'll be creating a simple TCP server here which will say "Hello World" to all incoming connections and then close them. Another thing to note is, the server will be communicating to clients iteratively, which means one client at a time. Make sure to check out relevant man pages as they might contain valuable information about each function call and socket structures.

We'll run the server with a port, so we'll take an argument for port number as well. Let's get started with code -

```

#include <cstring> // sizeof()
#include <iostream>
#include <string>

// headers for socket(), getaddrinfo() and friends
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/types.h>

#include <unistd.h> // close()

int main(int argc, char *argv[])
{
    // Let's check if port number is supplied or not..
    if (argc != 2) {
std::cerr << "Run program as 'program <port>'\n";
return -1;
    }

    auto &portNum = argv[1];
    const unsigned int backLog = 8; // number of connections allowed on the incoming queue
}

```

```

addrinfo hints, *res, *p; // 我们需要两个指针, res 用于保存, p 用于遍历
memset(&hints, 0, sizeof(hints));

// 更多说明请参考 man socket
hints.ai_family = AF_UNSPEC; // 目前不指定使用哪个 IP 版本
hints.ai_socktype = SOCK_STREAM; // SOCK_STREAM 指 TCP, SOCK_DGRAM 是?
hints.ai_flags = AI_PASSIVE;

// man getaddrinfo
int gAddRes = getaddrinfo(NULL, portNum, &hints, &res);
if (gAddRes != 0) {
    std::cerr << gai_strerror(gAddRes) << "" ; return -2;
}

std::cout << "检测地址" << std::endl;

unsigned int numOfAddr = 0;
char ipStr[INET6_ADDRSTRLEN]; // ipv6 长度确保此变量能存储 ipv4/6 地址

// 由于 getaddrinfo() 已经给了我们一个地址列表
// 我们将遍历这些地址并让用户选择一个
// 供程序绑定的地址
for (p = res; p != NULL; p = p->ai_next) {
    void *addr;
    std::string ipVer;

    // 如果地址是 IPv4 地址
    if (p->ai_family == AF_INET) {
        ipVer = "IPv4";
        sockaddr_in *ipv4 = reinterpret_cast<sockaddr_in *>(p->ai_addr);
        addr = &(ipv4->sin_addr);
        ++numOfAddr;
    }

    // 如果地址是 IPv6 地址
    else {
        ipVer = "IPv6";
        sockaddr_in6 *ipv6 = reinterpret_cast<sockaddr_in6 *>(p->ai_addr);
        addr = &(ipv6->sin6_addr);
        ++numOfAddr;
    }

    // 将 IPv4 和 IPv6 地址从二进制转换为文本形式
    inet_ntop(p->ai_family, addr, ipStr, sizeof(ipStr));
    std::cout << "(" << numOfAddr << ")" << ipVer << ":" << ipStr
        << std::endl;
}

// 如果没有找到地址 :(
if (!numOfAddr) {
    std::cerr << "未找到可用的主机地址"; return -3;
}

// 让用户选择一个地址
std::cout << "请输入要绑定的主机地址编号：" ;
unsigned int choice = 0;
bool madeChoice = false;

```

```

addrinfo hints, *res, *p; // we need 2 pointers, res to hold and p to iterate over
memset(&hints, 0, sizeof(hints));

// for more explanation, man socket
hints.ai_family = AF_UNSPEC; // don't specify which IP version to use yet
hints.ai_socktype = SOCK_STREAM; // SOCK_STREAM refers to TCP, SOCK_DGRAM will be?
hints.ai_flags = AI_PASSIVE;

// man getaddrinfo
int gAddRes = getaddrinfo(NULL, portNum, &hints, &res);
if (gAddRes != 0) {
    std::cerr << gai_strerror(gAddRes) << "\n";
    return -2;
}

std::cout << "Detecting addresses" << std::endl;

unsigned int numOfAddr = 0;
char ipStr[INET6_ADDRSTRLEN]; // ipv6 length makes sure both ipv4/6 addresses can be stored
in this variable

// Now since getaddrinfo() has given us a list of addresses
// we're going to iterate over them and ask user to choose one
// address for program to bind to
for (p = res; p != NULL; p = p->ai_next) {
    void *addr;
    std::string ipVer;

    // if address is ipv4 address
    if (p->ai_family == AF_INET) {
        ipVer = "IPv4";
        sockaddr_in *ipv4 = reinterpret_cast<sockaddr_in *>(p->ai_addr);
        addr = &(ipv4->sin_addr);
        ++numOfAddr;
    }

    // if address is ipv6 address
    else {
        ipVer = "IPv6";
        sockaddr_in6 *ipv6 = reinterpret_cast<sockaddr_in6 *>(p->ai_addr);
        addr = &(ipv6->sin6_addr);
        ++numOfAddr;
    }

    // convert IPv4 and IPv6 addresses from binary to text form
    inet_ntop(p->ai_family, addr, ipStr, sizeof(ipStr));
    std::cout << "(" << numOfAddr << ")" << ipVer << ":" << ipStr
        << std::endl;
}

// if no addresses found :(
if (!numOfAddr) {
    std::cerr << "Found no host address to use\n";
    return -3;
}

// ask user to choose an address
std::cout << "Enter the number of host address to bind with: ";
unsigned int choice = 0;
bool madeChoice = false;

```

```

do {
    std::cin >> choice;
    if (choice > (numOfAddr + 1) || choice < 1) {
        madeChoice = false;
    std::cout << "选择错误, 请重试!" << std::endl;
    } else
madeChoice = true;
} while (!madeChoice);

p = res;

// 让我们创建一个新的套接字, socketFD 作为描述符返回
// 有关更多信息, 请参阅 man socket
// 这些调用通常因某些错误返回 -1 作为结果
int sockFD = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
if (sockFD == -1) {
std::cerr << "创建套接字时出错"; freeaddrinfo(res);

    return -4;
}

// 让我们将地址绑定到刚创建的套接字上
int bindR = bind(sockFD, p->ai_addr, p->ai_addrlen);
if (bindR == -1) {
std::cerr << "绑定套接字时出错";

    // 如果发生错误, 确保关闭套接字并释放资源
    close(sockFD);
freeaddrinfo(res);
    return -5;
}

// 最后开始监听套接字上的连接
int listenR = listen(sockFD, backLog);
if (listenR == -1) {
std::cerr << "监听套接字时出错";

    // 如果发生错误, 确保关闭套接字并释放资源
    close(sockFD);
freeaddrinfo(res);
    return -6;
}

// 结构体足够大以存放客户端地址
sockaddr_storage client_addr;
socklen_t client_addr_size = sizeof(client_addr);

const std::string response = "Hello World";

// 一个新的无限循环, 用于与传入连接通信
// 这将一次处理一个客户端连接
// 在后续示例中, 我们将为每个客户端连接使用 fork() 调用
while (1) {

    // accept 调用将为我们提供一个新的套接字描述符
}

```

```

do {
    std::cin >> choice;
    if (choice > (numOfAddr + 1) || choice < 1) {
        madeChoice = false;
    std::cout << "Wrong choice, try again!" << std::endl;
    } else
        madeChoice = true;
} while (!madeChoice);

p = res;

// let's create a new socket, socketFD is returned as descriptor
// man socket for more information
// these calls usually return -1 as result of some error
int sockFD = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
if (sockFD == -1) {
    std::cerr << "Error while creating socket\n";
    freeaddrinfo(res);
    return -4;
}

// Let's bind address to our socket we've just created
int bindR = bind(sockFD, p->ai_addr, p->ai_addrlen);
if (bindR == -1) {
    std::cerr << "Error while binding socket\n";

    // if some error occurs, make sure to close socket and free resources
    close(sockFD);
    freeaddrinfo(res);
    return -5;
}

// finally start listening for connections on our socket
int listenR = listen(sockFD, backLog);
if (listenR == -1) {
    std::cerr << "Error while Listening on socket\n";

    // if some error occurs, make sure to close socket and free resources
    close(sockFD);
    freeaddrinfo(res);
    return -6;
}

// structure large enough to hold client's address
sockaddr_storage client_addr;
socklen_t client_addr_size = sizeof(client_addr);

const std::string response = "Hello World";

// a fresh infinite loop to communicate with incoming connections
// this will take client connections one at a time
// in further examples, we're going to use fork() call for each client connection
while (1) {

    // accept call will give us a new socket descriptor
}

```

```

int newFD
    = accept(sockFD, (sockaddr *) &client_addr, &client_addr_size);
if (newFD == -1) {
std::cerr << "在套接字上接受时出错";continue;
}

// send 调用发送你指定的第二个参数数据及其长度作为第三个参数，同时
// 返回实际发送的字节数
auto bytes_sent = send(newFD, response.data(), response.length(), 0);
close(newFD);
}

close(sockFD);
freeaddrinfo(res);

return 0;
}

```

以下程序运行方式为 -

```

检测地址
(1) IPv4 : 0.0.0.0
(2) IPv6 : ::

输入绑定的主机地址编号: 1

```

```

int newFD
    = accept(sockFD, (sockaddr *) &client_addr, &client_addr_size);
if (newFD == -1) {
    std::cerr << "Error while Accepting on socket\n";
    continue;
}

// send call sends the data you specify as second param and it's length as 3rd param, also
// returns how many bytes were actually sent
auto bytes_sent = send(newFD, response.data(), response.length(), 0);
close(newFD);
}

close(sockFD);
freeaddrinfo(res);

return 0;
}

```

The following program runs as -

```

Detecting addresses
(1) IPv4 : 0.0.0.0
(2) IPv6 : ::

Enter the number of host address to bind with: 1

```

第128章：常量正确性

第128.1节：基础知识

`const` 正确性 是设计代码的一种实践，确保只有需要修改实例的代码才能修改实例（即拥有写权限），反之，任何不需要修改实例的代码都无法修改实例（即只有读权限）。这防止实例被无意修改，使代码更不易出错，并且说明代码是否打算改变实例的状态。它还允许实例在不需要修改时被视为 `const`，或者在初始化后不需要更改时定义为 `const`，且不丢失任何功能。

这是通过给成员函数添加`const` CV限定符，并使指针/引用参数为`const`来实现的，除非它们需要写访问权限。

```
类 ConstCorrectClass {  
    int x;  
  
    公共:  
        int getX() const { return x; } // 函数是const：不修改实例。  
        void setX(int i) { x = i; } // 非const：修改实例。  
    };  
  
    // 参数是const：不修改参数。  
    int const_correct_reader(const ConstCorrectClass& c) {  
        return c.getX();  
    }  
  
    // 参数不是const：修改参数。  
    void const_correct_writer(ConstCorrectClass& c) {  
        c.setX(42);  
    }  
  
    const ConstCorrectClass invariant; // 实例是const：不能被修改。  
    ConstCorrectClass variant; // 实例不是常量：可以被修改。  
  
    // ...  
  
    const_correct_reader(不变); // 正确。在const实例上调用非修改函数。  
    const_correct_reader(可变); // 正确。在可修改实例上调用非修改函数。  
  
    const_correct_writer(可变); // 正确。在可修改实例上调用修改函数。  
    const_correct_writer(不变); // 错误。在const实例上调用修改函数。
```

由于`const`正确性的特性，这从类的成员函数开始，逐步向外扩展；如果你尝试从`const`实例调用非`const`成员函数，或者从被视为`const`的非`const`实例调用，编译器会报错，提示丢失cv限定符。

第128.2节：Const正确的类设计

在一个`const`正确的类中，所有不改变逻辑状态的成员函数，其`this`指针都被`cv`限定为`const`，表示它们不会修改对象（除了任何`mutable`字段，这些字段即使在`const`实例中也可以自由修改）；如果一个`const` `cv`限定的函数返回引用，该引用也应为`const`。这允许它们在常量和非`cv`限定的实例上都能被调用，因为`const T*`可以绑定到`T*`或`const T*`。这反过来允许函数在不需要修改传入参数时，将其声明为`const`引用参数，而不会丧失任何功能。

Chapter 128: Const Correctness

Section 128.1: The Basics

`const` correctness is the practice of designing code so that only code that *needs* to modify an instance (*i.e.* has write access), and conversely, that any code that doesn't need to modify an instance is unable to do so (*i.e.* only has read access). This prevents the instance from being modified unintentionally, making code less error-prone, and documents whether the code is intended to change the instance's state or not. It also allows instances to be treated as `const` whenever they don't need to be modified, or defined as `const` if they don't need to be changed after initialisation, without losing any functionality.

This is done by giving member functions `const` CV-qualifiers, and by making pointer/reference parameters `const`, except in the case that they need write access.

```
class ConstCorrectClass {  
    int x;  
  
    public:  
        int getX() const { return x; } // Function is const: Doesn't modify instance.  
        void setX(int i) { x = i; } // Not const: Modifies instance.  
    };  
  
    // Parameter is const: Doesn't modify parameter.  
    int const_correct_reader(const ConstCorrectClass& c) {  
        return c.getX();  
    }  
  
    // Parameter isn't const: Modifies parameter.  
    void const_correct_writer(ConstCorrectClass& c) {  
        c.setX(42);  
    }  
  
    const ConstCorrectClass invariant; // Instance is const: Can't be modified.  
    ConstCorrectClass variant; // Instance isn't const: Can be modified.  
  
    // ...  
  
    const_correct_reader(invariant); // Good. Calling non-modifying function on const instance.  
    const_correct_reader(variant); // Good. Calling non-modifying function on modifiable instance.  
  
    const_correct_writer(variant); // Good. Calling modifying function on modifiable instance.  
    const_correct_writer(invariant); // Error. Calling modifying function on const instance.
```

Due to the nature of `const` correctness, this starts with the class' member functions, and works its way outwards; if you try to call a non-`const` member function from a `const` instance, or from a non-`const` instance being treated as `const`, the compiler will give you an error about it losing cv-qualifiers.

Section 128.2: Const Correct Class Design

In a `const`-correct class, all member functions which don't change logical state have `this` cv-qualified as `const`, indicating that they don't modify the object (apart from any `mutable` fields, which can freely be modified even in `const` instances); if a `const` cv-qualified function returns a reference, that reference should also be `const`. This allows them to be called on both constant and non-cv-qualified instances, as a `const T*` is capable of binding to either a `T*` or a `const T*`. This, in turn, allows functions to declare their passed-by-reference parameters as `const` when they don't need to be modified, without losing any functionality.

此外，在`const`正确的类中，所有按引用传递的函数参数都会是`const`正确的，如“Const正确的函数参数”中所讨论的，这样它们只有在函数明确需要修改时才会被修改。

首先，让我们来看一下`this`的cv限定符：

```
// 假设有类 Field，包含成员函数 "void insert_value(int);".

class ConstIncorrect {
    Field fld;

    公共:
    ConstIncorrect(Field& f); // 修改操作。

    Field& getField(); // 可能会修改。同时以非常量引用暴露成员,
                        // 允许间接修改。
    void setField(Field& f); // 修改操作。

    void doSomething(int i); // 可能会修改。
    void doNothing(); // 可能会修改。
};

ConstIncorrect::ConstIncorrect(Field& f) : fld(f) {} // 修改操作。
Field& ConstIncorrect::getField() { return fld; } // 不修改。
void ConstIncorrect::setField(Field& f) { fld = f; } // 修改操作。
void ConstIncorrect::doSomething(int i) { // 修改。
    fld.insert_value(i);
}
void ConstIncorrect::doNothing() {} // 不修改。

class ConstCorrectCVQ {
    Field fld;

    公共:
    ConstCorrectCVQ(Field& f); // 修改。

    const Field& getField() const; // 不修改。以const引用暴露成员,
                                // 防止间接修改。
    void setField(Field& f); // 修改。

    void doSomething(int i); // 修改。
    void doNothing() const; // 不修改。
};

ConstCorrectCVQ::ConstCorrectCVQ(Field& f) : fld(f) {}
Field& ConstCorrectCVQ::getField() const { return fld; }
void ConstCorrectCVQ::setField(Field& f) { fld = f; }
void ConstCorrectCVQ::doSomething(int i) {
    fld.insert_value(i);
}
void ConstCorrectCVQ::doNothing() const {}

// 这将不起作用。
// 不能在const ConstIncorrect 实例上调用成员函数。
void const_correct_func(const ConstIncorrect& c) {
    Field f = c.getField();
    c.do_nothing();
}

// 但这将可以。
```

Furthermore, in a `const` correct class, all passed-by-reference function parameters will be `const` correct, as discussed in Const Correct Function Parameters, so that they can only be modified when the function explicitly needs to modify them.

First, let's look at `this` cv-qualifiers:

```
// Assume class Field, with member function "void insert_value(int);".

class ConstIncorrect {
    Field fld;

    public:
    ConstIncorrect(Field& f); // Modifies.

    Field& getField(); // Might modify. Also exposes member as non-const reference,
                       // allowing indirect modification.
    void setField(Field& f); // Modifies.

    void doSomething(int i); // Might modify.
    void doNothing(); // Might modify.
};

ConstIncorrect::ConstIncorrect(Field& f) : fld(f) {} // Modifies.
Field& ConstIncorrect::getField() { return fld; } // Doesn't modify.
void ConstIncorrect::setField(Field& f) { fld = f; } // Modifies.
void ConstIncorrect::doSomething(int i) {
    fld.insert_value(i);
}
void ConstIncorrect::doNothing() {} // Doesn't modify.

class ConstCorrectCVQ {
    Field fld;

    public:
    ConstCorrectCVQ(Field& f); // Modifies.

    const Field& getField() const; // Doesn't modify. Exposes member as const reference,
                                // preventing indirect modification.
    void setField(Field& f); // Modifies.

    void doSomething(int i); // Modifies.
    void doNothing() const; // Doesn't modify.
};

ConstCorrectCVQ::ConstCorrectCVQ(Field& f) : fld(f) {}
Field& ConstCorrectCVQ::getField() const { return fld; }
void ConstCorrectCVQ::setField(Field& f) { fld = f; }
void ConstCorrectCVQ::doSomething(int i) {
    fld.insert_value(i);
}
void ConstCorrectCVQ::doNothing() const {}

// This won't work.
// No member functions can be called on const ConstIncorrect instances.
void const_correct_func(const ConstIncorrect& c) {
    Field f = c.getField();
    c.do_nothing();
}

// But this will.
```

```
// 可以在 const ConstCorrectCVQ 实例上调用 getField() 和 doNothing()。
void const_correct_func(const ConstCorrectCVQ& c) {
    Field f = c.getField();
    c.do_nothing();
}
```

我们可以将其与Const Correct Function Parameters结合，使类完全符合const正确性。

```
class ConstCorrect {
    Field fld;

    public:
    ConstCorrect(const Field& f); // 修改实例。不修改参数。

    const Field& getField() const; // 不修改。以const引用暴露成员，// 防止间接修改。
    void setField(const Field& f); // 修改实例。不修改参数。

    void doSomething(int i); // 修改。不修改参数（按值传递）。
    void doNothing() const; // 不修改。
};

ConstCorrect::ConstCorrect(const Field& f) : fld(f) {}
Field& ConstCorrect::getField() const { return fld; }
void ConstCorrect::setField(const Field& f) { fld = f; }
void ConstCorrect::doSomething(int i) {
    fld.insert_value(i);
}
void ConstCorrect::doNothing() const {}
```

这也可以与基于const性的重载结合使用，适用于当实例是const时希望有一种行为，而不是时希望有另一种行为；这类用法常见于容器提供访问器，仅当容器本身非const时才允许修改。

```
class ConstCorrectContainer {
    int arr[5];

    public:
    // 下标运算符如果实例是const则提供只读访问，否则提供读写访问
    // ...
    int& operator[](size_t index) { return arr[index]; }
    const int& operator[](size_t index) const { return arr[index]; }

    // ...
};
```

这在标准库中很常见，大多数容器都会提供重载以考虑const性。

第128.3节：Const正确的函数参数

在const正确的函数中，所有按引用传递的参数都会被标记为const，除非函数直接或间接修改它们，这样可以防止程序员无意中修改不该修改的内容。这允许函数接受const和非cv限定的实例，反过来，当调用成员函数时，实例的this指针类型为const T*，其中T是类的类型。

```
struct 示例 {
    void 函数() { std::cout << 3 << std::endl; }
```

```
// getField() 和 doNothing() 可以被调用在 const ConstCorrectCVQ 实例上。
```

```
void const_correct_func(const ConstCorrectCVQ& c) {
    Field f = c.getField();
    c.do_nothing();
}
```

We can then combine this with Const Correct Function Parameters, causing the class to be fully `const`-correct.

```
class ConstCorrect {
    Field fld;

    public:
    ConstCorrect(const Field& f); // Modifies instance. Doesn't modify parameter.

    const Field& getField() const; // Doesn't modify. Exposes member as const reference,
                                  // preventing indirect modification.
    void setField(const Field& f); // Modifies instance. Doesn't modify parameter.

    void doSomething(int i); // Modifies. Doesn't modify parameter (passed by value).
    void doNothing() const; // Doesn't modify.
};

ConstCorrect::ConstCorrect(const Field& f) : fld(f) {}
Field& ConstCorrect::getField() const { return fld; }
void ConstCorrect::setField(const Field& f) { fld = f; }
void ConstCorrect::doSomething(int i) {
    fld.insert_value(i);
}
void ConstCorrect::doNothing() const {}
```

This can also be combined with overloading based on `constness`, in the case that we want one behaviour if the instance is `const`, and a different behaviour if it isn't; a common use for this is containers providing accessors that only allow modification if the container itself is non-`const`.

```
class ConstCorrectContainer {
    int arr[5];

    public:
    // Subscript operator provides read access if instance is const, or read/write access
    // otherwise.
    int& operator[](size_t index) { return arr[index]; }
    const int& operator[](size_t index) const { return arr[index]; }

    // ...
};
```

This is commonly used in the standard library, with most containers providing overloads to take `constness` into account.

Section 128.3: Const Correct Function Parameters

In a `const`-correct function, all passed-by-reference parameters are marked as `const` unless the function directly or indirectly modifies them, preventing the programmer from inadvertently changing something they didn't mean to change. This allows the function to take both `const` and non-cv-qualified instances, and in turn, causes the instance's `this` to be of type `const T*` when a member function is called, where `T` is the class' type.

```
struct Example {
    void func() { std::cout << 3 << std::endl; }
```

```

void 函数() const { std::cout << 5 << std::endl; }

void const_错误函数(示例& one, 示例* two) {
    one.函数();
    two->函数();
}

void const_正确函数(const 示例& one, const 示例* two) {
    one.函数();
    two->函数();
}

int main() {
    示例 a, b;
    const_错误函数(a, &b);
    const_正确函数(a, &b);
}

// 输出:
3
5
5
虽然其影响不如那样立即显现

```

const 正确的类设计（在const-正确函数和const-错误的类会导致编译错误，而const-正确的类和const-错误的函数将能正确编译），const正确的函数会捕获许多const错误函数会漏掉的错误，例如下面的例子。[但请注意，如果传入了const实例而函数期望非const实例，const-错误的函数将导致编译错误。]

```

// 从向量中读取值，然后计算并返回一个值。
// 缓存返回值以提高速度。
template<typename T>
const T& bad_func(std::vector<T>& v, Helper<T>& h) {
    // 缓存值，供将来使用。
    // 一旦计算出返回值，就缓存它并注册其索引。
    static std::vector<T> vals = {};

    int v_ind = h.get_index();           // v 的当前工作索引。
    int vals_ind = h.get_cache_index(v_ind); // 如果缓存索引未注册，则为-1。

    if (vals.size() && (vals_ind != -1) && (vals_ind < vals.size()) && !(h.needs_recalc())) {
        return vals[h.get_cache_index(v_ind)];
    }

    T temp = v[v_ind];

    temp -= h.poll_device();
    temp *= h.obtain_random();
    temp += h.do_tedious_calculation(temp, v[h.get_last_handled_index()]);

    // 我们突然感到疲倦，这种情况就会发生。
    if (vals_ind != -1) {
        vals[vals_ind] = temp;
    } else {
        v.push_back(temp); // 哟。应该访问的是 vals。
        vals_ind = vals.size() - 1;
        h.register_index(v_ind, vals_ind);
    }
}

```

```

void func() const { std::cout << 5 << std::endl; }

void const_incorrect_function(Example& one, Example* two) {
    one.func();
    two->func();
}

void const_correct_function(const Example& one, const Example* two) {
    one.func();
    two->func();
}

int main() {
    Example a, b;
    const_incorrect_function(a, &b);
    const_correct_function(a, &b);
}

// Output:
3
3
5
5

```

While the effects of this are less immediately visible than those of `const` correct class design (in that `const`-correct functions and `const`-incorrect classes will cause compilation errors, while `const`-correct classes and `const`-incorrect functions will compile properly), `const` correct functions will catch a lot of errors that `const` incorrect functions would let slip through, such as the one below. [Note, however, that a `const`-incorrect function *will* cause compilation errors if passed a `const` instance when it expected a non-`const` one.]

```

// Read value from vector, then compute & return a value.
// Caches return values for speed.
template<typename T>
const T& bad_func(std::vector<T>& v, Helper<T>& h) {
    // Cache values, for future use.
    // Once a return value has been calculated, it's cached & its index is registered.
    static std::vector<T> vals = {};

    int v_ind = h.get_index();           // Current working index for v.
    int vals_ind = h.get_cache_index(v_ind); // Will be -1 if cache index isn't registered.

    if (vals.size() && (vals_ind != -1) && (vals_ind < vals.size()) && !(h.needs_recalc())) {
        return vals[h.get_cache_index(v_ind)];
    }

    T temp = v[v_ind];

    temp -= h.poll_device();
    temp *= h.obtain_random();
    temp += h.do_tedious_calculation(temp, v[h.get_last_handled_index()]);

    // We're feeling tired all of a sudden, and this happens.
    if (vals_ind != -1) {
        vals[vals_ind] = temp;
    } else {
        v.push_back(temp); // Oops. Should've been accessing vals.
        vals_ind = vals.size() - 1;
        h.register_index(v_ind, vals_ind);
    }
}

```

```

    return vals[vals_ind];
}

// 常量正确版本。与上述版本相同，因此大部分内容将被跳过。
template<typename T>
const T& good_func(const std::vector<T>& v, Helper<T>& h) {
    // ...

    // 我们突然感到疲倦，这种情况就会发生。
    if (vals_ind != -1) {
        vals[vals_ind] = temp;
    } else {
        v.push_back(temp); // 错误：丢弃限定符。
        vals_ind = vals.size() - 1;
        h.register_index(v_ind, vals_ind);
    }

    return vals[vals_ind];
}

```

第128.4节：作为文档的常量正确性

关于`const`正确性最有用的方面之一是它作为代码文档的一种方式，为程序员和其他用户提供某些保证。这些保证由编译器通过`constness`强制执行，缺少`constness`则表示代码不提供这些保证。

`const CV`限定成员函数：

- 任何`const`成员函数都可以被假定为意图读取实例，并且：
 - 不得修改其调用的实例的逻辑状态。因此，除`mutable`变量外，不得修改其调用的实例的任何成员变量。
 - 不得调用任何会修改该实例成员变量的其他函数，除非是`mutable`变量。
- 相反，任何不是`const`的成员函数都可以被认为有意图修改该实例，并且：
 - 可能会也可能不会修改逻辑状态。
 - 可能会也可能不会调用其他修改逻辑状态的函数。

这可以用来对调用任意成员函数后对象的状态做出假设，即使没有看到该函数的定义：

```

// ConstMemberFunctions.h

class ConstMemberFunctions {
    int val;
    mutable int cache;
    mutable bool state_changed;

    公共:
        // 构造函数显然会改变逻辑状态。无需做出假设。
        ConstMemberFunctions(int v = 0);

        // 我们可以假设此函数不会改变逻辑状态，也不会调用
        // set_val()。它可能会调用也可能不会调用 squared_calc() 或 bad_func()。
        int calc() const;

        // 我们可以假设此函数不会改变逻辑状态，也不会调用
        // set_val()。它可能会调用也可能不会调用 calc() 或 bad_func()。
}
```

```

    return vals[vals_ind];
}

// Const correct version. Is identical to above version, so most of it shall be skipped.
template<typename T>
const T& good_func(const std::vector<T>& v, Helper<T>& h) {
    // ...

    // We're feeling tired all of a sudden, and this happens.
    if (vals_ind != -1) {
        vals[vals_ind] = temp;
    } else {
        v.push_back(temp); // Error: discards qualifiers.
        vals_ind = vals.size() - 1;
        h.register_index(v_ind, vals_ind);
    }

    return vals[vals_ind];
}

```

Section 128.4: Const Correctness as Documentation

One of the more useful things about `const` correctness is that it serves as a way of documenting code, providing certain guarantees to the programmer and other users. These guarantees are enforced by the compiler due to `constness`, with a lack of `constness` in turn indicating that code doesn't provide them.

`const CV-Qualified Member Functions:`

- Any member function which is `const` can be assumed to have intent to read the instance, and:
 - Shall not modify the logical state of the instance they are called on. Therefore, they shall not modify any member variables of the instance they are called on, except `mutable` variables.
 - Shall not call any *other* functions that would modify any member variables of the instance, except `mutable` variables.
- Conversely, any member function which isn't `const` can be assumed to have intent to modify the instance, and:
 - May or may not modify logical state.
 - May or may not call other functions which modify logical state.

This can be used to make assumptions about the state of the object after any given member function is called, even without seeing the definition of that function:

```

// ConstMemberFunctions.h

class ConstMemberFunctions {
    int val;
    mutable int cache;
    mutable bool state_changed;

    public:
        // Constructor clearly changes logical state. No assumptions necessary.
        ConstMemberFunctions(int v = 0);

        // We can assume this function doesn't change logical state, and doesn't call
        // set_val(). It may or may not call squared_calc() or bad_func().
        int calc() const;

        // We can assume this function doesn't change logical state, and doesn't call
        // set_val(). It may or may not call calc() or bad_func().
}
```

```

int squared_calc() const;

// 我们可以假设此函数不会改变逻辑状态，也不会调用
// set_val()。它可能会调用也可能不会调用 calc() 或 squared_calc()。
void bad_func() const;

// 我们可以假设此函数会改变逻辑状态，且可能会调用也可能不会调用
// calc()、squared_calc() 或 bad_func()。
void set_val(int v);
};

```

由于const规则，这些假设实际上将由编译器强制执行。

```

// ConstMemberFunctions.cpp

ConstMemberFunctions::ConstMemberFunctions(int v /* = 0 */)
    : cache(0), val(v), state_changed(true) {}

// 我们的假设是正确的。
int ConstMemberFunctions::calc() const {
    if (state_changed) {
        cache = 3 * val;
        state_changed = false;
    }

    return cache;
}

// 我们的假设是正确的。
int ConstMemberFunctions::squared_calc() const {
    return calc() * calc();
}

// 我们的假设是错误的。
// 函数无法编译，因为 `this` 失去了限定符。
void ConstMemberFunctions::bad_func() const {
    set_val(863);
}

// 我们的假设是正确的。
void ConstMemberFunctions::set_val(int v) {
    if (v != val) {
        val = v;
        state_changed = true;
    }
}

```

const 函数参数：

- 任何带有一个或多个**const**参数的函数，都可以假定其意图是读取这些参数，并且：
 - 不得修改这些参数，或调用任何会修改它们的成员函数。
 - 不得将这些参数传递给任何会修改它们和/或调用任何会修改它们的成员函数的其他函数。
- 相反，任何带有一个或多个非**const**参数的函数，都可以假定其意图是修改这些参数，并且：
 - 可能会或可能不会修改这些参数，或调用任何会修改它们的成员函数。
 - 可能会或可能不会将这些参数传递给会修改它们和/或调用任何会修改它们的成员函数的其他函数。

```

int squared_calc() const;

// We can assume this function doesn't change logical state, and doesn't call
// set_val(). It may or may not call calc() or squared_calc().
void bad_func() const;

// We can assume this function changes logical state, and may or may not call
// calc(), squared_calc(), or bad_func().
void set_val(int v);
};

```

Due to **const** rules, these assumptions will in fact be enforced by the compiler.

```

// ConstMemberFunctions.cpp

ConstMemberFunctions::ConstMemberFunctions(int v /* = 0 */)
    : cache(0), val(v), state_changed(true) {}

// Our assumption was correct.
int ConstMemberFunctions::calc() const {
    if (state_changed) {
        cache = 3 * val;
        state_changed = false;
    }

    return cache;
}

// Our assumption was correct.
int ConstMemberFunctions::squared_calc() const {
    return calc() * calc();
}

// Our assumption was incorrect.
// Function fails to compile, due to `this` losing qualifiers.
void ConstMemberFunctions::bad_func() const {
    set_val(863);
}

// Our assumption was correct.
void ConstMemberFunctions::set_val(int v) {
    if (v != val) {
        val = v;
        state_changed = true;
    }
}

```

const Function Parameters:

- Any function with one or more parameters which are **const** can be assumed to have intent to read those parameters, and:
 - Shall not modify those parameters, or call any member functions that would modify them.
 - Shall not pass those parameters to any *other* function which would modify them and/or call any member functions that would modify them.
- Conversely, any function with one or more parameters which aren't **const** can be assumed to have intent to modify those parameters, and:
 - May or may not modify those parameters, or call any member functions which would modify them.
 - May or may not pass those parameters to other functions which would modify them and/or call any member functions that would modify them.

这可以用来对参数在传递给任意函数后状态做出假设，即使没有看到该函数的定义。

```
// function_parameter.h

// 我们可以假设 c 没有被修改（且没有调用 c.set_val()），并且没有传递给 non_qualified_function_parameter()。如果传递给 one_const_one_not()，它是第一个参数。

void const_function_parameter(const ConstMemberFunctions& c);

// 我们可以假设 c 被修改和/或调用了 c.set_val()，并且可能传递也可能不传递给这些函数中的任意一个。如果传递给 one_const_one_not，它可能是任一参数。
void non_qualified_function_parameter(ConstMemberFunctions& c);

// 我们可以假设：
// l 没有被修改，且不会调用 l.set_val()。
// l 可能传递也可能不传递给 const_function_parameter()。
// r 被修改，和/或可能调用 r.set_val()。
// r 可能传递也可能不传递给前述任一函数。
void one_const_one_not(const ConstMemberFunctions& l, ConstMemberFunctions& r);

// 我们可以假设 c 没有被修改（且没有调用 c.set_val()），并且没有传递给 non_qualified_function_parameter()。如果传递给 one_const_one_not()，它是第一个参数。

void bad_parameter(const ConstMemberFunctions& c);
```

由于const规则，这些假设实际上将由编译器强制执行。

```
// function_parameter.cpp

// 我们的假设是正确的。
void const_function_parameter(const ConstMemberFunctions& c) {    std::cout << "使用当前值，输出为: " << c.calc() << "平方后为: " << c.squared_calc()
    << std::endl;
}

// 我们的假设是正确的。
void 非限定函数参数(ConstMemberFunctions& c) {
    c.set_val(42);
    std::cout << "对于值42，输出是: " << c.calc() << "平方后是: " << c.squared_calc()
    << std::endl;
}

// 我们的假设是正确的，以最丑陋的方式。
// 注意，const正确性并不阻止封装被故意破坏，// 它只是防止代码在不需要写访问权限时拥有写权限。
void 一个const一个非const(const ConstMemberFunctions& l, ConstMemberFunctions& r) {
    // 这里我们就直接无视访问修饰符和常识吧。
    struct 马基雅维利 {
        int val;
        int unimportant;
        bool state_changed;
    };
    reinterpret_cast<马基雅维利&>(r).val = l.calc();
    reinterpret_cast<马基雅维利&>(r).state_changed = true;

    const_function_parameter(l);
    const_function_parameter(r);
}
```

This can be used to make assumptions about the state of the parameters after being passed to any given function, even without seeing the definition of that function.

```
// function_parameter.h

// We can assume that c isn't modified (and c.set_val() isn't called), and isn't passed
// to non_qualified_function_parameter(). If passed to one_const_one_not(), it is the first
// parameter.
void const_function_parameter(const ConstMemberFunctions& c);

// We can assume that c is modified and/or c.set_val() is called, and may or may not be passed
// to any of these functions. If passed to one_const_one_not, it may be either parameter.
void non_qualified_function_parameter(ConstMemberFunctions& c);

// We can assume that:
// l is not modified, and l.set_val() won't be called.
// l may or may not be passed to const_function_parameter().
// r is modified, and/or r.set_val() may be called.
// r may or may not be passed to either of the preceding functions.
void one_const_one_not(const ConstMemberFunctions& l, ConstMemberFunctions& r);

// We can assume that c isn't modified (and c.set_val() isn't called), and isn't passed
// to non_qualified_function_parameter(). If passed to one_const_one_not(), it is the first
// parameter.
void bad_parameter(const ConstMemberFunctions& c);
```

Due to `const` rules, these assumptions will in fact be enforced by the compiler.

```
// function_parameter.cpp

// Our assumption was correct.
void const_function_parameter(const ConstMemberFunctions& c) {
    std::cout << "With the current value, the output is: " << c.calc() << '\n'
        << "If squared, it's: " << c.squared_calc()
        << std::endl;
}

// Our assumption was correct.
void non_qualified_function_parameter(ConstMemberFunctions& c) {
    c.set_val(42);
    std::cout << "For the value 42, the output is: " << c.calc() << '\n'
        << "If squared, it's: " << c.squared_calc()
        << std::endl;
}

// Our assumption was correct, in the ugliest possible way.
// Note that const correctness doesn't prevent encapsulation from intentionally being broken,
// it merely prevents code from having write access when it doesn't need it.
void one_const_one_not(const ConstMemberFunctions& l, ConstMemberFunctions& r) {
    // Let's just punch access modifiers and common sense in the face here.
    struct Machiavelli {
        int val;
        int unimportant;
        bool state_changed;
    };
    reinterpret_cast<Machiavelli&>(r).val = l.calc();
    reinterpret_cast<Machiavelli&>(r).state_changed = true;

    const_function_parameter(l);
    const_function_parameter(r);
}
```

```
// 我们的假设是错误的。  
// 函数无法编译，因为在 c.set_val() 中 this 失去了限定符。  
void bad_parameter(const ConstMemberFunctions& c) {  
    c.set_val(18);  
}
```

虽然可以绕过const正确性，从而破坏这些保证，但这必须由程序员有意为之（就像上面用Machiavelli破坏封装一样），并且很可能导致未定义行为。

```
class DealBreaker : public ConstMemberFunctions {  
public:  
    DealBreaker(int v = 0);  
  
    // 一个不祥的名字，但它是 const...  
    void no_guarantees() const;  
}  
  
DealBreaker::DealBreaker(int v /* = 0 */) : ConstMemberFunctions(v) {}  
  
// 我们的假设是错误的。  
// const_cast 移除 const 属性，使编译器认为我们知道自己在做什么。  
void DealBreaker::no_guarantees() const {  
    const_cast<DealBreaker*>(this)->set_val(823);  
}  
  
// ...  
  
const DealBreaker d(50);  
d.no_guarantees(); // 未定义行为：d 确实是 const，可能会被修改，也可能不会。
```

然而，由于这要求程序员非常明确地告诉编译器他们打算忽略 const 属性，并且在不同编译器间不一致，通常可以安全地假设 const 正确的代码除非另有说明，否则不会这样做。

```
// Our assumption was incorrect.  
// Function fails to compile, due to `this` losing qualifiers in c.set_val().  
void bad_parameter(const ConstMemberFunctions& c) {  
    c.set_val(18);  
}
```

While it is possible to circumvent const correctness, and by extension break these guarantees, this must be done intentionally by the programmer (just like breaking encapsulation with Machiavelli, above), and is likely to cause undefined behaviour.

```
class DealBreaker : public ConstMemberFunctions {  
public:  
    DealBreaker(int v = 0);  
  
    // A foreboding name, but it's const...  
    void no_guarantees() const;  
}  
  
DealBreaker::DealBreaker(int v /* = 0 */) : ConstMemberFunctions(v) {}  
  
// Our assumption was incorrect.  
// const_cast removes const-ness, making the compiler think we know what we're doing.  
void DealBreaker::no_guarantees() const {  
    const_cast<DealBreaker*>(this)->set_val(823);  
}  
  
// ...  
  
const DealBreaker d(50);  
d.no_guarantees(); // Undefined behaviour: d really IS const, it may or may not be modified.
```

However, due to this requiring the programmer to very specifically tell the compiler that they intend to ignore constness, and being inconsistent across compilers, it is generally safe to assume that const correct code will refrain from doing so unless otherwise specified.

