类的基本思想是数据抽象和封装（实现接口和实现的分离）。数据抽象是一种依赖于接口（包括用户所能执行的操作）和实现（包括类的数据成员、负责接口实现的函数体以及定义类所需的各种私有函数）分离的编程技术。类的用户是程序员，而非应用程序的最终使用者。

# 1.定义抽象数据类型：

不可以直接访问数据成员的是抽象数据类型。成员函数的声明必须在类的内部，定义则既可以在类的内部也可以在类的外部。定义在类内的函数是隐式的inline函数。

成员函数通过名为this的隐式参数来访问调用它里面的对象，用该函数所在的对象地址初始化this，可以直接调用类内函数的对象成员而无需通过成员访问运算符。任何对类成员的直接访问都被看作是this的隐式引用，编译器把对象的地址传递给this，this是一个常量指针，不允许改变，如：

string isbn() const{return bookNo;}

调用类内函数时等价于：

isbn(&total);

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

函数形参列表后紧随的const的作用是修改隐式this指针的类型。默认情况下，this是指向类类型非常量版本的常量指针，是T \*const（底层const）类型，所以我们不能把this绑定到一个常量对象上，这会使得我们不能在一个常量对象上调用成员函数，所以把this设置为指向常量的常量指针有助于提高函数的灵活性。像这样使用const成员函数被称作常量成员函数。常量对象、常量对象的引用、指针只能调用常量成员函数。

成员函数可以随意使用类中的其他成员而无需在意成员出现的次序，因为编译器分两步处理类：编译成员的声明、编译成员函数体。

类外部定义的成员的名字必须包含它所属的类名，使用作用域运算符（：：）。

返回this对象：return \*this;//解引用

## 1.1定义类相关的非成员函数：

如果函数在概念上属于类但是不定义在类中，则它一般应与类声明放在同一头文件中。

在函数中接受IO类型的参数要接受其引用，因为IO类属于不能被拷贝的类型，而且因为读取和写入的操作会改变流的内容，所以传递非const变量。

## 1.2构造函数：

构造函数的名字和类名相同，构造函数没有返回类型。构造函数不能被声明成const，当我们创建一个const对象时，知道构建函数完成初始化过程，对象才能真正取得其常量属性。

默认构造函数无需任何实参。默认构造函数按以下规则初始化类的数据成员：①用类内的初始值初始化成员，②默认初始化成员

某些类不能依赖于默认构造函数。原因有三：①一旦我们定义了一些其他的构造函数，那么除非我们再定义一个默认的构造函数，否则将没有默认函数 ②对于某些类，默认构造函数可能执行错误的操作，创建类对象是可能得到未定义的值 ③有的时候编译器不能为某些类合成默认的构造函数。比如类中包含一个其他类类型的成员且这个成员的类型没有默认构造函数时，就无法初始化。

Sale\_data() = default;

该构造函数不接受任何实参，所以是默认构造函数。如果我们需要默认的行为，那么可以通过在参数列表后面写上= default来要求编译器生成构造函数。如果=default在类的内部，则默认构造函数是内联的，反之不是内联的。如果我们提供了一个构造函数，编译器就不会自动生成默认的构造函数，如果我们需要默认构造函数，必须显式地将它声明出来。

Sale\_data(const string &s，int n):bookNo(s),num(n){}

该构造函数冒号及后面的部分是构造函数初始值列表,意为括号里的值传给前面的变量。当某个数据成员被构造函数初始值列表忽略时，其他的参数将以与默认构造函数相同的方式隐式初始化。

## 1.3拷贝、赋值和析构：

虽然系统可以替我们合成拷贝、赋值和析构的操作，但当类需要分配类对象之外的资源时，合成的版本常常会失效，比如说管理动态内存的类通常不能依赖于上述操作的合成版本。不过很多需要动态内存的类能使用vector对象或者string对象管理必要的存储空间，使用vector和string类能避免分配和释放内存带来的复杂性。

# 2.访问控制与封装：

我们使用访问控制符加强类的封装性。定义在public说明符之后的成员在整个程序内可被访问，定义在private说明符之后的成员只可以被类成员访问。

对于某个访问说明符能出现多少次没有严格的规定，每个访问说明符指定了接下来成员的访问级别，其有效范围知道下一个访问说明符或者到达类的结尾。

struct关键字和class关键字的默认访问权限不一样，如果我们使用struct关键字，则定义在第一个访问说明符之前的成员是public；如果我们使用class关键字，则这些成员是private的。

