第1章 C

1. printf

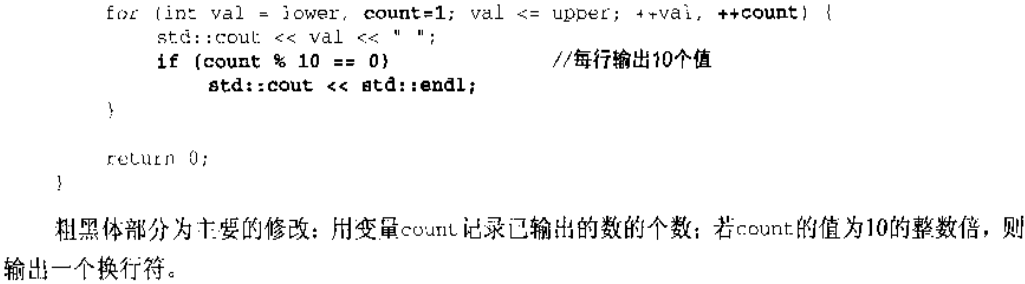
printf("%.1f\n",8/5);//0.0

printf("%d\n",8.0/5.0); //-1717986918

因为8/5的结果是按整数存储，打印是按浮点数读取

printf("%f\n",111111111.0\*111111111.0);//12345678987654320.000000

1. 凡是出现访问数组的地方，考虑有没有越界。出现加减法，考虑加减法的意义，尤其减数和被减数有没有搞错、搞颠倒。
2. 两增量的循环

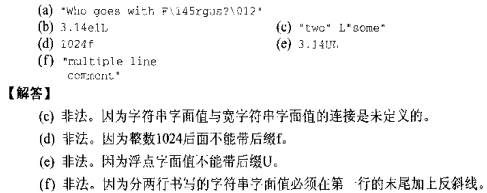


第2章 变量和基本类型

1. 指定字面值的类型

|  |  |  |  |
| --- | --- | --- | --- |
| 字符和字符串字面值 | | | |
| 前缀 | 含义 | | 类型 |
| u | Unicode 16字符 | | char16\_t |
| U | Unicode 32字符 | | char32\_t |
| L | 宽字符 | | wchar\_t |
| u8 | Utf-8（仅用于字符串字面常量） | | char |
| 整型字面值 | | 浮点型字面值 | |
| 后缀 | 最小匹配类型 | 后缀 | 类型 |
| u or U | unsigned | f或F | float |
| l or L | long | l或L | long double |
| ll or LL | long long |  |  |

以下哪些是非法字面值



字面值类型（literal type）：

算术类型、指针和引用都属于字面值类型，自定义类Sales\_item、IO库、string类型不属于字面值类型。

*什么是literal type因时代不同回答也不同。*

1. 转义序列：

换行符\n 横向制表符\t 报警符\a 纵向制表符\v 退格符\b 双引号\”

反斜线\\ 问号 \? 单引号\’ 回车符 \r 进纸符\f

泛化的转义序列：\x后紧跟多个十六进制数字，或者\后紧跟1-3个8进制数字

\12 (换行符) \115 (字母M)

1. C++标识符由字母、数字和下划线组成，其中必须以字母或下划线开头。标识符的长度没有限制，但是对大小写字母敏感。用户自定义的标识符不能连续出现2个下划线，也不能以下划线紧连大写字母开头。定义在函数体外的标识符不能以下划线开头。

变量命名规范： 要能体现实际含义；变量名一般用小写字母；用户自定义类名一般用大写字母开头，e.g. Sales\_item；如果identifier 由多个单词组成，单词间应有明显区分，如student\_loan或studentLoan

1. 管道：test.exe<infile>outfile
2. 类型char和signed char不一样，char具体表现为signed或unsigned由编译器决定。

(VC、x86上的GCC把char定义为signed，而arm-linux-gcc把char定义为unsigned)

1. 如果想在多个文件之间共享const对象，必须在变量的定义之前添加extern关键字

//file\_1.cc定义并初始化了一个常量，该常量能被其他文件访问

extern const int bufSize=fcn();

//file\_1.h头文件

extern const int bufSize;//与file\_1.cc中定义的bufSize是同一个

1. 如果某个类型别名指代的是复合类型或常量，那么把它用到声明语句里就会产生意想不到的后果。

typedef char \*pstring;

const pstring cstr=0;//cstr是指向char的常量指针

const pstring \*ps;//ps是一个指针，它的对象是指向char的常量指针

const char \*cstr=0;// cstr是指向char常量的指针

声明语句中用到pstring时，其基本数据类型是指针，而用char\*重写了声明语句后，数据类型就变成了char，\*成为了声明符的一部分

typedef int int\_array[4];

using int\_array=int[4]

1. auto会忽略顶层const(本身是常量)，同时底层const(指针所指对象是常量)会保留

int i=0,ia[]={0,1,2,3};

const int ci=i;

auto b=ci;//b是一个整数，顶层const被忽略

auto d=&i;//d是一个整形指针（整数的地址就是指向整数的指针）

auto e=&ci;//e是一个指向整数常量的指针（对常量对象取地址是底层const）

auto ia2(ia);//ia2是整形指针，指向ia的第一个元素，实际执行的初始化和下面类似：

auto ia2(&ia[0]);//ia2的类型是int\*

decltype(ia) ia3={1,2,3,4};//ia3是一个数组

1. decltype( (variable) )的结果永远是引用，而decltype(variable)的结果只有当variable本身是一个引用时才是引用。

如果decltype使用的表达式是一个变量，则decltype返回该变量的类型（包括顶层const和引用在内）

int i=0;

const int ci=0,&cj=ci,\*p=&ci;

decltype(ci) x =0;//x的类型是const int

decltype(cj) y =x;//y的类型是const int&

如果表达式的内容是解引用操作，则decltype将得到引用类型。解引用操作能得到指针所指的对象，而且还能给这个对象赋值。

decltype(\*p) d=i;//d的类型是const int&

引用从来都是作为其所指对象的同义词出现，只有用在decltype处是一个例外。

decltype(cj+0) b;//加法的结果是int，b的类型是int

赋值是会产生引用的一类典型表达式，e是int&；但赋值不会发生，即x不会变为y的值

decltype(x=y) e=i;//e的类型是 int&

整数的地址就是指向整数的指针，q是一个指针，q的地址就是指向指针的指针

int \*q;

decltype(&q) f;//f的类型是 int\*\*,是一个指向整型指针的指针

1. 基本内置类型：

|  |  |  |
| --- | --- | --- |
| 类型 | 含义 | 最小尺寸 |
| bool |  | 未定义 |
| char |  | 8位 |
| int、short、wchar\_t |  | 16位 |
| char16\_t、char32\_t | Unicode字符 | 16位、32位 |
| long、long long | 长整型 | 32位、64位 |
| double、float | 双、单精度浮点数 | 10、6位有效数字 |
| long double | 扩展精度浮点数 | 10位有效数字 |

c语言里面没有bool类型，也没有string类型，Java里面有boolean

c语言string是用的地址，char \*

指针大小是machine相关的，32位系统4bytes，64位系统64bits

long double也是machine相关的.

1. 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,声明为constexpr的变量一定是常量,而且必须用常量表达式初始化.

constexpr int mf=20;

constexpr int limit=mf+1;

constexpr int sz=size(); //只有当size是constexpr函数时才是正确的语句

在constexpr声明中如果定义了一个指针,constexpr仅对指针有效,与指针所指对象无关

constexpr int \*q=nullptr; //q是一个指向整数的常量指针

constexpr函数指能用于常量表达式的函数。

* 函数的返回类型和所有形参的类型都得是[字面值类型](#字面值类型)；
* 函数体中有且只有一条return语句，允许返回值并非一个常量；
* 函数体内可以包含其它运行时不执行任何操作的语句，如空语句、类型别名。

1. 头文件通常包含那些只能被定义一次的实体,如类、const和constexpr变量

#define指令把一个名字设为预处理变量

#ifdef当仅当变量已定义时为真

#ifndef当仅当变量未定义时为真

当检查结果为假,编译器将忽略#ifdef到#endif之间的部分

#ifndef SALES\_DATA\_H

#define SALES\_DATA\_H

#include <string>

struct Sales\_data {

string bookNo;

int quantity\_sold;

double revenue;

};

#endif

头文件不应包含using声明,因为头文件的内容会拷贝到所有引用它的文件中去,如果头文件里有某个using声明,那么每个使用了该头文件的文件就都会有这个声明.对于某些程序,由于不经意间包含一些名字,反而可能产生始料未及的冲突.

内联函数、函数声明放在头文件中。

1. 初始化小结

列表初始化，用花括号来初始化变量。当用于内置类型的变量时，如果我们使用列表初始化且存在丢失信息的风险，则编译器会报错。

double d=3.14;

int a{d}, b={d}; //错误，转换未执行，存在丢失信息的危险

int c(d), b=d; //正确，转换执行，且确实丢失了部分信息

默认初始化：

若定义变量时没有指定初值，则变量被默认初始化。定义于函数体外的内置类型的变量被初始化为0，定义在函数体内的内置类型变量不被初始化。类的对象如果没有显式地初始化，则其值由类决定。

拷贝初始化：

* 使用=定义变量
* 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{}列表初始化一个数组中的元素或一个聚合类中的成员
* 某些类类型会对所分配的对象使用拷贝初始化

当初始化标准库容器或是调用其insert、push成员，容器会对其元素进行拷贝初始化，用emplace成员创建的元素进行直接初始化。

值初始化：

可以只提供vector对象容纳的元素数量而略去初始值。此时库会创建一个值初始化的由vector对象中元素类型决定的初值，并把它赋给容器中的所有元素。

如果vector对象的元素是内置类型，则初始值为0；如果元素是某种类类型，如string，则元素由类默认初始化。

某些类要求明确提供初始值，如vector对象中元素的类型不支持默认初始化；如果只提供元素的数量而没有初始值，只能使用直接初始化。

对象(object)是内存的一块区域,具有某种类型,变量是命名了的对象.

因为无法令引用重新绑定到另外一个对象,引用必须初始化，初始值必须是一个对象;const对象一旦创建后其值不能改变，也必须初始化。

不改变const对象的操作:算术运算、转化成bool值、初始化等等。如果利用一个对象去初始化另外一个对象，则它们是不是const都无关紧要。

int i=42;

const int ci=i;

int j=ci;

引用不是对象，没有实际地址，所以不能定义指向引用的指针，引用只是为一个已经存在的对象另外取一个名字。引用、指针的类型必须和初始值的类型相同。返回左值引用的函数，和赋值、下标、解引用、前置递增递减运算符返回左值。

**例外：**初始化常量引用时允许用任意表达式作为初始值，只要表达式的结果能转换成引用的类型即可。­可以将const引用或右值引用绑定到：返回非引用类型的函数，算术、关系、位、后置递增递减运算符生成的右值。

int i = -1, &r = 0; // illegal, r must refer to an object.

const int i = -1, &r = 0; // legal.

