# 预备知识

Windows编译器

**Compile通常意味着对当前打开的文件中的代码进行编译。**

**Build和Make通常指编译项目中只有被修改的项目。**

**Build All重新编译所有的源代码文件。**

**Run或Execute 若没有执行前面的步骤，Run将在运行程序之前完成这些步骤。**

**Debug以步进方式执行程序。**

语言错误

如果违反了语言规则，编译器将生成错误消息进行提示。有时，真正的问题可能在标识行之前；有时，一个错误可能引发一连串的错误消息。

改正错误时，应首先改正第一个错误。若那一行找不到错误，请查看前一行。

程序能够通过某个编译器的编译并不意味着它是合法的C++程序，程序不能通过的某个编译器的编译也并不意味着它是非法的C++程序。

编译器在不完全地构建程序后将出现混乱，它将显示无法改正的、无意义的错误消息。

# 开始学习C++

函数与语句

函数体是指出函数应做什么的计算机指令。每条完整的指令都称为语句。所有的语句都以分号结束。

语句终止符是分号，是语句的组成部分，而不是语句之间的标记。

位于函数名前面的部分叫函数返回类型，描述从函数返回给调用它的函数的消息。

函数名后括号中的部分叫做形参列表。

头文件名

像iostream这样的文件叫做包含文件

名称空间

Using namespace std;

这叫做using编译指令。

这个using编译指令使得std名称空间中的所有名称都可用。更好的做法是，只使所需的名称可用，这可以通过使用using声明来实现：

using std::cout; //make cout available。

using std::endl; //make endl available。

cout<<”come up and C++ me some time.”;

<<符号表示该语句将把这个字符串发给cout;该符号指出了信息流动的路径。

程序中的声明语句叫做定义声明语句。它将导致编译器为变量分配内存空间。在较为复杂的情况下，还可能有引用声明。

对于声明变量，C++的做法是尽可能在首次使用变量前声明它。

类的初级了解

类是用户定义的一种数据类型。

类定义描述的是数据格式及其用法，而对象是根据数据格式规范创建的实体。

类描述了一种数据类型的全部属性，对象是根据这些描述创建的实体。

函数

C++函数分两种：有返回值的和没有返回值的。

有返回值的函数将生成一个值，而这个值可赋给变量或在其他表达式中使用。

X = sqrt(6.25);

圆括号中的值(6.25)是发给函数的信息。以这种方式发给函数的值叫做参数。

发送回去的值叫做函数的返回值。

在使用函数之前，C++编译器必须知道函数的参数类型和返回值类型。

C++程序应当为程序中使用的每个函数提供原型。

Sqrt()的函数原型像这样：

Double sqrt(double);

原型结尾的分号表明它是一条语句，这使得它是一个原型，而不是函数头。如果省略分号，编译器要求接着提供定义该函数的函数体。

函数原型只描述函数接口。函数定义包含了函数的代码。

C++库文件中包含了函数的编译代码，而头文件则包含了原型。

函数调用中必须包括括号，即使没有参数。

# 处理数据

成员函数初级了解

成员函数归类所有，只能通过特定类的特定对象来使用成员函数。要通过对象使用成员函数，必须用句点将对象名和函数名称连接起来。

句点被称为成员运算符。

# 复合类型

字符与数组

数组列表初始化，可用于所有类型。

列表初始化禁止缩窄转换。

字符串常量（使用双引号）不能与字符常量（使用单引号）互换。

字符常量是字符串编码的简写表示。

C++允许拼接字符串字面值。

C++程序常使用指针来处理字符串。

String类简介

因为添加string类扩展了C++库，所以现在可以定义string类型的变量（使用C++的话说是对象），//定义一个数据类型为类的变量被C++称为对象。

其他形式的字符串字面值

除char类型外，C++还有类型wchar\_t;而C++11新增了类型char16\_t和char32\_t。

对于这些类型的字符串字面值，C++分别使用前缀L、u、和U表示。如：

wchar\_t title[] = L”hief Astrogator";

char16\_t name[] = u"Felonia Ripova";

char32\_t car[] = U"Humber Super Snipe";

C++11新增的另一种类型是原始（raw）字符串。在原始字符串中，字符表示的就是自己。

原始字符串将”(和)”用作定界符，并使用前缀R来标识原始字符串。如：

cout << R"(Jim "King" Tutt uses "\n" instead of endl.)" << endl;

输出内容为：Jim "King" Tutt uses "\n" instead of endl.

输入原始字符串时，按回车键不仅会移动到下一行，还将在原始字符串中添加回车字符。

原始字符串语法允许您在表示字符串开头的”和(之间添加其他字符。

使用R”+\*(标识原始字符串的开头时，必须使用)+\*”标识原始字符串的结尾。

结构简介

结构是用户定义的类型，结构声明定义了这种类型的数据属性。

结构定义指出了新类型的特征。

用结构定义的数据类型的变量可以使用成员运算符(.)来访问各个成员。

可以使用赋值运算符（=）将结构赋给另一个同类型的结构，这种赋值被称为成员赋值。

如果定义一个结构数据类型的变量的数组变量。

数组变量本身是一个数组，而不是结构。

C++也允许指定占用特定位数的结构成员。

字段的类型应为整型或枚举，接下来是冒号，冒号后面是一个数字，它指定了使用的位数。

如：

unsigned int sn: 4;

bool goodInL: 1;

共用体

共用体是一种数据格式，它能够存储不同的数据类型，但只能同时存储其中的一种类型。

共用体的长度为其最大成员的长度。

枚举

枚举量是整型，可被提升为int类型，但int类型不能自动转换为枚举类型。

枚举没有定义运算符+，但用于算术表达式中时，枚举将被转换为整数，

如果int值是有效的，可以通过强制类型转换，将它赋给枚举变量。

**枚举更常被用来定义相关的符号变量**。

可以使用赋值运算符来显示地设置枚举量的值，指定的值必须是整数。也可以只显式地定义其中一些枚举量的值。

没有被初始化的枚举量的值将比其前面的枚举量大1.

可以创建多个值相同的枚举量。

指针和自由存储空间

指针是一个变量，其存储的是**值的地址。**

使用常规变量时，值时指定的量，而地址为派生量，

处理存储数据刚好相反，将地址视为指定的量，而将值视为派生量。一种特殊类型的变量--指针用于存储**值的地址**。指针名表示的是地址。\*运算符将其用于指针，可以得到该地址处存储的值。