第129章：参数包

第129.1节：带参数包的模板

```
template<class ... Types> struct Tuple {};
```

参数包是一个模板参数，可以接受零个或多个模板参数。如果一个模板至少有一个参数包，则称为可变参数模板。

第129.2节：参数包的展开

模式parameter_pack ...被展开为以逗号分隔的parameter_pack的每个参数的替代列表

```
template<class T> // 递归基
void variadic_printer(T last_argument) {
    std::cout << last_argument;
}

template<class T, class ...Args>
void variadic_printer(T first_argument, Args... other_arguments) { std::cout << first_argument << "";
variadic_printer(other_arguments...); // 参数包展开
}
```

上述代码调用variadic_printer(1, 2, 3, "hello");时，输出为

```
1
2
3
h
e
l
l
o
```

Chapter 129: Parameter packs

Section 129.1: A template with a parameter pack

```
template<class ... Types> struct Tuple {};
```

A parameter pack is a template parameter accepting zero or more template arguments. If a template has at least one parameter pack is a *variadic template*.

Section 129.2: Expansion of a parameter pack

The pattern parameter_pack ... is expanded into a list of comma-separated substitutions of parameter_pack with each one of its parameters

```
template<class T> // Base of recursion
void variadic_printer(T last_argument) {
    std::cout << last_argument;
}

template<class T, class ...Args>
void variadic_printer(T first_argument, Args... other_arguments) {
    std::cout << first_argument << "\n";
    variadic_printer(other_arguments...); // Parameter pack expansion
}
```

The code above invoked with variadic_printer(1, 2, 3, "hello"); prints

```
1
2
3
hello
```

第130章：构建系统

C++，像C语言一样，在编译工作流程和构建过程中有着悠久且多样的历史。如今，C++拥有多种流行的构建系统，用于编译程序，有时一个构建系统可以支持多个平台。这里将对几种构建系统进行回顾和分析。

第130.1节：使用CMake生成构建环境

[CMake](#)可以从单一的项目定义生成几乎适用于任何编译器或IDE的构建环境。以下示例将演示如何为跨平台的“Hello World”C++代码添加CMake文件。

CMake文件总是命名为“CMakeLists.txt”，并且应该已经存在于每个项目的根目录（也可能存在于子目录）中。一个基本的CMakeLists.txt文件如下所示：

```
cmake_minimum_required(VERSION 2.4)
project(HelloWorld)
add_executable(HelloWorld main.cpp)
```

可在[Coliru](#)上实时查看。

该文件告诉CMake项目名称、期望的文件版本，以及生成一个名为“HelloWorld”的可执行文件的指令，该可执行文件需要main.cpp。

从命令行已安装的编译器/IDE生成构建环境：

```
> cmake .
```

使用以下命令构建应用程序：

```
> cmake --build .
```

这将根据操作系统和已安装的工具生成系统的默认构建环境。通过使用“源外构建”（out-of-source builds）保持源代码不受任何构建产物的影响：

```
> mkdir build
> cd build
> cmake ..
> cmake --build .
```

CMake 还可以抽象前面示例中的平台 shell 基本命令：

```
> cmake -E make_directory build
> cmake -E chdir build cmake ..
> cmake --build build
```

CMake 包含多个常用构建工具和集成开发环境（IDE）的生成器。要为 Visual Studio 生成 makefile：_____nmake：

```
> cmake -G "NMake Makefiles" ..
> nmake
```

Chapter 130: Build Systems

C++, like C, has a long and varied history regarding compilation workflows and build processes. Today, C++ has various popular build systems that are used to compile programs, sometimes for multiple platforms within one build system. Here, a few build systems will be reviewed and analyzed.

Section 130.1: Generating Build Environment with CMake

[CMake](#) generates build environments for nearly any compiler or IDE from a single project definition. The following examples will demonstrate how to add a CMake file to the cross-platform “Hello World” C++ code.

CMake files are always named “CMakeLists.txt” and should already exist in every project's root directory (and possibly in sub-directories too.) A basic CMakeLists.txt file looks like:

```
cmake_minimum_required(VERSION 2.4)
project(HelloWorld)
add_executable(HelloWorld main.cpp)
```

See it [live on Coliru](#).

This file tells CMake the project name, what file version to expect, and instructions to generate an executable called “HelloWorld” that requires `main.cpp`.

Generate a build environment for your installed compiler/IDE from the command line:

```
> cmake .
```

Build the application with:

```
> cmake --build .
```

This generates the default build environment for the system, depending on the OS and installed tools. Keep source code clean from any build artifacts with use of “out-of-source” builds:

```
> mkdir build
> cd build
> cmake ..
> cmake --build .
```

CMake can also abstract the platform shell's basic commands from the previous example:

```
> cmake -E make_directory build
> cmake -E chdir build cmake ..
> cmake --build build
```

CMake includes [generators](#) for a number of common build tools and IDEs. To generate makefiles for [Visual Studio](#)'s nmake:

```
> cmake -G "NMake Makefiles" ..
> nmake
```

第130.2节：使用GNU make进行编译

介绍

GNU Make（写作make）是一个专门用于自动执行shell命令的程序。GNU Make是Make家族中的一个特定程序。Make在类Unix和类POSIX操作系统中仍然很受欢迎，包括基于Linux内核、Mac OS X和BSD的系统。

GNU Make特别值得注意的是它隶属于GNU项目，而GNU项目又隶属于流行的GNU/Linux操作系统。GNU Make也有兼容版本运行在各种Windows和Mac OS X系统上。它还是一个非常稳定且具有历史意义的版本，依然很受欢迎。正因为这些原因，GNU Make常与C和C++一起被教授。

基本规则

要使用make进行编译，请在项目目录中创建一个Makefile。你的Makefile可以非常简单，如下所示：

Makefile

```
# 设置一些变量以供命令使用
# 首先，我们将编译器设置为g++
CXX=g++

# 然后，我们说我们想用 g++ 推荐的警告选项和一些额外的警告选项进行编译。
CXXFLAGS=-Wall -Wextra -pedantic

# 这是输出文件
EXE=app

SRCS=main.cpp

# 当你在命令行调用 `make` 时，会调用这个“目标”。
# 右边的 $(EXE) 表示 `all` 目标依赖于 `$(EXE)` 目标。
# $(EXE) 会展开为 EXE 变量的内容# 注意：因为这是第一个目标，如果
# 调用 `make` 时没有指定目标，它会成为默认目标

all: $(EXE)

# 这相当于说
# app: $(SRCS)
# $(SRCS) 可以分开写，这意味着该目标依赖于每个文件。
# 注意该目标有一个“方法体”：由制表符缩进的部分（不是四个空格）。
# 当我们构建此目标时，make 将执行以下命令：
# g++ -Wall -Wextra -pedantic -o app main.cpp
# 即编译 main.cpp 并带有警告，输出文件为 ./app
$(EXE): $(SRCS)
@$(CXX) $(CXXFLAGS) -o $@ $(SRCS)# 该目标
```

应与 `all` 目标相反。如果你带参数调用 make，比如 `make clean`，# 则会调用对应的目标。

```
clean:
@rm -f $(EXE)
```

注意：请确保缩进使用的是制表符（tab），而不是四个空格。否则，你会遇到错误：Makefile:10: * 缺少分隔符。停止。**

Section 130.2: Compiling with GNU make

Introduction

The GNU Make (styled `make`) is a program dedicated to the automation of executing shell commands. GNU Make is one specific program that falls under the Make family. Make remains popular among Unix-like and POSIX-like operating systems, including those derived from the Linux kernel, Mac OS X, and BSD.

GNU Make is especially notable for being attached to the GNU Project, which is attached to the popular GNU/Linux operating system. GNU Make also has compatible versions running on various flavors of Windows and Mac OS X. It is also a very stable version with historical significance that remains popular. It is for these reasons that GNU Make is often taught alongside C and C++.

Basic rules

To compile with make, create a Makefile in your project directory. Your Makefile could be as simple as:

Makefile

```
# Set some variables to use in our command
# First, we set the compiler to be g++
CXX=g++

# Then, we say that we want to compile with g++'s recommended warnings and some extra ones.
CXXFLAGS=-Wall -Wextra -pedantic

# This will be the output file
EXE=app

SRCS=main.cpp

# When you call `make` at the command line, this "target" is called.
# The $(EXE) at the right says that the `all` target depends on the `$(EXE)` target.
# $(EXE) expands to be the content of the EXE variable
# Note: Because this is the first target, it becomes the default target if `make` is called without
# target
all: $(EXE)

# This is equivalent to saying
# app: $(SRCS)
# $(SRCS) can be separated, which means that this target would depend on each file.
# Note that this target has a "method body": the part indented by a tab (not four spaces).
# When we build this target, make will execute the command, which is:
# g++ -Wall -Wextra -pedantic -o app main.cpp
# I.E. Compile main.cpp with warnings, and output to the file ./app
$(EXE): $(SRCS)
@$(CXX) $(CXXFLAGS) -o $@ $(SRCS)

# This target should reverse the `all` target. If you call
# make with an argument, like `make clean`, the corresponding target
# gets called.
clean:
@rm -f $(EXE)
```

NOTE: Make absolutely sure that the indentations are with a tab, not with four spaces. Otherwise, you'll get an error of Makefile:10: * missing separator. Stop.**

要从命令行运行此操作，请执行以下步骤：

```
$ cd ~/Path/to/project  
$ make  
$ ls  
app main.cpp Makefile  
  
$ ./app  
你好，世界！
```

```
$ make clean  
$ ls  
main.cpp Makefile
```

增量构建

当你开始有更多文件时，make 变得更加有用。如果你修改了 a.cpp 但没有修改 b.cpp，重新编译 b.cpp 会花费更多时间。

使用以下目录结构：

```
.  
+-- src  
|   +-- a.cpp  
|   +-- a.hpp  
|   +-- b.cpp  
|   +-- b.hpp  
+-- Makefile
```

这将是一个好的 Makefile：

Makefile

```
CXX=g++  
CXXFLAGS=-Wall -Wextra -pedantic  
EXE=app
```

```
SRCS_GLOB=src/*.cpp  
SRCS=$(wildcard $(SRCS_GLOB))  
OBJS=$(SRCS:.cpp=.o)
```

```
all: $(EXE)
```

```
$(EXE): $(OBJS)  
    @$(CXX) -o $@ $(OBJS)
```

```
depend: .depend
```

```
.depend: $(SRCS)  
    @-rm -f ./depend  
    @$(CXX) $(CXXFLAGS) -MM $^>>./depend
```

```
clean:
```

```
    -rm -f $(EXE)  
    -rm $(OBJS)  
    -rm *~  
-rm .depend
```

```
include .depend
```

To run this from the command-line, do the following:

```
$ cd ~/Path/to/project  
$ make  
$ ls  
app main.cpp Makefile
```

```
$ ./app  
Hello World!
```

```
$ make clean  
$ ls  
main.cpp Makefile
```

Incremental builds

When you start having more files, make becomes more useful. What if you edited **a.cpp** but not **b.cpp**? Recompiling **b.cpp** would take more time.

With the following directory structure:

```
.  
+-- src  
|   +-- a.cpp  
|   +-- a.hpp  
|   +-- b.cpp  
|   +-- b.hpp  
+-- Makefile
```

This would be a good Makefile:

Makefile

```
CXX=g++  
CXXFLAGS=-Wall -Wextra -pedantic  
EXE=app
```

```
SRCS_GLOB=src/*.cpp  
SRCS=$(wildcard $(SRCS_GLOB))  
OBJS=$(SRCS:.cpp=.o)
```

```
all: $(EXE)
```

```
$(EXE): $(OBJS)  
    @$(CXX) -o $@ $(OBJS)
```

```
depend: .depend
```

```
.depend: $(SRCS)  
    @-rm -f ./depend  
    @$(CXX) $(CXXFLAGS) -MM $^>>./depend
```

```
clean:
```

```
    -rm -f $(EXE)  
    -rm $(OBJS)  
    -rm *~  
-rm .depend
```

```
include .depend
```

再次注意制表符。这个新的Makefile确保你只重新编译已更改的文件，从而最小化编译时间。

文档

关于make的更多内容，请参阅自由软件基金会的官方文档、[stackoverflow文档](#)以及dmckee在stackoverfl
ow上的详细回答。

第130.3节：使用SCons构建

你可以使用Scons（一种基于Python语言的软件构建工具）构建跨平台的“Hello World” C++代码。

首先，创建一个名为SConstruct的文件（注意SCons默认会查找这个确切名称的文件）。目前，该文件应位于与你的hello.cpp文件同一目录下。在新文件中写入以下内容

```
Program('hello.cpp')
```

现在，在终端运行 scons。你应该会看到类似如下内容

```
$ scons
scons: 正在读取SConscript文件 ...
scons: 完成读取 SConscript 文件。
scons: 正在构建目标...
g++ -o hello.o -c hello.cpp
g++ -o hello hello.o
scons: 目标构建完成。
```

（尽管具体细节会根据您的操作系统和已安装的编译器有所不同）。

Environment 和 Glob 类将帮助您进一步配置构建内容。例如，SConstruct 文件

```
env=Environment(CPPPATH='/usr/include/boost/',
                CPPDEFINES=[],
                LIBS=[],
                SCONS_CXX_STANDARD="c++11"
                )

env.Program('hello', Glob('src/*.cpp'))
```

构建可执行文件 hello，使用 src 目录下的所有 cpp 文件。其 CPPPATH 为 /usr/include/boost，且指定了 C++11 标准。

第130.4节：Autotools (GNU)

介绍

Autotools是一组程序，用于为给定的软件包创建GNU构建系统。它是一套协同工作的工具，用于生成各种构建资源，例如用于GNU Make的Makefile。因此，Autotools可以被视为事实上的构建系统生成器。

一些著名的Autotools程序包括：

- Autoconf
- Automake（不要与make混淆）

一般来说，Autotools旨在生成兼容Unix的脚本和Makefile，以允许以下命令

Again watch the tabs. This new Makefile ensures that you only recompile changed files, minimizing compile time.

Documentation

For more on make, see [the official documentation by the Free Software Foundation](#), the [stackoverflow documentation](#) and dmckee's elaborate answer on [stackoverflow](#).

Section 130.3: Building with SCons

You can build the cross-platform "Hello World" C++ code, using [Scons - A Python-language software construction tool](#).

First, create a file called SConstruct (note that SCons will look for a file with this exact name by default). For now, the file should be in a directory right along your hello.cpp. Write in the new file the line

```
Program('hello.cpp')
```

Now, from the terminal, run scons. You should see something like

```
$ scons
scons: 正在读取 SConscript 文件 ...
scons: 完成读取 SConscript 文件。
scons: 正在构建目标...
g++ -o hello.o -c hello.cpp
g++ -o hello hello.o
scons: 目标构建完成。
```

（尽管具体细节会根据您的操作系统和已安装的编译器有所不同）。

The Environment and Glob classes will help you further configure what to build. E.g., the SConstruct file

```
env=Environment(CPPPATH='/usr/include/boost',
                CPPDEFINES=[],
                LIBS=[],
                SCONS_CXX_STANDARD="c++11"
                )

env.Program('hello', Glob('src/*.cpp'))
```

builds the executable hello, using all cpp files in src. Its CPPPATH is /usr/include/boost and it specifies the C++11 standard.

Section 130.4: Autotools (GNU)

Introduction

The Autotools are a group of programs that create a GNU Build System for a given software package. It is a suite of tools that work together to produce various build resources, such as a Makefile (to be used with GNU Make). Thus, Autotools can be considered a de facto build system generator.

Some notable Autotools programs include:

- Autoconf
- Automake (not to be confused with make)

In general, Autotools is meant to generate the Unix-compatible script and Makefile to allow the following command

构建（以及安装）大多数软件包（在简单情况下）：

```
./configure && make && make install
```

因此，Autotools也与某些包管理器有关，特别是那些附属于符合POSIX标准的操作系统的包管理器。

第130.5节：忍者

介绍

Ninja 构建系统在其项目网站上被描述为“一个专注于速度的小型构建系统”。Ninja 设计上由构建系统文件生成器生成其文件，并采用低级别的构建系统方法，与 CMake 或 Meson 等高级构建系统管理器形成对比。

Ninja 主要使用 C++ 和 Python 编写，最初作为 Chromium 项目中 SCons 构建系统的替代方案创建。

第130.6节：NMAKE（微软程序维护工具）

介绍

NMAKE 是微软开发的命令行工具，主要用于配合 Microsoft Visual Studio 和/或 Visual C++ 命令行工具使用。

NMAKE 是属于 Make 系列的构建系统，但具有某些区别于类 Unix Make 程序的特性，例如支持 Windows 特有的文件路径语法（该语法本身不同于类 Unix 文件路径）。

to build (as well as install) most packages (in the simple case):

```
./configure && make && make install
```

As such, Autotools also has a relationship with certain package managers, especially those that are attached to operating systems that conform to the POSIX Standard(s).

Section 130.5: Ninja

Introduction

The Ninja build system is described by its project website as ["a small build system with a focus on speed."](#) Ninja is designed to have its files generated by build system file generators, and takes a low-level approach to build systems, in contrast to higher-level build system managers like CMake or Meson.

Ninja is primarily written in C++ and Python, and was created as an alternative to the SCons build system for the Chromium project.

Section 130.6: NMAKE (Microsoft Program Maintenance Utility)

Introduction

NMAKE is a command-line utility developed by Microsoft to be used primarily in conjunction with Microsoft Visual Studio and/or the Visual C++ command line tools.

NMAKE is build system that falls under the Make family of build systems, but has certain distinct features that diverge from Unix-like Make programs, such as supporting Windows-specific file path syntax (which itself differs from Unix-style file paths).

第131章：使用 OpenMP 的并发

本主题涵盖了使用 OpenMP 进行 C++ 并发的基础知识。OpenMP 在 OpenMP 标签中有更详细的文档说明。

并行或并发意味着代码的同时执行。

第131.1节：OpenMP：并行区段

此示例说明了如何并行执行代码段的基础知识。

由于OpenMP是内置的编译器特性，它可以在任何支持的编译器上工作，无需包含任何库。如果您想使用OpenMP的任何API功能，您可能需要包含`omp.h`。

示例代码

```
std::cout << "begin ";
// 该pragma语句提示编译器
// 大括号内的内容将作为
// 并行代码段使用OpenMP执行，编译器将
// 生成这部分代码以实现并行执行
#pragma omp parallel sections
{
    // 该pragma语句提示编译器// 这是一个可以与其他代码段并
    // 行执行的代码段，// 单个代码段将由单个线程执行。

    // 注意这里是“section”，而不是上面的“sections”
    #pragma omp section
    {
        std::cout << "hello " << std::endl;
        /* 执行某些操作 */
    }
    #pragma omp section
    {
        std::cout << "world " << std::endl;
        /* 执行某些操作 */
    }
}
// 这行代码将在上述所有部分终止后才会执行
std::cout << "end" << std::endl;
```

输出

此示例产生两种可能的输出，具体取决于操作系统和硬件。输出还展示了这种实现可能出现的“竞态条件”问题。

输出 A	输出 B
begin hello world end	begin world hello end

第131.2节：OpenMP：并行部分

此示例展示如何并行执行代码块

```
std::cout << "begin ";
// 并行部分开始
```

Chapter 131: Concurrency With OpenMP

This topic covers the basics of concurrency in C++ using OpenMP. OpenMP is documented in more detail in the OpenMP tag.

Parallelism or concurrency implies the execution of code at the same time.

Section 131.1: OpenMP: Parallel Sections

This example illustrates the basics of executing sections of code in parallel.

As OpenMP is a built-in compiler feature, it works on any supported compilers without including any libraries. You may wish to include `omp.h` if you want to use any of the openMP API features.

Sample Code

```
std::cout << "begin ";
// This pragma statement hints the compiler that the
// contents within the { } are to be executed in as
// parallel sections using openMP, the compiler will
// generate this chunk of code for parallel execution
#pragma omp parallel sections
{
    // This pragma statement hints the compiler that
    // this is a section that can be executed in parallel
    // with other section, a single section will be executed
    // by a single thread.
    // Note that it is "section" as opposed to "sections" above
    #pragma omp section
    {
        std::cout << "hello " << std::endl;
        /* Do something */
    }
    #pragma omp section
    {
        std::cout << "world " << std::endl;
        /* Do something */
    }
}
// This line will not be executed until all the
// sections defined above terminates
std::cout << "end" << std::endl;
```

Outputs

This example produces 2 possible outputs and is dependent on the operating system and hardware. The output also illustrates a **race condition** problem that would occur from such an implementation.

OUTPUT A	OUTPUT B
begin hello world end	begin world hello end

Section 131.2: OpenMP: Parallel Sections

This example shows how to execute chunks of code in parallel

```
std::cout << "begin ";
// Start of parallel sections
```

```

#pragma omp 并行区块
{
    // 并行执行这些区块
    #pragma omp 区块
    {
        ... 执行 某些操作 ...
        std::cout << "hello ";
    }
    #pragma omp section
    {
        ... 执行 某些操作 ...
        std::cout << "world ";
    }
    #pragma omp section
    {
        ... 执行 某些操作 ...
        std::cout << "forever ";
    }
}
// 并行区块结束
std::cout << "end";

```

输出

- 开始 hello world forever 结束
- 开始 world hello forever 结束
- 开始 hello forever world 结束
- 开始 forever hello world 结束

由于执行顺序不确定，您可能会看到上述任意一种输出。

第131.3节：OpenMP：并行For循环

此示例展示了如何将循环划分为相等的部分并并行执行。

```

// 将元素向量拆分为 element.size() / 线程数量
// 并为每个线程分配该范围。
#pragma omp parallel for
for (大小_t i = 0; i < element.大小(); ++i)
    element[i] = ...

// 示例分配（每个线程100个元素）
// 线程 1 : 0 ~ 99
// 线程 2 : 100 ~ 199
// 线程 3 : 200 ~ 299
// ...

// 继续处理
// 仅当所有线程完成其分配的
// 循环任务时
...

```

*请特别注意不要修改并行 for 循环中使用的向量大小，因为**分配的范围索引**不会自动更新。

第131.4节：OpenMP：并行收集 / 归约

此示例说明了如何使用std::vector和OpenMP执行归约或收集的概念。

```

#pragma omp parallel sections
{
    // Execute these sections in parallel
    #pragma omp section
    {
        ... do something ...
        std::cout << "hello ";
    }
    #pragma omp section
    {
        ... do something ...
        std::cout << "world ";
    }
    #pragma omp section
    {
        ... do something ...
        std::cout << "forever ";
    }
}
// end of parallel sections
std::cout << "end";

```

Output

- begin hello world forever end
- begin world hello forever end
- begin hello forever world end
- begin forever hello world end

As execution order is not guaranteed, you may observe any of the above output.

Section 131.3: OpenMP: Parallel For Loop

This example shows how to divide a loop into equal parts and execute them in parallel.

```

// Splits element vector into element.size() / Thread Qty
// and allocate that range for each thread.
#pragma omp parallel for
for (size_t i = 0; i < element.size(); ++i)
    element[i] = ...

// Example Allocation (100 element per thread)
// Thread 1 : 0 ~ 99
// Thread 2 : 100 ~ 199
// Thread 3 : 200 ~ 299
// ...

// Continue process
// Only when all threads completed their allocated
// loop job
...

```

*Please take extra care to not modify the size of the vector used in parallel for loops as **allocated range indices** doesn't update automatically.

Section 131.4: OpenMP: Parallel Gathering / Reduction

This example illustrates a concept to perform reduction or gathering using std::vector and OpenMP.

假设我们有一个场景，需要多个线程帮助我们生成一堆数据，int 在这里用于简化，可以替换为其他数据类型。

当你需要合并从线程获得的结果以避免段错误或内存访问违规，并且不想使用库或自定义同步容器库时，这尤其有用。

```
// 主线程向量
// 我们想要一个从线程收集结果的向量
std::vector<int> Master;

// 提示编译器并行化这段 {} 代码
// 使用所有可用线程（通常与逻辑处理器数量相同）
#pragma omp parallel
{
    // 在此区域，你可以为每个
    // 从线程编写任何代码，这里是一个向量来保存它们各自的结果
    // 我们不必担心启动了多少线程，也不必担心是否需要
    // 重复声明。
    std::vector<int> Slave;

    // 告诉编译器使用为此并行区域分配的所有线程// 来分部分执行此循环。实际负载约为 1000000
    // 线程数量// nowait 关键字告诉编译器，从线程不必等待所有其他从线程完成此 for 循环任务

#pragma omp for nowait
for (size_t i = 0; i < 1000000; ++i
{
    /* 做某事 */
    ...
    Slave.push_back(...);
}

// 完成其部分工作的从线程
// 将逐个线程执行此操作
// 临界区确保任何时候只有0个或1个线程执行
// {} 内的代码
#pragma omp critical
{
    // 将从线程合并到主线程
    // 使用移动迭代器，避免复制，除非
    // 你想在此部分之后继续使用它
    Master.insert(Master.end(),
    std::make_move_iterator(Slave.begin()),
        std::make_move_iterator(Slave.end())));
}

// 尽情使用 Master 向量
...
```

Supposed we have a scenario where we want multiple threads to help us generate a bunch of stuff, int is used here for simplicity and can be replaced with other data types.

This is particularly useful when you need to merge results from slaves to avoid segment faults or memory access violations and do not wish to use libraries or custom sync container libraries.

```
// The Master vector
// We want a vector of results gathered from slave threads
std::vector<int> Master;

// Hint the compiler to parallelize this {} of code
// with all available threads (usually the same as logical processor qty)
#pragma omp parallel
{
    // In this area, you can write any code you want for each
    // slave thread, in this case a vector to hold each of their results
    // We don't have to worry about how many threads were spawn or if we need
    // to repeat this declaration or not.
    std::vector<int> Slave;

    // Tell the compiler to use all threads allocated for this parallel region
    // to perform this loop in parts. Actual load appx = 1000000 / Thread Qty
    // The nowait keyword tells the compiler that the slave threads don't
    // have to wait for all other slaves to finish this for loop job
#pragma omp for nowait
for (size_t i = 0; i < 1000000; ++i
{
    /* Do something */
    ...
    Slave.push_back(...);
}

// Slaves that finished their part of the job
// will perform this thread by thread one at a time
// critical section ensures that only 0 or 1 thread performs
// the {} at any time
#pragma omp critical
{
    // Merge slave into master
    // use move iterators instead, avoid copy unless
    // you want to use it for something else after this section
    Master.insert(Master.end(),
        std::make_move_iterator(Slave.begin()),
        std::make_move_iterator(Slave.end()));
}

// Have fun with Master vector
...
```

第132章：资源管理

在C和C++中，最难做的事情之一是资源管理。幸运的是，在C++中，我们有许多方法来设计程序中的资源管理。本文希望解释一些用于管理已分配资源的惯用法和方法。

第132.1节：资源获取即初始化

资源获取即初始化（RAII）是资源管理中常见的惯用法。以动态内存为例，它使用智能指针来实现资源管理。使用RAII时，获取的资源会立即被赋予智能指针或等效资源管理器的所有权。资源只能通过该管理器访问，因此管理器可以跟踪各种操作。例如，`std::auto_ptr`会在其超出作用域或被删除时自动释放对应的资源。

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

int main() {
{
auto_ptr ap(new int(5)); // 动态内存是资源
    cout << *ap << endl; // 输出5
} // auto_ptr被销毁，其资源自动释放
}

版本 ≥ C++11
```

`std::auto_ptr`的主要问题是它不能在不转移所有权的情况下被复制：

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

int main() {
auto_ptr ap1(new int(5));
    cout << *ap1 << endl; // 输出5
auto_ptr ap2(ap1); // 从ap1复制ap2；所有权现在转移到ap2
    cout << *ap2 << endl; // 输出5
    cout << ap1 == nullptr << endl; // 输出 1；ap1 已经失去资源所有权
}
```

由于这些奇怪的拷贝语义，`std::auto_ptr` 不能用于容器等场景。它这样设计的原因是为了防止内存被重复释放：如果有两个 `auto_ptr`s 拥有同一个资源的所有权，当它们被销毁时都会尝试释放该资源。重复释放已经释放的资源通常会导致问题，因此必须避免这种情况。然而，`std::shared_ptr` 有一种方法可以避免这个问题，同时在拷贝时不转移所有权：

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

int main() {
shared_ptr sp2;
{
shared_ptr sp1(new int(5)); // 将所有权赋给 sp1
    cout << *sp1 << endl; // 输出 5
sp2 = sp1; // 从 sp1 拷贝 sp2；两者都拥有资源的所有权
}
```

Chapter 132: Resource Management

One of the hardest things to do in C and C++ is resource management. Thankfully, in C++, we have many ways to go about designing resource management in our programs. This article hopes to explain some of the idioms and methods used to manage allocated resources.

Section 132.1: Resource Acquisition Is Initialization

Resource Acquisition Is Initialization (RAII) is a common idiom in resource management. In the case of dynamic memory, it uses smart pointers to accomplish resource management. When using RAII, an acquired resource is immediately given ownership to a smart pointer or equivalent resource manager. The resource is only accessed through this manager, so the manager can keep track of various operations. For example, `std::auto_ptr` automatically frees its corresponding resource when it falls out of scope or is otherwise deleted.

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

int main() {
{
    auto_ptr ap(new int(5)); // dynamic memory is the resource
    cout << *ap << endl; // prints 5
} // auto_ptr is destroyed, its resource is automatically freed
}

Version ≥ C++11
```

`std::auto_ptr`'s main problem is that it can't copied without transferring ownership:

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

int main() {
    auto_ptr ap1(new int(5));
    cout << *ap1 << endl; // prints 5
    auto_ptr ap2(ap1); // copy ap2 from ap1; ownership now transfers to ap2
    cout << *ap2 << endl; // prints 5
    cout << ap1 == nullptr << endl; // prints 1; ap1 has lost ownership of resource
}
```

Because of these weird copy semantics, `std::auto_ptr` can't be used in containers, among other things. The reason it does this is to prevent deleting memory twice: if there are two `auto_ptr`s with ownership of the same resource, they both try to free it when they're destroyed. Freeing an already freed resource can generally cause problems, so it is important to prevent it. However, `std::shared_ptr` has a method to avoid this while not transferring ownership when copying:

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

int main() {
    shared_ptr sp2;
{
    shared_ptr sp1(new int(5)); // give ownership to sp1
    cout << *sp1 << endl; // prints 5
    sp2 = sp1; // copy sp2 from sp1; both have ownership of resource
}
```

```

cout << *sp1 << endl; // 输出 5
cout << *sp2 << endl; // 输出 5
} // sp1 超出作用域并被销毁; sp2 独自拥有资源
cout << *sp2 << endl;
} // sp2 超出作用域; 没有任何所有权, 资源被释放

```

第132.2节：互斥锁与线程安全

当多个线程尝试访问同一资源时，可能会发生问题。举一个简单的例子，假设我们有一个线程，它会将一个变量加一。它的操作是先读取变量的值，加一，然后再将结果存回去。假设我们将该变量初始化为1，然后创建两个这样的线程实例。在两个线程都执行完毕后，直觉上认为该变量的值应该是3。然而，下面的表格展示了可能出现的问题：

线程1	线程2
时间步骤1 从变量中读取1	
时间步骤2	从变量中读取1
时间步骤3 将1加1得到2	
时间步骤4	将1加1得到2
时间步骤5 将2存入变量	
时间步骤6	将2存入变量

如您所见，操作结束时，变量中是2，而不是3。原因是线程2在线程1完成更新之前读取了该变量。解决方案？互斥锁。

互斥锁（mutex，mutual exclusion 的合成词）是一种资源管理对象，旨在解决这类问题。当线程想要访问资源时，它会“获取”该资源的互斥锁。访问完资源后，线程会“释放”互斥锁。在互斥锁被获取期间，所有尝试获取该互斥锁的调用都会阻塞，直到互斥锁被释放。为了更好地理解，可以把互斥锁想象成超市的排队队伍：

线程通过尝试获取互斥锁进入排队，然后等待前面的线程完成，接着使用资源，最后通过释放互斥锁离开队伍。如果每个人都试图同时访问资源，场面将会一片混乱。

版本 ≥ C++11

std::mutex 是 C++11 对互斥锁的实现。

```

#include <thread>
#include <mutex>
#include <iostream>
using namespace std;

void add_1(int& i, const mutex& m) { // 在线程中运行的函数
    m.lock();
    i += 1;
    m.unlock();
}

int main() {
    int var = 1;
    mutex m;

    cout << var << endl; // 输出 1

    thread t1(add_1, var, m); // 创建带参数的线程
    thread t2(add_1, var, m); // 创建另一个线程
    t1.join(); t2.join(); // 等待两个线程完成
}

```

```

cout << *sp1 << endl; // prints 5
cout << *sp2 << endl; // prints 5
} // sp1 goes out of scope and is destroyed; sp2 has sole ownership of resource
cout << *sp2 << endl;
} // sp2 goes out of scope; nothing has ownership, so resource is freed

```

Section 132.2: Mutexes & Thread Safety

Problems may happen when multiple threads try to access a resource. For a simple example, suppose we have a thread that adds one to a variable. It does this by first reading the variable, adding one to it, then storing it back. Suppose we initialize this variable to 1, then create two instances of this thread. After both threads finish, intuition suggests that this variable should have a value of 3. However, the below table illustrates what might go wrong:

Thread 1	Thread 2
Time Step 1 Read 1 from variable	
Time Step 2	Read 1 from variable
Time Step 3 Add 1 plus 1 to get 2	
Time Step 4	Add 1 plus 1 to get 2
Time Step 5 Store 2 into variable	
Time Step 6	Store 2 into variable

As you can see, at the end of the operation, 2 is in the variable, instead of 3. The reason is that Thread 2 read the variable before Thread 1 was done updating it. The solution? Mutexes.

A mutex (portmanteau of **mutual exclusion**) is a resource management object designed to solve this type of problem. When a thread wants to access a resource, it "acquires" the resource's mutex. Once it is done accessing the resource, the thread "releases" the mutex. While the mutex is acquired, all calls to acquire the mutex will not return until the mutex is released. To better understand this, think of a mutex as a waiting line at the supermarket: the threads go into line by trying to acquire the mutex and then waiting for the threads ahead of them to finish up, then using the resource, then stepping out of line by releasing the mutex. There would be complete pandemonium if everybody tried to access the resource at once.

Version ≥ C++11

std::mutex is C++11's implementation of a mutex.

```

#include <thread>
#include <mutex>
#include <iostream>
using namespace std;

void add_1(int& i, const mutex& m) { // function to be run in thread
    m.lock();
    i += 1;
    m.unlock();
}

int main() {
    int var = 1;
    mutex m;

    cout << var << endl; // prints 1

    thread t1(add_1, var, m); // create thread with arguments
    thread t2(add_1, var, m); // create another thread
    t1.join(); t2.join(); // wait for both threads to finish
}

```

```
cout << var << endl; // 输出 3  
}
```

```
cout << var << endl; // prints 3  
}
```

第133章：存储类说明符

存储类说明符是可以用于声明的关键字。它们不影响声明的类型，但通常会修改实体的存储方式。

第133.1节：extern

根据上下文，`extern` 存储类说明符可以以下三种方式之一修饰声明：

1. 它可以用于声明变量而不定义变量。通常，这用于头文件中声明将在单独实现文件中定义的变量。

```
// 全局作用域
int x;           // 定义；x 将被默认初始化
extern int y;    // 声明；y 在其他地方定义，通常是另一个翻译单元
extern int z = 42; // 定义；此处“extern”无效（编译器可能会警告）
```

2. 即使是`const`或`constexpr`，否则也会为命名空间作用域的变量提供外部链接导致它具有内部链接。

```
// 全局作用域
const int w = 42;          // C++中为内部链接；C中为外部链接
static const int x = 42;    // C++和C中均为内部链接
extern const int y = 42;    // C++和C中均为外部链接
namespace {
    extern const int z = 42; // 但是，由于它在匿名命名空间中，具有内部链接
}
```

3. 如果之前已用链接声明了块作用域变量，则重新声明该变量。否则，声明一个具有链接的新变量，该变量是最近封闭命名空间的成员。

```
// 全局作用域
namespace {
    int x = 1;
    struct C {
        int x = 2;
        void f() {
            extern int x;           // 重新声明命名空间作用域的x
            std::cout << x << " "; // 因此，这里打印的是1，而不是2}
        }
    };
    void g() {
        extern int y; // y 具有外部链接；指向在其他地方定义的全局 y
    }
}
```

函数也可以声明为 `extern`，但这没有实际效果。它通常用作提示读者此处声明的函数是在另一个翻译单元中定义的。例如：

```
void f();          // 通常是前向声明；f 在此翻译单元稍后定义
extern void g(); // 通常不是前向声明；g 在另一个翻译单元定义
```

Chapter 133: Storage class specifiers

Storage class specifiers are keywords that can be used in declarations. They do not affect the type of the declaration, but typically modify the way in which the entity is stored.

Section 133.1: extern

The `extern` storage class specifier can modify a declaration in one of the three following ways, depending on context:

1. It can be used to declare a variable without defining it. Typically, this is used in a header file for a variable that will be defined in a separate implementation file.

```
// global scope
int x;           // definition; x will be default-initialized
extern int y;    // declaration; y is defined elsewhere, most likely another TU
extern int z = 42; // definition; "extern" has no effect here (compiler may warn)
```

2. It gives external linkage to a variable at namespace scope even if `const` or `constexpr` would have otherwise caused it to have internal linkage.

```
// global scope
const int w = 42;          // internal linkage in C++; external linkage in C
static const int x = 42;    // internal linkage in both C++ and C
extern const int y = 42;    // external linkage in both C++ and C
namespace {
    extern const int z = 42; // however, this has internal linkage since
                           // it's in an unnamed namespace
}
```

3. It redeclares a variable at block scope if it was previously declared with linkage. Otherwise, it declares a new variable with linkage, which is a member of the nearest enclosing namespace.

```
// global scope
namespace {
    int x = 1;
    struct C {
        int x = 2;
        void f() {
            extern int x;           // redeclares namespace-scope x
            std::cout << x << '\n'; // therefore, this prints 1, not 2
        }
    };
    void g() {
        extern int y; // y has external linkage; refers to global y defined elsewhere
    }
}
```

A function can also be declared `extern`, but this has no effect. It is usually used as a hint to the reader that a function declared here is defined in another translation unit. For example:

```
void f();          // typically a forward declaration; f defined later in this TU
extern void g(); // typically not a forward declaration; g defined in another TU
```

在上述代码中，如果将 `f` 改为 `extern`，将 `g` 改为非 `extern`，这不会影响程序的正确性或语义，但可能会让代码的读者感到困惑。

第 133.2 节：register

版本 < C++17

一种存储类说明符，提示编译器某变量将被频繁使用。“register”一词与编译器可能选择将该变量存储在 CPU 寄存器中以减少访问时钟周期数有关。从 C++11 开始，该关键字被弃用。

```
register int i = 0;
while (i < 100) {
    f(i);
    int g = i*i;
    i += h(i, g);
}
```

局部变量和函数参数都可以声明为 `register`。与 C 语言不同，C++ 对 `register` 变量的操作没有任何限制。例如，获取 `register` 变量的地址是有效的，但这可能会阻止编译器将该变量实际存储在寄存器中。

版本 ≥ C++17

关键字 `register` 未使用且被保留。使用关键字 `register` 的程序是格式错误的。

In the above code, if `f` were changed to `extern` and `g` to non-`extern`, it would not affect the correctness or semantics of the program at all, but would likely confuse the reader of the code.