允许初始化一个指向常量的指针指向一个非常量的对象。

double dval=3.14;

const double \*cptr = &dval; // legal. cptr的值初始化之后不变

第3章 string、vector和数组

初始化string、vector

cp类型是const char\*

|  |  |
| --- | --- |
| string s | |
| string s=s1; string s(s1); string s{s1}; | |
| string s=“value”; string s{“value”}; string s(“value”); | |
| string s(n,’c’); | s为连续n个字符c |
| string s{'b','c','\0'}; | 允许用字符数组初始化string对象 |
| string s(cp,n)  string s(cp) | s是cp指向的字符开始拷贝n个字符。指向的字符后最少应包含n个字符。若以空字符结尾可以不指定n |
| string s(s2,pos2) | s是s2从下标pos2开始的字符的拷贝。若pos2>s2.size()，构造函数的行为未定义 |
| string s(s2,pos2,len2)  s.substr(pos,n) | s是s2从下标pos2开始len2个字符的拷贝。不管len2的大小，最多拷贝到s2的末尾 |

1. 有多个初始值,如s(n,’c’),一般只能使用直接初始化
2. 如果使用的是一个类内初始值，则只能使用=或{}初始化,不能用()
3. 如果使用初始元素值的列表，则只能用{}，而不能用()

|  |  |
| --- | --- |
| s.erase(pos,n) | 在下标pos之后删除n个字符，如果省略n则删除pos开始的所有字符。返回指向s的引用 |

|  |  |
| --- | --- |
| s.assign(args) | 将s替换为args，返回指向s的引用 |
| s.append(args) | 将args追加到s末尾。返回指向s的引用 |
| s.insert(pos,args)  s.insert(iter,b,e) | 在下标pos之前插入  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.replace(pos,n,args)  s.replace(b,e,args) | 从下标pos开始的n个字符(或迭代器b、e之间的字符)替换为args。返回指向s的引用 |

args可以是下列形式：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | s.replace  (p,n,args) | s.insert  (pos,args) | s.replace  (b,e,args) | s.insert  (iter,args) |
| str(不能是s) | | √ | | √ |  |
| str,pos,len | |  |
| cp,len | cp指向的数组前len个字符 | √ |
| cp | 空字符结尾的字符数组 |
| n,c | n个字符c | √ | | | |
| b,e | 迭代器(不指向s) |  | | √ | |
| 列表 | {}包围的以,分隔的字符列表 |

string查找操作

|  |  |
| --- | --- |
| s.find(args)  s.rfind(args) | 查找s中args最后一次出现的位置 |
| s.find\_first\_of(args)  s.find\_last\_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\_first\_not\_of(args)  s.find\_last\_not\_of(args) | 在s中查找第一个不在args中的字符 |

args是以下形式

|  |  |
| --- | --- |
| c,pos | 从s中pos位置开始查找字符c，pos默认为0 |
| s2,pos | 从pos开始查找s2 |
| cp,pos | 从pos开始查找cp指向的以空字符结尾的字符数组 |
| cp,pos,n | 从pos开始查找cp指向数组的前n个字符 |

s.compare的参数

|  |  |
| --- | --- |
| s2  pos,n,s2  pos,n,s2,pos2,n2 | s中pos开始的n个字符与s2比较 |
| cp  pos,n,cp  pos,n,cp,n2 |  |

vector<T> v1;//默认初始化，v1是一个空vector,它潜在的元素是T类型的

vector<T> v2(v1);

vector<T> v2=v1;

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个重复执行了值初始化的对象

vector<T> v5{a,b,c,d…);//v5包含了初始值个数的元素

vector<T> v5={a,b,c,d…);//等价于v5{a,b,c,d…)

vector<string> v1{“a”,“an”,“the”};

vector<string> svec(10,“hi!”);//10个string类型元素，每个的值都是“hi!”

vector<int> ivec(10,-1);//10个int类型元素，每个的值都是-1

vector<int> v3(10);//10个元素，每个的值是0

vector<int> v4{10};//1个元素，值是10

如果初始化时使用了花括号的形式但是提供的值不能用来列表初始化，就要考虑用这样的值来构造vector对象了。

vector<string> v5{10};// v5有10个默认初始化的元素

vector<string> v6{10,“hi!”};// v6有10个值为“hi!”的元素

1. cin读入string对象会自动忽略开头的空白（即空格符、换行符、制表符etc.）并从第一个真正的字符开始读起，直到遇见下一处空白为止；getline的参数是一个输入流和一个string对象，函数从给定的输入流中读入内容，直到遇到换行符为止，然后把所读取的内容存到那个string对象中去，得到的string对象并不包含该换行符。

getline(cin,line);

if(!line.empty()&&line.size()>2)

cout<<line<<endl;

getline和cin混合使用读入多行输入时注意

while(getline(cin,s))

cin>>word1>>word2;

当输入为

ac

a ab

tho tho

s的第二个值是一个回车

1. size函数返回的是 一个string::size\_type类型的值，它是一个无符号数，不要用无符号数和负数比较大小，表达式s.size()<n的结果几乎肯定为true，因为负值n会自动转换成一个（比较大的）无符号数。
2. 基于范围的for语句小结：

for(declaration:expression)

statement

使用范围for语句改变字符串中的字符，引用被依次绑定到每个元素上

string s(“Hello world!!!”);

for(auto &c: s)

c=toupper(c);

cout<<s<<endl;

要使用范围for语句处理多维数组，除了最内层的循环外，其它所有循环的控制变量都应该是引用类型。（因为如果不是引用类型，编译器初始化的时候会自动将这些数组形式的元素自动转换成指向该数组内首元素的指针，显然内层的循环就不合法了，编译器将试图在一个指针内遍历。）

v.push\_back(t);// 用vector的成员函数push\_back向v的尾端添加一个值为t的元素

语句体内不应改变所遍历序列的大小,如不能在范围for循环中向vector对象添加元素.

1. 定义数组的时候必须指定数组的类型，不允许使用auto由初始值的列表推断类型。另外和vector一样，数组的元素应为对象

int \*arr[10];//一个数组，该数组含有10个整型指针

int \*(&arrRef)[10]=arr;//引用一个数组，该数组含有10个整型指针

int &refa[10]=/\*\*/;//错误，不存在引用的数组

1. 当把string对象和字符字面值及字符串字面值混在一条语句中使用时，必须确保每个加法运算符（+）的两侧的运算对象至少有一个是string：

string s4=s1+”,”;//正确

string s5=”hello” + ”,” + s2;//错误

string s6=s2 + ”hello” + ”,” ;/\*正确，（s2 + ”hello”）的结果是一个string对象，它同时作为第二个加法运算符的左侧运算对象\*/

1. c风格字符串

char cs1[]="hello ",cs2[]="world!";

char cs3[ strlen(cs1)+strlen(cs2)+1];

strcmp(cs1,cs2); //如果cs1>cs2,返回正值,"A"<"B" "a">"A" "computer">"compare"

strcpy(cs3,cs1); //将cs1拷贝给cs3,返回cs3

strcat(cs3,cs2); //将cs2加到cs3之后,返回cs3

比较string对象用的== != < <= > >=不能用于c-style字符串，因为实际比较的是两个指针。

1. 数组初始化小结

可以用字符串字面值对字符数组进行初始化，此时字符串字面值的结尾还有一个空字符。

char a1[]={‘c’,’+’,‘+’};

char a2[]={‘c’,‘+’,‘+’,‘\0’};

char a3[]=”c++”;//自动添加表示字符串结束的空字符

char a4[6]=”Daniel”;//错误，没有空间可以存放空字符

在string对象的加法运算、复合赋值运算中，任何出现字符串字面值的地方都可以用以空字符结尾的字符数组来替代。

为了使用string对象直接初始化指向字符的指针，string专门提供了一个c\_str的成员函数

string s;

const char \*str=s.c\_str();///\*不能保证c\_str函数返回的数组一直有效，如果后续操作改变了s的值就可能让之前返回的数组失效；如果想一直使用返回的数组，最好将该数组重新拷贝一份\*/

*char \*str =s;* //错误

不允许使用数组为另一个内置类型的数组赋初值，也不允许使用vector对象初始化数组，但允许使用数组初始化vector对象

int arr[]={0,1,2,3,4,5};

vector<int> vec(begin(arr),end(arr));//begin函数返回指向数组arr首元素的指针

vector<int> subVec(arr+1,arr+4);//拷贝arr[1]、arr[2]、arr[3]

1. 使用数组下标的时候，通常将其定义为size\_t（size\_t是一种机器相关的无符号类型，在cstddef头文件中定义）。标准库类型限定使用的下标是无符号类型，而内置类型的下标运算无此要求。

int ia[]={0,1,2,3,4,5};

int \*q=ia;

int i=\*(q+2);//i=ia[2]

int \*p=&ia[2];

int k=p[-2];//k=\*(p-2),即k=ia[0]

1. 遍历多维数组

int ia[2][4]= {1,2,3,4,5,6,7,8};

for(auto &p:ia)//int (&p)[4],一个引用,对象是含4个int的数组

for(auto q:p)

cout<<q<<' ';

cout<<endl;

for(auto p=0; p!=2;++p)//size\_t p

for(auto q=0; q!=4;++q)

cout<<ia[p][q]<<' ';

cout<<endl;

for(auto p=ia; p!=ia+2; ++p)//int (\*p)[4],一个指针,对象是含4个int的数组

for(auto q=\*p; q!=\*p+4; ++q)//int \*q

cout<<\*q<<' ';

第4章 表达式

1. 除了-m导致溢出时，其余时候 (-m)/n = m/(-n) = -(m/n) ；m%n=m%(-n)，(-m)%n=-(m/n)
2. sizeof

指针:得到指针本身所占空间的大小,machine相关

引用类型、解引用指针:被引用对象、指针所指对象的大小

数组：整个数组所占空间

string、vector：只返回该类型的大小，不计算对象中的元素占用多少空间

int ia[2]={1,2};

constexpr size\_t sz=sizeof(ia)/sizeof(\*ia); //sz=2

1. 数组自动转换成指向数组首元素的指针

**例外**:数组被用作decltype关键字的参数，作为&、sizeof、typeid等运算符的运算对象时，转换不会发生。

void\*指针可以存放任意对象的地址，但不能直接操作void\*指针所指的对象。

常量整数0或字面值nullptr能转换成任意指针类型；指向任意非常量的指针能转换成void\*；指向任意对象的指针能转换成const void\*。

形参不能是函数，但可以是指向函数的指针。

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

typedef bool Func(const string &,const string &);

typedef decltype(lengthCompare) Func2; //func和func2是相同的函数类型

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

typedef bool (\*FuncP)(const string &,const string &);

//decltype的结果是函数类型，需要加上\*才能得到指针

typedef decltype(lengthCompare) \*FuncP2;

using FuncP3= bool (\*)(const string &,const string &);

当把函数名当值使用时，函数自动转换成指针。

void uBig(const string &s1, const string &s2, bool pf(const string &,const string &)) ;//和下面3条语句等价

void uBig(const string &s1,const string &s2,bool (\*pf)(const string &,const string &)) ;

void uBig(const string &s1,const string &s2,Func) ;

void uBig(const string &s1,const string &s2,FuncP2) ;

uBig(s1,s2,lengthCompare);

pf=lengthCompare; //等价于pf=&lengthCompare

返回类型必须写成指针形式，编译器不会自动将函数返回类型当做对应的指针类型.

