第一章 开始

熟悉编译器

g++：

- 编译：`g++ --std=c++11 ch01.cpp -o main`
- 运行：`./prog1`

- 查看运行状态：`echo $?`
- 编译多个文件:`g++ ch2.cpp Sales_item.cc -o main`

输入 `g++ --help`，查看编译器选项：

```
Usage: g++ [options] file...
Options:
  -pass-exit-codes         Exit with highest error code from a phase
  --help                   Display this information
  --target-help            Display target specific command line options
  --help={common|optimizers|params|target|warnings|[^]
{joined|separate|undocumented}}[,...]
                           Display specific types of command line options
  (Use '-v --help' to display command line options of sub-processes)
  --version                Display compiler version information
  -dumpspecs               Display all of the built in spec strings
  -dumpversion             Display the version of the compiler
  -dumpmachine             Display the compiler's target processor
  -print-search-dirs       Display the directories in the compiler's search path
  -print-libgcc-file-name  Display the name of the compiler's companion library
  -print-file-name=<lib>   Display the full path to library <lib>
  -print-prog-name=<prog>  Display the full path to compiler component <prog>
  -print-multiarch         Display the target's normalized GNU triplet, used as
                           a component in the library path
  -print-multi-directory   Display the root directory for versions of libgcc
  -print-multi-lib         Display the mapping between command line options and
                           multiple library search directories
  -print-multi-os-directory Display the relative path to OS libraries
  -print-sysroot           Display the target libraries directory
  -print-sysroot-headers-suffix Display the sysroot suffix used to find headers
  -Wa,<options>            Pass comma-separated <options> on to the assembler
  -Wp,<options>            Pass comma-separated <options> on to the preprocessor
  -Wl,<options>            Pass comma-separated <options> on to the linker
  -Xassembler <arg>        Pass <arg> on to the assembler
  -Xpreprocessor <arg>     Pass <arg> on to the preprocessor
  -Xlinker <arg>           Pass <arg> on to the linker
  -save-temps              Do not delete intermediate files
  -save-temps=<arg>        Do not delete intermediate files
  -no-canonical-prefixes   Do not canonicalize paths when building relative
                           prefixes to other gcc components
  -pipe                    Use pipes rather than intermediate files
  -time                    Time the execution of each subprocess
  -specs=<file>            Override built-in specs with the contents of <file>
  -std=<standard>          Assume that the input sources are for <standard>
  --sysroot=<directory>    Use <directory> as the root directory for headers
                           and libraries
  -B <directory>           Add <directory> to the compiler's search paths
  -v                       Display the programs invoked by the compiler
  -###                     Like -v but options quoted and commands not executed
  -E                       Preprocess only; do not compile, assemble or link
  -S                       Compile only; do not assemble or link
  -c                       Compile and assemble, but do not link
  -o <file>                Place the output into <file>
  -pie                     Create a position independent executable
  -shared                  Create a shared library
  -x <language>            Specify the language of the following input files
                           Permissible languages include: c c++ assembler none
```

```
                            'none' means revert to the default behavior of
                            guessing the language based on the file's extension
```

输入 `g++ -v --help` 可以看到更完整的指令。 例如还有些常用的：

```
-h FILENAME, -soname FILENAME: Set internal name of shared library
-I PROGRAM, --dynamic-linker PROGRAM: Set PROGRAM as the dynamic linker to use
-l LIBNAME, --library LIBNAME: Search for library LIBNAME
-L DIRECTORY, --library-path DIRECTORY: Add DIRECTORY to library search path
```

获得程序状态:

- windows: `echo %ERRORLEVEL%`
- UNIX: `echo $?`

IO

- `#include <iostream>`
- `std::cout << "hello"`
- `std::cin >> v1`

记住`>>`和`<<`返回的结果都是左操作数，也就是输入流和输出流本身。

endl：这是一个被称为**操纵符**（manipulator）的特殊值，效果是结束当前行，并将设备关联的缓冲区（buffer）中的内容刷到设备中。

UNIX和Mac下键盘输入文件结束符：`ctrl+d`，Windows下：`ctrl+z`

头文件：类的类型一般存储在头文件中，标准库的头文件使用`<>`，非标准库的头文件使用`""`。申明写在`.h`文件，定义实现写在`.cpp`文件。

避免多次包含同一头文件:

```
#ifndef SALESITEM_H
#define SALESITEM_H
// Definition of Sales_itemclass and related functions goes here
#endif
```

成员函数（类方法）：使用`.`调用。

命名空间（namespace）：使用作用域运算符`::`调用。

注释

- 单行注释：`//`
- 多行注释：`/**/`。编译器将`/*`和`*/`之间的内容都作为注释内容忽略。注意不能嵌套。

while语句

循环执行，直到条件（condition）为假。

for语句

循环头由三部分组成：

- 一个初始化语句（init-statement）

- 一个循环条件（condition）
- 一个表达式（expression）

使用文件重定向

`./main <infile >outfile`

第二章 变量和基本类型

基本内置类型

基本算数类型：

类型	含义	最小尺寸
`bool`	布尔类型	8bits
`char`	字符	8bits
`wchar_t`	宽字符	16bits
`char16_t`	Unicode字符	16bits
`char32_t`	Unicode字符	32bits
`short`	短整型	16bits
`int`	整型	16bits (在32位机器中是32bits)
`long`	长整型	32bits
`long long`	长整型	64bits （是在C++11中新定义的）
`float`	单精度浮点数	6位有效数字
`double`	双精度浮点数	10位有效数字
`long double`	扩展精度浮点数	10位有效数字

如何选择类型

- 1.当明确知晓数值不可能是负数时，选用无符号类型；
- 2.使用`int`执行整数运算。一般`long`的大小和`int`一样，而`short`常常显得太小。除非超过了`int`的范围，选择`long long`。
- 3.算术表达式中不要使用`char`或`bool`。
- 4.浮点运算选用`double`。

类型转换

- 非布尔型赋给布尔型，初始值为0则结果为false，否则为true。
- 布尔型赋给非布尔型，初始值为false结果为0，初始值为true结果为1。

字面值常量

- 一个形如`42`的值被称作**字面值常量**（literal）。
    - 整型和浮点型字面值。
    - 字符和字符串字面值。
        - 使用空格连接，继承自C。
        - 字符字面值：单引号，`'a'`
        - 字符串字面值：双引号，`"Hello World""`
    - 转义序列。`\n`、`\t`等。
    - 布尔字面值。`true`，`false`。
    - 指针字面值。`nullptr`

变量

变量提供一个**具名**的、可供程序操作的存储空间。`C++`中**变量**和**对象**一般可以互换使用。

变量定义（define）

- **定义形式**：类型说明符（type specifier） + 一个或多个变量名组成的列表。如`int sum = 0, value, units_sold = 0;`
- **初始化**（initialize）：对象在创建时获得了一个特定的值。
    - **初始化不是赋值！**：
    - 初始化 = 创建变量 + 赋予初始值
    - 赋值 = 擦除对象的当前值 + 用新值代替
    - **列表初始化**：使用花括号`{}`，如`int units_sold{0};`
    - 默认初始化：定义时没有指定初始值会被默认初始化；在函数体内部的内置类型变量将不会被初始化。
    - 建议初始化每一个内置类型的变量。

变量的声明（declaration） vs 定义（define）

- 为了支持分离式编译，`C++`将声明和定义区分开。**声明**使得名字为程序所知。**定义**负责创建与名字关联的实体。
- **extern**：只是说明变量定义在其他地方。
- 只声明而不定义： 在变量名前添加关键字`extern`，如`extern int i;`。但如果包含了初始值，就变成了定义：`extern double pi = 3.14;`
- 变量只能被定义一次，但是可以多次声明。
- 名字的**作用域**（namescope）

左值和右值

- **左值**（l-value）**可以**出现在赋值语句的左边或者右边，比如变量；
- **右值**（r-value）**只能**出现在赋值语句的右边，比如常量。

复合类型

引用

- **引用**：引用是一个对象的别名，引用类型引用（refer to）另外一种类型。如`int &refVal = val;`。
- 引用必须初始化。
- 引用和它的初始值是**绑定bind**在一起的，而**不是拷贝**。

指针

- 是一种`"指向（point to）"`另外一种类型的复合类型。
- **定义**指针类型：`int *ip1;`，**从右向左读**，`ip1`是指向`int`类型的指针。
- 指针存放某个对象的**地址**。
- 获取对象的地址：`int i=42; int *p = &i;`。`&`是**取地址符**。
- 指针的值的四种状态：
  - 1.指向一个对象；
  - 2.指向紧邻对象的下一个位置；
  - 3.空指针；
  - 4.无效指针。
- 指针访问对象：`cout << *p;`，`*`是**解引用符**。
- 空指针不指向任何对象。
- `void*`指针可以存放**任意**对象的地址。
- 其他指针类型必须要与所指对象**严格匹配**。
- 两个指针相减的类型是`ptrdiff_t`。
- 建议：初始化所有指针。

const限定符

- 动机：希望定义一些不能被改变值的变量。

初始化和const

- const对象**必须初始化**，且**不能被改变**。
- const变量默认不能被其他文件访问，非要访问，必须在指定const前加extern。

const的引用

- **reference to const**（对常量的引用）：指向const对象的引用，如`const int ival=1; const int &refVal = ival;`，可以读取但不能修改`refVal`。
- **临时量**（temporary）对象：当编译器需要一个空间来暂存表达式的求值结果时，临时创建的一个未命名的对象。
- 对临时量的引用是非法行为。

指针和const

- **pointer to const**（指向常量的指针）：不能用于改变其所指对象的值, 如`const double pi = 3.14; const double *cptr = &pi;`。
- **const pointer**：指针本身是常量，如`int i = 0; int *const ptr = &i;`

顶层const

- `顶层const`：指针本身是个常量。
- `底层const`：指针指向的对象是个常量。拷贝时严格要求相同的底层const资格。

`constexpr`和常量表达式

- 常量表达式：指值不会改变，且在编译过程中就能得到计算结果的表达式。
- `C++11`新标准规定，允许将变量声明为`constexpr`类型以便由编译器来验证变量的值是否是一个常量的表达式。

处理类型

类型别名

- 传统别名：使用**typedef**来定义类型的同义词。 `typedef double wages;`
- 新标准别名：别名声明（alias declaration）： `using SI = Sales_item;`（C++11）

auto类型说明符

- **auto**类型说明符：让编译器**自动推断类型。**
- `int i = 0, &r = i; auto a = r;` 推断`a`的类型是`int`。
- 会忽略`顶层const`。
- `const int ci = 1; const auto f = ci;`推断类型是`int`，需要自己加`const`
- `C++11`

decltype类型指示符

- 从表达式的类型推断出要定义的变量的类型。
- **decltype**：选择并返回操作数的**数据类型。**
- `decltype(f()) sum = x;` 推断`sum`的类型是函数`f`的返回类型。
- 不会忽略`顶层const`。
- `C++11`

自定义数据结构

struct

- 类可以以关键字`struct`开始，紧跟类名和类体。
- 类数据成员：类体定义类的成员。
- `C++11`：可以为类数据成员提供一个**类内初始值**（in-class initializer）。

编写自己的头文件

- 头文件通常包含哪些只能被定义一次的实体：类、`const`和`constexpr`变量。

预处理器概述：

- **预处理器**（preprocessor）：确保头文件多次包含仍能安全工作。
- 当预处理器看到`#include`标记时，会用指定的头文件内容代替`#include`
- **头文件保护符**（header guard）：头文件保护符依赖于预处理变量的状态：已定义和未定义。

```
#ifndef SALES_DATA_H
#define SALES_DATA_H
strct Sale_data{
    ...
}
#endif
```

第三章 字符串、向量和数组

using声明

- 使用某个命名空间：例如 `using std::cin` 表示使用命名空间 `std` 中的名字 `cin`。
- 头文件中不应该包含 `using` 声明。这样使用了该头文件的源码也会使用这个声明，会带来风险。

string

- 标准库类型 `string` 表示可变长的字符序列。
- `#include <string>`，然后 `using std::string;`
- **string对象**：注意，不同于字符串字面值。

定义和初始化string对象

初始化 `string` 对象的方式：

方式	解释
`string s1`	默认初始化，`s1` 是个空字符串
`string s2(s1)`	`s2` 是 `s1` 的副本
`string s2 = s1`	等价于 `s2(s1)`，`s2` 是 `s1` 的副本
`string s3("value")`	`s3` 是字面值“value”的副本，除了字面值最后的那个空字符外
`string s3 = "value"`	等价于 `s3("value")`，`s3` 是字面值"value"的副本
`string s4(n, 'c')`	把 `s4` 初始化为由连续 `n` 个字符 `c` 组成的串

- 拷贝初始化（copy initialization）：使用等号 `=` 将一个已有的对象拷贝到正在创建的对象。
- 直接初始化（direct initialization）：通过括号给对象赋值。

string对象上的操作

`string` 的操作：

操作	解释
`os << s`	将`s`写到输出流`os`当中，返回`os`
`is >> s`	从`is`中读取字符串赋给`s`，字符串以空白分割，返回`is`
`getline(is, s)`	从`is`中读取一行赋给`s`，返回`is`
`s.empty()`	`s`为空返回`true`，否则返回`false`
`s.size()`	返回`s`中字符的个数
`s[n]`	返回`s`中第`n`个字符的引用，位置`n`从0计起
`s1+s2`	返回`s1`和`s2`连接后的结果
`s1=s2`	用`s2`的副本代替`s1`中原来的字符
`s1==s2`	如果`s1`和`s2`中所含的字符完全一样，则它们相等；`string`对象的相等性判断对字母的大小写敏感
`s1!=s2`	同上
`<`, `<=`, `>`, `>=`	利用字符在字典中的顺序进行比较，且对字母的大小写敏感

- string io：
  - 执行读操作`>>`：忽略掉开头的空白（包括空格、换行符和制表符），直到遇到下一处空白为止。
  - `getline`：读取一整行，**包括空白符。**
- 字符串字面值和string是不同的类型。

处理string对象中的字符

- **ctype.h vs. cctype**：C++修改了c的标准库，名称为去掉`.h`，前面加`c`。

`cctype`头文件中定义了一组标准函数：

函数	解释
`isalnum(c)`	当`c`是字母或数字时为真
`isalpha(c)`	当`c`是字母时为真
`iscntrl(c)`	当`c`是控制字符时为真
`isdigit(c)`	当`c`是数字时为真
`isgraph(c)`	当`c`不是空格但可以打印时为真
`islower(c)`	当`c`是小写字母时为真
`isprint(c)`	当`c`是可打印字符时为真
`ispunct(c)`	当`c`是标点符号时为真
`isspace(c)`	当`c`是空白时为真（空格、横向制表符、纵向制表符、回车符、换行符、进纸符）
`isupper(c)`	当`c`是大写字母时为真
`isxdigit(c)`	当`c`是十六进制数字时为真
`tolower(c)`	当`c`是大写字母，输出对应的小写字母；否则原样输出`c`
`toupper(c)`	当`c`是小写字母，输出对应的大写字母；否则原样输出`c`

- 遍历字符串：使用**范围for**（range for）语句：`for (auto c: str)`，或者`for (auto &c: str)`使用引用直接改变字符串中的字符。（C++11）

vector

- vector是一个**容器**，也是一个类模板；
- `#include <vector>`然后`using std::vector;`
- 容器：包含其他对象。
- 类模板：本身不是类，但可以**实例化instantiation**出一个类。`vector`是一个模板，`vector<int>`是一个类型。
- 通过将类型放在类模板名称后面的**尖括号**中来指定**类型**，如`vector<int> ivec`。

定义和初始化vector对象

初始化`vector`对象的方法

方法	解释
`vector<T> v1`	`v1`是一个空`vector`，它潜在的元素是`T`类型的，执行默认初始化
`vector<T> v2(v1)`	`v2`中包含有`v1`所有元素的副本
`vector<T> v2 = v1`	等价于`v2(v1)`，`v2`中包含`v1`所有元素的副本
`vector<T> v3(n, val)`	`v3`包含了n个重复的元素，每个元素的值都是`val`
`vector<T> v4(n)`	`v4`包含了n个重复地执行了值初始化的对象
`vector<T> v5{a, b, c...}`	`v5`包含了初始值个数的元素，每个元素被赋予相应的初始值
`vector<T> v5={a, b, c...}`	等价于`v5{a, b, c...}`

- 列表初始化：`vector<string> v{"a", "an", "the"};`（C++11）

向vector对象中添加元素

- `v.push_back(e)`在尾部增加元素。

其他vector操作

`vector`支持的操作：

操作	解释
`v.emtpy()`	如果`v`不含有任何元素，返回真；否则返回假
`v.size()`	返回`v`中元素的个数
`v.push_back(t)`	向`v`的尾端添加一个值为`t`的元素
`v[n]`	返回`v`中第`n`个位置上元素的**引用**
`v1 = v2`	用`v2`中的元素拷贝替换`v1`中的元素
`v1 = {a,b,c...}`	用列表中元素的拷贝替换`v1`中的元素
`v1 == v2`	`v1`和`v2`相等当且仅当它们的元素数量相同且对应位置的元素值都相同
`v1 != v2`	同上
`<`,`<=`,`>`,`>=`	以字典顺序进行比较

- 范围`for`语句内不应该改变其遍历序列的大小。
- `vector`对象（以及`string`对象）的下标运算符，只能对确知已存在的元素执行下标操作，不能用于添加元素。

迭代器iterator

- 所有标准库容器都可以使用迭代器。
- 类似于指针类型，迭代器也提供了对对象的间接访问。

使用迭代器

- `vector<int>::iterator iter`。
- `auto b = v.begin();`返回指向第一个元素的迭代器。
- `auto e = v.end();`返回指向最后一个元素的下一个（哨兵，尾后,one past the end）的迭代器（off the end）。
- 如果容器为空， `begin()`和 `end()`返回的是同一个迭代器，都是尾后迭代器。
- 使用解引用符`*`访问迭代器指向的元素。
- 养成使用迭代器和`!=`的习惯（泛型编程）。
- **容器**：可以包含其他对象；但所有的对象必须类型相同。
- **迭代器（iterator）**：每种标准容器都有自己的迭代器。`C++`倾向于用迭代器而不是下标遍历元素。
- **const_iterator**：只能读取容器内元素不能改变。
- **箭头运算符**： 解引用 + 成员访问，`it->mem`等价于 `(*it).mem`
- **谨记**：但凡是使用了迭代器的循环体，都不要向迭代器所属的容器添加元素。

标准容器迭代器的运算符:

运算符	解释
`*iter`	返回迭代器`iter`所指向的**元素的引用**
`iter->mem`	等价于`(*iter).mem`
`++iter`	令`iter`指示容器中的下一个元素
`--iter`	令`iter`指示容器中的上一个元素
`iter1 == iter2`	判断两个迭代器是否相等

迭代器运算

`vector`和`string`迭代器支持的运算：

运算符	解释
`iter + n`	迭代器加上一个整数值仍得到一个迭代器，迭代器指示的新位置和原来相比向前移动了若干个元素。结果迭代器或者指示容器内的一个元素，或者指示容器尾元素的下一位置。
`iter - n`	迭代器减去一个证书仍得到一个迭代器，迭代器指示的新位置比原来向后移动了若干个元素。结果迭代器或者指向容器内的一个元素，或者指示容器尾元素的下一位置。
`iter1 += n`	迭代器加法的复合赋值语句，将`iter1`加n的结果赋给`iter1`
`iter1 -= n`	迭代器减法的复合赋值语句，将`iter2`减n的加过赋给`iter1`
`iter1 - iter2`	两个迭代器相减的结果是它们之间的距离，也就是说，将运算符右侧的迭代器向前移动差值个元素后得到左侧的迭代器。参与运算的两个迭代器必须指向的是同一个容器中的元素或者尾元素的下一位置。
`>`、`>=`、`<`、`<=`	迭代器的关系运算符，如果某迭代器

- **difference_type**：保证足够大以存储任何两个迭代器对象间的距离，可正可负。

数组

- 相当于vector的低级版，**长度固定。**

定义和初始化内置数组

- 初始化：`char input_buffer[buffer_size];`，长度必须是const表达式，或者不写，让编译器自己推断。
- 数组不允许直接赋值给另一个数组。

访问数组元素

- 数组下标的类型：`size_t`。
- 字符数组的特殊性：结尾处有一个空字符，如`char a[] = "hello";`。
- 用数组初始化`vector`：`int a[] = {1,2,3,4,5}; vector<int> v(begin(a), end(a));`。

数组和指针

- 使用数组时，编译器一般会把它转换成指针。
- 标准库类型限定使用的下标必须是无符号类型，而内置的下标可以处理负值。
- **指针访问数组**：在表达式中使用数组名时，名字会自动转换成指向数组的第一个元素的指针。

C风格字符串

- 从C继承来的字符串。
- 用空字符结束（`\0`）。
- 对大多数应用来说，使用标准库`string`比使用C风格字符串更安全、更高效。
- 获取`string`中的`cstring`：`const char *str = s.c_str();`。

