C++ prime part3&4

part1 基础

Chap2 变量

变量命名规范

要体现实际含义

一般使用小写字母

用户自定义的类名一般使用大写字母开头，如：Sales_item

标识符若有多个单词组成，则单词应有明显区分

驼峰命名法，匈牙利命名法等

赋值和初始化区别

初始化：创建变量时赋予一个初始值

初始化：函数体外部初始化为0，内部不被初始化

赋值： 把对象的当前值擦除，而以一个新值代替

算术类型

查看类型

```
#include <typeinfo>
cout << typeid( var ).name() << endl;
```

bool

未定义

char

8位

前缀u8

wchar_t宽字符

16位

前缀L

char16_t

16位

前缀u

char32_t

32位

前缀U

short

16位

int

16位

long

32位

后缀l或L

long long

64位

后缀ll或LL

float

6位有效数字

后缀f或F

double

10位有效数字

long double

10位有效数字

后缀l或L

& 和 * 含义

int i = 42;

int &r = i ;

&紧随类型名出现，为声明的一部分，r为一个引用

int *p

*紧随类型名出现，为声明的一部分，p为一个指针

p = &r

&出现在表达式中，是一个取地址符

*p = i

* 出现在表达式中，是一个解引用符

int &r2 = *p

&是声明的一部分， *是一个解引用符

复合类型

&引用 ： 为对象起了另外一个名字，不是对象

不能定义引用的引用

int &refVal = ival

refVal指向ival,是ival的另外一个名字

*指针：指向另外一种类型， 为对象

允许赋值和拷贝；无须在定义时赋初始值

使用指针应该初始化所有指针

4种指针值

指向一个对象

指向紧邻对象所占空间的下一个对象

空指针，不指向任何对象

int *p1 = nullptr

C++11标准

int *p2 =0

#include cstdlib int *p3 = NULL;

无效指针，上述情况之外的其他值

void* 指针可以存放任意对象的地址

但不知道其对象类型，不能直接操作

面对一条复杂的指针或引用声明语句，从右往左读有助于理解

const限定符

初始化

const int bufSize = 512

int i = 42; const int vt = i;

const 引用 （常量引用）

不能用作修改其绑定的对象

指向常量的指针

不能改变其所指对象的值

const 指针

必须初始化，存放指针中的地址不能改变

const double pi = 3.14

const double *cptr = & pi

顶层const： 指针本身是个常量，更一般地，任意的对象是常量

底层const 指针所指的对象是一个常量

拷贝时，两对象都具有相同的底层const资格，或数据类型滥够转换

常量表达式 和constexpr

常量表达式： 值不会改变，编译过程就能得到计算结果

若认为是一个常量表达式，就可以使用constexpr声明

C++11特性

处理类型

类型别名—某种类型的同义词

typedef

typedef double wages; //wages是double 的同义词

别名声明 using

using SI = Sales_item； //SI是Sales)item的同义词

auto 编译器通过初始值来推算变量的类型

一般会忽略顶层的const,保留底层的const

decltype 选择并返回操作数的数据类型

decltype(f()) sum = x, //sum的类型为函数f的类型

注意变量名加不加括号区别

decltype((var)) 结果永远是引用

decltype(var) 只有var本身是引用才是引用

Chap3 字符串、向量、数组

命名空间的using声明

using namespace::name;

using std::cin

using namespace std ;

每个名字都需要独立的using声明

头文件不应包含using声明

string：可变长字符串

初始化 已包含： #include<string> using std::string

直接初始化—不使用等号

string s6("hiya")

string s7(10,'c'); // s7的内容为ccccccccccc

string s2(s1) ; //s2为s1的副本

拷贝初始化—使用等号

string s2 = s1;

string s3 = "values"

string s4 = string(10,'c')

字符串字面值和string是不同的类型

读取一整行—getline

string line; getline（cin， line）

返回string对象type

line.size（）

返回string::size_type的类型。无符号整形数

处理每一个字符—基于范围的for

for (declaration : expression) statement

下标运算符[],类型为 string::size_type

从0 到 s.size()-1

vector: 类型相同的对象的集合（容器）

不存在包含引用的容器

初始化 已包含： #include<vector> using std::vector;

vector<Type> vector_name;

添加元素

vec_name.push_back(var)

不能使用下标形式添加

索引与string类似，但类型为 vector<Type>::size_type 而非 vector::size_type

迭代器iterator：访问容器中的元素或在元素之间移动

若容器为空，则begin和end返回的是同一个迭代器，都为尾后迭代器

迭代器的运算符

*iter 返回迭代器iter所指元素的引用

iter->mem 解引用iter并获取该元素的名为mem的成员，等价于（*iter）.mem

++iter 令iter指示容器上一个元素

--iter 令iter指示容器的下一个元素

iter1 == iter2 两个迭代器指示为同一个元素或是同一个容器的尾后迭代器则相等

类型

iterator 对象可读可写

const_iterator 对象能读取但不能修改

迭代器的运算

iter + n

iter - n

iter1 += n

iter -= n

iter1 - iter2

没有定义加法！

> 、 >=、 < 、 <=

数组array : 类型相同对象的容器

数组由内到外阅读容易理解

与vector不同：数组大小固定不便，不能随意增加元素，运行时性能较好

使用for语句或下标访问，下标的类型为size_t

size_t为机器相关的无符号类型，设计成足够大来表示内存中任意对象大小

数组和指针

许多用到数组名字的地方，编译器会自动将其替换成一个指向数组首元素的指针

一些情况下，数组的操作实际是指针的操作

标准库函数begin()和end()

```
int *beg =  begin(ia);  //指向ia首元素的指针
int *last = end(ia) //指向ia尾元素下一个元素的指针
```

指针运算

从一指针加上(减去）某整数值，结果仍是指针

两个指针相减的结果是它们之间的距离

类型为ptrdiff_t（和size_t一样为机器相关类型)但为带符号类型

多维数组： 其实是数组的数组！

计算数组维度： sizeof(array)/sizeof(array[0])

