C++ Primer 要点整理

公众号：Jacen 的技术笔记

知乎：Jacenhu

欢迎关注我，共同学习和交流。

第一章 开始

1.1 编写一个简单的 C++程序

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

每个 C++程序都包含一个或多个函数，其中一个必须命名为 main.

1.2 初识输入输出

对象	用途
cin	标准输入
cout	标准输出
cerr	标准错误
clog	输出运行时的一般性消息

1.3 注释简介

两种：

单行注释： //

界定符：/* 和 */

1.4 控制流

while;for;if;

第二章 变量和基本类型

P30-P71

数据类型是程序的基础。C++语言支持广泛的数据类型。

基本内置类型

算术类型

类型	最小尺寸
bool	未定义
char	8 位
w*char*t	16 位
char16_t	16 位
char32_t	32 位
short	16 位
int	16 位
long	32 位
long long	64 位
float	6 位有效数字
double	10 位有效数字
long double	10 位有效数字

类型转换

不要混用符号类型和无符号类型。

变量

变量定义

（1）基本形式：

类型说明符，随后紧跟着一个或者多个变量名组成的列表，其中变量名以逗号分隔，最后以分号结束。

（2）初始值

在 C++中，初始化和赋值是 2 个完全不同的操作。初始化的含义是创建变量的时候赋予一个初始值，而赋值的含义是把对象的当前值擦除，用一个新值来替代。两者区别很小。

（3）列表初始化

用花括号来初始化变量的方式，称为列表初始化。

（4）默认初始化

如果定义变量没有指定初始值，则变量被默认初始化。

::: tip

例外情况：

定义在函数体内部的内置类型变量将不被初始化，其值未定义。

建议初始化每个内置类型的变量。
:::

变量声明和定义的关系

变量声明：规定了变量的类型和名字。

变量定义：除声明之外，还需要申请存储空间。

如果想声明一个变量，而非定义它，需要使用 extern 关键词。

```
extern int i;     // 声明 i 而非定义 i
int j;            // 声明并定义 j
```

::: tip
变量只能被定义一次，但可以被多次声明。
:::

名字的作用域

作用域：C++中大多数作用域都用花括号分隔。

作用域中一旦声明了某个名字，它所嵌套的所有作用域都能访问该名字。同时，允许在内层作用域中重新定义外层作用域中有的名字。

::: warning
如果函数有可能用到某全局变量，则不宜再定义一个同名的局部变量。
:::

复合类型

定义:
复合类型是基于其他类型定义的类型。

引用

引用：为对象起另外一个名字。

::: warning
引用必须被初始化。
引用本身不是对象，所以不能定义引用的引用。
引用要和绑定的对象严格匹配。
引用类型的初始值，必须是一个对象。
:::

指针

指针：本身就是一个对象。允许对指针赋值和拷贝。指针无须在定义的时候赋值。

（1）利用指针访问对象

如果指针指向了一个对象，则允许使用解引用符（*）来访问该对象。

（2）void* 指针

理解复合类型的声明

（1）指向指针的指针

** 表示指向指针的指针

*** 表示指向指针的指针的指针

（2）指向指针的引用

不能定义指向引用的指针。但指针是对象，所以存在对指针的引用。

const 限定符

定义：const 用于定义一个变量，它的值不能被改变。const 对象必须初始化。

::: tip
默认状态下，const 对象仅在文件内有效。当多个文件出现了同名的 const 变量时，等同于在不同文件中分别定义了独立的变量。

如果想让 const 变量在文件间共享，则使用 extern 修饰。
:::

（1）const 的引用

允许为一个常量引用绑定非常量的对象、字面值，甚至是个一般表达式。

一般，引用的类型必须与其所引用对象的类型一致，特殊情况是表达式。

（2）指针和 const

弄清楚类型，可以从右边往左边阅读。

（3）顶层 const

top-level const 表示指针本身是个常量

low-level const 表示指针所指的对象是一个常量。

（4）constexpr 和常量表达式

C++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表达式。

处理类型

类型别名

两种方法用于定义类型别名：

（1）使用关键词 typedef

```
typedef double wages; //wages 是 double 的同义词
typedef wages *p; // p 是 double*的同义词
```

（2）别名声明

```
using SI = Sales_item;  // SI 是 Sales_item 的同义词
```

auto 类型说明符：让编译器通过初始值来推算变量的类型。

decltype 类型指示符：选择并返回操作符的数据类型。只得到类型，不实际计算表达式的值。

自定义数据结构

（1）类

数据结构是把一组相关的数据元素组织起来，然后使用它们的策略和方法。

类一般不定义在函数体内，为了确保各个文件中类的定义一致，类通常被定义在头文件中，而且类所在头文件的名字应该与类的名字一样。

头文件通常包含那些被定义一次的实体。

（2）预处理器

```
#ifndef SALES_DATA_H
#define SALES_DATA_H
#endif
```

一般把预处理变量的名字全部大写。

术语

空指针：值为 0 的指针，空指针合法但是不指向任何对象。nullPtr 是表示空指针的字面值常量。

void*：可以指向任意非常量的指针类型，不能执行解引用操作。

第三章 字符串、向量和数组

P74-P118

string 表示可变长的字符序列，vector 存放的是某种给定类型对象的可变长序列。

命名空间的 using 声明

```
using namespace:name;
```

头文件不应包含 using 声明。

标准库类型 string

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

（1）定义和初始化

```
string s1;
sting s2(s1);
string s3("value");
string s3 = "value";
string s4(n, 'c');
```

（2）string 对象的操作

```
s.empty();      // 判空
s.size();       // 字符个数
s[n];           // s 中第 n 个字符的引用
s1+s2;          // s1 和 s2 连接
<,<=,>,>=       // 比较
```

::: warning

标准局允许把字面值和字符串字面值转换成 string 对象。字面值和 string 是不同的类型。

:::

(3)处理 string 对象中的字符

::: tip
C++程序的头文件应该使用 cname，而不应该使用 name.h 的形式
:::

遍历给定序列中的每个值执行某种操作

```
for (declaration : expression)
    statement
```

标准库类型 vector

标准库 vector 表示对象的集合，其中所有对象的类型都相同。

vector 是一个类模板，而不是类型。

（1）定义和初始化 vector 对象

```
vector<T> v1;
vector<T> v2(v1);
vector<T> v2 = v1;
vector<T> v3(n, val);
vector<T> v4(n);
vector<T> v5{a,b,c...}
vecrot<T> v5={a,b,c...}
```

如果用圆括号，那么提供的值是用来构造 vector 对象的。

如果用花括号，则是使用列表初始化该 vector 对象。

（2）向 vector 对象添加元素

先定义一个空的 vector 对象，在运行的时候使用 push_back 向其中添加具体指。

（3）其他 vector 操作

```
v.empty();
v.size();
v.push_back(t);
v[n];
```

::: warning

只能对确认已存在的元素执行下标操作。

:::

迭代器介绍

迭代器运算符

```
*iter            // 解引用，返回引用
iter->mem        // 等价于  (*iter).mem
++iter
--iter
iter1 == iter2
iter1 != iter2
```

```
iter + n
iter - n
iter += n
iter -= n
iter1 - iter2      // 两个迭代器相减的结果是它们之间的距离
>, >=, <, <=      // 位置比较
```

::: warning

凡是使用了迭代器的循环体，都不能向迭代器所属的容器添加元素。

:::