C标准库String函数，定义在`<cstring>`中：

函数	介绍
`strlen(p)`	返回`p`的长度，空字符不计算在内
`strcmp(p1, p2)`	比较`p1`和`p2`的相等性。如果`p1==p2`，返回0；如果`p1>p2`，返回一个正值；如果`p1<p2`，返回一个负值。
`strcat(p1, p2)`	将`p2`附加到`p1`之后，返回`p1`
`strcpy(p1, p2)`	将`p2`拷贝给`p1`，返回`p1`

多维数组

- **多维数组的初始化**：`int ia[3][4] = {{0,1,2,3}, ...}`。
- 使用范围for语句时，除了最内层的循环外，其他所有循环的控制变量都应该是引用类型。

指针vs引用

- 引用总是指向某个对象，定义引用时没有初始化是错的。
- 给引用赋值，修改的是该引用所关联的对象的值，而不是让引用和另一个对象相关联。

指向指针的指针

- 定义：`int **ppi = &pi;`
- 解引用：`**ppi`

动态数组

- 使用`new`和`delete`表达和c中`malloc`和`free`类似的功能，即在堆（自由存储区）中分配存储空间。
- 定义：`int *pia = new int[10];` 10可以被一个变量替代。
- 释放：`delete [] pia;`，注意不要忘记`[]`。

第四章 表达式

表达式基础

- **重载运算符**：当运算符作用在类类型的运算对象时，用户可以自行定义其含义。
- **左值和右值**：
  - C中原意：左值**可以**在表达式左边，右值不能。
  - `C++`：当一个对象被用作**右值**的时候，用的是对象的**值**（内容）；
  - 被用做**左值**时，用的是对象的**身份**（在内存中的位置）。

算术运算符

- **溢出**：当计算的结果超出该类型所能表示的范围时就会产生溢出。

逻辑运算符

- **短路求值**：逻辑与运算符和逻辑或运算符都是先求左侧运算对象的值再求右侧运算对象的值，当且仅当左侧运算对象无法确定表达式的结果时才会计算右侧运算对象的值。

赋值运算符

- 如果赋值运算的左右侧运算对象类型不同，则右侧运算对象将转换成左侧运算对象的类型。
- 赋值运算符满足右结合律，这点和其他二元运算符不一样。`ival = jval = 0;`等价于`ival = (jval = 0);`
- 赋值运算优先级比较低。

条件运算符

- 条件运算符（`?:`）允许我们把简单的`if-else`逻辑嵌入到单个表达式中去，按照如下形式：`cond? expr1: expr2`

位运算符

- 位运算符是作用于**整数类型**的运算对象。
- 二进制位向左移（`<<`）或者向右移（`>>`），移出边界外的位就被舍弃掉了。
- 位取反（`~`）、与（`&`）、或（`|`）、异或（`^`）

sizeof运算符

- 返回一条表达式或一个类型名字所占的**字节数**。返回的类型是`size_t`。
- 两种形式：`sizeof (type)`和`sizeof expr`

类型转换

隐式类型转换

- 比`int`类型小的整数值先提升为较大的整数类型。
- 条件中，非布尔转换成布尔。
- 初始化中，初始值转换成变量的类型。
- 算术运算或者关系运算的运算对象有多种类型，要转换成同一种类型。
- 函数调用时。

显式类型转换（尽量避免）

- **static_cast**：任何明确定义的类型转换，只要不包含底层const，都可以使用。`double slope = static_cast<double>(j);`
- **dynamic_cast**：支持运行时类型识别。
- **const_cast**：只能改变运算对象的底层const，一般可用于去除const性质。`const char *pc; char *p = const_cast<char*>(pc)`
- **reinterpret_cast**：通常为运算对象的位模式提供低层次上的重新解释。第五章 语句

简单语句

- **表达式语句**：一个表达式末尾加上分号，就变成了表达式语句。
- **空语句**：只有一个单独的分号。
- **复合语句（块）**：用花括号`{}`包裹起来的语句和声明的序列。一个块就是一个作用域。

条件语句

- **悬垂else**（dangling else）：用来描述在嵌套的`if else`语句中，如果`if`比`else`多时如何处理的问题。C++使用的方法是`else`匹配最近没有配对的`if`。

迭代语句

- **while**：当不确定到底要迭代多少次时，使用`while`循环比较合适，比如读取输入的内容。
- **for**：`for`语句可以省略掉`init-statement`，`condition`和`expression`的任何一个；**甚至全部。**
- **范围for**：`for (declaration: expression) statement`

跳转语句

- **break**：`break`语句负责终止离它最近的`while`、`do while`、`for`或者`switch`语句，并从这些语句之后的第一条语句开始继续执行。
- **continue**：终止最近的循环中的当前迭代并立即开始下一次迭代。只能在`while`、`do while`、`for`循环的内部。

try语句块和异常处理

- **throw表达式**：异常检测部分使用`throw`表达式来表示它遇到了无法处理的问题。我们说`throw`引发`raise`了异常。

- **try语句块**：以 `try` 关键词开始，以一个或多个 `catch` 字句结束。 `try` 语句块中的代码抛出的异常通常会被某个 `catch` 捕获并处理。 `catch` 子句也被称为**异常处理代码**。
- **异常类**：用于在 `throw` 表达式和相关的 `catch` 子句之间传递异常的具体信息。

第五章 语句

简单语句

```
- **表达式语句**：一个表达式末尾加上分号，就变成了表达式语句。
- **空语句**：只有一个单独的分号。
- **复合语句（块）**：用花括号 `{}`包裹起来的语句和声明的序列。一个块就是一个作用域。
```

条件语句

```
- **悬垂else**（dangling else）：用来描述在嵌套的`if else`语句中，如果`if`比`else`多时如何处理的问题。C++使用的方法是`else`匹配最近没有配对的`if`。
```

迭代语句

```
- **while**：当不确定到底要迭代多少次时，使用 `while`循环比较合适，比如读取输入的内容。
- **for**： `for`语句可以省略掉 `init-statement`， `condition`和 `expression`的任何一个；**甚至全部**。
- **范围for**： `for (declaration: expression) statement`
```

跳转语句

```
- **break**：`break`语句负责终止离它最近的`while`、`do while`、`for`或者`switch`语句，并从这些语句之后的第一条语句开始继续执行。
- **continue**：终止最近的循环中的当前迭代并立即开始下一次迭代。只能在`while`、`do while`、`for`循环的内部。
```

try语句块和异常处理

```
- **throw表达式**：异常检测部分使用 `throw`表达式来表示它遇到了无法处理的问题。我们说`throw`引发 `raise`了异常。
- **try语句块**：以 `try`关键词开始，以一个或多个 `catch`字句结束。 `try`语句块中的代码抛出的异常通常会被某个 `catch`捕获并处理。 `catch`子句也被称为**异常处理代码**。
- **异常类**：用于在 `throw`表达式和相关的 `catch`子句之间传递异常的具体信息。
```

第六章 函数

函数基础

- **函数定义**：包括返回类型、函数名字和0个或者多个**形参**（parameter）组成的列表和函数体。
- **调用运算符**：调用运算符的形式是一对圆括号 `()`，作用于一个表达式，该表达式是函数或者指向函数的指针。
- 圆括号内是用逗号隔开的**实参**（argument）列表。
- 函数调用过程：

    - 1.主调函数（calling function）的执行被中断。
    - 2.被调函数（called function）开始执行。
- **形参和实参**：形参和实参的**个数**和**类型**必须匹配上。
- **返回类型**：`void`表示函数不返回任何值。函数的返回类型不能是数组类型或者函数类型，但可以是指向数组或者函数的指针。
- **名字**：名字的作用于是程序文本的一部分，名字在其中可见。

局部对象

- **生命周期**：对象的生命周期是程序执行过程中该对象存在的一段时间。
- **局部变量**（local variable）：形参和函数体内部定义的变量统称为局部变量。它对函数而言是局部的，对函数外部而言是**隐藏**的。
- **自动对象**：只存在于块执行期间的对象。当块的执行结束后，它的值就变成**未定义**的了。
- **局部静态对象**：`static`类型的局部变量，生命周期贯穿函数调用前后。

函数声明

- **函数声明**：函数的声明和定义唯一的区别是声明无需函数体，用一个分号替代。函数声明主要用于描述函数的接口，也称**函数原型**。
- **在头文件中进行函数声明**：建议变量在头文件中声明；在源文件中定义。
- **分离编译**：`CC a.cc b.cc`直接编译生成可执行文件；`CC -c a.cc b.cc`编译生成对象代码`a.o b.o`；`CC a.o b.o`编译生成可执行文件。

参数传递

- 形参初始化的机理和变量初始化一样。
- **引用传递**（passed by reference）：又称传引用调用（called by reference），指**形参是引用类型**，引用形参是它对应的实参的别名。
- **值传递**（passed by value）：又称传值调用（called by value），指实参的值是通过**拷贝**传递给形参。

传值参数

- 当初始化一个非引用类型的变量时，初始值被拷贝给变量。
- 函数对形参做的所有操作都不会影响实参。
- **指针形参**：常用在C中，`C++`建议使用引用类型的形参代替指针。

传引用参数

- 通过使用引用形参，允许函数改变一个或多个实参的值。
- 引用形参直接关联到绑定的对象，而非对象的副本。
- 使用引用形参可以用于**返回额外的信息**。
- 经常用引用形参来避免不必要的复制。
- `void swap(int &v1, int &v2)`
- 如果无需改变引用形参的值，最好将其声明为常量引用。

const形参和实参

- 形参的顶层`const`被忽略。`void func(const int i);`调用时既可以传入`const int`也可以传入`int`。
- 我们可以使用非常量初始化一个底层`const`对象，但是反过来不行。
- 在函数中，不能改变实参的**局部副本**。
- 尽量使用常量引用。

数组形参

- 当我们为函数传递一个数组时，实际上传递的是指向数组首元素的指针。
- 要注意数组的实际长度，不能越界。

main处理命令行选项

- `int main(int argc, char *argv[]){...}`
- 第一个形参代表参数的个数；第二个形参是参数C风格字符串数组。

可变形参

`initializer_list`提供的操作（`C++11`）：

操作	解释
`initializer_list<T> lst;`	默认初始化；`T`类型元素的空列表
`initializer_list<T> lst{a,b,c...};`	`lst`的元素数量和初始值一样多；`lst`的元素是对应初始值的副本；列表中的元素是`const`。
`lst2(lst)`	拷贝或赋值一个`initializer_list`对象不会拷贝列表中的元素；拷贝后，原始列表和副本共享元素。
`lst2 = lst`	同上
`lst.size()`	列表中的元素数量
`lst.begin()`	返回指向`lst`中首元素的指针
`lst.end()`	返回指向`lst`中微元素下一位置的指针

`initializer_list`使用demo：

```
void err_msg(ErrCode e, initializer_list<string> il){
    cout << e.msg << endl;
    for (auto bed = il.begin(); beg != il.end(); ++ beg)
        cout << *beg << " ";
    cout << endl;
}

err_msg(ErrCode(0), {"functionX", "okay});
```

- 所有实参类型相同，可以使用`initializer_list`的标准库类型。
- 实参类型不同，可以使用`可变参数模板`。
- 省略形参符：`...`，便于`C++`访问某些C代码，这些C代码使用了`varargs`的C标准功能。

返回类型和return语句

无返回值函数

没有返回值的 `return` 语句只能用在返回类型是 `void` 的函数中，返回 `void` 的函数不要求非得有 `return` 语句。

有返回值函数

- `return` 语句的返回值的类型必须和函数的返回类型相同，或者能够**隐式地**转换成函数的返回类型。
- 值的返回：返回的值用于初始化调用点的一个**临时量**，该临时量就是函数调用的结果。
- **不要返回局部对象的引用或指针。**
- **引用返回左值**：函数的返回类型决定函数调用是否是左值。调用一个返回引用的函数得到左值；其他返回类型得到右值。
- **列表初始化返回值**：函数可以返回花括号包围的值的列表。（`C++11`）
- **主函数main的返回值**：如果结尾没有 `return`，编译器将隐式地插入一条返回0的 `return` 语句。返回0代表执行成功。

返回数组指针

- `Type (*function (parameter_list))[dimension]`
- 使用类型别名： `typedef int arrT[10];` 或者 `using arrT = int[10];`，然后 `arrT* func() {...}`
- 使用 `decltype`： `decltype(odd) *arrPtr(int i) {...}`
- **尾置返回类型**： 在形参列表后面以一个 `->` 开始：`auto func(int i) -> int(*)[10]` （`C++11`）

函数重载

- **重载**：如果同一作用域内几个函数名字相同但形参列表不同，我们称之为重载（overload）函数。
- `main` 函数不能重载。
- **重载和const形参**：
  - 一个有顶层const的形参和没有它的函数无法区分。 `Record lookup(Phone* const)` 和 `Record lookup(Phone*)` 无法区分。
  - 相反，是否有某个底层const形参可以区分。 `Record lookup(Account*)` 和 `Record lookup(const Account*)` 可以区分。
- **重载和作用域**：若在内层作用域中声明名字，它将隐藏外层作用域中声明的同名实体，在不同的作用域中无法重载函数名。

特殊用途语言特性

默认实参

- `string screen(sz ht = 24, sz wid = 80, char backgrnd = ' ');`
- 一旦某个形参被赋予了默认值，那么它之后的形参都必须要有默认值。

内联（inline）函数

- 普通函数的缺点：调用函数比求解等价表达式要慢得多。
- `inline` 函数可以避免函数调用的开销，可以让编译器在编译时**内联地展开**该函数。
- `inline` 函数应该在头文件中定义。

constexpr函数

- 指能用于常量表达式的函数。
- `constexpr int new_sz() {return 42;}`
- 函数的返回类型及所有形参类型都要是字面值类型。
- `constexpr`函数应该在头文件中定义。

调试帮助

- `assert`预处理宏（preprocessor macro）：`assert(expr);`

开关调试状态：

`CC -D NDEBUG main.c`可以定义这个变量`NDEBUG`。

```
  void print(){
#ifndef NDEBUG
          cerr << __func__ << "..." << endl;
#endif
  }
```

函数匹配

- 重载函数匹配的**三个步骤**：1.候选函数；2.可行函数；3.寻找最佳匹配。
- **候选函数**：选定本次调用对应的重载函数集，集合中的函数称为候选函数（candidate function）。
- **可行函数**：考察本次调用提供的实参，选出可以被这组实参调用的函数，新选出的函数称为可行函数（viable function）。
- **寻找最佳匹配**：基本思想：实参类型和形参类型越接近，它们匹配地越好。

函数指针

- **函数指针**：是指向函数的指针。
- `bool (*pf)(const string &, const string &);`注：两端的括号不可少。
- **函数指针形参**：
  - 形参中使用函数定义或者函数指针定义效果一样。
  - 使用类型别名或者`decltype`。
- **返回指向函数的指针**：1.类型别名；2.尾置返回类型。

第七章 类 （Class）

定义抽象数据类型

```
- **类背后的基本思想**：**数据抽象**（data abstraction）和**封装**（encapsulation）。
- 数据抽象是一种依赖于**接口**（interface）和**实现**（implementation）分离的编程技术。
```

类成员 （Member）

- 必须在类的内部声明，不能在其他地方增加成员。
- 成员可以是数据，函数，类型别名。

类的成员函数

- 成员函数的**声明**必须在类的内部。
- 成员函数的**定义**既可以在类的内部也可以在外部。
- 使用点运算符 `.` 调用成员函数。
- 必须对任何`const`或引用类型成员以及没有默认构造函数的类类型的任何成员使用初始化式。
- `ConstRef::ConstRef(int ii): i(ii), ci(i), ri(ii) { }`
- 默认实参： `Sales_item(const std::string &book): isbn(book), units_sold(0), revenue(0.0) { }`
- `*this`：
  - 每个成员函数都有一个额外的，隐含的形参`this`。
  - `this`总是指向当前对象，因此`this`是一个常量指针。
  - 形参表后面的`const`，改变了隐含的`this`形参的类型，如 `bool same_isbn(const Sales_item &rhs) const`，这种函数称为“常量成员函数”（`this`指向的当前对象是常量）。
  - `return *this;`可以让成员函数连续调用。
  - 普通的非`const`成员函数：`this`是指向类类型的`const`指针（可以改变`this`所指向的值，不能改变`this`保存的地址）。
  - `const`成员函数：`this`是指向const类类型的`const`指针（既不能改变`this`所指向的值，也不能改变`this`保存的地址）。

非成员函数

- 和类相关的非成员函数，定义和声明都应该在类的外部。

类的构造函数

- 类通过一个或者几个特殊的成员函数来控制其对象的初始化过程，这些函数叫做**构造函数**。
- 构造函数是特殊的成员函数。
- 构造函数放在类的`public`部分。
- 与类同名的成员函数。
- `Sales_item(): units_sold(0), revenue(0.0) { }`
- `=default`要求编译器合成默认的构造函数。(`C++11`)
- 初始化列表：冒号和花括号之间的代码： `Sales_item(): units_sold(0), revenue(0.0) { }`

访问控制与封装

- **访问说明符**（access specifiers）：
  - `public`：定义在 `public`后面的成员在整个程序内可以被访问； `public`成员定义类的接口。
  - `private`：定义在 `private`后面的成员可以被类的成员函数访问，但不能被使用该类的代码访问； `private`隐藏了类的实现细节。
- 使用 `class`或者 `struct`：都可以被用于定义一个类。唯一的却别在于访问权限。
  - 使用 `class`：在第一个访问说明符之前的成员是 `priavte`的。
  - 使用 `struct`：在第一个访问说明符之前的成员是 `public`的。

友元

- 允许特定的**非成员函数**访问一个类的**私有成员**.
- 友元的声明以关键字 `friend`开始。 `friend Sales_data add(const Sales_data&, const Sales_data&);`表示非成员函数`add`可以访问类的非公有成员。
- 通常将友元声明成组地放在**类定义的开始或者结尾**。
- 类之间的友元：
  - 如果一个类指定了友元类，则友元类的成员函数可以访问此类包括非公有成员在内的所有成员。

封装的益处

- 确保用户的代码不会无意间破坏封装对象的状态。
- 被封装的类的具体实现细节可以随时改变，而无需调整用户级别的代码。

类的其他特性

- 成员函数作为内联函数 `inline`：
  - 在类的内部，常有一些规模较小的函数适合于被声明成内联函数。
  - **定义**在类内部的函数是**自动内联**的。
  - 在类外部定义的成员函数，也可以在声明时显式地加上 `inline`。
- **可变数据成员** （mutable data member）：
  - `mutable size_t access_ctr;`
  - 永远不会是`const`，即使它是`const`对象的成员。
- **类类型**：
  - 每个类定义了唯一的类型。

类的作用域

- 每个类都会定义它自己的作用域。在类的作用域之外，普通的数据和函数成员只能由引用、对象、指针使用成员访问运算符来访问。
- 函数的**返回类型**通常在函数名前面，因此当成员函数定义在类的外部时，返回类型中使用的名字都位于类的作用域之外。
- 如果成员使用了外层作用域中的某个名字，而该名字代表一种**类型**，则类不能在之后重新定义该名字。
- 类中的**类型名定义**都要放在一开始。

构造函数再探

- 构造函数初始值列表：
  - 类似`python`使用赋值的方式有时候不行，比如`const`或者引用类型的数据，只能初始化，不能赋值。（注意初始化和赋值的区别）
  - 最好让构造函数初始值的顺序和成员声明的顺序保持一致。
  - 如果一个构造函数为所有参数都提供了默认参数，那么它实际上也定义了默认的构造函数。

委托构造函数 （delegating constructor, C++11）

- 委托构造函数将自己的职责委托给了其他构造函数。
- `Sale_data(): Sale_data("", 0, 0) {}`

隐式的类型转换

- 如果构造函数**只接受一个实参**，则它实际上定义了转换为此类类型的**隐式转换机制**。这种构造函数又叫**转换构造函数**（`converting constructor`）。
- 编译器只会自动地执行`仅一步`类型转换。
- 抑制构造函数定义的隐式转换：
  - 将构造函数声明为`explicit`加以阻止。
  - `explicit`构造函数只能用于直接初始化，不能用于拷贝形式的初始化。

聚合类 （aggregate class)

- 满足以下所有条件：
  - 所有成员都是`public`的。
  - 没有定义任何构造函数。
  - 没有类内初始值。
  - 没有基类，也没有`virtual`函数。
- 可以使用一个花括号括起来的成员初始值列表，初始值的顺序必须和声明的顺序一致。

字面值常量类

- `constexpr`函数的参数和返回值必须是字面值。
- **字面值类型**：除了算术类型、引用和指针外，某些类也是字面值类型。
- 数据成员都是字面值类型的聚合类是字面值常量类。
- 如果不是聚合类，则必须满足下面所有条件：
  - 数据成员都必须是字面值类型。
  - 类必须至少含有一个`constexpr`构造函数。
  - 如果一个数据成员含有类内部初始值，则内置类型成员的初始值必须是一条常量表达式；或者如果成员属于某种类类型，则初始值必须使用成员自己的`constexpr`构造函数。
  - 类必须使用析构函数的默认定义，该成员负责销毁类的对象。