访问多维数组

范围for处理

```
for (int (&clo)[10] : ia){        for (int &row : clo){            cout<<row<<"  "  ;        }    }
```

下标引用size_t类型

```
for (size_t i = 0; i != 3; ++i)   for (size_t j = 0; j != 4; ++j) cout << arr[i][j] << " ";
```

使用指针

```
for (int(*row)[4] = arr; row != arr + 3; ++row)   for (int *col = *row; col != *row + 4; ++col) cout << *col
<<" ";
```

Chap4 表达式

处理复合表达式时，如果改变了某个运算对象的值，在表达式的其他地方不要再使用这个运算对象

表达式左值和右值

左值lvalue

用的是对象的身份(在内存中的位置)

右值rvalue

用的是对象的内容(值)

递增和递减运算符

前置版本：首先将运算对象加一，后将改变的对象本身作为左值返回

后置版本： 也将对象加一，但返回的是对象的原始版本的副本

```
cout << *iter++ << endl;
```

成员访问运算符

. 点运算，获取类对象的一个成员

-> 箭头运算符 p->size() 等价于 (*p).size()

条件运算符

```
cond ? expr1 : expr2
```

首先求cond的值，条件为真对expr1求值并返回该值，否则对expr2求值并返回

```
cout << ( (grade<60) ? "fail" : "pass"  );
```

位运算符

～ 位求反

<< 左移

>> 右移

&位与
^位异或
|位或
返回一条表达式或一个类型名字所占的字节数 sizeof
类型转换
可以互相转换的在C++中是关联的
算术转换
隐式转化
整型提升： bool、char、signed char、unsigned char、short 和unsigned short等，若能将所有的值存在int中，就提升为int；否则提升为unsigned int
较大的char类型，提升为int、unsigned int、long、unsigned long、long long和unsigned long long 中最的类型
显式转换
形式： cast_name<type>(expression); type为转换的类型，expression为转换的值; cast_name 分为：
static_cast 适用于任何具有明确定义的类型转换，只要不包含底层const
const_cast 只能去掉对象的底层const
reinterpret_cast 为对象的位模式提供较低层次上的重新解释
dynamic_cast 和继承及运行是类型识别一起使用
旧式的转换（不推荐）
type (expr)　//函数形式
(type) expr;　// C语言风格
执行显式转换类型，就可以去掉较大的算术类型赋值为较小的模型的警告

Chap5 语句

条件语句
if语句—根据条件决定控制流
switch，计算整型表达式的值，然后由值从几条执行路径中选择一条
case 标签： 必须为整型常量表达式！
case 3.14 ：　//错误，不是整型
int ival = 42；
case ival：　// 错误，不是常量
default标签：所有case标签没匹配上将执行
任何case标签的值不能相同
注意case中的每一个break，没写break也应该加注释说明逻辑关系
注意: 不能在一个case里面声明另一个case也需要的变量，应在switch外声明
迭代语句
while （condition） statement;
for
传统for for(init-statement; condition; expression) statement;
省略condition 等于 条件永远为true
省略expression，则需要在statement中修改
范围for for（declaration : expression) statement;
等价于 for (auto beg = v.gegin(), end = v.end() ; beg != end; ++beg) {statement)
vector对象不能通过范围for修改

do statement while (condition);

跳转语句

break 只在迭代语句和switch中使用，终止最近的迭代并从这些语句后的第一条语句开始继续执行

continue 只适用于迭代语句，终止最近循环的迭代并立即开始下一次迭代

goto label； 无条件跳转到同一函数的另一条语句 不推荐使用！！！

return

异常处理

异常中断了程序的正常流程，鲁棒性

鲁棒性：（Robustness）是指一个计算机系统在执行过程中处理错误，以及算法在遭遇输入、运算等常时继续正常运行的能力。

throw表达式： 表示遇到了无法处理的问题，或称throw引发了异常

try 语句块 处理异常

try { program-statement } catch { exception-declaration} { handler-statement } catch { exception-declaration} { handler-statement } //.....

标准异常，定义在4个头文件中

exception 最通用的异常类，只报告发生，没其他额外信息

stdexcept 常见的异常类

exception 最常见的问题

runtime_error 运行时才可以检测的问题

range_error 运行时错误： 生成的结果超出了值域范围

overflow_error 运行时错误: 计算上溢

underflow_error 运行时错误:计算下溢

logic_error 程序逻辑错误

domain_error 逻辑错误：参数对应的结果值不存在

invaild_argument 逻辑错误：无效参数

length_error 逻辑错误：试图创建一个超出该类型范围最大长度的对象

out_of_range 逻辑错误：使用一个超出有效范围的值

new 定义了 bad_alloc异常类型

type_info 定义了bad _cast异常类型

Chap6 函数

名字有作用域，对象有生命周期

函数基础

形参：出现在函数定义的地方，规定了一个函数所接受数据的类型和数量

实参： 出现在函数调用的地方，数量和形参一样多

局部变量： 形参加上函数体内部定义的变量

自动变量： 只存在于块执行周期的对象为自动变量，如形参

局部静态变量： static类型，从程序执行路径第一次经过时候初始化，一直存在到程序终止才被销毁

参数传递

传值参数void f(int j)

参数的值被拷贝到形参，在函数f内部对形参做的改动不会影响实参的值

传引用参数void f(int &j)

形参是实参的别名，形参绑定到初始化它的对象，若改变了形参的值，也改变了对应实参的值

const形参和实参

常量引用 不允许改变形参的值，多使用

非常量引用

函数名前加了const修饰符，意味着该函数返回的值只能是读取，而不能被修改

函数后面加上const，说明函数的成员对象是不可修改的

需要更改加上mutable

数组形参

谨记： 数组不能被拷贝以及数组通常挥自动转为指针类型

```
void print(const int*);
void print(const int[]);
```