数组

（1）数组、指针

使用数组下标的时候，通常将其定义为 size_t 类型。

::: warning

定义数组必须指定数组的类型，不允许用 auto 推断。

不存在引用的数组。

如果两个指针分别指向不相关的对象，则不能进行对这 2 个指针进行比较。

:::

多维数组

多维数组实际上是数组的数组。

```
size_t cnt = 0;
for(auto &row : a)
      for (auto &col : row){
            col = cnt;
            ++cnt;
      }
int *ip[4];    // 整型指针的数组
int (*ip)[4];  // 指向含有 4 个整数的数组
```

术语

begin string 和 vector 的成员，返回指向第一个元素的迭代器。也是一个标准库函数，输入一个数组，返回指向该数组首元素的指针。

end string 和 vector 的成员，返回一个尾后迭代器。也是一个标准库函数，输入一个数组，返回指向该数组尾元素的下一个位置的指针。

第四章 表达式

P120-P151

4.1 基础

重载运算符：为已经存在的运算符赋予了另外一层含义。

左值、右值：

当一个对象用作右值得时候，用的是对象的值（内容）。

当对象被用作左值得时候，用的是对象的身份（在内存中的位置）。

4.2 算术运算符

%：参与取余运算的运算对象必须是整数类型。

4.3 逻辑和关系运算符

运算符
!
<
<=
>
>=
==
!=
&&
||

&& 运算符和 || 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。

::: warning

进行比较运算的时候，除非比较的对象是 bool 类型，否则不要使用布尔字面值 true,false 作为运算对象。

:::

4.4 赋值运算符

赋值运算符满足右结合律。

不要混淆相等运算符和赋值运算符

```
if (i = j)

if (i == j)
```

4.5 递增和递减运算符

递增运算符 ++

递减运算符 --

4.6 成员访问运算符

点运算符和箭头运算符

```
n = (*p).size();
n = p->size();
```

4.7 条件运算符

```
condition ? expression1 : expression2;
```

4.8 位运算符

运算符	功能	用法	备注
~	位求反	~expr	1 置为 0，0 置为 1
<<	左移	expr << expr2	在右侧插入值位 0 的二进制位
>>	右移	expr1 >> expr2	
&	位与	expr1 & expr2	对应位置都是 1，则结果 1，否则为 0。
^	位异或	expr1 ^ expr2	对应位置有且只有 1 个为 1，则结果是 1，否则为 0。
|	位或	expr1 | expr2	对应位置至少有 1 个位 1，则结果是 1，否则为 0。

4.9 sizeof 运算符

sizeof 运算符返回一条表达式或一个类型名字所占的字节数，其所得值是一个 size_t 类型，是一个常量表达式。

```
sizeof (type)
sizeof expr
```

4.10 逗号运算符

逗号运算符含有两个运算对象，按照从左向右的顺序依次求值。

4.11 类型转换

隐式转换

显式转换

命名的强制类型转换

```
cast-name<type>(expression)
```

```
// cast-name 是 static_cast,dynamic_cast,const_cast,reinterpret_cast
```

::: tip

由于强制类型转换干扰了正常的类型检查，因此建议避免强制类型转换。

:::

4.12 运算符优先级表

第五章 语句

P154-P178

5.1 简单语句

（1）空语句

```
;     // 空语句
```

（2）复合语句

复合语句是指用花括号括起来的（可能为空的）语句和声明的序列，复合语句也被称作块（block）。

{}

5.2 语句作用域

定义在控制结构当中的变量只在相应语句的内部可见，一旦语句结束，变量就超出其作用范围。

5.3 条件语句

（1）if 语句

（2）switch 语句

case 关键字和它对应的值一起被称为 case 标签。

case 标签必须是整形常量表达式。

如果某个 case 标签匹配成功，将从该标签开始往后顺序执行所有 case 分支，除非程序显示的中断了这一过程。

dedault 标签：如果没有任何一个 case 标签能匹配上 switch 表达式的值，程序将执行紧跟在 default 标签后面的语句。

5.4 迭代语句

（1）while 语句

```
while (condition)
                    statement
```

（2）传统 for 语句

```
for (initializar; condition; expression)
             statement
```

for 语句中定义的对象只在 for 循环体内可见。

（3）范围 for 语句

```
for (declaration : expression)
             statement
```

（4）do while 语句

```
do
      statement
while (condition)
```

5.5 跳转语句

break

break 只能出现在迭代语句或者 switch 语句内部。仅限于终止离它最近的语句，然后从这些语句之后的第一条语句开始执行。

continue

continue 语句终止最近的循环中的当前迭代并立即开始下一次迭代。

goto

goto 的作用是从 goto 语句无条件跳转到同一函数内的另一条语句。

容易造成控制流混乱，应禁止使用。

return

5.6 try 语句块和异常处理

C++中异常处理包括：throw 表达式、try 语句块。

try 和 catch，将一段可能抛出异常的语句序列括在花括号里构成 try 语句块。catch 子句负责处理代码抛出的异常。

throw 表达式语句，终止函数的执行。抛出一个异常，并把控制权转移到能处理该异常的最近的 catch 字句。

第六章 函数

P182-P225

6.1 函数基础

（1）形参和实参：

实参的类型必须与对应的形参类型匹配。

函数的调用规定实参数量应与形参数量一致。

（2）局部对象

形参和参数体内部定义的变量统称为**局部变量**，它们对函数而言是"局部"的，仅在函数的作用域内可见，同时局部变量还会隐藏外层作用域中同名的其他变量。

自动对象：只存在于块执行期间的对象。

局部静态对象：在程序的执行路径第一次经过对象定义语句时候进行初始化，并且直到程序终止才会被销毁。

```
size_t count_calls()
{
	static size_t ctr = 0;
	return ++ctr;
}
```

（3）函数声明

函数的三要素：（返回类型、函数名、形参类型）。

函数可被声明多次，但只能被定义一次。

（4）分离式编译

分离式编译允许把程序分割到几个文件中去，每个文件独立编译。

编译->链接

6.2 参数传递

当形参是引用类型，这时它对应的实参被引用传递或者函数被传引用调用。

当实参被拷贝给形参，这样的实参被值传递或者函数被传值调用。

（1）传值参数

（2）被引用传参

（3）const 形参和实参

（4）数组形参

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。

```
void print(const int*);
void pring(const int[]);
void print(const int[10]);
// 以上三个函数等价
```

数组引用实参： f(int (&arr)[10])

```
int *matrix[10];   // 10 个指针构成的数组
int (*matrix)[10]; // 指向含有 10 个整数的数组的指针
```

（5）含有可变形参的数组

initializer_list

```
for err_msg(initializer_list<string> li)
```

6.3 返回类型和 return 语句

2 种：无返回值函数和右返回值函数。

```
return;
return expression;
```

函数完成后，它所占用的存储空间也会随着被释放掉。

::: warning

返回局部对象的引用是错误的；返回局部对象的指针也是错误的。

:::

6.4 函数重载

重载函数：同一作用域内的几个函数名字相同但形参列表不通，我们称之为重载函数。（overloaded）。

不允许 2 个函数除了返回类型外其他所有的要素都相同。

重载与作用域

如果在内存作用域中声明名字，它将隐藏外层作用域中声明的同名实体。

6.5 特殊用途语言特性

（1）默认实参

函数调用时，实参按其位置解析，默认实参负责填补函数调用缺少的尾部实参。

```
typedef string::size_type sz;
string screen(sz ht = 24, sz wid = 80, char background = ' ');
```