Section 133.2: register

Version < C++17

A storage class specifier that hints to the compiler that a variable will be heavily used. The word "register" is related to the fact that a compiler might choose to store such a variable in a CPU register so that it can be accessed in fewer clock cycles. It was deprecated starting in C++11.

```
register int i = 0;
while (i < 100) {
    f(i);
    int g = i*i;
    i += h(i, g);
}
```

Both local variables and function parameters may be declared `register`. Unlike C, C++ does not place any restrictions on what can be done with a `register` variable. For example, it is valid to take the address of a `register` variable, but this may prevent the compiler from actually storing such a variable in a register.

Version ≥ C++17

The keyword `register` is unused and reserved. A program that uses the keyword `register` is ill-formed.

第 133.3 节：static

`static` 存储类说明符有三种不同的含义。

1. 给在命名空间作用域声明的变量或函数赋予内部链接。

```
// 内部函数；无法链接
static double semiperimeter(double a, double b, double c) {
    return (a + b + c)/2.0;
}
// 导出给客户端
double area(double a, double b, double c) {
    const double s = semiperimeter(a, b, c);
    return sqrt(s*(s-a)*(s-b)*(s-c));
}
```

2. 声明变量具有静态存储持续时间（除非它是 `thread_local`）。命名空间作用域的变量隐式为静态。静态局部变量只初始化一次，即第一次控制流经过其定义时，并且在每次离开其作用域时不会被销毁。

```
void f() {
    static int count = 0;
    std::cout << "f 已被调用 " << ++count << " 次了";}
```

3. 当用于类成员的声明时，声明该成员为静态成员。

```
struct S {
    static S* create() {
```

Section 133.3: static

The `static` storage class specifier has three different meanings.

1. Gives internal linkage to a variable or function declared at namespace scope.

```
// internal function; can't be linked to
static double semiperimeter(double a, double b, double c) {
    return (a + b + c)/2.0;
}
// exported to client
double area(double a, double b, double c) {
    const double s = semiperimeter(a, b, c);
    return sqrt(s*(s-a)*(s-b)*(s-c));
}
```

2. Declares a variable to have static storage duration (unless it is `thread_local`)。Namespace-scope variables are implicitly static. A static local variable is initialized only once, the first time control passes through its definition, and is not destroyed every time its scope is exited.

```
void f() {
    static int count = 0;
    std::cout << "f has been called " << ++count << " times so far\n";}
```

3. When applied to the declaration of a class member, declares that member to be a static member.

```
struct S {
    static S* create() {
```

```

        return new S;
    }
};

int main() {
    S* s = S::create();
}

```

注意，对于类的静态数据成员，规则2和3同时适用：`static`关键字既将成员变为静态数据成员，也使其成为具有静态存储期的变量。

第133.4节：auto

版本 ≤ C++03

声明变量具有自动存储期。此声明是多余的，因为自动存储期在块作用域中已是默认，且`auto`说明符不允许在命名空间作用域使用。

```

void f() {
    auto int x; // 等同于: int x;
    auto y;     // 在C++中非法；在C89中合法
}
auto int z; // 非法：命名空间作用域变量不能是自动存储类型

```

在 C++11 中，`auto` 的含义完全改变，不再是存储类说明符，而是用于类型推断。

第 133.5 节：mutable

一种说明符，可应用于类的非静态、非引用数据成员的声明。即使对象是 `const`，类的 `mutable` 成员也不是 `const`。

```

class C {
    int x;
    mutable int times_accessed;
public:
C(): x(0), times_accessed(0) {
}
int get_x() const {
    ++times_accessed; // 合法：const 成员函数可以修改 mutable 数据成员
    return x;
}
void set_x(int x) {
    ++times_accessed;
    this->x = x;
}
};

```

版本 ≥ C++11

C++11 中为 `mutable` 添加了第二个含义。当它跟在 `lambda` 的参数列表后时，会抑制 `lambda` 函数调用运算符上的隐式 `const`。因此，`mutable lambda` 可以修改通过值捕获的实体的值。详情见 `mutable lambda`。

```

std::vector<int> my_iota(int start, int count) {
    std::vector<int> result(count);
    std::generate(result.begin(), result.end(),
                 [start]() mutable { return start++; });
    return result;
}

```

```

        return new S;
    }
};

int main() {
    S* s = S::create();
}

```

Note that in the case of a static data member of a class, both 2 and 3 apply simultaneously: the `static` keyword both makes the member into a static data member and makes it into a variable with static storage duration.

Section 133.4: auto

Version ≤ C++03

Declares a variable to have automatic storage duration. It is redundant, since automatic storage duration is already the default at block scope, and the `auto` specifier is not allowed at namespace scope.

```

void f() {
    auto int x; // equivalent to: int x;
    auto y;     // illegal in C++; legal in C89
}
auto int z; // illegal: namespace-scope variable cannot be automatic

```

In C++11, `auto` changed meaning completely, and is no longer a storage class specifier, but is instead used for type deduction.

Section 133.5: mutable

A specifier that can be applied to the declaration of a non-static, non-reference data member of a class. A `mutable` member of a class is not `const` even when the object is `const`.

```

class C {
    int x;
    mutable int times_accessed;
public:
C(): x(0), times_accessed(0) {
}
int get_x() const {
    ++times_accessed; // ok: const member function can modify mutable data member
    return x;
}
void set_x(int x) {
    ++times_accessed;
    this->x = x;
}
};

```

Version ≥ C++11

A second meaning for `mutable` was added in C++11. When it follows the parameter list of a `lambda`, it suppresses the implicit `const` on the `lambda`'s function call operator. Therefore, a `mutable lambda` can modify the values of entities captured by copy. See `mutable lambdas` for more details.

```

std::vector<int> my_iota(int start, int count) {
    std::vector<int> result(count);
    std::generate(result.begin(), result.end(),
                 [start]() mutable { return start++; });
    return result;
}

```

}

注意, `mutable` 在这种用法中并不是存储类说明符, 而是用来形成可变的lambda表达式。

}

Note that `mutable` is *not* a storage class specifier when used this way to form a mutable lambda.

第134章：链接规范

链接规范告诉编译器以一种允许它们与用另一种语言（如C）编写的声明链接在一起的方式来编译声明。

第134.1节：类Unix操作系统的信号处理程序

由于信号处理程序将由内核使用C调用约定调用，我们必须告诉编译器在编译该函数时使用C调用约定。

```
volatile sig_atomic_t death_signal = 0;
extern "C" void cleanup(int signum) {
    death_signal = signum;
}
int main() {
bind(...);
listen(...);
    signal(SIGTERM, cleanup);
    while (int fd = accept(...)) {
        if (fd == -1 && errno == EINTR && death_signal) {printf(
            "捕获信号 %d; 正在关闭", death_signal);break;
        }
        // ...
    }
}
```

Chapter 134: Linkage specifications

A linkage specification tells the compiler to compile declarations in a way that allows them to be linked together with declarations written in another language, such as C.

Section 134.1: Signal handler for Unix-like operating system

Since a signal handler will be called by the kernel using the C calling convention, we must tell the compiler to use the C calling convention when compiling the function.

```
volatile sig_atomic_t death_signal = 0;
extern "C" void cleanup(int signum) {
    death_signal = signum;
}
int main() {
bind(...);
listen(...);
    signal(SIGTERM, cleanup);
    while (int fd = accept(...)) {
        if (fd == -1 && errno == EINTR && death_signal) {
            printf("Caught signal %d; shutting down\n", death_signal);
            break;
        }
        // ...
    }
}
```

第134.2节：使C库头文件兼容C++

通常可以将C库头文件包含到C++程序中，因为大多数声明在C和C++中都是有效的。

例如，考虑以下foo.h：

```
typedef struct Foo {
    int bar;
} Foo;
Foo make_foo(int);
```

函数make_foo的定义是单独编译并以目标文件形式与头文件一起分发的。

一个C++程序可以#include <foo.h>，但编译器不会知道make_foo函数被定义为C语言符号，可能会尝试用修饰过的名称查找它，结果找不到。即使能在库中找到make_foo的定义，不是所有平台都对C和C++使用相同的调用约定，C++编译器调用make_foo时会使用C++调用约定，如果make_foo期望用C调用约定调用，可能会导致段错误。

解决这个问题的方法是将头文件中几乎所有的声明都包裹在一个extern "C"块中。

```
#ifdef __cplusplus
extern "C" {
#endif

typedef struct Foo {
    int bar;
} Foo;
Foo make_foo(int);
```

Section 134.2: Making a C library header compatible with C++

A C library header can usually be included into a C++ program, since most declarations are valid in both C and C++. For example, consider the following foo.h:

```
typedef struct Foo {
    int bar;
} Foo;
Foo make_foo(int);
```

The definition of make_foo is separately compiled and distributed with the header in object form.

A C++ program can #include <foo.h>, but the compiler will not know that the make_foo function is defined as a C symbol, and will probably try to look for it with a mangled name, and fail to locate it. Even if it can find the definition of make_foo in the library, not all platforms use the same calling conventions for C and C++, and the C++ compiler will use the C++ calling convention when calling make_foo, which is likely to cause a segmentation fault if make_foo is expecting to be called with the C calling convention.

The way to remedy this problem is to wrap almost all the declarations in the header in an `extern "C"` block.

```
#ifdef __cplusplus
extern "C" {
#endif

typedef struct Foo {
    int bar;
} Foo;
Foo make_foo(int);
```

```
#ifdef __cplusplus  
} /* extern C"块结束 */  
#endif
```

现在当foo.h被C程序包含时，它只是普通的声明，但当foo.h被C++程序包含时，make_foo会在一个extern "C"块内，编译器会知道查找未修饰的名称并使用C调用约定。

```
#ifdef __cplusplus  
} /* end of "extern C" block */  
#endif
```

Now when `foo.h` is included from a C program, it will just appear as ordinary declarations, but when `foo.h` is included from a C++ program, `make_foo` will be inside an `extern "C"` block and the compiler will know to look for an unmangled name and use the C calling convention.

第135章：数字分隔符

第135.1节：数字分隔符

超过几位数的数字字面量难以阅读。

- 读出 7237498123。
- 比较 237498123 和 237499123 是否相等。
- 判断 237499123 和 20249472 哪个更大。

C++14 定义单引号 ' 作为数字和用户自定义字面量中的数字分隔符。这可以使人类读者更容易解析大数字。

版本 ≥ C++14

```
long long decn = 1'000'000'00011;
long long hexn = 0xFFFF'FFFF11;
long long octn = 00'23'0011;
long long binn = 0b1010'001111;
```

单引号标记在确定其值时会被忽略。

示例：

- 字面量 1048576、1'048'576、0X100000、0x10'0000 和 0'004'000'000 的值都相同。
- 字面量 1.602'176'565e-19 和 1.602176565e-19 的值相同。

单引号的位置无关紧要。以下所有写法等价：

版本 ≥ C++14

```
long long a1 = 12345678911;
long long a2 = 123'456'78911;
long long a3 = 12'34'56'78'911;
long long a4 = 12345'678911;
```

也允许在user-defined字面量中使用：

版本 ≥ C++14

```
std::chrono::seconds tiempo = 1'674'456s + 5'300h;
```

Chapter 135: Digit separators

Section 135.1: Digit Separator

Numeric literals of more than a few digits are hard to read.

- Pronounce 7237498123.
- Compare 237498123 with 237499123 for equality.
- Decide whether 237499123 or 20249472 is larger.

C++14 define Simple Quotation Mark ' as a digit separator, in numbers and user-defined literals. This can make it easier for human readers to parse large numbers.

Version ≥ C++14

```
long long decn = 1'000'000'00011;
long long hexn = 0xFFFF'FFFF11;
long long octn = 00'23'0011;
long long binn = 0b1010'001111;
```

Single quotes mark are ignored when determining its value.

Example:

- The literals 1048576, 1'048'576, 0X100000, 0x10'0000, and 0'004'000'000 all have the same value.
- The literals 1.602'176'565e-19 and 1.602176565e-19 have the same value.

The position of the single quotes is irrelevant. All the following are equivalent:

Version ≥ C++14

```
long long a1 = 12345678911;
long long a2 = 123'456'78911;
long long a3 = 12'34'56'78'911;
long long a4 = 12345'678911;
```

It is also allowed in user-defined literals:

Version ≥ C++14

```
std::chrono::seconds tiempo = 1'674'456s + 5'300h;
```

第136章：C语言不兼容性

本章描述了哪些C代码在C++编译器中会出错。

第136.1节：保留关键字

第一个例子是C++中具有特殊用途的关键字：以下代码在C语言中合法，但在C++中不合法。

```
int class = 5
```

这些错误很容易修复：只需重命名变量即可。

第136.2节：弱类型指针

在C语言中，指针可以被转换为`void*`，而在C++中需要显式转换。以下代码在C++中是非法的，但在

```
void* ptr;
int* intptr = ptr;
```

添加显式类型转换可以使它工作，但可能会引发更多问题。

第136.3节：goto或switch

在C++中，不能使用`goto`或`switch`跳过初始化。以下代码在C中有效，但在C++中无效：

```
goto foo;
int skipped = 1;
foo;
```

这些错误可能需要重新设计。

Chapter 136: C incompatibilities

This describes what C code will break in a C++ compiler.

Section 136.1: Reserved Keywords

The first example are keywords that have a special purpose in C++: the following is legal in C, but not C++.

```
int class = 5
```

These errors are easy to fix: just rename the variable.

Section 136.2: Weakly typed pointers

In C, pointers can be cast to a `void*`, which needs an explicit cast in C++. The following is illegal in C++, but legal in C:

```
void* ptr;
int* intptr = ptr;
```

Adding an explicit cast makes this work, but can cause further issues.

Section 136.3: goto or switch

In C++, you may not skip initializations with `goto` or `switch`. The following is valid in C, but not C++:

```
goto foo;
int skipped = 1;
foo;
```

These bugs may require redesign.

第137章：经典C++示例的并排比较，分别通过C++、C++11、C++14和C++17解决

第137.1节：遍历容器

在C++中，遍历序列容器c可以使用索引，方法如下：

```
for(size_t i = 0; i < c.size(); ++i) c[i] = 0;
```

虽然简单，但此类写法容易出现常见的语义错误，比如错误的比较运算符或错误的索引变量：

```
for(size_t i = 0; i <= c.size(); ++j) c[i] = 0;
      ^~~~~~^
```

也可以使用迭代器对所有容器进行循环，但存在类似的缺点：

```
for(iterator it = c.begin(); it != c.end(); ++it) (*it) = 0;
```

C++11 引入了基于范围的 for 循环和 auto 关键字，使代码变为：

```
for(auto& x : c) x = 0;
```

这里唯一的参数是容器 c 和用于保存当前值的变量 x。这避免了之前提到的语义错误。

根据 C++11 标准，底层实现等价于：

```
for(auto begin = c.begin(), end = c.end(); begin != end; ++begin)
{
    // ...
}
```

在这种实现中，表达式 auto begin = c.begin(), end = c.end(); 强制要求 begin 和 end 类型相同，同时 end 从不被递增或解引用。因此，基于范围的 for 循环仅适用于由一对迭代器/迭代器定义的容器。C++17 标准通过将实现更改为：

```
auto begin = c.begin();
auto end = c.end();
for(; begin != end; ++begin)
{
    // ...
}
```

这里 begin 和 end 可以是不同类型，只要它们可以进行不等比较。这允许遍历更多容器，例如由一对迭代器/哨兵定义的容器。

Chapter 137: Side by Side Comparisons of classic C++ examples solved via C++ vs C++11 vs C++14 vs C++17

Section 137.1: Looping through a container

In C++, looping through a sequence container c can be done using indexes as follows:

```
for(size_t i = 0; i < c.size(); ++i) c[i] = 0;
```

While simple, such writings are subject to common semantic errors, like wrong comparison operator, or wrong indexing variable:

```
for(size_t i = 0; i <= c.size(); ++j) c[i] = 0;
      ^~~~~~^
```

Looping can also be achieved for all containers using iterators, with similar drawbacks:

```
for(iterator it = c.begin(); it != c.end(); ++it) (*it) = 0;
```

C++11 introduced range-based for loops and `auto` keyword, allowing the code to become:

```
for(auto& x : c) x = 0;
```

Here the only parameters are the container c, and a variable x to hold the current value. This prevents the semantics errors previously pointed.

According to the C++11 standard, the underlying implementation is equivalent to:

```
for(auto begin = c.begin(), end = c.end(); begin != end; ++begin)
{
    // ...
}
```

In such implementation, the expression `auto begin = c.begin(), end = c.end();` forces begin and end to be of the same type, while end is never incremented, nor dereferenced. So the range-based for loop only works for containers defined by a pair iterator/iterator. The C++17 standard relaxes this constraint by changing the implementation to:

```
auto begin = c.begin();
auto end = c.end();
for(; begin != end; ++begin)
{
    // ...
}
```

Here begin and end are allowed to be of different types, as long as they can be compared for inequality. This allows to loop through more containers, e.g. a container defined by a pair iterator/sentinel.

第138章：编译与构建

用C++编写的程序需要先编译才能运行。根据操作系统的不同，有多种编译器可供选择。

第138.1节：使用GCC编译

假设有一个名为 `main.cpp` 的单一源文件，编译并链接一个非优化的可执行文件的命令如下（无优化编译适合初期开发和调试，尽管对于较新的GCC版本官方推荐使用 `-Og`）。

```
g++ -o app -Wall main.cpp -O0
```

要生成用于生产环境的优化可执行文件，请使用 `-O` 选项之一（参见：`-O1`、`-O2`、`-O3`、`-Os`、`-Ofast`）：

```
g++ -o app -Wall -O2 main.cpp
```

如果省略 `-O` 选项，默认使用 `-O0`，即不进行优化（指定 `-O` 而不带数字时，等同于 `-O1`）。

或者，直接使用 `O` 组的优化标志（或更具实验性的优化）。以下示例使用 `-O2` 优化级别构建，并加上 `-O3` 优化级别中的一个标志：

```
g++ -o app -Wall -O2 -ftree-partial-pre main.cpp
```

要生成针对特定平台优化的可执行文件（用于在具有相同架构的机器上生产环境运行），请使用：

```
g++ -o app -Wall -O2 -march=native main.cpp
```

上述任一命令都会生成一个二进制文件，可在 Windows 上通过 `.\app.exe` 运行，在 Linux、Mac OS 等系统上通过 `./app` 运行。

`-o` 标志也可以省略。此时，GCC 会在 Windows 上创建默认输出可执行文件 `a.exe`，在类 Unix 系统上创建 `a.out`。要编译文件但不进行链接，请使用 `-c` 选项：

```
g++ -o file.o -Wall -c file.cpp
```

这会生成一个名为 `file.o` 的目标文件，之后可以与其他文件链接生成二进制文件：

```
g++ -o 应用文件.o 其他文件.o
```

有关优化选项的更多信息可以在 [gcc.gnu.org](#) 找到。特别值得注意的是 `-Og`（以调试体验为重点的优化——推荐用于标准的编辑-编译-调试周期）和 `-Ofast`（所有优化，包括忽略严格标准兼容性的优化）。

`-Wall` 标志启用许多常见错误的警告，应该始终使用。为了提高代码质量，通常还建议使用 `-Wextra` 以及其他 `-Wall` 和 `-Wextra` 未自动启用的警告标志。

如果代码期望使用特定的 C++ 标准，可以通过包含 `-std=` 标志来指定使用的标准。支持的

Chapter 138: Compiling and Building

Programs written in C++ need to be compiled before they can be run. There is a large variety of compilers available depending on your operating system.

Section 138.1: Compiling with GCC

Assuming a single source file named `main.cpp`, the command to compile and link an non-optimized executable is as follows (Compiling without optimization is useful for initial development and debugging, although `-Og` is officially recommended for newer GCC versions).

```
g++ -o app -Wall main.cpp -O0
```

To produce an optimized executable for use in production, use one of the `-O` options (see: `-O1`, `-O2`, `-O3`, `-Ofast`):

```
g++ -o app -Wall -O2 main.cpp
```

If the `-O` option is omitted, `-O0`, which means no optimizations, is used as default (specifying `-O` without a number resolves to `-O1`).

Alternatively, use optimization flags from the `O` groups (or more experimental optimizations) directly. The following example builds with `-O2` optimization, plus one flag from the `-O3` optimization level:

```
g++ -o app -Wall -O2 -ftree-partial-pre main.cpp
```

To produce a platform-specific optimized executable (for use in production on the machine with the same architecture), use:

```
g++ -o app -Wall -O2 -march=native main.cpp
```

Either of the above will produce a binary file that can be run with `.\app.exe` on Windows and `./app` on Linux, Mac OS, etc.

The `-o` flag can also be skipped. In this case, GCC will create default output executable `a.exe` on Windows and `a.out` on Unix-like systems. To compile a file without linking it, use the `-c` option:

```
g++ -o file.o -Wall -c file.cpp
```

This produces an object file named `file.o` which can later be linked with other files to produce a binary:

```
g++ -o app file.o otherfile.o
```

More about optimization options can be found at [gcc.gnu.org](#). Of particular note are `-Og` (optimization with an emphasis on debugging experience -- recommended for the standard edit-compile-debug cycle) and `-Ofast` (all optimizations, including ones disregarding strict standards compliance).

The `-Wall` flag enables warnings for many common errors and should always be used. To improve code quality it is often encouraged also to use `-Wextra` and other warning flags which are not automatically enabled by `-Wall` and `-Wextra`.

If the code expects a specific C++ standard, specify which standard to use by including the `-std=` flag. Supported

值对应于每个 ISO C++ 标准版本的最终确定年份。截至 GCC 6.1.0, std= 标志的有效值为 c++98/c++03、c++11、c++14 和 c++17/c++1z。用斜杠分隔的值是等效的。

```
g++ -std=c++11 <file>
```

GCC 包含一些特定于编译器的扩展，当它们与 -std= 标志指定的标准冲突时会被禁用。要启用所有扩展，可以使用值 gnu++XX，其中 XX 是上述 c++ 值中使用的任一年份。

如果未指定标准，则使用默认标准。对于 GCC 6.1.0 之前的版本，默认是 -std=gnu++03；在 GCC 6.1.0 及更高版本中，默认是 -std=gnu++14。

请注意，由于GCC中的错误，编译和链接时必须包含-pthread标志，GCC才能支持C++11引入的C++标准线程功能，例如 std::thread 和 std::wait_for。使用线程函数时如果省略该标志，某些平台上可能不会有警告，但会导致无效结果。

与库链接：

使用-l选项传递库名：

```
g++ main.cpp -lpcre2-8
```

#pcre2-8是针对8位代码单元（UTF-8）的PCRE2库

如果库不在标准库路径中，使用-L选项添加路径：

```
g++ main.cpp -L/my/custom/path/ -lmylib
```

可以同时链接多个库：

```
g++ main.cpp -lmylib1 -lmylib2 -lmylib3
```

如果一个库依赖另一个库，将依赖库放在独立库之前：

```
g++ main.cpp -lchild-lib -lbase-lib
```

或者让链接器通过--start-group和--end-group自行确定顺序（注意：这会带来显著的性能开销）：

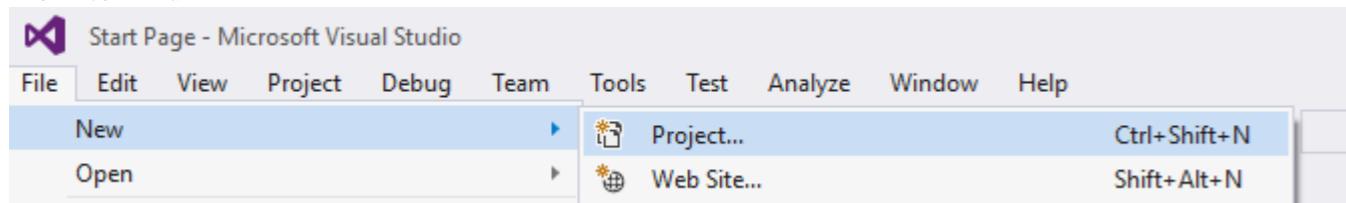
```
g++ main.cpp -Wl,--start-group -lbase-lib -lchild-lib -Wl,--end-group
```

第138.2节：使用Visual Studio（图形界面）编译 - Hello World

1. 下载并安装Visual Studio Community 2015

2. 打开Visual Studio Community

3. 点击 文件 -> 新建 -> 项目



4. 点击 模板 -> Visual C++ -> Win32控制台应用程序，然后将项目命名为MyFirstProgram。

values correspond to the year of finalization for each version of the ISO C++ standard. As of GCC 6.1.0, valid values for the std= flag are c++98/c++03, c++11, c++14, and c++17/c++1z. Values separated by a forward slash are equivalent.

```
g++ -std=c++11 <file>
```

GCC includes some compiler-specific extensions that are disabled when they conflict with a standard specified by the -std= flag. To compile with all extensions enabled, the value gnu++XX may be used, where XX is any of the years used by the c++ values listed above.

The default standard will be used if none is specified. For versions of GCC prior to 6.1.0, the default is -std=gnu++03; in GCC 6.1.0 and greater, the default is -std=gnu++14.

Note that due to bugs in GCC, the -pthread flag must be present at compilation and linking for GCC to support the C++ standard threading functionality introduced with C++11, such as std::thread and std::wait_for. Omitting it when using threading functions may result in no warnings but invalid results on some platforms.

Linking with libraries:

Use the -l option to pass the library name:

```
g++ main.cpp -lpcre2-8
```

#pcre2-8 is the PCRE2 library for 8bit code units (UTF-8)

If the library is not in the standard library path, add the path with -L option:

```
g++ main.cpp -L/my/custom/path/ -lmylib
```

Multiple libraries can be linked together:

```
g++ main.cpp -lmylib1 -lmylib2 -lmylib3
```

If one library depends on another, put the dependent library **before** the independent library:

```
g++ main.cpp -lchild-lib -lbase-lib
```

Or let the linker determine the ordering itself via --start-group and --end-group (note: this has significant performance cost):

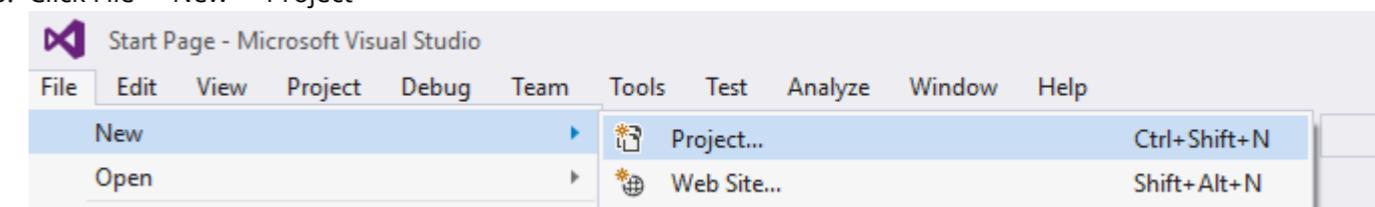
```
g++ main.cpp -Wl,--start-group -lbase-lib -lchild-lib -Wl,--end-group
```

Section 138.2: Compiling with Visual Studio (Graphical Interface) - Hello World

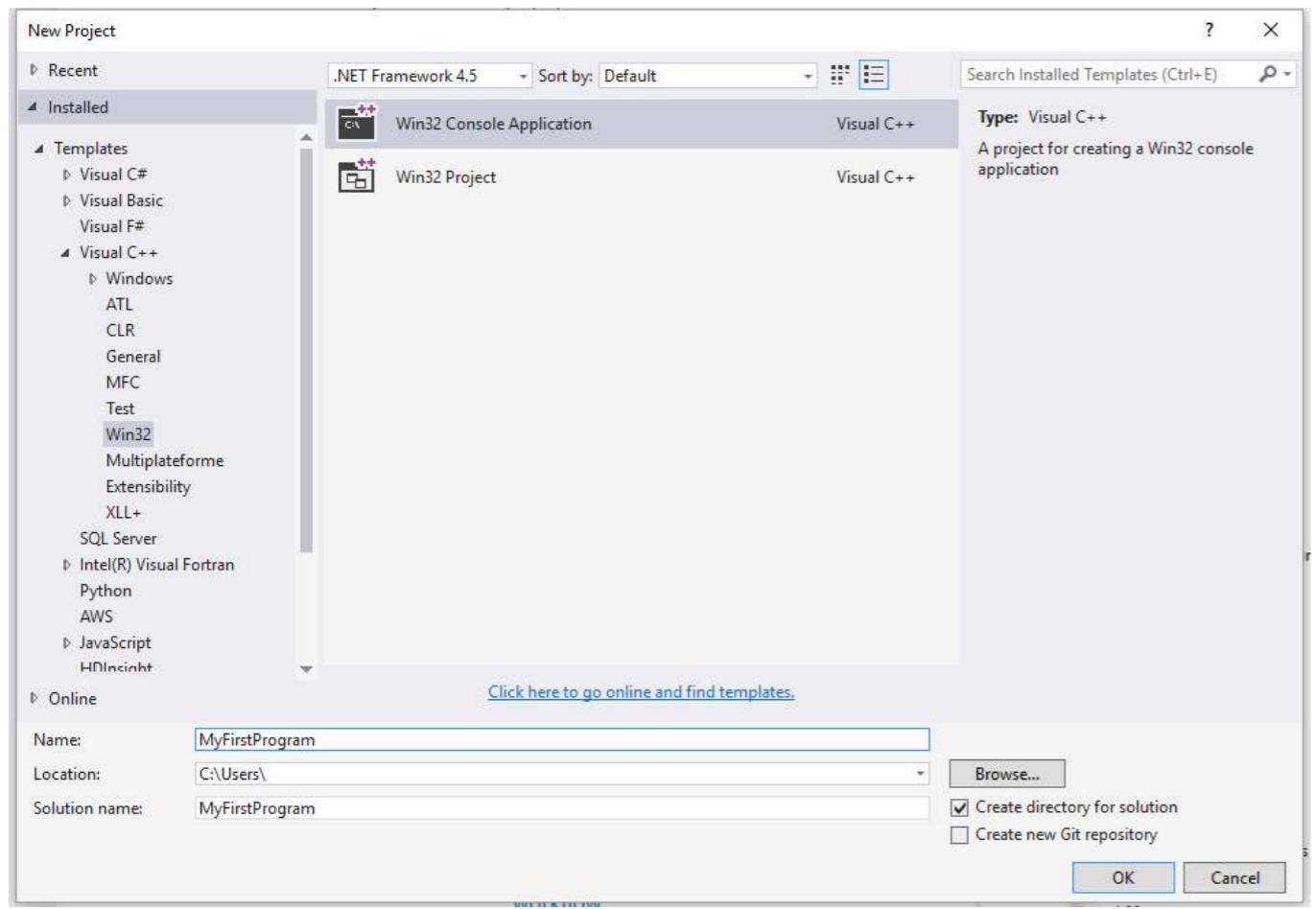
1. Download and install Visual Studio Community 2015