FuncP f1(int);

Func f1(int); //错误

Func \*f1(int);

bool (\*f1(int)) (const string &, const string &);

decltype(lengthCompare) \*f2(int);

auto f1(int)->bool (\*)(const string &, const string &);

使用指向函数的指针调用函数时，无需解引用指针。

bool b1=pf(“he”, “is”);//和下面2条语句等价

bool b2=(\*pf)(“he”, “is”);

bool b3=lengthCompare(“he”, “is”);

1. 任何具有明确定义的类型转换，只要不包含底层const，都可以使用static\_cast.

只有const\_cast能改变表达式的const属性，不能用const\_cast改变表达式的类型。

const char \*cp;

char \*q=static\_cast<char\*>(cp); //错误

string sc=const\_cast<string>(cp); //错误

string s=static\_cast<string>(cp); //正确

static\_cast：相关类型的转换，如char、int、bool， string与c-string，其余指针与void\*

reinterpret\_cast：不相关类型的转换（只是内存的简单拷贝），如int与指针

double d=0.4;

void\* p=&d; //void\* p=static\_cast<void\*>(&d);

double \*dp=static\_cast<double\*>(p);

当把指针存放在void\*中，并且使用static\_cast将其强制转换回原来的类型时，应该确保指针的值保持不变（确保转换后的类型就是指针所指的类型）。

1. 逗号运算符首先对左侧的表达式求值，然后将求值结果丢弃；逗号运算符的结果是右侧

表达式的值。

第5章 语句

1. do while(condition);语句中condition使用的变量必须定义在循环体之外。
2. break跳出最近的switch或循环；continue只能用在循环内，只有当switch嵌套在循环

内时才能在switch里使用continue.

第6章 函数

1. 如果函数实参数量未知但全部实参的类型相同，可以使用initializer\_list类型的实参

int add(initializer\_list<int> il); //类似于vector

cout<<add({1,2,3})<<endl; //attention to {}

第7章 类

1. IO类属于不能被拷贝的类型，因此只能通过引用来传递。
2. 可以将构造函数声明为explicit来阻止隐式转换。发生隐式转换的一种情况是当执行拷

贝初始化时，explicit构造函数只能用于直接初始化。

关键字explicit只对一个实参的构造函数有效；只能在类内声明构造函数时使用explict关键字。

可以显式强制转换：

item.combine(Sales\_data(null\_book)); //实参是一个显式构造的Sales\_data对象

//static\_cast可以使用explicit构造函数

item.combine(static\_cast<Sales\_data>(cin));

1. 聚合类

* 所有成员都是public；
* 没有定义构造函数；
* 没有类内初始值；
* 没有基类，没有virtual函数。

只有聚合类才能用{}括起来的初始值列表初始化。

1. 字面值常量类

* 数据成员都是[字面值类型](#字面值类型)；
* 至少含有一个constexpr构造函数；
* 如果一个数据成员含有类内初始值，则内置类型成员的初始值必须是一个常量表达式；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。

数据成员都是字面值类型的聚合类是字面值常量类。

1. 某类型在声明之后定义之前是不完全类型，即清楚是某类型，但是不清楚包含哪些成员。

可以定义指向不完全类型的指针或者引用，可以声明但是不能定义以不完全类型作为参数或返回类型的函数。

只有当类全部完成后才算被定义，所以一个类的成员类型不能是该类自己；然而一旦类名出现后就被认为是声明过了，因此类允许包含指向它自身类型的引用或指针。

1. 局部静态对象（local static object）

size\_t count\_calls(){

static size\_t ctr=0; //调用结束后这个值仍然有效

return ++ctr;

}

类的静态成员

静态成员可以是public或private，类型可以是常量、引用、指针、类类型等。static关键字只出现在类内部的声明语句中。在类的外部定义静态成员时不能重复static关键字。

静态成员函数不能声明成const的，也不能在static函数体内使用this指针。

静态成员存在于任何对象之外，对象中不包含与静态成员有关的数据。静态成员对象只存在一个且被所有类对象共享。可以用类的对象、引用或指针来访问静态成员。

使用作用域运算符直接访问静态成员：

class Account{

static double irate;

double amount;

public:

static double rate(){return irate;}

Account &clear(double=irate);

void f(){amount+=irate;}

}

double r=Account::rate();

Account ac1;

Account \*ac2=&ac1;

r=ac1.rate();

r=ac2->rate();

一般不能在类内部初始化静态成员，必须在类外定义和初始化静态成员。

如果静态成员是字面值常量类型的constexpr，可以为静态成员提供const int类型的类内初始值。即使一个常量静态数据成员在类内被初始化了，通常也应该在类外部定义一下该成员。

静态数据成员可以是不完全类型。可以使用静态成员作为默认实参，非静态成员不能。

第8章 IO库

1. 使用流之前:
2. 包含相应的的头文件

|  |  |  |
| --- | --- | --- |
| iostream | istream，wistream  ostream，  iostream |  |
| fstream | ifstream，  ofstream，  fstream | fstream fs; fs.close()调用close关闭流关联的文件才能调用open打开另一个文件  fstream fs(s);打开文件s fs.open()  fstream fs(s,mode); fs.is\_open() |
| sstring | istringstream，  ostringstream，  stringstream | sstream ss; ss.str()返回ss保存的string拷贝  sstream ss(s); 保存string s拷贝  ss.str(s) 将string s拷贝到ss，返回void |

1. 考虑流的状态

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| s.eof() | s.fail() | s.bad() | s.good() | s.clear() | s.setstate(flags) | s.rdstate() |
|  |  | 流崩溃 |  | s.clear(flags) |  | 返回流状态 |

ifstream input(argv[1]);

if(input) //如果打开成功就可以使用流

s.clear()将所有条件状态位复位，将流的状态置为有效，将给定的flags复位

需要运行参数的exe文件没有在命令行给正确的参数运行会出错。

1. 文件模式

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| in | out | app | ate | trunc | binary |
|  |  | append | at end | truncate |  |

//下面3条语句中file1都被截断

ofstream out(“file1”);

ofstream out2(“file1”,ofstream::out);

ofstream out3(“file1”,ofstream::out|ofstream::trunc);

//为了保留文件内容，必须显式指定app模式

ofstream app(“file2”,ofstream::app); //隐含为输出模式

ofstream app2(“file2”,ofstream::out|ofstream::app);

1. string流

while(getline(cin,s)) {

istringstream scin(s);

while(scin>>word) {

}

}

第9章 顺序容器

1. 养成使用迭代器和!=的习惯，就不用太在意用的是哪种容器类型。

for(auto it =s.begin();it!=s.end()&&!isspace(\*it);++it)

\*it=toupper(\*it);

\*iter 返回iter所指元素的引用

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

int ia[]={0,1,3,4,5};

int \*beg=begin(ia),\*last=end(ia);//指向ia尾元素的下一位置的指针,&ia[5]

1. 容器操作

p为iter

|  |  |
| --- | --- |
| 类型别名 | iterator const\_iterator size\_type  difference\_type 迭代器之间的距离  value\_type cosnt\_reference  reference 与value\_type&含义相同 |
| 构造函数 | C c(begin,end) 不支持array  只有顺序容器（不含array）的构造函数能接受大小参数  C c(n) n个值初始化的元素（string不支持） |
| 赋值与swap | c={a,b,c….}; 不适用于array  c.assign(b,e) 将c中元素替换为迭代器b、e之间元素，b、e不能指向c中元素  c.assign(il) c.assign(n,t) |
| 大小 | c.size() 不支持forward\_list  c.max\_size()可以保存的最大元素的数目 |
| 添加、删除元素  back\_inserter(c)  创建使用push\_back的迭代器  front\_inserter(c)  inserter(c,iter)  元素被插入到给定迭代器所指向元素之前的位置 | c.push\_back(t) c.push\_front(t)  c.emplace\_front(args) c.emplace\_back(args)  c.insert(p,t) 将t拷贝到p指向的元素之前  c.insert(p,n,t) 在p指向的元素之前插入n个t  c.insert(p,b,e) 在p指向的元素之前插入b和e之间的元素，返回指向新添加的第一个元素的iter；若范围为空，返回p  c.insert(p,il) 在p指向的元素之前插入{}括起来的元素值列表里的值，返回指向新添加的第一个元素的iter；若范围为空，返回p  c.emplace(p,inits) 使用inits在p指向的元素之前构造元素  c.pop\_back(t) c.pop\_front(t)  c.erase(p) 删除p指定的元素，返回指向被删元素之后位置的iter  c.erase(b,e) 删除b、e之间的元素，返回e  c.clear() 删除c中的所有元素，返回void  c.resize(n) 调整c的大小为n，多出的元素被丢弃，新添加的元素进行值初始化  c.resize(n,t) 新添加的元素初始化为t |
| 关系运算符 | == != < <= > >= 依照字典顺序进行比较；元素少的小于元素多的；只有当元素的值可以比较时对象才能被比较；无序、关联容器不支持 |
| 获取迭代器 | c.cbegin(),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iter  const\_reverse\_iterator  c.rbegin(),c.rend() 返回指向c的尾元素、c的首元素之前位置的iter  c.crbegin(),c.crend() 返回const\_reverse\_iterator |

forward\_list操作

|  |  |
| --- | --- |
| lst.before\_begin()  lst.cbefore\_begin() | 返回指向首元素之前位置的迭代器。cbefore\_begin()返回const\_iterator |
| lst.insert\_after(p,t)  lst.insert\_after(p,n,t)  lst.insert\_after(p,b,e)  lst.insert\_after(p,il) | 在迭代器之后的位置插入元素。返回指向最后一个插入元素的迭代器。若范围为空，返回p |
| emplace\_after(p,args) | 使用args在p指定的位置之后创建元素，返回指向这个新元素的迭代器 |
| lst.erase\_after(p)  lst.erase\_after(b,e) | 删除p指向位置之后的元素，返回被删元素之后位置的迭代器 |

array不支持插入删除

forward\_list不支持c.push\_back(t)、c.emplace\_back(args)、c.pop\_back()，没有定义insert、emplace、erase

vector和string不支持c.push\_front(t)、c.emplace\_front(args)、c.pop\_front()