在C++中，int\*是一种复合类型，是指向int的指针。

指针都是基于其他类型的。

地址的长度或值既不能指示关于变量的长度或类型的任何信息，也不能指示该地址上有什么数据。

C++中创建指针时，计算机将分配用来存储地址的内存，但不会分配用来存储指针所指向的数据的内存。为数据局提供空间是一个独立的步骤。

一定要在对指针应用解除引用运算符（\*）之前，将指针初始化为一个确定的、适当的地址。

变量是在编译时分配的有名称的内存。

new分配的内存块通常与常规变量声明分配的内存块不同。Int变量和int\*变量的值都存在栈中，而new存为堆中。

不要尝试释放已经释放的内存块，

不能使用delete来释放声明变量所获得的内存。

只能用delete来释放使用new分配的内存。然而，空指针也能使用delete。

使用delete的关键在于，将它用于new分配的内存。这并不意味着要使用用于new的指针，而是用于new的地址。

在编译时给数组分配内存被称为静态联编。

数组在运行时创建的被称为动态联编。

New运算符返回数组第一个元素的地址，该地址被赋给指针变量名。

不要使用delete来释放不是new分配的内存。

不要使用delete释放同一个内存块两次。

如果使用new[ ]为数组分配内存，应使用delete[ ]来释放。

如果使用new为一个实体分配内存，应使用delete来释放。

对空指针应用delete是安全的。

程序跟踪了分配的内存量，以便以后使用delete[ ]运算符时能够正确地释放这些内存。

但这种信息不是公开的。

指针、数组和指针算术

将指针变量+1后，增加的量等于它指向的类型的字节数。

C++将数组名解释为地址

数组名与指针名区别之一是，可以修改指针的值，而**数组名是常量。**

对数组应用sizeof运算符得到的是数组的长度，而对指针应用sizeof得到是指针的长度。

数组名被解释为其第一个元素的地址，而对数组名应用地址运算符时，得到的是整个数组的地址。

cout对象认为char的地址是字符串的地址，如果给cout提供一个字符的地址，则它将从该字符开始打印，知道遇到空字符为止。

字符串字面值是常量。

字符串字面值视为只读常量，如果试图修改它们，将导致运行阶段错误。在C++中字符串字面值都被视为常量。

有些编译器只使用字符串字面值的一个副本来表示程序中所有的该字面值。

创建动态结构时，不能将成员运算符句点用于结构名，因为这种结构没有名称，只是知道它的地址。C++为这种情况提供了**箭头成员运算符(->)**。

如果ps是指向结构的指针，则\*ps就是被指向的值——结构本身。由于\*ps是一个结构，因此(\*ps).price是该结构的price成员。

C++有3种管理数据内存的方式：**自动存储、静态存储和动态存储**。

在函数内部定义的常规变量使用自动存储空间，被称为自动变量，在所属的函数被调用时自动产生，在该函数结束时死亡。

自动变量是一个局部变量，其作用域为包含它的代码块。

自动变量通常存储在栈中。

静态存储是整个程序执行期间都存在的存储方式。一种是在函数外面定义它；另一个是在声明变量时使用关键字static。

变量可能存在于程序的整个生命周期，也可能只是在特定函数被执行存在。

数组的替代品

模板类vector

vector<typeName> vt(n\_elem);

模板类array

array<typeName,,n\_elem> arr;

array中n\_elem不能是变量。

循环和关系表达式

For循环的2种形式

For(表达式1;表达式2;表达式3){块语言}

For(数据类型 数据名: 数据类型数组或容器名){块语言}

# 分支语句和逻辑运算符

If语句

If(测试条件)

语句

如果测试条件为true，则程序将执行语句，后者可以是一条语句，也可以是语句块。

？:运算符

该运算符的通用格式如下

表达式1？表达式2:表达式3

如果表达式1为true，则整个条件表达式的值为表达式2的值;否则整个表达式的值为表达式3的值。

# 函数——C++的编程模块

复习函数知识

函数必须元素：

**提供函数定义；提供函数原型；调用函数。**

**返回值本身可以使常量、变量，也可以是表达式，返回值的类型必须为typeName类型或可以被转换为typeName。**

**返回值不能是数组**

**可以将数组作为结构或对象组成部分来返回。**

**函数通过将**返回值复制到指定的CPU寄存器或内存单元中来将其返回。

**返回函数和调用函数必须就该内存单元中存储的数据的类型达成一致。**

**原型描述了函数到编译器的接口，**它将函数返回值的类型以及参数的类型和数量告诉编译器。

函数原型是一条语句。

函数和数组

**C++将数组名视为指针。**

**当且仅当用于函数头或函数原型中，数组与指针的含义才是相同的。**

**两个恒等式**

**arr[i]==\*(ar+i);**

**&arr[i]==ar+I;**

**转递常规变量时，函数将使用该变量的拷贝；但转递数组时，函数将使用原来的数组。**

**指针本身并没有指出数组的长度。**

**为将数组类型和元素数量告诉数组处理函数，请通过两个不同参数来传递它们；**

**Void fillarray(int arr[],int size); 允许**

**而不要试图使用方括号表示法来传递数组长度；**

**Void fillArray(int arr[size]); 错误**

C++是按值传递数据。

C++禁止将const的地址赋给非const指针。但可以使用类型转换来突破这种限制。

因此仅当只有一层间接关系时，才可以将非const地址或指针赋给const指针。

如果数据类型本身并不是指针，则可以将const数据或非const数据的地址赋给指向const的指针，但只能将非const数据的地址赋给非const指针。

Const可以避免由于无意间修改数据而导致的编程错误；

使用const使得函数能够处理const和非const实参，否则将只能接受非const数据。

如果条件允许，则应将指针形参声明为指向const指针。

结构将其数据组合成单个实体或数据对象，该实体被视为一个整体。

如果程序在输入循环后还需要进行输入，则必须使用cin.clear()重置输入，然后，还可能需要通过读取不合法的输入法来丢弃他们。

函数和string array对象

arrray并非只能存储基本数据类型，它还可存储类对象

递归导致函数调用数(以及存储的变量数)翻倍，因此如果要求的递归层次很多，这种递归方式将是一种糟糕的选择；然而，如果递归层次较少，这将是一种精致而简单的选择。

函数指针

**函数也有地址。函数的地址是存储其机器语言代码的内存的开始地址。**

**获取函数的地址很简单：只要使用函数名即可。**

**声明指向某种数据的指针时，必须指定指针指向的类型。**

**自动类型推断只能用于单值初始化，而不能用于初始化列表。**

函数探幽

C++内联函数函数

执行到函数调用指令时，程序将在函数调用后立即存储该指令的内存地址，并将函数参数复制到堆栈（为此保留的内存块），跳到标记函数起点的内存单元，执行函数代码（也许还需将返回值放入寄存器中），然后跳回到地址被保存的指令处。来回跳跃并记录跳跃位置意味着以前使用函数时，需要一定的开销。

内联代码，程序无需跳到另一个位置处执行代码，再跳回来。但代价是需要占用更多内存。

使用这项特性必须采取下述措施之一

在函数声明前加上关键字inline

在函数定义前加上关键字inline

如果使用c语言的宏执行了类似函数的功能，应考虑将它们转换为C++内联函数。

引用变量

**引用是已定义的变量的别名。**

**但引用变量的主要用途是用作函数的形参。通过将引用变量用作参数，函数将使用**原始数据

C++给&符号赋予了另一个含义，将其用来声明引用。

Int rats;

Int& rodents=rats;

它们指向相同的值和内存单元，int&指的是指向int的引用。

引用经常被用作函数参数，这种传递参数的方法被称为按引用传递。

按引用传递允许被调用的函数能够访问调用的函数中的变量。

如果试实参与引用参数不匹配，C++将生成临时变量。当前，仅当参数为const引用时，C++才允许这样做。

两种情况下生成临时变量：

实参的类型正确，但不是左值；

实参的类型不正确，但可以传唤为正确的类型。

**左值参数是可以被引用的数据对象，如，变量、数组元素、结构成员、引用和解除引用的指针都是左值。**

**非左值包括字面常量和包含多项的表达式。**

左值最初指的是可出现在赋值语句左边的实体，现在常规变量和const变量都可以视为左值，因为可通过地址访问它们。

数组元素的行为与同类型的变量类似。

Double refube(const double& ra)

{

return ra\*ra\*ra;

}

Double side=3.0;

Double\* pd = &side;

Double& rd=side;

Long edge = 5L;

Double lens[4]={2.0,5.0,10.0,12.0};

Double c1 = refcube(side);

Double c2 = refcube(lens[2]);

Double c3 = refcube(rd);

Double c4 = refcube(\*pd);

Double c5 = refcube(edge);

Double c6 = refcube(7.0);

Double c7 = refcube(side+10.0);

Edge虽然是变量，类型却不正确，double引用不能指向long。另一方面，参数7.0和side+10.0的类型都正确，但没有名称，**在这些情况下，编译器都将生成一个临时匿名变量，并让ra指向它。**这些临时变量只在函数调用期间存在，此后编译器便可以随意将其删除。

如果接受引用参数的函数的意图是修改作为参数传递的变量，则创建临时变量将阻止这种意图的实现。

如果函数调用的参数不是左值或与相应的const引用参数的类型不匹配，则C++将创建类型正确的匿名变量，将函数调用的参数的值传递给该匿名变量，并让参数来引用该变量。

将引用参数声明为常量数据的引用的理由

使用const可以避免无意中修改数据的编程错误；

使用const使函数能够处理const和非const实参，否则将只能接受非const数据；

使用const引用使函数能够正确生成并使用临时变量。

右值引用。这种引用可指向右值，是使用&&声明的；

返回引用的函数实际上是被引用的变量的别名。

应避免返回函数终止时不再存在的内存单元引用。（这个灰常重要！）（这个灰常重要！）

如下：

Const free\_throws& clone2(free\_throws& ft)

{

free\_throws newguy;

newguy = ft;

return newguy;

