变量的声明和定义：  
  
对于下面的两句代码：  
void Func()  
{  
int a;  
int b=1;  
a=0;  
}  
对于第一行代码，编译器不会做任何事，它不会为它在栈中分配一点东西，直到第三句，a=0;时，编译器才会将其压入栈中。而对于int b=0;这一句，编译器就会生成一条指令，为它赋值。如果反汇编，看到的代码可能是这样的：  
push 1;  
push 0;  
当然，并不一定编译器就会样做，也有可能在声明int a时，编译器就会把一个废值入栈，到第三条再为其赋值，这要看编译器的具体取舍，所以，声明不一定不是定义，而定义一定是定义。  
但是，下面的声明，一定仅仅是声明：  
extern int a;  
这表时，有一个int变量a，它一定是在另外其他地方定义的，所以编译器此时一定不会做什么[分配内存](https://www.baidu.com/s?wd=%E5%88%86%E9%85%8D%E5%86%85%E5%AD%98&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)的事，因为它就是声明，仅仅表明下面的代码引用了一个符号，而这个符号是int类型的a而已。  
  
如果是指函数的声明和定义：  
声明：一般在头文件里，对编译器说：这里我有一个函数叫function() 让编译器知道这个函数的存在。  
  
定义：一般在源文件里，具体就是函数的实现过程 写明函数体。

**C++ 存储类**

存储类定义 C++ 程序中变量/函数的范围（可见性）和生命周期。这些说明符放置在它们所修饰的类型之前。下面列出 C++ 程序中可用的存储类：

## auto 存储类

自 C++ 11 以来，**auto** 关键字用于两种情况：声明变量时根据初始化表达式自动推断该变量的类型、声明函数时函数返回值的占位符。

C++98标准中auto关键字用于自动变量的声明，但由于使用极少且多余，在C++11中已删除这一用法。

根据初始化表达式自动推断被声明的变量的类型，如：

auto f=3.14; //double

auto s("hello"); //const char\*

auto z = new auto(9); // int\*

auto x1 = 5, x2 = 5.0, x3='r';//错误，必须是初始化为同一类型

## register 存储类

**register** 存储类用于定义存储在寄存器中而不是 RAM 中的局部变量。这意味着变量的最大尺寸等于寄存器的大小（通常是一个词），且不能对它应用一元的 '&' 运算符（因为它没有内存位置）。

{ register int miles; }

寄存器只用于需要快速访问的变量，比如计数器。还应注意的是，定义 'register' 并不意味着变量将被存储在寄存器中，它意味着变量可能存储在寄存器中，这取决于硬件和实现的限制。

## static 存储类

**static** 存储类指示编译器在程序的生命周期内保持局部变量的存在，而不需要在每次它进入和离开作用域时进行创建和销毁。因此，使用 static 修饰局部变量可以在函数调用之间保持局部变量的值。

static 修饰符也可以应用于全局变量。当 static 修饰全局变量时，会使变量的作用域限制在声明它的文件内。

在 C++ 中，当 static 用在类数据成员上时，会导致仅有一个该成员的副本被类的所有对象共享。

## 实例

#include <iostream> // 函数声明

void func(void);

static int count = 10; /\* 全局变量 \*/

int main() {

while(count--) {

func();

} return 0;

} // 函数定义

void func( void ) {

static int i = 5; // 局部静态变量

i++;

std::cout << "变量 i 为 " << i ;

std::cout << " , 变量 count 为 " << count << std::endl;

}

## extern 存储类

**extern** 存储类用于提供一个全局变量的引用，全局变量对所有的程序文件都是可见的。当您使用 'extern' 时，对于无法初始化的变量，会把变量名指向一个之前定义过的存储位置。

当您有多个文件且定义了一个可以在其他文件中使用的全局变量或函数时，可以在其他文件中使用 *extern* 来得到已定义的变量或函数的引用。可以这么理解，*extern* 是用来在另一个文件中声明一个全局变量或函数。

extern 修饰符通常用于当有两个或多个文件共享相同的全局变量或函数的时候，如下所示：

第一个文件：main.cpp

## 实例

#include <iostream>

int count ;

extern void write\_extern();

int main() {

count = 5;

write\_extern();

}

第二个文件：support.cpp

## 实例

#include <iostream>

extern int count;

void write\_extern(void) {

std::cout << "Count is " << count << std::endl;

}

在这里，第二个文件中的 *extern* 关键字用于声明已经在第一个文件 main.cpp 中定义的 count。现在 ，编译这两个文件，如下所示：

$ g++ main.cpp support.cpp -o write

这会产生 **write** 可执行程序，尝试执行 **write**，它会产生下列结果：

$ ./write

Count is 5

## mutable 存储类

**mutable** 说明符仅适用于类的对象，这将在本教程的最后进行讲解。它允许对象的成员替代常量。也就是说，mutable 成员可以通过 const 成员函数修改。

## thread\_local 存储类

使用 thread\_local 说明符声明的变量仅可在它在其上创建的线程上访问。 变量在创建线程时创建，并在销毁线程时销毁。 每个线程都有其自己的变量副本。

thread\_local 说明符可以与 static 或 extern 合并。

可以将 thread\_local 仅应用于数据声明和定义，thread\_local 不能用于函数声明或定义。

以下演示了可以被声明为 thread\_local 的变量：

thread\_local int x; // 命名空间下的全局变量

class X {

static thread\_local std::string s; // 类的static成员变量

};

static thread\_local std::string X::s; // X::s 是需要定义的

void foo() {

thread\_local std::vector<int> v; // 本地变量

}

函数**声明**告诉编译器函数的名称、返回类型和参数。函数**定义**提供了函数的实际主体。

# C++ 类 & 对象

C++ 在 C 语言的基础上增加了面向对象编程，C++ 支持面向对象程序设计。类是 C++ 的核心特性，通常被称为用户定义的类型。

类用于指定对象的形式，它包含了数据表示法和用于处理数据的方法。类中的数据和方法称为类的成员。函数在一个类中被称为类的成员。

## C++ 类定义

定义一个类，本质上是定义一个数据类型的蓝图。这实际上并没有定义任何数据，但它定义了类的名称意味着什么，也就是说，它定义了类的对象包括了什么，以及可以在这个对象上执行哪些操作。

类定义是以关键字 **class** 开头，后跟类的名称。类的主体是包含在一对花括号中。类定义后必须跟着一个分号或一个声明列表。例如，我们使用关键字 **class** 定义 Box 数据类型，如下所示：

class Box { //定义C++的类

public: double length; // 盒子的长度

double breadth; // 盒子的宽度

double height; // 盒子的高度

};