向vector、string中插入元素会使所有指向容器的迭代器、引用、指针失效；删除元素会使被删元素之后的迭代器、引用、指针失效。

向deque中除首尾之外的任何位置插入元素会使所有指向容器的迭代器、引用、指针失效；在首尾位置添加元素，迭代器失效；删除除首尾之外的任何元素会使所有迭代器、引用、指针失效；删除尾元素使尾后迭代器失效。

改变容器的操作可能使迭代器失效，因此如果需要使用迭代器则需要重新定位。

vector、string支持<,<=,>,>=,==,!=这些比较大小的操作，以字典顺序进行比较；vector、string的迭代器支持<,<=,>,>=,==,!=这些比较大小的操作，用指向的容器位置进行比较。

3

2

1

1<2<3

forword\_list迭代器不支持--操作，iter+n,iter-n,iter-iter,<,<=,>,>=只能应用于string、vector、deque、array，不能用于forward\_list、list。

1. 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。

当传递迭代器参数来拷贝一个范围时，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。

//可以将const char\*转换为string

vector<const char\*> articles{“a”, “an”, “the”};

forward\_list<string> words(articles.begin(), articles.end());

assign允许类型可以转换的赋值

list<string> slist(1);

slist.assign(10, “hiya”);

//下面的操作与上面的等价

slist.clear();

slist.insert(slist.begin(), 10 , “hiya”);

slist.assign(articles.begin(), articles.end());

1. 顺序容器中访问元素

|  |  |  |  |
| --- | --- | --- | --- |
| c.back() | c.front() | c[n] | c.at(n) |

at和下标操作只适用于array、deque、vector、string，如果下标越界at会抛出一个out\_of\_range异常。

c.back()返回尾元素的引用，不适用于forward\_list

成员函数返回的都是引用。

vector<int> c{1,23};

decltype(c.front()) f=c.front();

auto &v=c.back();

auto v2=c.back();//v2不是一个引用，是尾元素的拷贝

1. 默认情况下，stack、queue是基于deque实现的，priority\_queue是在vector上实现的。可以在创建适配器时将一个命名的顺序容器作为第二个类型参数来重载默认容器类型。

stack<string,vector<string>> str\_stk(svec);

|  |  |  |
| --- | --- | --- |
| stack | priority\_queue | queue |
| deque、vector、list | vector、deque | deque、list、vector |
| sqp.pop() 删除栈顶、队首元素 sp.top() 返回栈顶元素 q.front()返回队首元素  sqp.push(item) q.back()返回队尾元素  sqp.emplace(args) | | |

第10章 泛型算法

1. lambda表达式

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

可以向算法传递任何类别的callable object。对于一个对象或一个表达式，如果可以对其使用调用运算符()，则称它为可调用的。

callable object:函数，函数指针，重载了函数调用运算符的类，lambda表达式。

|  |  |
| --- | --- |
| [] | 空捕获列表 |
| [names] | names是一个,分隔的lambda所在函数的局部变量的列表。默认为值捕获，名字前如果使用了&则引用捕获 |
| [&] | 隐式引用捕获列表 |
| [=] | 隐式值捕获列表 |
| [&,identifier\_list] | identify\_list是一个,分隔的列表，这些变量采用值捕获方式，隐式捕获的变量使用引用捕获 |
| [=,identifier\_list] | 隐式捕获的变量使用值捕获，identify\_list中的名字不能包括this，名字前必须使用& |

1. 默认情况下bind的不是占位符的参数被拷贝到bind返回的可调用对象中。但对有些绑定的参数希望以引用的方式传递，或是要绑定的参数类型无法拷贝，则传递给bind一个对象而不拷贝它，可以使用标准库ref函数。

ostream &print(ostream &os,const string &s,char c){return os<<s<<c;}

for\_each(words.begin(),words.end(),bind(print,ref(os),\_1,’ ’));

函数ref返回一个对象，包含给定的引用，此对象是可以拷贝的。cref函数生成保存const引用的类。ref、cref、bind定义在functional中。

1. 流迭代器

|  |  |
| --- | --- |
| istream\_iterator<T> in(is);  istream\_iterator<T> eof; | in从输入流is读取类型为T的值  eof为空迭代器，从而表示尾后位置 |

istream\_iterator<int> ii(cin),eof;

vector<int> vec(ii,eof); //用迭代器构造vec

ostream\_iterator<int> oi(cout,” ”);

for(auto c:vec)

\*out\_iter++=c; //将元素写到cout

|  |  |
| --- | --- |
| ostream\_iterator<T> out(os);  ostream\_iterator<T> out(os,d); | out将类型为T的值写到输出流os中  每个值后都输出一个d |

1. 迭代器类别

|  |  |
| --- | --- |
| input iterator | 只读；单遍扫描，只能递增 |
| output iterator | 只写；单遍扫描，只能递增 |
| forward iterator | 可读写；多遍扫描，只能递增 |
| bidirectional iterator | 可读写；多遍扫描 |
| random-access iterator | 可读写；多遍扫描 |

1. list和forward\_list成员函数版本的算法

|  |  |
| --- | --- |
| l.merge(l2)  l.merge(l2,comp) | 使用<运算符将l2的元素合并到l。  使用给定的比较操作。l和l2都必须是有序的。合并之后l2为空。 |
| l.remove(v)  l.remove\_if(uP) | 调用erase删除==v的元素  使unary predicate为真的元素 |
| l.reverse() | 反转l中元素的顺序 |
| l.sort()  l.sort(comp) | 用<排序  用给定的比较操作排序 |
| l.unique()  l.unique(bP) | 使用==  使用给定的binary predicate |

|  |  |  |
| --- | --- | --- |
| args | l.splice(args) | fl.splice\_after(args) |
| (p,l2) l2类型必须与l或fl相同，不能为同一链表；元素从l2中删除 | p是指向l中元素的迭代器。将l2中所有元素移动到l中p之前的位置 | p是指向fl首前位置的迭代器，将l2中元素移动到p之后的位置 |
| (p,l2,p2) | 将p2指向的元素移到l中 | 将p2之后的元素移到fl中 |
| (p,l2,b,e) | 将给定范围中的元素从l2移动到l或fl。l2可以与l或fl为同一链表，但p不能指向给定范围中元素 | |

第11章 关联容器

1. 使用关键字类型的比较函数

bool compareIsbn(const Sales\_data &lhs,const Sales\_data &rhs){

return lhs.isbn()<rhs.isbn();

}

//向books添加元素时调用compareIsbn来为元素排序

multiset<Sales\_data,decltype(compareIsbn)\*> books(compareIsbn);

1. 关联容器操作

|  |  |
| --- | --- |
| c.insert(v)  c.emplace(args) | v是vlaue\_type类型的对象。对于map、set函数返回一个pair，包含一个指向具有指定关键字的迭代器，以及一个指示插入是否成功的bool值；对于multimap、multiset，返回指向新元素的iter |
| c.insert(b,e)  c.insert(li) | b、e表示c::value\_type类型值的范围。函数返回void |
| c.insert(p,v)  c.emplace(p,args) | 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。返回一个指向具有指定关键字的iter |

|  |  |
| --- | --- |
| c.erase(k)  c.erase(p)  c.erase(b,e)  c.clear() | 删除关键字为k的元素，返回size\_type值，指出删除元素的数量  删除p指定的元素。返回指向p之后位置的迭代器。  删除c中所有元素 |

|  |  |
| --- | --- |
| key\_type | 关键字类型 |
| mapped\_type | 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> |

|  |  |
| --- | --- |
| 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；若k不在c中，抛出一个out\_of\_range异常 |

下标和at操作只适用于非const的map、unordered\_map

|  |  |
| --- | --- |
| c.find(k) | 返回指向第一个关键字为k的元素的迭代器，没有则返回c.end() |
| c.count(k) | 返回关键字==k的元素的数量 |
| c.lower\_bound(k) | 返回指向第一个关键字>=k的元素的迭代器 |
| c.upper\_bound(k) | >k |
| c.equal\_range(k) | 返回一个迭代器pair，表示关键字==k的元素的范围，若k不存在，pair的两个成员都==c.end() |

lower\_bound、upper\_bound不适用于无序容器。

第12章 动态内存

q为内置指针

d为类型为D的对象

|  |  |  |
| --- | --- | --- |
| shared\_ptr | unique\_ptr | weak\_ptr |
| p.reset()若p是唯一指向其对象的shared\_ptr，释放对象，将p置为空 | | |
| p.reset(q) 令p指向q p.get()返回p中保存的指针 | | weap\_ptr<T> w(sp)  与shared\_ptr sp指向相同的对象，T必须能转换为sp指向的类型  w.expired()  若w.use\_count()==0,返回true  w.lock()  若w.expired()==true,返回一个空shared\_ptr,否则返回一个指向w的对象的shared\_ptr |
| sp.reset(q,d)  make\_shared<T> (args)  返回一个shared\_ptr，指向一个动态分配的使用args初始化的类型为T的对象  sp.unique()  若sp.use\_count==1,返回ture  spw.use\_count()  返回与sp共享对象的智能指针数  shared\_ptr<T> sp(q)  sp接管q指向的对象；q必须指向new分配的内存  shared\_ptr<T> sp(u)  shared\_ptr<T> sp(q,d)  sp调用对象d来代替delete  shared\_ptr<T> sp(p2,d) | u.reset(nullprt)  unique\_ptr<T> u  unique\_ptr<T,D> u  unique\_ptr<T,D> u(d)  u=nullptr释放u指向的对象，将u置为空  u.release()，返回指针，将u置为空  unique\_ptr<T[]> u  unique\_ptr<T[]> u(p)，u指向p所指向的动态分配的数组  u[i] 返回u指向的数组位置i处的对象 |

unique\_ptr不能直接拷贝或赋值。可以拷贝或赋值一个将要被销毁的unique\_ptr对象，可以从函数返回一个unique\_ptr.

|  |  |
| --- | --- |
| allocator<T> a | 定义一个名为a 的allocateor对象，可以为T类型对象分配内存 |
| 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，返回一个指向初始位置的迭代器 |
| a.construct(p,args) | p必须是T\*类型的指针，指向一块原始内存；args被传递给T类型的构造函数，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，对p指向的对象执行析构函数 |
| a.deallocate(p,n) | 释放T\*类型的指针p中地址开始的内存；p必须是先前由allocate返回的指针，n必须是p创建时所要求的大小。调用之前必须对内存中创建的对象调用destroy |

|  |  |
| --- | --- |
| uninitialized\_copy(b,e,b2) | 从迭代器b、e指出的范围中拷贝元素到b2指向的未构造的原始内存中。b2指向的内存必须能容纳输入序列中元素的拷贝，返回指向尾元素之后位置的iter |
| uninitialized\_copy\_n(b,n,b2) | 从b指向的元素开始，拷贝n个元素到b2指向的内存中 |
| uninitialized\_fill (b,e,t) | 在b、e指向的范围中创建值均为t的对象 |
| 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必须指向能够容纳给定数量对象的未构造的原始内存 |