::: tip

当设计含有默认实参的函数时，需要合理设置形参的顺序。一旦某个形参被赋予了默认值，它后面的所有形参都必须有默认值。

:::

（2）内联函数

使用关键词 inline 来声明内联函数。

内联用于优化规模较小，流程直接，频繁调用的函数。

（3）constexpr 函数

constexpr 函数是指能用于常量表达式的函数。

6.6 函数匹配

Step1:确定候选函数和可选函数。

Step2:寻找最佳匹配。

6.7 函数指针

函数指针指向的是函数而非对象。

```
void useBigger (const string &s1, const string &s2, bool pf(const strin
g &, const string &));
```

等价于

```
void useBigger (const string &s1, const string &s2, bool (*pf)(const st
ring &, const string &));
```

第七章 类

P228-P273

类的基本思想是数据抽象和封装。

抽象是一种依赖于接口和实现分离的编程技术。

封装实现了类的接口和实现的分离。

7.1 定义抽象数据类型

（1）this

任何对类成员的直接访问都被看作 this 的隐式引用。

```
std::string isbn() const {return bookNo;}
```

等价于

```
std::string isbn() const {return this->bookNo;}
```

（2）在类的外部定义成员函数

类外部定义的成员的名字必须包含它所属的类名。

```
double Sales_data::avg_price() const {
      if (units_sol)
            return revenue/units_sols;
      else
            return 0;
}
```

（3）构造函数

> 定义：类通过一个或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。
>
> 构造函数没有返回类型；
>
> 构造函数的名字和类名相同。

类通过一个特殊的构造函数来控制默认初始化过程，这个函数叫做**默认构造函数**。

编译器创建的构造函数被称为**合成的默认构造函数**。

::: tip

只有当类没有声明任何构造函数的时，编译器才会自动的生成默认构造函数。

一旦我们定义了一些其他的构造函数，除非我们再定义一个默认的构造函数，否则类将没有默认构造函数

:::

7.2 访问控制与封装

（1）访问控制

说明符	用途
public	使用 public 定义的成员，在整个程序内可被访问，public 成员定义类的接口。
private	使用 private 定义的成员可以被类的成员函数访问，但是不能被使用该类的代码访问，private 部分封装了类的实现细节。

（2）友元

类可以允许其他类或者函数访问它的非公有成员，方法是令其他类或者函数成为它的友元。

以 friend 关键字标识。

友元不是类的成员，不受访问控制级别的约束。

::: tip

友元的声明仅仅制定了访问的权限，而非通常意义的函数声明。必须在友元之外再专门对函数进行一次声明。

:::

```
// Sales_data.h

class Sales_data {
friend Sales_data add(const Sales_data&, const Sales_data&);
friend std::ostream &print(std::ostream&, const Sales_data&);
friend std::istream &read(std::istream&, Sales_data&);
}

// nonmember Sales_data interface functions
Sales_data add(const Sales_data&, const Sales_data&);
std::ostream &print(std::ostream&, const Sales_data&);
std::istream &read(std::istream&, Sales_data&);

//Sales_data.cpp

Sales_data
add(const Sales_data &lhs, const Sales_data &rhs)
{
      Sales_data sum = lhs;  // copy data members from lhs into sum
      sum.combine(rhs);      // add data members from rhs into sum
      return sum;
}
```

```
// transactions contain ISBN, number of copies sold, and sales price
istream&
read(istream &is, Sales_data &item)
{
      double price = 0;
      is >> item.bookNo >> item.units_sold >> price;
      item.revenue = price * item.units_sold;
      return is;
}

ostream&
print(ostream &os, const Sales_data &item)
{
      os << item.isbn() << " " << item.units_sold << " "
         << item.revenue << " " << item.avg_price();
      return os;
}
```

7.3 类的其他特性

（1）重载成员变量

```
Screen myScrren;
char ch = myScreen.get();
ch = myScreen.get(0,0);
```

（2）类数据成员的初始化

类内初始值必须使用=或者{}的初始化形式。

```
class Window_mgr{
private:
    std::vector<Screen> screens{Screen(24, 80, ' ')};
}
```

（3）基于 const 的重载

```
class Screen {
public:
      // display overloaded on whether the object is const or not
    Screen &display(std::ostream &os)
                  { do_display(os); return *this; }
    const Screen &display(std::ostream &os) const
                  { do_display(os); return *this; }
}
```

当某个对象调用 display 的时候，该对象是否是 const 决定了应该调用 display 的哪个版本。

（3）类类型

对于一个类来说，在我们创建他的对象之前该类必须被定义过，而不能仅被声明。

（4）友元

友元类

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。

```
class Screen {
      // Window_mgr 的成员可以访问 Screen 类的私有部分
      friend class Window_mgr;
}
```

令成员函数作为友元

```
class Screen {
      // Window_mgr::clear 必须在 Screen 类之前被声明
      friend void Window_mgr::clear(ScreenIndex);
}
```

7.4 类的作用域

一个类就是一个作用域。

7.5 构造函数再探

（1）构造函数的初始值有时必不可少

::: tip

如果成员是 const、引用，或者属于某种未提供默认构造函数的类类型化。我们必须通过构造函数初始值列表为这些成员提供初值。

:::

```
class ConstRef{
public:
    ConstRef (int i);
private:
      int i;
      const int ci;
      int &ri;
};

ConstRef:ConstRef(int ii) : i(ii), ci(ii), ri(i){  }
```

（2）成员初始化的顺序

成员初始化的顺序与它们在类定义中出现 的顺序一致。P259

（3）委托构造函数

使用它所述类的其他构造函数执行它自己的初始化过程。

（4）如果去抑制构造函数定义的隐式转换？

在类内声明构造函数的时候使用 explicit 关键字。

7.6 类的静态成员

（1）声明静态成员

在成员的声明之前加上关键词 static。

类的静态成员存在于任何对象之外，对象中不包含任何与静态成员有关的数据。

（2）使用类的静态成员

```
double r;
r = Account::rate();
```

小结

类有两项基本能力：

一是数据数据抽象，即定义数据成员和函数成员的能力；

二是封装，即保护类的成员不被随意访问的能力。

第八章 IO 库

P278-P290

C++语言不直接处理输入输出，而是通过一组定义在标准库中的类型来处理 IO。

- iostream 处理控制台 IO
- fstream 处理命名文件 IO
- stringstream 完成内存 string 的 IO

ifstream 和 istringstream 继承自 istream

ofstream 和 ostringstream 继承自 ostream

8.1 IO 类

（1）IO 对象无拷贝或复制。

进行 IO 操作的函数通常以引用方式传递和返回流。

（2）刷新输出缓冲区

flush 刷新缓冲区，但不输出任何额外的字符；

ends 向缓冲区插入一个空字符，然后刷新缓冲区。

8.2 文件输入输出

类	作用
ifstream	从一个给定文件读取数据
ofstream	从一个给定文件写入数据
fstream	读写给定文件

8.3 string 流

类	作用
istringstream	从 string 读取数据
ostringstream	向 string 写入数据
stringstream	既可从 string 读数据也可以向 string 写数据

```
      // will hold a line and word from input, respectively
      string line, word;

      // will hold all the records from the input
      vector<PersonInfo> people;

      // read the input a line at a time until end-of-file (or other er
ror)
      while (getline(is, line)) {
            PersonInfo info;            // object to hold this record's
 data
          istringstream record(line); // bind record to the line we jus
t read
            record >> info.name;        // read the name
          while (record >> word)      // read the phone numbers
                info.phones.push_back(word);  // and store them
            people.push_back(info); // append this record to people
      }


      // for each entry in people
      for (vector<PersonInfo>::const_iterator entry = people.begin();
                        entry != people.end(); ++entry) {
            ostringstream formatted, badNums; // objects created on eac
h loop

            // for each number
```

```
            for (vector<string>::const_iterator nums = entry->phones.be
gin();
                        nums != entry->phones.end(); ++nums) {
                  if (!valid(*nums)) {
                        badNums << " " << *nums;  // string in badNums
                  } else
                        // ``writes'' to formatted's string
                        formatted << " " << format(*nums);
            }
            if (badNums.str().empty())      // there were no bad number
s
                  os << entry->name << " "    // print the name
                     << formatted.str() << endl; // and reformatted num
bers
            else                        // otherwise, print the name and bad
 numbers
                  cerr << "input error: " << entry->name
                       << " invalid number(s) " << badNums.str() << end
l;
      }
```

