**变量和基本类型**

**字符和字符串字面值**

字符串字面值的类型实际上是由常量字符构成的数组

‘a’ //字符

“Hello World!” //字符串字面值

**初始化和赋值**

初始化的含义是创建变量时赋予一个初始值，而赋值的含义是把对象当前值擦除，而以一个新值来代替

**声明和定义**

declaration: 使名字为程序所知

definition: 创建与名字关联的实体

extern int i; //声明i而非定义i

PS: 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，其实等同于在不同文件中分别定义了独立的变量；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关键词

//file\_1.C定义并初始化一个常量

extern const int bufSize = fcn();

//file\_1.H头文件

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

**作用域**

全局作用域

块作用域

**引用（左值引用）**

引用（reference）为对象另取一个名字，引用类型通常要和与之绑定的对象严格匹配

int i = 1024;

int &refi = i; //refi是i的另一个名字

**指针**

指针是一个对象，允许指针拷贝和赋值；指针无需在定义时赋初值；通常指针类型要和它指向的对象严格匹配（两种例外情况，同引用）

空指针（nullptr或0）

void\*指针

一个void\*指针中存放在一个地址，但我们并不知道该地址中到底是什么类型的对象

取地址符（&） 解引用符（\*）

int ival = 1;

int \*p = &ival;

cout << \*p;

**const限定符**

默认情况下，const对象仅在文件内有效（除非声明extern），const对象必须初始化

const int i = get(); //正确，运行时初始化

const int j = 1; //正确，编译时初始化

**const的引用（简称“常量引用”）**

初始化常量引用时允许任意表达式作为初始值，只要这个表达式能转化为引用的类型即可，尤其，允许一个常量引用绑定到非常量的对象、字面值，甚至是一般的表达式

一个常量引用被绑定到另一种类型的对象时:

double pi = 3.14;

const int &rpi = pi;

上述代码相当于:

const int temp = pi;

const int &rpi = temp; //让rpi绑定这个临时量

对const的引用可能引用一个并非const的对象，但不能让一个非常量引用指向一个常量对象

**指针和const**

指向常量的指针（pointer to const）: 与常量引用相似

常量指针（const pointer）: 指针本身是常量，而不是指针所指的对象，必须初始化，并且初始化完成就不能改变了

int errNum = 0;

int \*const curErr = &errNum; //curErr将一直指向errNum

顶层const(top-level const): 指针本身是常量

底层const(low-level const): 指针所指的对象是常量

当执行拷贝时，拷入和拷出的对象必须具有相同的底层const资质，或者两个对象的数据类型必须能够转换，一般非常量可以转化为常量，反之不然 ---> 不好说

**constexpr和常量表达式**

常量表达式（const expression）是指不会改变并且在编译过程就能计算结果的表达式

const int sz = get\_size(); //sz不是常量表达式

声明为constexpr的变量一定是一个常量，而且必须用常量表达式初始化

constexpr int sz = get\_size(); //当get\_size是一个constexpr函数时才是一条正确的声明语句 const int \*p = nullptr;

constexpr int \*q = nullptr;

p是一个指向常量的指针，q是一个常量指针

* constexpr把它所定义的对象置为顶层const
* constexpr函数的参数和返回值必须是字面值类型

PS: 常量表达式是指在编译和执行值不会改变且可以立即得到值的表达式；并非所有const对象都是常量表达式，const仅标记对象为只读属性，对象在初始化过后无法改变；C++中允许将变量声明为constexpr类型以使编译器在编译时检查该变量是否是常量表达式；只有那些在编译时就确定地址指向的const指针才是constexpr(nullptr, NULL或指向固定地址的指针)；对于定义在函数体外的变量，其指针是constexpr的；此类变量要占用数据段，而程序运行时，代码段和数据段大小位置均不会改变，因此编译器可以确定地址指向的，是constexpr的；此外，函数内部定义的static静态变量，也会在数据区使用固定地址，因此指针也是constexpr的(程序在内存中的组织形式是段，有堆栈段、数据段和代码段)

**字面值类型(literal type)**

* 算术类型，引用和指针类型（自定义类不属于字面值类型）
* 字面值常量类 -> 1)数据成员必须都是字面值类型；2)类内必须至少含有一个constexpr构造函数；3)含有类内初始值的数据成员的初始值必须是一条常量表达式或成员自己的constexpr构造函数；4)类必须使用析构函数的默认定义
* 略

constexpr函数要求：唯一执行语句就是返回语句

constexpr构造函数：必须初始化所有数据成员，初始值或使用constexpr构造函数

**类型别名**

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

using SI = sales\_item; //SI是sales\_item的同义词

**auto类型说明符**

使用引用其实是使用引用的对象，特别是当引用被用作初始值时，真正参与初始化其实是引用对象的值，编译器以引用对象的类型作为auto的类型

auto一般会忽略掉顶层const，同时底层const则会保留下来，decltype则不会转换

数组作为auto变量的初始值时，推断得到的类型是指针而非数组

int ia[] = {1,2,3,4,5} //ia是一个含有5个整数的数组

auto ib(ia); //ib是一个整型指针，指向ia的第一个元素

decltype(ia) ic; //ic是一个含有5个整数的数组

**decltype类型说明符**

希望从表达式推断出要定义变量的类型

decltype(fun()) sum = x; //sum的类型就是函数fun的返回类型

decltype会返回表达式的类型（包括顶层const和引用在内）

int i = 42, \*p = &i;

decltype(\*p) dp; //错误，dp是int&，必须初始化

解引用指针可以得到指针所值的对象，因此，decltype(\*p)的结果类型是int&

decltype((var))的结果永远是引用

**字符串、向量和数组**

**初始化**

拷贝初始化（copy initialization）

string str1 = “hiya”;

直接初始化（direct initialization）

string str2(“hiya”);

列表初始化（list initialization）// 列表初始化也可能会调用构造函数

如果存在信息丢失的风险，则编译器会报错,列表初始化只能用花括号{}

vector<T> v{a, b, c};

vector<T> v = {a, b, c}; //等价于v{a, b, c}

默认初始化（default initialization）

没有指定初始值，定义在函数体内部的内置类型变量将不被初始化

值初始化（value-initialized）

* 在我们忽略初始值仅提供元素数量的情况下，元素value-initialized
* Element initializer depends on the type of the elements only

vector<int> ivec(10); //10个元素，值初始化

**vector标准库**

vector是一个类模板

关于初始化: 如果用圆括号()则是通过提供的值来构造（construct）vector对象，如果用花括号{}则是列表初始化，如果使用了花括号但提供的值又不能用来列表初始化，就要考虑通过这样的值来构造vector对象了

vector<int> v(10, 1); //10个元素都是1

vector<int> v{10, 1}; //2个元素，分别是10和1

v.push\_back(t); //向v的尾端添加一个值为t的元素

**迭代器iterator**

begin()和cbegin()

end()和cend() //off-the-end

\*iter //解引用

iter->mem //等价于(\*iter).mem

const\_iterator和指向常量的指针类似，能读取但不能修改所指的元素值；如果vector对象或string对象是一个常量，只能使用const\_iterator;如果不是常量，那么既能使用iterator也能使用const\_iterator