类的静态成员

- 非`static`数据成员存在于类类型的每个对象中。

- `static`数据成员独立于该类的任意对象而存在。

- 每个`static`数据成员是与类关联的对象，并不与该类的对象相关联。

- 声明：

  - 声明之前加上关键词`static`。

- 使用：

  - 使用**作用域运算符**`::`直接访问静态成员:`r = Account::rate();`
  - 也可以使用对象访问：`r = ac.rate();`

- 定义：

  - 在类外部定义时不用加`static`。

- 初始化：

- 通常不在类的内部初始化，而是在定义时进行初始化，如 `double Account::interestRate = initRate();`

- 如果一定要在类内部定义，则要求必须是字面值常量类型的`constexpr`。

第八章 IO库

前面章节已经在用的IO库设施

- **istream**：输入流类型，提供输入操作。
- **ostream**：输出流类型，提供输出操作
- **cin**：一个`istream`对象，从标准输入读取数据。
- **cout**：一个`ostream`对象，向标准输出写入数据。
- **cerr**：一个`ostream`对象，向标准错误写入消息。
- **>>运算符**：用来从一个`istream`对象中读取输入数据。
- **<<运算符**：用来向一个`ostream`对象中写入输出数据。
- **getline函数**：从一个给定的`istream`对象中读取一行数据，存入到一个给定的`string`对象中。

IO类

标准库定义的IO类型

- `iostream`头文件：从标准流中读写数据，`istream`、`ostream`等。
- `fstream`头文件：从文件中读写数据，`ifstream`、`ofstream`等。
- `sstream`头文件：从字符串中读写数据，`istringstream`、`ostringstream`

IO对象不可复制或赋值

- 1.IO对象不能存在容器里.
- 2.形参和返回类型也不能是流类型。
- 3.形参和返回类型一般是流的**引用**。
- 4.读写一个IO对象会改变其状态，因此传递和返回的引用不能是`const`的。

条件状态

状态	解释
`strm:iostate`	是一种机器无关的**类型**，提供了表达条件状态的完整功能
`strm:badbit`	用来指出流已经崩溃
`strm:failbit`	用来指出一个IO操作失败了
`strm:eofbit`	用来指出流到达了文件结束
`strm:goodbit`	用来指出流未处于错误状态，此值保证为零
`s.eof()`	若流`s`的`eofbit`置位，则返回`true`

```
    | `s.fail()`          | 若流`s`的`failbit`置位，则返回`true`
    |
    | `s.bad()`           | 若流`s`的`badbit`置位，则返回`true`
     |
    | `s.good()`          | 若流`s`处于有效状态，则返回`true`
   |
    | `s.clear()`         | 将流`s`中所有条件状态位复位，将流的状态设置成有效，返回`void`
|
    | `s.clear(flags)`    | 将流`s`中指定的条件状态位复位，返回`void`
   |
    | `s.setstate(flags)` | 根据给定的标志位，将流`s`中对应的条件状态位置位，返回`void`
|
    | `s.rdstate()`       | 返回流`s`的当前条件状态，返回值类型为`strm::iostate`
  |

    上表中，`strm`是一种IO类型，（如`istream`）， `s`是一个流对象。
```

管理输出缓冲

```
    - 每个输出流都管理一个缓冲区，执行输出的代码，文本串可能立即打印出来，也可能被操作系统保存在缓冲区内，随后再打印。
    - 刷新缓冲区，可以使用如下IO操纵符：
      - `endl`：输出一个换行符并刷新缓冲区。
      - `flush`：刷新流，单不添加任何字符。
      - `ends`：在缓冲区插入空字符`null`，然后刷新。
      - `unitbuf`：告诉流接下来每次操作之后都要进行一次`flush`操作。
      - `nounitbuf`：回到正常的缓冲方式。
```

文件输入输出

```
    - 头文件`fstream`定义了三个类型来支持文件IO：
      - `ifstream`从一个给定文件读取数据。
      - `ofstream`向一个给定文件写入数据。
      - `fstream`可以读写给定文件。
    - **文件流**：需要读写文件时，必须定义自己的文件流对象，并绑定在需要的文件上。
```

fstream特有的操作

```
    | 操作                          | 解释
            |
    | ------------------------- | ------------------------------------------------
------------- |
    | `fstream fstrm;`            | 创建一个未绑定的文件流。
          |
    | `fstream fstrm(s);`         | 创建一个文件流，并打开名为`s`的文件，`s`可以是
`string`也可以是`char`指针 |
    | `fstream fstrm(s, mode);` | 与前一个构造函数类似，但按指定`mode`打开文件
        |
    | `fstrm.open(s)`             | 打开名为`s`的文件，并和`fstrm`绑定
          |
    | `fstrm.close()`             | 关闭和`fstrm`绑定的文件
           |
    | `fstrm.is_open()`           | 返回一个`bool`值，指出与`fstrm`关联的文件是否成功打开
且尚未关闭 |

    上表中，`fstream`是头文件`fstream`中定义的一个类型，`fstrm`是一个文件流对象。
```

文件模式

```
    | 文件模式 | 解释                           |
    | -------- | ------------------------------ |
    | `in`     | 以读的方式打开                 |
    | `out`    | 以写的方式打开                 |
    | `app`    | 每次写操作前均定位到文件末尾 |
    | `ate`    | 打开文件后立即定位到文件末尾 |
    | `trunc`  | 截断文件                        |
    | `binary` | 以二进制方式进行IO操作。      |
```

string流

```
    - 头文件`sstream`定义了三个类型来支持内存IO：
      - `istringstream`从`string`读取数据。
      - `ostringstream`向`string`写入数据。
      - `stringstream`可以读写给定`string`。
```

stringstream特有的操作

```
    | 操作               | 解释                                |
    | ------------------ | ----------------------------------- |
    | `sstream strm`     | 定义一个未绑定的`stringstream`对象 |
    | `sstream strm(s)` | 用`s`初始化对象                      |
    | `strm.str()`       | 返回`strm`所保存的`string`的拷贝    |
    | `strm.str(s)`      | 将`s`拷贝到`strm`中，返回`void`     |

    上表中`sstream`是头文件`sstream`中任意一个类型。`s`是一个`string`。
```

第九章 顺序容器

顺序容器概述

- **顺序容器**（sequential container）：为程序员提供了控制元素存储和访问顺序的能力。这种顺序不依赖于元素的值，而是与元素加入容器时的位置相对应。

顺序容器类型

容器类型	介绍
`vector`	可变大小数组。支持快速随机访问。在尾部之外的位置插入或删除元素可能很慢。
`deque`	双端队列。支持快速随机访问。在头尾位置插入/删除速度很快。
`list`	双向链表。只支持双向顺序访问。在`list`中任何位置进行插入/删除操作速度都很快。
`forward_list`	单向链表。只支持单向顺序访问。在链表任何位置进行插入/删除操作速度都很快。
`array`	固定大小数组。支持快速随机访问。不能添加或者删除元素。
`string`	与`vector`相似的容器，但专门用于保存字符。随机访问块。在尾部插入/删除速度快。

- 除了固定大小的`array`外，其他容器都提供高效、灵活的内存管理。
- `forward_list`和`array`是新C++标准增加的类型。
- 通常使用`vector`是最好的选择，除非你有很好的理由选择其他容器。
- 新标准库的容器比旧版的快得多。

容器操作

类型

操作	解释
`iterator`	此容器类型的迭代器类型
`const_iterator`	可以读取元素但不能修改元素的迭代器类型
`size_type`	无符号整数类型，足够保存此种容器类型最大可能的大小
`difference_type`	带符号整数类型，足够保存两个迭代器之间的距离
`value_type`	元素类型
`reference`	元素的左值类型；和`value_type &`含义相同
`const_reference`	元素的`const`左值类型，即`const value_type &`

构造函数

操作	解释
`C c;`	默认构造函数，构造空容器
`C c1(c2);`或`C c1 = c2;`	构造`c2`的拷贝`c1`
`C c(b, e)`	构造`c`，将迭代器`b`和`e`指定范围内的所有元素拷贝到`c`
`C c(a, b, c...)`	列表初始化`c`

```
    | `C c(n)`                    | 只支持顺序容器，且不包括`array`，包含`n`个元素，这些元
素进行了值初始化 |
    | `C c(n, t)`                 | 包含`n`个初始值为`t`的元素
          |

    - 只有顺序容器的构造函数才接受大小参数，关联容器并不支持。
    - `array`具有固定大小。
    - 和其他容器不同，默认构造的`array`是非空的。
    - 直接复制：将一个容器复制给另一个容器时，类型必须匹配：容器类型和元素类型都必须相同。
    - 使用迭代器复制：不要求容器类型相同，容器内的元素类型也可以不同。
```

赋值和`swap`

```
    | 操作                  | 解释
      |
    | ------------------- | ----------------------------------------------------
------- |
    | `c1 = c2;`           | 将`c1`中的元素替换成`c2`中的元素
    |
    | `c1 = {a, b, c...}` | 将`c1`中的元素替换成列表中的元素（不适用于`array`）
|
    | `c1.swap(c2)`        | 交换`c1`和`c2`的元素
      |
    | `swap(c1, c2)`       | 等价于`c1.swap(c2)`
       |
    | `c.assign(b, e)`     | 将`c`中的元素替换成迭代器`b`和`e`表示范围中的元素，`b`和`e`
不能指向`c`中的元素 |
    | `c.assign(il)`       | 将`c`中的元素替换成初始化列表`il`中的元素
   |
    | `c.assign(n, r)`     | 将`c`中的元素替换为`n`个值是`t`的元素
   |

    - 使用非成员版本的`swap`是一个好习惯。
    - `assign`操作不适用于关联容器和`array`
```

大小

```
    | 操作            | 解释                                            |
    | -------------- | ---------------------------------------------- |
    | `c.size()`     | `c`中元素的数目（不支持`forward_list`）         |
    | `c.max_size()` | `c`中可保存的最大元素数目                       |
    | `c.empty()`    | 若`c`中存储了元素，返回`false`，否则返回`true` |
```

添加元素

```
    | 操作                     | 解释
         |
    | ----------------------- | ----------------------------------------------
----------- |
    | `c.push_back(t)`         | 在`c`尾部创建一个值为`t`的元素，返回`void`
      |
    | `c.emplace_back(args)`  | 同上
          |
```

```
    | `c.push_front(t)`        | 在`c`头部创建一个值为`t`的元素，返回`void`
      |
    | `c.emplace_front(args)` | 同上
          |
    | `c.insert(p, t)`         | 在迭代器`p`指向的元素之前创建一个值是`t`的元素，返回指向
新元素的迭代器 |
    | `c.emplace(p, args)`     | 同上
          |
    | `c.inset(p, n, t)`       | 在迭代器`p`指向的元素之前插入`n`个值为`t`的元素，返回指向
第一个新元素的迭代器；如果`n`是0，则返回`p` |
    | `c.insert(p, b, e)`      | 将迭代器`b`和`e`范围内的元素，插入到`p`指向的元素之前；如
果范围为空，则返回`p` |
    | `c.insert(p, il)`        | `il`是一个花括号包围中的元素值列表，将其插入到`p`指向的元
素之前；如果`il`是空，则返回`p` |

    - 因为这些操作会改变大小，因此不适用于`array`。
    - `forward_list`有自己专有版本的`insert`和`emplace`。
    - `forward_list`不支持`push_back`和`emplace_back`。
    - 当我们用一个对象去初始化容器或者将对象插入到容器时，实际上放入的是对象的拷贝。
    - `emplace`开头的函数是新标准引入的，这些操作是构造而不是拷贝元素。
    - 传递给`emplace`的参数必须和元素类型的构造函数相匹配。
```

访问元素

```
    | 操作          | 解释                                                          |
    | ------------ | ------------------------------------------------------------
|
    | `c.back()`   | 返回`c`中尾元素的引用。若`c`为空，函数行为未定义               |
    | `c.front()` | 返回`c`中头元素的引用。若`c`为空，函数行为未定义               |
    | `c[n]`       | 返回`c`中下标是`n`的元素的引用，`n`时候一个无符号证书。若
`n>=c.size()`，则函数行为未定义 |
    | `c.at(n)`    | 返回下标为`n`的元素引用。如果下标越界，则抛出`out_of_range`异常 |

    - 访问成员函数返回的是引用。
    - `at`和下标操作只适用于`string`、`vector`、`deque`、`array`。
    - `back`不适用于`forward_list`。
    - 如果希望下标是合法的，可以使用`at`函数。
```

删除元素

```
    | 操作              | 解释
  |
    | --------------- | ------------------------------------------------------------
--- |
    | `c.pop_back()`  | 删除`c`中尾元素，若`c`为空，则函数行为未定义。函数返回`void` |
    | `c.pop_front()` | 删除`c`中首元素，若`c`为空，则函数行为未定义。函数返回`void` |
    | `c.erase(p)`    | 删除迭代器`p`指向的元素，返回一个指向被删除元素之后的元素的迭代器，
若`p`本身是尾后迭代器，则函数行为未定义 |
    | `c.erase(b, e)` | 删除迭代器`b`和`e`范围内的元素，返回指向最后一个被删元素之后元素的
迭代器，若`e`本身就是尾后迭代器，则返回尾后迭代器 |
    | `c.clear()`     | 删除`c`中所有元素，返回`void`
|

    - 会改变容器大小，不适用于`array`。
    - `forward_list`有特殊版本的`erase`
    - `forward_list`不支持`pop_back`
    - `vector`和`string`不支持`pop_front`
```

特殊的forwad_list操作

```
    - 链表在删除元素时需要修改前置节点的内容，双向链表会前驱的指针，但是单向链表没有保存，因此需
要增加获取前置节点的方法。
    - `forward_list`定义了`before_begin`，即首前（off-the-begining）迭代器，允许我们再
在首元素之前添加或删除元素。

    | 操作                          | 解释
              |
    | ----------------------------- | ------------------------------------------
--------------- |
    | `lst.before_begin()`          | 返回指向链表首元素之前不存在的元素的迭代器，此迭代器不
能解引用。 |
    | `lst.cbefore_begin()`         | 同上，但是返回的是常量迭代器。
          |
    | `lst.insert_after(p, t)`      | 在迭代器`p`之后插入元素。`t`是一个对象
          |
    | `lst.insert_after(p, n, t)` | 在迭代器`p`之后插入元素。`t`是一个对象，`n`是数量。
若`n`是0则函数行为未定义 |
    | `lst.insert_after(p, b, e)` | 在迭代器`p`之后插入元素。由迭代器`b`和`e`指定范围。
        |
    | `lst.insert_after(p, il)`   | 在迭代器`p`之后插入元素。由`il`指定初始化列表。
          |
    | `emplace_after(p, args)`      | 使用`args`在`p`之后的位置，创建一个元素，返回一个指
向这个新元素的迭代器。若`p`为尾后迭代器，则函数行为未定义。 |
    | `lst.erase_after(p)`          | 删除`p`指向位置之后的元素，返回一个指向被删元素之后的
元素的迭代器，若`p`指向`lst`的尾元素或者是一个尾后迭代器，则函数行为未定义。 |
    | `lst.erase_after(b, e)`       | 类似上面，删除对象换成从`b`到`e`指定的范围。
          |
```

改变容器大小

```
    | 操作              | 解释
    |
    | ---------------- | ----------------------------------------------------------
---- |
    | `c.resize(n)`     | 调整`c`的大小为`n`个元素，若`n<c.size()`，则多出的元素被丢弃。
若必须添加新元素，对新元素进行值初始化 |
    | `c.resize(n, t)` | 调整`c`的大小为`n`个元素，任何新添加的元素都初始化为值`t`     |
```

获取迭代器

```
    | 操作                       | 解释                                          |
    | ------------------------ | ---------------------------------------- |
    | `c.begin()`, `c.end()`   | 返回指向`c`的首元素和尾元素之后位置的迭代器 |
    | `c.cbegin()`, `c.cend()` | 返回`const_iterator`                          |

    - 以`c`开头的版本是C++11新标准引入的
    - 当不需要写访问时，应该使用`cbegin`和`cend`。
```

反向容器的额外成员

```
    | 操作                         | 解释                                          |
    | -------------------------- | ---------------------------------------- |
    | `reverse_iterator`          | 按逆序寻址元素的迭代器                       |
    | `const_reverse_iterator`    | 不能修改元素的逆序迭代器                     |
    | `c.rbegin()`, `c.rend()`    | 返回指向`c`的尾元素和首元素之前位置的迭代器 |
    | `c.crbegin()`, `c.crend()` | 返回`const_reverse_iterator`                 |

    - 不支持`forward_list`
```

迭代器

```
    - 迭代器范围：`begin`到`end`，即第一个元素到最后一个元素的后面一个位置。
    - 左闭合区间：`[begin, end)`
    - 左闭合范围蕴含的编程设定：
      - 如果`begin`和`end`相等，则范围为空。
      - 如果二者不等，则范围至少包含一个元素，且`begin`指向该范围中的第一个元素。
      - 可以对`begin`递增若干次，使得`begin == end`。
```

容器操作可能使迭代器失效

- 在向容器添加元素后：
  - 如果容器是`vector`或`string`，且存储空间被重新分配，则指向容器的迭代器、指针、引用都会失效。
  - 对于`deque`，插入到除首尾位置之外的任何位置都会导致指向容器的迭代器、指针、引用失效。如果在首尾位置添加元素，迭代器会失效，但指向存在元素的引用和指针不会失效。
  - 对于`list`和`forward_list`，指向容器的迭代器、指针和引用依然有效。
- 在从一个容器中删除元素后：
  - 对于`list`和`forward_list`，指向容器其他位置的迭代器、引用和指针仍然有效。
  - 对于`deque`，如果在首尾之外的任何位置删除元素，那么指向被删除元素外其他元素的迭代器、指针、引用都会失效；如果是删除`deque`的尾元素，则尾后迭代器会失效，但其他不受影响；如果删除的是`deque`的头元素，这些也不会受影响。
  - 对于`vector`和`string`，指向被删元素之前的迭代器、引用、指针仍然有效。
  - 注意：当我们删除元素时，尾后迭代器总是会失效。
  - 注意：使用失效的迭代器、指针、引用是严重的运行时错误！
  - 建议：将要求迭代器必须保持有效的程序片段最小化。
  - 建议：不要保存`end`返回的迭代器。

容器内元素的类型约束

- 元素类型必须支持赋值运算；
- 元素类型的对象必须可以复制。
- 除了输入输出标准库类型外，其他所有标准库类型都是有效的容器元素类型。

vector对象是如何增长的

`vector`和`string`在内存中是连续保存的，如果原先分配的内存位置已经使用完，则需要重新分配新空间，将已有元素从就位置移动到新空间中，然后添加新元素。

管理容量的成员函数

操作	解释
`c.shrink_to_fit()`	将`capacity()`减少到和`size()`相同大小
`c.capacity()`	不重新分配内存空间的话，`c`可以保存多少个元素
`c.reverse(n)`	分配至少能容纳`n`个元素的内存空间

- `shrink_to_fit`只适用于`vector`、`string`和`deque`
- `capacity`和`reverse`只适用于`vector`和`string`。

额外的string操作

构造string的其他方法

```
    | 操作                          | 解释
              |
    | ------------------------- | ------------------------------------------
-------------- |
    | `string s(cp, n)`           | `s`是`cp`指向的数组中前`n`个字符的拷贝，此数组
       |
    | `string s(s2, pos2)`        | `s`是`string s2`从下标`pos2`开始的字符的拷贝。若
`pos2 > s2.size()`，则构造函数的行为未定义。 |
    | `string s(s2, pos2, len2)` | `s`是`string s2`从下标`pos2`开始的`len2`个字符的拷
贝。        |

    - `n`,`len2`,`pos2`都是无符号值。
```

substr操作

```
    | 操作                | 解释
      |
    | ------------------ | ------------------------------------------------------
------ |
    | `s.substr(pos, n)` | 返回一个`string`，包含`s`中从`pos`开始的`n`个字符的拷贝。
`pos`的默认值是0，`n`的默认值是`s.size() - pos`，即拷贝从`pos`开始的所有字符。 |
```