2. Open Visual Studio Community

3. Click File -> New -> Project

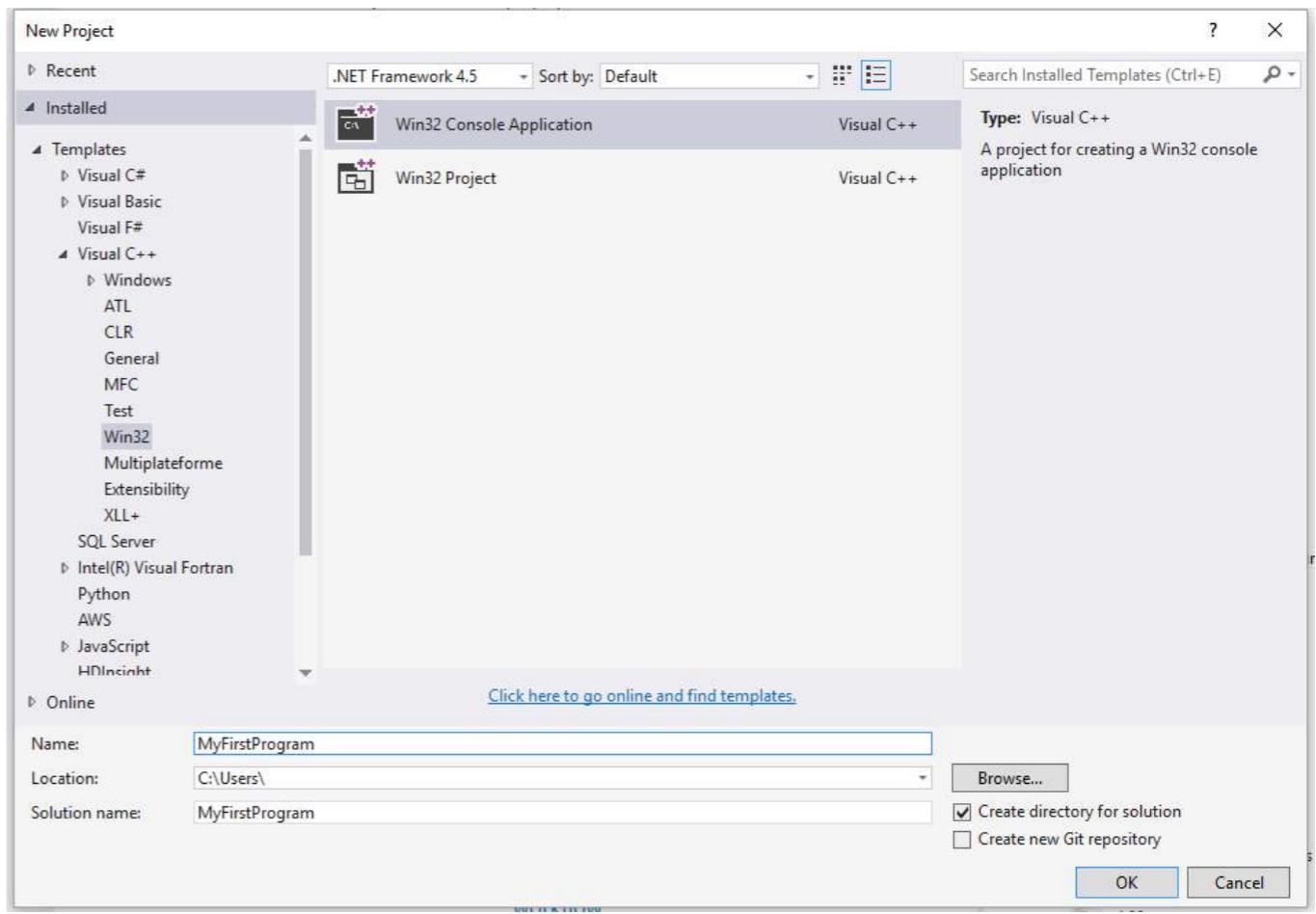


4. Click Templates -> Visual C++ -> Win32 Console Application and then name the project **MyFirstProgram**.



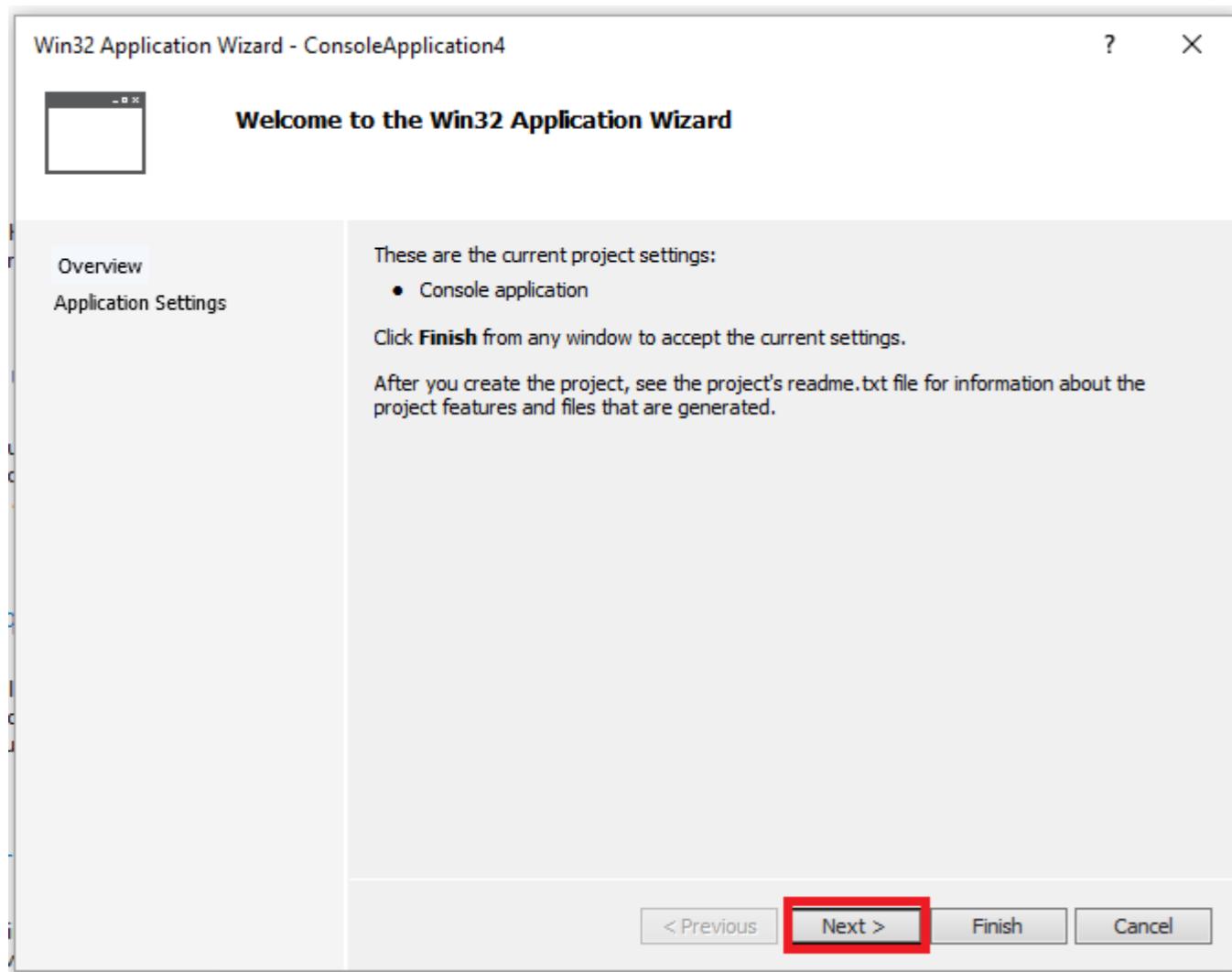
5.点击确定

6.在接下来的窗口中点击下一步。

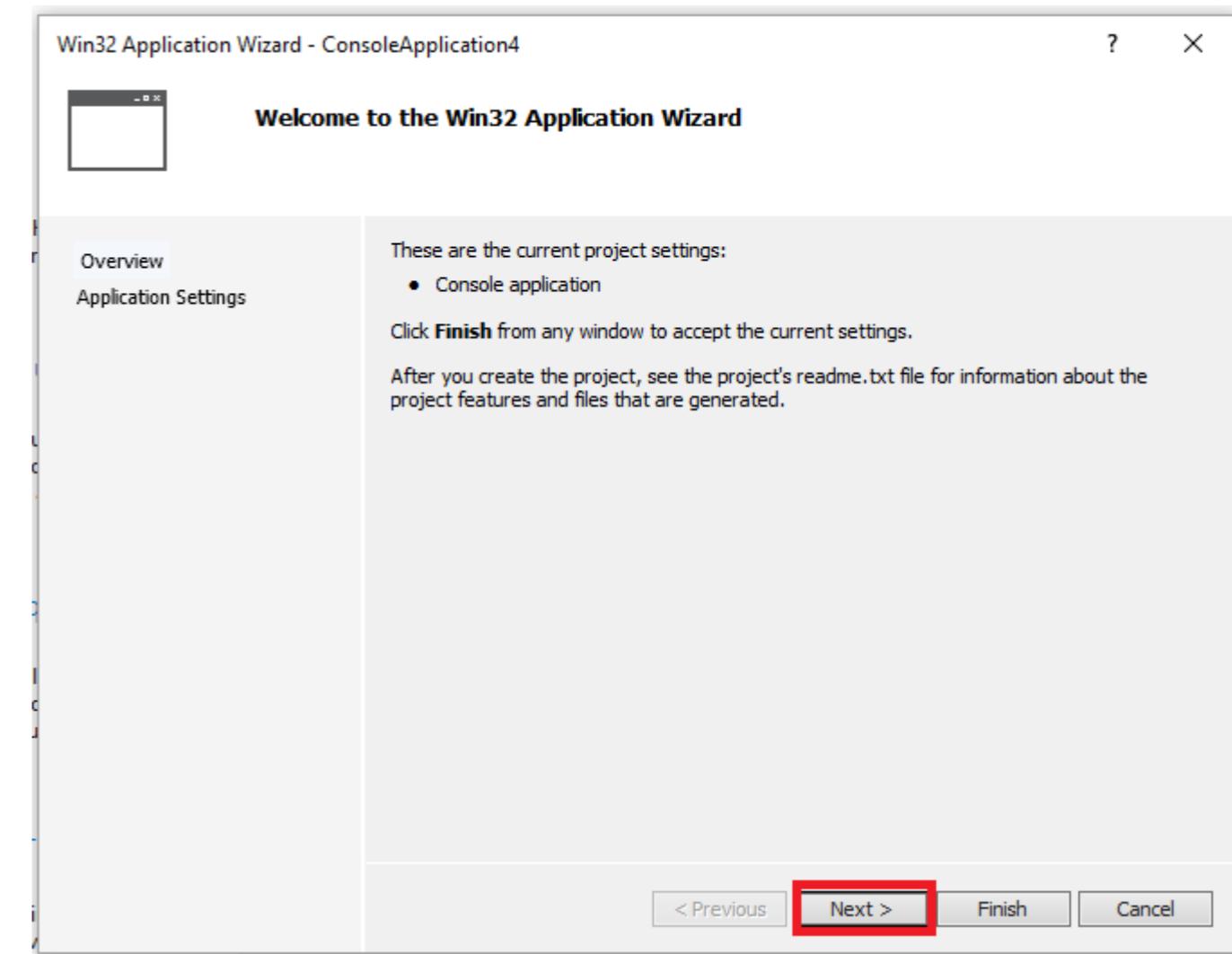


5. Click Ok

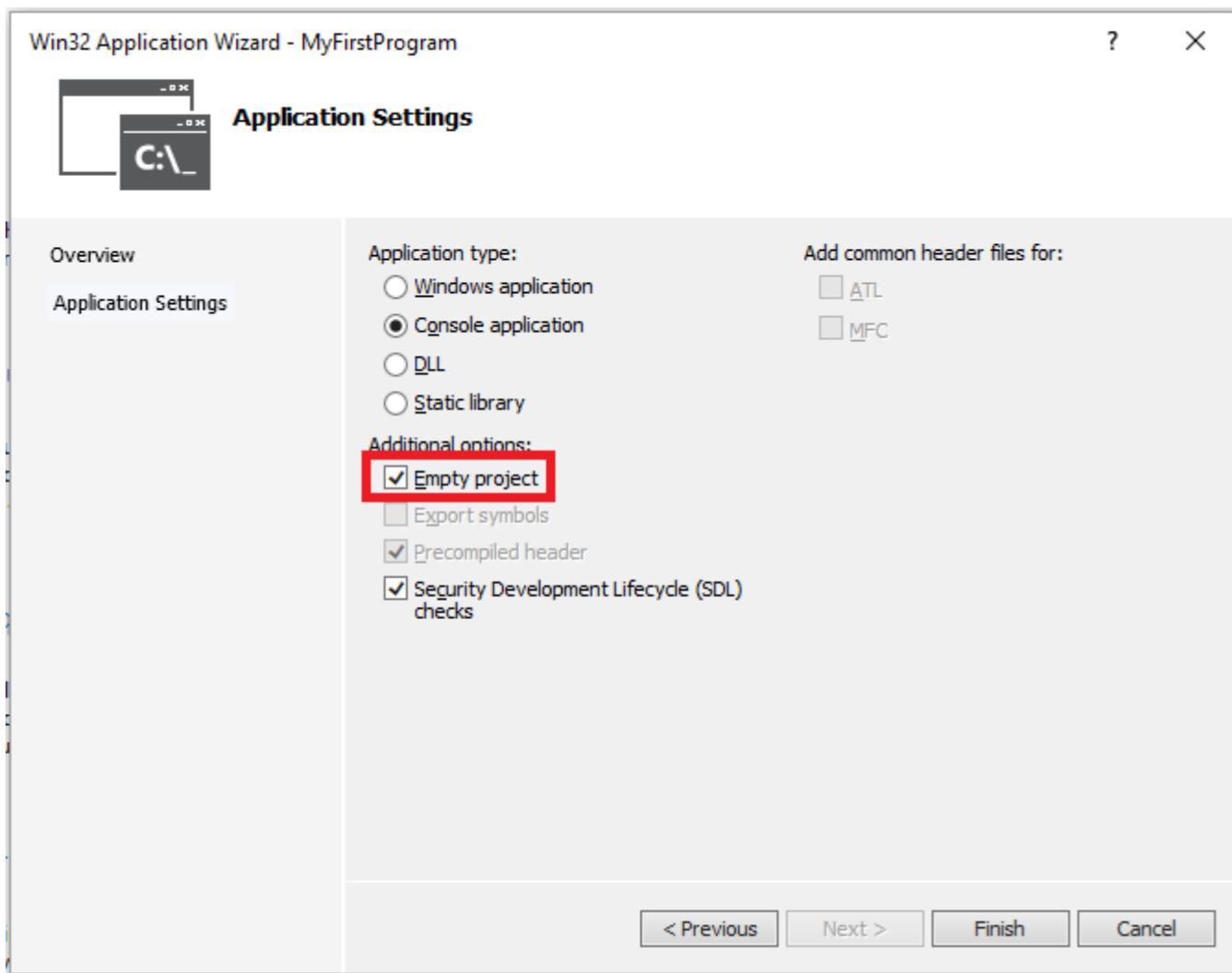
6. Click Next in the following window.



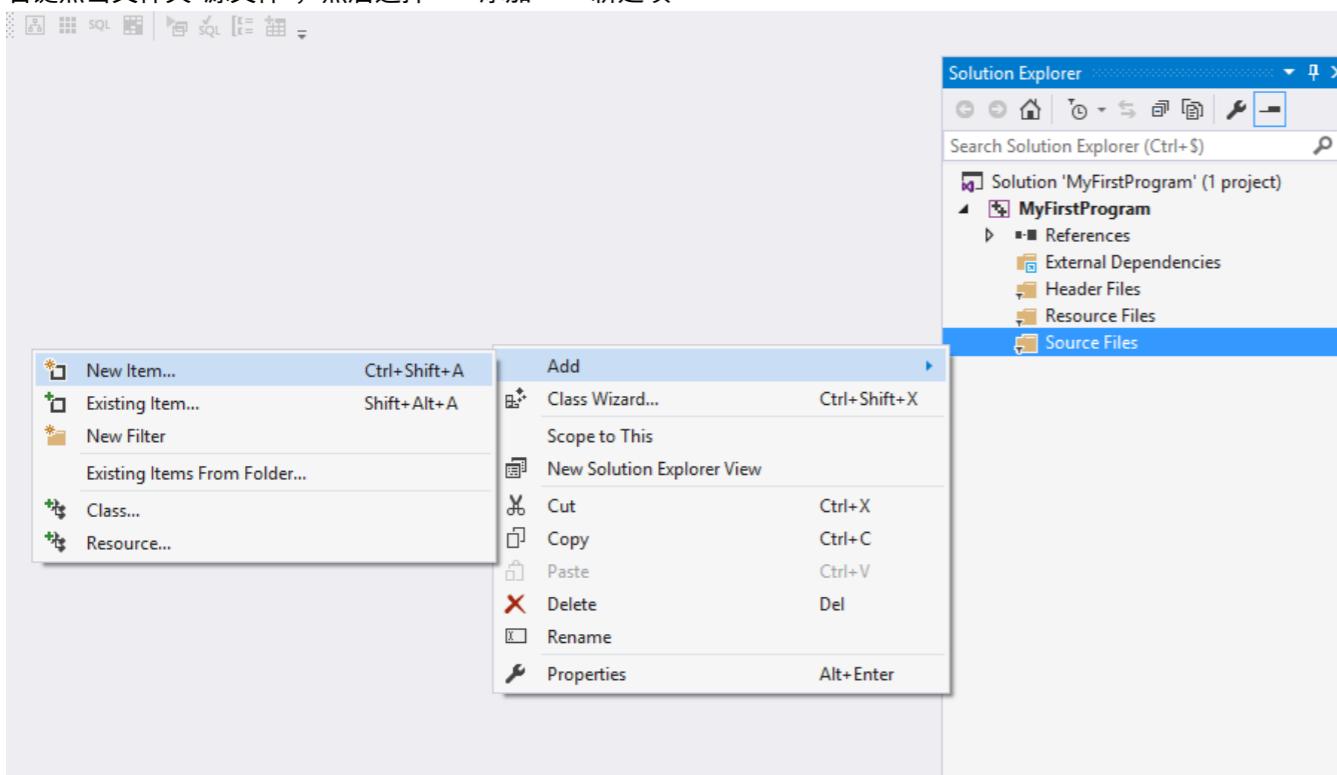
7. 勾选空项目框，然后点击完成：



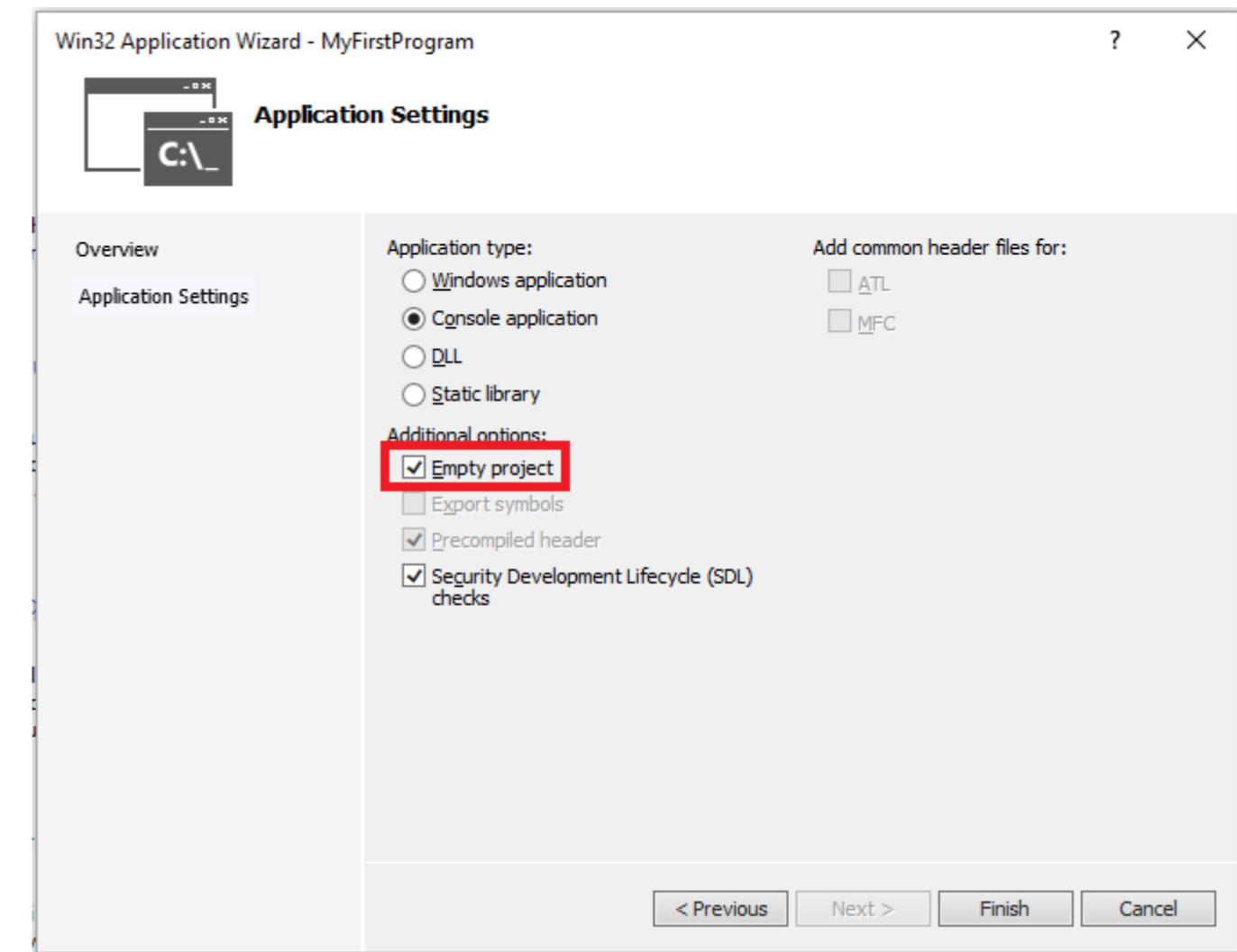
7. Check the Empty project box and then click Finish:



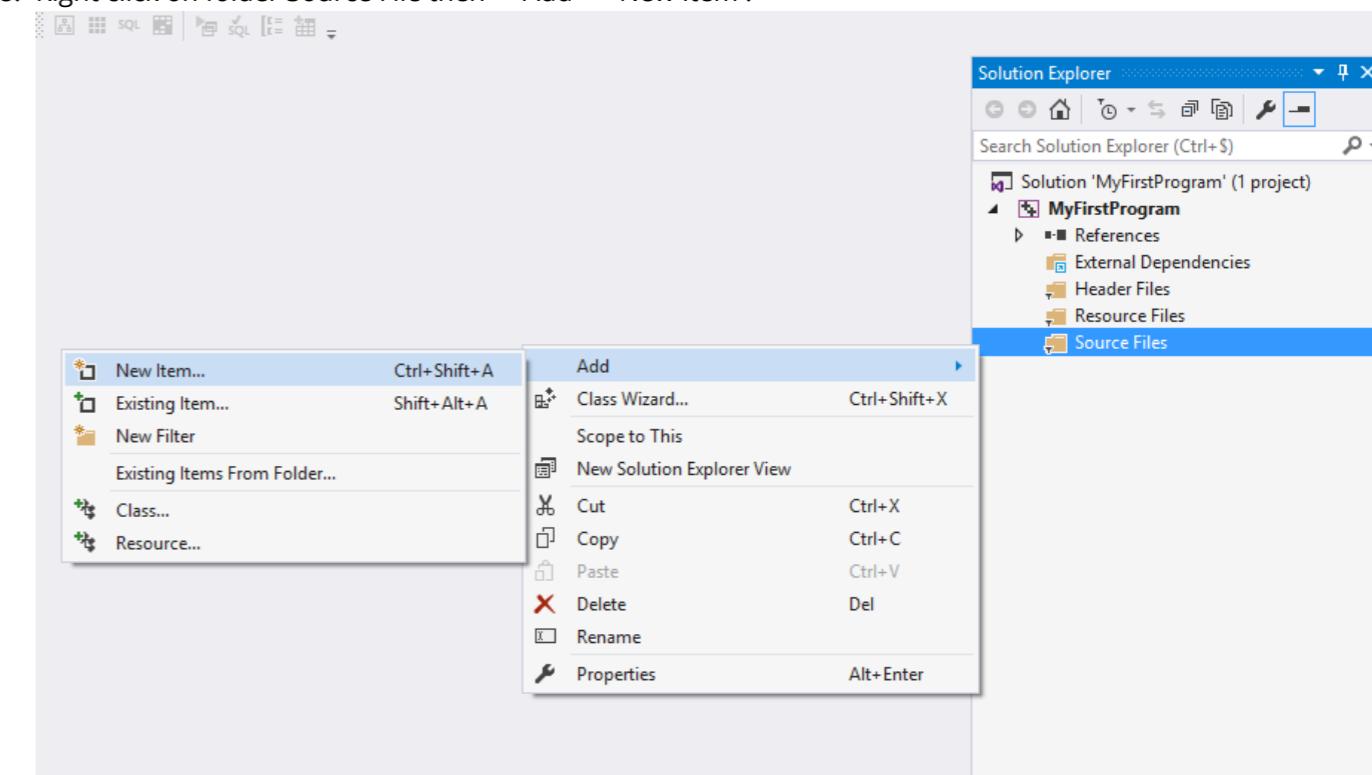
8. 右键点击文件夹“源文件”，然后选择 -> 添加 --> 新建项：



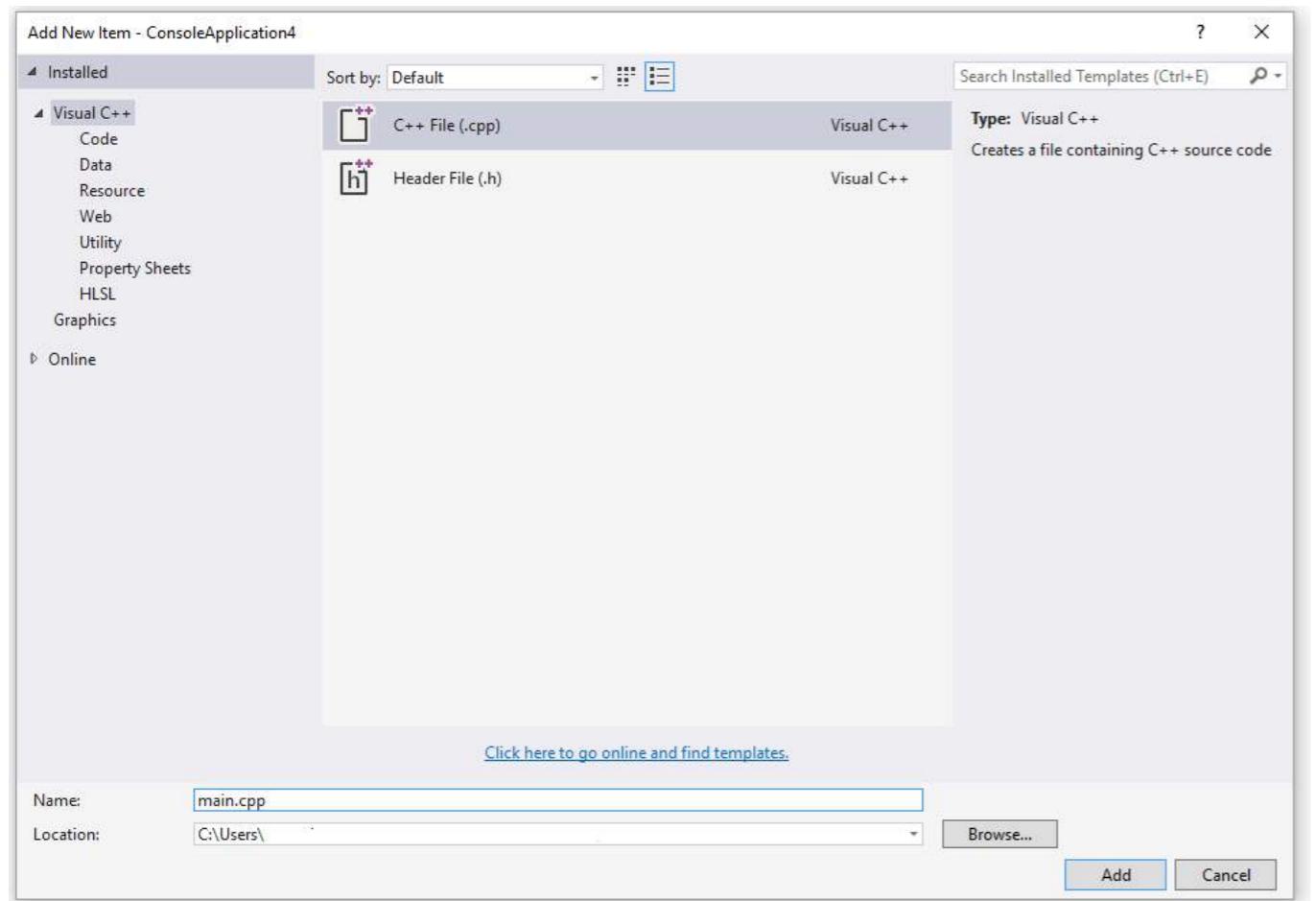
9. 选择 C++ 文件并将文件命名为 main.cpp，然后点击添加：



8. Right click on folder Source File then -> Add --> New Item :



9. Select C++ File and name the file main.cpp, then click Add:

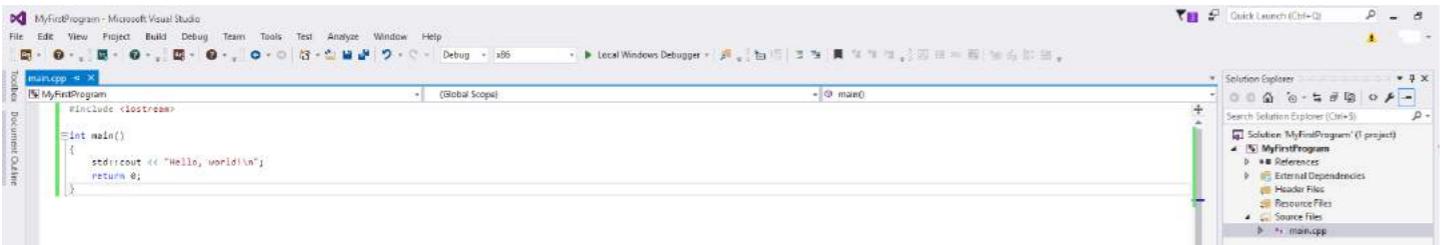


10: 在新文件 main.cpp 中复制并粘贴以下代码：

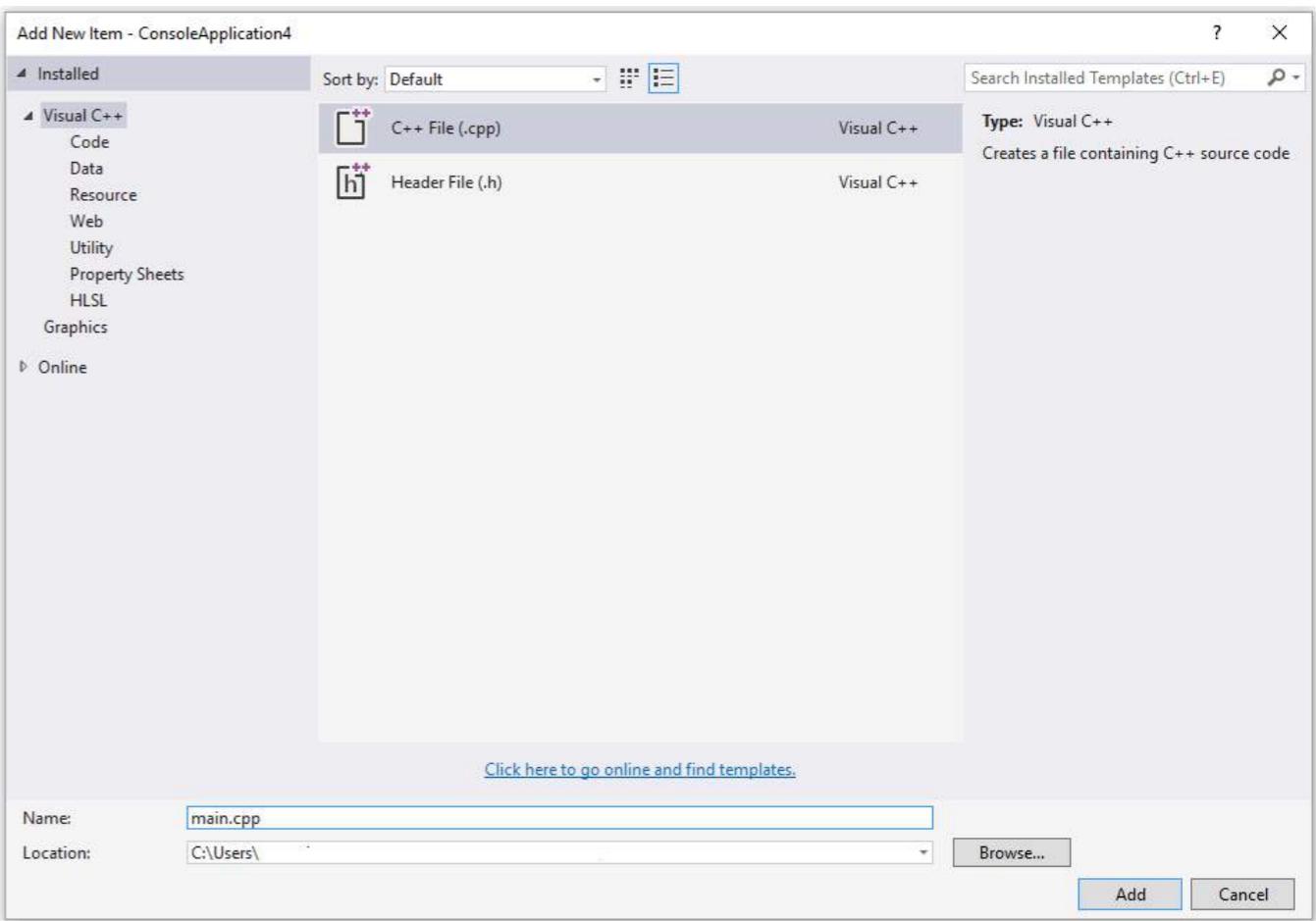
```
#include <iostream>

int main()
{
    std::cout << "Hello World!">> 0;
}
```

你的环境应如下所示：



11. 点击调试 -> 开始无调试（或按 ctrl + F5）：

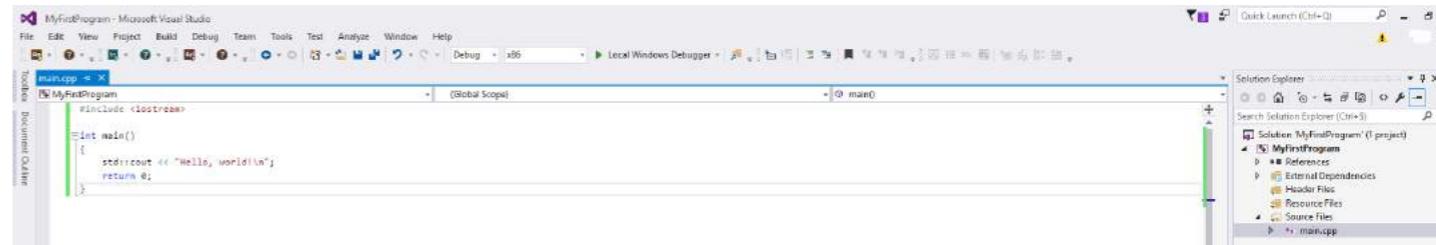


10: Copy and paste the following code in the new file main.cpp:

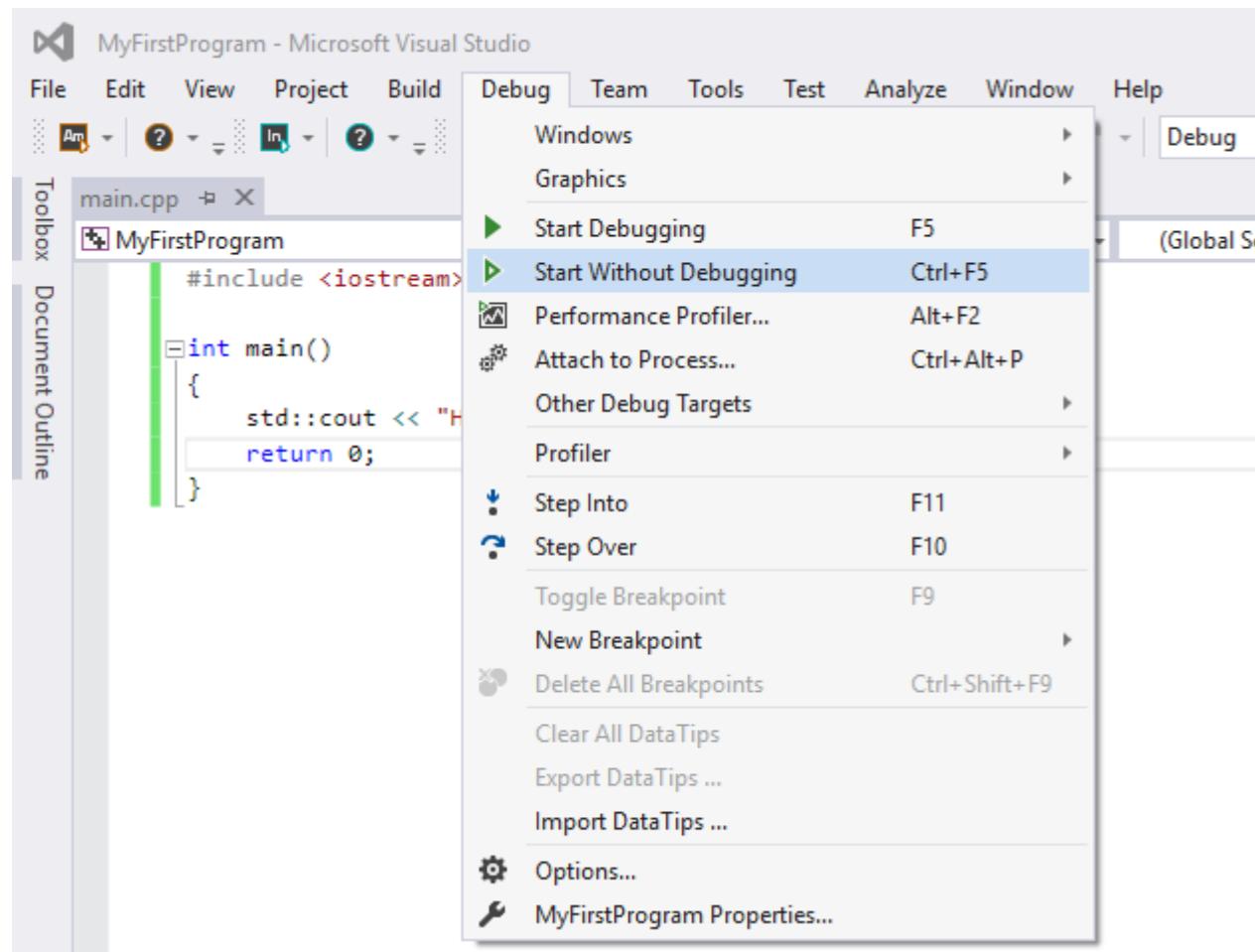
```
#include <iostream>

int main()
{
    std::cout << "Hello World!\n";
    return 0;
}
```

Your environment should look like:



11. Click Debug -> Start **Without** Debugging (or press ctrl + F5) :



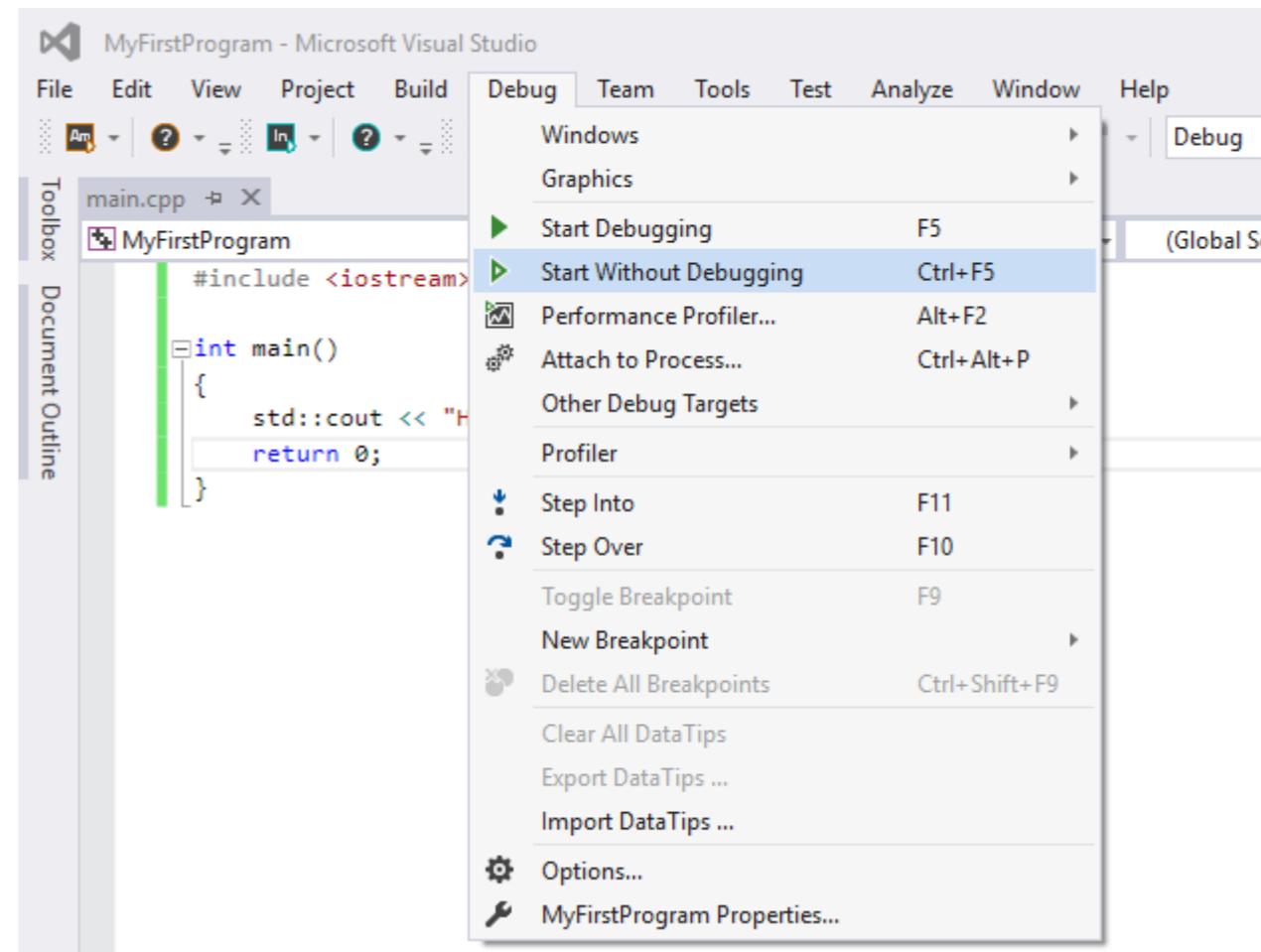
12. 完成。你应该会看到以下控制台输出：

```
C:\Windows\system32\cmd.exe
Hello World!
Press any key to continue . . .
```

第138.3节：在线编译器

各种网站提供对C++编译器的在线访问。在线编译器的功能集因网站而异，但通常它们允许执行以下操作：

- 将你的代码粘贴到浏览器中的网页表单中。
- 选择一些编译器选项并编译代码。
- 收集编译器和/或程序输出。



12. Done. You should get the following console output :

```
C:\Windows\system32\cmd.exe
Hello World!
Press any key to continue . . .
```

Section 138.3: Online Compilers

Various websites provide online access to C++ compilers. Online compiler's feature set vary significantly from site to site, but usually they allow to do the following:

- Paste your code into a web form in the browser.
- Select some compiler options and compile the code.
- Collect compiler and/or program output.

在线编译器网站的行为通常相当受限，因为它们允许任何人在服务器端运行编译器并执行任意代码，而远程任意代码执行通常被视为安全漏洞。

在线编译器可能对以下用途有帮助：

- 在没有C++编译器的设备（智能手机、平板电脑等）上运行一小段代码片段。
- 确保代码能在不同编译器下成功编译，并且无论使用哪种编译器编译，运行结果都相同。
- 学习或教授C++基础知识。
- 在本地机器上没有最新C++编译器的情况下，学习现代C++特性（近期的C++14和C++17）。
- 通过与大量其他编译器的比较，发现你的编译器中的错误。检查编译器错误是否在你机器上不可用的未来版本中被修复。
- 解决在线评测系统的问题。

在线编译器不应用于以下情况：

- 使用C++开发功能齐全的（即使是小型的）应用程序。通常在线编译器不允许链接第三方库或下载构建产物。
- 执行密集计算。服务器端计算资源有限，任何用户提供的程序在运行几秒后将被终止。允许的执行时间通常足够用于测试和学习。
- 攻击编译器服务器本身或网络上的任何第三方主机。

示例：

免责声明：文档作者与以下列出的任何资源无关。网站按字母顺序排列。

- <http://codepad.org/> 在线编译器，支持代码共享。编译后编辑代码时，如果有源代码警告或错误，效果不佳。
- <http://coliru.stacked-crooked.com/> 在线编译器，允许你指定命令行。提供GCC和Clang编译器供使用。
- <http://cpp.sh/> - 支持C++14的在线编译器。不允许编辑编译器命令行，但通过图形界面控件提供一些选项。
- <https://gcc.godbolt.org/> - 提供多种编译器版本、架构和反汇编输出。
当你需要检查代码被不同编译器编译成什么时非常有用。支持GCC、Clang、MSVC (CL)、Intel编译器 (icc)、ELLCC和Zapcc，这些编译器中有一个或多个可用于ARM、ARMv8（作为ARM64）、Atmel AVR、MIPS、MIPS64、MSP430、PowerPC、x86和x64架构。编译器命令行参数可编辑。
- <https://ideone.com/> - 网络上广泛使用的代码片段演示工具。提供GCC和Clang，但不允许编辑编译器命令行。
- <http://melpon.org/wandbox> - 支持多种Clang和GNU/GCC编译器版本。
- <http://onlinegdb.com/> - 极简IDE，包含编辑器、编译器 (gcc) 和调试器 (gdb)。
- <http://rextester.com/> - 提供Clang、GCC和Visual Studio编译器，支持C和C++（以及其他语言的编译器），并可使用Boost库。
- http://tutorialspoint.com/compile_cpp11_online.php - 配备完整功能的 UNIX shell，带有 GCC 和用户友好的项目浏览器。
- <http://webcompiler.cloudapp.net/> - 由微软提供的在线 Visual Studio 2015 编译器，作为

Online compiler website behavior is usually quite restrictive as they allow anyone to run compilers and execute arbitrary code on their server side, whereas ordinarily remote arbitrary code execution is considered as vulnerability.

Online compilers may be useful for the following purposes:

- Run a small code snippet from a machine which lacks C++ compiler (smartphones, tablets, etc.).
- Ensure that code compiles successfully with different compilers and runs the same way regardless the compiler it was compiled with.
- Learn or teach basics of C++.
- Learn modern C++ features (C++14 and C++17 in near future) when up-to-date C++ compiler is not available on local machine.
- Spot a bug in your compiler by comparison with a large set of other compilers. Check if a compiler bug was fixed in future versions, which are unavailable on your machine.
- Solve online judge problems.

What online compilers should **not** be used for:

- Develop full-featured (even small) applications using C++. Usually online compilers do not allow to link with third-party libraries or download build artifacts.
- Perform intensive computations. Server-side computing resources are limited, so any user-provided program will be killed after a few seconds of execution. The permitted execution time is usually enough for testing and learning.
- Attack compiler server itself or any third-party hosts on the net.

Examples:

Disclaimer: documentation author(s) are not affiliated with any resources listed below. Websites are listed alphabetically.