**数组**

int \*ptrs[10]; //ptrs是含有10个整型指针的数组

int (\*arrPtr)[10] = &arr; //arrPtr指向一个含10个整数的数组

int (&arrRef)[10] = arr; //arrRef引用一个含有10个整数的数组

数组的操作实际上是指针的操作

**表达式**

**左值和右值**

左值可以位于赋值语句的左侧，右值则不能

当一个对象被用作右值时，用的是对象的值（内容）；当对象被用作左值的时候，用的是对象的身份；当一个左值被当成右值使用时，实际上是使用它的内容（值）

取地址符作用于一个左值运算对象，返回一个指向该运算对象的指针，这个指针是一个右值

decltype作用于求值结果为左值的表达式得到一个引用类型

\*++iter; //先递增然后解引用

**递增和递减运算符**

前置版本将对象本身作为左值返回，后置版本则将对象原始值的副本作为右值返回（建议: 除非必须，否则不用后置版本）

**成员访问运算符**

ptr->mem等价于(\*ptr).mem

箭头运算符作用于一个指针类型的运算对象，结果是一个左值；点运算符结果（左值或右值）与成员所属对象有关

**sizeof**

sizeof运算符返回一条表达式结果类型的大小或一个类型名字所占的字节数

sizeof (type)

sizeof expr

对string或vector对象执行sizeof运算只返回该类型固定部分的大小

**类型转换**

隐式转换（implicit conversion）

算数类型之间的隐式转换被设计得尽可能避免进度损失 — integral promotion

数组表达式中，数组自动转换成指向数组首元素的指针，decltype, &(取址), sizeof和typeid除外

命名的强制类型转换

cast\_name<type>(expression);

static\_cast: 任何具有明确定义的类型转换，只要不包含底层const

const\_cast: 改变对象的底层const - 去掉const性质

reinterpret\_cast: 底层

dynamic\_cast: 主要用于在继承体系中安全的向下转换，是实现多态的一种方式

**实参类型转换 (注: 这里的前提是传值, 而非传引用)**

1. 精确匹配

* 实参类型和形参类型相同
* 实参从数组类型或函数类型转换成对应的指针类型
* 向实参添加顶层const或者从实参中删除顶层const

1. 通过const转换实现的匹配（将非常量转换为常量）
2. 通过类型提升实现匹配
3. 通过类类型转换实现匹配

**语句**

**for语句**

//传统for语句

for (init-statement; condition; expression)

statement

//范围for语句

for (declaration : expression)

statement

// i.e.

string s(“Hello World!!!”);

for (auto &c : s)

c = toupper(c); //把s中的字符变为大写

cout << s <<endl;

**跳转语句**

break: 负责终止离它最近的while, do while, for或switch语句

continue: 终止最近的循环中的当前迭代并立即开始下一次迭代

**函数**

形参（parameter）

实参（argument）

调用运算符（call operator）

* 调用函数: 一是用实参初始化函数对应的形参，二是将控制权转移给被调函数，此时主调函数（calling function）暂时中断，被调函数（called function）开始执行
* 函数的返回类型不能是数组类型或函数类型，但可以是指向数组或函数的指针
* 自动对象（automatic object）: 只存在于块执行期间的对象
* 局部静态对象（local static object）: 直到程序终止才被销毁

关键词: static

* 函数声明（或称function prototype）三要素: 返回类型，函数名，形参类型

**参数传递**

引用传递（passed by reference）或传引用调用

值传递(passed by value)或传值调用

指针形参: 通过指针间接访问它所指的对象，达到引用传递的效果

如果函数无须改变引用形参的值，最好声明为常量引用

当用实参初始化形参时会忽略掉顶层const，形参的初始化方式和变量的初始化方式是一样的

**数组形参**

void print(const int\*);

void print(const int[]);

void print(const int[10]);

尽管形式不同，但三个函数是等价的，每个函数的唯一形参都是const int\*

管理指针的常用技术

1）使用标准库规范

void print(const int\* beg, const int\* end) //传递指向首元素和尾后元素的指针

2）显式传递一个表示数组大小的形参

void print(const int \*ia, size\_t size)

数组引用形参

//i.e.

void print(int (&arr)[10]) //只能作用于大小为10的数组

**initializer\_list形参**

实现处理不同数量实参的函数，initializer\_list是一种标准库类型，用于表示某种特定类型的值的数组

initializer\_list<T> lst{a,b,c...}

lst.size()

lst.begin()

lst.end()

返回类型和return语句

* 返回一个值的方式和初始化一个变量或形参的方式完全一样
* 如果函数返回引用，则这个引用仅是它所引对象的一个别名
* 不要返回局部对象的引用或指针，因为函数完成后，它会被销毁
* 调用一个返回引用的函数得到左值，其他返回类型得到右值
* 函数可以返回花括号包围值的列表（列表初始化返回值）

**递归（recursive）**

函数调用自身

int factorial(int val){

if (val > 1)

return factorial(val – 1) \* val;

return 1;

} //recursive loop

**返回数组指针**

typedef int arr[10]; //arr是一个含有10个整数的数组的类型别名

using arr = int[10]; //arr的等价声明

arr\* func(int i) //func返回一个指向含有10个整数的数组的指针

Type (\*function(patameter\_list))[dimension] //返回数组指针的函数

auto func(int i) -> int(\*)[10] //tailing return type

使用decltype

int odd[] = {1,3,5,7,9}

int even[] = {0,2,4,6,8}

decltype(odd) \*func(int i){

return (i % 2) ? &odd : &even;

} //decltype并不负责把数组类型转化成对应的指针，所以func声明时加一个\*符号

**函数重载**

一个拥有顶层const的形参无法和另一个没有顶层const的形参区分开来

如果形参是指针或引用，则可以区分指向常量对象还是非常量对象实现重载，底层const

Record lookup(Phone);

Record lookup(const Phone); //重复声明

Record lookup(Phone\*);

Record lookup(Phone const\*) //重复声明

Record lookup(Phone&);

Record lookup(const Phone&); //重载

Record lookup(Phone\*);

Record lookup(const Phone\*); //重载

* 在不同作用域无法重载函数名，内层作用域声明的名字将隐藏外层作用域的同名实体
* 内联函数（inline）可避免函数调用的开销
* constexpr函数: 能用于常量表达式的函数，函数的返回类型及形参类型都必须是字面值类型，且函数体中有且只有一条return语句，constexpr函数被隐式地指定为inline
* 把内联函数和constexpr函数放在头文件内
* assert预处理宏和NDEBUG预处理变量