改变string的其他方法

```
    | 操作                        | 解释
            |
    | ------------------------- | ------------------------------------------
------------ |
    | `s.insert(pos, args)`     | 在`pos`之前插入`args`指定的字符。`pos`可以使是下标或者
迭代器。接受下标的版本返回指向`s`的引用；接受迭代器的版本返回指向第一个插入字符的迭代器。 |
    | `s.erase(pos, len)`       | 删除从`pos`开始的`len`个字符，如果`len`被省略，则删除
后面所有字符，返回指向`s`的引用。 |
    | `s.assign(args)`          | 将`s`中的字符替换成`args`指定的字符。返回一个指向`s`的
引用。 |
    | `s.append(args)`          | 将`args`指定的字符追加到`s`，返回一个指向`s`的引用。
      |
    | `s.replace(range, args)` | 删除`s`中范围`range`中的字符，替换成`args`指定的字符。
返回一个指向`s`的引用。 |
```

string搜索操作

```
    - `string`类提供了6个不同的搜索函数，每个函数都有4个重载版本。
    - 每个搜索操作都返回一个`string::size_type`值，表示匹配发生位置的下标。如果搜索失败则返
回一个名为`string::npos`的`static`成员（类型是`string::size_type`，初始化值是-1，也就是
`string`最大的可能大小）。

    | 搜索操作                       | 解释
   |
    | -------------------------- | ------------------------------------------
---- |
    | `s.find(args)`               | 查找`s`中`args`第一次出现的位置
  |
    | `s.rfind(args)`              | 查找`s`中`args`最后一次出现的位置
|
```

```
    | `s.find_first_of(args)`      | 在`s`中查找`args`中任何一个字符第一次出现的位置    |
    | `s.find_last_of(args)`       | 在`s`中查找`args`中任何一个字符最后一次出现的位置 |
    | `s.find_first_not_of(args)` | 在`s`中查找第一个不在`args`中的字符
|
    | `s.find_first_not_of(args)` | 在`s`中查找最后一个不在`args`中的字符
|

    args必须是一下的形式之一：

    | `args`形式    | 解释
|
    | ------------ | ------------------------------------------------------------
|
    | `c, pos`      | 从`s`中位置`pos`开始查找字符`c`。`pos`默认是0                  |
    | `s2, pos`     | 从`s`中位置`pos`开始查找字符串`s`。`pos`默认是0                |
    | `cp, pos`     | 从`s`中位置`pos`开始查找指针`cp`指向的以空字符结尾的C风格字符串。
`pos`默认是0 |
    | `cp, pos, n` | 从`s`中位置`pos`开始查找指针`cp`指向的前`n`个字符。`pos`和`n`无默认
值。 |
```

s.compare的几种参数形式

```
    逻辑类似于C标准库的`strcmp`函数，根据`s`是等于、大于还是小于参数指定的字符串，
`s.compare`返回0、正数或负数。

    | 参数形式                   | 解释
          |
    | ------------------------ | ------------------------------------------------
------------ |
    | `s2`                       | 比较`s`和`s2`
          |
    | `pos1, n1, s2`             | 比较`s`从`pos1`开始的`n1`个字符和`s2`
        |
    | `pos1, n1, s2, pos2, n2` | 比较`s`从`pos1`开始的`n1`个字符和`s2`
        |
    | `cp`                       | 比较`s`和`cp`指向的以空字符结尾的字符数组
       |
    | `pos1, n1, cp`             | 比较`s`从`pos1`开始的`n1`个字符和`cp`指向的以空字符结
尾的字符数组 |
    | `pos1, n1, cp, n2`         | 比较`s`从`pos1`开始的`n1`个字符和`cp`指向的地址开始
`n2`个字符 |
```

string和数值转换

```
    | 转换               | 解释
    |
    | ----------------- | ----------------------------------------------------------
----- |
    | `to_string(val)`  | 一组重载函数，返回数值`val`的`string`表示。`val`可以使任何算术
类型。对每个浮点类型和`int`或更大的整型，都有相应版本的`to_string()`。和往常一样，小整型会被
提升。 |
    | `stoi(s, p, b)`   | 返回`s`起始子串（表示整数内容）的数值，`p`是`s`中第一个非数值字
符的下标，默认是0，`b`是转换所用的基数。返回`int` |
    | `stol(s, p, b)`   | 返回`long`
      |
    | `stoul(s, p, b)`  | 返回`unsigned long`
     |
    | `stoll(s, p, b)`  | 返回`long long`
     |
    | `stoull(s, p, b)` | 返回`unsigned long long`
      |
    | `stof(s, p)`      | 返回`s`起始子串（表示浮点数内容）的数值，`p`是`s`中第一个非数值
字符的下标，默认是0。返回`float` |
    | `stod(s, p)`      | 返回`double`
      |
    | `stold(s, p)`     | 返回`long double`
     |
```

容器适配器（adapter）

```
    - 适配器是使一事物的行为类似于另一事物的行为的一种机制，例如`stack`可以使任何一种顺序容器
以栈的方式工作。
    - 初始化 `deque<int> deq; stack<int> stk(deq);` 从`deq`拷贝元素到`stk`。
    - 创建适配器时，指定一个顺序容器，可以覆盖默认的基础容器： `stack<string,
vector<string> > str_stk;`。
```

适配器的通用操作和类型

```
    | 操作              | 解释
    |
    | ---------------- | ----------------------------------------------------------
---- |
    | `size_type`      | 一种类型，须以保存当前类型的最大对象的大小                          |
    | `value_type`     | 元素类型
    |
    | `container_type` | 实现适配器的底层容器类型
|
    | `A a;`           | 创建一个名为`a`的空适配器
|
    | `A a(c)`         | 创建一个名为`a`的适配器，带有容器`c`的一个拷贝                  |
    | 关系运算符        | 每个适配器都支持所有关系运算符：`==`、`!=`、`<`、 `<=`、`>`、
`>=`这些运算符返回底层容器的比较结果 |
    | `a.empty()`      | 若`a`包含任何元素，返回`false`;否则返回`true`
|
    | `a.size()`       | 返回`a`中的元素数目
   |
    | `swap(a, b)`     | 交换`a`和`b`的内容，`a`和`b`必须有相同类型，包括底层容器类型也必
须相同 |
    | `a.swap(b)`      | 同上
    |
```

stack

```
    | 操作              | 解释
|
    | ---------------- | ----------------------------------------------------------
- |
    | `s.pop()`        | 删除栈顶元素，不返回。                                        |
    | `s.push(item)`   | 创建一个新元素，压入栈顶，该元素通过拷贝或移动`item`而来 |
    | `s.emplace(args)` | 同上，但元素由`args`来构造。                                  |
    | `s.top()`        | 返回栈顶元素，不删除。                                        |

    - 定义在`stack`头文件中。
    - `stack`默认基于`deque`实现，也可以在`list`或`vector`之上实现。
```

queue和priority_queue

```
    | 操作              | 解释                                        |
    | ---------------- | ------------------------------------------ |
    | `q.pop()`        | 删除队首元素，但不返回。                     |
    | `q.front()`      | 返回队首元素的值，不删除。                   |
    | `q.back()`       | 返回队尾元素的值，不删除。只适用于`queue` |
    | `q.top()`        | 返回具有最高优先级的元素值，不删除。        |
    | `q.push(item)`   | 在队尾压入一个新元素。                       |
    | `q.emplace(args)` |                                            |

    - 定义在`queue`头文件中。

    - `queue`默认基于`deque`实现，`priority_queue`默认基于`vector`实现。

    - `queue`可以在`list`或`vector`之上实现，`priority_queue`也可以用`deque`实现。
```

第十章 泛型算法

泛型算法

- 因为它们实现共同的操作，所以称之为“**算法**”；而“**泛型**”、指的是它们可以操作在多种容器类型上。
- 泛型算法本身不执行容器操作，只是单独依赖迭代器和迭代器操作实现。
- 头文件： `#include <algorithm>`或者 `#include <numeric>`(算数相关)
- 大多数算法是通过遍历两个迭代器标记的一段元素来实现其功能。
- 必要的编程假定：算法永远不会改变底层容器的大小。算法可能改变容器中保存的元素的值，也可能在容器内移动元素，但不能直接添加或者删除元素。

find

- `vector<int>::const_iterator result = find(vec.begin(), vec.end(), search_value);`
- 输入：两个标记范围的迭代器和目标查找值。返回：如果找到，返回对应的迭代器，否则返回第二个参数，即标记结尾的迭代器。

初识泛型算法

- 标准库提供了超过100个算法，但这些算法有一致的结构。
- 理解算法的最基本的方法是了解它们是否读取元素、改变元素、重排元素顺序。

只读算法

- 只读取范围中的元素，不改变元素。
- 如 `find`和 `accumulate`（在`numeric`中定义，求和）。
- `find_first_of`，输入：两对迭代器标记两段范围，在第一段中找第二段中任意元素，返回第一个匹配的元素，找不到返回第一段的`end`迭代器。
- 通常最好使用`cbegin`和`cend`。
- `equal`：确定两个序列是否保存相同的值。

写容器元素的算法

- 一些算法将新值赋予序列中的元素。
- 算法不检查写操作。
- `fill`： `fill(vec.begin(), vec.end(), 0);` 将每个元素重置为0
- `fill_n`： `fill_n(vec.begin(), 10, 0);`
- 插入迭代器`back_inserter`：
  - 用来确保算法有足够的空间存储数据。
  - `#include <iterator>`
  - `back_inserter(vec)`
- 拷贝算法`copy`：
- 输入：前两个参数指定输入范围，第三个指向目标序列。
- `copy (ilst.begin(), ilst.end(), back_inserter(ivec));`
- `copy`时必须保证目标目的序列至少要包含与输入序列一样多的元素。

重排容器元素的算法

- 这些算法会重排容器中元素的顺序。
- 排序算法`sort`：
  - 接受两个迭代器，表示要排序的元素范围。
- 消除重复`unique`：
  - 之前要先调用`sort`
  - 返回的迭代器指向最后一个不重复元素之后的位置。
  - 顺序会变，重复的元素被“删除”。
  - 并没有真正删除，真正删除必须使用容器操作。

定制操作

向算法传递函数：

- 谓词（`predicate`）：
  - 是一个**可调用的表达式**，返回结果是一个能用作条件的值
  - 一元谓词：接受一个参数
  - 二元谓词：接受两个参数

- 例子：
  - `stable_sort`：
    - 保留相等元素的原始相对位置。
    - `stable_sort(words.begin(), words.end(), isShorter);`

lambda表达式

- 有时可能希望操作可以接受更多的参数。
- `lambda`表达式表示一个可调用的代码单元，可以理解成是一个未命名的内联函数。
- 形式：`[capture list](parameter list) -> return type {function body}`。
  - 其中`capture list`捕获列表是一个`lambda`所在函数定义的局部变量的列表（通常为空）。不可忽略。
  - `return type`是返回类型。可忽略。
  - `parameter`是参数列表。可忽略。
  - `function body`是函数体。不可忽略。
  - `auto f = [] {return 42;}`

- 例子：
  - `find_if`:
    - 接受一对表示范围的迭代器和一个谓词，用来查找第一个满足特定要求的元素。返回第一个使谓词返回非0值的元素。
    - `auto wc = find_if(words.begin(), words.end(), [sz](const string &a) {return a.size() >= sz;});`
  - `for_each`：
    - 接受一个可调用对象，并对序列中每个元素调用此对象。
    - `for_each(wc, words.end(), [](const string &s){cout << s << " ";})`

lambda捕获和返回

- 定义`lambda`时会生成一个新的类类型和该类型的一个对象。
- 默认情况下，从`lambda`生成的类都包含一个对应该`lambda`所捕获的变量的数据成员，在`lambda`对象创建时被初始化。

- **值捕获**：前提是变量可以拷贝，`size_t v1 = 42; auto f = [v1] {return v1;};`。
- **引用捕获**：必须保证在`lambda`执行时，变量是存在的，`auto f2 = [&v1] {return v1;};`
- 尽量减少捕获的数据量，尽可能避免捕获指针或引用。
- **隐式捕获**：让编译器推断捕获列表，在捕获列表中写一个`&`（引用方式）或`=`（值方式）。`auto f3 = [=] {return v1;}`

lambda捕获列表：

捕获列表	解释
`[]`	空捕获列表。`lambda`不能使用所在函数中的变量。一个`lambda`只有在捕获变量后才能使用它们。
`[names]`	`names`是一个逗号分隔的名字列表，这些名字都是在`lambda`所在函数的局部变量，捕获列表中的变量都被拷贝，名字前如果使用了`&`，则采用引用捕获方式。
`[&]`	隐式捕获列表，采用引用捕获方式。`lambda`体中所使用的来自所在函数的实体都采用引用方式使用。
`[=]`	隐式捕获列表，采用值捕获方式。
`[&, identifier_list]`	`identifier_list`是一个逗号分隔的列表，包含0个或多个来自所在函数的变量。这些变量采用值捕获方式，而任何隐式捕获的变量都采用引用方式捕获。`identifier_list`中的名字前面不能使用`&`
`[=, identifier_list]`	`identifier_list`中的变量采用引用方式捕获，而任何隐式捕获的变量都采用值方式捕获。`identifier_list`中的名字不能包括`this`，且前面必须使用`&`

参数绑定

- `lambda`表达式更适合在一两个地方使用的简单操作。
- 如果是很多地方使用相同的操作，还是需要定义函数。
- 函数如何包装成一元谓词？使用参数绑定。
- 标准库`bind`函数：
  - 定义在头文件`functional`中，可以看做为一个通用的函数适配器。
  - `auto newCallable = bind(callable, arg_list);`
  - 我们再调用`newCallable`的时候，`newCallable`会调用`callable`并传递给它`arg_list`中的参数。
  - `_n`代表第n个位置的参数。定义在`placeholders`的命名空间中。`using std::placeholder::_1;`
  - `auto g = bind(f, a, b, _2, c, _1);`，调用`g(_1, _2)`实际上调用`f(a, b, _2, c, _1)`
  - 非占位符的参数要使用引用传参，必须使用标准库`ref`函数或者`cref`函数。

再探迭代器

插入迭代器

- 插入器是一种迭代器适配器，接受一个容器，生成一个迭代器，能实现向给定容器添加元素。
- 三种类型：
  - `back_inserter`：创建一个使用`push_back`的迭代器。
  - `front_inserter`创建一个使用`push_front`的迭代器。
  - `inserter`创建一个使用`insert`的迭代器。接受第二个参数，即一个指向给定容器的迭代器，元素会被查到迭代器所指向的元素之前。

插入迭代器操作：

操作	解释
`it=t`	在`it`指定的当前位置插入值`t`。假定`c`是`it`绑定的容器，依赖于插入迭代器的不同种类，此赋值会分别调用`c.push_back(t)`、`c.push_front(t)`、`c.insert(t, p)`，其中`p`是传递给`inserter`的迭代器位置
`*it, ++it, it++`	这些操作虽然存在，但不会对`it`做任何事情，每个操作都返回`it`

iostream迭代器

- 迭代器可与输入或输出流绑定在一起，用于迭代遍历所关联的 IO 流。
- 通过使用流迭代器，我们可以用泛型算法从流对象中读取数据以及向其写入数据。

istream_iterator的操作：

操作	解释
`istream_iterator<T> in(is);`	`in`从输入流`is`读取类型为`T`的值
`istream_iterator<T> end;`	读取类型是`T`的值的`istream_iterator`迭代器，表示尾后位置
`in1 == in2`	`in1`和`in2`必须读取相同类型。如果他们都是尾后迭代器，或绑定到相同的输入，则两者相等。
`in1 != in2`	类似上条
`*in`	返回从流中读取的值
`in->mem`	与`*(in).mem`含义相同
`++in, in++`	使用元素类型所定义的`>>`运算符从流中读取下一个值。前置版本返回一个指向递增后迭代器的引用，后置版本返回旧值。

ostream_iterator的操作：

操作	解释
`ostream_iterator<T> out(os);`	`out`将类型为`T`的值写到输出流`os`中
`ostream_iterator<T> out(os, d);`	`out`将类型为`T`的值写到输出流`os`中，每个值后面都输出一个`d`。`d`指向一个空字符结尾的字符数组。

| `out = val` | 用`<<`运算符将`val`写入到`out`所绑定的`ostream`中。`val`的类型必须和`out`可写的类型兼容。 |
| `*out, ++out, out++` | 这些运算符是存在的，但不对`out`做任何事情。每个运算符都返回`out`。 |

反向迭代器

- 反向迭代器就是在容器中从尾元素向首元素反向移动的迭代器。
- 对于反向迭代器，递增和递减的操作含义会颠倒。
- 实现向后遍历，配合`rbegin`和`rend`。

泛型算法结构

5类迭代器

迭代器类别	解释	支持的操作
输入迭代器	只读，不写；单遍扫描，只能递增	`==`,`!=`,`++`,`*`,`->`
输出迭代器	只写，不读；单遍扫描，只能递增	`++`,`*`
前向迭代器	可读写；多遍扫描，只能递增	`==`,`!=`,`++`,`*`,`->`
双向迭代器	可读写；多遍扫描，可递增递减	`==`,`!=`,`++`,`--`,`*`,`->`
随机访问迭代器	可读写，多遍扫描，支持全部迭代器运算	`==`,`!=`,`<`,`<=`,`>`,`>=`,`++`,`--`,`+`,`+=`,`-`,`-=`,`*`,`->`,`iter[n]`==`*(iter[n])`

算法的形参模式

- `alg(beg, end, other args);`
- `alg(beg, end, dest, other args);`
- `alg(beg, end, beg2, other args);`
- `alg(beg, end, beg2, end2, other args);`

其中，`alg`是算法名称，`beg`和`end`表示算法所操作的输入范围。`dest`、`beg2`、`end2`都是迭代器参数，是否使用要依赖于执行的操作。

算法命名规范

- 一些算法使用重载形式传递一个谓词。
- 接受一个元素值的算法通常有一个**不同名**的版本：加`_if`，接受一个谓词代替元素值。
- 区分拷贝元素的版本和不拷贝的版本：拷贝版本通常加`_copy`。

特定容器算法

- 对于`list`和`forward_list`，优先使用成员函数版本的算法而不是通用算法。

list和forward_list成员函数版本的算法：

```
      | 操作                      | 解释
           |
      | ------------------------ | ------------------------------------------------
------------- |
      | `lst.merge(lst2)`        | 将来自`lst2`的元素合并入`lst`，二者都必须是有序的，元
素将从`lst2`中删除。 |
      | `lst.merge(lst2, comp)` | 同上，给定比较操作。
        |
      | `lst.remove(val)`        | 调用`erase`删除掉与给定值相等(==)的每个元素
        |
      | `lst.remove_if(pred)`    | 调用`erase`删除掉令一元谓词为真的每个元素
        |
      | `lst.reverse()`          | 反转`lst`中元素的顺序
          |
      | `lst.sort()`             | 使用`<`排序元素
          |
      | `lst.sort(comp)`         | 使用给定比较操作排序元素
        |
      | `lst.unique()`           | 调用`erase`删除同一个值的连续拷贝。使用`==`。
        |
      | `lst.unique(pred)`       | 调用`erase`删除同一个值的连续拷贝。使用给定的二元谓词。
    |

      - 上面的操作都返回`void`

      **list和forward_list的splice成员函数版本的参数**：

      | 参数                | 解释
      |
      | ------------------ | ------------------------------------------------------
------- |
      | `(p, lst2)`         | `p`是一个指向`lst`中元素的迭代器，或者一个指向`flst`首前位置
的迭代器。函数将`lst2`中的所有元素移动到`lst`中`p`之前的位置或是`flst`中`p`之后的位置。将元
素从`lst2`中删除。`lst2`的类型必须和`lst`相同，而且不能是同一个链表。 |
      | `(p, lst2, p2)`    | 同上，`p2`是一个指向`lst2`中位置的有效的迭代器，将`p2`指向的
元素移动到`lst`中，或将`p2`之后的元素移动到`flst`中。`lst2`可以是于`lst`或`flst`相同的链
表。 |
      | `(p, lst2, b, e)` | `b`和`e`表示`lst2`中的合法范围。将给定范围中的元素从`lst2`移
动到`lst`或`first`中。`lst2`与`lst`可以使相同的链表，但`p`不能指向给定范围中的元素。 |

      - 使用`lst.splice(args)`或`flst.splice_after(args)`
```

第十一章 关联容器

- 关联容器和顺序容器的不同：关联容器中的元素时按照**关键字**来保存和访问的。
- 关联容器支持通过关键字来高效地查找和读取元素，基本的关联容器类型是 `map` 和 `set`。

关联容器类型：

容器类型	解释
按顺序存储	
`map`	关键数组：保存 `关键字-值` 对
`set`	关键字即值，即只保存关键字的容器
`multimap`	支持同一个键多次出现的 `map`
`multiset`	支持同一个键多次出现的 `set`
无序集合	
`unordered_map`	用哈希函数组织的 `map`
`unordered_set`	用哈希函数组织的 `set`
`unordered_multimap`	哈希组织的 `map`，关键字可以重复出现
`unordered_multiset`	哈希组织的 `set`，关键字可以重复出现

关联容器概述

定义关联容器

- 需要指定元素类型。
- 列表初始化：
  - `map`：`map<string, int> word_count = {{"a", 1}, {"b", 2}};`
  - `set`：`set<string> exclude = {"the", "a"};`