第九章 顺序容器

P292-P332

顺序容器为程序员提供了控制元素存储和访问顺序的能力。

9.1 顺序容器概述

类型	作用
vector	可变数组大小。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、随机访问快。在尾部插入/删除速度快。

9.2 容器库概述

一般，每个容器都定义在一个头文件中。

容器均定义为模板类。

类型别名

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容器的大小
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	元素的左值诶性：与 value_type&含义相同
const_reference	元素的 const 左值类型（即，const value_type&）

构造函数

C c;	默认构造函数，构造空容器
C c1(c2)	构造 c2 的拷贝 c1
C c(b, e)	构造 c，将迭代器 b 和 e 指定的范围内的元素拷贝到 c（array 不支持）
C c{a, b, c...}	列表初始化 c

赋值与 swap

c1=c2	将 c1 中的元素替换为 c2 中元素
c1 = {a, b, c...}	将 c1 中的元素替换为列表中元素（不适用于 array）
a.swap(b)	交换 a 和 b 的元素
swap(a, b)	与 a.swap(b)等价

大小

c.size()	c 中元素的数组（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/删除元素（不适用于 array）

c.insert(args)	将 args 中的元素拷贝进 c
c.emplace(inits)	使用 inits 构造 c 中的一个元素
c.erase(args)	删除 args 指定的元素
c.clear()	删除 c 中的所有元素，返回 void

关系运算符	
==, !=	所有容器都支持相等(不等运算符)
<,<=,>,>=	关系运算符(无序关联容器不支持）

获取迭代器	
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.cbengin(),c.cend()	返回 const_iterator

反向容器的额外成员（不支持 forward_list）	
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

（1）迭代器

标准库的迭代器允许我们访问容器中的元素，所有迭代器都是通过解引用运算符来实现这个操作。

一个迭代器返回由一对迭代器表示，两个迭代器分别指向同一个容器中的元素或者是尾元素之后的位置。它们标记了容器中元素的一个范围。

左闭合区间：[begin, end)

```
while (begin !=end){
        *begin = val;
        ++begin;
}
```

（2）容器类型成员

见概述

通过别名，可以在不了解容器中元素类型的情况下使用它。

（3）begin 和 end 成员

begin 是容器中第一个元素的迭代器

end 是容器尾元素之后位置的迭代器

（4）容器定义和初始化

P290

```
C c;              // 默认构造函数
C c1(c2)
C c1=c2
C c{a,b,c...}     // 列表初始化
C c={a,b,c...}
C c(b,e)          // c 初始化为迭代器 b 和 e 指定范围中的元素的拷贝
// 只有顺序容器（不包括 array）的构造函数才能接受大小参数
C seq(n)
C seq(n,t)
```

将一个容器初始化为另一个容器的拷贝:

当将一个容器初始化为另一个容器的拷贝时，两个容器的容器类型和元素类型都必须相同。

不过，当传递迭代器参数来拷贝一个范围时，就不要求容器类型相同，只要能将要拷贝的元素转换为要初始化的容器的元素类型即可。

标注库 array 具有固定大小:

不能对内置数组类型进行拷贝或对象赋值操作，但 array 并无此限制。P301

（5）赋值与 swap

arrray 类型不允许用花括号包围的值列表进行赋值。

```
array<int, 10> a2={0}; //所有元素均为 0
s2={0};  // 错误!
```

seq.assign(b,e) // 将 seq 中的元素替换为迭代器 b 和 e 所表示的范围中的元素。迭代器 b 和 e 不能指向 seq 中的元素。

swap 用于交换 2 个相同类型容器的内容。调用 swap 之后，两个容器中的元素将交换。

（6）容器大小操作

size 返回容器中元素的数目

empty 当 size 为 0 返回布尔值 true，否则返回 false

max_size 返回一个大于或等于该类型容器所能容纳的最大元素数的值

（7）关系运算符

关系运算符左右两边的元素符对象必须是相同类型的容器。

::: tip

只有当元素类型也定义了相应的比较运算符，才可以使用关系元素安抚来比较两个容器

:::

9.3 顺序容器操作

（1）向顺序容器添加元素

表格 P305

使用 push_back:追加到容器尾部

使用 push_front:插入到容器头部

在容器中的特定位置添加元素:使用 insert

```
vector <string> svec;
svec.insert(svec.begin(), "Hello!");
```

插入范围内元素:使用 insert

使用 emplace 操作:

emplace*front*、*emplace* 和 *emplace*back 分别对应 push*front*、*insert* 和 *push*back。

emplace 函数直接在容器中构造函数，不是拷贝。

（2）访问元素

P309

注意 end 是指向的是容器尾元素之后的元素。

在顺序容器中访问元素的操作

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 中首元素的引用。若 c 为空，哈数行为未定义
c[n]	返回 c 中下标为 n 的元素的引用，n 是一个无符号整数。若 n>=size(),则函数行为未定义
c.at[n]	返回下标为 n 的元素的引用。如果下标越界，则抛出 out*of*range 异常

（3）删除元素

顺序容器的删除操作	
c.pop_back()	删除 c 中尾元素。若 c 为空，则函数行为未定义。返回返回 void
c.pop_front()	删除 c 中首元素。若 c 为空，则函数行为未定义。返回 void
c.erase(p)	删除迭代器 p 所指定的元素，返回一个指向被删除元素之后元素的迭代器，如 p 指向尾元素，则返回尾后(off-the-end)迭代器。若 p 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 所指定范围内的元素。返回一个指向最后一个被删除元素之后元素的迭代器。若 e 本身就是尾后迭代器，则函数也返回尾后迭代器
c.claer()	删除 c 中的所有元素。返回 void

（4）特殊的 forwar_list 操作

P313

befor*begin();cbefore*begin();insert*after;emplace*after;erase_after;

（5）改变容器大小

reseize 用于扩大或者缩小容器。

resize 操作接受一个可选的元素值参数，用来初始化添加到容器内的元素。

如果容器保存的是类类型元素，且 resize 向容器中添加新元素，则必须提供初始值，或者元素类型必须提供一个默认构造函数。

9.4 vector 对象是如何增长的

为了避免影性能，标准库采用了可以减少容器空间重新分配次数的策略。当不得不获取新的内存空间时，vector 和 string 通常会分配比新的新的空间需求更大的内存空间。容器预留这些空间作为备用，可以用来保存更多的新元素。

容器管理的成员函数:

容器大小管理擦作	
c.shrink*to*fit()	请将 capacity()减少为与 size()相同大小
c.capacity()	不重新分配内存空间的话,c 可以保存多少元素
c.reverse()	分配至少能容纳 n 个元素的内存空间。reverse 并不改变容器中元素的数量，它仅影响 vector 预先分配多大的内存空间。调用 reverse 永远不减少容器占用的内存空间。

capcacity 和 size:

区别：

容器的 size 是指它已经保存的元素的数目；

capcacity 则是在不分配新的内存空间的前提下它最多可以保存多少元素。

注意：只有当迫不得已时才可以分配新的内存空间。

9.5 额外的 string 操作

（1）构造 string 的其他方法

构造 string 的其他方法

string s(cp, n)	s 是 cp 指向的数组中前 n 个字符的拷贝
string s(s2, pos2)	s 是 string s2 从下标 pos2 开始的字符的拷贝。
string s (s2, pos2, len2)	s 是 string s2 从下标 pos2 开始 len2 个字符的拷贝

substr 操作:

substr 操作返回一个 string，它是原始 string 的一部分或全部的拷贝。

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 的默认值为 0。n 的默认值为 s.size() - pos，即拷贝从 pos 开始的所有字符

（2）改变 string 的其他方法

assign 替换赋值，总是替换 string 中的所有内容

insert 插入

append 末尾插入，总是将新字符追加到 string 末尾

replace 删除再插入

（3）string 搜索操作

string 搜索操作

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*last*not_of(args)	在 s 中查找最后一个不在 args 中的字符

（4）compare 函数

compare 有 6 个版本，P327

（5）数值转换

P328

tostring

stod

9.6 容器适配器

顺序容器适配器：

stack; queue; priority_queue;

适配器是一种机制，能使某种事物看起来像另外一种事物。

定义一个适配器:

适配器有 2 个构造函数:

1、默认构造函数创建一个空对象
2、接受一个容器的构造函数

栈适配器:

栈的操作

s.pop()	删除栈顶元素，但不返回该元素值
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，或者由 args 构造
s.emplace(args)	由 arg 构造
s.top()	返回栈顶元素，但不将元素弹出栈

队列适配器:

queue 和
priority_queue 操作

q.pop()	返回 queue 的首元素或 priority_queue 的最高优先级的元素，但不删除此元素
q.front() q.back()	返回首元素或尾元素，但不删除此元素。只适用于 queue
q.top()	返回最高优先级元素，但不删除该元素。只适用于 priority_queue
q.push(item) q.empalce(args)	在 queue 末尾或 priority_queue 中恰当的位置创建一个元素，其值为 item,或者由 args 构造

术语

begin 容器操作：返回一个指向容器首元素的迭代器，如果容器为空，则返回尾后迭代器。是否返回 const 迭代器依赖于容器的类型。

cbegin 容器操作：返回一个指向容器尾元素之后的 const_iterator。

第十章 泛型算法

P336-P371

标准库并未给每个容器添加大量功能，而是提供了一组算法。这些算法是通用的，可以用于不同类型的容器和不同类型的元素。

10.1 概述

头文件：algorithm、numeric

算法不依赖于容器，但算法依赖于元素类型的操作。

10.2 初识泛型算法

（1）只读算法

accumulate 求和

equal 是否相等

（2）写容器元素的算法

算法不检查写操作

拷贝算法：copy

重排容器元素的算法：sort

::: tip

标准库函数对迭代器而不是容器进行操作。因此，算法不能直接添加或删除元素

:::

10.3 定制操作

标准库允许我们提供自己定义的操作来代替默认运算符。

（1）向算法传递函数

谓词:

谓词是一个可调用的表达式，其返回结果是一个能用作条件的值。

标准库算法的谓词分为两类：

1、一元谓词：只接受单一参数。

2、二元谓词：接受两个参数。

```
bool isShorter(const string &s1, const string &s2)
{
            retrun s1.size() < s2.size();
}

sort(words.begin(), words.end(), isShorter);
```

排序算法:

stable_sort 算法维持相等元素的原有顺序。

（2）lambda 表达式

lamba:

lambda 表达式表示一个可调用的代码单元。一个 lambda 具有一个返回类型、一个参数列表和一个函数体。

```
[capture list](parameter list) -> return type {function body}
// capture list 捕获列表，lambda 所在函数中定义的局部变量
// 捕获列表只用于局部非 static 变量，lambda 可以直接使用局部 static 变量和在它所在函数之外声明的名字
// lambda 必须使用尾置返回来指定返回类型
```

（3）lambda 捕获和返回

两种：值捕获、引用捕获

::: warnning

当以引用方式捕获一个变量时，必须保证在 lambda 执行时变量是存在的。

一般的，应该尽量减少捕获的数据量，来避免潜在的问题。

如果可能，避免捕获指针或引用。

:::

隐式捕获：

当混合使用隐式捕获和显式捕获时，捕获列表中的第一个元素必须是一个&或=。显式捕获的变量必须使用与隐式捕获不同的方式。

lambda 捕获列表 P352

可变 lambda:

若希望改变一个被捕获的变量的值，必须在参数列表首加上关键字 mutable。

指定 lambda 返回类型:

当需要为 lambda 定义返回类型时，必须使用尾置返回类型。

（4）参数绑定

标准库 bind 函数:

```
auto newCallable = bind(callable, arg_list);
// 调用 newCallable 时，newCallable 会调用 callable，并传递给它 arg_list 中的参数
```

10.4 再探迭代器

插入迭代器、流迭代器、反向迭代器、移动迭代器

（1）插入迭代器

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 inserter 的迭代器

（2）iostream 迭代器

istream_iterator 读取输入流

ostream_iterator 向一个输出流写数据

istream_iterator 操作:

istream-iterator 操作	
istream_iterator in(is);	in 从输入流 is 读取类型为 T 的值
istream_iterator end;	读取类型为 T 的值得 istream_iterator 迭代器，表示尾后位置
in1 == in2 in1 != in2	in1 和 in2 必须读取相同类型。如果它们都是尾后迭代器，或绑定到相同的输入，则两者相等
*in	返回从流中读取的值
in->mem	与(*in).mem 含义相同
++in, in++	用>>从输入流读取下一个值

ostream_iterator 操作:

ostream_iterator 操作	
ostream_iterator out(os);	out 将类型为 T 的值写到输出流 os 中
ostream_iterator 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 指向一个空字符串结尾的字符数组
out = val	用<<将 val 写入到 out 所绑定的 ostream 中
*out, ++out, out++	

（3）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。

10.5 泛型算法结构

迭代器类别	
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

10.6 特定容器算法

对于 list、forward_list，应该优先使用成员函数的算法而不是通用算法。

术语

cref 标准库函数：返回一个可拷贝的对象，其中保存了一个指向不可拷贝类型的 const 对象的引用

第十一章 关联容器

P374-P397

关联容器支持高效的关键字查找和访问。

类型	备注
map	关联数组，保存关键字-值对
set	值保存关键字的容器
multimap	关键字可重复出现的 map
multiset	关键字可重复出现的 set
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

11.1 使用关联容器

map 是关键词-值对的集合。

为了定义一个 map，我们必须指定关键字和值的类型。

```
// 统计每个单词在输入中出现的次数
map<string, size_t> word_count;
string word;
while (cin >> word)
    ++word_count[word];
for (const auto &w : word_count)
      count << w.first << " cccurs " < w.second
                        << ((w.second > 1) ? " times" : "time") << endl;
```

set 是关键字的简单集合。

为了定义一个 set，必须指定其元素类型。

```
// 统计输入中每个单词出现的次数，并忽略常见单词
map<string, size_t> word_count;
set<string> exclude = {"the", "But"};
string word;
while (cin >> word)
            // 只统计不在 exclude 中的单词
            if (exclude.find(word) == exclude.end())
                        ++word_count[word]; //获取并递增 word 的计数器
```