关键字 **public** 确定了类成员的访问属性。在类对象作用域内，公共成员在类的外部是可访问的。您也可以指定类的成员为 **private** 或 **protected**，这个我们稍后会进行讲解。

## 定义 C++ 对象

类提供了对象的蓝图，所以基本上，对象是根据类来创建的。声明类的对象，就像声明基本类型的变量一样。下面的语句声明了类 Box 的两个对象：

Box Box1; // 声明 Box1，类型为 Box

Box Box2; // 声明 Box2，类型为 Box

对象 Box1 和 Box2 都有它们各自的数据成员。

## 访问数据成员

类的对象的公共数据成员可以使用直接成员访问运算符 (.) 来访问。为了更好地理解这些概念，让我们尝试一下下面的实例：

## 实例

#include <iostream>

using namespace std;

class Box {

public: double length; // 长度

double breadth; // 宽度

double height; // 高度

};

int main( ) {

Box Box1; // 声明 Box1，类型为 Box

Box Box2; // 声明 Box2，类型为 Box

double volume = 0.0; // 用于存储体积 // box 1 详述

Box1.height = 5.0;

Box1.length = 6.0;

Box1.breadth = 7.0;

// box 2 详述

Box2.height = 10.0;

Box2.length = 12.0;

Box2.breadth = 13.0;

// box 1 的体积

volume = Box1.height \* Box1.length \* Box1.breadth;

cout << "Box1 的体积：" << volume <<endl;

// box 2 的体积

volume = Box2.height \* Box2.length \* Box2.breadth;

cout << "Box2 的体积：" << volume <<endl; return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

Box1 的体积：210

Box2 的体积：1560

需要注意的是，私有的成员和受保护的成员不能使用直接成员访问运算符 (.) 来直接访问。我们将在后续的教程中学习如何访问私有成员和受保护的成员。

# C++ 类成员函数

类的成员函数是指那些把定义和原型写在类定义内部的函数，就像类定义中的其他变量一样。类成员函数是类的一个成员，它可以操作类的任意对象，可以访问对象中的所有成员。

让我们看看之前定义的类 Box，现在我们要使用成员函数来访问类的成员，而不是直接访问这些类的成员：

class Box {

public: double length; // 长度

double breadth; // 宽度

double height; // 高度

double getVolume(void);// 返回体积

};

成员函数可以定义在类定义内部，或者单独使用**范围解析运算符 ::** 来定义。在类定义中定义的成员函数把函数声明为**内联**的，即便没有使用 inline 标识符。所以您可以按照如下方式定义 **Volume()** 函数：

class Box {

public: double length; // 长度

double breadth; // 宽度

double height; // 高度

double getVolume(void) {

return length \* breadth \* height;

}

};

您也可以在类的外部使用**范围解析运算符 ::** 定义该函数，如下所示：

double Box::getVolume(void) {

return length \* breadth \* height;

}

在这里，需要强调一点，在 :: 运算符之前必须使用类名。调用成员函数是在对象上使用点运算符（**.**），这样它就能操作与该对象相关的数据，如下所示：

Box myBox; // 创建一个对象

myBox.getVolume(); // 调用该对象的成员函数

让我们使用上面提到的概念来设置和获取类中不同的成员的值：

## 实例

#include <iostream>

using namespace std;

class Box {

public: double length; // 长度

double breadth; // 宽度

double height; // 高度

// 成员函数声明

double getVolume(void);

void setLength( double len );

void setBreadth( double bre );

void setHeight( double hei );

};

// 成员函数定义

double Box::getVolume(void) {

return length \* breadth \* height;

}

void Box::setLength( double len ) {

length = len;

}

void Box::setBreadth( double bre ) {

breadth = bre;

}

void Box::setHeight( double hei ) {

height = hei;

}

// 程序的主函数

int main( ) {

Box Box1; // 声明 Box1，类型为 Box

Box Box2; // 声明 Box2，类型为 Box

double volume = 0.0; // 用于存储体积 // box 1 详述

Box1.setLength(6.0);

Box1.setBreadth(7.0);

Box1.setHeight(5.0);

// box 2 详述

Box2.setLength(12.0); Box2.setBreadth(13.0); Box2.setHeight(10.0);

// box 1 的体积 volume = Box1.getVolume(); cout << "Box1 的体积：" << volume <<endl;

// box 2 的体积 volume = Box2.getVolume(); cout << "Box2 的体积：" << volume <<endl; return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

Box1 的体积： 210

Box2 的体积： 1560

# C++ 类访问修饰符

数据封装是面向对象编程的一个重要特点，它防止函数直接访问类类型的内部成员。类成员的访问限制是通过在类主体内部对各个区域标记 **public、private、protected** 来指定的。关键字 **public、private、protected** 称为访问修饰符。

一个类可以有多个 public、protected 或 private 标记区域。每个标记区域在下一个标记区域开始之前或者在遇到类主体结束右括号之前都是有效的。成员和类的默认访问修饰符是 private。

class Base { public: // 公有成员 protected: // 受保护成员 private: // 私有成员 };

## 公有（public）成员

**公有**成员在程序中类的外部是可访问的。您可以不使用任何成员函数来设置和获取公有变量的值，如下所示：

## 实例

#include <iostream>

using namespace std;

class Line {

public: double length;

void setLength( double len );

double getLength( void ); }; // 成员函数定义 double Line::getLength(void) { return length ; } void Line::setLength( double len ) { length = len; } // 程序的主函数 int main( ) { Line line; // 设置长度 line.setLength(6.0); cout << "Length of line : " << line.getLength() <<endl; // 不使用成员函数设置长度 line.length = 10.0; // OK: 因为 length 是公有的 cout << "Length of line : " << line.length <<endl; return 0; }

当上面的代码被编译和执行时，它会产生下列结果：

Length of line : 6

Length of line : 10

## 私有（private）成员

**私有**成员变量或函数在类的外部是不可访问的，甚至是不可查看的。只有类和友元函数可以访问私有成员。

默认情况下，类的所有成员都是私有的。例如在下面的类中，**width** 是一个私有成员，这意味着，如果您没有使用任何访问修饰符，类的成员将被假定为私有成员：

## 实例

class Box { double width; public: double length; void setWidth( double wid ); double getWidth( void ); };

实际操作中，我们一般会在私有区域定义数据，在公有区域定义相关的函数，以便在类的外部也可以调用这些函数，如下所示：

## 实例

#include <iostream> using namespace std;

class Box { public: double length; void setWidth( double wid ); double getWidth( void ); private: double width; };