关键字类型的要求

- 对于有序容器，关键字类型必须定义元素比较的方法。默认是 `<`。
- 如果想传递一个比较的函数，可以这样定义：`multiset<Sales_data, decltype(compareIsbn)*> bookstore(compareIsbn);`

pair

- 在 `utility` 头文件中定义。
- 一个 `pair` 保存两个数据成员，两个类型不要求一样。

pair的操作：

操作	解释
`pair<T1, T2> p;`	`p` 是一个 `pair`，两个类型分别是 `T1` 和 `T2` 的成员都进行了值初始化。
`pair<T1, T2> p(v1, v2);`	`first` 和 `second` 分别用 `v1` 和 `v2` 进行初始化。
`pair<T1, T2>p = {v1, v2};`	等价于`p(v1, v2)
`make_pair(v1, v2);`	`pair` 的类型从 `v1` 和 `v2` 的类型推断出来。
`p.first`	返回 `p` 的名为 `first` 的数据成员。
`p.second`	返回 p 的名为 `second` 的数据成员。
`p1 relop p2`	运算关系符按字典序定义。
`p1 == p2`	必须两对元素两两相等
`p1 != p2`	同上

关联容器操作

关联容器额外的类型别名：

类型别名	解释
`key_type`	此容器类型的关键字类型
`mapped_type`	每个关键字关联的类型，只适用于 `map`
`value_type`	对于 `map`，是 `pair<const key_type, mapped_type>`；对于 `set`，和 `key_type` 相同。

关联容器迭代器

- 解引用一个关联容器迭代器时，会得到一个类型为容器的 `value_type` 的值的引用。
- `set` 的迭代器是 `const` 的。
- 遍历关联容器：使用 `begin` 和 `end`，遍历 `map`、`multimap`、`set`、`multiset` 时，迭代器按**关键字升序**遍历元素。

添加元素

关联容器 `insert` 操作：

insert 操作	关联容器
c.insert(v) c.emplace(args)	v 是 value_type 类型的对象；args 用来构造一个元素。 对于 map 和 set，只有元素的关键字不存在 c 中才插入或构造元素。函数返回一个 pair，包含一个迭代器，指向具有指定关键字的元素，以及一个指示插入是否成功的 bool 值。对于 multimap 和 multiset 则会插入范围中的每个元素。
c.insert(b, e) c.insert(il)	b 和 e 是迭代器，表示一个 c::value_type 类型值的范围；il 是这种值的花括号列表。函数返回 void。对于 map 和 set，只插入关键字不在 c 中的元素。
c.insert(p, v) c.emplace(p, args)	类似 insert(v)，但将迭代器 p 作为一个提示，指出从哪里开始搜索新元素应该存储的位置。返回一个迭代器，指向具有给定关键字的元素。

向 map 添加元素：

- word_count.insert({word, 1});
- word_count.insert(make_pair(word, 1));
- word_count.insert(pair<string, size_t>(word, 1));
- word_count.insert(map<string, size_t>::value_type (word, 1));

删除元素

从关联容器中删除元素：

操作	解释
c.erase(k)	从 c 中删除每个关键字为 k 的元素。返回一个 size_type 值，指出删除的元素的数量。
c.erase(p)	从 c 中删除迭代器 p 指定的元素。 p 必须指向 c 中一个真实元素，不能等于 c.end()。返回一个指向 p 之后元素的迭代器，若 p 指向 c 中的尾元素，则返回 c.end()
c.erase(b, e)	删除迭代器对 b 和 e 所表示范围中的元素。返回 e。

下标操作

map 和 unordered_map 的下标操作：

操作	解释
c[k]	返回关键字为 k 的元素；如果 k 不在 c 中，添加一个关键字为 k 的元素，对其值初始化。
c.at(k)	访问关键字为 k 的元素，带参数检查；若 k 不存在在 c 中，抛出一个 out_of_range 异常。

查找元素

在一个关联容器中查找元素:

操作	解释
`c.find(k)`	返回一个迭代器，指向第一个关键字为`k`的元素，若`k`不在容器中，则返回尾后迭代器
`c.count(k)`	返回关键字等于`k`的元素的数量。对于不允许重复关键字的容器，返回值永远是0或1。
`c.lower_bound(k)`	返回一个迭代器，指向第一个关键字**不小于**`k`的元素。
`c.upper_bound(k)`	返回一个迭代器，指向第一个关键字**大于**`k`的元素。
`c.equal_range(k)`	返回一个迭代器`pair`，表示关键字等于`k`的元素的范围。若`k`不存在，`pair`的两个成员均等于`c.end()`。

- `lower_bound`和`upper_bound`不适用于无序容器。
- 下标和`at`操作只适用于非`const`的`map`和`unordered_map`。

无序容器

- 有序容器使用比较运算符来组织元素；无序容器使用哈希函数和关键字类型的`==`运算符。
- 理论上哈希技术可以获得更好的性能。
- 无序容器在存储上组织为一组桶(bucket)，每个桶保存零个或多个元素。无序容器使用一个哈希函数将元素映射到桶。

无序容器管理操作:

操作	解释
桶接口	
`c.bucket_count()`	正在使用的桶的数目
`c.max_bucket_count()`	容器能容纳的最多的桶的数目
`c.bucket_size(n)`	第 `n` 个桶中有多少个元素
`c.bucket(k)`	关键字为 `k` 的元素在哪个桶中
桶迭代	
`local_iterator`	可以用来访问桶中元素的迭代器类型
`const_local_iterator`	桶迭代器的 `const` 版本
`c.begin(n)`，`c.end(n)`	桶 `n` 的首元素迭代器
`c.cbegin(n)`，`c.cend(n)`	与前两个函数类似，但返回 `const_local_iterator`。
哈希策略	
`c.load_factor()`	每个桶的平均元素数量，返回 `float` 值。
`c.max_load_factor()`	`c` 试图维护的平均比桶大小，返回 `float` 值。`c` 会在需要时添加新的桶，以使得 `load_factor<=max_load_factor`
`c.rehash(n)`	重组存储，使得 `bucket_count>=n`，且 `bucket_count>size/max_load_factor`
`c.reverse(n)`	重组存储，使得 `c` 可以保存 `n` 个元素且不必 `rehash`。

第十二章 动态内存

- 对象的生命周期：
  - 全局对象在程序启动时分配，结束时销毁。
  - 局部对象在进入程序块时创建，离开块时销毁。
  - 局部 `static` 对象在第一次使用前分配，在程序结束时销毁。
  - 动态分配对象：只能显式地被释放。
- 对象的内存位置：
  - **静态内存**用来保存局部 `static` 对象、类 `static` 对象、定义在任何函数之外的变量。
  - **栈内存**用来保存定义在函数内的非 `static` 对象。
  - **堆内存**，又称自由空间，用来存储**动态分配**的对象。

动态内存与智能指针

- 动态内存管理：
  - `new`：在动态内存中为对象分配空间并返回一个指向该对象的指针。
  - `delete`：接受一个动态对象的指针，销毁该对象，并释放与之关联的内存。
- 智能指针：
  - 管理动态对象。
  - 行为类似常规指针。

- 负责自动释放所指向的对象。
- 智能指针也是模板。

shared_ptr类

shared_ptr和unique_ptr都支持的操作：

操作	解释
`shared_ptr<T> sp` `unique_ptr<T> up`	空智能指针，可以指向类型是`T`的对象
`p`	将`p`用作一个条件判断，若`p`指向一个对象，则为`true`
`*p`	解引用`p`，获得它指向的对象。
`p->mem`	等价于`(*p).mem`
`p.get()`	返回`p`中保存的指针，要小心使用，若智能指针释放了对象，返回的指针所指向的对象也就消失了。
`swap(p, q)` `p.swap(q)`	交换`p`和`q`中的指针

shared_ptr独有的操作：

操作	解释
`make_shared<T>(args)`	返回一个`shared_ptr`，指向一个动态分配的类型为`T`的对象。使用`args`初始化此对象。
`shared_ptr<T>p(q)`	`p`是`shared_ptr q`的拷贝；此操作会**递增**`q`中的计数器。`q`中的指针必须能转换为`T*`
`p = q`	`p`和`q`都是`shared_ptr`，所保存的指针必须能互相转换。此操作会**递减**`p`的引用计数，**递增**`q`的引用计数；若`p`的引用计数变为0，则将其管理的原内存释放。
`p.unique()`	若`p.use_count()`是1，返回`true`；否则返回`false`
`p.use_count()`	返回与`p`共享对象的智能指针数量；可能很慢，主要用于调试。

- **使用动态内存的三种原因**：
  - 程序不知道自己需要使用多少对象（比如容器类）。
  - 程序不知道所需要对象的准确类型。
  - 程序需要在多个对象间共享数据。

直接管理内存

- 用`new`动态分配和初始化对象。
  - `new`无法为分配的对象命名（因为自由空间分配的内存是无名的），因此是返回一个指向该对象的指针。
  - `int *pi = new int(123);`
  - 一旦内存耗尽，会抛出类型是`bad_alloc`的异常。
- 用`delete`将动态内存归还给系统。

    - 接受一个指针，指向要释放的对象。
    - `delete`后的指针称为空悬指针（dangling pointer）。
- 使用`new`和`delete`管理动态内存存在三个常见问题：
    - 1.忘记`delete`内存。
    - 2.使用已经释放掉的对象。
    - 3.同一块内存释放两次。
- 坚持只使用智能指针可以避免上述所有问题。

shared_ptr和new结合使用

定义和改变shared_ptr的其他方法：

操作	解释
`shared_ptr<T> p(q)`	`p`管理内置指针`q`所指向的对象；`q`必须指向`new`分配的内存，且能够转换为`T*`类型
`shared_ptr<T> p(u)`	`p`从`unique_ptr u`那里接管了对象的所有权；将`u`置为空
`shared_ptr<T> p(q, d)`	`p`接管了内置指针`q`所指向的对象的所有权。`q`必须能转换为`T*`类型。`p`将使用可调用对象`d`来代替`delete`。
`shared_ptr<T> p(p2, d)`	`p`是`shared_ptr p2`的拷贝，唯一的区别是`p`将可调用对象`d`来代替`delete`。
`p.reset()`	若`p`是唯一指向其对象的`shared_ptr`，`reset`会释放此对象。若传递了可选的参数内置指针`q`，会令`p`指向`q`，否则会将`p`置空。若还传递了参数`d`，则会调用`d`而不是`delete`来释放`q`。
`p.reset(q)`	同上
`p.reset(q, d)`	同上

智能指针和异常

- 如果使用智能指针，即使程序块由于异常过早结束，智能指针类也能确保在内存不需要的时候将其释放。
- **智能指针陷阱**：
    - 不用相同的内置指针初始化（或`reset`）多个智能指针
    - 不`delete get()`返回的指针。
    - 如果你使用`get()`返回的指针，记得当最后一个对应的智能指针销毁后，你的指针就无效了。
    - 如果你使用智能指针管理的资源不是`new`分配的内存，记住传递给它一个删除器。

unique_ptr

- 某一个时刻只能有一个`unique_ptr`指向一个给定的对象。
- 不支持拷贝或者赋值操作。
- 向后兼容：`auto_ptr`：老版本，具有`unique_ptr`的部分特性。特别是，不能在容器中保存`auto_ptr`，也不能从函数返回`auto_ptr`。

unique_ptr操作:

操作	解释
`unique_ptr<T> u1`	空`unique_ptr`，可以指向类型是`T`的对象。`u1`会使用`delete`来是释放它的指针。
`unique_ptr<T, D> u2`	`u2`会使用一个类型为`D`的可调用对象来释放它的指针。
`unique_ptr<T, D> u(d)`	空`unique_ptr`，指向类型为`T`的对象，用类型为`D`的对象`d`代替`delete`
`u = nullptr`	释放`u`指向的对象，将`u`置为空。
`u.release()`	`u`放弃对指针的控制权，返回指针，并将`u`置空。
`u.reset()`	释放`u`指向的对象
`u.reset(q)`	令`u`指向`q`指向的对象
`u.reset(nullptr)`	将`u`置空

weak_ptr

- `weak_ptr`是一种不控制所指向对象生存期的智能指针。
- 指向一个由`shared_ptr`管理的对象，不改变`shared_ptr`的引用计数。
- 一旦最后一个指向对象的`shared_ptr`被销毁，对象就会被释放，不管有没有`weak_ptr`指向该对象。

weak_ptr操作:

操作	解释
`weak_ptr<T> w`	空`weak_ptr`可以指向类型为`T`的对象
`weak_ptr<T> w(sp)`	与`shared_ptr`指向相同对象的`weak_ptr`。`T`必须能转换为`sp`指向的类型。
`w = p`	`p`可以是`shared_ptr`或一个`weak_ptr`。赋值后`w`和`p`共享对象。
`w.reset()`	将`w`置为空。
`w.use_count()`	与`w`共享对象的`shared_ptr`的数量。
`w.expired()`	若`w.use_count()`为0，返回`true`，否则返回`false`
`w.lock()`	如果`expired`为`true`，则返回一个空`shared_ptr`；否则返回一个指向`w`的对象的`shared_ptr`。

动态数组

new和数组

- `new`一个动态数组：
  - 类型名之后加一对方括号，指明分配的对象数目（必须是整型，不必是常量）。
  - 返回**指向第一个对象的指针。**
  - `int *p = new int[size];`
- `delete`一个动态数组：
  - `delete [] p;`
- `unique_ptr`和数组：
  - 指向数组的`unique_ptr`不支持成员访问运算符（点和箭头）。

操作	解释
`unique_ptr<T[]> u`	`u`可以指向一个动态分配的数组，整数元素类型为`T`
`unique_ptr<T[]> u(p)`	`u`指向内置指针`p`所指向的动态分配的数组。`p`必须能转换为类型`T*`。
`u[i]`	返回`u`拥有的数组中位置`i`处的对象。`u`必须指向一个数组。

allocator类

- 标准库`allocator`类定义在头文件`memory`中，帮助我们将内存分配和对象构造分离开。
- 分配的是原始的、未构造的内存。
- `allocator`是一个模板。
- `allocator<string> alloc;`

标准库allocator类及其算法：

操作	解释
`allocator<T> a`	定义了一个名为`a`的`allocator`对象，它可以为类型为`T`的对象分配内存
`a.allocate(n)`	分配一段原始的、未构造的内存，保存`n`个类型为`T`的对象。
`a.deallocate(p, n)`	释放从`T*`指针`p`中地址开始的内存，这块内存保存了`n`个类型为`T`的对象；`p`必须是一个先前由`allocate`返回的指针。且`n`必须是`p`创建时所要求的大小。在调用`deallocate`之前，用户必须对每个在这块内存中创建的对象调用`destroy`。
`a.construct(p, args)`	`p`必须是一个类型是`T*`的指针，指向一块原始内存；`args`被传递给类型为`T`的构造函数，用来在`p`指向的内存中构造一个对象。
`a.destroy(p)`	`p`为`T*`类型的指针，此算法对`p`指向的对象执行析构函数。

allocator伴随算法：

操作	解释
`uninitialized_copy(b, e, b2)`	从迭代器 `b` 和 `e` 给定的输入范围中拷贝元素到迭代器 `b2` 指定的未构造的原始内存中。 `b2` 指向的内存必须足够大，能够容纳输入序列中元素的拷贝。
`uninitialized_copy_n(b, n, b2)`	从迭代器 `b` 指向的元素开始，拷贝 `n` 个元素到 `b2` 开始的内存中。
`uninitialized_fill(b, e, t)`	在迭代器 `b` 和 `e` 执行的原始内存范围中创建对象，对象的值均为 `t` 的拷贝。
`uninitialized_fill_n(b, n, t)`	从迭代器 `b` 指向的内存地址开始创建 `n` 个对象。 `b` 必须指向足够大的未构造的原始内存，能够容纳给定数量的对象。

- 定义在头文件 `memory` 中。
- 在给定目的位置创建元素，而不是由系统分配内存给他们。

第十三章 拷贝控制

拷贝控制操作（copy control）：

- 拷贝构造函数（copy constructor）
- 拷贝赋值运算符（copy-assignment operator）
- 移动构造函数（move constructor）
- 移动赋值函数（move-assignement operator）
- 析构函数（destructor）

拷贝、赋值和销毁

拷贝构造函数

- 如果一个构造函数的第一个参数是**自身类类型的引用**，且任何额外参数都有默认值，则此构造函数是**拷贝构造函数。**
- `class Foo{ public: Foo(const Foo&); }`
- **合成的拷贝构造函数**（synthesized copy constructor）：会将参数的成员逐个拷贝到正在创建的对象中。
- **拷贝初始化**:
  - 将右侧运算对象拷贝到正在创建的对象中，如果需要，还需进行类型转换。
  - 通常使用拷贝构造函数完成。
  - `string book = "9-99";`
  - 出现场景:
    - 用 `=` 定义变量时。
    - 将一个对象作为实参传递给一个非引用类型的形参。
    - 从一个返回类型为非引用类型的函数返回一个对象。
    - 用花括号列表初始化一个数组中的元素或者一个聚合类中的成员。

拷贝赋值运算符

- **重载赋值运算符**：
  - 重写一个名为`operator=`的函数.
  - 通常返回一个指向其左侧运算对象的引用。
  - `Foo& operator=(const Foo&);`
- **合成拷贝赋值运算符**：
  - 将右侧运算对象的每个非`static`成员赋予左侧运算对象的对应成员。

析构函数

- 释放对象所使用的资源，并销毁对象的非`static`数据成员。
- 名字由波浪号接类名构成。没有返回值，也不接受参数。
- `~Foo();`
- 调用时机：
  - 变量在离开其作用域时。
  - 当一个对象被销毁时，其成员被销毁。
  - 容器被销毁时，其元素被销毁。
  - 动态分配的对象，当对指向它的指针应用`delete`运算符时。
  - 对于临时对象，当创建它的完整表达式结束时。
- **合成析构函数**：
  - 空函数体执行完后，**成员会被自动销毁。**
  - 注意：析构函数体本身并不直接销毁成员。

三/五法则

- 需要析构函数的类也需要拷贝和赋值操作。
- 需要拷贝操作的类也需要赋值操作，反之亦然。

使用=default

- 可以通过将拷贝控制成员定义为`=default`来显式地要求编译器生成合成的版本。
- 合成的函数将隐式地声明为内联的。

阻止拷贝

- 大多数类应该定义默认构造函数、拷贝构造函数和拷贝赋值运算符，无论是隐式地还是显式地。
- 定义删除的函数：`=delete`。
- 虽然声明了它们，但是不能以任何方式使用它们。
- 析构函数不能是删除的成员。
- 如果一个类有数据成员不能默认构造、拷贝、复制或者销毁，则对应的成员函数将被定义为删除的。
- 老版本使用`private`声明来阻止拷贝。

拷贝控制和资源管理

- 类的行为可以像一个值，也可以像一个指针。
  - 行为像值：对象有自己的状态，副本和原对象是完全独立的。
  - 行为像指针：共享状态，拷贝一个这种类的对象时，副本和原对象使用相同的底层数据。

交换操作

- 管理资源的类通常还定义一个名为`swap`的函数。
- 经常用于重排元素顺序的算法。
- 用`swap`而不是`std::swap`。

对象移动

- 很多拷贝操作后，原对象会被销毁，因此引入移动操作可以大幅度提升性能。
- 在新标准中，我们可以用容器保存不可拷贝的类型，只要它们可以被移动即可。
- 标准库容器、`string`和`shared_ptr`类既可以支持移动也支持拷贝。`IO`类和`unique_ptr`类可以移动但不能拷贝。

右值引用

- 新标准引入右值引用以支持移动操作。
- 通过`&&`获得右值引用。
- 只能绑定到一个将要销毁的对象。
- 常规引用可以称之为左值引用。
- 左值持久，右值短暂。

move函数:

- `int &&rr2 = std::move(rr1);`
- `move`告诉编译器，我们有一个左值，但我希望像右值一样处理它。
- 调用`move`意味着：除了对`rr1`赋值或者销毁它外，我们将不再使用它。

移动构造函数和移动赋值运算符

- **移动构造函数**:
  - 第一个参数是该类类型的一个引用，关键是，这个引用参数是一个**右值引用**。
  - `StrVec::StrVec(StrVec &&s) noexcept{}`
  - 不分配任何新内存，只是接管给定的内存。
- **移动赋值运算符**:
  - `StrVec& StrVec::operator=(StrVec && rhs) noexcept{}`
- 移动右值，拷贝左值。
- 如果没有移动构造函数，右值也被拷贝。
- 更新三/五法则：如果一个类定义了任何一个拷贝操作，它就应该定义所有五个操作。
- 移动迭代器:
  - `make_move_iterator`函数讲一个普通迭代器转换为一个移动迭代器。
- 建议：小心地使用移动操作，以获得性能提升。

右值引用和成员函数

- 区分移动和拷贝的重载函数通常有一个版本接受一个`const T&`，而另一个版本接受一个`T&&`。
- 引用限定符:
  - 在参数列表后面防止一个`&`，限定只能向可修改的左值赋值而不能向右值赋值。