}

函数返回一个指向临时变量的引用，函数运行完毕后它将不再存在。

在赋值语句中，左边必须是可修改的左值。左边的子表达式必须标识一个可修改的内存块。

常规（非引用）返回类型是右值——不能通过地址访问的值。这种表达式可出现在赋值语句的右边，但不能出现在左边。函数返回值位于临时内存单元中，也是右值。在运行到下一个语句时，它们可能不再存在。（易出现引用错误）

**应避免在设计中添加模糊的特性，因为模糊特性增加了犯错的机会。**

**String version1(const string& s1,const string& s2)**

**{**

**String temp;**

**Temp=s2+s1+s2;**

**Return temp;**

**}**

**Temp 是一个新的string对象，只在函数version1()中有效，该函数执行完后，它将不再存在。因此，**返回指向temp的引用不可行**，因此该函数的返回类型为string，**这意味着temp的内容将被复制到一个临时存储单元中，然后在main()中，该存储单元的内容被复制到一个名为result的string中。

**Const string& version3(string& s1,const string& s2)**

**{**

**String temp;**

**Temp=s2+s1+s2;**

**Return temp;**

**}**

此段代码程序试图引用已经被释放的内存。

**将特性从一个类传递给另一个类的语言特性被称为继承。**

**派生类继承了基类的方法，这意味派生类对象可以使用基类的特性。**

**基类引用可以指向派生类对象，无需进行强制类型转换。**

**使用引用参数的主要原因：**

**程序员能够修改调用函数中的数据对象。**

**通过传递引用而不是整个数据对象，可以提高程序的运行速度。**

使用传递的值而不作修改的函数：

如果数据对象很小，如内置数据类型或小型结构，按值传递。

如果数据对象是数组，则使用指针，因为这是唯一的选择，并将指针声明为指向const的指针。

如果数据对象是较大的结构，则使用const指针或const引用，以提高程序的效率。这样可以节省复制结构所需的时间和空间。

如果数据对象是类对象，则使用const引用。类设计的语义常常要求使用引用，因此传递类对象参数的标准方式是按引用传递。

如果数据对象是内置数据类型，则使用指针。如果看到诸如fixit(&x)这样的代码（其中x是int），则很明面，该函数将修改x。

如果数据对象是数据，则只能使用指针。

如果数据对象是结构，则使用引用或指针。

如果数据对象是类对象，则使用引用。

默认参数

默认参数指的是当函数调用中省略了实参是自动使用的一个值。

方法是将值赋给原型中的参数，

如：

Char\* left(const car\* str, int n=11);

上面的原型将n初始化为1.如果省略参数n，则它的值将为1；否则，传递的值将覆盖1.

**对于带参数列表的函数，必须从右向左添加默认值**。

**实参按从左到右的顺序依次被赋给相应的形参，而不能跳过任何参数**。

函数重载

函数多态能够使用多个同名的函数。

**函数多态允许函数可以有多种形式**。

**函数重载的关键是函数的参数列表——也称为函数特征标**。

如果两个函数的**参数数目和类型相同，同时参数的排列顺序也相同，则它们的特征标相同**。**没有匹配的原型并不会自动停止使用其中某个函数**，C++将尝试**使用标准类型强制进行匹配**。

编译器检查函数特征标时，**将把类型引用和类型本身视为同一个特征标**。

匹配函数，**并不区分const和非const变量**。

函数基本上**执行相同的任务，但使用不同形式的数据时，才采用函数重载**。

函数模板

函数模板是通用的函数描述，它们使用泛型来定义函数，其中泛型可用具体的类型替换。

Template<typename AnyType>

Void Swap(AnyType& a,AnyType& b)

{

AnyType temp;

Temp=a;

A=b;

B=temp;

}

**关键字template和typename是必需的，除非使用关键字class代替typename。必须使用尖括号**。

如果需要多个将同一种算法用于不同类型的函数，请使用模板。

**函数模板不能缩短可执行程序。最终代码不包含任何模板，而只包含了为程序生成的实际函数**。

并非所有的模板参数都必须是模板参数类型。

当无法使用模板重载来提供其他的代码，可以提供一个具体化函数定义——称为显式具体化。

在代码中包含模板本身并不会生成函数定义，它只是一个用于生成函数定义的方案。编译器使用模板为特定类型生成函数定义时，得到的是模板实例。

模板并非函数定义，但使用int的模板实例是函数定义。这种实例化方式被称为隐式实例化。

试图在同一个文件中使用童一鸿类型的显示实例和显式具体化将出错。

隐式实例化、显示实例化、和显式具体化统称为具体化。它们表示的都是具体类型的函数定义。

函数匹配从最佳到最差的顺序如下：

完全匹配，常规函数优先于模板。

提升转换（如char和short自动转换为int）

标准转换（如int转换为char）

用户定义的转换，如类声明中定义的转换。

表8.1

完全匹配允许的无关紧要转换

|  |  |
| --- | --- |
| 从实参 | 到形参 |
| Type | Type& |
| Type& | Type |
| Type[] | \*Type |
| Type(语句集合---argument-list) | Type(\*)(argument-list) |
| Type | const Type |
| Type | volatile Type |
| Type\* | const Type |
| Type\* | volatile Type\* |

一个完全匹配优于另一个的情况是，其中一个是非模板函数，而另一个不是。在这种情况下，非模板函数将优于模板函数。

如果两个完全匹配的函数都是模板函数，则较具体的模板函数优先。

术语最具体并不一定意味着显示具体化，而是指编译器推断使用哪种类型时执行的转换最少。

如：

Template<class Type> void recycle(Type t);#1

Template<class Type>void recycle(Type\* t);#2

Struct blot{int a;char b[10];};

Blot ink={25,”spots”};

…

Recycle(&ink);

Recycle(&ink)调用与#1模板匹配，匹配时将Type解释为blot\*。Recycle(&ink)函数调用也与#2模板匹配，这次Type被解释为ink。因此将两个隐式实例——recycle<blot\*>(blot\*)和recycle<blot>(blot\*)发送到可行函数池中。

在这两个模板函数中，recycle<blot\*>(blot\*)被认为是更具体的，因为在生成过程中，它需要进行转换更少。

**重载解析寻找最匹配的函数。如果只存在一个这样的函数，则选择它；如果存在多个这样的函数，但其中只有一个是非模板函数，则选择该函数；如果存在多个合适的函数，且它们都为模板函数，但其中有一个函数比其他函数更具体，则选择该函数。如果有多个同样合适的非模板函数或模板函数但没有一个函数比其他函数更具体，则函数调用将是不确定的**。

内存模型和名称空间

单独编译

可将程序分为三部分。

头文件：包含结构声明和使用这些结构的函数的原型。

源代码文件：包含与结构有关的函数的代码。

源代码文件：包含调用与结构相关的函数的代码。

头文件中常包含的内容:

函数原型。

使用#define 或 const定义的符号常量。

结构声明。

类声明。

模板声明。

内联函数。

被声明为const的数据和内联函数有特殊的链接属性。

在IDE中，不要将头文件加入到项目列表中，也不要在源代码文件中使用#include来包含其他源代码文件。

**在编译指令#ifndef下的代码意味着仅当没有使用预处理编译指令#define定义名称时，才处理#ifndef和#endif之间的语句**。

通常，使用#define语句来创建符号常量，如：

#define MAXIMUM 4096

为保持通用性，C++标准使用了术语翻译单元，而不是文件；文件并不是计算机组织信息时的唯一方式。

在链接编译模块时，请确保所有对象文件或库都是由同一个编译器生成的。如果有源代码，通常可以用自己的编译器重新编译源代码来消除链接错误。

存储持续性、作用域和链接性

C++几种不同的方案来存储数据：

自动存储持续性：在函数定义中声明的变量的存储持续性为自动的。它们在程序开始执行其所属的函数或代码块时被闯将，在执行完函数或代码块时，它们使用的内存被释放。

静态存储持续性：在函数定义外定义的变量和使用关键字static定义的变量的存储持续性都为静态。它们在程序整个运行过程中都存在。

线程存储持续性：当前，多核处理器很常见，这些CPU可同时处理多个执行任务。这些CPU可同时处理多个执行任务。这让程序能够将计算机放在可并行处理的不用线程中。如果变量是使用关键字

