## 第二章 变量和基本类型

**声明**（declaration）是让名字为程序所知

**定义**（definition）创建与名字相关的实体

**extern**可以声明而不定义

变量可以被多次声明，但只能被定义一次

**指向常量的指针**（底层const） const double \*cptr

**常量指针**（顶层cost） int \*const ptr

**常量表达式**（constexpr）编译就可以得到计算结果

**decltype**：操作是返回操作数的数据类型，但是并不实际计算表达式

## 第四章 表达式

对象被当做右值的时候，用的是对象的值（内容）

对象被当做左值的时候，用的是对象的地址

左值可以代替右值，反过来不行。

**dynamic\_cast：**

运行时类型转换，可以向下转换，父类指针转换成子类指针，比static\_cast多了类型检查的功能

**const\_cast：**

去掉const，针对指针和引用，去除指向对象的常量性

**static\_cast，**

一种类型转换成另一种类型（基本类型之间的转换、任何类型转换成void类型）

**reinterpret\_cast：**

改变指针或者引用的类型，或者指针类型与整数类型相互转换

## 第六章 函数

**形参与实参**：形参是函数定义中的参数，实参是调用函数时传递的参数，实参会传递值给形参

**返回数组指针**：Type (\*function(parameter\_list))[dimension]

可以使用尾置返回类型：auto func(int i) -> int(\*)[10];

**函数重载**：名字相同但是形参不同的叫重载函数

二义性调用：两个匹配的都不是最佳选择

**函数指针**：

void ff(unsigned int);

void (\*pf1)(unsigned int) = ff; // pf1 points to ff(unsigned)

## 第七章 类

**常成员函数**：函数后面加const，表示不会修改任何成员变量的值，可用于区分重载

const对象的指针为 const classA\* this,只能调用const对象

非const对象也可以调用const成员函数，但是cosnt对象只能调用const成员函数

**默认构造函数**：如果存在类内初始值，用它来初始化，否者默认初始化

某些类不能依赖于合成的默认构造函数

1. 类没有任何构造函数才会生成默认构造函数，如果存在一个就没有默认构造函数
2. 对于包含内置类型或者复合类型的成员，只有成员全部被赋予类内初始值才适合
3. 如果类包含其他类型的成员，该成员没有默认构造函数，则不会生成默认构造函数

可以使用 = default来生成厂默认构函数

某些类不能依赖合成的拷贝、复制、析构函数。尤其是需要分配类之外的资源。

**构造函数初始值列表**：Sales\_data(const std::string &s): bookNo(s) { }

**友元**：允许其他类或函数访问私有变量，友元只是指明了权限，并不是声明

友元函数可以在类的内部定义，但是必须要在外部提供相应的声明令函数可见

友元函数没有传递性

**类类型**：声明一个类而不定义，叫做前向声明，这时类是一个不完全类型

**隐式类型转换**：构造函数只接受一个实参，则默认定义了隐式转换机制

可以使用explicit关键字阻止，只对接受一个实参的构造函数有效，只能在类内使用

## 第十二章

**shard\_ptr：**

shard\_ptr<int> p3 = make\_shard<int>(42);

shard\_ptr<int> p(q, d) d是可调用对象代替delete

reset经常与unique一起使用，reset会更新引用计数，放弃之前的所有权，指向新的地址

if (!p.unique())

p.reset(new string(\*p)); // we aren't alone; allocate a new copy

**unique\_ptr：**

unique\_ptr<int> p2 = make\_unique<int>(42);

unique\_ptr<string> p2(p1.release()); // p1放弃所有权给p2

unique\_ptr不能拷贝，除非是即将销毁的unique\_ptr，例如函数返回值

**weak\_ptr：**

weak\_ptr指向由shard\_ptr管理的对象，但不会改变shard\_ptr的引用计数

auto p = make\_shared<int>(42);

weak\_ptr需要由shard\_ptr来初始化

weak\_ptr<int> wp(p);

使用weak\_ptr访问对象时，必须先调用lock函数。该函数检查weak\_ptr指向的对象是否仍然存在。如果存在，则返回指向共享对象的shared\_ptr，否则返回空指针。

if (shared\_ptr<int> np = wp.lock())

{

// true if np is not null

// inside the if, np shares its object with p

}

**动态数组：**

unique\_ptr：支持管理动态数组，shard\_ptr不支持，除非使用自定义删除器

unique\_ptr<int[]> up(new int[10]);

up.release(); // automatically uses delete[] to destroy its pointer

shared\_ptr<int> sp(new int[10], [](int \*p) { delete[] p; });

sp.reset(); // uses the lambda we supplied that uses delete[] to free the array

**allocator**：allocator类将内存分配和构造分开

allocator<string> alloc;

auto const p = alloc.allocate(n); // 分配了n个没有构造的string

auto q = p; // q will point to one past the last constructed element

alloc.construct(q++); // \*q 是空字符串

alloc.construct(q++, 10, 'c'); // \*q is cccccccccc

alloc.construct(q++, "hi"); // \*q is hi!