第十四章 重载运算与类型转换

基本概念

- 重载运算符是具有特殊名字的函数：由关键字`operator`和其后要定义的运算符号共同组成。
- 当一个重载的运算符是成员函数时，`this`绑定到左侧运算对象。动态运算符符函数的参数数量比运算对象的数量**少一个**。
- 只能重载大多数的运算符，而不能发明新的运算符号。
- 重载运算符的优先级和结合律跟对应的内置运算符保持一致。
- 调用方式：
  - `data1 + data2;`
  - `operator+(data1, data2);`
- 是否是成员函数：
  - 赋值（`=`）、下标（`[]`）、调用（`()`）和成员访问箭头（`->`）运算符必须是成员。
  - 复合赋值运算符一般来说是成员。
  - 改变对象状态的运算符或者和给定类型密切相关的运算符通常是成员，如递增、解引用。
  - 具有对称性的运算符如算术、相等性、关系和位运算符等，通常是非成员函数。

运算符:

可以被重载	不可以被重载
`+`, `-`, `*`, `/`, `%`, `^`	`::`, `.*`, `.`, `? :`,
`&`, `|`, `~`, `!`, `,`, `=`	
`<`, `>`, `<=`, `>=`, `++`, `--`	
`<<`, `>>`, `==`, `!=`, `&&`, `||`	
`+=`, `-=`, `/=`, `%=`, `^=`, `&=`	
`|=`, `*=`, `<<=`, `>>=`, `[]`, `()`	
`->`, `->*`, `new`, `new[]`, `delete`, `delete[]`	

输入和输出运算符

重载输出运算符<<

- 第一个形参通常是一个非常量的`ostream`对象的引用。非常量是因为向流中写入会改变其状态；而引用是因为我们无法复制一个`ostream`对象。
- 输入输出运算符必须是非成员函数。

重载输入运算符>>

- 第一个形参通常是运算符将要读取的流的因不用，第二个形参是将要读取到的（非常量）对象的引用。
- 输入运算符必须处理输入可能失败的情况，而输出运算符不需要。

算数和关系运算符（+、-、*、/）

- 如果类同时定义了算数运算符和相关的复合赋值运算符，则通常情况下应该使用复合赋值来实现算数运算符。

相等运算符==

- 如果定义了`operator==`，则这个类也应该定义`operator!=`。
- 相等运算符和不等运算符的一个应该把工作委托给另一个。
- 相等运算符应该具有传递性。
- 如果某个类在逻辑上有相等性的含义，则该类应该定义`operator==`，这样做可以使用户更容易使用标准库算法来处理这个类。

关系运算符

- 如果存在唯一一种逻辑可靠的`<`定义，则应该考虑为这个类定义`<`运算符。如果同时还包含`==`，则当且晋档`<`的定义和`++`产生的结果一直时才定义`<`运算符。

赋值运算符=

- 我们可以重载赋值运算符。不论形参的类型是什么，赋值运算符都必须定义为成员函数。
- 赋值运算符必须定义成类的成员，复合赋值运算符通常情况下也应该这么做。这两类运算符都应该返回左侧运算对象的引用。

下标运算符[]

- 下标运算符必须是成员函数。
- 一般会定义两个版本：
    - 1.返回普通引用。
    - 2.类的常量成员，并返回常量引用。

递增和递减运算符（++、--）

- 定义递增和递减运算符的类应该同时定义前置版本和后置版本。
- 通常应该被定义成类的成员。
- 为了和内置版本保持一致，前置运算符应该返回递增或递减后对象的引用。
- 同样为了和内置版本保持一致，后置运算符应该返回递增或递减前对象的值，而不是引用。
- 后置版本接受一个额外的，不被使用的`int`类型的形参。因为不会用到，所以无需命名。

成员访问运算符（*、->）

- 箭头运算符必须是类的成员。解引用运算符通常也是类的成员，尽管并非必须如此。
- 重载的箭头运算符必须返回类的指针或者自定义了箭头运算符的某个类的对象。
- 解引用和乘法的区别是一个是一元运算符，一个是二元运算符。

函数调用运算符

- 可以像使用函数一样，调用该类的对象。因为这样对待类同时也能存储状态，所以与普通函数相比更加灵活。
- 函数调用运算符必须是成员函数。
- 一个类可以定义多个不同版本的调用运算符，相互之间应该在参数数量或类型上有所区别。
- 如果累定义了调用运算符，则该类的对象称作**函数对象**。

`lambda`是函数对象

- `lambda`捕获变量：`lambda`产生的类必须为每个值捕获的变量建立对应的数据成员，同时创建构造函数。

标准库定义的函数对象

标准库函数对象:

算术	关系	逻辑
`plus<Type>`	`equal_to<Type>`	`logical_and<Type>`
`minus<Type>`	`not_equal_to<Type>`	`logical_or<Type>`
`multiplies<Type>`	`greater<Type>`	`logical_not<Type>`
`divides<Type>`	`greater_equal<Type>`	
`modulus<Type>`	`less<Type>`	
`negate<Type>`	`less_equal<Type>`	

- 可以在算法中使用标准库函数对象。

可调用对象与function

标准库function类型:

操作	解释
`function<T> f;`	`f`是一个用来存储可调用对象的空`function`，这些可调用对象的调用形式应该与类型`T`相同。
`function<T> f(nullptr);`	显式地构造一个空`function`
`function<T> f(obj)`	在`f`中存储可调用对象`obj`的副本
`f`	将`f`作为条件：当`f`含有一个可调用对象时为真；否则为假。
定义为`function<T>`的成员的类型	
`result_type`	该`function`类型的可调用对象返回的类型
`argument_type`	当`T`有一个或两个实参时定义的类型。如果`T`只有一个实参，则`argument_type`
`first_argument_type`	第一个实参的类型
`second_argument_type`	第二个实参的类型

- 例如：声明一个`function`类型，它可以表示接受两个`int`，返回一个`int`的可调用对象。`function<int(int, int)>`

重载、类型转换、运算符

类型转换运算符

- 类型转换运算符是类的一种特殊成员函数，它负责将一个类类型的值转换成其他类型。类型转换函数的一般形式如下：`operator type() const;`
- 一个类型转换函数必须是类的成员函数；它不能声明返回类型，形参列表也必须为空。类型转换函数通常应该是`const`。
- 避免过度使用类型转换函数。
- C++11引入了显式的类型转换运算符。
- 向`bool`的类型转换通常用在条件部分，因此`operator bool`一般定义成`explicit`的。

避免有二义性的类型转换

- 通常，不要为类第几个亿相同的类型转换，也不要在类中定义两个及以上转换源或转换目标是算术类型的转换。
- 在调用重载函数时，如果需要额外的标准类型转换，则该转换的级别只有当所有可行函数都请求同一个用户定义的类型转换时才有用。如果所需的用户定义的类型转换不止一个，则该调用具有二义性。

函数匹配与重载运算符

- 如果`a`是一种类型，则表达式`a sym b`可能是：
  - `a.operatorsym(b);`
  - `operatorsym(a,b);`
- 如果我们队同一个类既提供了转换目标是算术类型的类型转换，也提供了重载的运算符，则将会遇到重载运算符与内置运算符的二义性问题。

第十五章 面向对象程序设计

OOP：概述

- 面向对象程序设计（object-oriented programming）的核心思想是数据抽象、继承和动态绑定。
- **继承**（inheritance）：
  - 通过继承联系在一起的类构成一种层次关系。
  - 通常在层次关系的根部有一个**基类**（base class）。
  - 其他类直接或者简介从基类继承而来，这些继承得到的类成为**派生类**（derived class）。
  - 基类负责定义在层次关系中所有类共同拥有的成员，而每个派生类定义各自特有的成员。
  - 对于某些函数，基类希望它的派生类个自定义适合自己的版本，此时基类就将这些函数声明成**虚函数**（virtual function）。
  - 派生类必须通过使用**类派生列表**（class derivation list）明确指出它是从哪个基类继承而来。形式：一个冒号，后面紧跟以逗号分隔的基类列表，每个基类前都可以有访问说明符。`class Bulk_quote : public Quote{};`
  - 派生类必须在其内部对所有重新定义的虚函数进行声明。可以在函数之前加上`virtual`关键字，也可以不加。C++11新标准允许派生类显式地注明它将使用哪个成员函数改写基类的虚函数，即在函数的形参列表之后加一个`override`关键字。
- **动态绑定**（dynamic binding，又称运行时绑定）：
  - 使用同一段代码可以分别处理基类和派生类的对象。
  - 函数的运行版本由实参决定，即在运行时选择函数的版本。

定义基类和派生类

定义基类

- 基类通常都应该定义一个虚析构函数，即使该函数不执行任何实际操作也是如此。
- 基类通过在其成员函数的声明语句前加上关键字`virtual`使得该函数执行**动态绑定**。
- 如果成员函数没有被声明为虚函数，则解析过程发生在编译时而非运行时。
- 访问控制：
  - `protected`： 基类和和其派生类还有友元可以访问。
  - `private`： 只有基类本身和友元可以访问。

定义派生类

- 派生类必须通过类派生列表（class derivation list）明确指出它是从哪个基类继承而来。形式：冒号，后面紧跟以逗号分隔的基类列表，每个基类前面可以有一下三种访问说明符的一个：`public`、`protected`、`private`。
- C++11新标准允许派生类显式地注明它将使用哪个成员函数改写基类的虚函数，即在函数的形参列表之后加一个`override`关键字。
- 派生类构造函数：派生类必须使用基类的构造函数去初始化它的基类部分。
- 静态成员：如果基类定义了一个基类成员，则在整个继承体系中只存在该成员的唯一定义。
- 派生类的声明：声明中不包含它的派生列表。
- C++11新标准提供了一种防止继承的方法，在类名后面跟一个关键字`final`。

类型转换与继承

- 理解基类和派生类之间的类型抓换是理解C++语言面向对象编程的关键所在。
- 可以将基类的指针或引用绑定到派生类对象上。
- 不存在从基类向派生类的隐式类型转换。
- 派生类向基类的自动类型转换只对指针或引用类型有效，对象之间不存在类型转换。

虚函数

- 使用虚函数可以执行动态绑定。
- OOP的核心思想是多态性（polymorphism）。
- 当且仅当对通过指针或引用调用虚函数时，才会在运行时解析该调用，也只有在这种情况下对象的动态类型才有可能与静态类型不同。
- 派生类必须在其内部对所有重新定义的虚函数进行声明。可以在函数之前加上`virtual`关键字，也可以不加。
- C++11新标准允许派生类显式地注明它将使用哪个成员函数改写基类的虚函数，即在函数的形参列表之后加一个`override`关键字。
- 如果我们想覆盖某个虚函数，但不小心把形参列表弄错了，这个时候就不会覆盖基类中的虚函数。加上`override`可以明确程序员的意图，让编译器帮忙确认参数列表是否出错。
- 如果虚函数使用默认实参，则基类和派生类中定义的默认实参最好一致。
- 通常，只有成员函数（或友元）中的代码才需要使用**作用域运算符**（`::`）来回避虚函数的机制。

抽象基类

- **纯虚函数**（pure virtual）：清晰地告诉用户当前的函数是没有实际意义的。纯虚函数无需定义，只用在函数体的位置前书写`=0`就可以将一个虚函数说明为纯虚函数。
- 含有纯虚函数的类是**抽象基类**（abstract base class）。不能创建抽象基类的对象。

访问控制与继承

- 受保护的成员：
  - `protected`说明符可以看做是`public`和`private`中的产物。
  - 类似于私有成员，受保护的成员对类的用户来说是不可访问的。
  - 类似于公有成员，受保护的成员对于派生类的成员和友元来说是可访问的。
  - 派生类的成员或友元只能通过派生类对象来访问基类的受保护成员。派生类对于一个基类对象中的受保护成员没有任何访问特权。
- 派生访问说明符：
  - 对于派生类的成员（及友元）能否访问其直接积累的成员没什么影响。
  - 派生访问说明符的目的是：控制派生类用户对于基类成员的访问权限。比如`struct Priv_Drev: private Base{}`意味着在派生类`Priv_Drev`中，从`Base`继承而来的部分都是`private`的。
- 友元关系不能继承。
- 改变个别成员的可访问性：使用`using`。
- 默认情况下，使用`class`关键字定义的派生类是私有继承的；使用`struct`关键字定义的派生类是公有继承的。

继承中的类作用域

- 每个类定义自己的作用域，在这个作用域内我们定义类的成员。当存在继承关系时，派生类的作用域嵌套在其基类的作用域之内。
- 派生类的成员将隐藏同名的基类成员。
- 除了覆盖继承而来的虚函数之外，派生类最好不要重用其他定义在基类中的名字。

构造函数与拷贝控制

虚析构函数

- 基类通常应该定义一个虚析构函数，这样我们就能动态分配继承体系中的对象了。
- 如果基类的析构函数不是虚函数，则`delete`一个指向派生类对象的基类指针将产生未定义的行为。
- 虚析构函数将阻止合成移动操作。

合成拷贝控制与继承

- 基类或派生类的合成拷贝控制成员的行为和其他合成的构造函数、赋值运算符或析构函数类似：他们对类本身的成员依次进行初始化、赋值或销毁的操作。

派生类的拷贝控制成员

- 当派生类定义了拷贝或移动操作时，该操作负责拷贝或移动包括基类部分成员在内的整个对象。
- 派生类析构函数：派生类析构函数先执行，然后执行基类的析构函数。

继承的构造函数

- C++11新标准中，派生类可以重用其直接基类定义的构造函数。
- 如`using Disc_quote::Disc_quote;`，注明了要继承`Disc_quote`的构造函数。

容器与继承

- 当我们使用容器存放继承体系中的对象时，通常必须采用间接存储的方式。

- 派生类对象直接赋值给积累对象，其中的派生类部分会被切掉。
- 在容器中放置（智能）指针而非对象。
- 对于C++面向对象的编程来说，一个悖论是我们无法直接使用对象进行面向对象编程。相反，我们必须使用指针和引用。因为指针会增加程序的复杂性，所以经常定义一些辅助的类来处理这些复杂的情况。

文本查询程序再探

- 使系统支持：单词查询、逻辑非查询、逻辑或查询、逻辑与查询。

面向对象的解决方案

- 将几种不同的查询建模成相互独立的类，这些类共享一个公共基类：
  - `WordQuery`
  - `NotQuery`
  - `OrQuery`
  - `AndQuery`
- 这些类包含两个操作：
  - `eval`：接受一个`TextQuery`对象并返回一个`QueryResult`。
  - `rep`：返回基础查询的`string`表示形式。
- 继承和组合：
  - 当我们令一个类公有地继承另一个类时，派生类应当反映与基类的“是一种（Is A）”的关系。
  - 类型之间另一种常见的关系是“有一个（Has A）”的关系。
- 对于面向对象编程的新手来说，想要理解一个程序，最困难的部分往往是理解程序的设计思路。一旦掌握了设计思路，接下来的实现也就水到渠成了。

Query程序设计:

操作	解释
`Query` 程序接口类和操作	
`TextQuery`	该类读入给定的文件并构建一个查找图。包含一个 `query` 操作，它接受一个 `string` 实参，返回一个 `QueryResult` 对象；该 `QueryResult` 对象表示 `string` 出现的行。
`QueryResult`	该类保存一个 `query` 操作的结果。
`Query`	是一个接口类，指向 `Query_base` 派生类的对象。
`Query q(s)`	将 `Query` 对象 `q` 绑定到一个存放着 `string s` 的新 `WordQuery` 对象上。
`q1 & q2`	返回一个 `Query` 对象，该 `Query` 绑定到一个存放 `q1` 和 `q2` 的新 `AndQuery` 对象上。
`q1 | q2`	返回一个 `Query` 对象，该 `Query` 绑定到一个存放 `q1` 和 `q2` 的新 `OrQuery` 对象上。
`~q`	返回一个 `Query` 对象，该 `Query` 绑定到一个存放 `q` 的新 `NotQuery` 对象上。
`Query` 程序实现类	
`Query_base`	查询类的抽象基类
`WordQuery`	`Query_base` 的派生类，用于查找一个给定的单词
`NotQuery`	`Query_base` 的派生类，用于查找一个给定的单词
`BinaryQuery`	`Query_base` 的派生类，查询结果是 `Query` 运算对象没有出现的行的集合
`OrQuery`	`Query_base` 的派生类，返回它的两个运算对象分别出现的行的并集
`AndQuery`	`Query_base` 的派生类，返回它的两个运算对象分别出现的行的交集

第十六章 模板和泛型编程

- 面向对象编程和泛型编程都能处理在编写程序时不知道类型的情况。
    - OOP能处理类型在程序运行之前都未知的情况；
    - 泛型编程中，在编译时就可以获知类型。

定义模板

- **模板**：模板是泛型编程的基础。一个模板就是一个创建类或函数的蓝图或者说公式。

函数模板

- `template <typename T> int compare(const T &v1, const T &v2){}`
- 模板定义以关键字 `template` 开始，后接**模板形参表**，模板形参表是用**尖括号** `<>` 括住的一个或多个**模板形参**的列表，用逗号分隔，**不能为空**。
- 使用模板时，我们显式或隐式地指定模板实参，将其绑定到模板参数上。

- 模板类型参数：类型参数前必须使用关键字`class`或者`typename`，这两个关键字含义相同，可以互换使用。旧的程序只能使用`class`。
- 非类型模板参数：表示一个值而非一个类型。实参必须是常量表达式。`template <class T, size_t N> void array_init(T (&parm)[N]){}`
- 内联函数模板：`template <typename T> inline T min(const T&, const T&);`
- 模板程序应该尽量减少对实参类型的要求。
- 函数模板和类模板成员函数的定义通常放在头文件中。

类模板

- 类模板用于生成类的蓝图。
- 不同于函数模板，编译器不能推断模板参数类型。
- **定义类模板**：
  - `template <class Type> class Queue {};`
- 实例化类模板：提供显式模板实参列表，来实例化出特定的类。
- 一个类模板中所有的实例都形成一个独立的类。
- **模板形参作用域**：模板形参的名字可以在声明为模板形参之后直到模板声明或定义的末尾处使用。
- 类模板的成员函数：
  - `template <typename T> ret-type Blob::member-name(parm-list)`
- 默认情况下，对于一个实例化了的类模板，其成员只有在使用时才被实例化。
- 新标准允许模板将自己的类型参数成为友元。`template <typename T> class Bar{friend T;};`。
- 模板类型别名：因为模板不是一个类型，因此无法定义一个`typedef`引用一个模板，但是新标准允许我们为类模板定义一个类型别名：`template<typename T> using twin = pair<T, T>;`

模板参数

- 模板参数与作用域：一个模板参数名的可用范围是在声明之后，至模板声明或定义结束前。
- 一个特定文件所需要的所有模板的声明通常一起放置在文件开始位置。
- 当我们希望通知编译器一个名字表示类型时，必须使用关键字`typename`，而不能使用`class`。
- 默认模板实参：`template <class T = int> class Numbers{}`

成员模板

- 成员模板（member template）：本身是模板的函数成员。
  - 普通（非模板）类的成员模板。
  - 类模板的成员模板。

控制实例化

- 动机：在多个文件中实例化相同模板的额外开销可能非常严重。
- 显式实例化：
  - `extern template declaration; // 实例化声明`
  - `template declaration; // 实例化定义`

效率与灵活性

模板实参推断

- 对函数模板，编译器利用调用中的函数实参来确定其模板参数，这个过程叫**模板实参推断**。

类型转换与模板类型参数

- 能够自动转换类型的只有：
  - 和其他函数一样，顶层`const`会被忽略。
  - 数组实参或函数实参转换为指针。

函数模板显式实参

- 某些情况下，编译器无法推断出模板实参的类型。
- 定义：`template <typename T1, typename T2, typename T3> T1 sum(T2, T3);`
- 使用函数显式实参调用：`auto val3 = sum<long long>(i, lng); // T1是显式指定，T2和T3都是从函数实参类型推断而来`
- **注意**：正常类型转换可以应用于显式指定的实参。

尾置返回类型与类型转换

- 使用场景：并不清楚返回结果的准确类型，但知道所需类型是和参数相关的。
- `template <typename It> auto fcn(It beg, It end) -> decltype(*beg)`
- 尾置返回允许我们在参数列表之后声明返回类型。

标准库的**类型转换**模板：

- 定义在头文件`type_traits`中。

对 `Mod<T>`，其中 `Mod` 是：	若 `T` 是：	则 `Mod<T>::type` 是：
`remove_reference`	`X&` 或 `X&&`	`X`
	否则	`T`
`add_const`	`X&` 或 `const X` 或函数	`T`
	否则	`const T`
`add_lvalue_reference`	`X&`	`T`
	`X&&`	`X&`
	否则	`T&`
`add_rvalue_reference`	`X&` 或 `X&&`	`T`
	否则	`T&&`
`remove_pointer`	`X*`	`X`
	否则	`T`
`add_pointer`	`X&` 或 `X&&`	`X*`
	否则	`T*`
`make_signed`	`unsigned X`	`X`
	否则	`T`
`make_unsigned`	带符号类型	`unsigned X`
	否则	`T`
`remove_extent`	`X[n]`	`X`
	否则	`T`
`remove_all_extents`	`X[n1][n2]...`	`X`
	否则	`T`