动态存储持续性：用new运算符分配的内存将一直存在，直到使用delete运算符将其释放或程序结束为止。这种内存的存储持续性为动态，有时候称为自由存储或堆。

**作用域描述了名称在文件的多大范围可见。**

**链接性描述了名称如何在不同的单元间共享。**

**链接性为外部的名称可在文件间共享，链接性为内部的名称只能由一个文件中的函数共享。自动变量的名称没有链接性，因为它们不能共享。**

为了对自动变量进行管理。**常用办法是留出一段内存，并将其视为栈**。

被称为栈，是由于新数据被象征性地放在原有数据的上面。

程序**使用两个指针来跟踪栈，一个指针指向栈低——栈的开始位置，另一个指针指向堆顶—下一个可用内存单元**。

当函数被调用时，其**自动变量被加入到栈中，栈顶指针指向变量后面的下一个可用的内存单元**。

函数结束时，**栈顶指针被重置为函数被调用当前的值，从而释放新变量使用的内存**。

**栈是LIFO（后进先出）的**，即最后加入到栈中的变量首先被弹出。简化了参数传递。

register它建议编译器使用CPU寄存器来存储自动变量

现在关键字register只是显示地指出变量是自动

内部的静态持续变量，必须在代码块的外面声明它，并使用static限定符;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 存储描述 | 持续性 | **作用域** | **链接性** | **如何声明** |
| **自动** | **自动** | **代码块** | **无** | **在代码块中** |
| **寄存器** | **自动** | **代码块** | **无** | **在代码块中，使用关键字static** |
| **静态，无链接性** | **静态** | **代码块** | **无** | **在代码块中，使用关键字static** |
| **静态，外部链接性** | **静态** | **文件** | **外部** | **不在任何函数内** |
| **静态，内部链接性** | **静态** | **文件** | **内部** | **不在任何函数内，使用关键字static** |

以上表格分配内存的方案不适用于new运算符分配的内存

**C++代码中，**空指针用0表示，但内部可能采用非零表示**。**

**结**构成员被0初始化且填充位都被设置为0。

**零初始化和常量表达式初始化被统称为静态初始化，编译器处理文件时初始化变量。动态初始化意味着变量将在编译后初始化。**

**在每个使用外部变量的文件中，都必须声明它；C++有单定义规则，该规则指出，变量只能有一次定义。因此C++提供了两种变量声明，一种**是定义声明 **简称定义，它给变量分配存储空间；另一种是**引用声明 **简称声明，它不给变量分配存储空间，因为它引用已有的变量引用声明使用关键字extern，且不进行初始化。**

**如果要在多个文件中使用外部变量，**只需在一个文件中包含该变量的定义**，但在使用该变量的其他所有文件中，都**必须使用关键字extern声明它**。**

**程序越能避免对数据进行不必要的访问，就越能保持数据的完整性。很重要 很重要 很重要**

**在多文件程序中，可以在一个文件中定义一个外部变量。使用该变量的其他文件必须使用关键字extern声明它。**

**可使用外部变量在多文件程序的不同部分之间共享数据：**可使用链接性为内部的静态变量在同一个文件中到多个函数之间共享数据。

**关键字thread\_local指出变量的持续性与其所属线程的持续性相同。thread\_local变量之余线程，犹如常规静态变量之余整个程序。**

**关键字mutable的含义将根据const来解释。**

**关键字volatile表明，即使程序代码没有对内存单元进行修改，其值也可能发生变化。**

**关键字mutable表明，即使结构（或类）变量为const，其某个成员也可以修改。**

**如：**

**struct data**

**{**

**char name[30];**

**mutable int accesses;**

**};**

**const data veep={“这是char数组” , 0};**

**veep.accesses++;**

**程序员希望某个常量的链接性为外部的，则可以使用extern关键字来覆盖默认内部链接性：**

**Extern const int states=50;**

**函数也具有链接性，所有函数的存储持续性都自动为静态的，即在整个程序执行期间都一直存在。在默认情况下，函数的链接性为外部的，即可以在文件间共享。可以在函数原型中使用关键字extern来指出函数是在另一个文件中定义的。还可以使用关键字static将函数的链接性设置为内部的。必须同时在原型和定义中使用关键字**

**单定义规则适用于非内联函数。**

**C++要求同一个函数的所有内联定义都必须相同。**

如果该文件中的函数原型指出该函数时静态的，则编译器将只在该文件中查找函数定义；否则，编译器将在所有的程序文件中查找**。**

使用new运算符分配的内存，这种内存被称为动态内存**。**

**通常情况下，编译器使用三块独立的内存，一块用于静态变量，一块用于自动变量，一块用于动态存储。**

**如果要为内置的标量类型分配存储空间并初始化，可在类型名后面加上初始值，并将其用括号括起。如果要初始化常规结构或数组，需要使用大括号的列表初始化。**

**通常，new负责在堆中找到一个足以能够满足要求的内存块。new运算符还有一种变体，被称为定位new运算符，它让您能够指定要使用的位置。**

**要使用定位new特性，首先需要包含头文件new，然后将new运算符用于提供了所需地址的参数。**

**定位new运算符使用传递给它的地址，它不跟踪那些内存已被使用，也不查找未使用的内存块。**

delete指令只能用于指向常规new运算符分配的堆内存**。**

****名称空间****

这个小节没啥好说的呀。

使用关键字namespace；作用域解析运算符::和using指令

namespace使用如：

namespace name

{

double pail;

Void fetch{}

Struct well{. . .}

Double bucket(double n){ . . .}

}

using使用如：

using namespace name;

using name::pail;

如果名称空间个声明区域定义了相同的名称。如果试图使用using声明将名称空间的名称导入该声明区域，则这两个名称会发生冲突，从而出错。如果使用using编译指令将该名称空间的名称导入该声明区域，则局部版本将隐藏空间版本。

名称空间经常放在头文件中，头文件经常包含的内容：常量、结构定义和函数原型。

如果using声明没有描述函数的返回类型或函数特征标，而只给出了名称。如果函数被重载，则一个using声明将导入所有的版本**。**

****对象和类****

抽象和类

#### 访问控制

关键字private和public描述了对类成员的访问控制。使用类对象的程序都可以直接访问公有部分，但只能通过公有成员函数来访问对象的私有成员。

**类设计尽可能将公有接口与实现细节分开。公有接口表示设计的抽象组件。将实现细节放在一起并将它们与抽象分开被称为封装。**

数据隐藏不仅可以防止直接访问数据，还让开发者无需了解数据是如何被表示的。

从使用类的角度看，使用哪种方法没有什么区别。所需要知道的只是各种成员函数的功能：也就是说，只需要知道成员函数接受什么样的参数以及返回什么类型的值。原则是将实现细节从接口设计中分离出来。如果以后找到了更好的、实现数据表示或成员函数细节的方法，可以对这些细节进行修改，而无需修改程序接口，这使程序维护起来更容易。

#### 控制对成员的访问：公有还是私有

无论类成员是数据成员还是成员函数，都可以在类的公有部分或私有部分中声明它。由于隐藏数据的目标，因此数据项通常放在私有部分，组成类接口的成员函数放在公有部分：否则，就无法从程序中调用这些函数。

### 实现类成员函数

**成员函数特点：**

**定义成员函数时，使用作用域解析运算符（::）来标识函数所属的类；**

**类方法可以访问类的private组件。**

首先，成员函数的函数头使用作用域运算符解析（::）来指出函数所属的类。例如：

void Stock::update(double price)

这种表示方法意味着我们定义的update()函数时Stock类的成员。这不仅将update()标识为成员函数，还意味着我们可以将另一个类的成员函数也命名为update()。

#### 内联方法

其定义位于类声明中的函数都将自动成为内联函数，类声明常将短小的成员函数作为内联函数。

内联函数的特殊规则要求在每个使用它们的文件中都对其进行定义。确保内联定义对多文件程序中的所有文件都可用的、最简便的方法是：将内联定义放在定义类的头文件中。

#### 方法使用哪个对象

**创建的每个新对象都有自己的存储空间，用于存储其内部变量和类成员；但同一个类的所有对象共享同一组类函数，即每种方法只有一个副本。**

类的构造函数和析构函数

构造函数没有返回值，也没有void类型声明，构造函数没有声明类型。

### 使用构造函数

第一种为显式调用构造函数：

Stock food = Stock(“value”,250,1.25);

第二种是隐式调用构造函数；

Stock garment(“Furry Mason”,50,2.5);