// 成员函数定义 double Box::getWidth(void) { return width ; } void Box::setWidth( double wid ) { width = wid; } // 程序的主函数

int main( ) { Box box; // 不使用成员函数设置长度 box.length = 10.0; // OK: 因为 length 是公有的 cout << "Length of box : " << box.length <<endl; // 不使用成员函数设置宽度 // box.width = 10.0; // Error: 因为 width 是私有的 box.setWidth(10.0); // 使用成员函数设置宽度 cout << "Width of box : " << box.getWidth() <<endl; return 0; }

当上面的代码被编译和执行时，它会产生下列结果：

Length of box : 10

Width of box : 10

## 保护（protected）成员

**保护**成员变量或函数与私有成员十分相似，但有一点不同，保护成员在派生类（即子类）中是可访问的。

在下一个章节中，您将学习到派生类和继承的知识。现在您可以看到下面的实例中，我们从父类 **Box** 派生了一个子类 **smallBox**。

下面的实例与前面的实例类似，在这里 **width** 成员可被派生类 smallBox 的任何成员函数访问。

## 实例

#include <iostream>

using namespace std;

class Box {

protected: double width;

};

class SmallBox:Box // SmallBox 是派生类 {

public: void setSmallWidth( double wid );

double getSmallWidth( void );

};

// 子类的成员函数

double SmallBox::getSmallWidth(void) { return width ; }

void SmallBox::setSmallWidth( double wid ) { width = wid; } // 程序的主函数

int main( ) {

SmallBox box; // 使用成员函数设置宽度

box.setSmallWidth(5.0);

cout << "Width of box : "<< box.getSmallWidth() << endl; return 0; }

当上面的代码被编译和执行时，它会产生下列结果：

Width of box : 5

## 继承中的特点

有public, protected, private三种继承方式，它们相应地改变了基类成员的访问属性。

* 1.**public 继承：**基类 public 成员，protected 成员，private 成员的访问属性在派生类中分别变成：public, protected, private
* 2.**protected 继承：**基类 public 成员，protected 成员，private 成员的访问属性在派生类中分别变成：protected, protected, private
* 3.**private 继承：**基类 public 成员，protected 成员，private 成员的访问属性在派生类中分别变成：private, private, private

但无论哪种继承方式，上面两点都没有改变：

* 1.private 成员只能被本类成员（类内）和友元访问，不能被派生类访问；
* 2.protected 成员可以被派生类访问。

### public 继承

## 实例

#include<iostream> #include<assert.h>

using namespace std;

class A{

public: int a;

A(){ a1 = 1; a2 = 2; a3 = 3; a = 4; }

void fun(){ cout << a << endl; //正确 cout << a1 << endl; //正确 cout << a2 << endl; //正确 cout << a3 << endl; //正确 } public: int a1; protected: int a2; private: int a3; }; class B : public A{ public: int a; B(int i){ A(); a = i; } void fun(){ cout << a << endl; //正确，public成员 cout << a1 << endl; //正确，基类的public成员，在派生类中仍是public成员。 cout << a2 << endl; //正确，基类的protected成员，在派生类中仍是protected可以被派生类访问。 cout << a3 << endl; //错误，基类的private成员不能被派生类访问。 } }; int main(){ B b(10); cout << b.a << endl; cout << b.a1 << endl; //正确 cout << b.a2 << endl; //错误，类外不能访问protected成员 cout << b.a3 << endl; //错误，类外不能访问private成员 system("pause"); return 0; }

### protected 继承

## 实例

#include<iostream> #include<assert.h> using namespace std; class A{ public: int a; A(){ a1 = 1; a2 = 2; a3 = 3; a = 4; } void fun(){ cout << a << endl; //正确 cout << a1 << endl; //正确 cout << a2 << endl; //正确 cout << a3 << endl; //正确 } public: int a1; protected: int a2; private: int a3; }; class B : protected A{ public: int a; B(int i){ A(); a = i; } void fun(){ cout << a << endl; //正确，public成员。 cout << a1 << endl; //正确，基类的public成员，在派生类中变成了protected，可以被派生类访问。 cout << a2 << endl; //正确，基类的protected成员，在派生类中还是protected，可以被派生类访问。 cout << a3 << endl; //错误，基类的private成员不能被派生类访问。 } }; int main(){ B b(10); cout << b.a << endl; //正确。public成员 cout << b.a1 << endl; //错误，protected成员不能在类外访问。 cout << b.a2 << endl; //错误，protected成员不能在类外访问。 cout << b.a3 << endl; //错误，private成员不能在类外访问。 system("pause"); return 0; }

### private 继承

## 实例

#include<iostream> #include<assert.h> using namespace std; class A{ public: int a; A(){ a1 = 1; a2 = 2; a3 = 3; a = 4; } void fun(){ cout << a << endl; //正确 cout << a1 << endl; //正确 cout << a2 << endl; //正确 cout << a3 << endl; //正确 } public: int a1; protected: int a2; private: int a3; }; class B : private A{ public: int a; B(int i){ A(); a = i; } void fun(){ cout << a << endl; //正确，public成员。 cout << a1 << endl; //正确，基类public成员,在派生类中变成了private,可以被派生类访问。 cout << a2 << endl; //正确，基类的protected成员，在派生类中变成了private,可以被派生类访问。 cout << a3 << endl; //错误，基类的private成员不能被派生类访问。 } }; int main(){ B b(10); cout << b.a << endl; //正确。public成员 cout << b.a1 << endl; //错误，private成员不能在类外访问。 cout << b.a2 << endl; //错误, private成员不能在类外访问。 cout << b.a3 << endl; //错误，private成员不能在类外访问。 system("pause"); return 0; }

# C++ 类构造函数 & 析构函数

## 类的构造函数

类的**构造函数**是类的一种特殊的成员函数，它会在每次创建类的新对象时执行。

构造函数的名称与类的名称是完全相同的，并且不会返回任何类型，也不会返回 void。构造函数可用于为某些成员变量设置初始值。

下面的实例有助于更好地理解构造函数的概念：

## 实例

#include <iostream> using namespace std; class Line { public: void setLength( double len ); double getLength( void ); Line(); // 这是构造函数 private: double length; }; // 成员函数定义，包括构造函数 Line::Line(void) { cout << "Object is being created" << endl; } void Line::setLength( double len ) { length = len; } double Line::getLength( void ) { return length; } // 程序的主函数 int main( ) { Line line; // 设置长度 line.setLength(6.0); cout << "Length of line : " << line.getLength() <<endl; return 0; }