**函数匹配**

* 确定候选函数（candidate function）- 声明在调用点可见的重载函数集
* 确定可行函数（viable function）
* 寻找最佳匹配

**函数指针**

函数的“类型”由它的返回类型和形参类型共同决定与函数名无关

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

bool (\*pf)(const string&, const string&); //pf指向一个函数

当把函数名作为一个值使用时，函数名会自动转换成指针

pf = lengthComapre; //pf指向名为lengthCompare的函数

pf = &lengthCompare; //等价的赋值语句

我们还能直接使用指向函数的指针调用该函数，无需解引用

auto b1 = pf(“Bang”, “Bad”); //调用lengthCoampare

auto b2 = (\*pf)(“Bang”, “Bad”); //等价调用

重载函数的指针: 指针类型与重载函数必须精确匹配

//i.e. Func是函数类型

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

typedef decltype(lengthCompare) Func; //等价声明

using Func = bool(const string&, const string&); //等价声明

// pFunc是指向函数的指针

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

typedef decltype(lengthComapre) \*pFunc; //等价声明

using pFunc = bool(\*)(const string&, const string&); //等价声明

**类**

data abstraction(interface & implementation) | encapsulation

* 在成员函数内部可以通过this（指向类类型非常量版本的常量指针）直接调用对象成员
* 常量成员函数（const member function）: 此时this是一个指向常量的指针
* 定义类相关的非成员函数（类接口的一部分），这些函数的声明应该与类在同一个头文件内

**构造函数**

* 合成默认构造函数（synthesized default constructor）: 如果存在类内初始值，则用它来初始化成员，否则，默认初始化成员
* 只有当类没有声明任何构造函数时，编译器才会自动生成默认构造函数

Sales\_date() = default; //默认构造函数

构造函数初始值列表（constructor initialize list）

注意区分构造函数的初始化（构造函数初始值列表）和赋值（构造函数体内部）的区别，随着构造函数体开始执行，初始化就完成了；初始化const或者引用类型的成员唯一的机会就是通过构造函数初始值列表

类内初始值必须使用“=”的初始化形式或花括号括起来的直接初始化形式

**拷贝、赋值和析构**

如果不主动定义，编译器会自动生成这些操作，当类需要分配类对象之外的资源，合成版本常常会失效

public: 成员在整个程序内可访问

private: 成员可被类的成员函数访问

**友元（friend）**

* 类允许其他类或者函数访问它的非公有成员，友元声明只能出现在类定义的内部，一般来说最好在类开始或结束前位置集中声明友元
* 友元的声明仅仅指定了访问权限，而非一个通常意义的函数声明！！！
* 如果我们希望类的用户能够调用某个友元函数，那么我们必须在友元声明之外再专门对函数声明一次！！！
* 类之间的友元: 如果一个类指定了友元，则友元类的成员函数可以访问此类的所有成员
* 友元关系不存在传递性
* 定义在类内部的成员函数是自动inline的，我们可以在成员函数的声明或定义的地方说明inline，不过，最好只在类外部定义的地方说明inline
* 可变数据成员（mutable data member） - 关键字: mutable
* 一个const成员函数如果以引用形式返回\*this，那么它的返回类型是常量引用

类的定义分两步: 1)编译成员的声明 2)直到类全部可见后再编译函数体

**构造函数再探**

* 如果成员是const、引用或属于某种未提供默认构造函数的类类型，则必须通过构造函数初始值列表为这些成员提供初值
* 最好令构造函数初始值顺序与成员函数声明顺序保持一致
* 如果一个构造函数的所有实参都提供了默认值，实际上也定义了默认构造函数
* 委托构造函数（delegating constructor）
* 隐式类类型转换 – 编译器只允许一步类类型转换
* 抑制构造函数定义的隐式转换 – explicit
* 关键字explicit只对一个实参构造函数有效，需要多个实参构造函数不能用于执行隐式转换，explicit只允许在类内构造函数处声明

**类静态成员**

* 类静态成员存在于任何对象之外，静态成员函数也不与任何对象绑定在一起，不包含this指针，静态成员函数不能声明成const，静态成员static只能在类内部声明
* 静态数据成员不由类构造函数初始化，必须在类外部定义和初始化静态成员
* constexpr静态成员可以在类内初始化
* 即使一个常量静态数据成员已经在类类初始化了，也应该在类外部定义一下
* 静态成员和指针成员可以是不完全类型（如静态成员所属的类）
* 静态成员可以作默认实参

**聚合类(aggregate class)**

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

**IO库**

(w)istream/(w)ifstream/(w)istringstream – 从stream/file/string读取数据

(w)ostream/(w)ofstream/(w)ostringstream – 向stream/file/string写入数据

(w) – 操纵宽字符wchar\_t

fstream和stringstream都继承自iostream

通常可以将一个派生类（继承类）对象当作其基类对象来使用

**管理缓冲区**

每个输出流都管理一个缓冲器，用来保存程序读写的数据

显式刷新缓冲区: flush, endl, ends

一个输出流可能关联到另一个流，cin和cerr都关联到cout，意味着所有输出，包括用户提示信息，都会在读操作之前被打印出来

cout << unitbuf; //所有输出操作后立即刷新缓冲区

cout << nonunitbuf; //回到正常缓冲方式

**顺序容器**

**概述**

vector – 可变大小数组，随机访问

deque – 双端队列，随机访问

list – 双向链表，双向顺序访问

forward\_list – 单向链表，单向顺序访问

array – 固定大小数组，随机访问,更安全的数组类型

string – 与vector相似

任何位置插入/删除数据 <---> 快速随机访问

执行访问操作更多还是插入/删除更多决定了容器类型的选择

使用迭代器，不使用下标操作，避免随机访问

**容器操作**

iterator, const\_iterator //迭代器

size\_type, difference\_type //整型，足以储存容器大小和距离

value\_type //元素类型

reference, const\_reference //相当于value\_type&或const value\_type&

c.size(), c.max\_size, c.empty() //大小，forward\_list不支持size

c.begin(), c.end() //获取迭代器

reverse\_iterator, const\_reverse\_iterator //按逆序寻址的迭代器，不支持forward\_list

c.rbegin(), c.rend() //返回c的尾元素和首元素之前位置的迭代器

C c(b, e) //c初始化为迭代器b(begin)和e(end)指定范围内元素拷贝，元素类型必须相容

//assign（赋值操作）仅用于顺序容器，允许不同但相容的类型赋值，返回void

seq.assign(b, e), seq.assign(ini\_lst), seq.assign(n, t)

c.resize(n), c.resize(n, t) //改变容器大小

* 当一个容器初始化为另一个容器的拷贝时，容器类型和元素类型必须相同（相容还是相同？）
* 标准库array具有固定大小，大小是array类型的一部分
* swap(c1, c2), c1.swap(c2) //交换c1和c2中的元素
* swap不交换元素本身，仅交换两个容器的数据结构（array除外）
* 赋值运算会导致左边容器的迭代器、引用和指针失效，而swap则不会（array和string除外），仍指向swap操作之前的那些元素