每次创建类对象时，C++都使用类构造函数。下面是将构造函数与new一起使用的方法：

Stock\* pastook = new Stock(Electroshook Games”,18,19.0);

这条语句创建一个Stock对象，将其初始化为参数提供的值，并将该对象的地址赋给pstock指针。在这种情况下，对象没有名称，但可以使用指针来管理该对象。

#### 默认构造函数

默认构造函数是在未提供显式初始值时，用来创建对象的构造函数。也就是说，它是用于下面这种形式的构造函数：

Stock fluffy\_the\_cat;

**当且仅当没有定义任何构造函数时，编译器才会提供默认构造函数。为类定义了构造函数后，程序员就必须为它提供默认构造函数。**

定义默认构造函数的方式有两种。一种是给已有构造函数的所有参数提供默认值：

Stock(const string& co=”Error”,int n = 0,double pr = 0.0);

另一种方式是通过函数重载来定义另一个构造函数——一个没有参数的构造函数。

Stock();

在设计类时，通常应该提供对所有类成员做隐式初始化的默认构造函数。

#### 析构函数

**用构造函数创建对象后，程序负责跟踪该对象，直到其过期为止。**

析构函数与构造函数一样，析构函数也可以没有返回值和声明类型。但是构造函数没有参数。

代码说明：

Stock stock1(“value”,12,20.0);

创建一个名为stock1的Stock对象，并将其数据成员初始化为指定的值：

Stock stock2 = Stock(“value”,2,2.0);

创建一个名为stock2的Stock对象。

第一种是使其行为和第一种语法完全相同：

第二种方式是允许调用构造函数来创建一个临时对象，然后将该临时对象赋值到stock2中，并丢弃它。如果使用这种方式，则将为临对象调用析构函数，因此会执行析构函数中的语句。

Stock2=stock1;

这表明可以将一个对象赋给同类型的另一个对象。

在默认情况下，将一个对象赋给同类型的另一个对象时，C++将源对象的每个数据成员的内容复制到目标对象中相应的数据成员中。

Stock1=Stock(“value”,10,50.0);

如果Stock1对象已经存在，则这条语句不是对Stock1进行初始化，而是将新值赋给它。这是通过让构造程序创建一个新的、临时的对象，然后将其内容复制给stock1来实现。

如果即可以通过初始化，也可以通过赋值来设置对象的值，则应采用初始化方式。通常这种方式的效率更高。

Const成员函数

Const Stock land = Stock(“value”);

Land.show();

编译器会将拒绝第二行。因为show()函数的代码无法确保调用对象不被修改。

这里需要一种新的语法保证函数不会修改调用的对象。C++的解决方法是**将const关键字放在函数的括号后面。**如：

Void show() const;

函数定义的开头应该是：

Void stock::show() const

以这种方式声明和定义的类函数被称为const成员函数。就像应尽可能将const引用和指针用作函数形参一样，只要类方法不修改调用对象，就应将其声明为const。

this指针

this指针指向用来调用成员函数的对象

每个成员函数都有一个this指针。this指针指向调用对象。如果方法需要要引用整个调用对象，则可以使用表达式\*this。在函数的括号后面使用const限定符将this限定为const,这样将不能使用this来修改对象的值。然而，要返回的并不是this，因为this是对象的地址，而是对象本身，即\*this。

对象数组

数组声明要求，这个类要么没有显示地定义任何构造函数，要么定义了一个显示默认构造函数。

也可以用构造函数来初始化数组元素。在这种情况下，必须为每个元素调用构造函数。

初始化对象数组的方案是，首先使用默认构造函数创建数组元素，然后花括号的构造函数将创建临时对象，然后将临时对象的内容复制到相应的元素中。

类作用域

在类中定义的名称的作用域都为整个类，作用域为整个类的名称只在该类中是已知，在类外是不可知的。可以在不同类中使用相同的类成员而不会引起冲突。类作用域意味着不能从外部直接访问类的成员，公有成员也是一样。所以要调用公有成员函数，必须通过对象：

**在定义成员函数时，必须使用作用域解析运算符**：

**构造函数名称在被调用时，类成员名才能被识别，因为它的名称与类名相同**。

**使用类成员时，必须根据上下文使用直接成员运算符（.）、间接成员运算符（->）或作用域解析运算符（::）**。

### 作用域内枚举

传统的枚举存在一些问题，两个枚举定义中的枚举量可能发生冲突（因为违法了C++二义性）。

为此C++提供了一种新枚举，其枚举量的作用域为类。枚举声明如下：

Enum class egg{Small,Medium,Large,Jumbo};

Enum class t\_shirt{ Small , Medium , Large ,Xlarge};

也可以使用关键字struct代替class。这两种方式都需要使用枚举名来限定枚举量：

Egg choice = egg：：large;

t\_shir Floyd = t\_shirt::large;

枚举量作用域为类后，不同枚举定义中的枚举量就不会产生冲突。

抽象数据类型

栈特征：

可创建空栈

可将数据添加到堆顶（压入）

可从栈顶删除数据项（弹出）

可查看栈是否填满

可查看栈是否为空

使用类

运算符重载

使用运算符重载，需使用被称为运算符函数的特殊函数形式。运算符函数格式如下：

operatorop(argument-list)

如：

operatorop+()重载+运算符,operator\*()重载\*运算符。op必须是有效的C++运算符，不能虚构一个新的符号。例如，不能有operator@()这样的函数，因为C++中没有@运算符。但，operator[]()函数将重载[]运算符，因为[]是数组索引运算符。

案例警告：不要返回指向局部变量或临时对象的引用。函数执行完毕后，局部变量和临时变量将消失，引用将指向不存在的数据。

### 重载限制

多数C++运算符都可以使用表11.1的方式重载。

**可重载的运算符**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| + | - | \* | / | % | ^ |
| & | | | ~= | ! | = | < |
| > | += | -= | \*= | /= | %= |
| ^= | &= | |= | << | >> | >>= |
| <<= | == | != | <= | >= | && |
| || | ++ | -- | , | ->\* | -> |
| () | [] | new | delete | new[] | delete[] |

重载的运算符不必是成员函数，但必须至少有一个操作数是用户定义的类型。

运算符重载限制：

重载后的运算符必须至少有一个操作数是用户定义的类型，

使用运算符时不能违反运算符原来的句法规则。不能修改运算符的优先级。

不能创建新运算符。

不能重载以下运算符：

sizeof：sizeof运算符

.：成员运算符

.\*：成员指针运算符

::：作用域解析运算符

?:：条件运算符

typeid：一个RTTI运算符

const\_cast：强制类型转换运算符

dynamic\_cast：强制类型转换运算符

reinterpret\_cast：强制类型转换运算符

static\_cast：强制类型转换运算符

以下运算符只能通过成员函数重载

=：赋值运算符

()：函数调用运算符

[]：下标运算符

->：通过指针访问类成员的运算符

不要将某些\*运算符重载成交换两个数据对象的数据成员。表示法中没有任何内容可以表民运算符完成的工作，因此最好定义一个其具有说明性的类方法。

友元

友元的3种形式：

友元函数

友元类

友元成员函数

通过让函数成为类的友元，可以赋予该函数与类的成员函数相同的访问权限。

### 创建友元

创建友元函数的第一步是将其原型放在类声明中，并在原型声明前加上关键字friend：

friend typeName operator\*(double m, const typeName&t);

该原型意味着下面两点：

虽然operator\*()函数是在类声明中声明的，但它不是成员函数，因此不能使用成员运算符来调用；

虽然operator\*()函数不是成员函数，但它与成员函数访问权限相同。

因为它不是成员函数，所以不要使用typeName::限定符。不要在定义中使用关键字friend，定义如：

typeName operator\*(double m,const typeName& t)

{

typeName result;

long totalminutes = t.hours \* mult \* 60 +t.minutes\*mult;

result.hours = totalminutes/60;

result.minutes = totalminutes%60;

return result;

}

如果要为类重载运算符，并将非类的项作为其第一个操作数，则可以用友元函数来反转操作数的顺序。

警告：只有在类声明中的原型中才能使用fiend关键字。除非函数定义也是原型，否则不能在函数定义中使用该关键字。

重载运算符：作为成员函数还是非成员函数

一般情况下，非成员函数应是友元函数，这样它才能直接访问类的私有数据。

加法运算符需要两个操作数。**对于成员函数版本来说，一个操作数通过this指针隐式地传递，另一个操作数作为函数参数显示地传递；对于友元版本来说，两个操作数都作为参数来传递**。