多维数组

声明指向含有10个整型的数组的指针：

```
void print( int (*matrix)[10], int rowSize) {/*...*/}
```

注意*matrix两端的括号

等价于 void print (int matrix[][10], int rowSize) {/*...*/}

main处理命令行选项

int main (int argc, char *argv[]）{....}

等价成int main (int argc, char **argv）{....}

argc 表示数组中字符串的数量，也就是用户输入参数的数量

argv表示一个数组，其中argv[0]是程序的名字，所以可选实参从argv[1]开始

可变形参initializer_list<Type>

```
void error_msg(initializer_list<string>  i1) {....}
```

递归函数：调用自己本身

main函数不能递归调用

返回数组指针

类型别名

```
typedef string arr[10]; arr& func();
```

尾置返回类型

```
auto func () -> string(&) [10];
```

decltype关键字

```
string str[10]; decltype(str) &func();
```

函数重载

同一作用域内几个函数名称相同但是形参列表（数量或类型）不同

拥有顶层const的形参无法与没有顶层const的区分开

函数匹配

找到与实参最佳匹配

没找到实参匹配，编译器发出无匹配的错误

多个可以匹配，但都不是最佳选择，也会发生错误，称为二义性调用

重载对作用域的性质没什么改变

若在内层作用域声明名字，将隐藏外层作用域声明的同名实体

特殊用途语言特性

默认实参

某些形参多次调用都赋予同一个相同的值

string screen（int ht = 24， int wid = 89, char backgrnd = ' ');

可以为一个或多个赋予默认值，注意：一旦某个形参赋予了，后面所有的形参都必须也赋予
让不经常使用默认值的形参在前面！

内联函数

可避免函数调用的开销
在函数返回类型前面加上 inline
通常定义在头文件中

constexpr 函数

用于常量表达式，会被隐式声明为内联函数
通常定义在头文件中

调试帮助

assert预处理宏

assert (expr)

首先对expr求值，若为0，assert输出信息并终止程序执行；若为真，assert什么都没做

NDEBUG预处理变量

定义了NDEBUG，则assert什么都不做

函数指针

bool lengthCompare(const string &, const string &); bool (*pf) (const string &, const string &);

pf 两端的括号不可少

Chap7 类

在函数内部不需要使用成员访问运算符，因为任何对成员的访问都被直接看作this的隐式引用

this： 隐式的值，指向函数调用者的对象

构造函数

= default， 要求编译器生成构成函数，不管类是否已经有了其他构造函数
初始化对象的一种特殊成员函数
没有返回类型，不能声明为const
方式一：Sales_data (const std::string &s, …) : bookNo(s), …. { }
方式二：初始值列表： Sales_data (const std::string &s, …) { bookNo = s…. }
成员是const或引用则必须将其初始化
注意成员初始化顺序，最好与成员声明的顺序保持一致
委托构造函数：初始值列表只有一个入口，指定类的另一个构造函数执行初始化操作
默认构造函数： 当没有提供任何实参时使用的构造函数
隐式转换构造函数

类类型转换只允许一步(非多步转换)，且不总是有效的
抑制构造函数定义的隐式转换： 声明为explicit

聚合类：可以直接访问其成员，但注意顺序

成员都是public，没有定义任何构造函数，没有类内初始值，没有基类和virtual函数

字面值常量类： 数据成员都是字面值类型的聚合类，至少提供一个constexpr构造函数

访问说明符

public 成员在整个程序内可被访问，public成员定义类的接口
private 成员可被类的成员函数访问，但不能被使用该类的代码访问 ，private封装了类的实现细节
友元friend