第13章 拷贝控制

1. 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。

Sales\_data& Sales\_data::operator=(const Sales\_data &);

需要拷贝操作<=>需要赋值操作

1. 析构函数由~接类名构成，没有返回值也不接受参数。

~Sales\_data();

无论何时一个对象被销毁，就会自动调用其析构函数：

* 变量在离开其作用域时被销毁
* 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
* 对于动态分配的对象，当对指向它的指针应用delete运算符时被销毁
* 对于临时对象，当创建它的表达式结束时被销毁

如果一个类需要析构函数，那么它也同样需要拷贝和赋值操作。基类的析构函数并不遵循这个准则。一个基类需要析构函数，能将析构函数设为虚函数而令内容为空，因而无法推断该基类还需要赋值运算或拷贝构造函数。虚析构函数将阻止合成移动操作。

1. 删除的函数：虽然声明了，但不能以任何方式使用。

析构函数不能是删除的；如果析构函数是删除的，则不允许定义该类型的变量或创建该类型的对象。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。

* 如果类的某个成员的析构函数是删除的或不可访问的，或类有一个没有类内初始值的引用成员，或类有一个const成员，它没有类内初始值且其类型未显式定义默认构造函数，则该类的默认构造函数为删除的。
* 如果类的某个成员的析构函数是删除的或不可访问的，则类的合成拷贝构造函数、合成析构函数是删除的。
* 如果类的某个成员的拷贝构造函数是删除的或不可访问的，则类的合成拷贝构造函数是删除的。
* 如果类的某个成员的拷贝(移动)赋值运算符是删除的或不可访问的，或是类有一个const的或引用成员，则类的合成拷贝(移动)赋值运算符是删除的。

定义了一个移动构造函数或移动赋值运算符的类必须也定义自己的拷贝操作，否则这些成员默认的被定义为删除的。

只有当一个类没有定义任何版本的拷贝控制成员，且它的所有数据成员都能移动构造或移动赋值时，编译器才会为它合成移动构造函数或移动赋值运算符。

struct NoCopy{

Nocopy(const NoCopy&)=delete; //阻止拷贝

};

1. 不能将一个右值引用直接绑定到一个左值上，但可以显式的将一个左值转换为对应的右值引用类型。

int &&l=42;

int &&r=std::move(l); //除了对l重新赋值或销毁外不能再使用l的值

可以销毁一个移后源对象或赋予它新值，但不能使用它曾经保存的旧值。

对move不提供using 声明，直接调用std::move。

移动构造函数的第一个参数是该类类型的一个右值引用，任何额外参数都必须有默认值。

不抛出异常的移动构造函数和移动赋值运算符必须标记为noexcept。

如vector保证若调用push\_back时发生异常，vector自身不变。但调用可能会要求vector重新分配内存。若重新分配过程中使用了移动构造函数，且在移动了部分而不是全部元素之后发生了异常，此时vector将不能满足自身保持不变的要求。为了避免这种问题，除非vector知道元素类型的移动构造函数不会抛出异常，否则在重新分配内存的过程中，就必须使用拷贝构造函数而不是移动构造函数。若希望vector重新分配内存这类情况下对自定义类型的对象进行移动而不是拷贝，就必须显式的告诉标准库我们的移动构造函数可以安全使用。

向set插入一个元素要求分配内存可能会抛出bad\_alloc异常，因此不标记为noexcept。

1. 标准库的make\_move\_iterator函数将一个普通迭代器转换为一个移动迭代器。移动迭代

器的解引用运算符生成一个右值引用。迭代器的所有操作在移动迭代器中都照常工作。

auto newcapacity=size()?2\*size():1;

auto first=alloc.allocate(newcapacity);

auto last=uninitialized\_copy(make\_move\_iterator(begin()),

make\_move\_iterator(end()),first);

1. 成员函数通过一个名为this的额外隐式参数来访问调用它的那个对象。默认情况下this

的类型是指向类类型的非常量版本的常量指针，不能把this绑定到一个常量对象上。紧跟在参数列表后面的const表示this是一个指向常量的常量指针。

const成员函数不能改变类成员变量的值，非const成员函数不能被const成员对象调用。常量对象、常量对象的指针或引用都只能调用const成员函数。

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

构造函数不能被声明为const，当我们创建类的一个const对象时，直到构造函数完成初始化过程，对象才能真正取得const属性。因此，构造函数在const对象的构造过程中可以向其写值。

mutable data member永不会是const。const成员函数可改变mutable data member的值。

class Screen{

mutable size\_t access\_ctr;

public:

void some\_member() const;

};

void Screen::some\_member() const{  
 ++access\_ctr;

}

1. 引用限定符可以是&或&&，分别指出this可以指向一个左值或右值。类似const限定符，引用限定符只能用于非static成员函数，且必须出现在函数的声明和定义中。引用限定符跟随在const限定符之后。如果一个成员函数有引用限定符，具有相同参数列表的所有版本都必须有引用限定符。

class Foo{

Foo sorted()&&;//可用于可改变的右值，右值可原址排序

Foo sorted()const &;//可用于任何类型的Foo

Foo sorted()const;//错误，必须加上引用限定符

Foo &operator=(const Foo&)&;//只能向可修改的左值赋值

};

Foo &Foo::operator=(const Foo &rhs)&{

return \*this;

}

第14章 重载运算与类型转换

1. 运算符函数或者是类的成员，或者至少含有一个类类型的参数。可以像调用普通函数那样直接调用运算符函数。

data1+=data2;

data1.operator+=data2;

|  |  |
| --- | --- |
| 不能被重载的运算符 | ：： .\* . ?: |
| 不应该重载的运算符 | , & && || |

* =、[]、()、->必须是成员
* 复合赋值运算符应该是成员
* 改变对象状态的运算符或与给定类型密切相关的运算符，如++、--、\*解引用应该是成员
* 具有对称性的运算符，如算术、相等性、关系、位运算符等应该是非成员函数

1. 函数调用运算符

struct absInt{

int operator()(int val) const {return var<0?-val:val;}};

int i=-42;

absInt absObj;

int ui=absObj(i); //将i传递给absObj.operator()

//将vector中元素传递给IsEqual(3)

std::replace\_if(vec.begin(), vec.end(), IsEqual(3), 5);

1. 标准库function类型

|  |  |
| --- | --- |
| function<T> f; | f是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函数类型T相同(T是retType(args)) |
| function<T> f(nullptr); | 显式的构造一个空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 |
| f(args) | 调用f中的对象，参数是args |

|  |  |
| --- | --- |
| result\_type | 该function类型的可调用对象返回的类型 |
| argument\_type  first\_argument\_type  second\_argument\_type | 若T只有一个实参，argument\_type是该类型的同义词；若T有两个实参，first\_argument\_type和second\_argument\_type分别代表两个实参的类型 |

int add(int,int);

struct divide{};

auto mod=[](int i,int j){return i%j;};

function<int(int,int)> f1=add;

function<int(int,int)> f2=divide();//f2(4,2)=2;

function<int(int,int)> f3=[](int i,int j){return i%j;};

map<string,function<int(int,int)>> b={{“+”,add},{“%”,mod}};

1. 类型转换运算符没有返回类型也没有参数

operator type() const;

operator int()const{return i;}

当表达式出现在下列位置，显式的类型转换被隐式执行：

* if、while、do、for、条件运算符（?:）的条件部分
* 逻辑运算符的运算对象

第15章 面向对象程序设计

1. 对于某些函数，基类希望它的派生类各自定义适合自身的版本，此时基类将这些函数声明成虚函数。

class Quote {

public:

virtual double net\_price(std::size\_t)const;

}

使用基类的引用（或指针）调用一个虚函数时将发生动态绑定。一旦某个函数被声明成虚函数，则在所有派生类中它都是虚函数。

添加关键字显式的注明使用某个成员函数覆盖了它继承的虚函数const & noexcept override;

如果使用override标记了某个函数，但该函数并没有覆盖已知的虚函数，则编译器会报错。

任何尝试覆盖指定为final 的函数将引发错误。

如果某次函数调用使用了默认实参，则该实参由本次调用的静态类型决定，通过基类的引用调用函数则使用基类定义的默认实参。

可以使用域运算符强制执行虚函数的某个特定版本

double u=baseP->Quote::net\_price(42);

1. 纯虚函数无需定义，因为这个函数没有实际意义。可以为pure virtual function提供定义，但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 noexcept =0;

}

含有纯虚函数的类是abstract base class，不能直接创建一个抽象基类的对象。

1. 派生类向基类转换的可访问性

假定D继承自B，DD继承自D

* 只有当D继承B的方式是others，用户代码others才能使用D向B的转换。
* 无论D继承B的方式是什么，D的成员函数和友元都能使用D向B的转换。
* 如果D继承B的方式是others or group，则DD的成员和友元可以使用D向B的转换

第16章 模版与泛型编程

1. 非类型模版参数

template<unsigned N, unsigned M> inline

int compare(const char (&p1)[N],const char(&p2)[M]){

return strcmp(p1,p2);

}

1. 在模版内不能重用模版参数名

typedef double A;

template<typename A, typename B>void f(A a,B b){

A t=a; //t的类型不是double

double B; //错误

}

1. 默认情况下，c++假定作用域运算符访问的名字不是类型，通过使用关键字typename显式告诉编译器该名字是一个类型。

template<typename T>typename T::value\_type f(T &c){

if(!c.empty())

return c.back();

else

return typename T::value\_type();

}

1. 在某些情况下编译器无法推断模版实参的类型。

template<typename T>int compare(const T &v1,const T &v2);

//无法推断T1的类型，每次使用都必须提供显式模版实参

template<typename T1,typename T2,typename T3>T1 cp(const T2 &v1,const T3 &v2);

long lg;

int i;

auto v=cp<double>(i,lg); //double cp(int,long)

compare(lg,1023); //错误:模版参数不匹配

compare<long>(lg,1023); //compare(long,long)

尾置返回类型出现在参数列表之后，可以使用函数的参数。

template<typename T>auto f(const T beg,const T end)->decltype(\*beg){

return \*beg;