11.2 关联容器概述

（1）定义关联容器

定义 map 时，必须指明关键字类型又指明值类型；

定义 set 时，只需指明关键字类型。

（2）pair 类型

pair 标准库类型定义在头文件 utility 中。

一个 pair 保存两个数据成员。当创建一个 pair 时，必须提供两个类型名。

```
pair<string, string> anon; // 保存两个 string
pair<string, string> author{"James", "Joyce"}; // 也可为每个成员提供初始化器
```

pair 的数据类型是 public 的，两个成员分别命名为 first 和 second。

pair 上的操作，见表，P380

11.3 关联容器操作

关联容器额外的类型别名

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 set，与 key*type 相同；对于 map，为 pair<const* key*type, mapped_type>*

（1）关联容器迭代器

set 的迭代器是 const 的

set 和 map 的关键字都是 const 的

遍历关联容器：

map 和 set 都支持 begin 和 end 操作。使用 beigin、end 获取迭代器，然后用迭代器来遍历容器。

（2）添加元素

关联容器 insert 操作

c.insert(v)	v 是 value_type 类型的对象;
c.emplace(args)	args 用来构造一个元素
c.insert(b, e)	
c.insert(il)	
c.insert(p, v)	
c.emplace(p ,args)	

（3）删除元素

从关联容器删除元素

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.end()
c.erase(b, e)	删除迭代器 b 和 e 所表示的范围中的元素。返回 e

（4）map 的下标操作

map 和 unorder_map 的 下标操作	
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，对其进行值初始化
c.at[k]	访问关键字为 k 的元素，带参数检查；若 k 不在 c 中，抛出一个 out*of*range 异常

::: tip

map 进行下标操作，会获得 mapped*type 对象；当解引用时，会得到 value*type 对象。

:::

（5）访问元素

```
c.find(k)  // 返回一个迭代器，指向第一个关键字 k 的元素，如 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；不适用于无序容器
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；不适用于无序容器
c.equal_bound(k)  // 返回一个迭代器 pair，表示关键字等于 k 的元素的范围。如 k 不存在，pair 的两个成员均等于 c.end()
```

11.4 无序容器

无序容器使用关键字类型的==运算符和一个 hash<key_type>类型的对象来组织元素。

无序容器在存储上组织为一组桶，适用一个哈希函数将元素映射到桶。

无序容器管理操作，表格，P395

还可以自定义自己的 hash 模板 P396

```
using SD_multiset = unordered_multiset<Sales_data, decltype(hasher)*, decltype(eqOp)*>;
SD_multiset bookStore(42, haser, eqOp);
```

第十二章 动态内存

P400-P436

12.1 动态指针与智能指针

智能指针	用途
shared_ptr	提供所有权共享的智能指针：对共享对象来说，当最后一个指向它的 shared_ptr 被销毁时会被释放。
unique_ptr	提供独享所有权的智能指针：当 unique*ptr 被销毁的时，它指向的独享被释放。unique*ptr 不能直接拷贝或赋值。
weak_ptr	一种智能指针，指向由 shared*ptr 管理的对象。在确定是否应释放对象视，shared*ptr 并不把 weak_ptr 统计在内。

（1）shared_ptr 类

```
shared_ptr<string> p1;
```

make_shared 函数:

make*shared 在动态内存中分配一个对象并初始化它，返回此对象的 shared*ptr。

```
share_ptr<int> p3 = make_shared<int>(42);
```

shared_ptr 的拷贝和赋值:

每个 shared*ptr 都有一个关联的计数器，称为引用计数。一旦一个 shared*ptr 的引用计数变为 0，就会自动释放自己所管理的对象。

（2）直接管理内存

运算符 new 分配分配内存，delete 释放 new 分配的内存。

使用 new 动态分配和初始化对象:

```
// 默认情况下，动态分配的对象是默认初始化的
int *pi = new int; // pi 指向一个动态分配的、未初始化的无名对象

// 直接初始化方式
int *pi = new int(1024); // pi 指向的对象的值为1024

// 对动态分配的对象进行值初始化，只需在类型名之后加上一对空括号
int *pi1 = new int;   // 默认值初始化；*pi1 的值未定义
int *pi2 = new int(); // 值初始化为0；*pi2 为0
```

动态分配的 const 对象:

```
const int *pci = new const int(1024);
```

释放动态内存:

```
delete p;
```

delete 表达式执行两个动作：销毁给定的指针指向的对象；释放对应的内存。

（3）unique_ptr

某个时刻，只能有一个 unique_ptr 指向一个给定对象。

当 unique_ptr 销毁时，它所指向的对象也被销毁。

unique_ptr 操作
unique_ptr u1
unique_ptr<T, D> u2
unique_ptr<T, D> u(d)
u = nullptr
u.release()
u.reset()
u.reset(p)
u.reset(nullptr)

（4）weak_ptr

weak+ptr 是一种不受控制所指向对象生存期的智能指针，它指向由一个 shared*ptr 管理的对象，而且不会改变 shared*ptr 的引用计数。

weak_ptr 操作	
weak_ptr w	
weak_ptr w(sp)	
w = p	
w.reset()	将 w 置空
w.use_count()	与 w 共享对象的 shared_ptr 的数量
w.expired()	
w.lock()	

使用 weak*ptr 之前，需要调用 lock，检查 weak*ptr 指向的对象是否存在。

12.2 动态数组

（1）new 和数组

在类型名之后跟一对方括号，在其中指明要分配的对象的数目。

释放动态数组：

```
delete p;       // p 必须指向一个动态分配的对象或为空
delete [] pa;   // pa 必须指向一个动态分配的数组或为空
```

智能指针和动态数组

```
unique_ptr<T []> u;
unique_ptr<T []> u(p);
u[i];
```

（2）allocator 类

标准库 allocator 类定义在头文件 memory 中，帮助将内存和对象构造分离开来。

```
allocator<string> alloc;
auto const p = alloc.allocate(n);
```

表达式	作用
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.construct(p, args)	为了使用 allocate 返回的内存，我们必须使用 construct 构造对象。使用未构造的内存，其行为是未定义的。
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

术语

new：从自由空间分配内存。new T 分配并构造一个类型为 T 的指针。如果 T 是一个数组类型，new 返回一个指向数组首元素的指针。类似的，new [n] T 分配 n 个类型为 T 的对象，并返回指向数组首元素的指针。

空悬指针：一个指针，指向曾经保存一个对象但现在已释放的内存。

智能指针：标准库类型。负责在恰当的时候释放内存。

第十三章 拷贝控制

P440-P486

五种拷贝控制操作：

拷贝构造函数、拷贝赋值运算符、移动构造函数、移动赋值运算符、析构函数。

拷贝构造函数、移动构造函数定义了当用同类型的另一个对象初始化本对象时做什么。

拷贝赋值运算符、移动赋值运算符定义了将一个对象赋予同类型的另一个对象时做什么。

析构函数定义了当此类型对象销毁时做什么。

13.1 拷贝、赋值与销毁

（1）拷贝构造函数

拷贝构造函数的第一个参数必须是一个引用类型。

```
class Foo {
public :
	Foo(); // 默认构造函数
	Foo(const Foo&); // 拷贝构造函数
}
```