## 友元：

private的数据成员不能被相关类外函数调用，类外函数会失效，但是友元会让类允许其他类或者函数访问它的非公有成员。如果想把一个函数作为它的友元，只需增加一条以friend关键字开始的函数声明语句，友元不是类的成员也不受它所在区域访问控制级别的约束，在类内出现的位置不限，最好在类开始或结束的位置集中声明友元。 友元声明仅仅指定了访问的权限，而非一个函数声明。

# 3.类的其他特性：

## 3.1令成员作为内联函数：

定义在类内部的成员函数是自动inline的，我们可以在类内把inline作为声明的一部分显式地声明成员函数，同时在类的外部用inline修饰函数定义，最好在类外部定义函数的地方说明inline，只在定义的地方说明inline即可，声明的时候不管。虽然我们无需再声明和定义的地方同时说明inline，但是这么做其实是合法的。

inline Screen &Screen::move(pos r,pos c)//在类外指定inline

{

pos row = r \* width;

cursor = row + c;

return \*this;

}

## 3.2可变数据成员：

一个可变数据成员永远不会是const，即使它是const对象的成员。因此，一个const成员函数（说明其不会修改数据成员)任何不会修改数据成员的函数都应该声明为const 类型。如果在编写const 成员函数时，不慎修改了数据成员，或者调用了其它非const 成员函数，编译器将指出错误）可以改变一个可变成员的值。在变量的声明最前面加上mutable关键字即可。

## 3.3类数据成员的初始值：

类内初始值必须使用=的初始化形式或者花括号括起来的直接初始化形式。

vector<Screen> screens{Screen(24,80,’ ’)};

## 3.4返回\*this的成员函数：

一个const成员函数如果以引用的形式返回\*this，那么它的返回类型将是常量引用。

myScreen.move(4,0).set();//把光标移动到一个指定的位置，然后设置该位置的字符值，move返回了\*this，所以可以再次调用成员函数。要注意如果move返回常量对象，set将无法调用，不能通过编译。

## 3.5类类型：

即使两个类的成员列表完全一致，它们也是不同的类型，对于一个类来说，它的成员函数和其他任何类的成员都不是一回事。

声明类变量的时候，可以这么声明：class Sale\_data item;

我们也能仅仅声明类而暂时不定义它：class Screen；这种声明有时被称作前向声明，它向程序中引入了名字Screen并指明Screen是一种类类型，它在声明之后定义之前是一个不完全类型，此时我们已知Screen是一个类类型，但是不清楚它到底包含哪些成员。只能用于：定义指向这种类型的指针或引用，声明以不完全类型作为参数或者返回类型的函数。但也有特殊情况：直到类被定义后数据成员才能被声明称这种类类型，一旦一个类的名字出现后，他就被认为是声明过了，比如说在要声明的类中又有了该类的成员。

## 3.6友元再探：

3.6.1类之间的友元：

friend class Screen;//把类Screen作为友元，可以访问其私有部分

友元关系不存在传递性

3.6.2成员函数作为友元：

类A只为类B中的一个成员函数提供访问权限，而不是令整个类作为友元。声明的时候要指出该成员函数属于哪个类。

friend void Window\_mgr::clear();

对于重载函数，我们要挨个声明同名的重载函数。

如果友元函数在类内被声明，必须要在类的外部提供相应的生命从而使得函数可见，友元本身不一定生命在类中。友元声明的作用是影响访问权限，并非是普通意义上的声明。

struct X

{

friend void f(){}//友元函数可以定义在类的内部

X(){f();}//错误，f未被声明

void g();

void h();

};

void X::g(){return f();}//错误，f未被声明

void f();

void X::h(){return f();}

# 4.类的作用域：

在类的作用域外，普通的数据和函数成员只能由对象、引用、指针使用成员运算符来访问。

函数的返回类型通常出现在函数名之前。因此当成员函数定义在类的外部时，返回类型中使用的名字都位于类的作用域之外，此时返回类型必须指明它是哪个类的成员，如：

class Window\_mgr

{

public:

ScreenIndex addScreen(const Screen&);

};

//因为返回类型出现在类名之前，所以事实上它是位于Window\_mgr类的作用域之外的，要想使用ScreenIndex作为返回类型，需要明确指定哪个类定义了它。

Window\_mgr::ScreenIndex Window\_mgr::addScreen(const Screen &s)

{

screens.push\_back(s);

return screens.size() – 1;