}

1. 标准类型转换模版Mod<T>

|  |  |  |
| --- | --- | --- |
| Mod | T | Mod<T>::type |
| remove\_reference | X&或X&&  否则 | X  T |
| add\_const | X&、const X或函数  否则 | T  const T |
| add\_lvalue\_reference | X&  X&&  否则 | T  X&  T& |
| add\_rvalue\_reference | X&或X&&  否则 | T  T&& |
| remove\_pointer | X\*  否则 | X  T |
| add\_pointer | X&或X&&  否则 | X\*  T\* |
| make\_signed | unsigned X  否则 | X  T |
| make\_unsigned | 带符号类型  否则 | unsigned X  T |
| remove\_extent | X[n]  否则 | X  T |
| remove\_all\_extents | X[n1][n2]…  否则 | X  T |

1. X& &、X& &&、X&& &都折叠成类型X&

X&& &&折叠成X&&

1. 在调用中使用std::forward保持类型信息

template<typename T>void f(T &&t){

foo(std::forward<T>(t));

}

如果实参是一个右值，forward<T>返回T&&；如果实参是一个左值，T是一个左值引用类型，forward<T>返回T& &&，折叠成一个左值引用类型。

1. typename…指出接下来的参数表示任意个类型的列表

template<typename T> ostream &f(ostream &os,const T &t){

return os<<t;

}

//Args是模版参数包，rest是函数参数包

template<typename T,typename… Args>

ostream &f(ostream &os,const T &t,const Args&… rest){

//f(os,debug(a1),debug(a2),…)

return f(os,debug(rest)…);

// f(os,debug(rest…))是错误的，没有匹配函数

}

int i=0;double d;string s(“whr”);

f(cout,s,42, “whr”); //包中2个参数

可以使用sizeof…运算符知道包中有多少元素。

cout<<sizeof…(Args)<<endl;

1. 一个特例化版本本质上是一个实例，而不是一个重载版本。

//template<typename T> int compare(const T&,cosnt T&)的一个特例化

// T是const char\*

template<> int compare(const char\* const &l,const char\* const &r){

return strcmp(l,r);

}

第17章 标准库特殊设施

|  |  |
| --- | --- |
| tuple<T1,T2,…> t;  tuple<T1,T2,…> t(v1,v2,…) | 第i个成员的类型为Ti，所有成员进行值初始化  第i个成员用vi初始化，此构造函数为explicit |
| make\_tuple(v1,v2,…) | 返回用给定值初始化和推断类型的tuple |
| t1==t2  t1!=t2 | tuple的关系运算符使用字典序，两个tuple必须有相同数量的成员，使用<比较对应成员 |
| get<i>(t) | 返回t 的第i个成员的引用；若t是左值，返回左值引用，否则返回右值引用 |
| tuple\_size<tupleType>::value | 表示给定tuple中成员数量的public成员constexpr static size\_t value |
| tuple\_element<i,tupleType>::type | 表示tuple中指定成员的类型的public成员type |

|  |  |
| --- | --- |
| bitset<n> b;  bitset<n> b(u); | 两个构造函数都为constexpr  b是unsigned long long u的低n位的拷贝，若n大于unsigned long long的大小，则b 的高位被置零 |
| bitset<n> b(s,pos,m,zero,one);  bistset<n> b(cp,pos,m,zero,one); | b是string s从pos开始m个值的拷贝，s只能包含字符zero或one，pos默认为0，m默认为string::npos,zero默认为’0’，one默认为’1’  cp指向字符数组。若未提供m，则cp必须指向c风格字符串；若提供了m，则从pos开始必须有m个字符 |

|  |  |
| --- | --- |
| b.any() b.all() b.none()  b.test(pos)  b.count()  b.size() | b中是否存在置位的二进制位  位置pos是否置位  b中置位的位数  b中的位数 |
| b.set(pos,v) b.set()  b.reset(pos) b.reset() | 将pos位置为v，v默认为true；将所有位置位  将pos位复位 |
| b.flip(pos) b.flip() | 改变pos位的状态 |
| b[pos] | 访问pos位 |
| b.to\_long() b.to\_llong()  b.to\_string(zero,one) | 返回一个unsigned long。如果b中的位不能放入指定的结果类型会抛出错误 |
| os<<b  is>>b | 将b中的位打印为字符1或0到流os  当下一个字符不是1或0，或已经读入b.size()个位时停止读取 |

1. regex\_search和regex\_match的参数

seq可以是string、表示范围的iter对、指向c风格字符串的指针

r是regex对象

m是match对象，用来保存匹配结果的细节

mft是可选的regex\_constants::match\_flag\_type值，默认值是format\_default

|  |  |
| --- | --- |
| (seq,m,r,mft) (seq,r,mft) | 在seq中查找r，返回bool值指出是否找到匹配 |

string p=“[[:alpha:]]\*[^c]ei[[:alpha:]]\*”;

regex r(p);

smatch result;

string s(“friend receipt theif”);

if(regex\_search(s,result,r))

cout<<result.str()<<endl;

1. regex选项

f是指出对象如何操作的标志：

|  |  |
| --- | --- |
| icase | 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  basic extend awk grep egrep | 默认值  POSIX |

re表示一个正则表达式，可以是string、表示字符范围的迭代器对、指向c风格字符串的指针、字符指针、计数器或{}包围的字符列表

|  |  |
| --- | --- |
| regex r(re); regex r(re,f);  r=re; r.assign(re,f); | 与=效果相同 |
| r.mark\_count() | r中子表达式的数目，子表达式用()括起来 |
| r.flags() | 返回r 的标志集 |

|  |  |
| --- | --- |
| 输入序列类型 | 正则表达式类 |
| string  const char\*  wstring  const wchar\_t\* | regex、smatch、ssub\_match、sregex\_iterator  regex、cmatch、csub\_match、cregex\_iterator  wregex、wsmatch、wssub\_match、wsregex\_iterator  wregex、wcmatch、wcsub\_match、wcregex\_iterator |

1. sregex\_iterator操作

这些操作也适用于cregex\_iterator、wsregex\_iterator、wcregex\_iterator

|  |  |
| --- | --- |
| sregex\_iterator it(b,e,r); | 遍历迭代器b、e表示的string，调用sregex\_search(b,e,r)将it定位到第一个匹配的位置 |
| sregex\_iterator end; | 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++ | 从输入当前匹配位置开始调用regex\_search，返回迭代器 |
| it1==it2  it2!=it2 | 两个尾后迭代器相等；两个非尾后迭代器从相同的输入和regex对象构造，则相等 |

1. smatch操作

这些操作也适用于cmatch、wsmatch、wcmatch、csub\_match、wssub\_match、wcsub\_match

|  |  |
| --- | --- |
| m.ready() | 是否已经通过调用regex\_search或regex\_match设置了m，若m.ready()==false，对m进行操作是未定义的 |
| m.size() | 返回最后一次匹配的正则表达式中子表达式的数目 |
| m.empty() | 是否m.size()==0 |
| m.prefix()  m.suffix() | 一个ssub\_match对象，表示当前匹配之前的序列  之后 |
| m.begin() m.end() | m.cbegin() m.cend() |

for(sregex\_iterator it(s.begin(),s.end(),r),end\_it;it!=end\_it;++it){

auto pos=it->prefix().length();//前缀的大小

pos=pos>40?pos-40:0; //只要最多40个字符

cout<<it->prefix().str().substr(pos)

<<”\n”<<it->str()<<”\n”

<<it->suffix().str().substr(0,40)<<endl;

}

fmt是格式字符串，可以是string或指向c风格字符串的指针

|  |  |
| --- | --- |
| m.format(dest,fmt,mft)  m.format(fmt,mft) | 使用fmt生成格式化输出，写入dest指向的位置 |
| regex\_replace(dest,seq,r,fmt,mft)  regex\_replace(seq,r,fmt,mft) | 遍历seq，用regex\_search查找与r匹配的子串，使用fmt生成输出写入dest |

正则表达式通常用()表示子表达式

string phone=”(\\()?(\\d{3})(\\))?([-. ])?(\\d{3})([-. ])?(\\d{4})”;

string fmt=”$2.$5.$7”; //将号码格式改为ddd.ddd.dddd

n的默认值为0，且n<m.size()，第一个子匹配索引为0，表示整个匹配

|  |  |
| --- | --- |
| m.length(n) | 第n个匹配的子表达式的大小 |
| m.position(n) | 第n个子表达式距序列开始的距离 |
| m.str(n) | 第n个子表达式匹配的string |
| m[n] | 第n个子表达式对应的ssub\_match对象 |

子匹配操作

这些操作适用于ssub\_match、csub\_match、wssub\_match、wcsub\_match

|  |  |
| --- | --- |
| matched | ssub\_match是否匹配 |
| first second | 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==second |
| length() | 匹配的大小 |
| str() | 返回包含输入中匹配部分的string |
| s=ssub | 将ssub\_match对象ssub转化为string s，等价于s=ssub.str() |

1. 随机数引擎操作

s为整形值

u为unsigned long long

|  |  |
| --- | --- |
| Engine e;  Engine e(s); | 使用默认种子的默认构造函数  使用s作为种子 |
| e.seed(s) | 使用种子s重置引擎的状态 |
| e.min() e.max() | 此引擎可以生成的最小/最大值 |
| Engine::result\_type | 此引擎生成的unsigned整形类型 |
| e.discard(u) | 将引擎推进u步 |

1. 分布类型的操作

e是一个随机数引擎对象

|  |  |
| --- | --- |
| Dist d; | explicit的默认构造函数，使d准备好被使用 |
| d(e) | 若用相同的e连续调用d，会根据d的分布式类型生成一个随机数序列 |
| d.min() d.max() | 返回d(e)能生成的最小/最大值 |
| d.reset() | 重建d的状态，使得随后对d的使用不依赖于d已经生成的值 |

1. 定义在iostream中的操作符(并列的第一个为默认流状态)

|  |  |
| --- | --- |
| noboolalpha boolalpha | 将true、false输出为1、0 / 字符串 |
| noshowbase showbase | 不生成 / 对整形生成表示进制的前缀 |
| noshowpoint showpoint | 浮点值包含小数部分才显示小数点/总是显示 |
| noshowpos showpose | 对非负数不显示+ / 总是显示 |
| nouppercase uppercase | 十六进制打印0x，科学计数法打印e / 0X，E |
| dec hex oct | 整型值显示为十进制/十六/八 |
| right left  internal | 在值的左侧/右侧添加填充字符  在符号和值之间 |
| fixed  scientific  hexfloat  defaultfloat | 浮点值显示为定点十进制  科学计数法  十六进制  重置浮点数格式为十进制 |
| nounitbuf unitbuf | 恢复正常的缓冲区刷新方式/每次输出后都刷新 |
| skipws noskipws | 输入运算符跳过空白符/不跳过 |
| flush  ends  endl | 刷新ostream缓冲区  插入空字符然后刷新  插入换行然后刷新 |