- <http://codepad.org/> Online compiler with code sharing. Editing code after compiling with a source code warning or error does not work so well.
- <http://coliru.stacked-crooked.com/> Online compiler for which you specify the command line. Provides both GCC and Clang compilers for use.
- <http://cpp.sh/> - Online compiler with C++14 support. Does not allow you to edit compiler command line, but some options are available via GUI controls.
- <https://gcc.godbolt.org/> - Provides a wide list of compiler versions, architectures, and disassembly output. Very useful when you need to inspect what your code compiles into by different compilers. GCC, Clang, MSVC (CL), Intel compiler (icc), ELLCC, and Zapcc are present, with one or more of these compilers available for the ARM, ARMv8 (as ARM64), Atmel AVR, MIPS, MIPS64, MSP430, PowerPC, x86, and x64 architectures. Compiler command line arguments may be edited.
- <https://ideone.com/> - Widely used on the Net to illustrate code snippet behavior. Provides both GCC and Clang for use, but doesn't allow you to edit the compiler command line.
- <http://melpon.org/wandbox> - Supports numerous Clang and GNU/GCC compiler versions.
- <http://onlinegdb.com/> - An extremely minimalistic IDE that includes an editor, a compiler (gcc), and a debugger (gdb).
- <http://rextester.com/> - Provides Clang, GCC, and Visual Studio compilers for both C and C++ (along with compilers for other languages), with the Boost library available for use.
- http://tutorialspoint.com/compile_cpp11_online.php - Full-featured UNIX shell with GCC, and a user-friendly project explorer.
- <http://webcompiler.cloudapp.net/> - Online Visual Studio 2015 compiler, provided by Microsoft as part of

第138.4节：使用 Visual C++ 编译（命令行）

对于从 GCC 或 Clang 转到 Visual Studio 的程序员，或更习惯于使用命令行的程序员，您可以使用 Visual C++ 编译器的命令行版本，也可以使用 IDE。

如果您希望在 Visual Studio 中通过命令行编译代码，首先需要设置命令行环境。可以通过打开 Visual Studio 命令提示符/开发者命令提示符/x86 本机工具命令提示符/x64 本机工具命令提示符或类似工具（由您的 Visual Studio 版本提供），或者在命令提示符下，导航到编译器安装目录的 VC 子目录（通常是 \Program Files(x86)\Microsoft Visual Studio x\VC，其中 x 是版本号（例如 10.0 代表 2010，或 14.0 代表 2015）），然后运行带有命令行参数的 VCVARSALL 批处理文件。

请注意，与 GCC 不同，Visual Studio 不通过编译器 (cl.exe) 为链接器 (link.exe) 提供前端，而是将链接器作为单独的程序提供，编译器在退出时调用它。 cl.exe 和 link.exe 可以分别用于不同的文件和选项，或者如果两个任务一起完成，cl 可以被告知将文件和选项传递给 link。任何传递给 cl 的链接选项都会被转换为 link 的选项，任何未被 cl 处理的文件将直接传递给 link。由于这主要是 Visual Studio 命令行编译的简单指南，暂时不会描述 link 的参数；如果您需要参数列表，请参见此处。

请注意，传递给 cl 的参数区分大小写，而传递给 link 的参数不区分大小写。

[请注意，以下某些示例在指定绝对路径时使用了 Windows shell 的“当前目录”变量 %cd%。对于不熟悉此变量的人，它会展开为当前工作目录。在命令行中，它是您运行 cl 时所在的目录，并且默认在命令提示符中指定（例如，如果您的命令提示符是 C:\src>，则 %cd% 为 C:\src\）。] 假设当前文件夹中有一个名为 main.cpp 的单个源文件，编译并链接一个未优化的可执行文件（适用于初期开发和调试）的命令是（以下任一命令均可）：

```
cl main.cpp
// 生成目标文件 "main.obj"。
// 使用 "main.obj" 进行链接。
// 生成可执行文件 "main.exe"。
```

```
cl /Od main.cpp
// 与上述相同。
// "/Od" 是"优化：禁用"选项，当未指定 /O 时为默认选项。
```

假设同一目录下还有一个源文件 "niam.cpp"，使用以下命令：

```
cl main.cpp niam.cpp
// 生成目标文件 "main.obj" 和 "niam.obj"。
// 使用 "main.obj" 和 "niam.obj" 进行链接。
// 生成可执行文件 "main.exe"。
```

你也可以像预期的那样使用通配符：

```
cl main.cpp src\*.cpp
// 生成目标文件 "main.obj"，以及文件夹
// "%cd%\src" 中每个 ".cpp" 文件对应的对象文件。
// 使用 "main.obj" 和所有额外生成的目标文件进行链接。
// 所有目标文件都将位于当前文件夹中。
```

Section 138.4: Compiling with Visual C++ (Command Line)

For programmers coming from GCC or Clang to Visual Studio, or programmers more comfortable with the command line in general, you can use the Visual C++ compiler from the command line as well as the IDE.

If you desire to compile your code from the command line in Visual Studio, you first need to set up the command line environment. This can be done either by opening the [Visual Studio Command Prompt/Developer Command Prompt/x86 Native Tools Command Prompt/x64 Native Tools Command Prompt or similar](#) (as provided by your version of Visual Studio), or at the command prompt, by navigating to the VC subdirectory of the compiler's install directory (typically \Program Files (x86)\Microsoft Visual Studio x\VC, where x is the version number (such as [10.0](#) for 2010, or [14.0](#) for 2015)) and running the VCVARSALL batch file with a command-line parameter specified [here](#).

Note that unlike GCC, Visual Studio doesn't provide a front-end for the linker (link.exe) via the compiler (cl.exe), but instead provides the linker as a separate program, which the compiler calls as it exits. cl.exe and link.exe can be used separately with different files and options, or cl can be told to pass files and options to link if both tasks are done together. Any linking options specified to cl will be translated into options for link, and any files not processed by cl will be passed directly to link. As this is mainly a simple guide to compiling with the Visual Studio command line, arguments for link will not be described at this time; if you need a list, see [here](#).

Note that arguments to cl are case-sensitive, while arguments to link are not.

[Be advised that some of the following examples use the Windows shell "current directory" variable, %cd%, when specifying absolute path names. For anyone unfamiliar with this variable, it expands to the current working directory. From the command line, it will be the directory you were in when you ran cl, and is specified in the command prompt by default (if your command prompt is C:\src>, for example, then %cd% is C:\src\).]

Assuming a single source file named [main.cpp](#) in the current folder, the command to compile and link an unoptimised executable (useful for initial development and debugging) is (use either of the following):

```
cl main.cpp
// Generates object file "main.obj".
// Performs linking with "main.obj".
// Generates executable "main.exe".
```

```
cl /Od main.cpp
// Same as above.
// "/Od" is the "Optimisation: disabled" option, and is the default when no /O is specified.
```

Assuming an additional source file "niam.cpp" in the same directory, use the following:

```
cl main.cpp niam.cpp
// Generates object files "main.obj" and "niam.obj".
// Performs linking with "main.obj" and "niam.obj".
// Generates executable "main.exe".
```

You can also use wildcards, as one would expect:

```
cl main.cpp src\*.cpp
// Generates object file "main.obj", plus one object file for each ".cpp" file in folder
// "%cd%\src".
// Performs linking with "main.obj", and every additional object file generated.
// All object files will be in the current folder.
```

```
// 生成可执行文件 "main.exe"。
```

要重命名或移动可执行文件，请使用以下方法之一：

```
cl /o name main.cpp  
// 生成名为 "name.exe" 的可执行文件。
```

```
cl /o folder\ main.cpp  
// 在文件夹 "%cd%\folder" 中生成名为 "main.exe" 的可执行文件。
```

```
cl /o folderame main.cpp  
// 生成名为 "name.exe" 的可执行文件，位于文件夹 "%cd%\folder" 中。
```

```
cl /Fename main.cpp  
// 与 "/o name" 相同。
```

```
cl /Fefolder\ main.cpp  
// 与 "/o folder\" 相同。
```

```
cl /Fefolderame main.cpp // 与  
"/o folderame" 相同。
```

无论是 /o 还是 /Fe 都会将它们的参数（我们称之为 o-param）传递给 link，形式为 /OUT:o-param，并根据需要在 "name" o-param 后添加适当的扩展名（通常是 .exe 或 .dll）。据我所知，/o 和 /Fe 在功能上是相同的，但后者在 Visual Studio 中更受推荐。/o 被标记为已弃用，主要是为了方便更熟悉 GCC 或 Clang 的程序员使用。

注意，虽然 /o 与指定的文件夹和/或名称之间的空格是可选的，但 /Fe 与指定的文件夹和/或名称之间 不能 有空格。

同样地，要生成优化过的可执行文件（用于生产环境），请使用：

```
cl /O1 main.cpp  
// 优化可执行文件大小。生成体积较小的程序，但可能以执行速度变慢为代价。
```

```
cl /O2 main.cpp  
// 优化执行速度。生成运行速度快的程序，但可能会导致文件体积增大。
```

```
cl /GL main.cpp other.cpp  
// 生成用于全程序优化的特殊目标文件，使 CL 在优化时能够考虑每个模块（翻译单元）。
```

```
// 将选项 "/LTCG"（链接时代码生成）传递给 LINK，告诉它在链接阶段调用 CL 以执行额外的优化。如果此时不进行链接，  
生成的目标文件应使用 "/LTCG" 进行链接。
```

```
// 可以与其他 CL 优化选项一起使用。
```

最后，为了生成针对特定平台优化的可执行文件（用于在具有指定架构的机器上生产环境使用），请选择适合目标平台的命令提示符或 VCVARSALL 参数。

link 应该从目标文件中检测所需平台；如果没有检测到，请使用 /MACHINE 选项显式指定目标平台。

```
// 如果为 x64 编译，而 LINK 未自动检测目标平台：  
cl main.cpp /link /machine:X64
```

以上任意命令都会生成一个可执行文件，文件名由 /o 或 /Fe 指定；如果两者都未提供，则文件名与传递给编译器的第一个源文件或目标文件相同。

```
// Generates executable "main.exe".
```

To rename or relocate the executable, use one of the following:

```
cl /o name main.cpp  
// Generates executable named "name.exe".
```

```
cl /o folder\ main.cpp  
// Generates executable named "main.exe", in folder "%cd%\folder".
```

```
cl /o folder\name main.cpp  
// Generates executable named "name.exe", in folder "%cd%\folder".
```

```
cl /Fename main.cpp  
// Same as "/o name".
```

```
cl /Fefolder\ main.cpp  
// Same as "/o folder\".
```

```
cl /Fefolder\name main.cpp  
// Same as "/o folder\name".
```

Both /o and /Fe pass their parameter (let's call it o-param) to link as /OUT:o-param, appending the appropriate extension (generally .exe or .dll) to "name" o-params as necessary. While both /o and /Fe are to my knowledge identical in functionality, the latter is preferred for Visual Studio. /o is marked as deprecated, and appears to mainly be provided for programmers more familiar with GCC or Clang.

Note that while the space between /o and the specified folder and/or name is optional, there *cannot* be a space between /Fe and the specified folder and/or name.

Similarly, to produce an optimised executable (for use in production), use:

```
cl /O1 main.cpp  
// Optimise for executable size. Produces small programs, at the possible expense of slower  
// execution.
```

```
cl /O2 main.cpp  
// Optimise for execution speed. Produces fast programs, at the possible expense of larger  
// file size.
```

```
cl /GL main.cpp other.cpp  
// Generates special object files used for whole-program optimisation, which allows CL to  
// take every module (translation unit) into consideration during optimisation.  
// Passes the option "/LTCG" (Link-Time Code Generation) to LINK, telling it to call CL during  
// the linking phase to perform additional optimisations. If linking is not performed at this  
// time, the generated object files should be linked with "/LTCG".  
// Can be used with other CL optimisation options.
```

Finally, to produce a platform-specific optimized executable (for use in production on the machine with the specified architecture), choose the appropriate command prompt or VCVARSALL parameter for the target platform. link should detect the desired platform from the object files; if not, use the /MACHINE option to explicitly specify the target platform.

```
// If compiling for x64, and LINK doesn't automatically detect target platform:  
cl main.cpp /link /machine:X64
```

Any of the above will produce an executable with the name specified by /o or /Fe, or if neither is provided, with a name identical to the first source or object file specified to the compiler.

```
cl a.cpp b.cpp c.cpp  
// 生成 "a.exe".  
  
cl d.obj a.cpp q.cpp  
// 生成 "d.exe".  
  
cl y.lib n.cpp o.obj  
// 生成 "n.exe".  
  
cl /o yo zp.obj pz.cpp  
// 生成 "yo.exe".
```

要编译文件但不进行链接，请使用：

```
cl /c main.cpp  
// 生成目标文件 "main.obj".
```

这告诉cl退出而不调用link，并生成一个目标文件，稍后可以与其他文件链接以生成二进制文件。

```
cl main.obj niam.cpp  
// 生成目标文件 "niam.obj".  
// 使用 "main.obj" 和 "niam.obj" 进行链接。  
// 生成可执行文件 "main.exe".  
  
link main.obj niam.obj  
// 使用 "main.obj" 和 "niam.obj" 进行链接。  
// 生成可执行文件 "main.exe".
```

还有其他有用的命令行参数，用户了解这些参数会非常有帮助：

```
cl /EHsc main.cpp  
// "/EHsc" 指定只捕获标准 C++ ("同步") 异常，  
// 并且 `extern "C"` 函数不会抛出异常。  
// 编写可移植、平台无关代码时推荐使用此选项。  
  
cl /clr main.cpp  
// "/clr" 指定代码应编译为使用公共语言运行时，  
// 即 .NET 框架的虚拟机。  
// 除了标准 ("本机") C++ 外，还支持使用微软的 C++/CLI 语言，  
// 并生成需要 .NET 运行的可执行文件。  
  
cl /Za main.cpp  
// "/Za" 指定禁用微软扩展，代码应严格按照 ISO C++ 规范编译。  
  
// 推荐用于保证代码的可移植性。  
  
cl /Zi main.cpp  
// "/Zi" 生成一个程序数据库 (PDB) 文件，用于调试程序，  
// 不影响优化设置，并将选项 "/DEBUG" 传递给 LINK。  
  
cl /LD dll.cpp  
// "/LD" 告诉 CL 配置 LINK 生成 DLL 而不是可执行文件。  
// LINK 将输出 DLL，此外还会生成用于链接时使用的 LIB 和 EXP 文件。  
// 要在其他程序中使用 DLL，编译这些程序时需将其关联的 LIB 传递给 CL 或 LINK。  
  
cl main.cpp /link /LINKER_OPTION  
// "/link" 会将其后面的所有内容直接传递给 LINK，不进行任何解析。
```

```
cl a.cpp b.cpp c.cpp  
// Generates "a.exe".  
  
cl d.obj a.cpp q.cpp  
// Generates "d.exe".  
  
cl y.lib n.cpp o.obj  
// Generates "n.exe".  
  
cl /o yo zp.obj pz.cpp  
// Generates "yo.exe".
```

To compile a file(s) without linking, use:

```
cl /c main.cpp  
// Generates object file "main.obj".
```

This tells cl to exit without calling link, and produces an object file, which can later be linked with other files to produce a binary.

```
cl main.obj niam.cpp  
// Generates object file "niam.obj".  
// Performs linking with "main.obj" and "niam.obj".  
// Generates executable "main.exe".  
  
link main.obj niam.obj  
// Performs linking with "main.obj" and "niam.obj".  
// Generates executable "main.exe".
```

There are other valuable command line parameters as well, which it would be very useful for users to know:

```
cl /EHsc main.cpp  
// "/EHsc" specifies that only standard C++ ("synchronous") exceptions will be caught,  
// and `extern "C"` functions will not throw exceptions.  
// This is recommended when writing portable, platform-independent code.  
  
cl /clr main.cpp  
// "/clr" specifies that the code should be compiled to use the common language runtime,  
// the .NET Framework's virtual machine.  
// Enables the use of Microsoft's C++/CLI language in addition to standard ("native") C++,  
// and creates an executable that requires .NET to run.  
  
cl /Za main.cpp  
// "/Za" specifies that Microsoft extensions should be disabled, and code should be  
// compiled strictly according to ISO C++ specifications.  
// This is recommended for guaranteeing portability.  
  
cl /Zi main.cpp  
// "/Zi" generates a program database (PDB) file for use when debugging a program, without  
// affecting optimisation specifications, and passes the option "/DEBUG" to LINK.  
  
cl /LD dll.cpp  
// "/LD" tells CL to configure LINK to generate a DLL instead of an executable.  
// LINK will output a DLL, in addition to an LIB and EXP file for use when linking.  
// To use the DLL in other programs, pass its associated LIB to CL or LINK when compiling those  
// programs.  
  
cl main.cpp /link /LINKER_OPTION  
// "/link" passes everything following it directly to LINK, without parsing it in any way.
```

```
// 将 "/LINKER_OPTION" 替换为任何所需的 LINK 选项。
```

对于更熟悉 *nix 系统和/或 GCC/Clang 的用户，cl、link 以及其他 Visual Studio 命令行工具可以接受用连字符指定的参数（例如 -c），而不是斜杠（例如 /c）。此外，Windows 识别斜杠或反斜杠作为有效的路径分隔符，因此也可以使用 *nix 风格的路径。这使得将简单的编译器命令行从 g++ 或 clang++ 转换为 cl，或反之，变得非常容易，且只需做最小的修改。

```
g++ -o app src/main.cpp  
cl -o app src/main.cpp
```

当然，当移植使用更复杂的 g++ 或 clang++ 选项的命令行时，需要查阅相应编译器文档和/或资源网站上的等效命令，但这使得以最少的时间学习新编译器即可更容易地开始工作。

如果您的代码需要特定的语言特性，则需要特定版本的 MSVC。从 Visual C++ 2015 更新 3 开始，可以通过 `/std` 标志选择编译所用的标准版本。可用的值有 `/std:c++14` 和 `/std:c++latest` (`/std:c++17` 将很快支持)。

注意：在该编译器的旧版本中，曾提供特定的特性标志，但这主要用于新特性的预览。

第138.5节：使用 Clang 编译

由于 Clang 前端设计为兼容 GCC，大多数可以通过 GCC 编译的程序，在构建脚本中将 g++ 替换为 clang++ 后也能编译。如果未指定 `-std=version`，则默认使用 gnu11。

习惯使用 MSVC 的 Windows 用户可以将 cl.exe 替换为 clang-cl.exe。默认情况下，clang 尝试兼容已安装的最高版本 MSVC。

在使用 Visual Studio 编译时，可以通过更改项目属性中的 Platform toolset 来使用 clang-cl。

在这两种情况下，clang 仅通过其前端实现兼容，尽管它也尝试生成二进制兼容的目标文件。clang-cl 用户应注意，与 MSVC 的兼容性尚未完全实现。

要使用 clang 或 clang-cl，可以使用某些 Linux 发行版上的默认安装版本，或使用与 IDE 捆绑的版本（如 Mac 上的 XCode）。对于其他版本的该编译器或未预装的平台，可以从官方下载页面下载。

如果您使用CMake来构建代码，通常可以通过设置CC和CXX环境变量来切换编译器，方法如下：

```
mkdir build  
cd build  
CC=clang CXX=clang++ cmake ..  
cmake --build .
```

另请参见Cmake简介。

第138.6节：C++编译过程

当你开发C++程序时，下一步是在运行之前编译程序。编译是将用人类可读语言（如C、C++等）编写的程序转换为机器代码的过程，

```
// Replace "/LINKER_OPTION" with any desired LINK option(s).
```

For anyone more familiar with *nix systems and/or GCC/Clang, cl, link, and other Visual Studio command line tools can accept parameters specified with a hyphen (such as -c) instead of a slash (such as /c). Additionally, Windows recognises either a slash or a backslash as a valid path separator, so *nix-style paths can be used as well. This makes it easy to convert simple compiler command lines from g++ or clang++ to cl, or vice versa, with minimal changes.

```
g++ -o app src/main.cpp  
cl -o app src/main.cpp
```

Of course, when porting command lines that use more complex g++ or clang++ options, you need to look up equivalent commands in the applicable compiler documentations and/or on resource sites, but this makes it easier to get things started with minimal time spent learning about new compilers.

In case you need specific language features for your code, a specific release of MSVC was required. From [Visual C++ 2015 Update 3](#) on it is possible to choose the version of the standard to compile with via the `/std` flag. Possible values are `/std:c++14` and `/std:c++latest` (`/std:c++17` will follow soon).

Note: In older versions of this compiler, specific feature flags were available however this was mostly used for previews of new features.

Section 138.5: Compiling with Clang

As the Clang front-end is designed for being compatible with GCC, most programs that can be compiled via GCC will compile when you swap g++ by clang++ in the build scripts. If no `-std=version` is given, gnu11 will be used.

Windows users who are used to MSVC can swap cl.exe with clang-cl.exe. By default, clang tries to be compatible with the highest version of MSVC that has been installed.

In the case of compiling with visual studio, clang-cl can be used by changing the Platform toolset in the project properties.

In both cases, clang is only compatible via its front-end, though it also tries to generate binary compatible object files. Users of clang-cl should note that [the compatibility with MSVC is not complete yet](#).

To use clang or clang-cl, one could use the default installation on certain Linux distributions or those bundled with IDEs (like XCode on Mac). For other versions of this compiler or on platforms which don't have this installed, this can be download from the [official download page](#).

If you're using CMake to build your code you can usually switch the compiler by setting the CC and CXX environment variables like this:

```
mkdir build  
cd build  
CC=clang CXX=clang++ cmake ..  
cmake --build .
```

See also introduction to Cmake.

Section 138.6: The C++ compilation process

When you develop a C++ program, the next step is to compile the program before running it. The compilation is the process which converts the program written in human readable language like C, C++ etc into a machine code,

该机器代码可被中央处理单元直接理解。例如，如果你有一个名为prog.cpp的C++源代码文件，执行编译命令时，

```
g++ -Wall -ansi -o prog prog.cpp
```

从源文件创建可执行文件主要涉及4个阶段。

1. C++预处理器处理C++源代码文件中的头文件（#include）、宏定义（#define）和其他预处理指令。
2. 由C++预处理器生成的扩展C++源代码文件被编译成该平台的汇编语言。
3. 编译器生成的汇编代码被汇编成该平台的目标代码。
4. 汇编器生成的目标代码文件与所使用的库函数的目标代码文件链接，生成库文件或可执行文件。

预处理

预处理器处理预处理指令，如#include和#define。它不关心C++的语法，因此必须谨慎使用。

它一次处理一个C++源文件，通过用相应文件的内容（通常只是声明）替换#include指令，替换宏定义（#define），并根据#if、#ifdef和#ifndef指令选择不同的文本部分。

预处理器处理的是一串预处理标记。宏替换定义为用其他标记替换标记（操作符##在合适的情况下用于合并两个标记）。

完成上述操作后，预处理器生成一个标记流作为输出，包含上述转换结果。它还添加一些特殊标记，告诉编译器每行代码的来源，以便编译器生成合理的错误信息。

在此阶段，通过巧妙使用#if和#error指令可能会产生一些错误。

通过使用以下编译器标志，我们可以在预处理阶段停止进程。

```
g++ -E prog.cpp
```

编译

编译步骤针对预处理器的每个输出执行。编译器解析纯C++源代码（此时不含任何预处理指令），并将其转换为汇编代码。然后调用底层后端（工具链中的汇编器）将该代码汇编成机器码，生成某种格式（ELF、COFF、a.out等）的实际二进制文件。该目标文件包含输入中定义的符号的编译代码（二进制形式）。目标文件中的符号通过名称引用。

目标文件可以引用未定义的符号。当你使用声明但未提供定义时，就会出现这种情况。编译器对此不介意，只要源代码格式正确，就会愉快地生成目标文件。

编译器通常允许你在此阶段停止编译。这非常有用，因为你可以分别编译每个源代码文件。这样做的好处是，如果只修改了单个文件，就不需要重新编译所有内容。

directly understood by the Central Processing Unit. For example, if you have a C++ source code file named prog.cpp and you execute the compile command,

```
g++ -Wall -ansi -o prog prog.cpp
```

There are 4 main stages involved in creating an executable file from the source file.

1. The C++ preprocessor takes a C++ source code file and deals with the headers(#include), macros(#define) and other preprocessor directives.
2. The expanded C++ source code file produced by the C++ preprocessor is compiled into the assembly language for the platform.
3. The assembler code generated by the compiler is assembled into the object code for the platform.
4. The object code file produced by the assembler is linked together with the object code files for any library functions used to produce either a library or an executable file.

Preprocessing

The preprocessor handles the preprocessor directives, like #include and #define. It is agnostic of the syntax of C++, which is why it must be used with care.

It works on one C++ source file at a time by replacing #include directives with the content of the respective files (which is usually just declarations), doing replacement of macros (#define), and selecting different portions of text depending of #if, #ifdef and #ifndef directives.

The preprocessor works on a stream of preprocessing tokens. Macro substitution is defined as replacing tokens with other tokens (the operator ## enables merging two tokens when it makes sense).

After all this, the preprocessor produces a single output that is a stream of tokens resulting from the transformations described above. It also adds some special markers that tell the compiler where each line came from so that it can use those to produce sensible error messages.

Some errors can be produced at this stage with clever use of the #if and #error directives.

By using below compiler flag, we can stop the process at preprocessing stage.

```
g++ -E prog.cpp
```

Compilation

The compilation step is performed on each output of the preprocessor. The compiler parses the pure C++ source code (now without any preprocessor directives) and converts it into assembly code. Then invokes underlying backend(assembler in toolchain) that assembles that code into machine code producing actual binary file in some format(ELF, COFF, a.out, ...). This object file contains the compiled code (in binary form) of the symbols defined in the input. Symbols in object files are referred to by name.

Object files can refer to symbols that are not defined. This is the case when you use a declaration, and don't provide a definition for it. The compiler doesn't mind this, and will happily produce the object file as long as the source code is well-formed.

Compilers usually let you stop compilation at this point. This is very useful because with it you can compile each source code file separately. The advantage this provides is that you don't need to recompile everything if you only change a single file.

生成的目标文件可以放入称为静态库的特殊归档中，便于以后重复使用。

正是在此阶段，会报告“常规”的编译器错误，如语法错误或重载解析失败错误。

为了在编译步骤后停止进程，我们可以使用 -S 选项：

```
g++ -Wall -ansi -S prog.cpp
```

汇编

汇编器创建目标代码。在UNIX系统上，你可能会看到带有.o后缀的文件（MSDOS上为.OBJ），表示目标代码文件。在此阶段，汇编器将汇编代码转换为机器级指令，生成的文件是可重定位的目标代码。因此，编译阶段生成了可重定位的目标程序，该程序可以在不同位置使用，而无需重新编译。

要在汇编步骤后停止进程，可以使用-c选项：

```
g++ -Wall -ansi -c prog.cpp
```

链接

链接器负责从汇编器生成的目标文件中生成最终的编译输出。该输出可以是共享（或动态）库（虽然名称相似，但与前面提到的静态库差别很大）或可执行文件。

它通过用正确的地址替换对未定义符号的引用来链接所有目标文件。这些符号中的每一个都可以在其他目标文件或库中定义。如果它们定义在标准库以外的库中，你需要告诉链接器这些库。

此阶段最常见的错误是缺少定义或重复定义。前者意味着定义不存在（即未编写），或者包含定义的目标文件或库未提供给链接器。后者很明显：同一符号在两个不同的目标文件或库中被定义。

第138.7节：使用Code::Blocks进行编译（图形界面）

1. 在此下载并安装 Code::Blocks。如果您使用的是 Windows，请注意选择文件名为包含mingw，其他文件不安装任何编译器。

2. 打开 Code::Blocks 并点击“创建新项目”：

The produced object files can be put in special archives called static libraries, for easier reusing later on.

It's at this stage that "regular" compiler errors, like syntax errors or failed overload resolution errors, are reported.

In order to stop the process after the compile step, we can use the -S option:

```
g++ -Wall -ansi -S prog.cpp
```

Assembling

The assembler creates object code. On a UNIX system you may see files with a .o suffix (.OBJ on MSDOS) to indicate object code files. In this phase the assembler converts those object files from assembly code into machine level instructions and the file created is a relocatable object code. Hence, the compilation phase generates the relocatable object program and this program can be used in different places without having to compile again.

To stop the process after the assembly step, you can use the -c option:

```
g++ -Wall -ansi -c prog.cpp
```

Linking

The linker is what produces the final compilation output from the object files the assembler produced. This output can be either a shared (or dynamic) library (and while the name is similar, they don't have much in common with static libraries mentioned earlier) or an executable.

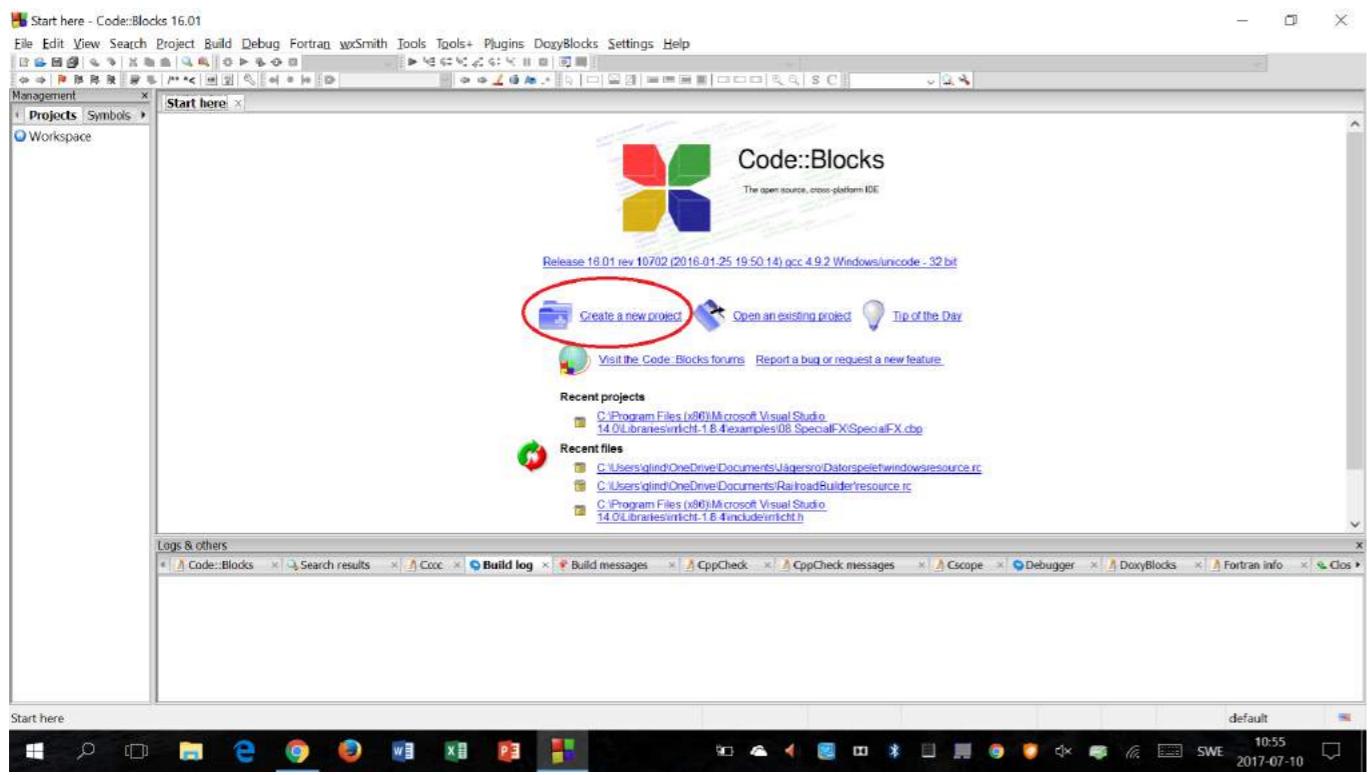
It links all the object files by replacing the references to undefined symbols with the correct addresses. Each of these symbols can be defined in other object files or in libraries. If they are defined in libraries other than the standard library, you need to tell the linker about them.

At this stage the most common errors are missing definitions or duplicate definitions. The former means that either the definitions don't exist (i.e. they are not written), or that the object files or libraries where they reside were not given to the linker. The latter is obvious: the same symbol was defined in two different object files or libraries.

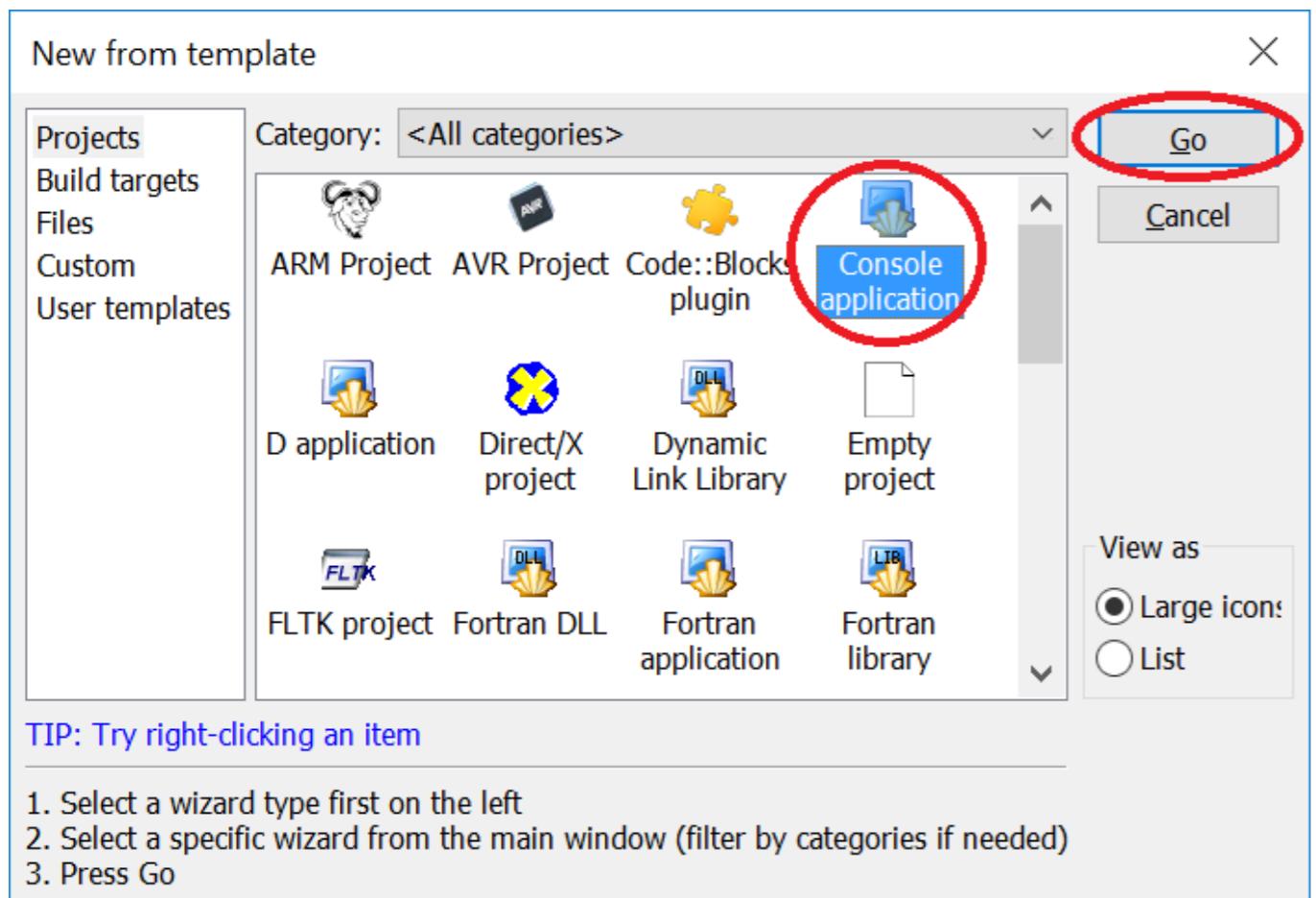
Section 138.7: Compiling with Code::Blocks (Graphical interface)

1. Download and install Code::Blocks [here](#). If you're on Windows, be careful to select a file for which the name contains mingw, the other files don't install any compiler.

2. Open Code::Blocks and click on "Create a new project":

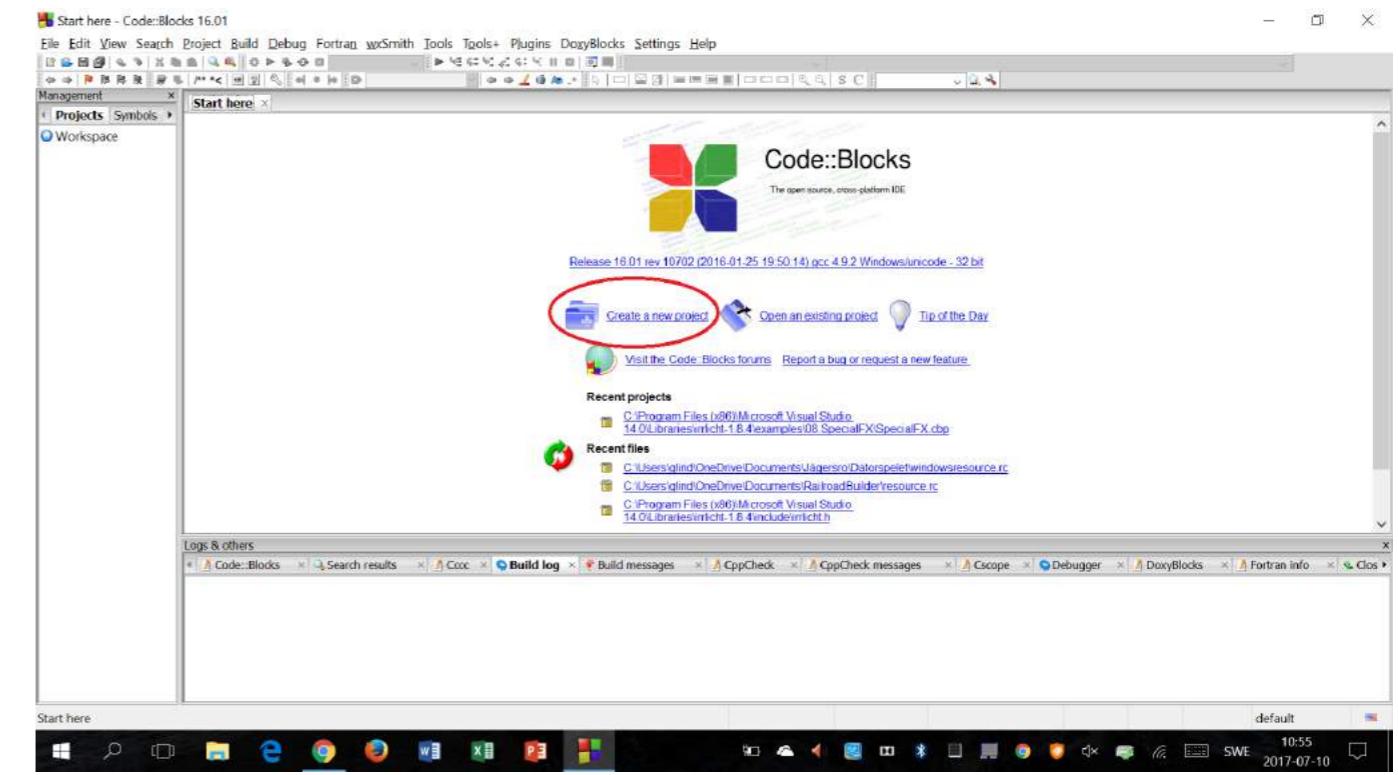


3. 选择“控制台应用程序”并点击“开始”：

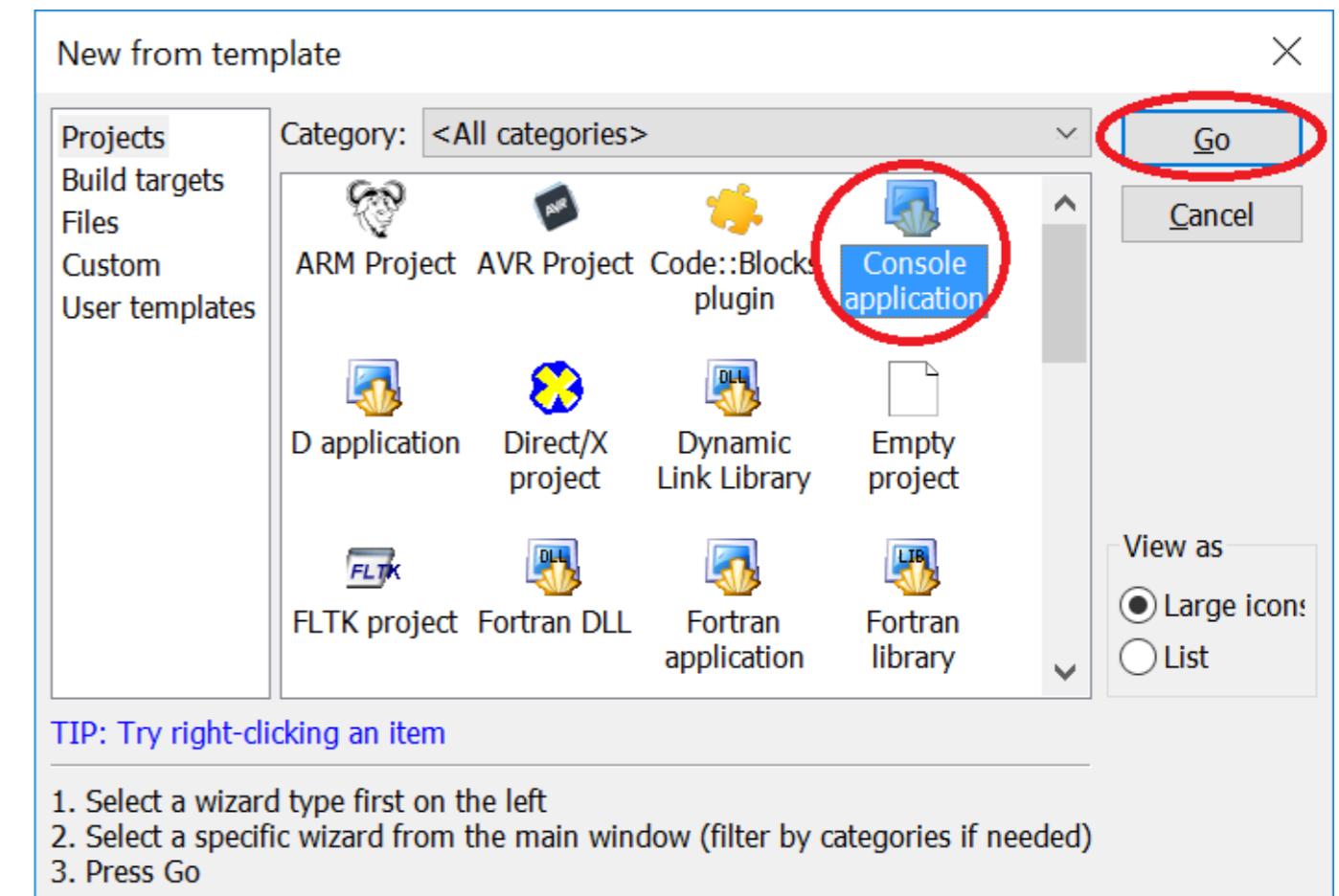


4. 点击“下一步”，选择“C++”，点击“下一步”，为你的项目选择一个名称并选择保存文件夹，点击“下一步”，然后点击“完成”。

5. 现在你可以编辑和编译代码了。控制台中已经有一段默认代码打印“Hello world!”