非成员版本的重载运算符函数所需的形参数目与运算符使用的操作数数目相同；而成员版本所需的参数数目最少一个，因为其中的一个操作数是被隐式地传递的调用对象。

类的动态内存分配

内存分布最好是在程序运行时（而不是编译时）确定诸如使用多少内存等问题。

C++的方法是，在类构造函数中使用new运算符在程序运行时分配所需的内存。

动态内存和类

C++使用new和delete运算符来动态控制内存。但在类中使用这些运算符将导致寻多编程问题。所以析构函数时必需的，有时还需要重载赋值运算符。

### 静态类成员

静态类成员有一个特点：无论创建多少对象程序都只创建一个静态类变量副本。

不能在类声明中初始化静态成员变量，因为声明描述了如何分配内存，但并不分配内存。

可以在类声明外使用单独语句进行初始化，因为静态类成员时单独存储，而不是对象的组成部分。

静态数据成员在类声明中声明，在包含类方法的文件中初始化。初始化时使用作用域运算符来指出静态成员所属的类。如果静态成员时const整数类型或枚举类型，则可以在类声明中初始化。

删除对象可以释放对象本身占用的内存，但并不能自动释放属于对象成员的指针指向的内存（需要使用析构函数）

在构造函数中使用new来分配内存时，必须在相应的析构函数中使用delete来释放内存。如果使用new[]来分配内存，则应使用delete[]来释放内存。

### 特殊成员函数

C++自动提供了以下成员函数：

默认构造函数，如果没有定义构造函数

默认析构函数，如果没有定义

复制构造函数，如果没有定义

赋值运算符，如果没有定义

地址运算符，如果没有定义

#### 复制构造函数

复制构造函数用于将一个对象复制到新创建的对象中。

复制构造函数原型：class\_name(const Class\_name&);

#### 何时调用复制构造函数

新建一个对象并将其初始化为同类现有对象时，复制构造函数都将被调用。

每当程序生成了对象副本时，编译器都将使用复制构造函数。

由于按值传递对象将调用复制构造函数，因此应该按引用传递对象。（可以寄节省调用构造函数的时间及存储新对象的空间）

#### 默认的复制构造函数的功能

默认的复制构造函数逐个复制非静态成员（成员复制=浅复制），复制的是成员的值。

如果类中包含了使用new初始化的指针成员，应当定义一个复制构造函数，以复制指向的数据，而不是指针（深复制），

函数应当避免将对象赋给自身；否则，给对象重新赋值前，释放内存操作可能删除对象的内容。

## 静态类成员函数

如果函数定义是独立的，则其中不能包含关键字static。

静态成员函数不能使用this指针。

静态成员函数不与特定的对象相关联，只能使用静态数据成员。

在构造函数中使用new时应注意的事项

如果在构造函数中使用new来初始化指针成员，则应在析构函数中使用delete。

new和delete必须相互兼容。new对应于delete，new[]对应于delete[]。

如果有多个析构函数，则必须以相同的方式使用new，要么都带中括号，要么都不带。因为只有一个析构函数，所有的构造函数都必须与它兼容。然而可以在一个构造函数中使用new初始化指针，而在另一个构造函数中将指针初始化为空，这是因为delete可以用于空指针。

应定义一个复制构造函数，通过深度复制将一个对象初始化为另一个对象。

应定义一个赋值运算符，通过深度复制将一个对象复制给另一个对象。

有关返回对象的说明

### 返回指向对象的引用

//版本1

Vector Max(const Vecor& v1,const Vector& v2)

{

If(v1.magval()>v2.magval())

Return v1;

else

return v2;

}

//版本2

const Vector& Max(const Vector& v1,const & v2)

{

If(v1.magval()>v2.magval())

Return v1;

else

return v2;

}

返回对象将调用复制构造函数，而返回引用不会。

### 返回对象

如果返回的对象时被调用函数中的局部变量，则不应该按引用方式返回它，因为在被调用函数执行完毕时，局部对象将调用其析构函数。

### 返回const对象

如果方法或函数要返回局部对象，则应返回对象，而不是指向对象的引用。在这种情况下，将使用复制构造函数来生成返回的对象。如果方法或函数要返回一个没有公有复制构造函数的类的对象，它必须返回一个指向这种对象的引用

## 使用指向对象的指针

### 再谈new和delete

在下述情况下析构函数将被调用

如果对象时动态变量，则当执行完定义该对象的程序块时，将调用该对象的析构函数。

如果对象时静态变量，则在程序结束时将调用对象的析构函数。

如果对象使用new创建的，则仅当您显示使用delete删除对象时，其析构函数才会被调用。

### 指针和对象小结

使用对象指针时，需要注意几点

使用常规表示法来声明指向对象的指针：  
String\* glamour;

可以将指针初始化为指向已有的对象：

String\* first = &sayings[0];

可以使用new来初始化指针，这将创建一个新的对象

String\* favorite = new String(sayings[choice]);

对类使用new将调用相应的类构造函数来初始化新创建的对象：

可以使用->运算符通过指针访问类方法：

可以对对象指针应用解除引用运算符(\*)来获得对象：

### 再谈定位new运算符

delete可与常规new运算符配合使用，但不能与定位new运算符配合使用。

这种情况解决方案是，显式地为使用new运算符创建的对象调用析构函数。显式地调用析构函数函数时，必须制定要销毁的对象。

仅当所有对象都被销毁后，才能释放用于存储这些对象的缓冲区。

## 复习各种技术

### 其构造函数使用new类

如果类使用new运算符来分配类成员指向的内存在设计时应采取一些预防措施

对于指向的内存是由new分配的所有类成员，都应在类的析构函数中对其使用delete，该运算符将释放分配的内存。

如果析构函数通过对指针类成员使用delete来释放内存，则每个构造函数使用的是new[]

来初始化指针，或将它设置为空指针。

构造函数中要么使用new[]，要么使用new，而不能混用。如果构造函数使用的是new[]，则析构函数应使用delete[]; 如果构造函数使用的是new，则析构函数应使用delete。

应定义一个分配内存的复制构造函数。这样程序将能够将类对象初始化为另一个类对象。

这种构造函数的原型如下：

className(const className&)