当上面的代码被编译和执行时，它会产生下列结果：

Object is being created

Length of line : 6

## 带参数的构造函数

默认的构造函数没有任何参数，但如果需要，构造函数也可以带有参数。这样在创建对象时就会给对象赋初始值，如下面的例子所示：

## 实例

#include <iostream> using namespace std; class Line { public: void setLength( double len ); double getLength( void ); Line(double len); // 这是构造函数 private: double length; }; // 成员函数定义，包括构造函数 Line::Line( double len) { cout << "Object is being created, length = " << len << endl; length = len; } void Line::setLength( double len ) { length = len; } double Line::getLength( void ) { return length; } // 程序的主函数 int main( ) { Line line(10.0); // 获取默认设置的长度 cout << "Length of line : " << line.getLength() <<endl; // 再次设置长度 line.setLength(6.0); cout << "Length of line : " << line.getLength() <<endl; return 0; }

当上面的代码被编译和执行时，它会产生下列结果：

Object is being created, length = 10

Length of line : 10

Length of line : 6

## 使用初始化列表来初始化字段

使用初始化列表来初始化字段：

Line::Line( double len): length(len) { cout << "Object is being created, length = " << len << endl; }

上面的语法等同于如下语法：

Line::Line( double len) { length = len; cout << "Object is being created, length = " << len << endl; }

假设有一个类 C，具有多个字段 X、Y、Z 等需要进行初始化，同理地，您可以使用上面的语法，只需要在不同的字段使用逗号进行分隔，如下所示：

C::C( double a, double b, double c): X(a), Y(b), Z(c) { .... }

## 类的析构函数

类的**析构函数**是类的一种特殊的成员函数，它会在每次删除所创建的对象时执行。

析构函数的名称与类的名称是完全相同的，只是在前面加了个波浪号（~）作为前缀，它不会返回任何值，也不能带有任何参数。析构函数有助于在跳出程序（比如关闭文件、释放内存等）前释放资源。

下面的实例有助于更好地理解析构函数的概念：

## 实例

#include <iostream> using namespace std; class Line { public: void setLength( double len ); double getLength( void ); Line(); // 这是构造函数声明 ~Line(); // 这是析构函数声明 private: double length; }; // 成员函数定义，包括构造函数 Line::Line(void) { cout << "Object is being created" << endl; } Line::~Line(void) { cout << "Object is being deleted" << endl; } void Line::setLength( double len ) { length = len; } double Line::getLength( void ) { return length; } // 程序的主函数 int main( ) { Line line; // 设置长度 line.setLength(6.0); cout << "Length of line : " << line.getLength() <<endl; return 0; }

当上面的代码被编译和执行时，它会产生下列结果：

Object is being created

Length of line : 6

Object is being deleted

# C++ 拷贝构造函数

**拷贝构造函数**是一种特殊的构造函数，它在创建对象时，是使用同一类中之前创建的对象来初始化新创建的对象。拷贝构造函数通常用于：

* 通过使用另一个同类型的对象来初始化新创建的对象。
* 复制对象把它作为参数传递给函数。
* 复制对象，并从函数返回这个对象。

如果在类中没有定义拷贝构造函数，编译器会自行定义一个。如果类带有指针变量，并有动态内存分配，则它必须有一个拷贝构造函数。拷贝构造函数的最常见形式如下：

classname (const classname &obj) { // 构造函数的主体 }

在这里，**obj** 是一个对象引用，该对象是用于初始化另一个对象的。

## 实例

#include <iostream> using namespace std; class Line { public: int getLength( void ); Line( int len ); // 简单的构造函数 Line( const Line &obj); // 拷贝构造函数 ~Line(); // 析构函数 private: int \*ptr; }; // 成员函数定义，包括构造函数 Line::Line(int len) { cout << "调用构造函数" << endl; // 为指针分配内存 ptr = new int; \*ptr = len; } Line::Line(const Line &obj) { cout << "调用拷贝构造函数并为指针 ptr 分配内存" << endl; ptr = new int; \*ptr = \*obj.ptr; // 拷贝值 } Line::~Line(void) { cout << "释放内存" << endl; delete ptr; } int Line::getLength( void ) { return \*ptr; } void display(Line obj) { cout << "line 大小 : " << obj.getLength() <<endl; } // 程序的主函数 int main( ) { Line line(10); display(line); return 0; }

当上面的代码被编译和执行时，它会产生下列结果：

调用构造函数

调用拷贝构造函数并为指针 ptr 分配内存

line 大小 : 10

释放内存

释放内存

下面的实例对上面的实例稍作修改，通过使用已有的同类型的对象来初始化新创建的对象：

## 实例

#include <iostream> using namespace std; class Line { public: int getLength( void ); Line( int len ); // 简单的构造函数 Line( const Line &obj); // 拷贝构造函数 ~Line(); // 析构函数 private: int \*ptr; }; // 成员函数定义，包括构造函数 Line::Line(int len) { cout << "调用构造函数" << endl; // 为指针分配内存 ptr = new int; \*ptr = len; } Line::Line(const Line &obj) { cout << "调用拷贝构造函数并为指针 ptr 分配内存" << endl; ptr = new int; \*ptr = \*obj.ptr; // 拷贝值 } Line::~Line(void) { cout << "释放内存" << endl; delete ptr; } int Line::getLength( void ) { return \*ptr; } void display(Line obj) { cout << "line 大小 : " << obj.getLength() <<endl; } // 程序的主函数 int main( ) { Line line1(10); Line line2 = line1; // 这里也调用了拷贝构造函数 display(line1); display(line2); return 0; }

当上面的代码被编译和执行时，它会产生下列结果：

调用构造函数

调用拷贝构造函数并为指针 ptr 分配内存

调用拷贝构造函数并为指针 ptr 分配内存

line 大小 : 10

释放内存

调用拷贝构造函数并为指针 ptr 分配内存

line 大小 : 10

释放内存

释放内存

释放内存

# C++ 友元函数

类的友元函数是定义在类外部，但有权访问类的所有私有（private）成员和保护（protected）成员。尽管友元函数的原型有在类的定义中出现过，但是友元函数并不是成员函数。

友元可以是一个函数，该函数被称为友元函数；友元也可以是一个类，该类被称为友元类，在这种情况下，整个类及其所有成员都是友元。

如果要声明函数为一个类的友元，需要在类定义中该函数原型前使用关键字 **friend**，如下所示：

class Box

{

double width;

public:

double length;

friend void printWidth( Box box );

void setWidth( double wid );

};

声明类 ClassTwo 的所有成员函数作为类 ClassOne 的友元，需要在类 ClassOne 的定义中放置如下声明：

friend class ClassTwo;