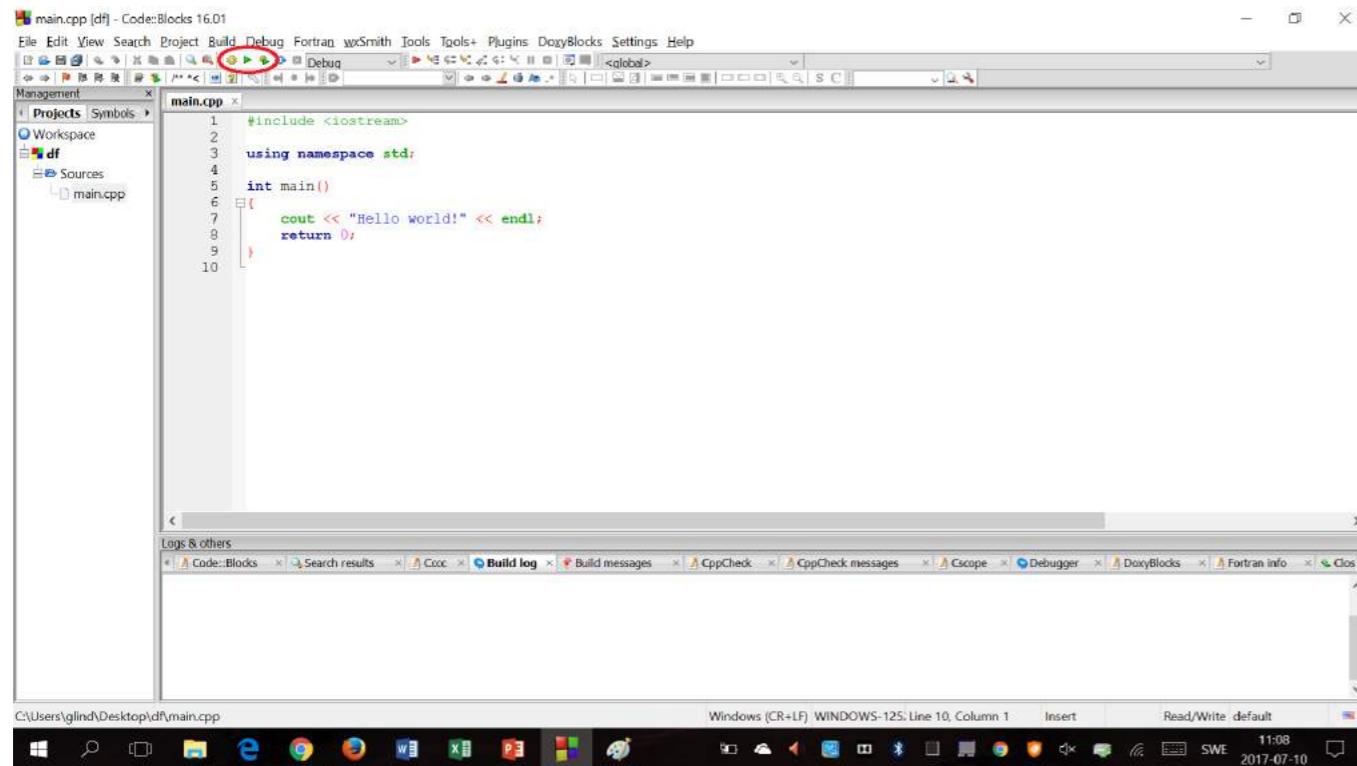


3. Select "Console application" and click "Go":



4. Click "Next", select "C++", click "Next", select a name for your project and choose a folder to save it in, click "Next" and then click "Finish".
5. Now you can edit and compile your code. A default code that prints "Hello world!" in the console is already

要编译和/或运行程序，按工具栏中的三个编译/运行按钮之一：



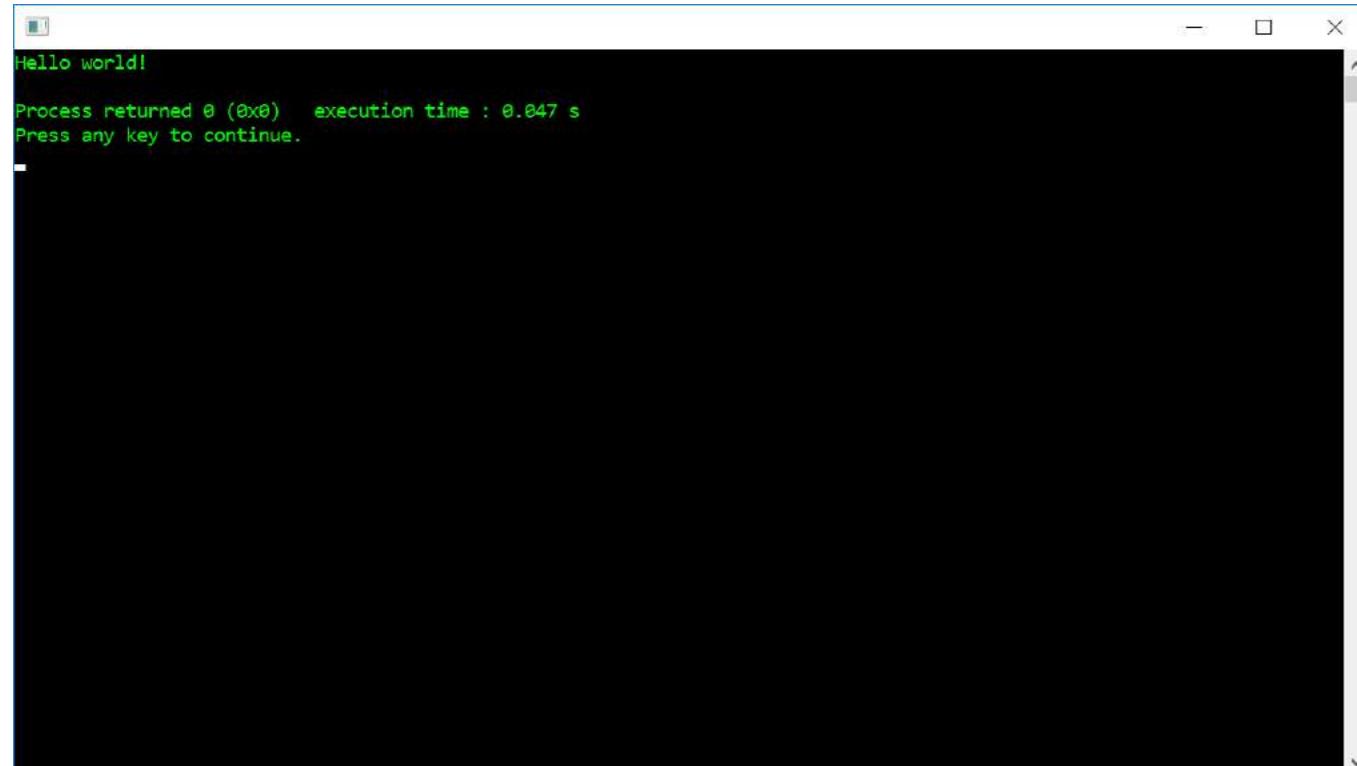
The screenshot shows the Code::Blocks IDE interface. The toolbar at the top has several icons, with the first three (gear, green triangle, blue triangle) circled in red. The main window displays a C++ code editor with the following code:

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

The status bar at the bottom indicates the file path as C:\Users\glind\Desktop\df\main.cpp, the encoding as Windows (CR+LF) WINDOWS-125, the line as Line 10, the column as Column 1, and the time as 11:08 2017-07-10.

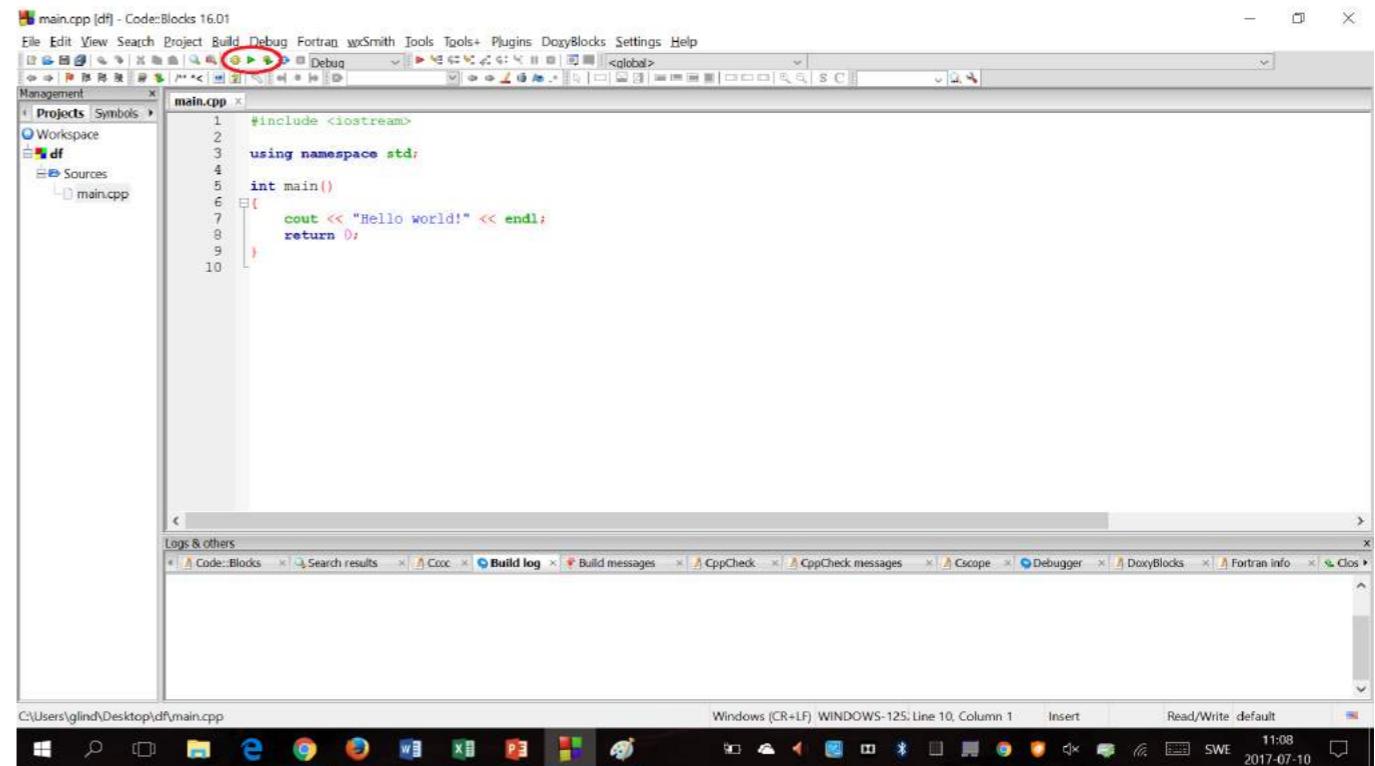
要编译但不运行，按 ，要运行但不重新编译，按  并且要编译然后运行，按下 .

编译并运行默认的“Hello world!”代码会得到以下结果：



```
Hello world!
Process returned 0 (0x0)   execution time : 0.047 s
Press any key to continue.
```

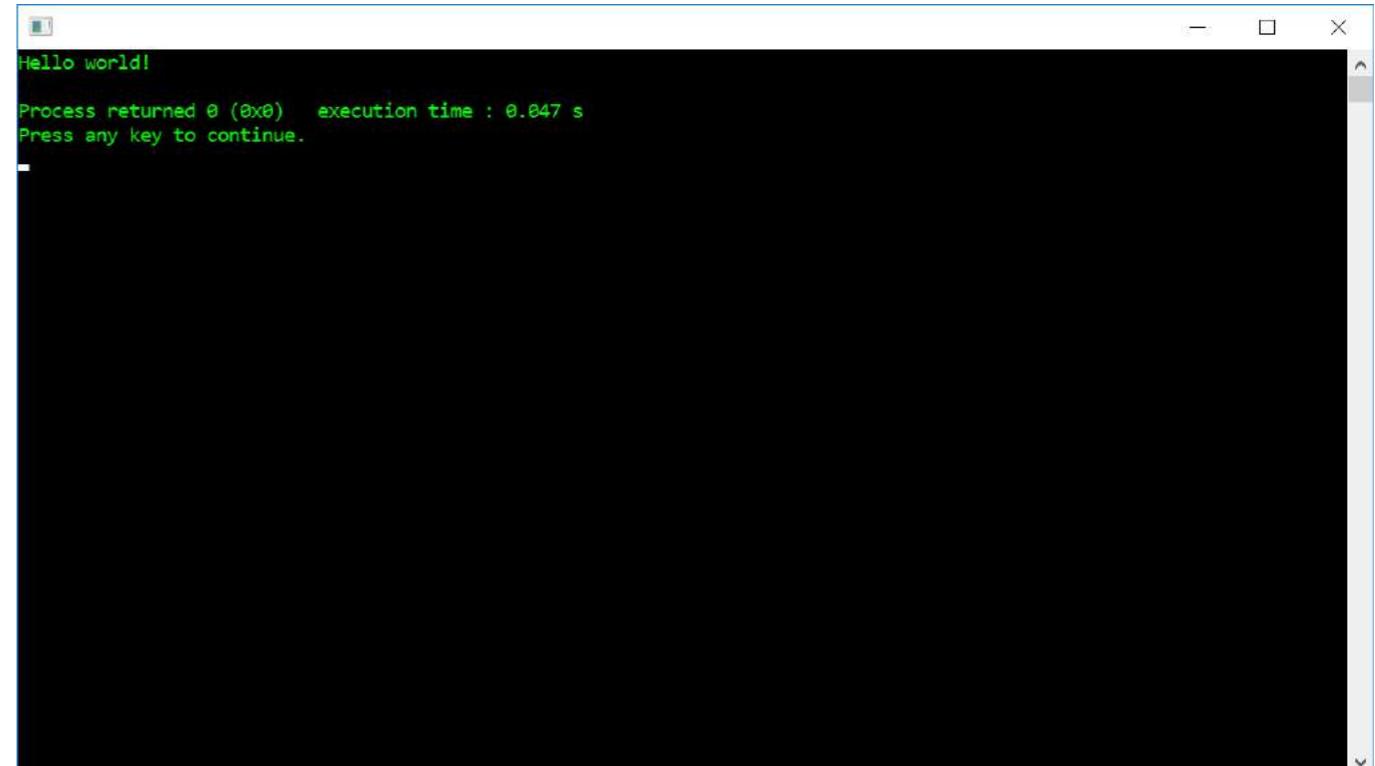
there. To compile and/or run your program, press one of the three compile/run buttons in the toolbar:



The screenshot shows the Code::Blocks IDE interface, similar to the one on the left. The toolbar at the top has several icons, with the first three (gear, green triangle, blue triangle) circled in red. The main window displays the same C++ code as the left screenshot.

To compile without running, press , to run without compiling again, press  and to compile and then run, press .

Compiling and running the default "Hello world!" code gives the following result:



```
Hello world!
Process returned 0 (0x0)   execution time : 0.047 s
Press any key to continue.
```

第139章：常见的编译/链接器错误 (GCC)

第139.1节：未定义的对 `***` 的引用

当链接器找不到使用的符号时，会发生此链接器错误。大多数情况下，这是因为未链接所使用的库。

qmake :

```
LIBS += nameOfLib
```

cmake :

```
TARGET_LINK_LIBRARIES(target nameOfLib)
```

g++ 调用：

```
g++ -o main main.cpp -Llibrary/dir -lnameOfLib
```

有人也可能忘记编译和链接所有使用的.cpp文件 (functionsModule.cpp定义了所需的函数)：

```
g++ -o binName main.o functionsModule.o
```

第139.2节：错误：'***'未在此作用域中声明

如果使用了未知对象，就会发生此错误。

变量

未编译：

```
#include <iostream>

int main(int argc, char *argv[])
{
    {
        int i = 2;
    }

    std::cout << i << std::endl; // i不在main函数的作用域内

    return 0;
}
```

修复：

```
#include <iostream>

int main(int argc, char *argv[])
{
    {
        int i = 2;
        std::cout << i << std::endl;
    }
}
```

Chapter 139: Common compile/linker errors (GCC)

Section 139.1: undefined reference to `***'

This linker error happens, if the linker can't find a used symbol. Most of the time, this happens if a used library is not linked against.

qmake:

```
LIBS += nameOfLib
```

cmake:

```
TARGET_LINK_LIBRARIES(target nameOfLib)
```

g++ call:

```
g++ -o main main.cpp -Llibrary/dir -lnameOfLib
```

One might also forget to compile and link all used .cpp files (functionsModule.cpp defines the needed function):

```
g++ -o binName main.o functionsModule.o
```

Section 139.2: error: '***' was not declared in this scope

This error happens if a unknown object is used.

Variables

Not compiling:

```
#include <iostream>

int main(int argc, char *argv[])
{
    {
        int i = 2;
    }

    std::cout << i << std::endl; // i is not in the scope of the main function

    return 0;
}
```

Fix:

```
#include <iostream>

int main(int argc, char *argv[])
{
    {
        int i = 2;
        std::cout << i << std::endl;
    }
}
```

```
    }

    return 0;
}
```

函数

大多数情况下，如果没有包含所需的头文件（例如使用`std::cout`但未包含`#include <iostream>`）

未编译：

```
#include <iostream>

int main(int argc, char *argv[])
{
doCompile();

    return 0;
}

void doCompile()
{
std::cout << "不！" << std::endl;
}
```

修复：

```
#include <iostream>

void doCompile(); // 函数的前向声明

int main(int argc, char *argv[])
{
doCompile();

    return 0;
}

void doCompile()
{
std::cout << "不！" << std::endl;
}
```

或者：

```
#include <iostream>

void doCompile() // 在使用前定义函数
{
std::cout << "不！" << std::endl;
}

int main(int argc, char *argv[])
{
doCompile();

    return 0;
}
```

```
    }

    return 0;
}
```

Functions

Most of the time this error occurs if the needed header is not included (e.g. using `std::cout` without `#include <iostream>`)

Not compiling:

```
#include <iostream>

int main(int argc, char *argv[])
{
doCompile();

    return 0;
}

void doCompile()
{
std::cout << "No!" << std::endl;
}
```

Fix:

```
#include <iostream>

void doCompile(); // forward declare the function

int main(int argc, char *argv[])
{
doCompile();

    return 0;
}

void doCompile()
{
std::cout << "No!" << std::endl;
}
```

Or:

```
#include <iostream>

void doCompile() // define the function before using it
{
std::cout << "No!" << std::endl;
}

int main(int argc, char *argv[])
{
doCompile();

    return 0;
}
```

}

注意： 编译器从上到下解释代码（简化说明）。所有内容必须至少在使用前声明（或定义）。

第139.3节：致命错误：***：没有此类文件或目录

编译器找不到文件（源文件使用了#include "someFile.hpp"）。

qmake :

```
INCLUDEPATH += 目录/的/文件
```

cmake :

```
include_directories(目录/的/文件)
```

g++ 调用：

```
g++ -o main main.cpp -I目录/的/文件
```

}

Note: The compiler interprets the code from top to bottom (simplification). Everything must be at least declared (or defined) before usage.

Section 139.3: fatal error: ***: No such file or directory

The compiler can't find a file (a source file uses #include "someFile.hpp").

qmake:

```
INCLUDEPATH += dir/Of/File
```

cmake:

```
include_directories(dir/Of/File)
```

g++ call:

```
g++ -o main main.cpp -Idir/Of/File
```

第140章：C++中的更多未定义行为

更多关于C++可能出错的示例。

未定义行为的延续

第140.1节：在初始化列表中引用非静态成员

在构造函数开始执行之前，在初始化列表中引用非静态成员可能导致未定义行为。这是因为此时并非所有成员都已被构造。根据标准草案：

§ 12.7.1：对于具有非平凡构造函数的对象，在构造函数开始执行之前，引用该对象的任何非静态成员或基类将导致未定义行为。

示例

```
结构体 W { int j; };
结构体 X : public virtual W { };
结构体 Y {
    int *p;
X x;
Y() : p(&x.j) { // 未定义, x尚未构造
}
};
```

Chapter 140: More undefined behaviors in C++

More examples on how C++ can go wrong.

Continuation from Undefined Behavior

Section 140.1: Referring to non-static members in initializer lists

Referring to non-static members in initializer lists before the constructor has started executing can result in undefined behavior. This results since not all members are constructed at this time. From the standard draft:

§12.7.1: For an object with a non-trivial constructor, referring to any non-static member or base class of the object before the constructor begins execution results in undefined behavior.

Example

```
struct W { int j; };
struct X : public virtual W { };
struct Y {
    int *p;
X x;
Y() : p(&x.j) { // undefined, x is not yet constructed
}
};
```

第141章：C++中的单元测试

单元测试是软件测试的一个层级，用于验证代码单元的行为和正确性。

在C++中，“代码单元”通常指类、函数或它们的组合。单元测试通常使用专门的“测试框架”或“测试库”进行，这些框架或库往往使用复杂的语法或使用模式。

本章节将回顾不同的策略以及单元测试库或框架。

第141.1节：Google测试框架

Google测试框架是由谷歌维护的C++测试框架。构建测试用例文件时，需要构建gtest库并将其链接到你的测试框架中。

最小示例

```
// main.cpp

#include <gtest/gtest.h>
#include <iostream>

// Google测试用例是通过C++预处理器宏创建的
// 这里提供了“测试套件”名称和具体的“测试名称”。
TEST(module_name, test_name) {
    std::cout << "Hello world!" << std::endl;
    // Google Test 还将提供断言的宏。
    ASSERT_EQ(1+1, 2);
}

// Google Test 可以从 main() 函数手动运行
// 或者，它可以链接到 gtest_main 库，
// 该库提供了一个已设置好的 main() 函数，准备接受 Google Test 测试用例。
int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);

    return RUN_ALL_TESTS();
}

// 构建命令：g++ main.cpp -lgtest
```

第141.2节：Catch

Catch 是一个仅包含头文件的库，允许你使用 TDD 和 BDD 单元测试风格。

以下代码片段摘自此链接的 Catch 文档页面：

```
SCENARIO( "向量可以设置大小和调整大小", "[vector]" ) {
    GIVEN( "一个包含若干元素的向量" ) {
        std::vector v( 5 );

        REQUIRE( v.size() == 5 );
        REQUIRE( v.capacity() >= 5 );

        WHEN( "大小增加时" ) {
            v.resize( 10 );

            THEN( "大小和容量发生变化" ) {
                REQUIRE( v.size() == 10 );
            }
        }
    }
}
```

Chapter 141: Unit Testing in C++

Unit testing is a level in software testing that validates the behavior and correctness of units of code.

In C++, “units of code” often refer to either classes, functions, or groups of either. Unit testing is often performed using specialized “testing frameworks” or “testing libraries” that often use non-trivial syntax or usage patterns.

This topic will review different strategies and unit testing libraries or frameworks.

Section 141.1: Google Test

Google Test is a C++ testing framework maintained by Google. It requires building the gtest library and linking it to your testing framework when building a test case file.

Minimal Example

```
// main.cpp

#include <gtest/gtest.h>
#include <iostream>

// Google Test test cases are created using a C++ preprocessor macro
// Here, a "test suite" name and a specific "test name" are provided.
TEST(module_name, test_name) {
    std::cout << "Hello world!" << std::endl;
    // Google Test will also provide macros for assertions.
    ASSERT_EQ(1+1, 2);
}

// Google Test can be run manually from the main() function
// or, it can be linked to the gtest_main library for an already
// set-up main() function primed to accept Google Test test cases.
int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);

    return RUN_ALL_TESTS();
}

// Build command: g++ main.cpp -lgtest
```

Section 141.2: Catch

Catch is a header only library that allows you to use both TDD and BDD unit test style.

The following snippet is from the Catch documentation page at [this link](#):

```
SCENARIO( "vectors can be sized and resized", "[vector]" ) {
    GIVEN( "A vector with some items" ) {
        std::vector v( 5 );

        REQUIRE( v.size() == 5 );
        REQUIRE( v.capacity() >= 5 );

        WHEN( "the size is increased" ) {
            v.resize( 10 );

            THEN( "the size and capacity change" ) {
                REQUIRE( v.size() == 10 );
            }
        }
    }
}
```

```

REQUIRE( v.capacity() >= 10 );
}
WHEN( "大小被缩减" ) {
v.resize( 0 );

THEN( "大小变化但容量不变" ) {
    REQUIRE( v.size() == 0 );
REQUIRE( v.capacity() >= 5 );
}
WHEN( "预留更多容量" ) {
v.reserve( 10 );

THEN( "容量变化但大小不变" ) {
    REQUIRE( v.size() == 5 );
REQUIRE( v.capacity() >= 10 );
}
当( "保留的容量较少" ) {
v.reserve( 0 );

那么( "大小和容量都不变" ) {
    REQUIRE( v.size() == 5 );
REQUIRE( v.capacity() >= 5 );
}
}
}
}

```

方便的是，这些测试在运行时将按如下方式报告：

场景：向量可以调整大小和重新调整大小前提：一个包含若干元素的向量当：保留更多容量那么：
容量发生变化，但大小不变

```

REQUIRE( v.capacity() >= 10 );
}
WHEN( "the size is reduced" ) {
v.resize( 0 );

THEN( "the size changes but not capacity" ) {
    REQUIRE( v.size() == 0 );
    REQUIRE( v.capacity() >= 5 );
}
WHEN( "more capacity is reserved" ) {
v.reserve( 10 );

THEN( "the capacity changes but not the size" ) {
    REQUIRE( v.size() == 5 );
    REQUIRE( v.capacity() >= 10 );
}
WHEN( "less capacity is reserved" ) {
v.reserve( 0 );

THEN( "neither size nor capacity are changed" ) {
    REQUIRE( v.size() == 5 );
    REQUIRE( v.capacity() >= 5 );
}
}
}
}

```

Conveniently, these tests will be reported as follows when run:

Scenario: vectors can be sized and resized Given: A vector with some items When: more capacity is reserved Then:
the capacity changes but not the size

第142章：C++调试及调试预防工具与技术

大量C++开发者的时间花费在调试上。本主题旨在协助完成此任务并提供技术灵感。不要期望这里有工具修复的问题和解决方案的详尽列表，或是提及工具的使用手册。

第142.1节：静态分析

静态分析是一种检查代码中与已知错误相关的模式的技术。使用这种技术比代码审查耗时更少，尽管其检查仅限于工具中编程的内容。

检查可以包括 if 语句后面错误的分号 (if (var);) 直到确定变量未初始化的高级图算法

编译器警告

启用静态分析很简单，最简单的版本已经内置在你的编译器中：

- `clang++ -Wall -Weverything -Werror ...`
- `g++ -Wall -Weverything -Werror ...`
- `cl.exe /W4 /WX ...`

如果你启用这些选项，你会发现每个编译器都会发现其他编译器找不到的错误，并且你会在某些技术上收到错误提示，这些技术可能是有效的，或者在特定上下文中有效。`while (staticAtomicBool);`可能是可接受的，即使`while (localBool);`不是。

所以，与代码审查不同，你是在与一个理解你代码的工具作斗争，它会告诉你很多有用的错误，有时还会与你意见不合。在这种情况下，你可能需要在本地抑制该警告。

由于上述选项启用了所有警告，它们可能会启用你不想要的警告。（为什么你的代码需要兼容 C++98？）如果是这样，你可以简单地禁用那个特定的警告：

- `clang++ -Wall -Weverything -Werror -Wno-errortoaccept ...`
- `g++ -Wall -Weverything -Werror -Wno-errortoaccept ...`
- `cl.exe /W4 /WX /wd<no of warning>...`

当编译器警告在开发过程中帮助你时，它们会显著减慢编译速度。这就是为什么你可能并不总是想默认启用它们。你可以选择默认运行它们，或者启用一些持续集成来进行更严格的检查（或者全部检查）。

外部工具

如果你决定使用持续集成，使用其他工具也不是难事。像clang-tidy这样的工具拥有一系列检查，涵盖了广泛的问题，以下是一些示例：

- 实际的错误
 - 防止切片
 - 带有副作用的断言
- 可读性检查
 - 误导性缩进
 - 标识符命名检查
- 现代化检查

Chapter 142: C++ Debugging and Debug-prevention Tools & Techniques

A lot of time from C++ developers is spent debugging. This topic is meant to assist with this task and give inspiration for techniques. Don't expect an extensive list of issues and solutions fixed by the tools or a manual on the mentioned tools.

Section 142.1: Static analysis

Static analysis is the technique in which on checks the code for patterns linked to known bugs. Using this technique is less time consuming than a code review, though, its checks are only limited to those programmed in the tool.

Checks can include the incorrect semi-colon behind the if-statement (`if (var);`) till advanced graph algorithms which determine if a variable is not initialized.

Compiler warnings

Enabling static analysis is easy, the most simplistic version is already build-in in your compiler:

- `clang++ -Wall -Weverything -Werror ...`
- `g++ -Wall -Weverything -Werror ...`
- `cl.exe /W4 /WX ...`

If you enable these options, you will notice that each compiler will find bugs the others don't and that you will get errors on techniques which might be valid or valid in a specific context. `while (staticAtomicBool);` might be acceptable even if `while (localBool);` ain't.

So unlike code review, you are fighting a tool which understands your code, tells you a lot of useful bugs and sometimes disagrees with you. In this last case, you might have to suppress the warning locally.

As the options above enable all warnings, they might enable warnings you don't want. (Why should your code be C++98 compatible?) If so, you can simply disable that specific warning:

- `clang++ -Wall -Weverything -Werror -Wno-errortoaccept ...`
- `g++ -Wall -Weverything -Werror -Wno-errortoaccept ...`
- `cl.exe /W4 /WX /wd<no of warning>...`

Where compiler warnings assist you during development, they slow down compilation quite a bit. That is why you might not always want to enable them by default. Either you run them by default or you enable some continuous integration with the more expensive checks (or all of them).

External tools

If you decide to have some continuous integration, the use of other tools ain't such a stretch. A tool like `clang-tidy` has an [list of checks](#) which covers a wide range of issues, some examples:

- Actual bugs
 - Prevention of slicing
 - Asserts with side effects
- Readability checks
 - Misleading indentation
 - Check identifier naming
- Modernization checks

- 使用`make_unique()`
- 使用`nullptr`
- 性能检查
 - 查找不必要的拷贝
 - 查找低效的算法调用

列表可能不会很大，因为Clang已经有很多编译器警告，然而它会让你更接近高质量的代码库一步。

其他工具

存在其他具有类似用途的工具，例如：

- 作为外部工具的 [Visual Studio 静态分析器](#)
- [clazy](#), 一款用于检查 Qt 代码的 Clang 编译器插件

结论

有许多针对 C++ 的静态分析工具，既有内置于编译器中的，也有外部工具。尝试使用它们对于简单的配置来说并不费时，而且它们能发现代码审查中可能遗漏的错误。

第142.2节：使用 GDB 进行段错误分析

让我们使用上面相同的代码作为本例。

```
#include <iostream>

void fail() {
    int *p1;
    int *p2(NULL);
    int *p3 = p1;
    if (p3) {
        std::cout << *p2 << std::endl;
    }
}

int main() {
    fail();
}
```

首先让我们编译它

```
g++ -g -o main main.cpp
```

让我们用 `gdb` 运行它

```
gdb ./main
```

现在我们将进入 `gdb` 命令行。输入 `run`。

```
(gdb) run
正在调试的程序已经启动。
从头开始吗？(y或n)
启动程序：/home/opencog/code-snippets/stackoverflow/a.out
```

程序收到信号 `SIGSEGV`, 段错误。
`0x000000000400850` 在 `fail()` 函数中，位于 `debugging_with_gdb.cc` 文件的第 11 行

- Use `make_unique()`
- Use `nullptr`
- Performance checks
 - Find unneeded copies
 - Find inefficient algorithm calls

The list might not be that large, as Clang already has a lot of compiler warnings, however it will bring you one step closer to a high quality code base.

Other tools

Other tools with similar purpose exist, like:

- [the visual studio static analyzer](#) as external tool
- [clazy](#), a Clang compiler plugin for checking Qt code

Conclusion

A lot static analysis tools exist for C++, both build-in in the compiler as external tools. Trying them out doesn't take that much time for easy setups and they will find bugs you might miss in code review.

Section 142.2: Segfault analysis with GDB

Lets use the same code as above for this example.

```
#include <iostream>

void fail() {
    int *p1;
    int *p2(NULL);
    int *p3 = p1;
    if (p3) {
        std::cout << *p2 << std::endl;
    }
}

int main() {
    fail();
}
```

First lets compile it

```
g++ -g -o main main.cpp
```

Lets run it with `gdb`

```
gdb ./main
```

Now we will be in `gdb` shell. Type `run`.

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/opencog/code-snippets/stackoverflow/a.out
```

Program received signal `SIGSEGV`, Segmentation fault.
`0x000000000400850` in `fail ()` at `debugging_with_gdb.cc:11`

```
11     std::cout << *p2 << std::endl;
```

我们看到段错误发生在第11行。所以这一行唯一使用的变量是指针p2。让我们通过打印来检查它的内容。

```
(gdb) 打印 p2  
$1 = (int *) 0x0
```

现在我们看到 p2 被初始化为 0x0，代表空指针（NULL）。在这一行，我们知道我们正试图解引用一个空指针。因此我们去修复它。

第142.3节：整洁代码

调试始于理解你正在调试的代码。

错误代码：

```
int main() {  
    int value;  
    std::vector<int> vectorToSort;  
    vectorToSort.push_back(42); vectorToSort.push_back(13);  
    for (int i = 52; i; i = i - 1)  
    {  
        vectorToSort.push_back(i *2);  
    }  
    /// 针对小型向量排序的优化  
    if (vectorToSort.size() == 1);  
    否则  
    {  
        if (vectorToSort.size() <= 2)  
            std::sort(vectorToSort.begin(), std::end(vectorToSort));  
    }  
    for (value : vectorToSort) std::cout << value << ' ';  
    return 0; }
```

更好的代码：

```
std::vector<int> createSemiRandomData() {  
    std::vector<int> data;  
    data.push_back(42);  
    data.push_back(13);  
    for (int i = 52; i; --i)  
        vectorToSort.push_back(i *2);  
    return data;  
}  
  
/// 针对小向量排序进行优化  
void sortVector(std::vector &v) {  
    if (vectorToSort.size() == 1)  
        return;  
    if (vectorToSort.size() > 2)  
        return;  
  
    std::sort(vectorToSort.begin(), vectorToSort.end());  
}  
  
void printVector(const std::vector<int> &v) {  
    for (auto i : v)  
        std::cout << i << ' ';
```

```
11     std::cout << *p2 << std::endl;
```

We see the segmentation fault is happening at line 11. So the only variable being used at this line is pointer p2. Lets examine its content typing print.