函数指针和实参推断

- 当使用一个函数模板初始化一个函数指针或为一个函数指针赋值时，编译器使用指针的类型来推断模板实参。

模板实参推断和引用

- 从左值引用函数推断类型：若形如 `T&`，则只能传递给它一个左值。但如果是 `const T&`，则可以接受一个右值。
- 从右值引用函数推断类型：若形如 `T&&`，则只能传递给它一个右值。
- 引用折叠和右值引用参数：
    - 规则1：当我们将一个左值传递给函数的右值引用参数，且右值引用指向模板类型参数时（如 `T&&`），编译器会推断模板类型参数为实参的左值引用类型。

- 规则2：如果我们间接创造一个引用的引用，则这些引用形成了**折叠**。折叠引用只能应用在间接创造的引用的引用，如类型别名或模板参数。对于一个给定类型`X`：
  - `X& &`、`X& &&`和`X&& &`都折叠成类型`X&`。
  - 类型`X&& &&`折叠成`X&&`。
- 上面两个例外规则导致两个重要结果：
  - 1.如果一个函数参数是一个指向模板类型参数的右值引用（如`T&&`），则它可以被绑定到一个左值上；
  - 2.如果实参是一个左值，则推断出的模板实参类型将是一个左值引用，且函数参数将被实例化为一个左值引用参数（`T&`）。

理解std::move

- 标准库`move`函数是使用右值引用的模板的一个很好的例子。
- 从一个左值`static_cast`到一个右值引用是允许的。

```
template <typename T>
typename remove_reference<T>::type&& move(T&& t)
{
  return static_cast<typename remove_reference<T>::type&&>(t);
}
```

转发

- 使用一个名为`forward`的新标准库设施来传递参数，它能够保持原始实参的类型。
- 定义在头文件`utility`中。
- 必须通过显式模板实参来调用。
- `forward`返回显式实参类型的右值引用。即，`forward<T>`的返回类型是`T&&`。

重载与模板

- 多个可行模板：当有多个重载模板对一个调用提供同样好的匹配时，会选择最特例化的版本。
- 非模板和模板重载：对于一个调用，如果一个非函数模板与一个函数模板提供同样好的匹配，则选择非模板版本。

可变参数模板

可变参数模板就是一个接受可变数目参数的模板函数或模板类。

- 可变数目的参数被称为参数包。
  - 模板参数包：标识另个或多个模板参数。
  - 函数参数包：标识另个或者多个函数参数。
- 用一个省略号来指出一个模板参数或函数参数，表示一个包。
- `template <typename T, typename... Args>`，`Args`第一个模板参数包。
- `void foo(const T &t, const Args& ... rest);`，`rest`是一个函数参数包。
- `sizeof...`运算符，返回参数的数目。

编写可变参数函数模板

- 可变参数函数通常是递归的：第一步调用处理包中的第一个实参，然后用剩余实参调用自身。

包扩展

- 对于一个参数包，除了获取它的大小，唯一能做的事情就是**扩展**（expand）。
- 扩展一个包时，还要提供用于每个扩展元素的**模式**（pattern）。

转发参数包

- 新标准下可以组合使用可变参数模板和`forward`机制，实现将实参不变地传递给其他函数。

模板特例化（Specializations）

- 定义函数模板特例化：关键字`template`后面跟一个空尖括号对（`<>`）。
- 特例化的本质是实例化一个模板，而不是重载它。特例化不影响函数匹配。
- 模板及其特例化版本应该声明在同一个头文件中。所有同名模板的声明应该放在前面，然后是特例化版本。
- 我们可以部分特例化类模板，但不能部分特例化函数模板。

第十七章 标准库特殊设施

tuple类型

- `tuple`是类似`pair`的模板，每个成员类型都可以不同，但`tuple`可以有任意数量的成员。
- 但每个确定的`tuple`类型的成员数目是固定的。
- 我们可以将`tuple`看做一个“快速而随意”的数据结构。

tuple支持的操作：

操作	解释
`tuple<T1, T2, ..., Tn> t;`	`t`是一个`tuple`，成员数为`n`，第`i`个成员的类型是`Ti`所有成员都进行值初始化。
`tuple<T1, T2, ..., Tn> t(v1, v2, ..., vn);`	每个成员用对应的初始值`vi`进行初始化。此构造函数是`explicit`的。
`make_tuple(v1, v2, ..., vn)`	返回一个用给定初始值初始化的`tuple`。`tuple`的类型从初始值的类型**推断**。
`t1 == t2`	当两个`tuple`具有相同数量的成员且成员对应相等时，两个`tuple`相等。
`t1 relop t2`	`tuple`的关系运算使用**字典序**。两个`tuple`必须具有相同数量的成员。
`get<i>(t)`	返回`t`的第`i`个数据成员的引用：如果`t`是一个左值，结果是一个左值引用；否则，结果是一个右值引用。`tuple`的所有成员都是`public`的。
`tuple_size<tupleType>::value`	一个类模板，可以通过一个`tuple`类型来初始化。它有一个名为`value`的`public constexpr static`数据成员，类型为`size_t`，表示给定`tuple`类型中成员的数量。
`tuple_element<i, tupleType>::type`	一个类模板，可以通过一个整型常量和一个`tuple`类型来初始化。它有一个名为`type`的`public`成员，表示给定`tuple`类型中指定成员的类型。

定义和初始化tuple

定义和初始化示例：

- `tuple<size_t, size_t, size_t> threeD;`
- `tuple<size_t, size_t, size_t> threeD{1,2,3};`
- `auto item = make_tuple("0-999-78345-X", 3, 2.00);`

访问tuple成员：

- `auto book = get<0>(item);`
- `get<2>(item) *= 0.8;`

使用tuple返回多个值

- `tuple`最常见的用途是从一个函数返回多个值。

bitset类型

- 处理二进制位的有序集；
- `bitset`也是类模板，但尖括号中输入的是`bitset`的长度而不是元素类型，因为元素类型是固定的，都是一个二进制位。

初始化`bitset`的方法：

操作	解释
`bitset<n> b;`	`b` 有 `n` 位；每一位均是0.此构造函数是一个 `constexpr`。
`bitset<n> b(u);`	`b` 是 `unsigned long long` 值 `u` 的低 `n` 位的拷贝。如果 `n` 大于 `unsigned long long` 的大小，则 `b` 中超出 `unsigned long long` 的高位被置为0。此构造函数是一个 `constexpr`。
`bitset<n> b(s, pos, m, zero, one);`	`b` 是 `string s` 从位置 `pos` 开始 `m` 个字符的拷贝。 `s` 只能包含字符 `zero` 或 `one`：如果 `s` 包含任何其他字符，构造函数会抛出 `invalid_argument` 异常。字符在 `b` 中分别保存为 `zero` 和 `one`。 `pos` 默认为0， `m` 默认为 `string::npos`， `zero` 默认为'0'， `one` 默认为'1'。
`bitset<n> b(cp, pos, m, zero, one);`	和上一个构造函数相同，但从 `cp` 指向的字符数组中拷贝字符。如果未提供 `m`，则 `cp` 必须指向一个 `C` 风格字符串。如果提供了 `m`，则从 `cp` 开始必须至少有 `m` 个 `zero` 或 `one` 字符。

初始化案例；

- `bitset<13> bitvec1(0xbeef);`
- `bitset<32> bitvec4("1100");`

`bitset` 操作：

操作	解释
`b.any()`	`b`中是否存在1。
`b.all()`	`b`中都是1。
`b.none()`	`b`中是否没有1。
`b.count()`	`b`中1的个数。
`b.size()`	
`b.test(pos)`	`pos`下标是否是1
`b.set(pos)`	`pos`置1
`b.set()`	所有都置1
`b.reset(pos)`	将位置`pos`处的位复位
`b.reset()`	将`b`中所有位复位
`b.flip(pos)`	将位置`pos`处的位取反
`b.flip()`	将`b`中所有位取反
`b[pos]`	访问`b`中位置`pos`处的位；如果`b`是`const`的，则当该位置位时，返回`true`；否则返回`false`。
`b.to_ulong()`	返回一个`unsigned long`值，其位模式和`b`相同。如果`b`中位模式不能放入指定的结果类型，则抛出一个`overflow_error`异常。
`b.to_ullong()`	类似上面，返回一个`unsigned long long`值。
`b.to_string(zero, one)`	返回一个`string`，表示`b`中位模式。`zero`和`one`默认为0和1。
`os << b`	将`b`中二进制位打印为字符`1`或`0`，打印到流`os`。
`is >> b`	从`is`读取字符存入`b`。当下一个字符不是1或0时，或是已经读入`b.size()`个位时，读取过程停止。

正则表达式

- 正则表达式（requlat expression）是一种描述字符序列的方法，是一种很强大的工具。

正则表达式库组件：

组件	解释
`regex`	表示一个正则表达式的类
`regex_match`	将一个字符序列与一个正则表达式匹配
`regex_search`	寻找第一个与正则表达式匹配的子序列
`regex_replace`	使用给定格式替换一个正则表达式
`sregex_iterator`	迭代器适配器，调用`regex_searcg`来遍历一个`string`中所有匹配的子串
`smatch`	容器类，保存在`string`中搜索的结果
`ssub_match`	`string`中匹配的子表达式的结果

`regex_match`和`regex_search`的参数：

操作	解释
`(seq, m, r, mft)`	在字符序列`seq`中查找`regex`对象`r`中的正则表达式。`seq`可以是一个`string`、标识范围的一对迭代器、一个指向空字符结尾的字符数组的指针。
`(seq, r, mft)`	`m`是一个`match`对象，用来保存匹配结果的相关细节。`m`和`seq`必须具有兼容的类型。`mft`是一个可选的`regex_constants::match_flag_type`值。

- 这些操作会返回`bool`值，指出是否找到匹配。

使用正则表达式库

- `regex`使用的正则表达式语言是`ECMAScript`，模式`[[::alpha::]]`匹配任意字母。
- 由于反斜线是C++中的特殊字符，在模式中每次出现`\`的地方，必须用一个额外的反斜线`\\`告知C++我们需要一个反斜线字符。
- 简单案例：
  - `string pattern("[^c]ei"); pattern = "[[:alpha:]]*" + pattern + "[[:alpha:]]*"` 查找不在字符c之后的字符串ei
  - `regex r(pattern);` 构造一个用于查找模式的regex
  - `smatch results;` 定义一个对象保存搜索结果
  - `string test_str = "receipt freind theif receive";`
  - `if (regex_search(test_str, results, r)) cout << results.str() << endl;` 如有匹配子串，打印匹配的单词。

`regex`（和`wregex`）选项：

操作	解释
`regex r(re)` `regex r(re, f)`	`re` 表示一个正则表达式，它可以是一个 `string`、一对表示字符范围的迭代器、一个指向空字符结尾的字符数组的指针、一个字符指针和一个计数器、一个花括号包围的字符列表。`f` 是指出对象如何处理的标志。`f` 通过下面列出来的值来设置。如果未指定 `f`，其默认值为 `ECMAScript`。
`r1 = re`	将 `r1` 中的正则表达式替换Wie `re`。`re` 表示一个正则表达式，它可以是另一个 `regex` 对象、一个 `string`、一个指向空字符结尾的字符数组的指针或是一个花括号包围的字符列表。
`r1.assign(re, f)`	和使用赋值运算符（=）的效果相同：可选的标志 `f` 也和 `regex` 的构造函数中对应的参数含义相同。
`r.mark_count()`	`r` 中子表达式的数目
`r.flags()`	返回 `r` 的标志集

定义 `regex` 时指定的标志：

操作	解释
`icase`	在匹配过程中忽略大小写
`nosubs`	不保存匹配的子表达式
`optimize`	执行速度优先于构造速度
`ECMAScript`	使用 `ECMA-262` 指定的语法
`basic`	使用 `POSIX` 基本的正则表达式语法
`extended`	使用 `POSIX` 扩展的正则表达式语法
`awk`	使用 `POSIX` 版本的 `awk` 语言的语法
`grep`	使用 `POSIX` 版本的 `grep` 的语法
`egrep`	使用 `POSIX` 版本的 `egrep` 的语法

- 可以将正则表达式本身看做是一种简单程序语言设计的程序。在运行时，当一个 `regex` 对象被初始化或被赋予新模式时，才被“编译”。
- 如果编写的正则表达式存在错误，会在运行时抛出一个 `regex_error` 的异常。
- 避免创建不必要的正则表达式。构建一个 `regex` 对象可能比较耗时。

匹配与regex迭代器类型

`sregex_iterator` 操作（用来获得所有匹配）：

操作	解释
`sregex_iterator it(b, e, r);`	一个`sregex_iterator`，遍历迭代器`b`和`e`表示的`string`。它调用`sregex_search(b, e, r)`将`it`定位到输入中第一个匹配的位置。
`sregex_iterator end;`	`sregex_iterator`的尾后迭代器
`*it`, `it->`	根据最后一个调用`regex_search`的结果，返回一个`smatch`对象的引用或一个指向`smatch`对象的指针。
`++it`, `it++`	从输入序列当前匹配位置开始调用`regex_search`。前置版本返回递增后迭代器；后置版本返回旧值。
`it1 == it2`	如果两个`sregex_iterator`都是尾后迭代器，则它们相等。两个非尾后迭代器是从相同的输入序列和`regex`对象构造，则它们相等。

示例：

```
// 将字符串file中所有匹配模式r的子串输出
for (sregex_iterator it(file.begin(), file.end(), r), end_it; it != end_it;
++it){
    cout << it ->str() << endl;
}
```

`smatch`操作：

操作	解释
`m.ready()`	如果已经通过调用`regex_search`或`regex_match`设置了`m`，则返回`true`；否则返回`false`。如果`ready`返回`false`，则对`m`进行操作是未定义的。
`m.size()`	如果匹配失败，则返回0，；否则返回最近一次匹配的正则表达式中子表达式的数目。
`m.empty()`	等价于`m.size() == 0`
`m.prefix()`	一个`ssub_match`对象，标识当前匹配之前的序列
`m.suffix()`	一个`ssub_match`对象，标识当前匹配之后的部分
`m.format(...)`	
`m.length(n)`	第`n`个匹配的子表达式的大小
`m.position(n)`	第`n`个子表达式距离序列开始的长度
`m.str(n)`	第`n`个子表达式匹配的`string`
`m[n]`	对应第`n`个子表达式的`ssub_match`对象
`m.begin()`, `m.end()`	表示`m`中`ssub_match`元素范围的迭代器。
`m.cbegin()`, `m.cend()`	常量迭代器

使用子表达式

- 正则表达式语法通常用括号表示子表达式。
- 子表达式的索引从1开始。
- 在`fmt`中用`$`后跟子表达式的索引号来标识一个特定的子表达式。

示例：

```
if (regex_search(filename, results, r))
    cout << results.str(1) << endl;  // .str(1)获取第一个子表达式匹配结果
```

`ssub_match`子匹配操作：

操作	解释
`matched`	一个 `public bool` 数据成员，指出 `ssub_match` 是否匹配了
`first`，`second`	`public` 数据成员，指向匹配序列首元素和尾后位置的迭代器。如果未匹配，则 `first` 和 `second` 是相等的。
`length()`	匹配的大小，如果 `matched` 为 `false`，则返回0。
`str()`	返回一个包含输入中匹配部分的 `string`。如果 `matched` 为 `false`，则返回空 `string`。
`s = ssub`	将 `ssub_match` 对象 `ssub` 转化为 `string` 对象 `s`。等价于 `s=ssub.str()`，转换运算符不是 `explicit` 的。

使用regex_replace

正则表达式替换操作：

操作	解释
`m.format(dest, fmt, mft)`，`m.format(fmt, mft)`	使用格式字符串 `fmt` 生成格式化输出，匹配在 `m` 中，可选的 `match_flag_type` 标志在 `mft` 中。第一个版本写入迭代器 `dest` 指向的目的为止，并接受 `fmt` 参数，可以是一个 `string`，也可以是一个指向空字符结尾的字符数组的指针。`mft` 的默认值是 `format_default`。
`rege_replace(dest, seq, r, fmt, mft)`，`regex_replace(seq, r, fmt, mft)`	遍历 `seq`，用 `regex_search` 查找与 `regex` 对象 `r` 相匹配的子串，使用格式字符串 `fmt` 和可选的 `match_flag_type` 标志来生成输出。`mft` 的默认值是 `match_default`

示例：

```
string phone = "(\\()?(\\d{3})(\\))?([-. ])?(\\d{3})([-. ]?)(\\d{4})"
string fmt = "$2.$5.$7";  // 将号码格式改为ddd.ddd.dddd
regex r(phone);  // 用来寻找模式的regex对象
string number = "(908) 555-1800";
cout << regex_replace(number, r, fmt) << endl;
```

匹配标志：

操作	解释
`match_default`	等价于`format_default`
`match_not_bol`	不将首字符作为行首处理
`match_not_eol`	不将尾字符作为行尾处理
`match_not_bow`	不将首字符作为单词首处理
`match_not_eow`	不将尾字符作为单词尾处理
`match_any`	如果存在多于一个匹配，则可以返回任意一个匹配
`match_not_null`	不匹配任何空序列
`match_continuous`	匹配必须从输入的首字符开始
`match_prev_avail`	输入序列包含第一个匹配之前的内容
`format_default`	用`ECMAScript`规则替换字符串
`format_sed`	用`POSIX sed`规则替换字符串
`format_no_copy`	不输出输入序列中未匹配的部分
`format_first_only`	只替换子表达式的第一次出现

随机数

- 新标准之前，C和C++都依赖一个简单的C库函数`rand`来生成随机数，且只符合均匀分布。
- 新标准：**随机数引擎** + **随机数分布类**， 定义在`random`头文件中。
- C++程序应该使用`default_random_engine`类和恰当的分布类对象。

随机数引擎和分布

随机数引擎操作

操作	解释
`Engine e;`	默认构造函数；使用该引擎类型默认的种子
`Engine e(s);`	使用整型值`s`作为种子
`e.seed(s)`	使用种子`s`重置引擎的状态
`e.min()`，`e.max()`	此引擎可生成的最小值和最大值
`Engine::result_type`	此引擎生成的`unsigned`整型类型
`e.discard(u)`	将引擎推进`u`步；`u`的类型为`unsigned long long`

示例：

```
// 初始化分布类型
uniform_int_distribution<unsigned> u(0, 9);
// 初始化引擎
default_random_engine e;
// 随机生成0-9的无符号整数
cout << u(e) << endl;
```

设置随机数发生器种子：

- 种子就是一个数值，引擎可以利用它从序列中一个新位置重新开始生成随机数。
- 种子可以使用系统函数`time(0)`。

其他随机数分布

分布类型的操作：

操作	解释
`Dist d;`	默认够赞函数；使`d`准备好被使用。其他构造函数依赖于`Dist`的类型；分布类型的构造函数是`explicit`的。
`d(e)`	用相同的`e`连续调用`d`的话，会根据`d`的分布式类型生成一个随机数序列；`e`是一个随机数引擎对象。
`d.min()`,`d.max()`	返回`d(e)`能生成的最小值和最大值。
`d.reset()`	重建`d`的状态，是的随后对`d`的使用不依赖于`d`已经生成的值。

IO库再探

格式化输入与输出

- 使用操纵符改变格式状态。
- 控制布尔值的格式：`cout << boolalpha << true << endl;`
- 指定整型的进制：`cout << dec << 20 << endl;`

定义在`iostream`中的操纵符：

操纵符	解释
`boolalpha`	将`true`和`false`输出为字符串
`* noboolalpha`	将`true`和`false`输出为1,0
`showbase`	对整型值输出表示进制的前缀
`* noshowbase`	不生成表示进制的前缀
`showpoint`	对浮点值总是显示小数点
`* noshowpoint`	只有当浮点值包含小数部分时才显示小数点
`showpos`	对非负数显示`+`
`* noshowpos`	对非负数不显示`+`
`uppercase`	在十六进制中打印`0X`，在科学计数法中打印`E`
`* nouppercase`	在十六进制中打印`0x`，在科学计数法中打印`e`
`* dec`	整型值显示为十进制
`hex`	整型值显示为十六进制
`oct`	整型值显示为八进制
`left`	在值的右侧添加填充字符
`right`	在值的左侧添加填充字符
`internal`	在符号和值之间添加填充字符
`fixed`	浮点值显示为定点十进制
`scientific`	浮点值显示为科学计数法
`hexfloat`	浮点值显示为十六进制（C++11）
`defaultfloat`	充值浮点数格式为十进制（C++11）
`unitbuf`	每次输出操作后都刷新缓冲区

1| `* nounitbuf` | 恢复正常的缓冲区刷新模式 | | `* skipws` | 输入运算符跳过空白符 | | `noskipws` | 输入运算符不跳过空白符 | | `flush` | 刷新`ostream`缓冲区 | | `ends` | 插入空字符，然后刷新`ostream`缓冲区 | | `endl` | 插入换行，然后刷新`ostream`缓冲区 |