为非成员接口函数提供了访问其私有成员的能力

class 和 struct的区别

class 默认访问权限为private

struct 默认成员为public

三大特性：封装、继承、多态

封装： 分离类的实现与接口，从而隐藏了类的实现细节

（通过把实现部分设为private）

其他特性

类成员

定义类型在类中的别名(也存在访问限制)

```
typedef std::string::size::size_type pos;
```

令成员成为内联函数

隐式内联： 定义在类内部的成员是自动内联的

显示内联： 函数在类的外部，并定义之前显示指定inline

支持成员函数重载

可变数据成员

在变量声明中加入mutable

永远不是const

类类型

每个类定义唯一的类型，即使成员完全一样也是不同的类型

类的声明：

```
class Screen； // 也称为向前声明，是一个不完全类型
```

可以指向或引用不完全类型的对象，但不能创建

友元

类可以把其他类定义成友元，也可以把其他类的成员函数定义为友元

定义类成员函数为友元的组织结构

如clear是window_mgr的成员，是Screen的友元

1. 先定义Window_mgr类，其中声明clear函数但不能定义
2. 定义Screen函数，指明clear是其友元
3. 定义clear函数

作用域

每个类会定义自己的作用域，一个类就是一个作用域

类的定义步骤

先编译成员的声明

直到类全部可见之后彩编译函数体

名字查找过程：

首先在名字所在的块寻找，只考虑使用之前的声明

上一步无果，则继续寻找外层作用域

最终没找到，报错

静态成员：声明语句之前带有关键字的static成员

不是任何单独对象的组成部分，由该类的全体对象所共享

与类本身关联，而不是与类的具体对象关联

可以作为默认实参。普通数据成员不可以

不能在类的内部初始化，除了静态常量成员

part2 C++标准库

chap8 IO库

IO类

头文件

iostream

istream, wistream 从流读取数据，如cin

ostream, wostream 向流写入数据， 如cout、cerr

iostream，wiostream 读写流

fstream

ifstream， wifstream 从文件读取数据

ofstream， wofstream 向文件写入数据

fstream， wfstream 读写文件

sstream

istringstream, wistringstream 从string读取数据

ostringstream， wostringstream 向string写入数据

stringstream，wstringstream 读写string

继承机制可以让忽略不同类型的流之间的差异

例如，ifstream继承自 istream， 可以像使用istream一样使用ifstream

IO对象没拷贝或赋值，通常以引用方式传递和返回流

i.e ofsteam print（ofsteam）； //错误，不能初始化ofstream参数

条件状态：访问或操纵流

strm:: iostate 机器相关类型，提供表达条件状态的完整功能

strm:: badbit 指出流已崩溃

strm::failbit 指出IO操作失败了

strm:: eofbit 指出流到达了文件结束

strm::goodbit 指出流未处于错误状态。此值保证为0

s.eof() 流s的eofbit置位，返回true

s.bad() 、s.fail() 、 s.good() 将xx置位，返回true

s.clear() s.clear(flags) s.setstate(flags) 复位或置位，返回void

s.rdstate() 返回当前条件状态，返回类型为 strm::iostate

输出缓存

刷新缓存区

endl 换行并刷新

flush 刷新但不输出任何额外的字符

ends 刷新并插入一个空字符

程序崩溃也不会刷新缓冲区，调试时注意!

关联输入和输出流：每次输入先刷新输出流

如 cin.tie(&cout) ;

文件输入输出fstream

fstream fstrm(s, mode) 打开名为s的文件，并且按照mode模式

fstrm.open(s)

返回void

fstrm.is_open()

返回bool值，指出文件是否成功打开且尚未关闭

fstrm.close()

返回void

文件mode模式

in 以读方式打开

out 以写方式打开

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 二进制方式进行IO

参考网站 https://github.com/xuelangZF/CS_Offer/blob/master/C%2B%2B/InputOutput.md

string流

chap9 顺序容器

类型

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

若在中间插入，又需要快速访问，可以使用vector后sort排序

deque 双端队列。支持快速随机访问。从头尾位置插入/删除速度很快

list 双向链表。只支持双向顺序访问。在list中任何位置插入/删除操作速度很快

forward_list 单向链表。只支持单向顺序访问。在链表任何位置插入/删除操度都很快

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类似，但专门用来保存字符。随机访问快，尾部插入/删除速度快

不确定使用那种容器，则只使用 vector/list公共操作：使用迭代器，不使用下标，避免随机访问。

类型别名：

iterator 此容器类型的迭代器类型

const_iterator 可以读取，但不能更改因素的迭代器类型

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；与value_type& 的含义相同

const_reference 元素的const左值类型（即， const value_type& ）

容器定义和初始化

C 为容器类型，c 为容器名称

C c； 默认构造函数。若C为array，则c中元素默认初始化，否则为空

C c1(c2) 等价于 C c1=c2。 c1初始化为c2的拷贝。

c1和c2必须相同类型（容器类型和元素类型都相同），对于array还必须相同大小

C c{a, b, c....} 等价于 C c = {a,b,c.....} c初始化为列表中元素的拷贝

列表中的元素类型和C的元素类型相同。对于array，列表中的元素数目都必须小于或等于array的大小

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

范围中元素的类型必须于C的元素类型相容！（array不使用）

C seq（n） seq包括n个元素，这些元素进行了值初始化；构造函数为explicit的

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，t） seq包括n个初始化为t的元素

标准库array

具有固定大小

如： array< int,42>

使用array类型，同时指定元素类型和大小

array<int, 10> ::size_type i;

可以对array进行拷贝和对象赋值操作，不像内置数组

赋值和swap

c1 = c2; // c1内容为c2 的拷贝

c1 = { a,b,c}

swap (c1 ,c2) 等价于c1.swap(c2)

交换c1,c2的元素（只是交换容器的内部数据结构。类型必须相同。不对元素进行拷贝，删除或插入）常数时间内完成

但对于array，swap会真正交换它们的元素。所需时间与元素数目成正比

assign （不适合关联容器和array）

seq.assign(b,e)；

将seq的于元素替换为迭代器b和e所表示范围的元素。迭代器不能指向seq中的元素

seq.assign(i1)

将seq中的元素替换为初始化列表i1中的元素

seq.assign(n,t)

替换为n个值为t的元素

容器大小操作

size返回容器中元素的数目

empty当size为0时返回true

max_size返回一个大于或等于该类型所能容纳的最大元素数的值

关系运算符

所有容器类型支持 ==和！=

除了无序关联容器外都支持关系运算符（>、>=. <、<=)

顺序容器操作

添加元素

操作会改变容器大小；array不支持这些操作

向vector, string或deque插入元素会使所有指向容器的迭代器、引用和指针失效

c.push_back(t); 等价于c.emplace_back(args)

//在c的尾部创建一个t或由arg创建的元素。返回void

c.push_front(t) 等价于c.emplace_front(args)

//在c的头部创建一个t或由arg创建的元素。返回void

c.insert(p,t) 等价于c.emplace(p, args)

在迭代器p指向的元素之前创建一个t或由arg创建的元素。返回指向新添加的元素的迭代器！！！

c.insert(p, n, t)

在迭代器p指向的元素之前插入n个t。返回指向新添加的第一个元素的迭代器！若n为0，则返回p

c.insert(p, b, e)

在迭代器p指向的元素之前插入迭代器b和e指定范围的元素。b和e不能指向c中的元素。返回指向新添加的第一个元素的迭代器！若范围为0，则返回p

c.insert(p,i1)

在迭代器p指向的元素之前插入元素值列表i1。返回指向新添加的第一个元素的迭代器！若列表为空

，则返回p

访问元素

c.back() 返回c中尾元素的引用！若c为空，函数行为未定义

不适用forword_list

c.front() 返回c中首元素的引用！若c为空，函数行为未定义

auto &v = c.back(); v = 1024; //改变c中的元素

auto v2 = c.back(); v2 = 0; // 未改变c中的元素

c[n] 返回c中下标为n元素的引用，n是一个无符号整数。若n>= c.size(),则函数行为未定义

c.at(n) 返回下标为n的元素的引用。若下标越界，则抛出out_of_range异常

访问成员函数返回的是引用

提供快速随机访问的容器业提供下标运算符

删除元素

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所指定的元素，返回一个指向被删元素之后元素的迭代器

c.erase(b，e) ; 删除迭代器b和e所指定范围的元素，返回一个指向最后被删元素之后元素的迭代器

c.clear() 删除c中所有的元素，返回void

特殊forward_list操作

1st.before_begin() ; 或lst.cbefore_begin();