请看下面的程序：

#include <iostream>

using namespace std;

class Box

{

double width;

public:

friend void printWidth( Box box );

void setWidth( double wid );

};

// 成员函数定义

void Box::setWidth( double wid )

{

width = wid;

}

// 请注意：printWidth() 不是任何类的成员函数

void printWidth( Box box )

{

/\* 因为 printWidth() 是 Box 的友元，它可以直接访问该类的任何成员 \*/

cout << "Width of box : " << box.width <<endl;

}

// 程序的主函数

int main( )

{

Box box;

// 使用成员函数设置宽度

box.setWidth(10.0);

// 使用友元函数输出宽度

printWidth( box );

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

Width of box : 10

# C++ 内联函数

C++ **内联函数**是通常与类一起使用。如果一个函数是内联的，那么在编译时，编译器会把该函数的代码副本放置在每个调用该函数的地方。

对内联函数进行任何修改，都需要重新编译函数的所有客户端，因为编译器需要重新更换一次所有的代码，否则将会继续使用旧的函数。

如果想把一个函数定义为内联函数，则需要在函数名前面放置关键字 **inline**，在调用函数之前需要对函数进行定义。如果已定义的函数多于一行，编译器会忽略 inline 限定符。

在类定义中的定义的函数都是内联函数，即使没有使用 **inline** 说明符。

下面是一个实例，使用内联函数来返回两个数中的最大值：

#include <iostream>

using namespace std;

inline int Max(int x, int y)

{

return (x > y)? x : y;

}

// 程序的主函数

int main( )

{

cout << "Max (20,10): " << Max(20,10) << endl;

cout << "Max (0,200): " << Max(0,200) << endl;

cout << "Max (100,1010): " << Max(100,1010) << endl;

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

Max (20,10): 20

Max (0,200): 200

Max (100,1010): 1010

# C++ this 指针

在 C++ 中，每一个对象都能通过 **this** 指针来访问自己的地址。**this** 指针是所有成员函数的隐含参数。因此，在成员函数内部，它可以用来指向调用对象。

友元函数没有 **this** 指针，因为友元不是类的成员。只有成员函数才有 **this** 指针。

下面的实例有助于更好地理解 this 指针的概念：

## 实例

#include <iostream> using namespace std; class Box { public: // 构造函数定义 Box(double l=2.0, double b=2.0, double h=2.0) { cout <<"Constructor called." << endl; length = l; breadth = b; height = h; } double Volume() { return length \* breadth \* height; } int compare(Box box) { return this->Volume() > box.Volume(); } private: double length; // Length of a box double breadth; // Breadth of a box double height; // Height of a box }; int main(void) { Box Box1(3.3, 1.2, 1.5); // Declare box1 Box Box2(8.5, 6.0, 2.0); // Declare box2 if(Box1.compare(Box2)) { cout << "Box2 is smaller than Box1" <<endl; } else { cout << "Box2 is equal to or larger than Box1" <<endl; } return 0; }

当上面的代码被编译和执行时，它会产生下列结果：

Constructor called.

Constructor called.

Box2 is equal to or larger than Box1

# C++ 指向类的指针

一个指向 C++ 类的指针与指向结构的指针类似，访问指向类的指针的成员，需要使用成员访问运算符 **->**，就像访问指向结构的指针一样。与所有的指针一样，您必须在使用指针之前，对指针进行初始化。

下面的实例有助于更好地理解指向类的指针的概念：

#include <iostream>

using namespace std;

class Box

{

public:

// 构造函数定义

Box(double l=2.0, double b=2.0, double h=2.0)

{

cout <<"Constructor called." << endl;

length = l;

breadth = b;

height = h;

}

double Volume()

{

return length \* breadth \* height;

}

private:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

int main(void)

{

Box Box1(3.3, 1.2, 1.5); // Declare box1

Box Box2(8.5, 6.0, 2.0); // Declare box2

Box \*ptrBox; // Declare pointer to a class.

// 保存第一个对象的地址

ptrBox = &Box1;

// 现在尝试使用成员访问运算符来访问成员

cout << "Volume of Box1: " << ptrBox->Volume() << endl;

// 保存第二个对象的地址

ptrBox = &Box2;

// 现在尝试使用成员访问运算符来访问成员

cout << "Volume of Box2: " << ptrBox->Volume() << endl;

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

Constructor called.

Constructor called.

Volume of Box1: 5.94

Volume of Box2: 102

# C++ 类的静态成员

我们可以使用 **static** 关键字来把类成员定义为静态的。当我们声明类的成员为静态时，这意味着无论创建多少个类的对象，静态成员都只有一个副本。

静态成员在类的所有对象中是共享的。如果不存在其他的初始化语句，在创建第一个对象时，所有的静态数据都会被初始化为零。我们不能把静态成员的初始化放置在类的定义中，但是可以在类的外部通过使用范围解析运算符 **::** 来重新声明静态变量从而对它进行初始化，如下面的实例所示。

下面的实例有助于更好地理解静态成员数据的概念：

## 实例

#include <iostream> using namespace std; class Box { public: static int objectCount; // 构造函数定义 Box(double l=2.0, double b=2.0, double h=2.0) { cout <<"Constructor called." << endl; length = l; breadth = b; height = h; // 每次创建对象时增加 1 objectCount++; } double Volume() { return length \* breadth \* height; } private: double length; // 长度 double breadth; // 宽度 double height; // 高度 }; // 初始化类 Box 的静态成员 int Box::objectCount = 0; int main(void) { Box Box1(3.3, 1.2, 1.5); // 声明 box1 Box Box2(8.5, 6.0, 2.0); // 声明 box2 // 输出对象的总数 cout << "Total objects: " << Box::objectCount << endl; return 0; }

当上面的代码被编译和执行时，它会产生下列结果：

Constructor called.

Constructor called.