while (q != p)

alloc.destroy(--q); // 销毁构造的string

alloc.deallocate(p, n); //回收空间

## 第十三章

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

**拷贝构造函数：**

第一个参数是对自身类型的引用（几乎都是const），其他额外参数都有默认值

通常不是explict，因为某些情况下会隐式调用

合成拷贝构造函数会逐个将非static拷贝到对象下

**拷贝赋值运算符：**

Foo& operator=(const Foo&);

未定义则会有合成拷贝赋值运算符，将右侧的非static成员逐个拷贝到左边，然后返回左边对象的引用

**析构函数：**

不接受参数

合成析构函数函数体为空

首先执行函数体再销毁对象

**三/五法则：（如果需要一个，其他的也都需要）**

需要析构函数的类一般也需要拷贝和赋值操作。

需要拷贝操作的类一般也需要赋值操作，反之亦然。

**=default/=delete**

=delete可以对任何函数使用，=default只能对有合成版本的函数使用

行为像值的类：

class HasPtr

{

public:

HasPtr(const std::string &s = std::string()):

ps(new std::string(s)), i(0) { }

// each HasPtr has its own copy of the string to which ps points

HasPtr(const HasPtr &p):

ps(new std::string(\*p.ps)), i(p.i) { }

HasPtr& operator=(const HasPtr &);

~HasPtr() { delete ps; }

private:

std::string \*ps;

int i;

};

赋值运算符需要注意是不是自身赋值，可以先拷贝到临时对象再释放之前的对象

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

{

auto newp = new string(\*rhs.ps); // copy the underlying string

delete ps; // free the old memory

ps = newp; // copy data from rhs into this object

i = rhs.i;

return \*this; // return this object

}

行为像指针的类：

class HasPtr

{

public:

// constructor allocates a new string and a new counter, which it sets to 1

HasPtr(const std::string &s = std::string()):

ps(new std::string(s)), i(0), use(new std::size\_t(1)) {}

// copy constructor copies all three data members and increments the counter

HasPtr(const HasPtr &p):

ps(p.ps), i(p.i), use(p.use) { ++\*use; }

HasPtr& operator=(const HasPtr&);

~HasPtr();

private:

std::string \*ps;

int i;

std::size\_t \*use; // member to keep track of how many objects share \*ps

};

析构函数释放内存前应该判断是否还有其他对象指向这块内存。

HasPtr::~HasPtr()

{

if (--\*use == 0)

{ // if the reference count goes to 0

delete ps; // delete the string

delete use; // and the counter

}

}

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

{

++\*rhs.use; // increment the use count of the right-hand operand

if (--\*use == 0)

{ // then decrement this object's counter

delete ps; // if no other users

delete use; // free this object's allocated members

}

ps = rhs.ps; // copy data from rhs into this object

i = rhs.i;

use = rhs.use;

return \*this; // return this object

}

**对象移动：**

IO类和unique\_ptr类可以移动不能拷贝

右值只能绑定到即将销毁的变量上

int &&rr1 = 42;

int &&rr2 = std::move(rr1);

移动构造函数和移动赋值运算符（属于优化手段，没有的话，即使拷贝右值也是由拷贝操作来处理）

通常需要加noexcept，声明和定义中都需要加

因为STL容量resizeing经常用移动构造函数来移动move\_if\_noexcept()，为了保证数据安全，会调用有noexcept的移动构造函数，否则会调用拷贝构造函数，因为如果抛出异常会导致正在处理的原始数据丢失。

当类没有任何拷贝控制成员且每个非static数据成员都可以移动的时候，编译器才会合成移动构造函数和移动赋值运算符

定义了移动构造函数或者移动赋值运算符的类必须也定义自己的拷贝操作，否则这些成员会默认定义为删除的函数

HasPtr& operator=(HasPtr rhs) //非引用的可以即实现拷贝也实现赋值

引用限定符：

规定this对象是左值还是右值

用&放在函数的后面，必须同时出现在函数的声明和定义中

## 第十四章 重载运算与类型转换

**友元：**

重载二元运算符的时候，常常需要友元，例如A = 2.75 \* B，就不能对int做重载

**输出运算符**：

ostream &operator<<(ostream &os, const Sales\_data &item)

{

os << item.isbn() << " " << item.units\_sold << " "

<< item.revenue << " " << item.avg\_price();

return os;

}

**输入运算符**：必要要处理输入失败的情况，出现失败的情况，负责从错误状态恢复过来

istream &operator>>(istream &is, Sales\_data &item)

{

double price;

is >> item.bookNo >> item.units\_sold >> price;

if (is) // 检查输入是否成功

item.revenue = item.units\_sold \* price;

else

item = Sales\_data(); // 输入失败，赋予默认状态

return is;