返回指向链表首元素之前不存在的元素的迭代器。此迭代器不能解用.

lst.insert_after(p,t) 或lst.insert_after(p,n,t)

在迭代器p之后的位置插入元素。t是一个对象，n是数量

1st.insert_after(p, b,e)或 lst.insert_after(p, i1)

b,e是一对迭代器，i1是一个花括号列表

emplace_after(p, args)

使用args在p指定的位置之后创建一个元素

lst.erase_after(p) 或lst.erase_after(b,e)

删除p所指的位置或从b到（不含）e之间的元素。返回一个指向被删元素之后元素的迭代器

改变容器的大小

不适用于array

c.resize(n) 或c.resize(n ,t)

vector 实现方式等

连续存储， 每个元素紧挨前一个元素

size指的是保存元素的数目，capacity指的是最多可以保存多少元素能力

vector, string大小操作

c.shrink_to_fit( ); //将capacity减小为size大小请求（不一定实现）

c.capacity( ) ; // 不重新分配内存下，c可以保存多少元素

c.reserve(n) ; // 分配至少能容纳n个元素的内存空间

string额外操作

构造string的其他方法

string s (cp , n) ; //s是cp指向的数组前n个字符的拷贝

string s (s2, pos2); //从s2下标pos2开始字符的拷贝

string s (s2, pos2,len2)； //从下标pos2开始len2字符的拷贝

修改string

顺序容器的assign、insert、erase、赋值操作

接受下标的insert和erase版本

接受c风格字符数组的insert和assign版本

append和replace函数

搜索操作

s.find(args)

s.find_first_of(args)

s.find_first_not_of(args)

compare

s.compare(s2);

s.compare(pos1, n1, s2);

s.compare(pos1, n1, s2, pos2, n2);

s.compare(cp);

s.compare(pos1, n1, cp);

s.compare(pos1, n1, cp, n2);

数值转换

to_string(val);

//转换成整数

//p是第一个非数值字符的下标，默认值为0

//b是基数，默认值为10

stoi(s, p, b);

stol(s, p, b);

stoul(s, p, b);

stoull(s, p, b);

//转换成浮点

stof(s,p)

stod(s,p)

stold(s,P)

容器适配器

标准容器三个顺序适配器

stack

默认基于deque

s.pop()

s.push(item)

s.emplace(args)

s.top()

队列适配器

queue

默认基于deque

priority_queue

默认基于vector

q.pop()

q.front()

q.back()

q.top()

q.push(item)

q.emplace(args)

chap10 泛型算法

泛型算法永远不会执行容器的操作，只会运行在迭代器之上 或者说算法根本不该知道容器的存在

分类

只读算法

对于只读而不改变的算法，最好使用 cbegin()和cend （）

计数器count(iterator1, iterator2, val)

匹配find(iterator1, iterator2, val)

求和 accumulate (iterator1, iterator2, val) , val 决定了使用那个加法运算符和返回值类型

确定两个序列是否保存相同的值 equal (roster1.cbegin(), roster1.cend(), roster2.cbegin())

相同为true，不相等则为false

假定第二个序列至少与第一个序列一样长

写容器算法

fill (iterator1, iterator2, val) , 将val 赋值给指定范围的元素

fill_n (dest, n , val) 将val 赋值到从dest开始的n个元素

拷贝算法 copy (iterator1, iterator2, vector2) 将指定范围的元素拷贝到目标序列中

重排容器算法

排序sort(iter1, iter2) 利用<运算符实现

unique (iter1, iter2) 将相邻重复项移到队列最后，返回一个指向不重复值范围末尾的迭代器

定制操作

谓词：返回可以转换bool类型值的函数。

泛型算法经常用来检测元素。标准库使用的是一元(接受一个参数）或二元（接受两个参数）的

lambda表达式（匿名函数）：可调用的代码单元，类似一个未命名的内敛函数

[capture list] (parameter list) -> return type { function body}

capture list(捕获列表）为所定义局部变量的列表（通常为空）

捕获列表

[] 空捕获列表。只有捕获变量后才能使用

[name] 默认情况下，列表中的变量都被拷贝

[&] 隐式捕获列表，采用引用捕获

[=] 隐式捕获列表，采用值捕获

[&, identifer_list]

[= , idengtifer_list]

可变lambda （可以改变被捕获的变量的值）

在参数列表首加上mutable

参数绑定bind

auto newCallable = bind(callable, arg_list) 其中，newCallable 是一个可调用对象，arg_list为参数列表

arg_list可以包含_n 形式的名字

_n定义在 std::placeholders的命名空间中，可以使用 using namespace namespace_name 来声明

迭代器

种类

插入迭代器

back_inserter创建一个使用push_back的迭代器

front_inserter 使用push_front的迭代器

inserter (list, list.begin()) ;

使用insert的迭代器。 接受第二个参数，指向给定容器的迭代器。元素插入迭代器表示的元素之前

流迭代器

读取输入流 istream_iterator（可为定义>> 创建）

eg: istream_iterator<T> in(is) ； in从输入流is读取类型为i的值

istream_iterator<T> end；

*in 返回从流读取的值

in-> mem = （*in).mem

++ in, in++

向输出流写数据 ostream_iterator(可为定义<<创建）

ostream_iterator<T> out(os, d)

out = val

*out, ++out, out++; 不对out做任何操作，返回out

反向迭代器

向后移动，rbegin（）或 rend()