Total objects: 2

## 静态成员函数

如果把函数成员声明为静态的，就可以把函数与类的任何特定对象独立开来。静态成员函数即使在类对象不存在的情况下也能被调用，**静态函数**只要使用类名加范围解析运算符 **::** 就可以访问。

静态成员函数只能访问静态成员数据、其他静态成员函数和类外部的其他函数。

静态成员函数有一个类范围，他们不能访问类的 this 指针。您可以使用静态成员函数来判断类的某些对象是否已被创建。

***静态成员函数与普通成员函数的区别：***

* 静态成员函数没有 this 指针，只能访问静态成员（包括静态成员变量和静态成员函数）。
* 普通成员函数有 this 指针，可以访问类中的任意成员；而静态成员函数没有 this 指针。

下面的实例有助于更好地理解静态成员函数的概念：

## 实例

#include <iostream> using namespace std; class Box { public: static int objectCount; // 构造函数定义 Box(double l=2.0, double b=2.0, double h=2.0) { cout <<"Constructor called." << endl; length = l; breadth = b; height = h; // 每次创建对象时增加 1 objectCount++; } double Volume() { return length \* breadth \* height; } static int getCount() { return objectCount; } private: double length; // 长度 double breadth; // 宽度 double height; // 高度 }; // 初始化类 Box 的静态成员 int Box::objectCount = 0; int main(void) { // 在创建对象之前输出对象的总数 cout << "Inital Stage Count: " << Box::getCount() << endl; Box Box1(3.3, 1.2, 1.5); // 声明 box1 Box Box2(8.5, 6.0, 2.0); // 声明 box2 // 在创建对象之后输出对象的总数 cout << "Final Stage Count: " << Box::getCount() << endl; return 0; }

当上面的代码被编译和执行时，它会产生下列结果：

Inital Stage Count: 0

Constructor called.

Constructor called.

Final Stage Count: 2

# C++ 继承

面向对象程序设计中最重要的一个概念是继承。继承允许我们依据另一个类来定义一个类，这使得创建和维护一个应用程序变得更容易。这样做，也达到了重用代码功能和提高执行时间的效果。

当创建一个类时，您不需要重新编写新的数据成员和成员函数，只需指定新建的类继承了一个已有的类的成员即可。这个已有的类称为**基类**，新建的类称为**派生类**。

继承代表了 **is a** 关系。例如，哺乳动物是动物，狗是哺乳动物，因此，狗是动物，等等。

## 基类 & 派生类

一个类可以派生自多个类，这意味着，它可以从多个基类继承数据和函数。定义一个派生类，我们使用一个类派生列表来指定基类。类派生列表以一个或多个基类命名，形式如下：

class derived-class: access-specifier base-class

其中，访问修饰符 access-specifier 是 **public、protected** 或 **private** 其中的一个，base-class 是之前定义过的某个类的名称。如果未使用访问修饰符 access-specifier，则默认为 private。

假设有一个基类 **Shape**，**Rectangle** 是它的派生类，如下所示：

## 实例

#include <iostream> using namespace std; // 基类 class Shape { public: void setWidth(int w) { width = w; } void setHeight(int h) { height = h; } protected: int width; int height; }; // 派生类 class Rectangle: public Shape { public: int getArea() { return (width \* height); } }; int main(void) { Rectangle Rect; Rect.setWidth(5); Rect.setHeight(7); // 输出对象的面积 cout << "Total area: " << Rect.getArea() << endl; return 0; }

当上面的代码被编译和执行时，它会产生下列结果：

Total area: 35

## 访问控制和继承

派生类可以访问基类中所有的非私有成员。因此基类成员如果不想被派生类的成员函数访问，则应在基类中声明为 private。

我们可以根据访问权限总结出不同的访问类型，如下所示：

|  |  |  |  |
| --- | --- | --- | --- |
| **访问** | **public** | **protected** | **private** |
| 同一个类 | yes | yes | yes |
| 派生类 | yes | yes | no |
| 外部的类 | yes | no | no |

一个派生类继承了所有的基类方法，但下列情况除外：

* 基类的构造函数、析构函数和拷贝构造函数。
* 基类的重载运算符。
* 基类的友元函数。

## 继承类型

当一个类派生自基类，该基类可以被继承为 **public、protected** 或 **private** 几种类型。继承类型是通过上面讲解的访问修饰符 access-specifier 来指定的。

我们几乎不使用 **protected** 或 **private** 继承，通常使用 **public** 继承。当使用不同类型的继承时，遵循以下几个规则：

* **公有继承（public）：**当一个类派生自**公有**基类时，基类的**公有**成员也是派生类的**公有**成员，基类的**保护**成员也是派生类的**保护**成员，基类的**私有**成员不能直接被派生类访问，但是可以通过调用基类的**公有**和**保护**成员来访问。
* **保护继承（protected）：** 当一个类派生自**保护**基类时，基类的**公有**和**保护**成员将成为派生类的**保护**成员。
* **私有继承（private）：**当一个类派生自**私有**基类时，基类的**公有**和**保护**成员将成为派生类的**私有**成员。

## 多继承

多继承即一个子类可以有多个父类，它继承了多个父类的特性。

C++ 类可以从多个类继承成员，语法如下：

class <派生类名>:<继承方式1><基类名1>,<继承方式2><基类名2>,…

{

<派生类类体>

};

其中，访问修饰符继承方式是 **public、protected** 或 **private** 其中的一个，用来修饰每个基类，各个基类之间用逗号分隔，如上所示。现在让我们一起看看下面的实例：

## 实例

#include <iostream> using namespace std; // 基类 Shape class Shape { public: void setWidth(int w) { width = w; } void setHeight(int h) { height = h; } protected: int width; int height; }; // 基类 PaintCost class PaintCost { public: int getCost(int area) { return area \* 70; } }; // 派生类 class Rectangle: public Shape, public PaintCost { public: int getArea() { return (width \* height); } }; int main(void) { Rectangle Rect; int area; Rect.setWidth(5); Rect.setHeight(7); area = Rect.getArea(); // 输出对象的面积 cout << "Total area: " << Rect.getArea() << endl; // 输出总花费 cout << "Total paint cost: $" << Rect.getCost(area) << endl; return 0; }

当上面的代码被编译和执行时，它会产生下列结果：

Total area: 35

Total paint cost: $2450

# C++ 重载运算符和重载函数

C++ 允许在同一作用域中的某个**函数**和**运算符**指定多个定义，分别称为**函数重载**和**运算符重载**。

重载声明是指一个与之前已经在该作用域内声明过的函数或方法具有相同名称的声明，但是它们的参数列表和定义（实现）不相同。

当您调用一个**重载函数**或**重载运算符**时，编译器通过把您所使用的参数类型与定义中的参数类型进行比较，决定选用最合适的定义。选择最合适的重载函数或重载运算符的过程，称为**重载决策**。

## C++ 中的函数重载

在同一个作用域内，可以声明几个功能类似的同名函数，但是这些同名函数的形式参数（指参数的个数、类型或者顺序）必须不同。您不能仅通过返回类型的不同来重载函数。

下面的实例中，同名函数 **print()** 被用于输出不同的数据类型：

## 实例

#include <iostream> using namespace std; class printData { public: void print(int i) { cout << "整数为: " << i << endl; } void print(double f) { cout << "浮点数为: " << f << endl; } void print(char c[]) { cout << "字符串为: " << c << endl; } }; int main(void) { printData pd; // 输出整数 pd.print(5); // 输出浮点数 pd.print(500.263); // 输出字符串 char c[] = "Hello C++"; pd.print(c); return 0; }