**添加元素**

c.push\_back(t), c.emplace\_back(args) //返回void

c.push\_front(t), c.emplace\_front(args)

//在迭代起p指向的元素之前添加元素，返回指向新添加第一个元素的迭代器

c.insert(p, t), c.emplace(p, args)

* 向一个vector, string或deque添加/删除对象会使迭代器，引用和指针失效
* 将元素插入到vector，string和deque的任何位置都是合法的，但可能很耗时
* 当调用push或insert时，我们将元素类型的对象传递给它们，这些对象被拷贝到容器中；当调用emplace时，则将参数传递给元素类型的构造函数，这些参数在容器管理的内存空间中直接构造元素

**访问元素**

front(), back() //首或尾元素，forward\_list不支持back

c[n], c.at(n) //at和下标操作只适用于string, vector, deque和array

vector<int> ivec;

cout << ivec.at(0); //抛出out\_of\_range异常

**删除元素**

c.pop\_back(), c.pop\_front() //forward\_list不支持pop\_back

//vector和string不支持pop\_front

//返回一个指向被删除元素之后的迭代器

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

c.clear() //清空

**forward\_list的特殊操作**

lst.before\_begin() //返回首前迭代器（off-the-beginning）

//在p之后

lst.insert\_after(p)

lst.emplace\_after(p, args)

lst.erase\_after(p) //返回一个指向被删除元素之后的迭代器或尾后迭代器，删除p指向位置之后的元素

* 由于向迭代器添加/删除元素可能会使迭代器失效，因此必须保证每次改变容器操作之后正确地重新定位迭代器，这个建议对vector, string和deque尤为重要
* 不要保存end返回的迭代器

**容量管理的成员函数**

c.capacity() //不重新分配内存的话，c可以保存多少元素

c.reserve(n) //分配至少能容纳n个元素的空间

c.shrink\_to\_fit() //将capacity()减少为与size()相同大小

**容器适配器（adaptor）**

三个顺序容器适配器: stack, queue和priority\_queue

一个适配器接收一种已有的容器类型，使其行为看起来像一种不同的类型

* 构造函数: 1)默认构造函数创建一个空对象；2)接收一个容器的构造函数通过拷贝容器来初始化适配器
* 默认情况stack和queue基于deque实现，priority\_queue基于vector实现
* 可以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<int> stk(deq); //假设deq是deque<int>, 从deq拷贝元素到stk

stack(<string, vector<string>) str\_stk; //str\_stk在vector上实现

stack可以通过array和forward\_list之外任何容器类型构造

queue通过list和deque构造

priority\_queue通过deque和vector构造，支持随机访问

stack<int> i\_stk; //空栈

for (size\_t ix = 0; ix != 10; ++ix)

i\_stk.push(ix);

while(!i\_stk.empty()){

int val = i\_stk.top(); //使用栈顶代码

i\_stk.pop(); //弹出栈顶元素，继续循环

}

**初识泛型算法**

* 一些算法从两个容器中读取元素，构成两个序列的元素可以来自不同类型的容器，两个序列中元素类型也不要求严格匹配
* 用单一迭代器表示第二个序列的算法都假定第二个序列至少和第一个序列一样长
* 标准库算法对迭代器而不是容器进行操作，因此算法不能直接添加/删除元素

fill\_n(b, n, t) //b指向一个元素，从b开始至少包含n个元素

vector<int> vec; //空向量

fill\_n(vec.begin(), 10, 0); //灾难

插入迭代器(insert iterator): 保证算法有足够元素空间来容纳输入数据，插入迭代器是一种向容器中添加元素的迭代器

back\_inserter接收一个指向容器的引用，返回一个与容器绑定的插入迭代器

vector<int> vec; //空向量

fill\_n(back\_inserter(vec), 10, 0) //正确

//拷贝算法接收三个迭代器，目的序列至少要包含与输入序列一样多的元素

int a1[] = {1,2,3,4,5};

int a2[5];

//数组拷贝，copy返回其目的位置迭代器（递增后）的值，ret恰好是尾后

auto ret = copy(begin(a1), end(a1), a2);

**替换算法replace**

replace(ilst.begin(), ilst.end(), 0, 1); //把所有值为0替换为1

//替换后储存在ivec, ilst不变

replace\_copy(ilst.begin(), ilst.end(), back\_inserter(ivec), 0, 1);

**排序算法**

sort(words.begin(), words.end()); //按字典排序words

//将相邻的重复项“消除”，重新排序，返回一个指向不重复值范围末尾之后的迭代器

auto end\_uni = unique(words.begin(), words.end());

//erase删除迭代器开区间[b, e)范围内指向的元素

word.erase(end\_uni, words.end()); //删除重复

**谓词(predicate)**

谓词是一种可调用的表达式，其返回结果是一个能作为条件的值，接受谓词参数的算法对输入序列中的元素调用谓词

**lambda表达式**

可调用对象(callable object): 函数或函数指针，重载了函数调用运算符的类或lambda表达式

一个lambda表达式表示一个可调用的代码单元，可以理解为一个未命名的内联函数

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

capture list是一个lambda所在函数中定义的局部变量列表

//可以忽略参数列表和返回类型，但必须包含捕获列表和函数体

auto f = [] {return 42;}

* 如果lambda的函数体包含任何单一return语句之外的内容且未指定返回类型，则返回void
* 默认情况，如果一个lambda体包含return语句之外的任何语句，则编译器假定返回void，被推断返回void的lambda不能返回值；lambda体是单一return语句时，无须直接指定返回类型而可以直接推断出来
* 当需要为一个lambda定义返回类型时，必须使用尾置返回类型

//使用lambda来调用stable\_sort

stable\_sort(words.begin(), words.end(),

[](const string &a, const string &b)

{ return a.size() > b.size(); });

* 一个lambda通过局部变量包含在其捕获列表中来表明将会使用这些变量，只有在捕获列表中捕获了所在函数中的局部变量，才能在lambda函数体中使用
* 捕获列表只用于局部非static变量，lambda可以直接使用局部static变量和所在函数之外声明的名字

for\_each(words.begin(), words.end(), [](const string &s)

{ cout << s << “ ”; });

auto f = [v] {return v;} //值捕获，局部变量v必须能拷贝

auto f = [&v] {return v;} //引用捕获

* 函数可以直接返回一个可调用对象(lambda)，或者返回一个类对象，如果函数返回一个lambda，则与函数不能返回一个局部变量的引用类似，此lambda也不能包含引用捕获

// = 代表值捕获方式，& 代表引用捕获方式（隐式捕获）

wc = find\_if(words.begin(), words.end(), [=](const string &s)