应定义一个重载赋值运算符的类成员函数，

## 队列模拟

### 队列类

C++提供了一种特殊的语法，它叫做成员初始化列表。用来在创建类对象时进行数据初始化。

只有构造函数可以使用这种初始化列表语法。对于const类成员，必须使用这种语法。被声明为引用的类成员，也必须使用这种语法；

初始化工作实在对象创建时完成的，此时还未执行括号中的任何代码。注意：

这种格式只能用于构造函数；

必须使用这种格式来初始化非静态const数据成员（C++11之前）；

必须用这种格式来初始化引用数据成员。

数据成员被初始化的顺序与它们出现在类声明中的顺序相同，与初始化器中的排列顺序无关。

不能将成员初始化列表语法用于构造函数之外的其他类

类继承

通过继承能够完成的工作：

可以在已有的类的基础上添加功能。

可以给类添加数据。

可以修改类方法的行为。

## 一个简单的基类

从一个类派生出另一个类时，原始类称为基类，继承类称为派生类。

### 派生一个类

通过公有派生，派生类对象将包含基类对象。

使用公有派生，基类的公有成员将称为派生类的公有成员；基类的私有部分也将成为派生类的一部分，但只能通过基类的公有和保护方法访问。

所以派生类有以下特征：

派生类对象存储了基类的数据成员

派生类对象可以使用基类的方法

需要在派生类继承特性中添加：

派生类自己的构造函数。

派生类可以根据需要添加额外的数据成员和成员函数。

构造函数必须给新成员和继承的成员提供数据。

### 构造函数：访问权限

派生类不能直接访问基类的私有成员，而必须通过基类方法进行访问。

**创建派生类对象时，程序首先创建基类对象。从概念上说，基类对象应在当程序进入派生类构造函数之前被创建**。C++使用成员初始化列表语法来完成这种工作。

如果省略成员初始化列表：

必须首先创建基类对象，如果不调用基类构造函数，程序将使用默认的基类构造函数。

派生类构造函数的要点如下：

首先创建基类对象

派生类构造函数应通过成员初始化列表将基类信息传递给基类构造函数；

派生类构造函数应初始化派生类新增的数据成员。

如果没有提供显示构造函数，因此将使用隐式构造函数。释放对象的顺序与创建对象的顺序顺序相反，即首先创建执行派生类的析构函数，然后自动调用基类的析构函数。

### 派生类和基类之间的特殊关系

派生类与基类之间有一些特殊关系。其中之一是派生类对象可以使用基类的方法，条件是方法不是私有的；

另外两个重要的关系是；基类指针可以在不进行显示类型转换的情况下指向派生类对象；基类引用可以在不进行显示类型转换的情况下引用派生类对象；

C++要求引用和和指针类型与赋给的类型匹配，但有列外。

这种例外是单向的，不可以将基类对象和地址赋给派生类引用和指针；

对于形参为指向基类的指针的函数，也有相似的关系。它可以使用基类对象的地址或派生类对象的地址为实参；

引用兼容性属性也让您能够将基类对象初始化为派生类对象。

## 继承：is-a关系

C++有3中继承方式：公有继承、保护继承和私有继承。公有继承是最常用的方式，它建议一种is-a关系，即派生类对象也是一个基类对象，可以对基类对象执行的任何操作，也可以对派生类对象执行。

## 多态公有继承

多态----具有多种形态：

方法或函数的行为应取决于调用该方法的对象。

即同一个方法的行为随上下文而异。

有两种重要的机制可用于时态多态公有继承：

在派生类中重新定义基类的方法

使用虚方法

### C++中虚的使用

如果方法是通过引用或指针而不是对象调用的，它将确定使用哪一种方法。如果没有使用关键字virtual，程序将根据引用类型或指针类型选择方法；如果使用了virtual，程序将根据引用或指针指向的对象的类型来选择方法。

方法在基类中被声明为虚的后，它在派生类中将自动成为虚方法

如果要在派生类中重新定义基类的方法，通常应将基类方法声明为虚的。这样，程序将根据对象类型而不是引用引用或指针的类型来选择方法版本。为基类声明一个虚析构函数也是一种惯例。

**非构造函数不能使用成员初始化列表，但派生类方法可以调用公有的基类方法**。

## 静态联编和动态联编

将源代码中的函数调用解释为执行特定的函数代码块被称为函数名联编。

在编译过程中进行联编被称为静态联编。

动态联编：

编译器必须生成能够在程序运行时选择正确的虚方法的代码，这被称为动态联编。

### 指针和引用类型的兼容性

将派生类引用或指针转换为基类引用或指针被称为向上强制转换。

相反的过程---将基类指针或引用转换为派生类指针或引用—称为向下强制转换如果不使用显式类型转换，则向下强制转换是不允许的。

隐式向上强制转换使基类指针或引用可以指向基类对象或派生类对象，因此需要动态联编。C++使用虚成员函数来满足需求。

### 有关虚函数注意事项

在基类方法的声明中使用关键字virtual可使该方法在基类以及所有的派生类中是虚的。

如果使用指向对象的引用或指针来调用虚方法，程序将使用为对象类型定义的方法，而不使用为引用或指针类型定义的方法。这称为动态联编或晚期联编。这种行为十分重要，这样基类只恨或引用可以指向派生类对象。

如果定义的类将被用作基类，则应将那些要在派生类中重新定义的类方法声明为虚的。

通常应给基类提供一个虚析构函数，即使它并不需要析构函数。

#### 没有重定义

如果派生类没有重定义函数，将使用该函数的基类版本。如果派生类位于派生链中，则将使用最新的虚数版本，例外的情况的基类版本是隐藏的。

#### 重新定义将隐藏方法

重新定义不会生成函数的两个重载版本，而是隐藏了之前的基类版本。总之，重新定义继承的方法并不是重载。

**如果重新定义继承的方法，应确保与原来的原型相同，但如果如果按返回类型是基类引用或指针，则可以修改为指向派生类的引用或指针。这种特性被称为返回类型协变，因为允许返回类型随类类型的变化而变化**。

**如果基类声明被重载了，则应在派生类红重新定义所有的基类版本**。

## 访问控制：protected

派生类的成员可以直接访问基类的保护成员，但不能直接访问基类的私有成员。

最好对类数据成员采用私有访问控制，不要使用保护访问控制；同时通过基类方法使派生类能够访问基类数据。

## 抽象基类

C++纯虚函数：

在虚函数声明的结尾处为=0。

## 继承和动态内存分布

翻阅书籍p516-p522

## 类设计回顾

### 编译器生成的成员函数

#### 默认构造函数

默认构造函数要么没有参数，要么所有的参数都有默认值。

自动生成的默认构造函数的功能之一是：调用基类的默认构造函数以及调用本身是对象的成员所属的默认构造函数。

如果派生类构造函数的成员初始化列表中没有显式调用基类构造函数，则编译器将使用基类的默认构造函数来构造派生类对象的基类部分。如果基类没有构造函数，将导致编译阶段错误。

如果定义了某种构造函数，编译器将不会定义默认构造函数。如果需要默认构造函数则必须自己提供。

提供构造函数意义之一是确保对象总能被正确地初始化。如果类包含指针成员，则必须初始化这些成员。所以最好提供一个显式默认构造函数。

#### 复制构造函数

复制构造函数接受其所属类的对象作为参数。

下述情况下，将使用复制构造函数：

将新对象初始化为一个同类对象；

按值将对象传递给函数；

函数按值返回对象；

编译器生成临时对象；

某些情况下，成员初始化是不合适的。

如，使用new初始化的成员指针通常要求执行深复制，或者类可能包含需要修改的静态变量。

在上述情况下，需要定义自己的复制构造函数。

#### 赋值运算符

默认的赋值运算符用于处理同类对象之间的赋值。

语句创建新的对象，则使用初始化；如果语句修改已有对象的值，则是赋值；

默认赋值为成员函数。如果成员为类对象，则默认成员赋值将使用相应类的赋值运算符。

### 其他的类方法

#### 析构函数

一定要定义显式析构函数来释放类构造函数使用new分配的所有内存，并完成类对象所需的任何特殊的清理工作。

#### 转换

转换函数：

使用一个参数就可以调用的构造函数定义了从参数类型到类类型的转换。

将可转换的类型传递给以类为参数的函数时，将调用转换构造函数。

在带一个参数的构造函数原型（转换构造函数）中使用explicit将禁止进行隐式转换，但仍允许显示转换；

转换函数可以是没有参数的类成员函数，也可以时返回类型被声明为目标类型的类成员函数。

#### 按值传递对象与传递引用

编写使用对象作为参数的函数时，应按引用而不是按值来传递对象。

按值传递对象涉及到生成临时拷贝，调用复制构造函数，然后调用析构函数。调用这些函数需要时间。

在继承使用虚函数时，被定义为接收基类引用参数的函数可以接受派生类。

如果函数不修改对象，应将参数声明为const引用。

#### 返回对象和返回引用

有些成员函数直接返回对象，而另一些则返回引用。有时方法必须返回对象，但如果可以不返回对象，则应返回引用。

返回引用原因在于，返回对象涉及生成返回对象的临时副本。

然而，并不总是可以返回引用。函数不能返回在函数中创建的临时对象的引用，因为当函数结束时，临时对象将消失，这是非法引用。这种情况下，应返回对象，以生成一个调用程序可以使用的副本。

通用的规则是，如果函数返回在函数中创建的临时对象，则不要引用。

如果函数返回的是通过引用或指针传递给它的对象，则应按引用返回对象。

#### 使用const

使用const时应特别注意。可以用它来确保方法不修改参数：

可以使用const来确保方法不修改调用它的对象

可以将返回引用的函数放在赋值语句的左侧，但可以使用const来确保引用或指针返回的值不能用于修改对象中的数据：

如果函数将参数声明为指向const的引用或指针，则不能将该参数传递给另一个函数，除非后者也确保了参数不会被修改。

### 公有继承的考虑因素

#### Is-a关系

表示is-a关系的方式之一是，无需进行显式类型转换，基类指针就可以指向派生类对象，基类引用可以引用派生类对象。

#### 什么不能被继承

构造函是不能继承的。

如果派生类构造函数没有使用成员初始化语法显式调用基类构造函数，将使用基类的默认构造函数。在继承链中每个类都可以使用成员初始化列表将信息传递给相邻的基类。C++11新增了一种让您能够继承构造函数的机制，但默认仍不继承构造函数。