其中`*`表示默认的流状态。

未格式化的输入/输出操作

单字节低层IO操作：

操作	解释
`is.get(ch)`	从 `istream is` 读取下一个字节存入字符 `cn` 中。返回 `is`。
`os.put(ch)`	将字符 `ch` 输出到 `ostream os`。返回 `os`。
`is.get()`	将 `is` 的下一个字节作为 `int` 返回
`is.putback(ch)`	将字符 `ch` 放回 `is`。返回 `is`。
`is.unget()`	将 `is` 向后移动一个字节。返回 `is`。
`is.peek()`	将下一个字节作为 `int` 返回，但不从流中删除它。

多字节低层IO操作：

操作	解释
`is.get(sink, size, delim)`	从 `is` 中读取最多 `size` 个字节，并保存在字符数组中，字符数组的起始地址由 `sink` 给出。读取过程直到遇到字符 `delim` 或读取了 `size` 个字节或遇到文件尾时停止。如果遇到了 `delim`，则将其留在输入流中，不读取出来存入 `sink`。
`is.getline(sink, size, delim)`	与接收三个参数的 `get` 版本类似，但会读取并丢弃 `delim`。
`is.read(sink, size)`	读取最多 `size` 个字节，存入字符数组 `sink` 中。返回 `is`。
`is.gcount()`	返回上一个未格式化读取从 `is` 读取的字节数
`os.write(source, size)`	将字符数组 `source` 中的 `size` 个字节写入 `os`。返回 `os`。
`is.ignore(size, delim)`	读取并忽略最多 `size` 个字符，包括 `delim`。与其他未格式化函数不同，`ignore` 有默认参数：`size` 默认值是1，`delim` 的默认值为文件尾。

- 注意：一般情况下，主张使用标准库提供的高层抽象，低层函数容易出错。

流随机访问

- 只适用于 `fstream` 和 `sstream`。
- 通过将标记 `seek` 到一个给定位置来重定位它。
- `tell` 告诉我们标记的当前位置。

操作	解释
`tellg()`，`tellp`	返回一个输入流中（`tellg`）或输出流中（`tellp`）标记的当前位置。
`seekg(pos)`，`seekp(pos)`	在一个输入流或输出流中将标记重定位到给定的绝对地址。`pos`通常是一个当前`teelg`或`tellp`返回的值。
`seekp(off, from)`，`seekg(off, from)`	在一个输入流或输出流中将标记定位到`from`之前或之后`off`个字符，`from`可以是下列值之一：`beg`，偏移量相对于流开始位置；`cur`，偏移量相对于流当前位置；`end`，偏移量相对于流结尾位置。

第十八章 用于大型程序的工具

大规模应用程序的特殊要求包括：

- 在独立开发的子系统之间协同处理错误的能力。
- 使用各种库进行协同开发的能力。
- 对比较复杂的应用概念建模的能力。

异常处理

异常处理（exception handling）机制允许程序中独立开发的部分能够在运行时就出现的问题进行通信并作出相应的处理。

抛出异常

在C++语言中，我们通过**抛出**（throwing）一条表达式来**引发**（raised）一个异常。异常类型和当前的调用链决定了哪段**处理代码**（handler）将用来处理该异常。

程序的控制权从`throw`转移到`catch`模块。

栈展开：当`throw`出现在一个`try`语句块时，检查该`try`语句块相关的`catch`字句，若有匹配则处理；若无匹配，则继续检查外层的`try`匹配的`catch`。

若一个异常没有被捕获，则它将终止当前的程序。

对象销毁：

- 块退出后，它的局部对象将被销毁。
- 若异常发生在构造函数中，即使某个对象只构造了一部分，也要确保已构造的成员正确地被销毁。
- 将资源释放放在类的析构函数中，以保证资源能被正确释放。析构函数本身不会引发异常。

捕获异常

若无需访问抛出的异常对象，则可以忽略捕获形参的名字。

通常，若`catch`接受的异常与某个继承体系有关，则最好将该`catch`的参数定义成引用类型。

搜索`catch`未必是最佳匹配，而是第一个匹配，因此，越细化的`catch`越应该放在`catch`列表前段。

重新抛出：`catch`代码执行一条`throw;`将异常传递给另一个`catch`语句。

捕获所有异常：`catch(...)`

构造函数

处理构造函数初始值异常的唯一方法是将构造函数协程函数`try`语句块。

示例：

```
template <typename T>
Blob<T>::Blob(std::initializer_list<T> il) try:
    data(std::make_shared<std::vector<T> >(il){
        /*函数体*/
    } catch(const std::bad_alloc &e){ handle_out_of_memory(e); }
```

noexcept异常说明

使用`noexcept`说明指定某个函数不会抛出异常。

示例：

```
void recoup(int) noexcept; //C++11
coid recoup(int) throw(); //老版本
```

异常类层次

标准exception层次：

- exception
  - bad_cast
  - bad_alloc
  - runtime_error
    - overflow_error
    - underflow_error
    - range_error
  - logic_error
    - domain_error
    - invalid_argument
    - out_of_range
    - length_error

自定义异常类：

示例：

```
class out_of_stock: public std::runtime_error {
    explicit out_of_stock(const std::string &s):
    std::runtime_error(s){ }
};
```

命名空间

多个库将名字放置在全局命名空间中将引发**命名空间污染**（namespace pollution）。**命名空间**（namespace）分割了全局命名空间，其中每个命名空间是一个作用域。

命名空间定义

命名空间的定义包含两部分：1.关键字`namespace`；2.命名空间名称。后面是一系列由花括号括起来的声明和定义。命名空间作用域后面无需分号。

示例：

```
namespace cplusplus_primer{

}
```

每个命名空间都是一个**作用域**。定义在某个命名空间内的名字可以被该命名空间内的其他成员直接访问，也可以被这些成员内嵌套作用域中的任何单位访问。位于该命名空间之外的代码必须明确指出所用的名字是属于哪个命名空间的。

命名空间可以是**不连续**的。这点不同于其他作用域，意味着同一命名空间可以在多处出现。

内联命名空间（C++11）：

无需使用该命名空间的前缀，通过外层命名空间就可以直接访问。

示例：

```
namespace cplusplus_primer{
    inline namespace FifthEd{
        // 表示本书第5版代码
        class Query_base {};
    }
}

cplusplus_primer::Query_base qb;
```

未命名的命名空间：

指关键字`namespace`后面紧跟花括号的用法。未命名的命名空间中定义的变量拥有静态的声明周期：在第一次使用前创建，直到程序结束才销毁。不能跨越多个文件。

使用命名空间成员

像`namespace_name::member_name`这样使用命名空间的成员非常繁琐。

命名空间的别名：

```
namespace primer = cplusplus_primer;
```

using声明（using declaration）：

一条`using`声明语句一次只引入命名空间的一个成员。

```
using std::string;

string s = "hello";
```

using指示（using directive）：

使得某个特定的命名空间中所有的名字都可见。

```
using namespace std;

string s = "hello";
```

类、命名空间与作用域

```
namespace A{
    class C1{
        public:
            int f3();
    }
}

A::C1::f3
```

重载与命名空间

`using`声明语句声明的是一个名字，而非特定的函数，也就是包括该函数的所有版本，都被引入到当前作用域中。

多重继承与虚继承

多重继承

类型转换与多个基类

多重继承下的类作用域

- 当一个类拥有多个基类时，有可能出现派生类从两个或更多基类中继承了同名成员的情况。此时，不加前缀限定符直接使用该名字将引发二义性。

虚继承

- 虚继承的目的是令某个类做出声明，承诺愿意共享它的基类。其中，共享的基类子对象成为**虚基类**。在这种机制下，不论虚基类在继承体系中出现了多少次，在派生类中都只包含唯一一个共享的虚基类子对象。
- 虚派生只影响从指定了虚基类的派生类中进一步派生出的类，它不会影响派生类本身。

构造函数与虚继承

- h含有虚基类的对象的构造顺序与一般的顺序稍有**区别**：首先使用提供给最底层派生类构造函数的初始值初始化该对象的虚基类子部分，接下来按照直接基类在派生列表中出现的次序对其进行初始化。
- 虚基类总是先于非虚基类构造，与它们在继承体系中的次序和位置无关。

第十九章 特殊工具与技术

控制内存分配

重载new和delete

- `new`**表达式的工作机理**:

```
string *sp = new string("a value"); //分配并初始化一个string对象
string *arr = new string[10];   // 分配10个默认初始化的string对象
```

- 上述代码实际执行了**三步操作**:
  - `new`表达式调用一个名为`operator new`(或`operator new []`)的标准库函数，它分配一块**足够大的、原始的、未命名的**内存空间以便存储特定类型的对象(或对象的数组)。
  - 编译器运行相应的构造函数以构造这些对象，并为其传入初始值。
  - 对象被分配了空间并构造完成，返回一个指向该对象的指针。
- `delete`**表达式的工作机理**:

```
delete sp;  // 销毁*sp，然后释放sp指向的内存空间
delete [] arr;  // 销毁数组中的元素，然后释放对应的内存空间
```

- 上述代码实际执行了**两步操作**:
  - 对`sp`所指向的对象或者`arr`所指的数组中的元素执行对应的析构函数。
  - 编译器调用名为`operator delete`(或`operator delete[]`)的标准库函数释放内存空间。
- 当自定义了全局的`operator new`函数和`operator delete`函数后，我们就担负起了控制动态内存分配的职责。这两个函数**必须是正确的**。因为它们是程序整个处理过程中至关重要的一部分。
- 标准库定义了`operator new`函数和`operator delete`函数的8个重载版本:

```
// 这些版本可能抛出异常
void *operator new(size_t); // 分配一个对象
void *operator new[](size_t);   // 分配一个数组
void *operator delete(void*) noexcept;  // 释放一个对象
void *operator delete[](void*) noexcept;    // 释放一个数组

// 这些版本承诺不会抛出异常
void *operator new(size_t, nothrow_t&) noexcept;
void *operator new[](size_t, nothrow_t&) noexcept;
void *operator delete(void*, nothrow_t&) noexcept;
void *operator delete[](void*, nothrow_t&) noexcept;
```

- 应用程序可以自定义上面函数版本中的任意一个，前提是自定义的版本必须位于**全局作用域**或者**类作用域**中。
- **注意：** 提供新的`operator new`函数和`operator delete`函数的目的在于改变内存分配的方式，但是不管怎样，都不能改变`new`运算符和`delete`运算符的基本含义。
- 使用从C语言继承的函数`malloc`和`free`函数能实现以某种方式执行分配内存和释放内存的操作:

```
#include <cstdlib>

void *operator new(size_t size) {
    if(void *mem = malloc(size))
        return mme;
    else
        throw bad_alloc();
}

void operator delete(void *mem) noexcept {
    free(mem);
}
```

定位new表达式

- 应该使用new的定位`new(placement new)`形式传递一个地址，定位`new`的形式如下：

```
new (place_address) type
new (place_address) type (initializers)
new (place_address) type [size]
new (place_address) type [size] {braced initializer list}
// place_address必须是一个指针，同时在initializers中提供一个(可能为空的)以逗号分隔的初始值列表，该初始值列表将用于构造新分配的对象。
```

- 当只传入一个指针类型的实参时，定位`new`表达式构造对象但是不分配内存。
- 调用析构函数会销毁对象，但是不会释放内存。

```
string *sp = new string("a value"); // 分配并初始化一个string对象
sp->~string();
```

运行时类型识别

- 运行时类型识别`(run-time type identification, RTTI)`的功能由两个运算符实现：
    - `typeid`运算符， 用于返回表达式的类型。
    - `dynamic_cast`运算符，用于将基类的指针或引用安全地转换曾派生类的指针或引用。
- 使用`RTTI`必须要加倍小心。在可能的情况下，最好定义虚函数而非直接接管类型管理的重任。

dynamic_cast运算符

- dynamic_cast运算符的使用形式如下：

```
dynamic_cast<type*>(e)  // e必须是一个有效的指针
dynamic_cast<type&>(e)  // e必须是一个左值
dynamic_cast<type&&>(e) // e不能是左值
// 以上，type类型必须时一个类类型，并且通常情况下该类型应该含有虚函数。
// e的类型必须符合三个条件中的任意一个，它们是：
// 1. e的类型是目标type的公有派生类；
// 2. e的类型是目标type的共有基类；
// 3. e的类型就是目标type的类型；

// 指针类型的dynamic_cast
// 假设Base类至少含有一个虚函数，Derived是Base的共有派生类。
if (Derived *dp = dynamic_cast<Derived*>(bp)) {
```

```
    // 使用dp指向的Derived对象
} else {    // bp指向一个Base对象
    // 使用dp指向的Base对象
}

// 引用类型的dynamic_cast
void f(const Base &b) {
    try {
        const Derived &d = dynamic_cast<const Derived&>(b);
        // 使用b引用的Derived对象
    } catch (bad_cast) {
        // 处理类型转换失败的情况
    }
}
```

- 可以对一个空指针执行`dynamic_cast`，结果是所需类型的空指针。

typeid运算符

- `typeid运算符(typeid operator)`，它允许程序向表达式提问：**你的对象是什么类型?**
- `typeid`表达式的形式是`typeid(e)`，其中`e`可以是任意表达式或类型的名字，它操作的结果是一个常量对象的引用。它可以作用于任意类型的表达式。
- 通常情况下，使用typeid比较两条表达式的类型是否相同，或者比较一条表达式的类型是否与指定类型相同：

```
Derived *dp = new Derived;
Base *bp = dp;

if (typeid(*bp) == typeid(*dp)) {
    // bp和dp指向同一类型的对象
}

if (typeid(*bp) == typeid(Derived)) {
    // bp实际指向Derived对象
}
```

- 当typeid作用于指针时(而非指针所指向的对象)，返回的结果是该指针的静态编译时类型。

```
// 下面的检查永远是失败的：bp的类型是指向Base的指针
if (typeid(bp) == typeid(Derived)) {
    // 永远不会执行
}
```

使用RTTI

- 用途：为具有继承关系的类实现相等运算符时。对于两个对象来说，如果它们的类型相同并且对应的数据成员取值相同，则说这两个对象是相等的。

```
// 类的层次关系
class Base {
    friend bool operator==(const Base&, const Base&);
public:
    // Base的接口成员
protected:
    virtual bool equal(const Base&) const;
```

```
    // Base的数据成员和其他用于实现的成员
};

class Derived: public Base {
public:
    // Derived的其他接口成员
protected:
    bool equal(const Base&) const;
    // Derived的数据成员和其他用于实现的成员
};

// 类型敏感的相等运算符
bool operator==(const Base &lhs, const Base &rhs) {
    // 如果typeid不相同，返回false；否则虚调用equal
    return typeid(lhs) == typeid(rhs) && lhs.equal(rhs);
}

// 虚equal函数
bool Derived::equal(const Base &rhs) const {
    auto r = dynamic_cast<const Derived&>(rhs);
    // 执行比较两个Derived对象的操作并返回结果
}

// 基类equal函数
bool Base::equal(const Base &rhs) const {
    // 执行比较Base对象的操作
}
```

type_info类

枚举类型

- 枚举类型`(enumeration)`使我们可以将一组整型常量组织在一起。枚举属于字面值常量类型。
- **限定作用域的枚举类型(scoped enumeration)**：首先是关键字`enum class(或enum struct)`，随后是枚举类型名字以及用花括号括起来的以逗号分隔的枚举成员列表，最后是一个分号。

```
enum class open_modes {input, output, append};
```

- 不限定作用域的枚举类型`(unscoped enumeration)`：省略关键字`class(或struct)`，枚举类型的名字是可选的。

```
enum color {red, yellow, green};

enum {floatPrec = 6, doublePrec = 10, double_doublePrec = 10};
```

类成员指针

成员指针：指可以指向类的非静态成员的指针。

数据成员指针

- 和其他指针一样，在声明成员指针时也使用*来表示当前声明的名字是一个指针。与普通指针不同的时，成员指针还必须包含成员所属的类。

```
// pdata可以指向一个常量(非常量)Screen对象的string成员
const string Screen::*pdata;

// C++11
auto pdata = &Screen::contents;
```

- 当我们初始化一个成员指针或为成员指针赋值时，该指针没有指向任何数据。成员指针指定了成员而非该成员所属的对象，只有当解引用成员指针时才提供对象的信息。

```
Screen myScreen, *pScreen = &myScreen;

auto s = myScreen.*pdata;

s = pScreen->*pdata;
```

成员函数指针

- 因为函数调用运算符的优先级较高，所以在声明指向成员函数的指针并使用这些的指针进行函数调用时，括号必不可少：`(C::*p)(parms)`和`(obj.*p)(args)`。

将成员函数用作可调用对象

嵌套类

- 一个类可以定义在另一个类的内部，前者称为嵌套类(nested class)或嵌套类型(nested type)。**嵌套类常用于定义作为实现部分的类。**
- 嵌套类是一个独立的类，与外层类基本没有什么关系。特别是，外层类的对象和嵌套类的对象是相互独立的。
- 嵌套类的名字在外层类作用域中是可见的，在外层类作用域之外不可见。

union：一种节省空间的类

- `联合(union)`是一种特殊的类。一个`union`可以有多个数据成员，但是在任意时刻只有一个数据成员可以有值。**它不能含有引用类型的成员和虚函数。**

```
// Token类型的对象只有一个成员，该成员的类型可能是下列类型中的任意一种
union Token {
    // 默认情况下成员是共有的
    char cval;
    int ival;
    double dval;
};
```

- `匿名union(anonymous union)`是一个未命名的`union`，并且在右花括号和分号之间没有任何声明。

```
union {
    char cval;
    int ival;
    double dval;
};

// 可以直接访问它的成员
cal = 'c';
ival = 42;
```

- **注意：** `匿名union`不能包含受保护的成员或私有成员，也不能定义成员函数。

局部类

- `局部类(local class)`：可以定义在某个函数的内部的类。它的类型只在定义它的作用域内可见。和嵌套类不同，局部类的成员受到严格限制。
- 局部类的所有成员(包括函数在内)都必须完整定义在类的内部。因此，局部类的作用与嵌套类相比相差很远。
- **局部类不能使用函数作用域中的变量。**

```
int a, val;
void foo(int val) {
    static inti si;
    enum loc { a = 1024, b};

    // Bar是foo的局部类
    struct Bar {
        Loc locVal; // 正确：使用一个局部类型名
        int barVal;

        void fooBar(Loc l = a) {    // 正确：默认实参是Loc::a
            barVal = val;   // 错误：val是foo的局部变量
            barVal == ::val;    // 正确：使用一个全局对象
            barVal = si;    // 正确：使用一个静态局部对象
            locVal = b; // 正确：使用一个枚举成员
        }
    };
}
```

固有的不可移植的特性

所谓不可移植的特性是指**因机器而异的特性**，当将含有不可移植特性的程序从一台机器转移到另一台机器上时，通常需要重新编写该程序。

位域

- 类可以将其(非静态)数据成员定义成**位域(bit-field)**，在一个位域中含有一定数量的二进制位。当一个程序需要向其他程序或硬件设备传递二进制数据时，通常会用到位域。
- 位域在内存中的布局是与机器相关的。
- 位域的类型必须是整型或枚举类型。因为带符号位域的行为是由具体实现确定的，通常情况下我们使用无符号类型保存一个位域。

```
typedef unsigned int Bit;
```

```
class File {
    Bit mode: 2;
    Bit modified: 1;
    Bit prot_owner: 3;
    Bit prot_group: 3;
    Bit prot_world: 3;
public:
    enum modes {READ = 01, WRITE = 02, EXECUTE = 03};
    File &open(modes);
    void close();
    void write();
    bool isRead() const;
    void setWrite();
}

// 使用位域
void File::write() {
    modified = 1;
    // ...
}

void File::close() {
    if( modified)
        // ...保存内容
}

File &File::open(File::modes m) {
    mode |= READ;    // 按默认方式设置READ
    // 其他处理
    if(m & WRITE)    // 如果打开了READ和WRITE
        // 按照读/写方式打开文件
    return *this;
}
```

volatile限定符

- 当对象的值可能在程序的控制或检测之外被改变时，应该将该对象声明为`volatile`。关键字`volatile`告诉编译器不应对这样的对象进行优化。
- `const`和`volatile`的一个重要区别是不能使用合成的拷贝/移动构造函数及赋值运算符初始化`volatile`对象或者从`volatile`对象赋值。

链接指示：extern "C"

- `C++`使用`链接指示(linkage directive)`指出任意非`C++`函数所用的语言。
- 要想把`C++`代码和其他语言(包括`C`语言)编写的代码放在一起使用，要求我们必须有权访问该语言的编译器，并且这个编译器与当前的`C++`编译器是兼容的。
- `C++`从C语言继承的标准库函数可以定义为`C`函数，但并非必须：决定使用`C`还是`C++`实现的`C`标准库，是每个`C++`实现的事情。
- 有时需要在C和C++中编译同一个源文件，为了实现这一目的，在编译C++版本的程序时预处理器定义`__cplusplus`。

```
#ifdef __cplusplus
extern "C"
#endif
int strcmp(const char*, const char*);
```