{return s.size() >= sz;}) //sz为隐式捕获，值捕获方式

//可变lambda – 关键词: mutable

auto f = [v] () mutable {return ++v;};

**参数绑定**

* 如果lambda捕获列表为空，通常可以用函数来代替，但对于捕获局部变量的lambda，用函数代替就没那么容易了

标准库bind函数 – 可以看作一个General Function Adapter

auto newCallable = bind(callable, arg\_list);

* arg\_list可能包含形如\_n的名字 – 占位符
* \_n的名字定义在一个名为placeholders的命名空间，placeholders本身定义在std命名空间

using std::placeholders::\_1; //i.e. \_1对应的using声明

//bind i.e.

bool check\_size(const string &s, size\_t sz);

auto check6 = bind(check\_size, \_1, 6);

wc = find\_if(words.begin(), words.end(), bind(check\_size, \_1, sz));

**迭代器再探 - 标准库iterator**

**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insert的迭代器

当调用inserter(c, iter)时，得到一个迭代器，会将元素插入到iter原来所指元素之前的位置

\*it = val; //it是由inserter生成的迭代器

//等价代码

it = c.insert(it, val); //it指向新加入的元素

++it;

**iostream迭代器**

istream\_iterator读取输入流，ostream\_iterator向输出流写数据

istream\_iterator<int> int\_iter(std::cin); //从std::cin读取int

istream\_iterator<int> eof; //默认初始化迭代器创建尾后迭代器

//i.e. istream\_iterator

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

vector<int> vec(in\_iter, eof); //从迭代器范围构造vec

//i.e. ostream\_iterator

ostream\_iterator<int> out\_iter(cout, “ ”);

for (auto e : vec)

\*out\_iter++ = e; //赋值语句实际上将元素写到cout

**反向迭代器reverse\_iterator**

除了forward\_list之外，其它都支持反向迭代器

普通迭代器和反向迭代器反映了左闭合区间的特性

**迭代器类别**

输入迭代器 – 只读，不写；单遍扫描，只能递增: istream\_iterator

输出迭代器 – 只写，不读；单遍扫描，只能递增: ostream\_iterator

前向迭代器 – 可读写；多遍扫描，只能递增 : forward\_list

双向迭代器 – 可读写；多遍扫描，可递增递减

随机访问迭代器 – 支持全部迭代器运算

**算法的形参模式**

*alg*(beg, end, *args*);

*alg*(beg, end, dest, *args*);

*alg*(beg, end, beg2, end2, *args*);

dest参数表示算法可以写入的目的位置的迭代器，算法假定(assume): 不管写入多少元素都是安全的，dest常常绑定到一个插入迭代器或一个ostream\_iterator

//\_if版本的算法 - 接收一个谓词代替元素值

find(beg, end, val); //查找val第一次出现的位置

find\_if(beg, end, pre); //查找第一个令pre为真的元素

* 一个链表(list/forward\_list)可以通过改变元素间的链接而不是真正交换它们的值来快速“交换”元素
* 对于list和forward\_list应该优先使用成员函数版本的算法而不是通用算法

**关联容器**

关联容器支持通过关键字高效查找和提取元素，对关键字的使用将关联容器和顺序容器区分开来，顺序容器是通过位置访问元素的

三个维度: 1) set or map; 2) 是否重复关键词 – multi; 3) 无序或者有序

map: 关键字-值对的集合 {*key*, *value*} 元素包含*key*和*value*

set: 关键字集合

* 关联容器不支持顺序容器的位置相关的操作，关联容器的迭代器都是双向的
* 对于有序容器，关键字必须定义元素比较的方法；传递给排序算法的可调用对象必须满足与关联容器中关键字一样的类型要求

严格弱序(strict weak ordering): less than

**pair类**

pair<T1, T2> p; pair<T1, T2> p(v1, v2); <-> pair<T1, T2> p = {v1, v2};

make\_pair(v1, v2); //返回一个pair对象，pair的类型从v1和v2推断出来

pair的公有数据成员: pair.first & pair.second

**关联容器额外的类型别名**

key\_type

mapped\_type

value\_type //对于set, 与key\_type相同

//对于map, 为pair<const key\_type, mapped\_type>

* 在一个map中，每个元素是一个pair对象，我们不能改变一个元素的关键字，因为这些pair的关键字部分是const的
* set的关键字是const的，set的迭代器是const的，可以用迭代器（不管是iterator, 还是const\_iterator）读取关键字，但不能修改

**添加元素**

//返回一个pair，包含一个指向指定关键字的元素的迭代器，和一个指示插入是否成功的bool值，

//对multimap和multiset总会插入成功，并返回一个指向新元素的迭代器，v是value\_type类型的对象

c.insert(v), c.emplace(args)

//返回void，b和e是迭代器，表示c::value\_type类型值的范围

c.insert(b, e), c.insert(il)

//迭代器p指示从哪里开始搜索新元素应该储存的位置，返回一个迭代器，指向具有给定关键字的元素，

//emplace通过args来构造一个元素

c.insert(p, v), c.emplace(p, args)

**删除元素**

c.erase(k) //返回一个size\_type值，指出删除元素的数量，k是关键字

e.erase(p) //返回一个指向p之后元素的迭代器

c.erase(b, e) //返回e

**下标操作**

c[k] //返回关键字为k的元素(mapped\_type)，如果k不在c中，

//则创建一个元素插入到map中，关联值将进行值初始化

c.at(k) //若k不在c中，抛出out\_of\_range异常

**访问元素**

c.find(k) //返回一个指向第一个关键字为k的元素的迭代器

c.count(k) //返回关键字等于k的元素的数量

//不适用于无序容器

c.lower\_bound(k), c.upper\_bound(k)

c.equal\_range(k) //返回一个迭代器pair, 表示关键字等于k的元素的范围

**无序容器(unordered associative container)**

不使用比较运算符来组织容器，二是使用一个哈希函数(hash function)和关键字类型的==运算符

**管理桶**

The unordered containers are organized as a collection of buckets, each of which holds zero or more elements. These containers use a hash function to map elements to buckets. To access an element, the container first computes the element’s hash code, which tells which bucket to search. The container puts all of its elements with a given hash value into the same bucket. If the container allows multiple elements with a given key, all the elements with the same key will be in the same bucket. As a result, the performance of an unordered container depends on the quality of its hash function and on the number and size of its buckets.