}

## 4.1名字查找：

名字查找的过程：①在名字所在的块中寻找其声明语句（只考虑在名字的使用之前出现的声明），②如果没找到，查找外层作用域，③没找到，程序报错。

但是在类内部的成员函数来说，名字查找的方式与上述有区别，不过不明显。

类的定义分两步：①编译成员的声明，②类全部可见后编译函数体。

4.1.1用于类成员声明的名字查找：

这种两阶段的处理方式只是用于成员函数中使用的名字，声明中使用的名字（包括返回类型或者参数列表中使用的名字）都必须在使用前确保可见。

一般来说，内层作用域可以重新定义外层作用域中的名字，即使该名字已经在内层作用域中使用过。然而在类中，如果成员使用了外层作用域中某个名字，而该名字代表一种类型，则类不能在之后重新定义该名字。尽管重新定义类型名字是一种错误行为，但是编译器不报错。

typedef double Money;

string bal;

class Account

{

public:

Money balance(){return bal;}

private:

typedef double Money;//错误，不能重新定义

Money bal;

};

当编译器看淡balance函数声明时，它将在Account类的范围去寻找Money，没找到，所以去Account的外层作用域中查找。

balance函数在整个类可见后才被处理，所以return的bal是类中的bal而不是string类型的bal。

4.1.2成员定义中的普通块作用域的名字查找：

成员函数中使用的名字按照如下方式解析：

①在名字所在的块中寻找其声明语句（只考虑在名字的使用之前出现的声明），②如果没找到，在类内继续查找，这时类的所有成员都可以被考虑，③如果没找到，在成员函数定义之前的作用域内继续查找。

不建议使用类内成员的名字作为某个成员函数的参数,如下：

int height;

class Screen

{

public:

typedef string::size\_type pos;

void dummy\_fun(pos height){cursor = width \* height;}

private:

pos cursor = 0;

pos height = 0,width = 0;

};

这段程序dummy\_fun中的height有歧义，有两个意思：①成员height：等价于this->height， Screen::height ②传入的参数height

这段程序中外围的height被掩盖了，如果我们需要外层作用域中的名字，可以显式地通过作用域运算符来请求：cursor = width \* ::height;

# 5.构造函数再探：

## 5.1构造函数的初始值列表：

有时我们可以忽略数据成员用列表初始化和赋值之间的差异，但如果成员是const或者是引用的话，必须将其列表初始化，当成员属于某种类类型且该类没有定义默认构造函数时，也必须将这个成员列表初始化。在很多类中，列表初始化和赋值的区别事关底层效率问题：前者直接初始化数据成员，后者则先初始化再赋值。

列表初始化中成员的初始化顺序与它们在类定义中的出现顺序一致，构造函数初始值列表中初始值的前后位置关系不会影响实际的初始化顺序。最好令构造函数初始值的顺序与成员声明的顺序保持一致，而且如果可能的话，尽量避免使用某些成员初始化其他成员。

如果一个构造函数为所有参数都提供了默认实参，则它实际上也定义了默认构造函数：

Sale\_data(string s = “”):book(s){}//定义默认构造函数，使其与只接受一个string实参的构造函数功能相同

## 5.2委托构造函数：

委托构造函数使用它所属类的其他构造函数执行它自己的初始化过程，它把自己的一些指责委托给了其他构造函数，其参数列表必须与所委托的构造函数匹配，如：

Sales\_data():Sales\_data(“”,0,0){}

Sales\_data(string &s):Sales\_data(s,0,0){}

## 5.3默认构造函数的作用：

当对象被默认初始化或值初始化时自动执行默认构造函数。默认初始化在以下情况下发生：①当我们在块作用域内部使用任何初始值定义一个非静态变量，②当一个类本身含有类类型的成员且使用合成的默认构造函数时，③当类类型的成员没有在构造函数初始值列表中显式地初始化时。值初始化在以下情况下发生：①在数组初始化的过程中如果我们提供的初始值数量少于数组的大小时，②当我们部使用初始值定义一个局部静态变量时，③当我们通过书写形如T()的表达式显式地请求值初始化时（其中T是类型名，vector的一个构造函数只接受一个实参用于说明vector的大小）。

如果定义了其他构造函数，那么也要提供一个默认构造函数。

struct B

{

B(){}

};

注意：用默认构造函数声明对象：

Sales\_data obj();//错误，obj是一个函数而非对象

Sales\_data obj;//正确

## 5.4隐式的类类转换：

构造函数接受参数，它也定义了转换为此类类型的隐式转换机制，有时我们把这种构造函数称作转换构造函数。

只允许自动地执行一步类型转换：

item.combine(“9-999-99999-9”);//错误，包含两步转换：①转换成string，②转换成Sales\_data

item.combine(string(“9-999-99999-9”));//正确

item.combine(Sales\_data(“9-999-99999-9”));//正确

item.combine(cin);//这个转换执行了接受一个istream的Sales\_data构造函数，创建了一个临时的Sale\_data对象，随后传递给combine。