定义在iomanip中的操作符

|  |  |
| --- | --- |
| setfill(ch) | 用ch填充空白 |
| setw(w) | 读或写值的宽度为w个字符 |
| setbase(b) b只能为8、10、16 | 将整数输出为b进制 |
| setprecision(n) cout.precision(n)  cout.precision() | 将浮点精度设置为n  返回当前精度值 |

1. 单字节低层IO操作

is是istream，os是ostream

|  |  |
| --- | --- |
| is.get(ch) | 从is读取一个字节存入ch中，返回is |
| os.put(ch) | 将ch输入到os，返回os |
| is.get() | 将is的下一个字节作为int返回 |
| is.putback() | 将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

delim为字符

sink表示字符数组的起始地址

|  |  |
| --- | --- |
| is.get(sink,size,delim)  is.getline(sink,size,delim) | 从is读取最多size字节存入sink，直到遇到delim或到达文件尾；若遇到delim，将其留在流中  读取并丢弃delim |
| is.read(sink,size) | 读取最多size个字节存入sink，返回is |
| is.gcount() | 返回上一个未格式化读取操作从is读取的字节数 |
| os.write(sink,size) | 将sink中的size自己写入os，返回os |
| is.ignore(size,delim) | 读取并忽略最多size(默认1)个字符，包括delim(默认为文件尾) |

1. 流随机访问

g版本表示get，p表示put

pos通常是前一个tellp()或tellg()返回的值

from可以是

beg，偏移量相对于流开始的位置

cur，当前

end，结尾

|  |  |
| --- | --- |
| tellg() tellp() | 返回流中标记的当前位置 |
| seekg(pos) seekp(pos) | 在流中将标记重定位到给定的绝对地址 |
| seekg(off,from) seekp(off,from) | 在流中将标记重定位到from之前或之后off个字符 |

第18章 用于大型程序的工具

1. 栈展开

如果对throw exception的fuction的调用语句位于一个try语句块内，则检查与try关联的catch子句。如果找到了使用该catch处理，如果该try语句嵌套在其他try语句中，则继续检查与外层try匹配的catch子句。

栈展开过程中对象被自动销毁。

1. 要求throw exception的类型与catch声明的类型精确匹配：

* 允许从非const向const转换
* 允许派生类向基类
* 数组、函数向指针

1. 重新抛出

若一个单独的catch语句不能完整的处理某个异常，可以通过一个不含表达式的throw语句将异常传递给调用链更上一层的catch语句。空的throw语句只能出现在catch语句或catch语句直接调用的函数内，否则编译器将调用teminate.

catch(…)可以捕获所有exception

1. noexcept运算符

当e调用的所有函数都作了不抛出说明且e本身不含有throw语句时，noexcept(e)为true，否则noexcept(e)返回false.

//f和g的异常说明一致，noexcept(true)即noexcept

void f()noexcept(noexcept(g()));

不throw exception的指针只能指向不throw exception的函数，可能throw exception的指针可以指向所有函数。

如果virtual fuction承诺不throw exception，则派生出的函数也应该承诺不throw exception;若基类的virtual fuction允许throw exception，派生类的函数随意。

1. 在所有类、函数及命名空间之外定义的名字定义在全局命名空间。全局命名空间以隐式的方式声明，并且在所有的程序中都存在。

::member\_name

表示全局命名空间中的一个成员。

内联命名空间中的名字可以被外层命名空间直接使用。关键字inline必须出现在命名空间第一次定义的地方，后续再打开命名空间的时候可以写inline也可以不写。

inline namespace fifthEd{}

namespace cplusplus\_primer{

#include “fifthEd.h”

#include “fourthEd.h”

}

形如cplusplus\_primer::的代码可以直接获得fifthEd的成员。非内联的须加上完整的名字，如cplusplus\_primer::fourthEd::Query\_base.

未命名的命名空间指关键字后跟{}括起来的一系列声明语句，其中定义的变量拥有静态生命周期，它们在第一次使用前创建，直到程序结束才销毁。

未命名的命名空间可以在某个给定的文件中不连续，但不能跨越多个文件。如果头文件中定义了未命名的命名空间，则其中定义的名字将在每个文件中对应不同实体。

如果未命名的命名空间定义在文件的最外层作用域中，则其中的名字一定要与全局作用域中的名字不同。

namespace local{

namespace{

int i;

}

}

local::i=42;

1. 当传递给函数一个类类型的对象时，除了在常规的作用域中查找外还会查找实参类及类的基类所属的命名空间。这个例外对传递类的引用或指针有效。

namespace A{

class B{

friend void f();//除非另有声明，否则不会被找到

friend void f2(const B&);//根据实参相关的规则可以被找到

}

}

int main{

A::B bobj;

f2(bobj); //在A::B的友元声明中找到A::f2

f();//错误，A::f没有声明

}

1. using声明引入的函数将重载该声明所属作用域中已有的其它同名函数。若using声明出现在局部作用域中，则引入的名字将隐藏外层作用域的声明。若声明所在的作用域已经有一个函数与新引入的函数同名且形参列表相同，则该using声明引发错误。

using指示将命名空间的成员提升到外层作用域，若命名空间的某个函数与该命名空间所属作用域的函数同名，则命名空间的函数将被条件到重载集合中。此时只要指明调用的版本即可。

第19章 特殊工具与技术

1. 定位new表达式

new (place\_address) type//使用operator new(size\_t,void\*),不分配内存，只返回指针实参，然后由new表达式负责在指定的地址初始化对象；无法自定义的operator new版本

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，initializers是一个可能为空的以,分隔的初始值列表

1. dynamic\_cast运算符

dynamic\_cast<type\*>(e)，e必须是一个有效的指针，若转换失败返回0

dynamic\_cast<type&>(e)，e必须是一个左值，若转换失败抛出bad\_cast

dynamic\_cast<type&&>(e)，e不能是一个左值

type必须是一个类类型，并且通常应该含有虚函数；e是type 的public派生类、public基类或与type类型相同。

Query\_base \*q=new AndQuery(Query("fiery"),Query("bird"));

if(typeid(\*q)==typeid(AndQuery));

if(AndQuery \*aq=dynamic\_cast<AndQuery \*>(q));

try{

AndQuery &aq2=dynamic\_cast<AndQuery &>(\*q);

}catch(std::bad\_cast){

}

1. type\_info操作

t1，t2，t为type\_info对象

|  |  |
| --- | --- |
| t1==t2 t1!=t2 | 若t1和t2表示同一种类型，返回true |
| t.name() | 返回c风格字符串，表示类型名字的可打印形式，生成方式因系统而异 |
| t1.before(t2) | 返回一个bool，表示t1是否位于t2之前，采用的顺序关系依赖于编译器 |

1. enum属于字面型常量。enumerator的名字在在enum的作用域外是不可访问的。定义不限定作用域的enum时省略关键字class（或struct），enum的名字是可选的。若enum是未命名的，则只能在定义该enum时定义它的对象。在不限定作用域的enum中，enumerator的作用域与enum的作用域相同。

enum color:long long{red,green=2};//不限定作用域的潜在类型因机器而异

enum class color1{red=3,green=3};//默认成员类型int

color c=green;

color1 c1=green; //错误，color1的成员不在作用域中，color::green类型错误

color1 c2=color1::green;

默认情况下enum值从0开始，依次加1。可以为enumerator指定专门的值。enumerator是const，因此初始值必须是constexpr。

只要enum有名字，就能定义并初始化该类型的成员。必须用该enum的一个enumerator或对象初始化或为enum对象赋值。

1. 数据成员指针

const string Screen::\*p;

p=&Screen::contents;

auto pdata=&Screen::contents; //与上面两条语句等价

Screen s,\*ps=&s;

auto c=s.\*p;

c=ps->\*p;

成员函数指针

using Action=char (Screen::\*)()const;

Action pf1=&Screen::get; //与下面两条语句等价

char (Screen::\*pf)()const;

pf=&Screen::get; //成员函数和指针间不存在自动转换

char c1=(s.\*pf)();

char c2=(ps->\*pf)();

将成员函数作为可调用对象

function<bool (const string&)> fc=&string::empty;

//下面三条语句等价

find\_if(vs.begin(),vs.end(),fc);

//mem\_fn和bind生成的对象可以通过对象或指针调用

find\_if(vs.begin(),vs.end(),mem\_fn(&string::empty)); find\_if(vs.begin(),vs.end(),bind(&string::empty，\_1));

1. 一个union可以有多个数据成员，但是在任意时刻只有一个有值，给某个成员赋值之后，其它成员就变成未定义状态了。

union Token{

char cval;

int ival;

double dval;

};

默认情况下union是未初始化的，可以使用一对{}内的初始值显式的初始化一个union，初始值被用于初始化第一个成员。

Token t{‘a’};

anonymous union在右}和;之间没有任何声明，一旦定义了一个匿名union，编译器会自动创建一个未命名的对象。

union {

char cval;

int ival;

};

cval=’a’; //匿名union所在的作用域内该union的成员都可以直接访问

匿名union不能用于定义对象，不能包含成员函数，不能包含private或protected成员。

算法概览：

b、e是表示元素范围的迭代器；若无e2，则假定b2表示的序列与b、e表示的序列一样大

d是表示目的序列的迭代器，必须保证能保存算法需要生成的元素

uP是unary predicate，bP是binary predicate

uO、bO是unary、binary callable object

v是value

iter是iterator

c是count

简单查找算法

|  |  |
| --- | --- |
| find(b,e,v) find\_if(b,e,uP)  find\_if\_not(b,e,uP)  count(b,e,v) count\_if | 返回指向v的iter，没找到返回e |
| all\_of(b,e,uP)  any\_of  none\_of | 若序列为空，any\_of返回false，  all\_of和none\_of返回true |
| adjacent\_find(b,e) (b,e,bP) | 返回第一对相邻重复元素的iter |
| search\_n(b,e,c,v) (b,e,c,v,bP) | 返回iter，从此位置开始有c个v |
| search(b,e,b2,e2) (b,e,b2,e2,bP)  find\_end | 返回第二个输入范围在第一个中首次出现的位置  最后一次 |
| find\_first\_of(b,e,b2,e2) (b,e,b2,e2,bP) | 返回一个迭代器，指向第二个输入范围中任意元素在第一个范围中首次出现的位置 |
| for\_each(b,e,uO) | 对每个元素调用uO |
| mismatch(b,e,b2) (b,e,b2,bP) | 返回一个迭代器pair，表示两个序列中第一个不匹配的元素。若都匹配，则iter1==e，iter2指向偏移量等于第一个序列长度的位置 |
| equal(b,e,b2) (b,e,b2,bP) | 返回是否相等，第二个序列至少与第一个一样长 |

cp是满足关联容器中对关键字序要求的bP

下表的算法要求有序序列，用<或cp(x,y)检测元素

m是mid

|  |  |
| --- | --- |
| lower\_bound(b,e,v) (b,e,v,cp) | 返回一个iter，指向首个>=v的元素 |
| upper\_bound(b,e,v) (b,e,v,cp) | >v |
| equal\_range(b,e,v) (b,e,v,cp) | 返回一个iter pair，[)，相等的范围 |
| binary\_search(b,e,v) (b,e,v,cp) | 返回bool，指出是否有值==v |
| merge(b,e,b2,e2,d) (b,e,b2,e2,d,cp)  inplace\_merge(b,m,e) (b,m,e,cp) | 将合并后的序列写入d  b、m和m、e之间的序列被合并入原序列 |
| includes(b,e,b2,e2) (b,e,b2,e2,cp) | 若第一个序列包含第二个序列，返回true |
| set\_union(b,e,b2,e2,d) (b,e,b2,e2,d,cp)  set\_intersection(b,e,b2,e2,d) (b,e,b2,e2,d,cp)  set\_difference(b,e,b2,e2,d) (b,e,b2,e2,d,cp)  set\_symmetric\_difference(b,e,b2,e2,d)  (b,e,b2,e2,d,cp) | 创建两个序列的有序序列，写入d，两个序列都出现的元素只保留一个；包括下面的都返回递增后的d  对两个都包含的元素创建一个有序序列  对出现在第一个但不在第二个  对只出现在一个序列中的元素 |