桶接口

c.bucket\_count() - 正在使用的桶的数目

c.max\_bucket\_count() - 容器能容纳最多的桶的数目

c.bucket\_size(n) – 第n个桶有多少个元素

c.bucket(k) – 关键字为k的元素在哪个桶

桶迭代

local\_iterator, const\_local\_iterator – 可以用来访问桶中元素的迭代器类型

c.begin(n), c.end(n) – 桶n的首元素迭代器和尾后迭代器

哈希策略

c.load\_factor() - 每个桶的平均元素数量，返回float值

c.max\_load\_factor() - c添加新桶的方式使得load\_factor<=max\_load\_factor

c.rehash(n) – 重组存储，使得bucket\_count>=n && bucket\_count>size/max\_load\_factor

c.reserve(n) - 重组存储，使得c可以保存n个元素且不必rehash

* hash<key\_type>类型的对象来生成每个元素的哈希值
* 标准库为内置类型（包括指针）提供了hash模板，以及一些标准库类型和智能指针

**动态内存**

静态内存（局部static对象，类static数据成员和定义在函数之外的变量）、栈内存（保存定义在函数内的非static对象）

程序用来存储动态分配(dynamically allocate)的对象 – 自由空间或堆(heap)

shared\_ptr – 允许多个指针指向同一个对象

unique\_ptr - “独占”所指的对象

weak\_ptr – 伴随类，弱引用，指向shared\_ptr所管理的对象

**shared\_ptr和unique\_ptr都支持的操作**

shared\_ptr<T> sp, unique\_ptr<T> up

\*p – 解引用

p->mem – 等价于(\*p).mem

p.get() - 返回p中保存的指针（内置指针）

PS: 使用get返回的代码不能delete此指针

swap(p, q), p.swap(q)

**shared\_ptr的操作**

make\_shared<T>(args) – 返回一个shared\_ptr，指向一个用args初始化的对象

shared\_ptr<T> p(q) – p是q的拷贝，此操作递增会q的计数器，q中的指针必须能转化成T\*

p = q – p和q都是shared\_ptr, 所保存的指针必须能相互转换，此操作会递减q的引用计数，递增q的引用计数

p.unique() - 若p.use\_count()为1, 返回true, 否则返回false

p.use\_count() - 返回与p共享对象的智能指针的量

* 类似容器emplace成员，make\_shared用参数来构造给定类型的对象
* 每个shared\_ptr都关联一个计数器，称为引用计数(reference count)

**直接管理内存**

使用new动态分配和初始化对象

int \*p = new int; //默认初始化

int \*p = new int(1024); //直接初始化

vector<int> \*pv = new vector<int>{0, 1, 2}; //列表初始化

内置指针管理的动态内存在被显式释放之前一直都存在，内置类型的对象被销毁时什么也不会发生，特别是，当一个指针离开其作用域，它所指的对象什么也不会发生

**释放动态内存**

delete p; //p必须指向一个动态分配的对象或一个空指针

空悬指针(dangling pointer): delete之后重置指针值...

**shared\_ptr & new**

接收指针参数的智能指针构造函数是explicit的，不能将一个内置指针隐式地转换为一个智能指针，必须使用直接初始化形式

shared\_ptr<int> p = new int(1024); //错误!

shared\_ptr<int> p(new int(1024)); //正确: 直接初始化

一个用来初始化智能指针的普通指针必须指向动态内存(new)

不要混合使用普通指针和智能指针

不要使用get初始化另一个智能指针或为智能指针赋值

* 不使用相同的内置指针初始化（或reset）多个智能指针
* 不delete get()返回的指针
* 不使用get()初始化或reset另一个智能指针

**unique\_ptr操作**

u = nullptr – 释放u指向的对象，将u置为空

u.release() - u放弃对指针的控制权，返回指针，并将u置空

u.reset(), u.reset(q), u.reset(nullptr) – 如果提供了内置指针q,令u指向这个对象；否则将u置为空

**weak\_ptr**

weak\_ptr指向一个shared\_ptr管理的对象，weak\_ptr不控制所指对象生存期

由于对象可能不存在,我们不能使用weak\_ptr直接访问对象,而必须调用lock检查weak\_ptr所指向的对象是否存在

//若expired为true, wp.lock()返回一个空的shared\_ptr, 否则返回一个指向共享对象的shared\_ptr

if (shared\_ptr<int> sp = wp.lock()) { /.../ }

**动态数组**

int \*pia = new int[42];

//等价定义

typedef int arrT[42]; //using arrT = int[42];

int \*pia = new arrT;

delete [] pia; //释放动态数组

动态数组并不是数组类型，这个很重要！动态数组类似于分配内存的操作

**allocator类**

allocator<T> a – 创建allocator对象: 为类型为T的对象分配内存

a.allocate(n) – 分配内存

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) – 释放内存

a.construct(p, args) – 在p指向的对象通过args构造一个对象

allocator类的伴随算法，可以在未初始化内存中创建对象

//用于在给定目的位置创建元素，而不是由系统分配内存给他们

uninitialized\_copy(b, e, b2), uninitialized\_copy\_n(b, n, b2)

uninitialized\_fill(b, e, t), uninitialized\_fill\_n(b, n, t)

类似于copy, uninitialized\_copy返回（递增后的）目的位置迭代器，即指向最后一个构造元素之后的位置

**拷贝控制**

constructor(构造): 同类型的对象初始化本对象时做什么

assignment(赋值): 同类型的对象赋予同类型的另一个对象时做什么

拷贝构造函数: 一个构造函数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

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: 对类类型的成员，会使用其拷贝构造函数来拷贝；内置类型的成员则直接拷贝

拷贝初始化: 要求编译器将右侧运算对象拷贝到正在创建的对象中，如果需要的话还要进行类型转化

直接初始化: 要求编译器使用普通的函数匹配来选择我们提供的参数最匹配的构造函数

注意区分拷贝初始化和直接初始化的关系，以及拷贝初始化和拷贝赋值运算的关系

拷贝初始化通常使用拷贝构造函数来完成(或拷贝赋值运算)，拷贝初始化被用来初始化非引用类类型参数

string dots(10, ‘.’); //direct initialize

string s1(dots); //direct initialize

string s2 = dots; //copy initialize

string nines = string(10, ‘9’); //copy initialize

* 在函数调用过程中，具有非引用类型的参数要进行拷贝初始化；拷贝构造函数被用来初始化非引用类类型参数
* 不能隐式使用一个explicit构造函数

vector<int> v1(10); //正确，直接初始化

vector<int> v2 = 10; //错误，接受大小参数的构造函数是explicit的

编译器可以绕过拷贝构造/移动构造函数，但他们必须存在且可访问（不能是private的）

string id = “1-999-9999”; //拷贝初始化

string id(“1-999-9999”); //编译器略过拷贝构造函数直接创建对象

**拷贝赋值运算符（合成拷贝赋值运算符 - 编译器自动生成）**

若未定义，则编译器自动生成synthesized copy-assignment operator

//等价于合成拷贝赋值运算(for all non-static member object)

Sale\_date& Sales\_date::operator=(const Sales\_date &rhs){

bookNo = rhs.bookNo; //调用string::operator=

units\_sold = rhs.units\_sold; //使用内置的int赋值

//...