当上面的代码被编译和执行时，它会产生下列结果：

整数为: 5

浮点数为: 500.263

字符串为: Hello C++

## C++ 中的运算符重载

您可以重定义或重载大部分 C++ 内置的运算符。这样，您就能使用自定义类型的运算符。

重载的运算符是带有特殊名称的函数，函数名是由关键字 operator 和其后要重载的运算符符号构成的。与其他函数一样，重载运算符有一个返回类型和一个参数列表。

Box operator+(const Box&);

声明加法运算符用于把两个 Box 对象相加，返回最终的 Box 对象。大多数的重载运算符可被定义为普通的非成员函数或者被定义为类成员函数。如果我们定义上面的函数为类的非成员函数，那么我们需要为每次操作传递两个参数，如下所示：

Box operator+(const Box&, const Box&);

下面的实例使用成员函数演示了运算符重载的概念。在这里，对象作为参数进行传递，对象的属性使用 **this** 运算符进行访问，如下所示：

## 实例

#include <iostream> using namespace std; class Box { public: double getVolume(void) { return length \* breadth \* height; } void setLength( double len ) { length = len; } void setBreadth( double bre ) { breadth = bre; } void setHeight( double hei ) { height = hei; } // 重载 + 运算符，用于把两个 Box 对象相加 Box operator+(const Box& b) { Box box; box.length = this->length + b.length; box.breadth = this->breadth + b.breadth; box.height = this->height + b.height; return box; } private: double length; // 长度 double breadth; // 宽度 double height; // 高度 }; // 程序的主函数 int main( ) { Box Box1; // 声明 Box1，类型为 Box Box Box2; // 声明 Box2，类型为 Box Box Box3; // 声明 Box3，类型为 Box double volume = 0.0; // 把体积存储在该变量中 // Box1 详述 Box1.setLength(6.0); Box1.setBreadth(7.0); Box1.setHeight(5.0); // Box2 详述 Box2.setLength(12.0); Box2.setBreadth(13.0); Box2.setHeight(10.0); // Box1 的体积 volume = Box1.getVolume(); cout << "Volume of Box1 : " << volume <<endl; // Box2 的体积 volume = Box2.getVolume(); cout << "Volume of Box2 : " << volume <<endl; // 把两个对象相加，得到 Box3 Box3 = Box1 + Box2; // Box3 的体积 volume = Box3.getVolume(); cout << "Volume of Box3 : " << volume <<endl; return 0; }

当上面的代码被编译和执行时，它会产生下列结果：

Volume of Box1 : 210

Volume of Box2 : 1560

Volume of Box3 : 5400

## 可重载运算符/不可重载运算符

下面是可重载的运算符列表：

|  |  |
| --- | --- |
| 双目算术运算符 | + (加)，-(减)，\*(乘)，/(除)，% (取模) |
| 关系运算符 | ==(等于)，!= (不等于)，< (小于)，> (大于>，<=(小于等于)，>=(大于等于) |
| 逻辑运算符 | ||(逻辑或)，&&(逻辑与)，!(逻辑非) |
| 单目运算符 | + (正)，-(负)，\*(指针)，&(取地址) |
| 自增自减运算符 | ++(自增)，--(自减) |
| 位运算符 | | (按位或)，& (按位与)，~(按位取反)，^(按位异或),，<< (左移)，>>(右移) |
| 赋值运算符 | =, +=, -=, \*=, /= , % = , &=, |=, ^=, <<=, >>= |
| 空间申请与释放 | new, delete, new[ ] , delete[] |
| 其他运算符 | **()**(函数调用)，**->**(成员访问)，**,**(逗号)，**[]**(下标) |

下面是不可重载的运算符列表：

* **.**：成员访问运算符
* **.\***, **->\***：成员指针访问运算符
* **::**：域运算符
* **sizeof**：长度运算符
* **?:**：条件运算符
* **#**： 预处理符号

# C++ 一元运算符重载

一元运算符只对一个操作数进行操作，下面是一元运算符的实例：

* [递增运算符（ ++ ）和递减运算符（ -- ）](https://www.runoob.com/cplusplus/increment-decrement-operators-overloading.html)
* 一元减运算符，即负号（ - ）
* 逻辑非运算符（ ! ）

一元运算符通常出现在它们所操作的对象的左边，比如 !obj、-obj 和 ++obj，但有时它们也可以作为后缀，比如 obj++ 或 obj--。

下面的实例演示了如何重载一元减运算符（ - ）。

## 实例

#include <iostream> using namespace std; class Distance { private: int feet; // 0 到无穷 int inches; // 0 到 12 public: // 所需的构造函数 Distance(){ feet = 0; inches = 0; } Distance(int f, int i){ feet = f; inches = i; } // 显示距离的方法 void displayDistance() { cout << "F: " << feet << " I:" << inches <<endl; } // 重载负运算符（ - ） Distance operator- () { feet = -feet; inches = -inches; return Distance(feet, inches); } }; int main() { Distance D1(11, 10), D2(-5, 11); -D1; // 取相反数 D1.displayDistance(); // 距离 D1 -D2; // 取相反数 D2.displayDistance(); // 距离 D2 return 0; }

当上面的代码被编译和执行时，它会产生下列结果：

F: -11 I:-10

F: 5 I:-11

# C++ 二元运算符重载

二元运算符需要两个参数，下面是二元运算符的实例。我们平常使用的加运算符（ + ）、减运算符（ - ）、乘运算符（ \* ）和除运算符（ / ）都属于二元运算符。就像加(+)运算符。

下面的实例演示了如何重载加运算符（ + ）。类似地，您也可以尝试重载减运算符（ - ）和除运算符（ / ）。

## 实例

#include <iostream> using namespace std; class Box { double length; // 长度 double breadth; // 宽度 double height; // 高度 public: double getVolume(void) { return length \* breadth \* height; } void setLength( double len ) { length = len; } void setBreadth( double bre ) { breadth = bre; } void setHeight( double hei ) { height = hei; } // 重载 + 运算符，用于把两个 Box 对象相加 Box operator+(const Box& b) { Box box; box.length = this->length + b.length; box.breadth = this->breadth + b.breadth; box.height = this->height + b.height; return box; } }; // 程序的主函数 int main( ) { Box Box1; // 声明 Box1，类型为 Box Box Box2; // 声明 Box2，类型为 Box Box Box3; // 声明 Box3，类型为 Box double volume = 0.0; // 把体积存储在该变量中 // Box1 详述 Box1.setLength(6.0); Box1.setBreadth(7.0); Box1.setHeight(5.0); // Box2 详述 Box2.setLength(12.0); Box2.setBreadth(13.0); Box2.setHeight(10.0); // Box1 的体积 volume = Box1.getVolume(); cout << "Volume of Box1 : " << volume <<endl; // Box2 的体积 volume = Box2.getVolume(); cout << "Volume of Box2 : " << volume <<endl; // 把两个对象相加，得到 Box3 Box3 = Box1 + Box2; // Box3 的体积 volume = Box3.getVolume(); cout << "Volume of Box3 : " << volume <<endl; return 0; }