移动迭代器

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，多遍读写；支持全部迭代器运算

chap11 关联容器

类型

按关键字有序保存

map 关联数组，保存关键字-值对

set 关键字即值，即只保存关键字的容器

multimap 关键字可重复的map

multiset 关键字可重复的set

无序集合

unordered_map 用哈希表组织的map

unordered_set 用哈希表组织的set

unordered_multimap

unordered_mulitset

关键字key类型要求

需要遵循严格弱序

可以把严格弱序看为“小于等于”

pair类型

定义在头文件utility中

保存两个数据类型，不要求类型一样

数据成员是public

map的元素是pair

定义pair

pair<T1,T2> p(v1,v2)

pair操作

p.first / p.second 返回名为first的数据成员

p1 relop p2; 关系运算利用< 中字典序来排序

p1 ==p2 ; p1 != p2

关联容器的操作

类型别名

key_type； 此容器的关键字类型

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>

关联容器迭代器

set的迭代器是const的

通常不对关联容器使用泛型算法，因为set类型中元素为const的，map元素是pair，其第一个成员是const的

添加元素

c.insert(v) c.emplace(args) c.insert(b,end) //b和e代表 迭代器范围

返回类型pair< map<string, vector<int>>:: iterator, bool>

删除元素

c.erase(k) c.erase(p) c.erase(b,e)

返回实际删除元素数量

无序容器

在存储上组织为一组桶，每个桶保存零个或多个元素

使用hash而不是比较操作赖存储和访问元素，性能依赖于hash函数的质量

chap12 动态内存

智能指针 定义memory中

share_ptr 允许多个指针指向同一个对象

最安全的分配方法是使用make_shared函数

share_ptr<int> p3 = make_share<int>(42);

自动销毁所管理的对象

自动释放相关联的内存

拷贝一个share_ptr会递增其引用数

将一个share_ptr赋予另一个share_ptr，递增右边的而递减左边的。

当一个share_ptr的引用数为0时，所指向的对象会被自动销毁

unique_ptr 指针独占所指向的对象

weak_ptr 伴随类，一种弱引用，指向share_ptr所管理的对象

直接管理内存

new分配内存

string *p1 = new string(10,'9');

auto p1 = new auto（obj);
const int *pci = new const int(1024);

内存耗尽

Int *pi = new int; / /分配失败，new抛出std:: bad_alloc

定位new placement new

int *pi2 = new (nothrow) int; //分配失败，new返回一个空指针

delete释放new分配的内存

必须指向动态分配的内存，或是一个空指针

直接管理内存常见问题，应坚持只用智能指针

忘记delete内存,造成“内存泄漏”

使用已经释放掉的对象

同一块内存释放两次，破坏自由空间

unique_ptr

定义一个unique_ptr时，需要将其绑定到一个new返回的指针上

声明： unique_ptr< T> u1

unique_ptr 不支持普通的拷贝或赋值，需要调用release 或reset将指针的所有权转移

u.reset() 释放u所指的对象

u.release() u放弃对指针的控制权，返回指针，并将u置空

weak_ptr

不控制所指向对象生存期的智能指针，指向share_ptr管理的对象

若最后一个指向对象的share_ptr被销毁，即使有weak_ptr指向对象，也还是会释放

part3 类设计

chap13拷贝控制

拷贝控制操作

定义用同类型的另一对象初始化本对象做什么

拷贝构造函数 copy constructor

合成拷贝构造函数

编译器为未显示定义对应的构造函数的类生成的拷贝或移动构造函数 的版本

拷贝/直接初始化

直接初始化,使用普通的函数匹配

string dots(10, ' . ');

string s(dots)

拷贝初始化，将右值拷贝给对象，有时进行类型转换

string s2 = string（10，' 9');
string s2 = dots;

拷贝构造函数的参数必须是引用类型

移动构造函数

定义将一对象赋予同类型的另一对象做什么

拷贝赋值运算符

重载赋值运算符

重定义了运算符应用于类类型的对象时的含义

Foo& operator = (const Foo&); // 赋值运算符

拷贝赋值运算符是一种重载 赋值运算符

左侧运算对象绑定到隐含的this参数，右侧则为所属类类型的

定义该类型对象销毁做什么

析构函数 destructor

执行于构造函数相反操作

名字由波浪号接类名组成，没有返回值，也不接受参数（不代表函数体为空）

```
class Foo { public:          ~Foo();   // 构析函数 };
```

一个类只能有一个，因为不能重载

完成之后，成员会自动销毁

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

对象移动

右值引用

绑定到右值的引用

通过&&而非&来获得右值引用

重要性质：只能绑定一个将要销毁的对象

左值引用、右值引用区别

左值持久；变量是左值

右值短暂；字面变量或临时对象

标准库move函数

```
int &&r3 = std::move(rr1);
```

move调用告诉编译器：将一个左值当作是右值处理，也即承诺除了对rr1赋值和销毁外，不再使用rr1

只有确定使用移动函数不会出现其他的问题再使用

移动构造函数和移动赋值运算符

```
eg:   HasPtr::HasPtr(HasPtr &&p) noexcept : ps(p.ps), i(p.i) {  p.ps = 0;  std::cout << "call move constructor" <
< std::endl; }
```

noexcept 通知构造函数不抛出任何异常

移动迭代器

生成的迭代器会在解引用时得到一个右值引用

chap14 重载运算与类型转换

基本概念

具有特殊名字的函数：其名字由关键字operator和其后要定义的运算符号共同合成。

也包含返回类型、参数列表以及函数体。

参数数量应该和该运算符作用的运算数量一样多

运算符可被重载?