析构函数也是不能继承。然而，在释放对象时，程序将首先调用派生类的析构函数，然后调用基类的析构函数。如果基类有默认析构函数，编译器将为派生类生成默认析构函数。

赋值预算福是不能继承的。

#### 赋值运算符

如果编译器发现程序将一个对象赋给同一个类的另一个对象，它将自动为这个类提供一个赋值运算符。

这个运算符的默认或隐式版本将采用成员赋值，即将原对象的相应成员赋给目标对象的每个成员。

如果对象属于派生类，编译器将使用基类的赋值运算符来处理派生对象红基类部分的赋值。

如果类构造函数使用new来初始化指针，则需要提供一个显式赋值运算符。因为对于派生类的基类部分，C++将使用基类的赋值运算符，所以不需要为派生类重新定义赋值运算符。

然而，如果派生类使用了new，则必须提供显式赋值运算符。

#### 私有成员与保护成员

对派生类而言，保护成员类似于公有成员；但对于外部而言，保护成员与私有成员类似。派生类可以直接访问基类的保护成员，但只能通过基类的成员函数来访问私有成员。

#### 虚方法

如果希望派生类能够重新定义方法，则应在基类中将方法定义为虚的。

#### 析构函数

基类的析构函数应到是虚的。

#### 友元函数

友元函数并非成员，因此不能继承

#### 有关使用基类方法的说明

以公有方式派生的类的对象可以通过多种方法来使用基类的方法。

派生类对象自动使用继承而来的基类方法，如果派生类没有重新定义该方法

派生类的构造函数自动调用基类的构造函数。

派生类的构造函数自动调用基类的默认构造函数，如果没有在成员初始化列表中指定其它构造函数。

派生类构造函数显式地调用成员初始化列表中的指定的基类构造函数。

派生类方法可以使用作用域解析运算符来调用公有的和受保护的基类方法。

派生类的友元函数可以通过强制类型转换，将派生类引用或指针转换为基类引用或指针，然后使用该引用或指针来调用基类的友元函数。

成员函数属性

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 函数 | 能否继承 | 成员还是友元 | 默认能否生成 | 能否为虚函数 | 是否可以有返回类型 |
| 构造函数 | 否 | 成员 | 能 | 否 | 否 |
| 析构函数 | 否 | 成员 | 能 | 能 | 否 |
| = | 否 | 成员 | 能 | 能 | 能 |
| & | 能 | 任意 | 能 | 能 | 能 |
| 转换函数 | 能 | 成员 | 否 | 能 | 否 |
| () | 能 | 成员 | 否 | 能 | 能 |
| [] | 能 | 成员 | 否 | 能 | 能 |
| -> | 能 | 成员 | 否 | 能 | 能 |
| op= | 能 | 任意 | 否 | 能 | 能 |
| new | 能 | 静态成员 | 否 | 否 | Void\* |
| delete | 能 | 静态成员 | 否 | 否 | void |
| 其他运算符 | 能 | 任意 | 否 | 能 | 能 |
| 其他成员 | 能 | 成员 | 否 | 能 | 能 |
| 友元 | 否 | 友元 | 否 | 否 | 能 |

C++中的代码重用

## 包含对象成员的类

### Valarray类简介

Valarray类是由头文件valarray支持的。它支持诸如将数组中所有元素的值相加以及在数组中找出最大和最小的值等操作。

私有继承

私有继承会将基类的公有成员和保护成员都将称为派生类的私有成员。

包含将对象作为一个命名的成员对象添加到类中，而私有继承将对象作为一个未被命名的继承对象添加到类中。我们使用术语“子对象”来实现通过继承或包含添加的对象。

使用多个基类的继承被称为多重继承（multiple inheritance,MI）。

### 使用包含还是私有继承

应使用包含来建立has-a关系；如果新类需要访问原有类的保护成员，或需要重新定义虚函数，则应使用私有继承。

|  |  |  |  |
| --- | --- | --- | --- |
| 特征 | 公有继承 | 保护继承 | 私有继承 |
| 公有成员变成 | 派生类的公有成员 | 派生类的保护成员 | 派生类的私有成员 |
| 保护成员变成 | 派生类的保护成员 | 派生类的保护成员 | 派生类的私有成员 |
| 私有成员变成 | 只能通过基类接口访问 | 只能通过基类接口访问 | 只能通过基类接口访问 |
| 能否隐式向上转换 | 是 | 是（在派生类中） | 否 |

## 多重继承

在继承时必须使用关键字public来限定每一个基类。这是因为，除非特别指出，否则编译器将认为是私有派生。

#### 虚基类

虚基类使得从多个类派生出的对象只能继承一个基类对象。

C++在基类是虚的时，禁止信息通过中间类自动传递给基类。

如果类由间接虚基类，则除非只需使用该虚基类的默认构造函数，否则必须显式地调用该虚基类的某个构造函数。

多重继承可能导致函数调用的二义性。

#### 混合使用虚基类和非虚基类

通过多种途径继承一个基类的派生类的情况。如果基类是虚基类，派生类将包含基类的一个子对象；如果基类不是虚基类，派生类将包含多个子对象。

当类通过多条虚途径和非虚途径继承某个特定的基类时，该类将包含一个表示所有的虚途径的基类子对象和分别表示各条非虚途径的多个基类子对象。

#### 虚基类和支配

派生类中的名称优先于直接或间接祖先类中的相同名称。

虚二义性规则与访问规则无关。

### MI小结

如果一个类从两个不同的类哪里继承了两个同名的成员，则需要在派生类中使用类限定符来区分它们。

虚基类变化：

从虚基类的一个或多个实例派生而来的类将只继承了一个基类对象。

为满足上述情况，必须满足以下要求：

有间接虚基类的派生类包含直接调用间接基类构造函数的构造函数，这对于间接非虚基类来说是非法的；

通过优先规则解决名称二义性

## 类模板

使用模板类注意事项：

必须显式地提供所需的类型，这与常规的函数模板是不同的，因为编译器可以根据函数的参数类型来确定要生成哪种函数。

实例化模板时，用作表达参数的值必须是常量表达式。

### 模板多功能性

#### 默认类型的模板参数

类模板的另一种新特性：可以为类型参数提供默认值

虽然可以为类模板类型参数提供默认值，但不能为函数模板参数提供默认值。然而可以为非类型参数提供默认值。

### 模板的具体化

#### 隐式实例化

它们声明一个或多个对象，指出所需的类型，而编译器使用通用模板提供的处方生成具体的类定义。

#### 显式实例化

当使用关键字template并指出所需类型来声明类时，编译器将生成类声明的显式实例化。声明必须位于模板定义所在的名称空间中。

#### 显式具体化

显式具体化是特定类型的定义。

#### 部分具体化

部分限制模板的通用性。

也可以通过为指针提供特殊版本来部分具体化现有的模板；

### 模板类和友元

非模板友元

约束模板友元，即友元的类型取决于类被实例化时的类型

非约束模板友元，即友元的所有具体化都是类的每一个具体化的友元

友元、异常和其他

这一章强烈建议看书

## 友元

友元类的所有方法都可以访问原始类的私有成员和保护成员

有啥好说的看网上的代码

## 嵌套类

包含类的成员函数可以创建和使用被嵌套类的对象；而仅当声明位于公有部分，才能在包含类的外面使用嵌套类，而且必须使用作用域解析运算符。

### 嵌套类和访问权限

|  |  |  |  |
| --- | --- | --- | --- |
| 声明位置 | 包含它的类是否可以使用它 | 从包含它的类派生而来的类是否可以使用它 | 在外部是否可以使用 |
| 私有部分 | 是 | 否 | 否 |
| 保护部分 | 是 | 是 | 否 |
| 共有部分 | 是 | 是 | 是，通过类限定符来使用 |

## 异常

### 异常机制

异常处理有3个组成部分：

引发异常；

使用处理程序捕获异常；

使用try块

执行throw语句类似于执行返回语句，因为它也将终止函数的执行；但throw不是将控制权返回给调用程序，而是导致程序沿函数调用序列后退，直到找到包含try块的函数