Gen()是生成器对象

ov是old value

nv是new value

写容器元素的算法

|  |  |
| --- | --- |
| fill(b,e,v)  fill\_n(d,c,v)  generate(b,e,Gen)  generate\_n(d,c,Gen) | 将v赋值给每个元素  向d写入c个v，返回写入的末个元素的后一个位置  将Gen()生成的元素赋给每个元素 |
| copy(b,e,d)  copy\_if(b,e,d,uP)  copy\_n(b,c,d)  copy\_backward(b,e,d) | 将输入拷贝到d  拷贝满足uP的元素  拷贝c个元素，输入序列必须至少有c个元素  逆向拷贝到d，返回移动后的d |
| move(b,e,d)  move\_backward(b,e,d) | 对输入的每个元素调用std::move |
| transform(b,e,d,uO) (b,e,b2,d,bO) | 对每个元素调用uO |
| replace\_copy(b,e,d,ov,nv)  replace\_copy\_if(b,e,d,uP,nv)  replace(b,e,ov,nv)  replace\_if(b,e,uP,nv) | 将输入拷贝到d，将ov或满足uP的替换为nv  替换 |
| iter\_swap(iter1,iter2)  swap\_ranges(b,e,b2) | 交换iter1和iter2指向的元素  将所以元素与b2开始的元素交换，返回递增后的b2，指向最后一个交换元素之后的位置 |

划分与排序算法

nth是iter

|  |  |
| --- | --- |
| is\_partitioned(b,e,uP) | 若所有满足uP的元素都在不满足的之前，返回true；序列为空返回true |
| partition\_copy(b,e,d,d2,uP) | 满足uP的元素拷贝到d，不满足的拷贝到d2，返回iter pair，分别指向两个序列末尾 |
| partition\_point(b,e,uP) | 输入序列已经用uP划分过，返回iter指向的元素以及其后的元素都不满足uP |
| stable\_partition(b,e,uP)  partition(b,e,uP) | 使用uP划分序列，返回iter指向partition\_point |
| sort(b,e) (b,e,cp)  stable\_sort(b,e) (b,e,cp) |  |
| is\_sorted(b,e) (b,e,cp)  is\_sorted\_until(b,e) (b,e,cp) | 返回一个bool，指出是否有序  查找最长有序子序列，返回其尾后iter |
| partial\_sort(b,m,e) (b,m,e,cp)  partial\_sort\_copy(b,e,dB,dE) (b,e,dB,dE,cp) | 排m-b个元素，即最小的前m-b个元素  排序输入范围中的元素，最多填满目的范围，返回目的范围有序部分的尾后iter |
| nth\_element(b,nth,e) (b,nth,e,cp) | nth之前的元素都<=\*nth，之后的>=\*nth |

rand接受一个正int i，生成一个[0,i]的服从均匀分布的随机int

Uniform\_rand满足均匀分布随机数生成器的要求

重排算法

|  |  |
| --- | --- |
| remove(b,e,v)  remobe\_if(b,e,uP)  remove\_copy(b,e,d,v)  remove\_copy\_if(b,e,d,uP) | 返回iter，指向最后一个删除元素的尾后位置  将重排后元素写入d |
| unique(b,e) (b,e,bP)  unique\_copy(b,e,d)  unique\_copy\_if(b,e,d,uP) | 删除相邻的重复元素，返回iter，指向不重复元素的尾后位置 |
| rotate(b,m,e)  rotate\_copy(b,m,e,d) | 将m、e之间的元素移动到b、m之间的元素之前；返回指向原\*b的iter |
| reverse(b,e)  reverse(b,e,d) | 翻转序列中的元素，返回void  返回指向目的序列的尾后iter |
| random\_shuffle(b,e) (b,e,rand)  shuffle(b,e,Uniform\_rand) | 随机排列序列中的元素，返回void |

排列算法

|  |  |
| --- | --- |
| is\_permutation(b,e,b2) (b,e,b2,bP) | 若第二个序列的某个排列和第一个序列相同，则返回true |
| next\_permutation(b,e) (b,e,cp) | 若序列已经是最后一个，则重排为最小的序列，返回false；否则将序列转换为下一个排列，返回true |
| prev\_permutation(b,e) (b,e,cp) | previous |

最大值和最小值

il是initial list

|  |  |
| --- | --- |
| min(v,v2) (v,v2,cp) (il) (il,cp)  max(v,v2) (v,v2,cp) (il) (il,cp) | 两个实参的类型必须完全一致，参数和返回类型都是const & |
| minmax(v,v2) (v,v2,cp) (il) (il,cp) | 返回pair，first成员为min，second为max |
| min\_element(b,e) (b,e,cp)  max\_element(b,e) (b,e,cp) | 返回iter |
| minmax\_element(b,e) (b,e,cp) | 返回pair |
| lexicographical\_compare(b,e,b2,e2) (b,e,b2,e2,cp) | 若第一个序列在字典序中小于第二个，返回true；字符相等，较短的小 |

init是initiation

数值算法

|  |  |
| --- | --- |
| accumulate(b,e,init) (b,e,init,bO) | 求和，返回类型与init相同 |
| inner\_product(b,e,b2,init) (b,e,b2,init,bO,bO2) | 求内积，bO替换+，bO2替换\* |
| partial\_sum(b,e,d) (b,e,d,bO) | 新元素的值等于当前元素与之前所有元素的和，返回递增后的d |
| adjacent\_difference(b,e,d) (b,e,d,bO) | 每个(除首个外)新元素的值都等于输入范围中当前元素和前一个元素之差 |
| iota(b,e,v) | 将v 的值赋予首元素并递增val，将递增后的值赋予下一个元素，重复操作 |

随机数：

RealT：float、double、long double

IntT：(unsigned)( short、int、long、long long)

均匀分布：

uniform\_int\_distribution<IntT> u(m,n);

uniform\_real\_distribution<RealT> u(x,y);

m默认为0，n默认为IntT可以表示的最大值；x默认为0.0，y默认为1.0

Bernoulli分布

p默认值0.5

t默认值1，k默认值1

|  |  |
| --- | --- |
| bernoulli\_distribution b(p); | 以给定概率p生成true |
| binomial\_distribution<IntT> b(t,p); | 分布是按采样大小t、概率p生成的 |
| geometric\_distribution<IntT> g(p) | 每次实验成功概率p |
| negative\_binomial\_distribution<IntT> nb(k,p) | k次实验成功概率p |

poisson分布

lam默认值为1.0

a为形状参数α，b为尺度参数β，b默认值1.0

|  |  |
| --- | --- |
| poisson\_distribution<IntT> p(x) | 均值为x的分布 |
| exponential\_distribution<RealT> e(lam) |  |
| gamma\_distribution<RealT> g(a,b) | a默认值1.0 |
| weibull\_distribution<RealT> w(a,b) | a默认值1.0 |
| extreme\_distribution<RealT> e(a,b) | a默认值0.0 |

正态分布

m为均值，默认值为0.0

s为标准差，默认值为1.0

l、n为自由度，默认值为1

|  |
| --- |
| normal\_distribution<RealT> n(m,s) |
| lognormal\_distribution<RealT> ln(m,s) |
| chi\_squared\_distribution<RealT> c(n) |
| cauchy\_distribution<RealT> c(a,b) a默认值0.0 |
| fisher\_f\_distribution<RealT> f(l,n) |
| student\_t\_distribution<RealT> s(n) |

抽样分布

i、j是权重序列的输入iter，il是一个权重的{}列表

b、e、w是输入iter

|  |
| --- |
| discrete\_distribution<IntT> d(i,j)  discrete\_distribution<IntT> d(il) |
| piecewise\_constant\_distribution<IntT> pc(b,e,w) |
| piecewise\_linear\_distribution<IntT> pl(b,e,w) |

随机数引擎

|  |  |
| --- | --- |
| default\_random\_engine | 某个其它引擎类型的类型别名，用于大多数情况 |
| linear\_congruential\_engine | 乘数 模 增量  minstd\_rand0 16807 2147483647 0  minstd\_rand 48271 2147483647 0 |
| mersenne\_twister\_engine | mt19937为32位无符号生成器  mt19937\_64为64位 |
| subtract\_with\_carry\_engine | ranlux24\_base为32位无符号生成器  ranlux48\_base为64位 |
| discard\_block\_engine | 引擎适配器，将底层引擎的结果丢弃。用底层引擎、块大小、旧块大小来参数化  引擎 块大小 旧块大小  ranlux24 ranlux24\_base 223 23  ranlux48 ranlux24\_base 389 11 |
| independent\_bits\_engine | 引擎适配器，生成指定位数的随机数。用底层引擎、结果的位数、保存生成的二进制位的无符号整型类型来参数化。指定的位数必须小于指定的无符号类型所能保存的位数 |
| shuffle\_order\_engine | 引擎适配器。用底层引擎、要混洗的元素数目来参数化。  knuth\_b使用minstd\_rand0，表大小256 |

常见错误：

std:

'string' does not name a type

'runtime\_error' does not name a type

ambiguous:

reference to 'minus' is ambiguous //c++中有minus函数，所以不能命名同名函数

有参数的函数可能因为参数有默认值而引起和无参函数的歧义

'buildMap' was not declared in this scope：

error: ‘trans\_map’ does not name a type

auto trans\_map=buildMap(map\_file);

(perhaps a semicolon is missing after the definition of 'StrBlob')：类名{}之后要加分号;

new types may not be defined in a return type

class StrBlob{

不能拷贝或赋值unique\_ptr：

use of deleted function ‘std::unique\_ptr

名词解释：

sscanf : string scan format