在如上要求隐式转换的程序上下文中，我们可以通过将构造函数声明为explocot加以阻止，explicit只对一个实参的构造函数有效，需要多个实参的构造函数不能用于执行隐式转换，所以无需定义为explicit的。只能在类内声明构造函数时使用explicit关键字，在类外部定义构造函数时不应重复使用。

发生隐式转换的一种情况是当我们用“=”执行拷贝形式的初始化时。此时我们只能使用直接初始化而不能使用explicit构造函数，explicit关键字声明构造函数时，它将只能以直接初始化的形式使用。

Sales\_data item1(null\_book);//正确，直接形式的初始化

Sales\_data item2 = null\_book;//错误，不能将explicit构造函数用于拷贝形式的初始化。

explicit构造函数不会用于隐式转换过程，但是可以使用这样的构造函数显式地强制进行转换。

item.combine(Sales\_data(null\_book)); //正确

item.combine(static\_cast<Sale\_data>(cin));//正确

标准库中含有显式构造函数的类：①接受一个单参数的const char\*的string构造函数不是explicit的，②接受一个容量参数vector构造函数是explicit的。

## 5.5聚合类：

聚合类：①所有成员都是public的，②没有定义任何构造函数，③没有类内初始值，④没有基类，也没有virtual函数。

我们可以提供一个花括号括起来的成员初始值列表，并用它初始化聚合类的数据成员，初始值的顺序必须与声明的顺序一致。

Data vall = {0,”Anna”}；//注意顺序

与初始化数组元素的规则一样，如果初始值列表中的元素个数少于类的成员数量，则靠后的成员被值初始化，初始值列表的元素个数绝对不能超过类的成员数量。

## 5.6字面值常量类：

字面值类型的类可能含有constexpr函数成员，这样的成员必须符合constexpr函数的所有要求，它们是隐式const的。

数据成员都是字面值类型的聚合类是字面值常量类，否则，如果符合下述要求，则它也是一个字面值常量类。

①数据成员都必须是字面值类型，②类必须至少含有一个constexpr构造函数，③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条常量表达式；或者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，④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。

尽管构造函数不能是const的，但字面值常量类的构造函数可以是constexpr函数，事实上，一个字面值常量类必须至少提供一个constexpr构造函数。constexpr构造函数可以声明成=default形式或删除函数的形式，否则，constexpr构造函数就必须既符合构造函数的要求（不能包含返回语句），又符合constexpr函数的要求（它能拥有的唯一可执行语句就是返回语句），所以constexpr构造函数体一般来说应该是空的。

constexpr构造函数必须初始化所有数据成员，初始值或者使用constexpr构造函数，或者是一条常量表达式。

# 6.类的静态成员：

## 6.1声明：

在成员的声明之前加上static使得其与类类型相关联在一起，类的静态成员存在于任何对象之外。静态数据成员的类型可以是常量、引用、指针、类类型等。静态成员函数不包含\*this指针，返回值不能声明成const。

## 6.2使用：

①使用作用域运算符直接访问静态成员，②使用类对象、指针、引用访问静态成员。

## 6.3定义：

在类的外部定义静态成员时，不能重复static关键字。

因为静态数据成员不属于类的任何一个对象，所以它们并不是在创建类的对象时被定义的，这意味着它们不是由类的构造函数初始化的。而且不能在类的内部初始化静态成员，一个静态数据成员只能被定义一次。类似于全局变量。

double Account::interestRate = initRate();//需要指定对象的类型名、类名、作用域运算符以及成员自己的名字

## 6.4类内初始化：

我们可以为静态成员提供const整数类型的类内初始值，不过要求静态成员类型（等号左侧）必须是字面值常量类型的constexpr。初始值（等号右侧）必须是常量表达式。

即使一个常量静态数据成员在类内部被初始化了，通常情况下也应该在类的外部定义一下该成员，但是外部就不能再指定初始值了。当该成员只在类内使用的时候，不用在类外定义。

静态数据成员的类型可以是不完全类型（有声明、没定义的类型），静态数据成员的类型可以是它所属的类类型，非静态数据成员则只能声明它所属类的指针或引用。

我们可以使用静态成员作为默认实参，非静态数据成员不能作为默认实参，因为它的值本身属于对象的一部分，这么做的结果是无法提供一个对象以便从中获取成员的值。