合成拷贝构造函数:

若未定义拷贝构造函数，编译器会定义一个。

拷贝初始化:

拷贝初始化，要求编译器将右运算对象拷贝到正在创建的对象中。拷贝初始化通常使用拷贝构造函数来完成。

（2）拷贝赋值运算符

重载赋值运算符：oprator=

合成拷贝赋值运算符：若一个类未定义自己的拷贝赋值运算符，编译器会为它生成一个合成拷贝赋值运算符。

（3）析构函数

析构函数：用于释放对象使用的资源，销毁对象的非 static 数据成员。

```
class Foo {
public:
	~Foo(); // 析构函数，一个类只会有唯一一个析构函数。
}
```

在一个析构函数中，不存在类似构造函数中初始化列表的东西来控制成员如何销毁，析构部分是隐式的。销毁类类型的成员需要执行成员自己的析构函数。

合成析构函数：当一个类未定义自己的析构函数时，编译器会为它定义一个合成析构函数。

析构函数体本身并不直接销毁成员。

（4）三五法则

P447

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

（5）使用 default=

将拷贝控制成员定义为=dafault 来显式地要求编译器生活才能合成的版本。

```
class Sales_data {
public:
            Sales_data(const Sales_data&) = default;
}
```

（6）阻止拷贝

在函数参数列表后面加上=delete。

=delete 必须出现在函数第一次声明的时候。

析构函数不能是删除的成员

合成的拷贝控制成员可能是删除的:

如果一个类有数据成员不能默认构造、拷贝、复制或销毁，则对应的成员函数将被定义为删除的。

13.2 拷贝控制和资源管理

（1）行为像值的类

为了提供类值的行为，对于类管理的对象，每个对象都应该拥有一份自己的拷贝。

类值拷贝赋值运算符:通常组合了析构函数和构造函数的操作。

```
HasPtr& HasPtr::operator=(const HasPtr &rhs)
{
      auto newp = new string(*rhs.ps);
      delete ps;
      ps = newp;
      i = rhs.i;
      return *this;
}
```

（2）行为像指针的类

如果需要可直接管理资源，可以使用引用计数。

13.3 交换操作

swap

13.4 拷贝控制示例

P460

13.5 动态内存管理类

P464

13.6 对象移动

与任何赋值运算符一样，移动赋值运算符必须销毁左侧运算对象的旧状态。

（1）右值引用

可通过 move 函数开获得绑定到左值上的右值引用。

```
int && rr3 = std::move(rr1);
```

（2）移动构造函数和移动赋值运算符

移动构造函数的第一个参数是该类类型的一个右值引用。

移动赋值运算符:

```
StrVec &StrVec::operator=(StrVec &&rhs) noexcept
{

}
```

合成的移动操作：

若一个类定义了自己的拷贝构造函数、拷贝赋值运算符或者析构函数，编译器就不会为它合成移动构造函数和移动赋值运算符。

如果一个类没有移动操作，类会使用对应的拷贝操作来代替移动操作。

移动迭代器:

移动迭代器的解引用运算符生成一个右值引用。

（3）右值引用和成员函数

::: tip

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&&。

:::

右值和左值引用成员函数:

指出 this 的左值/右值属性的方式与定义 const 成员函数相同，在参数列表后放置一个引用限定符。P483

::: tip

如果一个成员函数有引用限定符，则具有相同参数列表的所有版本都必须有引用限定符。P485

:::

术语

引用限定符：被&限定的函数只能用于坐值；被&&限定的函数只能用于右值。

第十四章 重载运算与类型转换

P490-P523

通过运算符重载可重新定义该运算符的含义。

14.1 基本概念

定义：重载运算符是具有特殊名字的函数。名字由 operator 和符号组成。重载运算符包含返回类型、参数列表和函数体。

::: tip

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。成员运算符函数的显式参数数量比运算对象的数量少一个。

对于一个运算符来说，它或者是类的成员，或者至少含有一个类类型的参数。

我们只能重载已有的运算符。

:::

直接调用一个重载的运算符函数

```
data1 + data2;
operator+(data1, data2);
// 以上 2 个调用等价
```

14.2 输入和输出运算符

（1）重载输出运算符<<

```
ostream &operator<<(ostream &os, const Sales_data &item)
{
	os << item.isbn() << " " << item.unites_sold << " " << item.reven
ue << " " << item.avg_price();
	return os;
}
```

（2）重载输入运算符>>

```
istream &operator>>(istream &is, Sales_data &item)
{
	double price;
	is >> item.bookNo >> item.units_sold >> price;
```

```
	if (is)
		item.revenue = items.units_sold * price;
	else
		item = Sales_data();
	return is;
}
```

14.3 算术和关系运算符

（1）相等运算符

```
bool operator==(const Sales_data &lhs, const Sales_data &rhs)
{
	return lhs.isbn() == rhs.isbn() &&
			lhs.unites_sold == rhs.units_sold &&
			lhs.revenue == rhs.revenue;
}
```

（2）关系运算符

operator<

14.4 赋值运算符

operator=

operator+=

14.5 下标运算符

operator[]

下标运算符必须是成员函数。

```
class StrVec{
public:
	std::string& operator[](std::size_t n){
		return elements[n];
	}
	const std::string& operator[](std::size_t n) const{
		return elements[n];
	}
private:
	std::string *elements;
}
```

14.6 递减和递增运算符

递增运算符（++）

递减运算符（--）

定义前置递增/递减运算符:

```
class StrBlobPtr{
public:
      StrBlobPtr& operator++();  // 前置运算符
      StrBlobPtr& operator--();
}
```

区分前置和后置运算符:

```
class StrBlobPtr{
public:
      StrBlobPtr operator++(int); // 后置运算符
      StrBlobPtr operator--(int);
}
```

14.7 成员访问运算符

operator*

operator->

14.8 函数调用运算符

如果类重载了函数调用运算符，则我们可以像使用函数一样使用该类的对象。

```
struct absInt{
      int operator()(int val) const {
            return val < 0 ? -val : val;
      }
};

absInt absObj;
int ui = absObj(i);
```

如果定义了调用运算符，则该类的对象称为函数对象。

14.9 重载、类型转换与运算符

（1）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，将一个类类型的值转换成其他类型。形式：

operator type() const;

（2）避免有二义性的类型转换

（3）函数匹配与重载运算符

::: warning

如果对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，将会遇到重载运算符与内置运算符的二义性问题。

:::

术语

类类型转换:由构造函数定义的从其他类型到类类型的转换以及由类型转换运算符定义的从类类型到其他类型的转换。

第十五章 面向对象程序设计

P526-P575

15.1 OOP:概述

（1）**面对对象程序设计（object-oriented programming）的核心思想：**

数据抽象、继承和动态绑定。

（2）**继承：**

> 继承是一种类联系在一起的一种层次关系。这种关系中，根部是*基类*，从基类继承而来的类成为*派生类*。

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。

虚函数：virtual function。基类希望派生类各自定义自身版本的函数。

```
class Quote {
public:
            std::string isbn() const;
            virtual double net_price(std::size_t n) const;
}
```

（3）**动态绑定：**

::: tip

在 C++语言中，当我们使用基类的引用（或者指针）调用一个虚函数时将发生动态绑定（也称*运行时绑定*）。P527

:::