大部分可以

通常不应该重载 逗号、取地址、逻辑与、或运算符

不可以： ::　　.*　　.　　?:

输入输出运算符

输入输出运算符必须是非成员函数

重载输出运算符<<

通常第一个形参是非常量ostream的引用，第二个形参是需要打印类型的常量引用

输出运算符尽量减少格式化操作

重载输入运算符

输入运算符必须处理输入失败的情况，而输出运算符不需要

当读取操作发生错误时，输入运算符应该负责从错误中恢复

算术和关系运算符

定义为非成员函数

一般需要改变运算对象的形态，所以都是常量的引用

若定义了算术运算符，一般定义一个对应的复合赋值运算符。所以最有效的方法就是使用复合赋值来定义算术运算符

相等和不等运算符

若类在逻辑上有相等性的含义，则应该定义operator==

！= 可以直接调用 == 来实现

return ！(lhs == rhs);

关系运算符

下标运算符

下标运算符必须是成员函数

一般定义两个版本

一个返回普通引用

另一个是类的常量，并返回常量引用

递增递减运算符

建议为成员函数

应该同时定义前置版本和后置版本

为与内置版本保持一致，前置运算符应该返回递增或递减后对象的引用

前置版本

```
class StrBlobPtr& StrBlobPtr::operator++ () {  // 注意形参列表，没有形参  check( curr, " increment past nd of the StrBlobPtr");  ++ curr;  //将curr当前状态下向前移动一个元素  return *this; }
StrBlobPtr P(a1);  //p指向a1中的vector P.operator++（）；  // 调用前置版本的operator++
```

后置版本

接受一个不被使用的形参，用来区分前置版本和 后置版本

递增对象之前应该记录对象的状态，用于返回原值

```
class StrBlobPtr& StrBlobPtr::operator++ (int) {  // 注意形参列表，有一个不使用的 int  StrBlobStr ret = *this ;  // 保存当前的值  ++ *this;  // 调用前置版本，需要检测递增的有效性  return ret;  // 返回保存的值 }
StrBlobPtr P(a1);  //p指向a1中的vector P.operator++（0）；  // 调用后置版本的operator++
```

成员访问运算符

成员函数

```
class StrBlobPtr { public：  std::string & operator*() const ;  // 不会改变对象的状态  { auto p = check (curr, "dereference past end");  return (*p)[curr] ;  // (*p)是对象所指的vector }
```

函数调用运算符

像使用函数一样使用该类的对象

类若定义了调用运算符，则该类的对象称为函数对象

```
eg: operator() (int ) {  }
```

lambda是对象函数

标准库定义了一组表示运算符的类，这些类型在functional头文件中

C++ 可以调用的对象

类型

函数

函数指针

lambda表达式

bind创建的对象

重载了函数调用运算符的类

可以使用模板 function 来解决类型不匹配问题

详见 calculator

类类型转换与运算符

类型转换运算符

operator type() const { function body};

type 表示某种类型

避免有二义性的类型转换，即尽可能限制那些“显然正确”的非显式构造函数

类型转换优先级

1. 精确匹配
2. const 转换。
3. 类型提升
4. 算术转换
5. 类类型转换

chap15 面向对象程序设计

核心思想

数据抽象： 将类的接口与实现分离

继承: 定义相似的类型并对其相似关系建模

动态绑定： 一定程度上忽略相似类型的区别，以统一的方式使用它们的对象

访问控制与继承

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。

定义基类和派生类

类派生列表

class Bulk_quote : public Quote { ...} ; // Bulk_quote 继承自Quote

移动赋值运算符

访问符可以是 public, private, protected

防止继承的发生

class Base final {....} ;

类的后面加上关键字final

派生类到基类的类型转换

派生类中含有与其基类对应的组成部分

派生类对象向基类的引用或指针

派生类的构造函数

应该每个类都有自己的成员初始化过程

但应该遵循基类的接口，

静态类型和动态类型

静态类型在编译时总是已知的，它是变量声明时的类型或表达式生成的类型。

动态类型则是变量或表达式表示的内存中的对象的类型。动态类型直到运行时才可知

虚函数

一旦某个函数被声明为虚函数，则所有的派生类中都是虚函数

final和override说明符

override 的含义是重写基类中相同名称的虚函数

final 是阻止它的派生类重写当前虚函数

抽象基类

称带有纯虚函数的类为抽象类

不能生成对象

纯虚函数

基类中实现纯虚函数的方法是在函数原型后加“=0”

基类中声明的虚函数，它在基类中没有定义，但要求任何派生类都要定义自己的实现方法。

目的在于，使派生类仅仅只是继承函数的接口。

访问控制与继承

成员继承类型

公有继承(public)

保护继承(protected)

私有继承(private)

成员的访问权限影响因素

基类中该成员的访问说明符

决定派生类的成员是否可以访问基类

派生类列表中的访问说明符

控制派生类用户（包括派生类的派生类）对于基类成员的访问权限

改变个别成员的可访问性 using

继承中的类作用域

构造函数和拷贝控制

基类通常应该定义一个虚析构函数

但可以不需要拷贝和赋值操作

容器和继承

基类通常应该定义一个虚析构函数。

在容器中应该放置（智能）指针而非对象

chap16 模板和泛型编程

OOP 能处理类型在程序运行之前都未知的情况

泛型编程中，编译时就能获知类型了

定义模板

函数模板

示例

```
template <typename T> int compare(const T &v1, const T &v2) {    if (v1 < v2) return -1;    if (v2 < v1) return 1;    return 0; }
```

一个函数模版就是一个公式，可用来生成针对特定类型的函数版本。

类型参数关键字typename可以使用class来替代

注意： 应该尽量编写与类型无关的代码

如上例中就不要使用 >

注意inline static 等位置

`template <typename T> inline T foo(T, unsigned int*);`

函数模板和类模板成员函数的定义通常放在头文件中

模板程序应该尽量减少对实参类型的要求

大多数编译错误再实例化期间才可以报告

模板直到实例化才会生成代码

类模板

类模版是用来生成类的蓝图的

函数模版的不同之处是，编译器不能为类模版推断模版参数类型。为了使用类模版，必须在模版名后的尖括号中提供额外信息。

一个类模版的每个实例都形成一个独立的类。

template <typename T> class Blob { .....} // 详见blob.h

模板参数

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。

声明为 typename 的类型参数和声明为 class 的类型参数 没有 什么区别

成员模板

一个类中的成员函数本身是模板

成员函数不能是虚函数

普通类的成员模板

类模板的成员模板

同时提供类模板（前）和成员模板参数（后）

控制实例化

一个类模板的实例化定义中，所用类型必须能用于模板的所有成员函数

效率与灵活性

模板实参推断

从函数实参来确定模板实参的过程

std::move 是使用右值应用的函数模板

可变参数模块

参数包 -可变数目的参数

模板参数包： 零个或多个模板参数

template<typename T, typename... Args>

函数参数包： 零个或多个函数参数

void foo(const T &t, const Args& ... rest);