当上面的代码被编译和执行时，它会产生下列结果：

Volume of Box1 : 210

Volume of Box2 : 1560

Volume of Box3 : 5400

# C++ 关系运算符重载

C++ 语言支持各种关系运算符（ < 、 > 、 <= 、 >= 、 == 等等），它们可用于比较 C++ 内置的数据类型。

您可以重载任何一个关系运算符，重载后的关系运算符可用于比较类的对象。

下面的实例演示了如何重载 < 运算符，类似地，您也可以尝试重载其他的关系运算符。

#include <iostream>

using namespace std;

class Distance

{

private:

int feet; // 0 到无穷

int inches; // 0 到 12

public:

// 所需的构造函数

Distance(){

feet = 0;

inches = 0;

}

Distance(int f, int i){

feet = f;

inches = i;

}

// 显示距离的方法

void displayDistance()

{

cout << "F: " << feet << " I:" << inches <<endl;

}

// 重载负运算符（ - ）

Distance operator- ()

{

feet = -feet;

inches = -inches;

return Distance(feet, inches);

}

// 重载小于运算符（ < ）

bool operator <(const Distance& d)

{

if(feet < d.feet)

{

return true;

}

if(feet == d.feet && inches < d.inches)

{

return true;

}

return false;

}

};

int main()

{

Distance D1(11, 10), D2(5, 11);

if( D1 < D2 )

{

cout << "D1 is less than D2 " << endl;

}

else

{

cout << "D2 is less than D1 " << endl;

}

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

D2 is less than D1

# C++ 输入/输出运算符重载

C++ 能够使用流提取运算符 >> 和流插入运算符 << 来输入和输出内置的数据类型。您可以重载流提取运算符和流插入运算符来操作对象等用户自定义的数据类型。

在这里，有一点很重要，我们需要把运算符重载函数声明为类的友元函数，这样我们就能不用创建对象而直接调用函数。

下面的实例演示了如何重载提取运算符 >> 和插入运算符 <<。

## 实例

#include <iostream> using namespace std; class Distance { private: int feet; // 0 到无穷 int inches; // 0 到 12 public: // 所需的构造函数 Distance(){ feet = 0; inches = 0; } Distance(int f, int i){ feet = f; inches = i; } friend ostream &operator<<( ostream &output, const Distance &D ) { output << "F : " << D.feet << " I : " << D.inches; return output; } friend istream &operator>>( istream &input, Distance &D ) { input >> D.feet >> D.inches; return input; } }; int main() { Distance D1(11, 10), D2(5, 11), D3; cout << "Enter the value of object : " << endl; cin >> D3; cout << "First Distance : " << D1 << endl; cout << "Second Distance :" << D2 << endl; cout << "Third Distance :" << D3 << endl; return 0; }

当上面的代码被编译和执行时，它会产生下列结果：

$./a.out

Enter the value of object :

70

10

First Distance : F : 11 I : 10

Second Distance :F : 5 I : 11

Third Distance :F : 70 I : 10

# C++ ++ 和 -- 运算符重载

递增运算符（ ++ ）和递减运算符（ -- ）是 C++ 语言中两个重要的一元运算符。

下面的实例演示了如何重载递增运算符（ ++ ），包括前缀和后缀两种用法。类似地，您也可以尝试重载递减运算符（ -- ）。

## 实例

#include <iostream> using namespace std; class Time { private: int hours; // 0 到 23 int minutes; // 0 到 59 public: // 所需的构造函数 Time(){ hours = 0; minutes = 0; } Time(int h, int m){ hours = h; minutes = m; } // 显示时间的方法 void displayTime() { cout << "H: " << hours << " M:" << minutes <<endl; } // 重载前缀递增运算符（ ++ ） Time operator++ () { ++minutes; // 对象加 1 if(minutes >= 60) { ++hours; minutes -= 60; } return Time(hours, minutes); } // 重载后缀递增运算符（ ++ ） Time operator++( int ) { // 保存原始值 Time T(hours, minutes); // 对象加 1 ++minutes; if(minutes >= 60) { ++hours; minutes -= 60; } // 返回旧的原始值 return T; } }; int main() { Time T1(11, 59), T2(10,40); ++T1; // T1 加 1 T1.displayTime(); // 显示 T1 ++T1; // T1 再加 1 T1.displayTime(); // 显示 T1 T2++; // T2 加 1 T2.displayTime(); // 显示 T2 T2++; // T2 再加 1 T2.displayTime(); // 显示 T2 return 0; }

当上面的代码被编译和执行时，它会产生下列结果：

H: 12 M:0

H: 12 M:1

H: 10 M:41

H: 10 M:42

# C++ 赋值运算符重载

[[C++ 重载运算符和重载函数](https://www.runoob.com/cplusplus/cpp-overloading.html) C++ 重载运算符和重载函数](https://www.runoob.com/cplusplus/cpp-overloading.html)

就像其他运算符一样，您可以重载赋值运算符（ = ），用于创建一个对象，比如拷贝构造函数。

下面的实例演示了如何重载赋值运算符。

## 实例

#include <iostream>

using namespace std;

class Distance { private: int feet; // 0 到无穷 int inches; // 0 到 12 public: // 所需的构造函数 Distance(){ feet = 0; inches = 0; } Distance(int f, int i){ feet = f; inches = i; } void operator=(const Distance &D ) { feet = D.feet; inches = D.inches; } // 显示距离的方法 void displayDistance() { cout << "F: " << feet << " I:" << inches << endl; } }; int main() { Distance D1(11, 10), D2(5, 11); cout << "First Distance : "; D1.displayDistance(); cout << "Second Distance :"; D2.displayDistance(); // 使用赋值运算符 D1 = D2; cout << "First Distance :"; D1.displayDistance(); return 0; }

当上面的代码被编译和执行时，它会产生下列结果：

First Distance : F: 11 I:10

Second Distance :F: 5 I:11

First Distance :F: 5 I:11