15.2 定义基类和派生类

（1）定义基类

虚函数：基类希望派生类进行覆盖的函数。

基类将该函数定义为虚函数（virtual）。

基类通过在其成员函数的声明语句之前加上关键词 virtual 使得该函数执行动态绑定。

关键词 virtual 只能出现在类内部的声明语句之前而不能用于类外部的函数定义。

如果基类把一个函数声明成虚函数，则该函数在派生类中隐式的也是虚函数。

（2）定义派生类

派生类必须通过派生类列表明确指出它是从哪个基类继承而来的。

```
class Bulk_quote : public Quote {
... // 省略
}
```

对于派生类中的虚函数的处理：

若派生类未覆盖基类中的虚函数，则该虚函数的行为类似其他普通成员。

C++允许派生类显式注明覆盖了基类的虚函数，可通过添加 override 关键字。

派生类对象:

一个派生类对象包含多个部分：自己定义的成员的子对象，以及基类的子对象。

派生到基类的类型转换：

由于派生类对象中含有与其基类对象的组成部分，因此可以进行隐式的执行派生类到基类的转换。

```
Quote item;         // 基类
Bulk_quote bulk;    // 派生类
Quote *p = &item;   // p 指向Quote 对象
p = &bulk;          // p 指向bulk 的Quote 部分
Quote &r = bulk;    // r 绑定到bulk 的Quote 部分。
```

派生类构造函数：

每个类控制自己的成员的初始化过程。派生类首先初始化基类的部分，然后按照声明的顺序依次初始化派生类的成员。

派生类使用基类的成员：

派生类可以访问基类的公有成员和受保护成员。

::: tip

派生类对象不能直接初始化基类的成员。派生类应该遵循基类的借口，通过调用基类的构造函数来初始化从基类继承来的成员。

:::

被用作基类的类:

若使用某个类作为基类，则该类必须已被定义而非仅仅声明。

派生类包含它的直接基类的子对象以及每个间接基类的子对象。

防止继承发生:

在类名后面跟着一个关键字 final。

```
class NoDerived final {};   // NoDerived 不能作为基类
```

（3）类型转换与继承

我们可以将基类的指针或引用绑定到派生类对象上。

静态类型与动态类型:

静态类型：在编译时已知，是变量声明时的类型或表达式生成的类型。

动态类型：运行时才可知，是变量或表达式表示的内存中的对象的类型。

如果表达式既不是引用也不是指针，则动态类型与静态类型永远一致。

不存在基类向派生类隐式类型转换:

```
Quote base;
Bulk_quote *bulkP = &base;    // 错误!
Bulk_quote *bulkRef = base;   // 错误!
```

::: warning

当我么用一个派生类对象为一个基类对象初始化或赋值时，只有该派生类对象中的基类部分会被拷贝、移动或赋值，它的派生类部分会被忽略掉。

:::

15.3 虚函数

C++的多态性：使用这些类型的多种形式，而无须在意它们的差异。

派生类中的虚函数:

一个派生类如果覆盖了某个继承而来的虚函数，则它的形参类型必须与被它覆盖的基类函数完全一致。

final 和 override 说明符:

如果用 override 标记了某个函数，但是该函数并没有覆盖已存在的虚函数，此时编译器将报错。

如果用 final 标记了某个函数， 则之后任何尝试覆盖该函数的操作都将错误。

虚函数与默认实参：

如果虚函数某次被调用使用了默认实参，则该实参值由本次调用的静态类型决定。

15.4 抽象基类

纯虚函数:

书写=0 可以将一个虚函数说明为纯虚函数（pure virtual），纯虚函数无须定义。

不能在类的内部为一个=0 的函数提供函数体。

```
class Disc_quote : public Quote {
public:
            double net_price(std::size_t) const = 0;
}
```

抽象基类：

含有纯虚函数的类是抽象基类。

不能创建抽象基类的对象。

15.5 访问控制与继承

受保护的成员:

派生类的成员和友元只能访问派生类对象中的基类部分的受保护成员；对于普通的基类对象中的成员不具有特殊的访问权限。P543

公有、私有和受保护继承：

派生访问说明符对于派生类的成员（及友元）能否访问其直接基类的成员无影响；

对基类成员的访问权限只与基类中的访问说明符有关。

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。

改变个别成员的可访问性:

通过在类的内部使用 using 声明语句，我们可以将该类的直接或间接基类中的任何可访问成员标记出来。

```
class Derived : private Base {
public:
            using Base::size;
}
```

::: tip

派生类只能为它可访问的名字提供 using 声明。

:::

默认的继承保护级别:

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共有继承的。

```
class Base {};
struct D1 : Base {};  // 默认 public 继承
class D2 : Base {};   // 默认 private 继承
```

15.6 继承中的类作用域

在编译时进行名字查找:

一个对象、引用或指针的静态类型决定了该对象的哪些成员是可见的。

名字冲突与继承:

派生类的成员将隐藏同名的基类成员。

::: tip

出了覆盖继承而来的虚函数外，派生类最好不雅重用其他定义在基类中的名字。

:::

如果派生类的成员函数与基类的某个成员函数同名，则派生类将在其作用域内隐藏掉该基类成员函数。

::: tip

非虚函数不会发生动态绑定。

:::

15.7 构造函数与拷贝控制

（1）虚析构函数

在基类中将析构函数定义成虚函数以确保执行正确的析构函数版本。

```
Quote *itemP = new Quote;
delete itemP;           // 调用Quote 的析构函数
itemP = new Bulk_quote;
delete itemP;           // 调用Bulk_quote 的析构函数
```

虚析构函数会阻止合成移动操作。

（2）合成拷贝控制与继承

基类缺少移动操作会阻止派生类拥有自己的合成移动操作，所以当确实要执行移动操作的时候就要首先在基类中进行显式定义。P554

（3）派生类的拷贝控制成员

派生类的拷贝或移动构造函数:

::: tip

默认情况下，基类默认构造函数初始化派生类对象的基类部分。如果我们想拷贝（或移动）基类部分，则必须在派生类的构造函数初始值列表中显式的使用基类的拷贝（或移动）构造函数。

:::

派生类的赋值运算符:

派生类的赋值运算符必须显式的为其基类部分赋值。

派生类的析构函数:

派生类函数只负责销毁由派生类自己分配的资源。

15.8 容器与继承

当使用容器存放继承体系中的对象时，必须采用间接存储的方式。因为不允许在容器中保存不同类型的元素。

术语

覆盖：override，派生类中定义的虚函数如果与基类中定义的同名虚函数与相同的形参列表，则派生类版本将覆盖基类的版本。

多态：程序能够通引用或指针的动态类型获取类型特定行为的能力。

第十六章 模板与泛型编程

P578-P630

（1）控制实例化

当编译器遇到 extern 模板声明时，它不会在本文件中生成实例化代码。将一个实例化声明为 extern 就表示承诺在程序其他位置有该实例化的一个非 extern 声明（定义）。对于一个给定的实例化版本，可能有多个 extern 声明，但必须只有一个定义。

（2）
模板是标准库的基础。

生成特定类或者函数的过程称为实例化。

（3）术语

类模板： 模板定义，可从它实例化出特定的类。类模板的定义以关键词 template 开始，后面跟尖括号对<和>，其内为一个用逗号分隔的一个或多个模板参数的列表，随后是类的定义。

函数模板： 模板定义，可从它实例化出特定函数。函数模板的定义以关键词 template 开始，后跟尖括号<和>，其内以一个用逗号分隔的一个或多个模板参数的列表，随后是函数的定义。