模板特例化

模板的重定义，指定了某些（全部）模板参数。必须出现在原模板之后，使用模板之前。

part4 高级主题

chap17 标准库特殊设施

tuple 类型

快速而随意的结构，类似pair 的模板

任意数量的成员，每个成员类型可以都不相同

定义和初始化

默认构造函数： tuple< T1, T2, ..., Tn >t;

必须使用直接初始化(7.5.4节）

make_tuple auto item = make_tuple< "str", 1,22.2>;

访问tuple成员

auto book = ger<0>(item) ;

也可以结合tuple_size , tuple_element使用

bitset类型

位运算，类模板，具有固定的大小

定义和初始化

需要声明含有二进制位的数量

bitset<32> bitvec(1U）；32位，低位为1，其他位为0

P17.2.1

用unsigned long long类型初始化

bitset<128> bitvec1(~0ULL）; 0～63 位为1，其他为0

从string初始化

bitset<32> bitvec2("1100"); // 2，3，位为1，其他位为0

注意string的下标编号习惯与bitset恰好相反

bitset操作（表17.3）

正则表达式

RE定义在regex头文件中

regex组件

随机数

C/C++ 可以使用rand来生成随机数（0～ 系统最大值之间）

C++11：随机数库（定义在random）中

随机数库组成(详见附录A3）

引擎

类型，生成随机unsigned整数序列

分布

类型，使用引擎返回服从特定概率分布的随机数

例子

给定的随机数发生器生成相同的随机数序列

default_random_engine e; uniform_int_distribution<unsigned> u（0,9）; cout<< u(e)；// 0，2，3，2，2，4...

若不相同，可以定义引擎和关联的分布对象为static

IO库再探

格式化输入和输出

不再需要特殊格式时将流恢复默认状态

注意表17.17

控制bool值的格式

控制整数型的进制

在输出中指出进制

控制浮点数格式

指定打印精度

指定浮点数计数法
打印小数点
控制输入格式
未格式化的输入
注意使用，容易出错
底层操作，支持未格式化IO：将流当作一个无解释的字节序列来处理
单字节操作
多字节操作
流随机访问
iostream不支持，适用于fstream和sstream

chap18 用于大型程序的工具

异常处理
程序独立开发的部分能够在运行时就出现的问题进行通信并做出相应的处理
抛出异常
抛出 throwing 一条表达式来引发一个异常
控制权从一处转移到另一处
沿着调用链的函数可能会提前退出
一旦程序开始执行异常处理代码，沿着调用链创建的对象可能被销毁
栈展开
搜索catch 时依次退出函数的过程
栈展开过程中，对象将被自动销毁
捕获异常
处理异常部分 catch(异常声明) {处理语句}
最好将catch 的参数定义成引用类型
捕获所以异常 catch all （....) {}
可以在构造函数中加入 try 语句块
noexcept 异常说明
运算符，返回bool值，表示表达式是否会抛出异常
该表达式不会被求值，结果是常量表达式。当不含throw 且只调用了不抛出说明的函数时，结果为true

```
void recoup(int)  noexcept;   // 不会抛出异常
```

异常类层次
标准库异常类 图18.1
使用自己的异常

```
throw my_own_mistake(....)

try （。。。） catch （ my_own_mistake &e) {...};
```

命名空间
定义
一个命名空间是一个作用域,可以是不连续的
命名空间污染
由于当 所有类和函数的名字都放置于全局命名空间导致
定义 namespace cplus_prime{ 类等声明、定义}
别名 namespace N1 = Name; // 可以含有多个别名，原名与别名等价

全局命名空间

以隐式方式声明，在所有的程序中存在

:: member_name ； //表示全局命名空间中的一个成员

嵌套的命名空间 : 定义在其他命名空间的命名空间

namespace cpp_prime{ namespace chap1{ class list{/* */}...} namespace chap2{...} }

cpp_prime::chap1::list

内敛的命名空间 : 可以被外层命名空间直接使用

inline namespace FifthEd{ class Query{....} }; namespace cpp_prime{ #include "FifthEd .h"

cpp_prime:: Query //可以访问FitrhEd

未命名的命名空间 namespace {/* 省略..... */

可以取代文件中的静态声明 static

使用命名空间成员

声明别名

using 声明 将命名空间的某个名字注入当前作用域 例如 using std::cout

using 指示 将命名空间的所有名字可见 例如 using std

注意可能会导致命名空间污染

若在头文件顶层作用域中使用则会将名字注入到所有包含该头文件的文件中！

类、命名空间与作用域

名字查找规则： 由内向外依次查找每个外层作用域

注意std::move,std::forward使用，一般带有std,表示使用的是标准库的版本

重载与命名空间

using 声明或using 指示可以将某些函数添加到候选函数集，重载时选择

多重继承

有多个直接基类的类，可以为每个基类分别设定访问说明符

派生类构造函数初始化所有基类

类型转换: 派生类的指针或引用能自动转换成一个可访问基类的指针或引用

派生类可能从多个基类继承同名函数，避免二义性需要加前缀限定符

虚继承： 基类被继承多次，但是派生类共享该基类的惟一一份副本

虚基类：派生列表中使用关键字virtual 的基类，最底层的派生类应该含有其所有虚基类的初始值

chap19 特殊工具与技术