return \*this; //返回一个此对象的引用

}

**析构函数**

在一个构造函数中，成员初始化是在函数体执行之前完成的，且按照它们在类中出现的顺序进行初始化；在一个析构函数中，首先执行函数体，然后销毁成员（按成员初始化顺序的逆序销毁）；析构部分是隐式的，内置类型没有析构；隐式销毁一个内置指针类型的成员不会delete它所指的对象；通常，析构函数释放对象生存周期分配的所有资源

什么时候析构: 无论何时一个对象被销毁时，自动调用析构函数

* 变量离开其作用域时被销毁
* 容器被销毁时，其元素被销毁
* 动态分配的对象，指针应用delete运算符时对象被销毁
* 临时对象，当创建它的完整表达式结束时被销毁

当指向一个对象的引用或指针（内置指针）离开作用域时，（对象的）析构函数不会执行

如果一个类需要自定义析构函数，那么也需要自定义拷贝构造函数和拷贝赋值运算符

合成析构函数(synthesized destructor) - 编译器自动生成的

认识到析构函数体自身并不直接销毁成员非常重要

析构函数不能是删除的成员；对于析构函数是删除的类型，不能定义该类型的变量或释放该类型动态分配对象的指针

**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

可以将拷贝控制成员或默认构造函数定义为=default来显式的要求编译器生成合成的版本

//i.e.

Sales\_data() = default;

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

**阻止拷贝**

定义删除的函数(deleted function) - 拷贝构造函数或拷贝赋值运算

NoCopy(const NoCopy&) = delete; //阻止拷贝

NoCopy& operator=(const NoCopy&) = delete; //阻止赋值

* 如果类的某个成员的拷贝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，则类的合成拷贝赋值运算被定义为删除的（可能原因: const和引用成员类型必须初始化，const成员不能的值不能修改，引用不能二次绑定，所以不支持拷贝赋值操作）
* 如果类的某个成员的析构函数是删除的或不可访问的，或是类有一个引用成员且没有类内初始化器，或是类有一个const成员，没有类内初始化器且其类型未显式定义默认构造函数，则该类合成构造函数被定义为删除的（所以当我们定义一个引用类型或定义一个const数据类型时应该初始化）
* 本质上，当不可能拷贝、赋值或销毁成员时，类的合成拷贝控制成员就被定义为删除的

PS: 类内初始值(in-class initializer) - 创建对象时，类内初始值将用于初始化数据成员，没有初始值的成员将被默认初始化

通过将拷贝构造函数和拷贝赋值运算声明（但不定义）为private的也可以阻止拷贝；声明但不定义一个成员函数是合法的，成员函数或友元函数中的拷贝操作将会导致链接时错误（试图访问一个未定义的成员）

**行为像值的类**

+拷贝构造函数拷贝指针成员指向的对象，而不是拷贝指针

+析构函数释放指针指向的对象

+通常，赋值运算符组合了析构函数和构造函数的操作，因为赋值运算需要销毁左侧对象的资源

**行为像指针的类**

+拷贝构造函数拷贝指针本身而不是指针指向的对象

+引用计数(reference count): 记录有多少对象共享状态

**交换操作**

资源管理类（如容器）通常会定义一个名为swap的函数 - 通常一次拷贝两次赋值

除array外，swap不对任何元素进行拷贝、删除或插值操作

swap函数应该调用swap，而不是std::swap

//i.e. 定义Foo的swap函数，Foo有一个HasPtr成员h

void swap(Foo &lhs, Foo &rhs){

using std::swap;

swap(lhs.h, rhs.h); //使用HasPtr版本的swap

//交换其他

}

拷贝并交换 - 在赋值运算符中使用swap

//i.e. 注意rhs是按值传递的，意味着HasPtr的拷贝构造函数将右侧对象拷贝到rhs

HasPtr& HasPtr::operator=(HasPtr rhs){

swap(\*this, rhs); //交换左侧运算对象和局部变量rhs的内容

return \*this; //rhs被销毁，从而delete了rhs中的指针

}

//但rhs按传值传递的话，讲道理就需要调用拷贝赋值运算，这样程序就陷入了循环，会不会没对？

//函数实参初始化非引用类型形参时调动拷贝构造函数而不是拷贝赋值运算!

//不是这样的，按传值传递或以值的方式从函数返回的话会调用拷贝构造函数

**对象移动**

有些类包括不能被共享的资源(如unique\_ptr指针或IO缓冲)，它们不能拷贝只能移动

右值引用(rvalue reference)

+只能绑定到一个将要销毁的对象

+一般，一个左值表达式表示一个对象的身份，一个右值表达式表示一个对象的值

+右值引用也不过是某个对象的另一个名字而已

+返回非引用类型的函数，连同算术、关系、位以及后置递增/递减运算符，都生成右值

+变量可以看作只有一个运算对象而没有运算符的表达式，变量表达式都是左值

标准库move函数

int &&ri = std::move(i); //OK, i是int类型变量

使用move的代码应该使用std::move而不是move, 以避免名字冲突

**移动构造函数和移动赋值运算 - 从给定的对象“窃取”资源而不是拷贝资源**

一旦完成移动，源对象必须不再指向被移动的资源——这些资源的所有权已经归属新创建的对象

StrVec::StrVec(StrVec &&s) noexcept //指出移动操作不应抛出任何异常

: elements(s.elements), first\_free(s.first\_free), cap(s.cap)

{

s.elements = s.first\_free = s.cap = nullptr;

}

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

移动赋值运算符执行与析构函数和移动构造函数相同的工作

移动后源对象必须可析构，移动操作对移后源对象的值没有任何要求（即不依赖于移后源对象中的数据）

我们不能对移动源后对象做任何假设

**合成移动操作**

* 若不声明自己的拷贝操作(拷贝操作包括拷贝构造函数和拷贝赋值运算符)，编译器会自动生成，拷贝操作要么被认定为逐成员拷贝，要么被定义为对象赋值，要么被定义为删除函数
* 只有当一个类没有定义任何自己版本的拷贝控制成员，且类的每个非static数据成员都是可以移动的，编译器才会为它合成移动操作
* 编译器可以移动内置成员，若类类型成员有对应的移动操作，编译器也能移动这个成员
* 移动操作永远不会隐式地定义为删除的函数，除非我们显式地要求编译生成=default的，且编译器不能移动所有成员，则编译器会定义移动操作为删除的函数
* 什么时候将合成移动操作定义为删除的函数遵循与定义删除的合成拷贝操作类似的原则
* 如果类定义了移动构造函数或移动赋值运算，则该类合成拷贝构造函数和拷贝赋值运算会被定义为删除的

**移动右值，拷贝左值...**

**...但如果没有移动构造函数，右值也被拷贝，拷贝赋值运算符类似**

建议: 如果一个类定义了任何一个拷贝操作，它就应该定义所有五个操作

**移动迭代器(move interator)**

一个迭代器解引用返回一个指向元素的左值，而一个移动迭代器解引用生成一个右值引用

//i.e.