```
(gdb) print p2  
$1 = (int *) 0x0
```

Now we see that p2 was initialized to 0x0 which stands for NULL. At this line, we know that we are trying to dereference a NULL pointer. So we go and fix it.

Section 142.3: Clean code

Debugging starts with understanding the code you are trying to debug.

Bad code:

```
int main() {  
    int value;  
    std::vector<int> vectorToSort;  
    vectorToSort.push_back(42); vectorToSort.push_back(13);  
    for (int i = 52; i; i = i - 1)  
    {  
        vectorToSort.push_back(i *2);  
    }  
    /// Optimized for sorting small vectors  
    if (vectorToSort.size() == 1);  
    else  
    {  
        if (vectorToSort.size() <= 2)  
            std::sort(vectorToSort.begin(), std::end(vectorToSort));  
        }  
        for (value : vectorToSort) std::cout << value << ' ';  
    return 0; }
```

Better code:

```
std::vector<int> createSemiRandomData() {  
    std::vector<int> data;  
    data.push_back(42);  
    data.push_back(13);  
    for (int i = 52; i; --i)  
        vectorToSort.push_back(i *2);  
    return data;  
}  
  
/// Optimized for sorting small vectors  
void sortVector(std::vector &v) {  
    if (vectorToSort.size() == 1)  
        return;  
    if (vectorToSort.size() > 2)  
        return;  
  
    std::sort(vectorToSort.begin(), vectorToSort.end());  
}  
  
void printVector(const std::vector<int> &v) {  
    for (auto i : v)  
        std::cout << i << ' ';
```

```

int main() {
    auto vectorToSort = createSemiRandomData();
    sortVector(std::ref(vectorToSort));
    printVector(vectorToSort);

    return 0;
}

```

无论你偏好和使用何种编码风格，保持一致的编码（和格式）风格将有助于你理解代码。

观察上面的代码，可以发现一些改进点，以提升可读性和调试性：

为不同操作使用不同的函数

使用不同的函数可以让你在调试时跳过一些函数，如果你对细节不感兴趣的话。在这个具体的例子中，你可能对数据的创建或打印不感兴趣，只想进入排序部分。

另一个优点是，在逐步调试代码时，你需要阅读（并记忆）的代码更少。现在你只需阅读main()中的3行代码就能理解它，而不是整个函数。

第三个优点是你需要看的代码更少，这有助于训练有素的眼睛在几秒钟内发现这个错误。

使用一致的格式/结构

使用一致的格式和结构可以减少代码中的杂乱，使你更容易专注于代码本身而非文本。关于“正确”的格式风格有很多讨论。不管是哪种风格，代码中保持单一一致的风格都会提高熟悉度，使你更容易专注于代码。

由于格式化代码是一个耗时的任务，建议使用专门的工具来完成。大多数集成开发环境（IDE）至少都支持某种格式化功能，且格式化效果通常比人工更一致。

你可能注意到，这种风格不仅限于空格和换行符，因为我们不再混用自由风格函数和成员函数来获取容器的开始/结束位置。（`v.begin()` 与 `std::end(v)`）。

关注代码中的重要部分。

无论你选择哪种风格，上述代码包含几个标记，可能会提示你哪些部分是重要的：

- 一条注释说明了优化，这表明使用了一些高级技术
- 在`sortVector()`中有一些提前返回，表明我们正在做一些特殊处理
- `std::ref()`表明 `sortVector()` 中发生了一些事情

结论

拥有干净的代码将有助于你理解代码，并减少调试所需的时间。在第二个例子中，代码审查者甚至可能一眼就发现错误，而在第一个例子中，错误可能隐藏在细节中。（附注：错误出现在与2的比较中。）

```

int main() {
    auto vectorToSort = createSemiRandomData();
    sortVector(std::ref(vectorToSort));
    printVector(vectorToSort);

    return 0;
}

```

Regardless of the coding styles you prefer and use, having a consistent coding (and formatting) style will help you understanding the code.

Looking at the code above, one can identify a couple of improvements to improve readability and debuggability:

The use of separate functions for separate actions

The use of separate functions allow you to skip over some functions in the debugger if you ain't interested in the details. In this specific case, you might not be interested in the creation or printing of the data and only want to step into the sorting.

Another advantage is that you need to read less code (and memorize it) while stepping through the code. You now only need to read 3 lines of code in `main()` in order to understand it, instead of the whole function.

The third advantage is that you simply have less code to look at, which helps a trained eye in spotting this bug within seconds.

Using consistent formatting/constructions

The use of consistent formatting and constructions will remove clutter from the code making it easier to focus on the code instead of text. A lot of discussions have been fed on the 'right' formatting style. Regardless of that style, having a single consistent style in the code will improve familiarity and make it easier to focus on the code.

As formatting code is time consuming task, it is recommended to use a dedicated tool for this. Most IDEs have at least some kind of support for this and can do formatting more consistent than humans.

You might note that the style is not limited to spaces and newlines as we no longer mix the free-style and the member functions to get begin/end of the container. (`v.begin()` vs `std::end(v)`).

Point attention to the important parts of your code.

Regardless of the style you determine to choose, the above code contains a couple of markers which might give you a hint on what might be important:

- A comment stating optimized, this indicates some fancy techniques
- Some early returns in `sortVector()` indicate that we are doing something special
- The `std::ref()` indicates that something is going on with the `sortVector()`

Conclusion

Having clean code will help you understanding the code and will reduce the time you need to debug it. In the second example, a code reviewer might even spot the bug at first glance, while the bug might be hidden in the details in the first one. (PS: The bug is in the compare with 2.)

第143章：C++中的优化

第143.1节：性能简介

C和C++因其高性能而闻名——这在很大程度上归功于大量的代码定制，允许用户通过选择结构来指定性能。

在优化时，重要的是对相关代码进行基准测试，并完全理解代码的使用方式。

常见的优化错误包括：

- 过早优化：复杂的代码在优化后可能表现得更差，浪费时间和精力。
首要任务应是编写正确且可维护的代码，而非优化代码。
- 针对错误的使用场景进行优化：为1%的情况增加开销，可能不值得导致其他99%的情况变慢。
- 微观优化：编译器对此非常高效，微观优化甚至可能损害编译器进一步优化代码的能力

典型的优化目标是：

- 减少工作量
- 使用更高效的算法/结构
- 更好地利用硬件

优化后的代码可能带来负面影响，包括：

- 更高的内存使用
- 复杂的代码——难以阅读或维护
- 妥协的API和代码设计

第143.2节：空基类优化

对象的大小不能小于1字节，否则该类型数组的成员将具有相同的地址。因此，`sizeof(T)>=1` 总是成立。同样，派生类的大小也不能小于其任何基类的大小。然而，当基类为空时，其大小不一定会加到派生类上：

```
class Base {};\n\nclass Derived : public Base\n{\n    公共:\n        int i;\n};
```

在这种情况下，不需要为Base在Derived中分配一个字节来为每个类型的每个对象拥有不同的地址。如果执行了空基类优化（且不需要填充），那么`sizeof(Derived) == sizeof(int)`，也就是说，空基类不会额外分配空间。这在多重继承中也适用（在C++中，多重基类不能有相同类型，因此不会出现相关问题）。

请注意，只有当Derived的第一个成员与任何基类的类型不同，才能执行此优化。

这包括任何直接或间接的公共基类。如果它与某个基类的类型相同（或者存在公共基类），则至少需要分配一个字节，以确保同一类型的两个不同对象不会拥有相同的地址。

Chapter 143: Optimization in C++

Section 143.1: Introduction to performance

C and C++ are well known as high-performance languages - largely due to the heavy amount of code customization, allowing a user to specify performance by choice of structure.

When optimizing it is important to benchmark relevant code and completely understand how the code will be used.

Common optimization mistakes include:

- **Premature optimization:** Complex code may perform worse after optimization, wasting time and effort.
First priority should be to write *correct* and *maintainable* code, rather than optimized code.
- **Optimization for the wrong use case:** Adding overhead for the 1% might not be worth the slowdown for the other 99%
- **Micro-optimization:** Compilers do this very efficiently and micro-optimization can even hurt the compilers ability to further optimize the code

Typical optimization goals are:

- To do less work
- To use more efficient algorithms/structures
- To make better use of hardware

Optimized code can have negative side effects, including:

- Higher memory usage
- Complex code -being difficult to read or maintain
- Compromised API and code design

Section 143.2: Empty Base Class Optimization

An object cannot occupy less than 1 byte, as then the members of an array of this type would have the same address. Thus `sizeof(T)>=1` always holds. It's also true that a derived class cannot be smaller than *any* of its base classes. However, when the base class is empty, its size is not necessarily added to the derived class:

```
class Base {};\n\nclass Derived : public Base\n{\n    public:\n        int i;\n};
```

In this case, it's not required to allocate a byte for Base within Derived to have a distinct address per type per object. If empty base class optimization is performed (and no padding is required), then `sizeof(Derived) == sizeof(int)`, that is, no additional allocation is done for the empty base. This is possible with multiple base classes as well (in C++, multiple bases cannot have the same type, so no issues arise from that).

Note that this can only be performed if the first member of Derived differs in type from any of the base classes. This includes any direct or indirect common bases. If it's the same type as one of the bases (or there's a common base), at least allocating a single byte is required to ensure that no two distinct objects of the same type have the same address.

第143.3节：通过执行更少的代码进行优化

最直接的优化方法是执行更少的代码。这种方法通常会带来固定的加速效果，但不会改变代码的时间复杂度。

尽管这种方法能带来明显的加速，但只有在代码被频繁调用时，才能看到显著的性能提升。

移除无用代码

```
void func(const A *a); // 某个随机函数
```

```
// 对实例进行无用的内存分配和释放
auto a1 = std::make_unique<A>();
func(a1.get());
```

```
// 使用栈对象可以防止
auto a2 = A{};
func(&a2);
```

版本 ≥ C++14

从 C++14 开始，编译器允许优化此代码以移除分配及相应的释放。

只执行一次代码

```
std::map<std::string, std::unique_ptr<A>> lookup;
// 插入/查找较慢
// 在此函数中，我们将遍历两次 map 查找元素
// 如果元素不存在，还会第三次遍历
const A *lazyLookupSlow(const std::string &key) {
    if (lookup.find(key) != lookup.cend())
        lookup.emplace_back(key, std::make_unique<A>());
    return lookup[key].get();
}
```

在此函数中，我们将获得与慢速版本相同的明显效果，但速度加倍，因为只遍历一次代码

```
const A *lazyLookupSlow(const std::string &key) {
    auto &value = lookup[key];
    if (!value)
        value = std::make_unique<A>();
    return value.get();
}
```

类似的优化方法可以用来实现一个稳定版本的unique

```
std::vector<std::string> stableUnique(const std::vector<std::string> &v) {
    std::vector<std::string> result;
    std::set<std::string> checkUnique;
    for (const auto &s : v) {
        // 由于 insert 返回插入是否成功，我们可以判断元素是否已经存在
        // 这避免了对每个唯一元素都进行遍历映射的插入操作
        // 因此，如果 v 中没有重复元素，我们几乎可以获得 50% 的性能提升
        if (checkUnique.insert(s).second)
            result.push_back(s);
    }
    return result;
}
```

Section 143.3: Optimizing by executing less code

The most straightforward approach to optimizing is by executing less code. This approach usually gives a fixed speed-up without changing the time complexity of the code.

Even though this approach gives you a clear speedup, this will only give noticeable improvements when the code is called a lot.

Removing useless code

```
void func(const A *a); // Some random function
```

```
// useless memory allocation + deallocation for the instance
auto a1 = std::make_unique<A>();
func(a1.get());
```

```
// making use of a stack object prevents
auto a2 = A{};
func(&a2);
```

Version ≥ C++14

From C++14, compilers are allowed to optimize this code to remove the allocation and matching deallocation.

Doing code only once

```
std::map<std::string, std::unique_ptr<A>> lookup;
// Slow insertion/lookup
// Within this function, we will traverse twice through the map lookup an element
// and even a third time when it wasn't in
const A *lazyLookupSlow(const std::string &key) {
    if (lookup.find(key) != lookup.cend())
        lookup.emplace_back(key, std::make_unique<A>());
    return lookup[key].get();
}
```

```
// Within this function, we will have the same noticeable effect as the slow variant while going at
// double speed as we only traverse once through the code
const A *lazyLookupSlow(const std::string &key) {
    auto &value = lookup[key];
    if (!value)
        value = std::make_unique<A>();
    return value.get();
}
```

A similar approach to this optimization can be used to implement a stable version of unique

```
std::vector<std::string> stableUnique(const std::vector<std::string> &v) {
    std::vector<std::string> result;
    std::set<std::string> checkUnique;
    for (const auto &s : v) {
        // As insert returns if the insertion was successful, we can deduce if the element was
        // already in or not
        // This prevents an insertion, which will traverse through the map for every unique element
        // As a result we can almost gain 50% if v would not contain any duplicates
        if (checkUnique.insert(s).second)
            result.push_back(s);
    }
    return result;
}
```

防止无用的重新分配和复制/移动

在前面的例子中，我们已经避免了在 std::set 中的查找，然而 std::vector 仍然包含一个增长算法，它需要重新分配其存储空间。通过先为合适的大小预留空间，可以避免这种情况。

```
std::vector<std::string> stableUnique(const std::vector<std::string> &v) {
    std::vector<std::string> result;
    // 通过预留 'result'，我们可以确保在向量中不会进行复制或移动
    // 因为它将有足够的容量容纳我们将插入的最大元素数量
    // 如果我们假设大小为零时不会发生分配
    // 并且分配大块内存所需时间与分配小块内存相同
    // 这将永远不会使程序变慢
    // 旁注：编译器甚至可以预测这一点，并从生成的代码中移除增长的检查

    result.reserve(v.size());
    std::set<std::string> checkUnique;
    for (const auto &s : v) {
        // 见上面的示例
        if (checkUnique.insert(s).second)
            result.push_back(s);
    }
    return result;
}
```

第143.4节：使用高效容器

通过在合适的时间使用合适的数据结构进行优化，可以改变代码的时间复杂度。

```
// 这个 stableUnique 的变体具有 N log(N) 的复杂度
// N > v 中元素的数量
// log(N) > 插入 std::set 的复杂度
std::vector<std::string> stableUnique(const std::vector<std::string> &v) {
    std::vector<std::string> result;
    std::set<std::string> checkUnique;
    for (const auto &s : v) {
        // 通过执行更少的代码来优化
        if (checkUnique.insert(s).second)
            result.push_back(s);
    }
    return result;
}
```

通过使用一种用于存储元素的不同实现的容器（哈希容器而非树），我们可以将实现的复杂度转变为N。作为副作用，我们将调用std::string的比较运算符less，因为只有当插入的字符串应该放入同一个桶时才需要调用它。

```
// 这个stableUnique的变体具有N的复杂度
// N > v中的元素数量
// 1 > std::unordered_set的插入复杂度
std::vector<std::string> stableUnique(const std::vector<std::string> &v) {
    std::vector<std::string> result;
    std::unordered_set<std::string> checkUnique;
    for (const auto &s : v) {
        // 通过执行更少的代码来优化
        if (checkUnique.insert(s).second)
            result.push_back(s);
    }
    return result;
}
```

Preventing useless reallocating and copying/moving

In the previous example, we already prevented lookups in the std::set, however the std::vector still contains a growing algorithm, in which it will have to realloc its storage. This can be prevented by first reserving for the right size.

```
std::vector<std::string> stableUnique(const std::vector<std::string> &v) {
    std::vector<std::string> result;
    // By reserving 'result', we can ensure that no copying or moving will be done in the vector
    // as it will have capacity for the maximum number of elements we will be inserting
    // If we make the assumption that no allocation occurs for size zero
    // and allocating a large block of memory takes the same time as a small block of memory
    // this will never slow down the program
    // Side note: Compilers can even predict this and remove the checks the growing from the
    // generated code
    result.reserve(v.size());
    std::set<std::string> checkUnique;
    for (const auto &s : v) {
        // See example above
        if (checkUnique.insert(s).second)
            result.push_back(s);
    }
    return result;
}
```

Section 143.4: Using efficient containers

Optimizing by using the right data structures at the right time can change the time-complexity of the code.

```
// This variant of stableUnique contains a complexity of N log(N)
// N > number of elements in v
// log(N) > insert complexity of std::set
std::vector<std::string> stableUnique(const std::vector<std::string> &v) {
    std::vector<std::string> result;
    std::set<std::string> checkUnique;
    for (const auto &s : v) {
        // See Optimizing by executing less code
        if (checkUnique.insert(s).second)
            result.push_back(s);
    }
    return result;
}
```

By using a container which uses a different implementation for storing its elements (hash container instead of tree)，we can transform our implementation to complexity N. As a side effect, we will call the comparison operator for std::string less, as it only has to be called when the inserted string should end up in the same bucket.

```
// This variant of stableUnique contains a complexity of N
// N > number of elements in v
// 1 > insert complexity of std::unordered_set
std::vector<std::string> stableUnique(const std::vector<std::string> &v) {
    std::vector<std::string> result;
    std::unordered_set<std::string> checkUnique;
    for (const auto &s : v) {
        // See Optimizing by executing less code
        if (checkUnique.insert(s).second)
            result.push_back(s);
    }
    return result;
}
```

第143.5节：小对象优化

小对象优化是一种用于底层数据结构的技术，例如`std::string`（有时称为短字符串/小字符串优化）。它旨在使用栈空间作为缓冲区，而不是在内容足够小以适合预留空间时使用分配的内存。

通过增加额外的内存开销和额外的计算，它试图避免昂贵的堆分配。这种技术的好处取决于使用情况，如果使用不当甚至会影响性能。

示例

实现带有此优化的字符串的一种非常简单的方法如下：

```
#include <cstring>

class string final
{
    constexpr static auto SMALL_BUFFER_SIZE = 16;

    bool _isAllocated{false};           ///<-- 记录是否分配了内存char *_buffer{nullptr};
    //指向我们使用的缓冲区的指针char _smallBuffer[SMALL_BUFFER_SIZE] = {'\0'};  ///<-- 用于小
    对象优化的栈空间

    public:
    ~string()
    {
        if (_isAllocated)
            delete [] _buffer;
    }

    explicit string(const char *cStyleString)
    {
        auto stringSize = std::strlen(cStyleString);
        _isAllocated = (stringSize > SMALL_BUFFER_SIZE);
        if (_isAllocated)
            _buffer = new char[stringSize];
        else
            _buffer = &_smallBuffer[0];
        std::strcpy(_buffer, &cStyleString[0]);
    }

    string(string &&rhs)
        : _isAllocated(rhs._isAllocated)
        , _buffer(rhs._buffer)
        , _smallBuffer(rhs._smallBuffer) //< 如果已分配则不需要
    {
        if (_isAllocated)
        {
            // 防止内存被重复释放
            rhs._buffer = nullptr;
        }
       否则
        {
            // 复制数据
            std::strcpy(_smallBuffer, rhs._smallBuffer);
            _buffer = &_smallBuffer[0];
        }
    }
}
```

Section 143.5: Small Object Optimization

Small object optimization is a technique which is used within low level data structures, for instance the `std::string` (Sometimes referred to as Short/Small String Optimization). It's meant to use stack space as a buffer instead of some allocated memory in case the content is small enough to fit within the reserved space.

By adding extra memory overhead and extra calculations, it tries to prevent an expensive heap allocation. The benefits of this technique are dependent on the usage and can even hurt performance if incorrectly used.

Example

A very naive way of implementing a string with this optimization would be the following:

```
#include <cstring>

class string final
{
    constexpr static auto SMALL_BUFFER_SIZE = 16;

    bool _isAllocated{false};           ///<-- Remember if we allocated memory
    char *_buffer{nullptr};           ///<-- Pointer to the buffer we are using
    char _smallBuffer[SMALL_BUFFER_SIZE] = {'\0'};  ///<-- Stack space used for SMALL OBJECT
    OPTIMIZATION

    public:
    ~string()
    {
        if (_isAllocated)
            delete [] _buffer;
    }

    explicit string(const char *cStyleString)
    {
        auto stringSize = std::strlen(cStyleString);
        _isAllocated = (stringSize > SMALL_BUFFER_SIZE);
        if (_isAllocated)
            _buffer = new char[stringSize];
        else
            _buffer = &_smallBuffer[0];
        std::strcpy(_buffer, &cStyleString[0]);
    }

    string(string &&rhs)
        : _isAllocated(rhs._isAllocated)
        , _buffer(rhs._buffer)
        , _smallBuffer(rhs._smallBuffer) //< Not needed if allocated
    {
        if (_isAllocated)
        {
            // Prevent double deletion of the memory
            rhs._buffer = nullptr;
        }
        else
        {
            // Copy over data
            std::strcpy(_smallBuffer, rhs._smallBuffer);
            _buffer = &_smallBuffer[0];
        }
    }
}
```

```
}

// 其他方法，包括其他构造函数、拷贝构造函数、  
// 赋值操作符已为可读性省略  
};
```

如上面代码所示，为了防止一些new和delete操作，增加了一些额外的复杂性。除此之外，该类占用的内存空间更大，除非在少数几种

通常会尝试通过位操作将bool值_isAllocated编码到指针_buffer中，以减少单个实例的大小（Intel 64位：可减少8字节）。这种优化只有在已知平台的对齐规则时才可能实现。

何时使用？

由于该优化增加了大量复杂性，不建议在每个类中都使用此优化。它通常出现在常用的低级数据结构中。在常见的C++11标准库实现中，可以在std::basic_string<>和std::function<>中看到其使用。

由于该优化仅在存储的数据小于缓冲区时防止内存分配，因此只有当类经常用于小数据时才会带来好处。

该优化的最后一个缺点是移动缓冲区时需要额外的工作，使得移动操作比不使用缓冲区时更昂贵。当缓冲区包含非POD类型时尤其如此。

```
}

// Other methods, including other constructors, copy constructor,  
// assignment operators have been omitted for readability  
};
```

As you can see in the code above, some extra complexity has been added in order to prevent some new and delete operations. On top of this, the class has a larger memory footprint which might not be used except in a couple of cases.

Often it is tried to encode the bool value _isAllocated, within the pointer _buffer with bit manipulation to reduce the size of a single instance (intel 64 bit: Could reduce size by 8 byte). An optimization which is only possible when its known what the alignment rules of the platform is.

When to use?

As this optimization adds a lot of complexity, it is not recommended to use this optimization on every single class. It will often be encountered in commonly used, low-level data structures. In common C++11 standard library implementations one can find usages in std::basic_string<> and std::function<>.

As this optimization only prevents memory allocations when the stored data is smaller than the buffer, it will only give benefits if the class is often used with small data.

A final drawback of this optimization is that extra effort is required when moving the buffer, making the move-operation more expensive than when the buffer would not be used. This is especially true when the buffer contains a non-POD type.

第144章：优化

编译时，编译器通常会修改程序以提高性能。这是被“as-if”规则允许的，该规则允许进行任何不会改变可观察行为的转换。

第144.1节：内联展开/内联

内联展开（也称为内联）是一种编译器优化，它用函数体替换对该函数的调用。这可以节省函数调用的开销，但代价是空间，因为函数体可能会被复制多次。

```
// source:  
  
int process(int value)  
{  
    return 2 * value;  
}  
  
int foo(int a)  
{  
    return process(a);  
}  
  
// program, after inlining:  
  
int foo(int a)  
{  
    return 2 * a; // the body of process() is copied into foo()  
}
```

内联通常用于小函数，因为函数调用的开销相对于函数体的大小来说是显著的。

第144.2节：空基类优化

任何对象或成员子对象的大小要求至少为1，即使该类型是空的class类型（即没有非静态数据成员的class或struct），这是为了保证同一类型的不同对象的地址总是不同的。

然而，基类class子对象没有这样的限制，可以完全从对象布局中优化掉：

```
#include <cassert>  
  
struct Base {};// 空类  
  
struct Derived1 : Base {  
    int i;  
};  
  
int main()  
{  
    // 任何空类类型的大小至少为1  
    assert(sizeof(Base) == 1);  
  
    // 应用空基类优化  
    assert(sizeof(Derived1) == sizeof(int));
```

Chapter 144: Optimization

When compiling, the compiler will often modify the program to increase performance. This is permitted by the [as-if rule](#), which allows any and all transformations that do not change observable behavior.

Section 144.1: Inline Expansion/Inlining

Inline expansion (also known as inlining) is compiler optimisation that replaces a call to a function with the body of that function. This saves the function call overhead, but at the cost of space, since the function may be duplicated several times.

```
// source:  
  
int process(int value)  
{  
    return 2 * value;  
}  
  
int foo(int a)  
{  
    return process(a);  
}  
  
// program, after inlining:  
  
int foo(int a)  
{  
    return 2 * a; // the body of process() is copied into foo()  
}
```

Inlining is most commonly done for small functions, where the function call overhead is significant compared to the size of the function body.

Section 144.2: Empty base optimization

The size of any object or member subobject is required to be at least 1 even if the type is an empty `class` type (that is, a `class` or `struct` that has no non-static data members), in order to be able to guarantee that the addresses of distinct objects of the same type are always distinct.

However, base `class` subobjects are not so constrained, and can be completely optimized out from the object layout:

```
#include <cassert>  
  
struct Base {};// empty class  
  
struct Derived1 : Base {  
    int i;  
};  
  
int main()  
{  
    // the size of any object of empty class type is at least 1  
    assert(sizeof(Base) == 1);  
  
    // empty base optimization applies  
    assert(sizeof(Derived1) == sizeof(int));
```

}

空基类优化通常被分配器感知的标准库类（如std::vector、std::function、std::shared_ptr等）使用，以避免当分配器无状态时为其分配器成员占用额外存储。这是通过存储所需的数据成员之一（例如vector的begin、end或capacity指针）来实现的。

参考：[cppreference](#)

}

Empty base optimization is commonly used by allocator-aware standard library classes (`std::vector`, `std::function`, `std::shared_ptr`, etc) to avoid occupying any additional storage for its allocator member if the allocator is stateless. This is achieved by storing one of the required data members (e.g., `begin`, `end`, or `capacity` pointer for the `vector`).

Reference: [cppreference](#)

第145章：性能分析

第145.1节：使用gcc和gprof进行性能分析

GNU gprof 分析器 [gprof](#) 允许您对代码进行性能分析。要使用它，您需要执行以下步骤：

1. 使用生成分析信息的设置构建应用程序
2. 通过运行构建好的应用程序生成分析信息
3. 使用 gprof 查看生成的分析信息

为了使用生成分析信息的设置构建应用程序，我们添加-pg标志。例如，我们可以使用

```
$ gcc -pg *.cpp -o app
```

或者

```
$ gcc -O2 -pg *.cpp -o app
```

等等。

一旦应用程序，比如app，构建完成，像平常一样执行它：

```
$ ./app
```

这将生成一个名为gmon.out的文件。

要查看分析结果，现在运行

```
$ gprof app gmon.out
```

(注意我们同时提供了应用程序和生成的输出)。

当然，你也可以使用管道或重定向：

```
$ gprof app gmon.out | less
```

等等。

最后一条命令的结果应该是一个表格，表格的行是函数，列则表示调用次数、总耗时、自身耗时（即函数内部执行时间，不包括调用子函数的时间）。

第145.2节：使用gperf2dot生成调用图

对于更复杂的应用程序，平面执行分析结果可能难以理解。这就是为什么许多分析工具也会生成某种形式的带注释调用图信息。

[gperf2dot](#)将许多分析器（Linux perf、callgrind、oprofile等）的文本输出转换为调用图。

你可以通过运行你的分析器来使用它（以gprof为例）：

```
# 使用分析标志进行编译  
g++ *.cpp -pg
```

Chapter 145: Profiling

Section 145.1: Profiling with gcc and gprof

The GNU gprof profiler, [gprof](#), allows you to profile your code. To use it, you need to perform the following steps:

1. Build the application with settings for generating profiling information
2. Generate profiling information by running the built application
3. View the generated profiling information with gprof

In order to build the application with settings for generating profiling information, we add the -pg flag. So, for example, we could use

```
$ gcc -pg *.cpp -o app
```

or

```
$ gcc -O2 -pg *.cpp -o app
```

and so forth.

Once the application, say app, is built, execute it as usual:

```
$ ./app
```

This should produce a file called [gmon.out](#).

To see the profiling results, now run

```
$ gprof app gmon.out
```

(note that we provide both the application as well as the generated output).

Of course, you can also pipe or redirect:

```
$ gprof app gmon.out | less
```

and so forth.

The result of the last command should be a table, whose rows are the functions, and whose columns indicate the number of calls, total time spent, self time spent (that is, time spent in the function excluding calls to children).

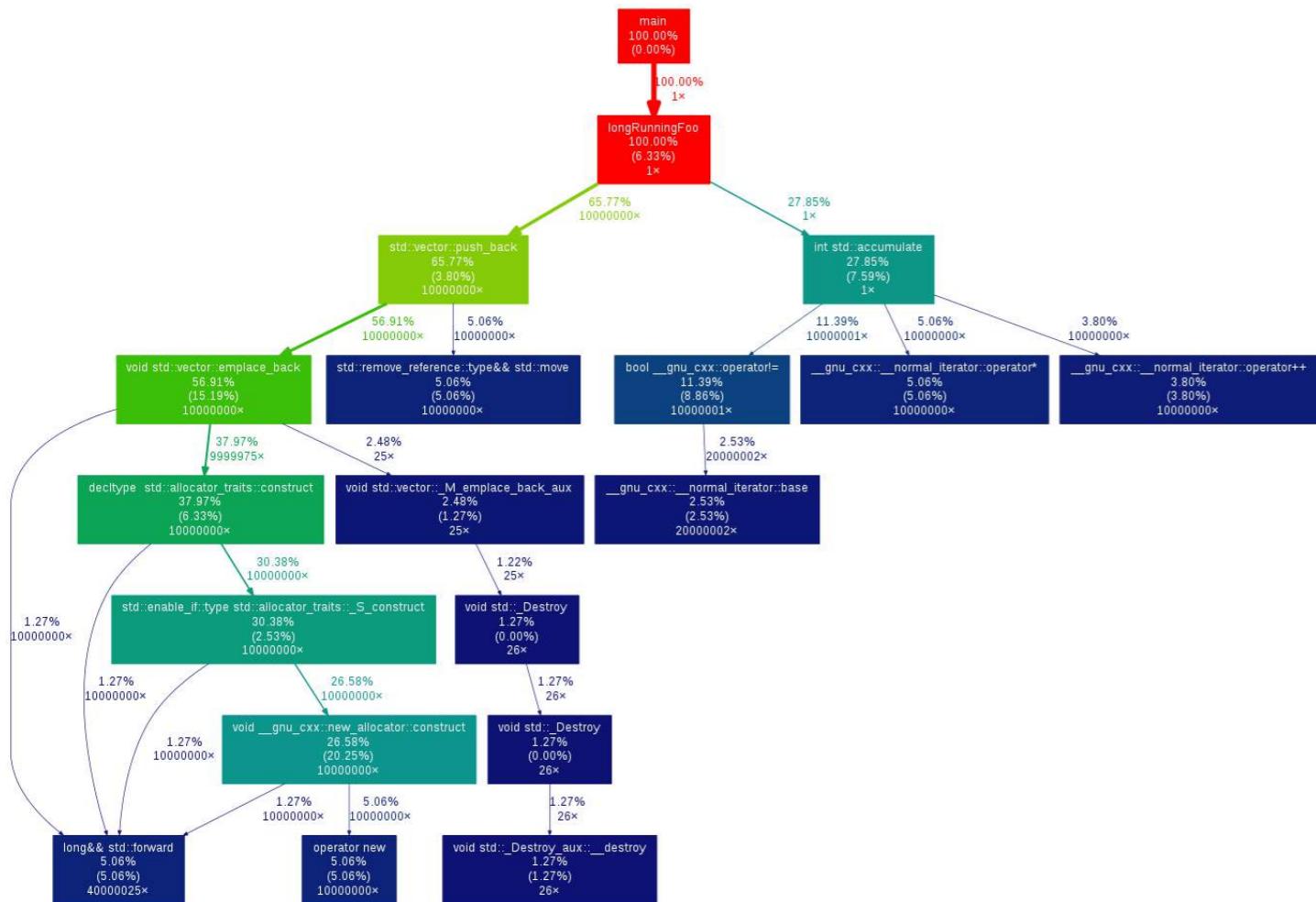
Section 145.2: Generating callgraph diagrams with gperf2dot

For more complex applications, flat execution profiles may be difficult to follow. This is why many profiling tools also generate some form of annotated callgraph information.

[gperf2dot](#) converts text output from many profilers (Linux perf, callgrind, oprofile etc.) into a callgraph diagram. You can use it by running your profiler (example for gprof):

```
# compile with profiling flags  
g++ *.cpp -pg
```

```
# 运行以生成性能分析数据
./main
# 将性能分析数据转换为文本，创建图像
gprof ./main | gprof2dot -s | dot -Tpng -o output.png
```



第145.3节：使用gcc和Google Perf Tools进行CPU使用情况分析

Google Perf Tools还提供了一个CPU分析器，界面稍微友好一些。使用方法如下：

1. 安装Google Perf Tools
2. 像平常一样编译代码
3. 在运行时将libprofiler分析器库添加到库加载路径中
4. 使用pprof生成平面执行分析报告，或调用图示意图

例如：

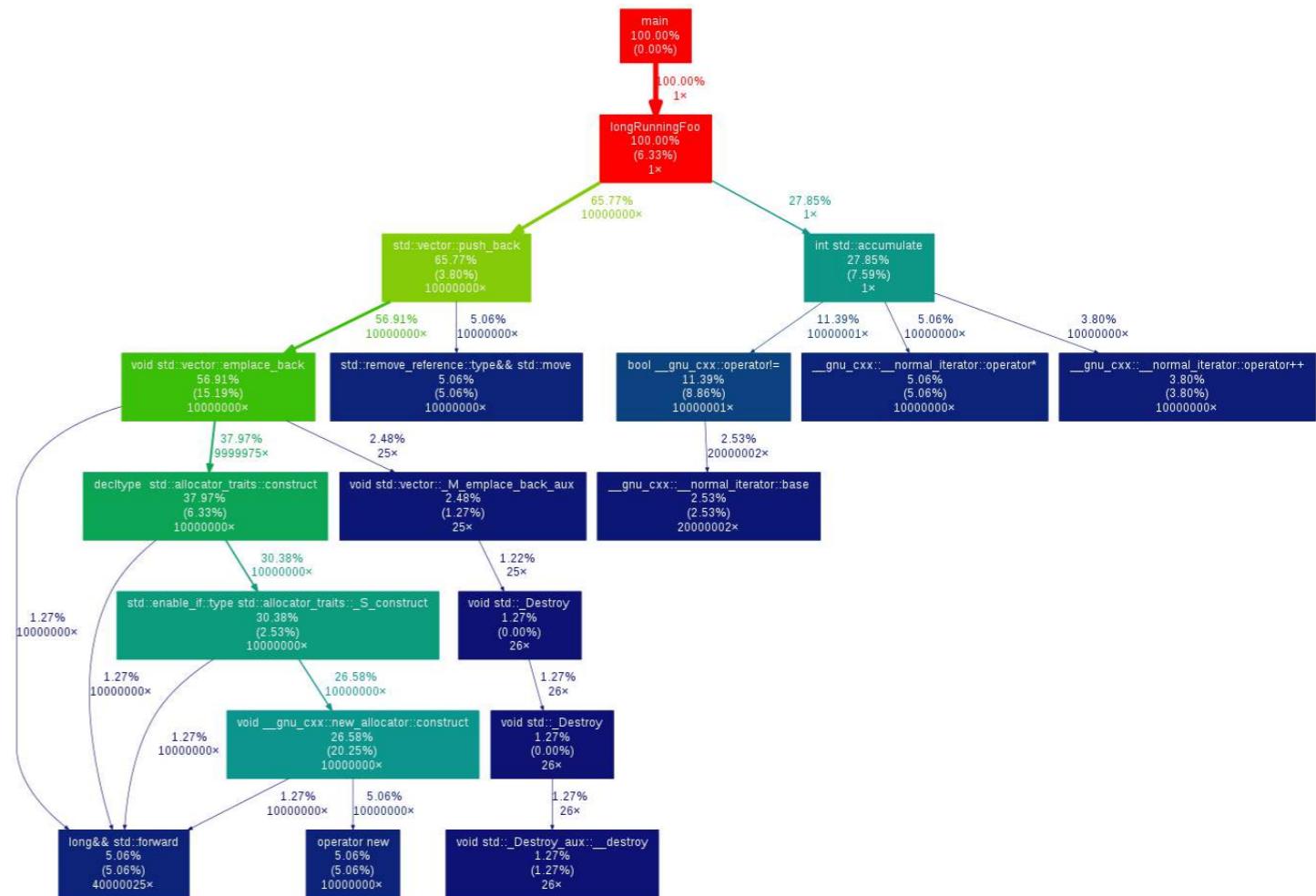
```
# 编译代码
g++ -O3 -std=c++11 main.cpp -o main

# 使用分析器运行
LD_PRELOAD=/usr/local/lib/libprofiler.so CPUPROFILE=main.prof CPUPROFILE_FREQUENCY=100000 ./main
```

说明：

- CPUPROFILE 表示分析数据的输出文件
- CPUPROFILE_FREQUENCY 表示分析器采样频率；

```
# run to generate profiling data
./main
# translate profiling data to text, create image
gprof ./main | gprof2dot -s | dot -Tpng -o output.png
```



Section 145.3: Profiling CPU Usage with gcc and Google Perf Tools

Google Perf Tools also provides a CPU profiler, with a slightly friendlier interface. To use it:

1. [Install Google Perf Tools](#)
2. Compile your code as usual
3. Add the libprofiler profiler library to your library load path at runtime
4. Use pprof to generate a flat execution profile, or a callgraph diagram

For example:

```
# compile code
g++ -O3 -std=c++11 main.cpp -o main

# run with profiler
LD_PRELOAD=/usr/local/lib/libprofiler.so CPUPROFILE=main.prof CPUPROFILE_FREQUENCY=100000 ./main
```

where:

- CPUPROFILE indicates the output file for profiling data
- CPUPROFILE_FREQUENCY indicates the profiler sampling frequency;

使用 pprof 对分析数据进行后处理。

你可以生成文本格式的平面调用分析报告：

```
$ pprof --text ./main main.prof
PROFILE: interrupts/evictions/bytes = 67/15/2016
pprof --text -lines ./main main.prof
使用本地文件 ./main。
使用本地文件 main.prof。
总计：67个样本
22 32.8% 32.8% 67 100.0% longRunningFoo ??:0
20 29.9% 62.7% 20 29.9% __memmove_ssse3_back
/build/eglibc-3GlaMS/eglibc-2.19/string/../sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:1627
4 6.0% 68.7% 4 6.0% __memmove_ssse3_back
/build/eglibc-3GlaMS/eglibc-2.19/string/../sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:1619
3 4.5% 73.1% 3 4.5% __random_r /build/eglibc-3GlaMS/eglibc-2.19/libc/random_r.c:388
3 4.5% 77.6% 3 4.5% __random_r /build/eglibc-3GlaMS/eglibc-2.19/libc/random_r.c:401
2 3.0% 80.6% 2 3.0% __munmap
/build/eglibc-3GlaMS/eglibc-2.19/misc/../sysdeps/unix/syscall-template.S:81
2 3.0% 83.6% 12 17.9% __random /build/eglibc-3GlaMS/eglibc-2.19/libc/random.c:298
2 3.0% 86.6% 2 3.0% __random_r /build/eglibc-3GlaMS/eglibc-2.19/libc/random_r.c:385
2 3.0% 89.6% 2 3.0% rand /build/eglibc-3GlaMS/eglibc-2.19/libc/rand.c:26
1 1.5% 91.0% 1 1.5% __memmove_ssse3_back
/build/eglibc-3GlaMS/eglibc-2.19/string/../sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:1617
1 1.5% 92.5% 1 1.5% __memmove_ssse3_back
/build/eglibc-3GlaMS/eglibc-2.19/string/../sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:1623
1 1.5% 94.0% 1 1.5% __random /build/eglibc-3GlaMS/eglibc-2.19/libc/random.c:293
1 1.5% 95.5% 1 1.5% __random /build/eglibc-3GlaMS/eglibc-2.19/libc/random.c:296
1 1.5% 97.0% 1 1.5% __random_r /build/eglibc-3GlaMS/eglibc-2.19/libc/random_r.c:371
1 1.5% 98.5% 1 1.5% __random_r /build/eglibc-3GlaMS/eglibc-2.19/libc/random_r.c:381
1 1.5% 100.0% 1 1.5% rand /build/eglibc-3GlaMS/eglibc-2.19/libc/rand.c:28
0 0.0% 100.0% 67 100.0% __libc_start_main /build/eglibc-3GlaMS/eglibc-2.19/cs/libc-start.c:287
0 0.0% 100.0% 67 100.0% __start ??:0
0 0.0% 100.0% 67 100.0% main ??:0
0 0.0% 100.0% 14 20.9% rand /build/eglibc-3GlaMS/eglibc-2.19/libc/rand.c:27
0 0.0% 100.0% 27 40.3% std::vector::_M_emplace_back_aux ??:0
```

... 或者你可以用以下命令生成带注释的调用图pdf：

```
pprof --pdf ./main main.prof > out.pdf
```

Use pprof to post-process the profiling data.

You can generate a flat call profile as text:

```
$ pprof --text ./main main.prof
PROFILE: interrupts/evictions/bytes = 67/15/2016
pprof --text -lines ./main main.prof
Using local file ./main.
Using local file main.prof.
Total: 67 samples
22 32.8% 32.8% 67 100.0% longRunningFoo ??:0
20 29.9% 62.7% 20 29.9% __memmove_ssse3_back
/build/eglibc-3GlaMS/eglibc-2.19/string/../sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:1627
4 6.0% 68.7% 4 6.0% __memmove_ssse3_back
/build/eglibc-3GlaMS/eglibc-2.19/string/../sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:1619
3 4.5% 73.1% 3 4.5% __random_r /build/eglibc-3GlaMS/eglibc-2.19/libc/random_r.c:388
3 4.5% 77.6% 3 4.5% __random_r /build/eglibc-3GlaMS/eglibc-2.19/libc/random_r.c:401
2 3.0% 80.6% 2 3.0% __munmap
/build/eglibc-3GlaMS/eglibc-2.19/misc/../sysdeps/unix/syscall-template.S:81
2 3.0% 83.6% 12 17.9% __random /build/eglibc-3GlaMS/eglibc-2.19/libc/random.c:298
2 3.0% 86.6% 2 3.0% __random_r /build/eglibc-3GlaMS/eglibc-2.19/libc/random_r.c:385
2 3.0% 89.6% 2 3.0% rand /build/eglibc-3GlaMS/eglibc-2.19/libc/rand.c:26
1 1.5% 91.0% 1 1.5% __memmove_ssse3_back
/build/eglibc-3GlaMS/eglibc-2.19/string/../sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:1617
1 1.5% 92.5% 1 1.5% __memmove_ssse3_back
/build/eglibc-3GlaMS/eglibc-2.19/string/../sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:1623
1 1.5% 94.0% 1 1.5% __random /build/eglibc-3GlaMS/eglibc-2.19/libc/random.c:293
1 1.5% 95.5% 1 1.5% __random /build/eglibc-3GlaMS/eglibc-2.19/libc/random.c:296
1 1.5% 97.0% 1 1.5% __random_r /build/eglibc-3GlaMS/eglibc-2.19/libc/random_r.c:371
1 1.5% 98.5% 1 1.5% __random_r /build/eglibc-3GlaMS/eglibc-2.19/libc/random_r.c:381
1 1.5% 100.0% 1 1.5% rand /build/eglibc-3GlaMS/eglibc-2.19/libc/rand.c:28
0 0.0% 100.0% 67 100.0% __libc_start_main /build/eglibc-3GlaMS/eglibc-2.19/cs/libc-start.c:287
0 0.0% 100.0% 67 100.0% __start ??:0
0 0.0% 100.0% 67 100.0% main ??:0
0 0.0% 100.0% 14 20.9% rand /build/eglibc-3GlaMS/eglibc-2.19/libc/rand.c:27
0 0.0% 100.0% 27 40.3% std::vector::_M_emplace_back_aux ??:0
```

... or you can generate an annotated callgraph in a pdf with:

```
pprof --pdf ./main main.prof > out.pdf
```

第146章：重构技术

重构是指将现有代码修改为改进版本。虽然重构通常在添加功能或修复错误时进行，但该术语特别指的是在不一定添加功能或修复错误的情况下改进代码。

第146.1节：Goto清理

在曾经是C语言的C++代码库中，可以发现`goto cleanup`的模式。由于`goto`命令使函数的工作流程更难理解，因此通常避免使用。通常，它可以被`return`语句、循环、函数替代。不过，对于`goto cleanup`，需要去除清理逻辑。

```
short calculate(VectorStr **data) {
    short result = FALSE;
    VectorStr *vec = NULL;
    if (!data)
        goto cleanup; //< 可以改为 return false

    // ... 计算过程中使用了 'new' 创建 VectorStr

    结果 = 真;
    清理:
        delete [] vec;
        return result;
}
```

在 C++ 中可以使用 RAII 来解决这个问题：

```
struct VectorRAII final {
    VectorStr *data{nullptr};
    VectorRAII() = default;
    ~VectorRAII() {
        delete [] data;
    }
    VectorRAII(const VectorRAII &) = delete;
};

short calculate(VectorStr **data) {
    VectorRAII vec{};
    if (!data)
        return FALSE; //< 可能改为 return false

    // ... 计算过程中使用 'new' 创建 VectorStr 并存储在 vec.data 中

    return TRUE;
}
```

从此处开始，可以继续重构实际代码。例如，将 `VectorRAII` 替换为 `std::unique_ptr` 或 `std::vector`。

Chapter 146: Refactoring Techniques

Refactoring refers to the modification of existing code into an improved version. Although refactoring is often done while changing code to add features or fix bugs, the term particularly refers improving code without necessarily adding features or fixing bugs.

Section 146.1: Goto Cleanup

In C++ code bases which used to be C, one can find the pattern `goto cleanup`. As the `goto` command makes the workflow of a function harder to understand, this is often avoided. Often, it can be replaced by `return` statements, loops, functions. Though, with the `goto cleanup` one needs to get rid of cleanup logic.