}

**算数运算符**：

Sales\_data operator+(const Sales\_data &lhs, const Sales\_data &rhs)

{

Sales\_data sum = lhs; // 复制lhs成员

sum += rhs; // 把rhs的值添加给sum

return sum;

}

**相等运算符**： ==和！=，应该把其中一个工作委托给另一个

bool operator==(const Sales\_data &lhs, const Sales\_data &rhs)

{

return lhs.isbn() == rhs.isbn() &&

lhs.units\_sold == rhs.units\_sold &&

lhs.revenue == rhs.revenue;

}

bool operator!=(const Sales\_data &lhs, const Sales\_data &rhs)

{

return !(lhs == rhs);

}

**赋值运算符**：包括复合赋值运算符

StrVec &StrVec::operator=(initializer\_list<string> il)

{

auto data = alloc\_n\_copy(il.begin(), il.end());

free();

elements = data.first;

first\_free = cap = data.second;

return \*this;

}

Sales\_data& Sales\_data::operator+=(const Sales\_data &rhs)

{

units\_sold += rhs.units\_sold;

revenue += rhs.revenue;

return \*this;

}

**下标运算符**：通常两个版本，普通引用和常量引用，因为非const版本需要**修改返回值**，const版本不允许修改。

std::string& operator[](std::size\_t n)

{ return elements[n]; }

const std::string& operator[](std::size\_t n) const

{ return elements[n]; }

递增递减运算符：分为前缀版本和后缀版本

class StrBlobPtr

{

public:

// increment and decrement

StrBlobPtr& operator++(); // prefix operators

StrBlobPtr& operator--();

StrBlobPtr operator++(int); // postfix operators

StrBlobPtr operator--(int);

};

前缀版本：增加后直接返回目前的值

StrBlobPtr**&** StrBlobPtr::operator++()

{

++curr;

return \*this;

}

后缀版本：保存当前的值后再修改对象

StrBlobPtr StrBlobPtr::operator++(**int**)

{

StrBlobPtr ret = \*this;

++\*this;

return ret;

}

手动调用必须要传递一个值

StrBlobPtr p(a1); // p points to the vector inside a1

p.operator++(0); // call postfix operator++

p.operator++(); // call prefix operator++

成员访问运算符：operator\*是为了让指针表象的像值一样，operator->()是为了让对象表现的像指针一样。

class StrBlobPtr

{

public:

std::string& operator\*() const

{

return (\*p)[curr];

}

std::string\* operator->() const

{ // delegate the real work to the dereference operator

return & this->operator\*();

}

};

函数调用运算符：

必须为成员函数，且可以定义多个参数不同的版本的调用运算符

定义了调用运算符的对象成为函数对象

编写一个lambda后， 编译器会把表达式转换成一个未命名的对象，含有一个函数调用运算符

可调用对象与function

int add(int i, int j) { return i + j; }

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

cout << f1(4,2) << endl; // prints 6

类类型：

## 第十五章 面向对象程序设计

希望派生类定义自己合适的版本，则用虚函数，函数前添加virtual

使用基类的指针或者引用调用一个虚函数时会产生动态绑定

类可以添加final禁止其他类继承

虚函数：

派生类覆盖某个虚函数的时候，可以再次使用virtual说明性质，也可以不同，因为一个函数声明为虚函数，所有的派生类中都是虚函数

基类与派生类的虚函数形参必须严格匹配

如果不匹配，编译器会认为两个虚函数是相互独立的

可以使用override显示的覆盖虚函数，如果标记了但是没有覆盖到虚函数，则编译器会报错

函数也可以使用final来禁止覆盖操作

可以使用作用域运算符：：来强制执行某个版本的虚函数，不进行动态绑定，一般只有成员函数或者友元才需要这么做

抽象基类：

含有纯虚函数的类是抽象基类，在函数声明的分号前加=0可以声明为纯虚函数

double net\_price(std::size\_t) const = 0;

抽象基类负责定义接口，不能创建基类的对象

派生访问说明符：

派生访问说明符的作用是控制派生类（包括派生类的派生类）用户对于基类成员的访问权限。

如果使用公有继承，则基类的公有成员和受保护成员在派生类中属性不发生改变。

如果使用受保护继承，则基类的公有成员和受保护成员在派生类中变为受保护成员。

如果使用私有继承，则基类的公有成员和受保护成员在派生类中变为私有成员。

派生类到基类转换的可访问性（假定D继承自B）：

只有当D公有地继承B时，用户代码才能使用派生类到基类的转换。

不论D以什么方式继承B，D的成员函数和友元都能使用派生类到基类的转换。

如果D继承B的方式是公有的或者受保护的，则D的派生类的成员函数和友元可以使用D到B的类型转换；反之，如果D继承B的方式是私有的，则不能使用。