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

建议不要随意使用移动操作，当我们调用std::move必须确定移后源对象没有其他用户

**右值引用和成员函数**

void push\_back(const X&); //拷贝: 绑定到任意类型的X

void push\_back(X&&); //移动: 只能绑定到类型X的可修改的右值，因为我们从实参“窃取”数据

我们可以在一个对象上调用成员函数，不管对象是左值还是右值

引用限定符(reference qualifier) - 强制左侧运算对象（即this指向的对象）是一个左值（或右值，&&）

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

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，只能用于非static成员函数，同时出现在声明和定义，若函数const, 引用限定符必须位于const限定符之后

可以综合引用限定符和const来区分一个函数的重载版本（非const成员优先调用非const版本）

**重载运算与类型转换**

* 除了重载的函数调用运算符，其它的重载运算符不能含有默认实参
* 当一个重载运算是成员函数时，this绑定到左侧运算对象
* 运算对象求值顺序无法应用到重载运算符上(逻辑与运算，逻辑或运和逗号运算)
* 选择作为成员或者非成员原则:

具有对称性的运算符可能转换任意一端的运算对象，例如算术、相等性、关系和位运算符等，通常应该是普通的非成员

* 当把运算符定义为对象时，左侧运算对象显然应该是运算符所属类的一个对象

**重载输出运算符<<**

ostream& operator<<(ostream &os, const Sales\_data &item){ /.../ }

第二个形参一般是常量

**重载输入运算符>>**

istream &operator>>(istream &is, Sales\_data &item){ /.../ }

第二个形参必须是非常量；输入运算符必须处理可能会失败的情况，而输出运算符不需要

**算术和关系运算符**

如果定义了算术运算符，也应该定义相应的复合运算符，通常应该使用复合运算符；算术运算符的结果常常位于一个局部变量之内；通过复合运算来定义算术运算是一个好主意

赋值运算符

StrVec& operator=(initializer\_list<string>); //接收花括号内的元素列表作为参数

**下标运算符 - operator[]**

最好定义下标运算符的常量版本和非常量版本，当作用于一个常量对象时，则下标运算符返回常量引用

**递增/递减运算符**

StrBlobPtr& operator++(); //前置版本，返回递增/递减后对象的引用

StrBlobPtr operator--(int); //后置版本（接受一个额外的int类型的形参），返回对象的原值（递增/递减之前的值）

通常通过前置版本来定义后置版本: ++\*this/--\*this

**成员访问运算符**

operator\*()和operator->()

箭头运算符必须是类的成员，解引用运算符通常也是类的成员；箭头运算符必须用于获取成员

i.e. 表达式point->mem

1) point是一个内置的指针类型，等价于(\*point).meme

2) point是重载了operator->的类的一个对象，等价于point.operator->()->mem

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\

1、如果返回值是一个原始指针，那么就将运算符的右操作数当作这个原始指针所指向类型的成员进行访问；

2、如果返回值是另一个类型的实例，那么就继续调用这个返回类型的operator->()，直到有一个调用返回一个原始指针为止，然后就按第一种情况处理。

\\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**函数调用运算符**

像函数一样使用的对象；函数调用运算符必须是成员函数；一个类可以定义多个不同版本的函数调用运算符，相互之间应该在参数数量或类型上有所区别

//i.e.

int operator()(int val){ /.../ }

void operator()(const string &s) const { /.../ }

函数对象常常作为泛型算法的实参

lambda是函数对象: 当我们编写了一个lambda后，编译器将表达式翻译成一个未命名类的未命名对象

默认情况下lambda不能改变它捕获的变量，因为默认情况下，由lambda产生的类当中的函数调用运算符是一个const成员函数；如果lambda声明成可变(mutable), 则不再是const的

//f是函数对象

auto f = [capture list](parameter list) mutable -> return type {function body}

通过值捕获的变量被拷贝到lambda中，这种lambda产生的类必须为每个变量建立对应的数据成员，同时创建构造函数；lambda产生的类不含默认构造函数、赋值运算符和默认析构函数

**标准库定义的函数对象**

//i.e. - #include <functional>

plus<double> doubleAdd; //可执行double加法的函数

比较两个指针会产生未定义的行为，关联容器使用less<key\_type>对元素排序，因此可以定义一个指针的set或者在map中使用指针作为关键值而无须直接声明less

vector<string\*> nameTable;

sort(nameTable.begin(), nameTable.end(), less<string\*>()); //标准库规定指针的less是定义良好的

**可调用对象与function**

可调用对象也有类型；两个不同类型的可调用对象却可能共享同一种调用形式(call signature); 一种调用形式对应一个函数类型

int(int, int) - 接受两个int, 返回一个int

定义一个函数表(function table)用于存储指向这些可调用对象的“指针”

int add(int i, int j) {return i + j;} //普通函数

auto mod = [](int i, int j) {return i % j;} //lambda产生一个未命名的函数对象类

map<string, int(\*)(int, int)> binops;

binops.insert({“+”, add}); //正确: add是一个指向正确类型函数的指针

binops.insert({“%”, mod}); //错误: mod不是一个函数指针

标准件function类型可解决上述错误 -> 可调用对象模板

function是一个模板，表示某种调用形式

function<T> f; - 可调用对象形式应该与函数类型T相同(T是*renType(args)*)

fucntion<T> f(nullptr); - 显示地构造一个空function

function<T> f(obj); - f存储可调用对象obj的副本

f(args) - 调用f中的对象

function<T>的成员类型

result\_type, argument\_type, fisrt\_argument\_type, second\_argument\_type

//i.e.

map<string, function<int(int, int)>> binops = {

{“+”, add},

{“-”, std::minus<int>}, //标准库函数对象

{“%”, mod}, //命名了的lambda

{“\*”, [](int i, int j){return i \* j;} //未命名的lambda

};

重载的函数与function

我们不能直接将重载函数的名字存入function类型的对象中；可以通过存储函数指针或lambda来消除二义性（因为指针可以避开重载函数的“同名”，lambda会指定参数类型）

**类型转换运算符**

operator type() const;

类型转换运算符既没有显式的返回类型，也没有形参，必须定义成类的成员函数，一般应该被定义成const成员

隐式的用户定义的类型转换可以置于一个标准（内置）类型转换之前或之后

显示的类型转换运算符(explicit conversion operator)

explicit operator type() const; //static\_cast<type>(T)

除非if, while和for等条件表达式部分，或者逻辑运算(包括? :运算), 显示的类型转换将不能隐式地执行

* 最好不要创建两个转换源或转换对象都是算术类型的类型转换
* 无论什么时候在条件中使用流对象，都会使用IO类型定义的operator bool
* 调用重载函数时，最好不要存在使用构造函数或强制类型转换来改变实参类型
* 表达式中运算符的候选函数集既包括成员函数，又包括非成员函数

重载之类的二义性问题挺复杂的，遇到的时候参考C++ Primer