```
short calculate(VectorStr **data) {
    short result = FALSE;
    VectorStr *vec = NULL;
    if (!data)
        goto cleanup; //< Could become return false

    // ... Calculation which 'new's VectorStr

    result = TRUE;
cleanup:
    delete [] vec;
    return result;
}
```

In C++ one could use RAII to fix this issue:

```
struct VectorRAII final {
    VectorStr *data{nullptr};
    VectorRAII() = default;
    ~VectorRAII() {
        delete [] data;
    }
    VectorRAII(const VectorRAII &) = delete;
};

short calculate(VectorStr **data) {
    VectorRAII vec{};
    if (!data)
        return FALSE; //< Could become return false

    // ... Calculation which 'new's VectorStr and stores it in vec.data

    return TRUE;
}
```

From this point on, one could continue refactoring the actual code. For example by replacing the `VectorRAII` by `std::unique_ptr` or `std::vector`.

致谢

非常感谢所有来自Stack Overflow Documentation的人员帮助提供这些内容，
更多更改可以发送至web@petercv.com, 以发布或更新新内容

0xf3759df	第38章
1337忍者	第47章
3442	第47章
4444	第143章
A. 萨里德	第6章和第25章
aaronsnoswell	第24章
阿比纳夫·高尼亚尔	第127章
Abyx	第33章
亚当·特隆	第142章
阿多克沙杰·米什拉	第138章
阿迪提亚	第23章
阿贾伊	第7、33、73、102和119章
阿兰	第73章
亚历杭德罗	第80章
阿列克谢·古谢诺夫	第72章
阿列克谢·沃伊藤科	第33章和第34章
alter igel	第35章
amanuel2	第21章、第29章、第32章、第39章和第128章
amchacon	第80章
阿米·塔沃里	第13章、第49章、第62章、第104章、第130章和第145章
an0o0nym	第3章
anatolyg	第49章和第67章
anderas	第12章、第16章、第33章、第34章、第44章、第49章和第73章
安德里亚·蔡 (Andrea Chua)	第44章、第96章和第131章
安德里亚·科贝利 (Andrea Corbelli)	第26章、第47章、第50章和第73章
AndyG	第49章和第110章
匿名1847	第132章
anotherGatsby	第11章和第15章
安东尼奥·巴雷托	第112章
AProgrammer	第12章
阿拉文德·肯	第39章
坎特伯雷大主教	第1章、第36章和第138章
阿塔卢斯	第108章
asantacreu	第146章
Asu	第26章
阿泰斯·戈拉尔	第36章
巴赫蒂亚尔·哈桑	第35章
巴伦·阿克拉莫维奇	第30章
巴里	第6、9、11、16、18、24、33、36、40、44、47、49、51、63、67、73、74、77、79、82、83、98、103、105、108、110和138章
bcmpinc	第73章
本·H	第1章
本·斯特凡	第138章
本吉·凯斯勒	第77章
大O符号	第78章
比姆	第39章和第54章

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content,
more changes can be sent to web@petercv.com for new content to be published or updated

0xf3759df	Chapter 38
1337ninja	Chapter 47
3442	Chapter 47
4444	Chapter 143
A. Sarid	Chapters 6 and 25
aaronsnoswell	Chapter 24
Abhinav Gauniyal	Chapter 127
Abyx	Chapter 33
Adam Trhon	Chapter 142
Adhokshaj Mishra	Chapter 138
Aditya	Chapter 23
Ajay	Chapters 7, 33, 73, 102 and 119
alain	Chapter 73
Alejandro	Chapter 80
Alexey Guseynov	Chapter 72
Alexey Voytenko	Chapters 33 and 34
alter igel	Chapter 35
amanuel2	Chapters 21, 29, 32, 39 and 128
amchacon	Chapter 80
Ami Tavyory	Chapters 13, 49, 62, 104, 130 and 145
an0o0nym	Chapter 3
anatolyg	Chapters 49 and 67
anderas	Chapters 12, 16, 33, 34, 44, 49 and 73
Andrea Chua	Chapters 44, 96 and 131
Andrea Corbelli	Chapters 26, 47, 50 and 73
AndyG	Chapters 49 and 110
Anonymous1847	Chapter 132
anotherGatsby	Chapters 11 and 15
Antonio Barreto	Chapter 112
AProgrammer	Chapter 12
Aravind .KEN	Chapter 39
ArchbishopOfBanterbury	Chapters 1, 36 and 138
Artalus	Chapter 108
asantacreu	Chapter 146
Asu	Chapter 26
Ates Goral	Chapter 36
Bakhtiar Hasan	Chapter 35
Baron Akramovic	Chapter 30
Barry	Chapters 6, 9, 11, 16, 18, 24, 33, 36, 40, 44, 47, 49, 51, 63, 67, 73, 74, 77, 79, 82, 83, 98, 103, 105, 108, 110 and 138
bcmpinc	Chapter 73
Ben H	Chapter 1
Ben Steffan	Chapter 138
Benjy Kessler	Chapter 77
BigONotation	Chapter 78
Bim	Chapters 39 and 54

[布莱恩](#)
[C.W.霍尔曼二世](#)
[CaffeineToCode](#)
[callyalater](#)
[烛术师](#)
[caps](#)
[cb4](#)
[celtschk](#)
[Chachmu](#)
[干杯，祝你好运。](#)
[chema989](#)
[ChemiCalChems](#)
[CHess](#)
[chrisb2244](#)
[ChrisN](#)
[克里斯托夫](#)
[克里斯托弗·厄兹贝克](#)
[Cid1025](#)
[CinCout](#)
[CodeMouse92](#)
[科迪·格雷 \(Cody Gray\)](#)
[CoffeeandCode](#)
[科林·巴斯内特 \(Colin Basnett\)](#)
[ColleenV](#)
[ComicSansMS](#)
[cpplearner](#)
[crea7or](#)
[CroCo](#)
[cshu](#)
[Curious](#)
[cute_ptr](#)
[天鹅座X1](#)
[守护进程](#)
[达克什·古普塔](#)
[丹·赫尔姆](#)
[丹](#)
[丹尼尔](#)
[丹尼尔·乔尔](#)
[丹尼尔·凯弗](#)
[丹尼尔·斯特拉多夫斯基](#)
[丹尼尔·帕拉斯特雷利](#)
[darkpsychic](#)
[davidsheldon](#)
[DawidPi](#)
[Dean Seo](#)
[DeepCoder](#)
[deepmax](#)
[定义cindy常量](#)
[defube](#)
[demonplus](#)
[Denkkar](#)
[didiz](#)

第1、2、3、15、20、21、22、23、34、36、44、46、64、69、71、72、73、77、79、80、84、91、95、97、98、99、100、104、105、115、119、120、121、133和134章
第63章
第33章和第80章
第34、72和75章
第36章
第47章
第77章
第1章、第16章、第24章、第77章、第90章、第108章和第124章
第12章
第1章、第8章、第75章、第94章和第106章
第144章
第11章、第74章和第106章
第49章
第1章和第34章
第11章
第25章
第24、33、47和73章
第114章
第48章和第49章
第77章
第1章、第6章和第49章
第35章
第16章、第34章、第49章和第77章
第11章
第12、33、49和80章
第33章
第47章
第6章
第104章
第1章和第47章
第9章和第49章
第50章和第75章
第1章和第10章
第1、26、33、38、48、49和115章
第34章
第107章
第62章和第67章
第9章
第69章
第49章
第33、107和108章
第1章和第26章
第50章
第16章
第18章
第1章、第24章、第49章和第77章
第16章、第84章、第113章和第117章
第99章
第36章、第80章和第83章
第54章和第58章
第68章
第3、80、81、85、86、87、88、105、112和140章

[Brian](#)
[C.W.Holeman II](#)
[CaffeineToCode](#)
[callyalater](#)
[Candlemancer](#)
[caps](#)
[cb4](#)
[celtschk](#)
[Chachmu](#)
[Cheers and hth.](#)
[chema989](#)
[ChemiCalChems](#)
[CHess](#)
[chrisb2244](#)
[ChrisN](#)
[Christophe](#)
[Christopher Oezbek](#)
[Cid1025](#)
[CinCout](#)
[CodeMouse92](#)
[Cody Gray](#)
[CoffeeandCode](#)
[Colin Basnett](#)
[ColleenV](#)
[ComicSansMS](#)
[cpplearner](#)
[crea7or](#)
[CroCo](#)
[cshu](#)
[Curious](#)
[cute_ptr](#)
[CygnusX1](#)
[Daemon](#)
[Daksh Gupta](#)
[Dan Hulme](#)
[Danh](#)
[Daniel](#)
[Daniel Jour](#)
[Daniel Käfer](#)
[Daniel Stradowski](#)
[Daniele Pallastrelli](#)
[darkpsychic](#)
[davidsheldon](#)
[DawidPi](#)
[Dean Seo](#)
[DeepCoder](#)
[deepmax](#)
[define cindy const](#)
[defube](#)
[demonplus](#)
[Denkkar](#)
[didiz](#)

Chapters 1, 2, 3, 15, 20, 21, 22, 23, 34, 36, 44, 46, 64, 69, 71, 72, 73, 77, 79, 80, 84, 91, 95, 97, 98, 99, 100, 104, 105, 115, 119, 120, 121, 133 and 134
Chapter 63
Chapters 33 and 80
Chapters 34, 72 and 75
Chapter 36
Chapter 47
Chapter 77
Chapters 1, 16, 24, 77, 90, 108 and 124
Chapter 12
Chapters 1, 8, 75, 94 and 106
Chapter 144
Chapters 11, 74 and 106
Chapter 49
Chapters 1 and 34
Chapter 11
Chapter 25
Chapters 24, 33, 47 and 73
Chapter 114
Chapters 48 and 49
Chapter 77
Chapters 1, 6 and 49
Chapter 35
Chapters 16, 34, 49 and 77
Chapter 11
Chapters 12, 33, 49 and 80
Chapter 33
Chapter 47
Chapter 6
Chapter 104
Chapters 1 and 47
Chapters 9 and 49
Chapters 50 and 75
Chapters 1 and 10
Chapters 1, 26, 33, 38, 48, 49 and 115
Chapter 34
Chapter 107
Chapters 62 and 67
Chapter 9
Chapter 69
Chapter 49
Chapters 33, 107 and 108
Chapters 1 and 26
Chapter 50
Chapter 16
Chapter 18
Chapters 1, 24, 49 and 77
Chapters 16, 84, 113 and 117
Chapter 99
Chapters 36, 80 and 83
Chapters 54 and 58
Chapter 68
Chapters 3, 80, 81, 85, 86, 87, 88, 105, 112 and 140

[diegodfrf](#)

第49、119和135章
[迪特马尔·库尔](#)

第60章
[Dim_ov](#)

第1章
[dkg](#)

第49章
[唐老鸭](#)

第1和138章
[Dr_t](#)

第1章、第12章、第49章和第72章
[Dragma](#)

第34章
[drov](#)

第47章
[杜利·金斯基](#)

第49章和第62章
[Dutow](#)

第71章
[Edd](#)

第73章
[埃德加·罗克jan](#)

第62章
[爱德华](#)

第1章、第11章、第33章、第47章、第66章、第108章和第146章
[ehudt](#)

第49章
[Ela782](#)

第24章
[elimad](#)

第52章
[elvis.dukaj](#)

第141章
[埃尔·罗兰德](#)

第47章
[emlai](#)

第33章
[埃马纽埃尔·马蒂](#)

第69章和第98章
[伊纳穆尔·哈桑](#)

第1章、第24章、第47章、第49章、第50章、第67章和第138章
[enzom83](#)

第36章
[错误](#)

叶夫根尼
[EvgeniyZh](#)

法利亚斯
[了不起的狐狸先生](#)

第1章、第24章、第34章、第47章、第49章、第50章、第68章、第75章和第138章
[fbrereto](#)

第9章和第47章
[FedeWar](#)

第30章和第77章
[Florian](#)

第1章和第130章
[Fox](#)

第49章和第75章
[foxcub](#)

第33、49和50章
[加布里埃尔](#)

第79章
[加尔·德雷曼](#)

第9章
[加利克](#)

第12、24、49、50、81和113章
[高拉夫·库马尔·加尔格](#)

第9章
[高拉夫·塞加尔](#)

第76章
[grr](#)

第104章
[吉里什·库尼亞爾](#)

第104章
[granmirupa](#)

第49章
[格雷格](#)

第77章
[纪尧姆·帕斯卡尔](#)

第62章和第63章
[纪尧姆·拉西科](#)

第106章
[哈。](#)

第65章
[你好](#)

第82章
[亨克斯曼](#)

第28章
[欣德里克·斯特根加](#)

第30章
[霍尔米奇兹](#)

第11章
[霍尔特](#)

第16、47、49和77章
[鸣笛](#)

第1、9、11、12、24、33、47、50、73、77、79和82章
[胡姆姆·赫尔法维](#)

第1章
[赫尔基尔](#)

第9章、第12章和第49章
[hyoslee](#)

[diegodfrf](#)

Chapters 49, 119 and 135
[Dietmar Kühl](#)

Chapter 60
[Dim_ov](#)

Chapter 1
[dkg](#)

Chapter 49
[Donald Duck](#)

Chapters 1, 12, 49 and 72
[Dr_t](#)

Chapter 34
[Dragma](#)

Chapter 47
[drov](#)

Chapters 49 and 62
[Duly Kinsky](#)

Chapter 71
[Dutow](#)

Chapter 73
[Edd](#)

Chapter 62
[Edgar Rokjan](#)

Chapters 1, 11, 33, 47, 66, 108 and 146
[Edward](#)

Chapter 49
[ehudt](#)

Chapter 24
[Ela782](#)

Chapter 52
[elimad](#)

Chapter 141
[elvis.dukaj](#)

Chapter 47
[Emil Rowland](#)

Chapter 33
[emlai](#)

Chapters 69 and 98
[Emmanuel Mathi](#)

Chapters 1, 24, 47, 49, 50, 67 and 138
[Enamul Hassan](#)

Chapter 36
[enzom83](#)

Chapters 48 and 117
[Error](#)

Chapter 52
[Evgeniy](#)

Chapter 9
[EvgeniyZh](#)

Chapter 49
[Falias](#)

Chapters 1, 24, 34, 47, 49, 50, 68, 75 and 138
[Fantastic Mr Fox](#)

Chapters 9 and 47
[fbrereto](#)

Chapters 30 and 77
[FedeWar](#)

Chapters 1 and 130
[Florian](#)

Chapters 49 and 75
[Fox](#)

Chapters 33, 49 and 50
[foxcub](#)

Chapter 79
[Gabriel](#)

Chapter 9
[Gal Dreiman](#)

Chapters 12, 24, 49, 50, 81 and 113
[Galik](#)

Chapter 9
[Gaurav Kumar Garg](#)

Chapter 76
[Gaurav Sehgal](#)

Chapter 104
[ggrr](#)

Chapter 104
[GIRISH kuniyal](#)

Chapter 49
[granmirupa](#)

Chapter 77
[Greg](#)

Chapters 62 and 63
[Guillaume Pascal](#)

Chapter 106
[Guillaume Racicot](#)

Chapter 65
[Ha.](#)

Chapter 82
[hello](#)

Chapter 28
[Henkersmann](#)

Chapter 30
[Hindrik Stegenga](#)

Chapter 11
[holmicz](#)

Chapters 16, 47, 49 and 77
[Holt](#)

Chapters 1, 9, 11, 12, 24, 33, 47, 50, 73, 77, 79 and 82
[honk](#)

Chapter 1
[Humam Helfawi](#)

Chapters 9, 12 and 49
[Hurkyl](#)

Chapter 85
[hyoslee](#)

伊恩·林罗斯	第75章	Ian Ringrose	Chapter 75
伊戈尔·奥克斯	第108章	Igor Oks	Chapter 108
immerhart	第49章和第139章	immerhart	Chapters 49 and 139
计算机模拟	第101章	In silico	Chapter 101
伊万·库什	第63章	Ivan Kush	Chapter 63
杰雷米·罗伊	第44章	Jérémie Roy	Chapter 44
杰克	第47章	Jack	Chapter 47
贾希德	第47、71、72和138章	Jahid	Chapters 47, 71, 72 and 138
詹姆斯·阿德基森	第36和80章	James Adkison	Chapters 36 and 80
贾里德·佩恩	第33和51章	Jared Payne	Chapters 33 and 51
Jarod42	第6、16、17、25、34、44、68、72、78、83、90、103、108、112、113、114、120、121和138章	Jarod42	Chapters 6, 16, 17, 25, 34, 44, 68, 72, 78, 83, 90, 103, 108, 112, 113, 114, 120, 121 and 138
杰森·沃特金斯	第49、80、130和138章	Jason Watkins	Chapters 49, 80, 130 and 138
贾廷	第17和35章	latin	Chapters 17 and 35
让	第73章	Jean	Chapter 73
杰瑞·科芬	第34章	Jerry Coffin	Chapter 34
吉姆·克拉克	第1章	Jim Clark	Chapter 1
约翰·伦德伯格	第1、24、33、49、82、108、113和138章	Johan Lundberg	Chapters 1, 24, 33, 49, 82, 108, 113 and 138
约翰内斯·绍布	第11、24、35、37、44、73、74、101、105和122章	Johannes Schaub	Chapters 11, 24, 35, 37, 44, 73, 74, 101, 105 and 122
约翰·伯杰	第31章	John Burger	Chapter 31
约翰·迪菲尼	第43章	John DiFini	Chapter 43
约翰·伦敦	第23章	John London	Chapter 23
乔纳森·李	第78和103章	Jonathan Lee	Chapters 78 and 103
乔纳森·米	第47章和第70章	Jonathan Mee	Chapters 47 and 70
jotik	第1章、第33章、第34章、第47章、第71章、第72章、第92章和第138章	jotik	Chapters 1, 33, 34, 47, 71, 72, 92 and 138
JPNotADragon	第9章	JPNotADragon	Chapter 9
jpo38	第47章和第49章	jpo38	Chapters 47 and 49
朱利安	第44章	Julien	Chapter 44
贾斯汀	第1章、第16章、第17章、第33章、第70章、第77章和第130章	Justin	Chapters 1, 16, 17, 33, 70, 77 and 130
贾斯汀·泰姆	第1章、第11章、第20章、第23章、第25章、第32章、第34章、第35章、第38章、第41章、第71章、第75章、第82章、第95章、第106章、第128章和第138章	Justin Time	Chapters 1, 11, 20, 23, 25, 32, 34, 35, 38, 41, 71, 75, 82, 95, 106, 128 and 138
JVApen	第1、3、6、12、15、27、33、35、44、49、59、63、73、83、89、91、106、107、115、118、123、126、130、138、142、143和146章	JVApen	Chapters 1, 3, 6, 12, 15, 27, 33, 35, 44, 49, 59, 63, 73, 83, 89, 91, 106, 107, 115, 118, 123, 126, 130, 138, 142, 143 and 146
K48	第1章	K48	Chapter 1
kd1508	第104章	kd1508	Chapter 104
肯·Y	第47和75章	Ken Y	Chapters 47 and 75
Kerrek SB	第21、33和34章	Kerrek SB	Chapters 21, 33 and 34
凯沙夫·夏尔马	第1章	Keshav Sharma	Chapter 1
kiner_shah	第1章和第57章	kiner_shah	Chapters 1 and 57
krOoze	第1章、第40章、第49章和第75章	krOoze	Chapters 1, 40, 49 and 75
库纳尔·泰亚吉	第37章	Kunal Tyagi	Chapter 37
L.V.拉奥	第11章	L.V.Rao	Chapter 11
利安德罗斯	第1章	Leandros	Chapter 1
legends2k	第39章	legends2k	Chapter 39
让我成为	第24章	Let_Me_Be	Chapter 24
lorro	第118章和第143章	lorro	Chapters 118 and 143
Loufylouf	第73章	Loufylouf	Chapter 73
卢克·丹顿	第103章	Luc Danton	Chapter 103
maccard	第67章	maccard	Chapter 67
madduci	第115章和第138章	madduci	Chapters 115 and 138
马尔科姆	第1章和第48章	Malcolm	Chapters 1 and 48
马利克	第1和138章	Malick	Chapters 1 and 138
曼利奥	第1、5、6、11、12、25、47、49、50、63、65、71、104、111和138章	manlio	Chapters 1, 5, 6, 11, 12, 25, 47, 49, 50, 63, 65, 71, 104, 111 and 138
Marc.2377	第47章	Marc.2377	Chapter 47

[marcinj](#) 第66章
[马尔科·A.](#) 第58、63、75、100、118和129章
[马克·加德纳](#) 第1章
[marquesm91](#) 第69章
[马丁·约克](#) 第5章、第12章、第24章、第49章、第73章、第77章、第82章和第83章
[MasterHD](#) 第1章
[MathSquared](#) 第123章
[马特](#) 第1和138章
[马修·布里恩](#) 第8章
[马修·M.](#) 第16章
[Maxito](#) 第77章
[米娜·阿尔方斯](#) 第125章
[merlinND](#) 第65章
[Meysam](#) 第33章、第47章和第50章
[迈克尔·加斯基尔](#) 第138章
[迈克H](#) 第9章
[迈克MB](#) 第67章
[Mikitori](#) 第59章
[Mimouni](#) 第1章
[mindriot](#) 第12章和第143章
[错误进化](#) 第142章
[MKAROL](#) 第67章
[mkluwe](#) 第15章
[MotKohn](#) 第49章
[Motti](#) 第38章、第49章和第104章
[mpromonet](#) 第13章和第47章
[MSalters](#) 第63章和第77章
[MSD](#) 第138章
[mtb](#) 第119章
[mtk](#) 第49章
[穆罕默德·阿拉丁](#) 第1章
[muXXmit2X](#) 第1章
[n.m.](#) 第75章和第138章
[纳奥尔·哈达尔](#) 第66章
[内森·奥斯曼](#) 第1章、第130章和第138章
[纳文·米塔尔](#) 第50章
[尼尔·A.](#) 第1章
[内曼贾·博里奇](#) 第1章、第72章和第138章
[尼尔](#) 第24章、第47章、第49章和第83章
[尼古拉斯](#) 第11章
[尼科尔·博拉斯](#) 第11、12、16、24、27、30、44、49、52、67、68、73、74、75、79、82、88、100和109章
[尼古拉·瓦西列夫](#) 第2、48、53、54、55、57、91和112章
[尼廷库马尔·安贝卡尔](#) 第30章
[nnrales](#) 第115章
[非数字](#) 第116章
[空值](#) 第11、44、47、50、72和82章
[nwp](#) 第80章
[Omnifarious](#) 第20章
[奥兹](#) 第9章
[pandaman1234](#) 第49章
[潘卡杰·库马尔·布拉](#) 第84章
[patmanpato](#) 第12章和第49章

[marcinj](#) Chapter 66
[Marco A.](#) Chapters 58, 63, 75, 100, 118 and 129
[Mark Gardner](#) Chapter 1
[marquesm91](#) Chapter 69
[Martin York](#) Chapters 5, 12, 24, 49, 73, 77, 82 and 83
[MasterHD](#) Chapter 1
[MathSquared](#) Chapter 123
[Matt](#) Chapters 1 and 138
[Matthew Brien](#) Chapter 8
[Matthieu M.](#) Chapter 16
[Maxito](#) Chapter 77
[Meena Alfons](#) Chapter 125
[merlinND](#) Chapter 65
[Meysam](#) Chapters 33, 47 and 50
[Michael Gaskill](#) Chapter 138
[Mike H](#) Chapter 9
[MikeMB](#) Chapter 67
[Mikitori](#) Chapter 59
[Mimouni](#) Chapter 1
[mindriot](#) Chapters 12 and 143
[Misgevolution](#) Chapter 142
[MKAROL](#) Chapter 67
[mkluwe](#) Chapter 15
[MotKohn](#) Chapter 49
[Motti](#) Chapters 38, 49 and 104
[mpromonet](#) Chapters 13 and 47
[MSalters](#) Chapters 63 and 77
[MSD](#) Chapter 138
[mtb](#) Chapter 119
[mtk](#) Chapter 49
[Muhammad Aladdin](#) Chapter 1
[muXXmit2X](#) Chapter 1
[n.m.](#) Chapters 75 and 138
[Naor Hadar](#) Chapter 66
[Nathan Osman](#) Chapters 1, 130 and 138
[Naveen Mittal](#) Chapter 50
[Neil A.](#) Chapter 1
[Nemanja Boric](#) Chapters 1, 72 and 138
[Niall](#) Chapters 24, 47, 49 and 83
[Nicholas](#) Chapter 11
[Nicol Bolas](#) Chapters 11, 12, 16, 24, 27, 30, 44, 49, 52, 67, 68, 73, 74, 75, 79, 82, 88, 100 and 109
[Nikola Vasilev](#) Chapters 2, 48, 53, 54, 55, 57, 91 and 112
[Nitinkumar Ambekar](#) Chapter 30
[nnrales](#) Chapter 115
[NonNumeric](#) Chapter 116
[Null](#) Chapters 11, 44, 47, 50, 72 and 82
[nwp](#) Chapter 80
[Omnifarious](#) Chapter 20
[Oz.](#) Chapter 9
[pandaman1234](#) Chapter 49
[Pankaj Kumar Boora](#) Chapter 84
[patmanpato](#) Chapters 12 and 49

帕特里克·奥巴拉

保罗

保罗·贝金汉姆

帕维尔·斯特拉霍夫

PcAF

Ped7g

佩雷特·巴雷拉

彼得

phandinhlan 第62章

第49章和第145章

第49章

第1章

第33章和第34章

第11章和第49章

第7章

第24、50、71、75、104和138章

第75章

第30章

第48章

第11章

第17章

第49章和第73章

第11章

第15章

第1和138章

第49章

Rakete1111 第11、12、24、33、34、35、36、44、47、49、73、77、80、104和110章

ralismark 第104和144章

RamenChef 第2、15、20、21、22和69章

拉维·钱德拉 第54和67章

Reuben Thomas 第40章

理查德·达利 第33、47、49、50、75和138章

rockoder 第26章

rodrigo 第33章

罗兰 第33、44、84、92、101和114章

RomCoo 第12章

罗嫩·内斯 第72章

rtmh 第16章

鲁希凯什·德什潘德 第1章和第49章

瑞安·海宁 第73章和第79章

R_Kapp 第124章

圣 第49章

萨吉斯P 第67章

萨默图法伊尔 第49章

肖恩 第35章和第75章

谢尔盖 第9章、第13章、第19章、第34章、第38章、第77章和第138章

谢里科夫 第12章、第47章、第49章和第73章

鞋子 第1章和第49章

sigalor 第114章

silvergasp 第34章、第49章、第75章和第93章

秋之歌者 第123章

SirGuy 第1章

斯基珀 第47章和第49章

天怒法师 第34章

Smeeheey 第77章

雪鹰 第73章

苏维克·马吉 第50章

sp2danny 第103章

stackptr 第68章

start2learn 第36章和第133章

斯蒂芬 第24章、第49章和第89章

Patryk Obara

paul

Paul Beckingham

Pavel Strakhov

PcAF

Ped7g

Perette Barella

Peter

phandinhlan

Pietro Saccardi

plasmacel

pmelanson

Podgorskiy

Praetorian

Pyves

Qchmqs

Quirk

R. Martinho Fernandes

Rakete1111

ralismark

RamenChef

Ravi Chandra

Reuben Thomas

Richard Dally

rockoder

rodrigo

Roland

RomCoo

Ronen Ness

rtmh

Rushikesh Deshpande

Ryan Haining

R_Kapp

Saint

SajithP

Samer Tufail

Sean

Sergey

Serikov

Shoe

sigalor

silvergasp

SingerOfTheFall

SirGuy

Skipper

Skywrath

Smeeheey

Snowhawk

SouvikMaji

sp2danny

stackptr

start2learn

Stephen

Chapter 62

Chapters 49 and 145

Chapter 49

Chapter 1

Chapters 33 and 34

Chapters 11 and 49

Chapter 7

Chapters 24, 50, 71, 75, 104 and 138

Chapter 75

Chapter 30

Chapter 48

Chapter 11

Chapter 17

Chapters 49 and 73

Chapter 11

Chapter 15

Chapters 1 and 138

Chapter 49

Chapters 11, 12, 24, 33, 34, 35, 36, 44, 47, 49, 73, 77, 80, 104 and 110

Chapters 104 and 144

Chapters 2, 15, 20, 21, 22 and 69

Chapters 54 and 67

Chapter 40

Chapters 33, 47, 49, 50, 75 and 138

Chapter 26

Chapter 33

Chapters 33, 44, 84, 92, 101 and 114

Chapter 12

Chapter 72

Chapter 16

Chapters 1 and 49

Chapters 73 and 79

Chapter 124

Chapter 49

Chapter 67

Chapter 49

Chapters 35 and 75

Chapters 9, 13, 19, 34, 38, 77 and 138

Chapters 12, 47, 49 and 73

Chapters 1 and 49

Chapter 114

Chapters 34, 49, 75 and 93

Chapter 123

Chapter 1

Chapters 47 and 49

Chapter 34

Chapter 77

Chapter 73

Chapter 50

Chapter 103

Chapter 68

Chapters 36 and 133

Chapters 24, 49 and 89

sth 第16章和第49章
斯特拉迪戈斯 第113章
strangeqargo 第49章
SU3 第103章
Sumurai8 第33章、第65章和第83章
土壤 第38章和第49章
tambre 第6章
单宁 第83章和第104章
Tarod 第52章
TartanLlama 第93章和第109章
Tejendra 第15章
tenpercent 第17、18、24、44和75章
塔林杜·库马拉 第138章
博学者 第75章
theo2003 第49章
死神 第17章、第92章、第111章和第115章
托比 第138章
汤姆 第49章
towi 第49章
特雷弗·希基 第6章、第67章和第104章
TriskalJM 第1章、第40章、第49章和第82章
特里格韦·劳格斯托尔 第138章
tulak.hord 第61章
turoni 第3章
txtechhelp 第5章和第111章
UncleZeiv 第1章
user1336087 第47章、第49章和第50章
user2176127 第47章和第49章
user3384414 第24章
user3684240 第33章
vdaras 第50章
Venki 第16章
VermillionAzure 第1章、第45章、第130章和第141章
Vijayabhaskarreddy CH 第42章
Ville 第1章和第24章
弗拉基米尔·加马良 第49章
弗拉基米尔S 第11章
沃尔克A 第50章
W.F. 第16章和第77章
w1th0utnam3 第103章
沃尔特 第1章
这有帮助吗 第137章
沃尔夫 第8、47、49和77章
WQYeo 第1和8章
Wyzard 第50章
Xirema 第4章
Yakk 第9、11、24、33、36、42、44、49、51、56、57、73、80、90、103、107、108章和121
尤素福·阿扎德 第33章
ysdx 第16章
Yuushi 第80章
ФХосё 웃 Переўра 웃 第8章
阿列克谢·涅乌达钦 第5章和第12章

sth Chapters 16 and 49
Stradigos Chapter 113
strangeqargo Chapter 49
SU3 Chapter 103
Sumurai8 Chapters 33, 65 and 83
T.C. Chapters 38 and 49
tambre Chapter 6
Tannin Chapters 83 and 104
Tarod Chapter 52
TartanLlama Chapters 93 and 109
Tejendra Chapter 15
tenpercent Chapters 17, 18, 24, 44 and 75
Tharindu Kumara Chapter 138
The Philomath Chapter 75
theo2003 Chapter 49
ThyReaper Chapters 17, 92, 111 and 115
Toby Chapter 138
Tom Chapter 49
towi Chapter 49
Trevor Hickey Chapters 6, 67 and 104
TriskalJM Chapters 1, 40, 49 and 82
Trygve Laugstøl Chapter 138
tulak.hord Chapter 61
turoni Chapter 3
txtechhelp Chapters 5 and 111
UncleZeiv Chapter 1
user1336087 Chapters 47, 49 and 50
user2176127 Chapters 47 and 49
user3384414 Chapter 24
user3684240 Chapter 33
vdaras Chapter 50
Venki Chapter 16
VermillionAzure Chapters 1, 45, 130 and 141
Vijayabhaskarreddy CH Chapter 42
Ville Chapters 1 and 24
Vladimir Gamalyan Chapter 49
VladimirS Chapter 11
VolkA Chapter 50
W.F. Chapters 16 and 77
w1th0utnam3 Chapter 103
Walter Chapter 1
wasthishelpful Chapter 137
Wolf Chapters 8, 47, 49 and 77
WQYeo Chapters 1 and 8
Wyzard Chapter 50
Xirema Chapter 4
Yakk Chapters 9, 11, 24, 33, 36, 42, 44, 49, 51, 56, 57, 73, 80, 90, 103, 107, 108 and 121
Yousuf Azad Chapter 33
ysdx Chapter 16
Yuushi Chapter 80
ФХосё 웃 Переўра 웃 Chapter 8
Алексей Неудачин Chapters 5 and 12

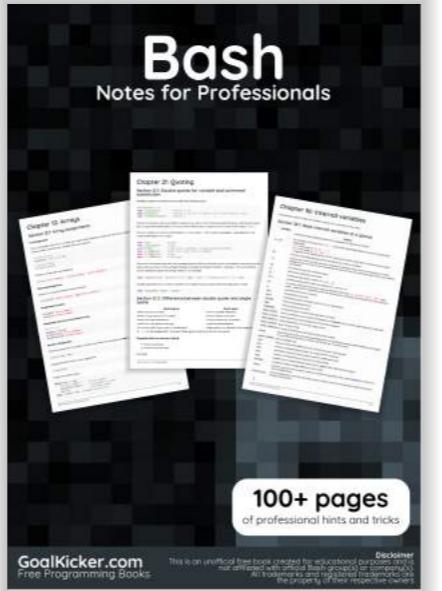
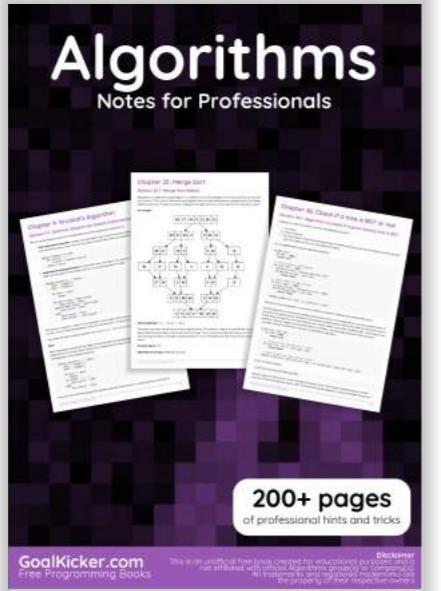
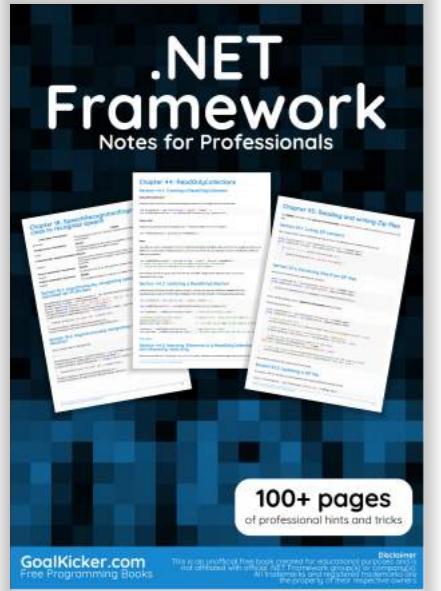
弗拉基米尔·斯特列茨
帕斯卡尔

第10章和第14章
第1章、第75章和第136章

Владимир Стрелец
パスカル

Chapters 10 and 14
Chapters 1, 75 and 136

你可能也喜欢



You may also like

