C++面向对象

智能指针

智能指针是行为**类似于指针**的**类对象**，使用前要包含头文件**<memory>**

使用常规指针的函数在指针变量消亡前，还需要手动释放指针指向的内存

智能指针作为一个类对象，自带**析构函数**，因此可以**借助析构函数**将**所指内存**释放。

auto_ptr

```
auto_ptr<class X> pointer-name(new X)//声明一个指向X类的指针,并申请一块内存进行构造
```

对于特定的对象，只能有一个**auto_ptr**可以指向它，

当第二个智能指针指向它时，原先的**auto_ptr放弃所有权**，变为**空指针**

unique_ptr

```
unique_ptr<class X> pointer-name(new X)//声明一个指向X类的指针,并申请一块内存进行构造
```

在一个**auto_ptr让出其所有权**后，再用它访问对象会导致错误

unique_ptr与auto_ptr相比，就是**不允许使用这种让出所有权的指针**，**更加安全**,而且还可以**使用new[]**，相比之下auto_ptr和shared_ptr都是不允许的

shared_ptr

```
shared_ptr<class X> pointer-name(new X)//声明一个指向X类的指针,并申请一块内存进行构造
```

shared_ptr**记录其指向对象被引用的次数**，当最后一个指向这个对象的shared_ptr消亡时，才将**这片内存给释放掉**

智能指针的选择

- 使用**多个指向同一个对象**的指针，应该用**shared_ptr**
- **不需要多个指向同一个对象**的指针，应该用**unique_ptr**
- 部分编译器不支持C++11标准，只能用**auto_ptr**

t

类与成员基本概念

类的概念

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，是一种用户**自定义的数据类型**，它是一种**封装了数据和函数**的组合。

类中的数据称为**成员变量**，函数称为**成员函数。**类可以被看作是**一种模板**，可以用来**创建具有相同属性和行为的多个对象。**

类定义

定义一个类需要使用关键字 **class**，然后指定类的名称，并类的主体是包含在一对花括号中，主体包含类的**成员变量**和**成员函数。**

定义一个类，本质上是定义一个数据类型的蓝图，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。

通常**类名首字母会大写**

以下实例我们使用关键字 **class** 定义 Box 数据类型，包含了三个成员变量 length、breadth 和 height：

```
class Box
{
   public:
      double length;   // 盒子的长度
      double breadth;  // 盒子的宽度
      double height;   // 盒子的高度
};
```

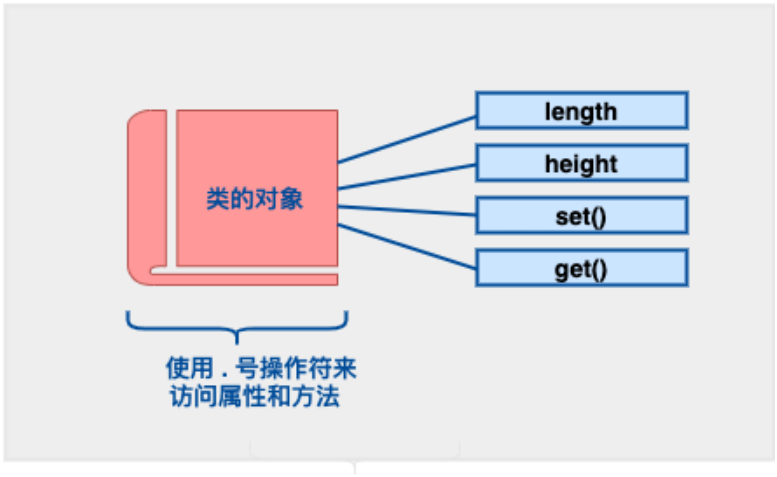
定义 C++ 对象

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 的一个对象：

```
Box Box1;          // 声明 Box1，类型为 Box
```

访问数据成员

类似C中的构造体，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：

```
#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 get(void);
      void set( double len, double bre, double hei );
};
// 成员函数定义
double Box::get(void)
{
    return length * breadth * height;
}

void Box::set( double len, double bre, double hei)
{
    length = len;
```

```
    breadth = bre;
    height = hei;
}
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;     // 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;


   // box 3 详述
   Box3.set(16.0, 8.0, 12.0);
   volume = Box3.get();
   cout << "Box3 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
Box3 的体积：1536
```

成员的访问权限

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。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：

```
#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，如果没有使用任何访问修饰符，类的成员将被假定为私有成员：

```
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：

```
#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

protected（受保护）成员变量或函数与私有成员十分相似，

但有一点不同，protected（受保护）成员在派生类（即子类）中是可访问的。

一般将数据成员声明为私有访问，成员函数可以声明为保护访问，使派生类可以用public无法使用的成员函数

```
#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
```

友元机制

友元函数

友元可以是一个函数，该函数被称为**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。**

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 );
};
```

如果友元函数在另一个类中，可以使用**域解析符**

```
class aaa
{
        void printWidth();
};

class Box
{
   double width;
public:
   double length;
   friend void aaa:: printWidth( );
   void setWidth( double wid );
};
```

友元类

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。**

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;
```

看下面的程序：

```
#include <iostream>

class B; // 前向声明类B

class A {
public:
    void displayB(B& b); // 声明一个成员函数，参数为B类的引用
};

class B {
private:
    int valueB;

public:
```

```
    B() : valueB(0) {}

    friend class A; // 声明A为B的友元类，使A可以访问B的私有成员

    void setValue(int value) {
        valueB = value;
    }
};

void A::displayB(B& b) {
    std::cout << "Value of B: " << b.valueB << std::endl;
}

int main() {
    A a;
    B b;
    b.setValue(42);

    a.displayB(b); // 输出：Value of B: 42

    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
```

友元并**不具有传递性**。A是B的友元，B又是C的友元，如果没有直接声明，A不是C的友元。

前向声明

假设有下面这段代码

```
class Remote
{
...
}

class Tv
{
        friend void Remote::set_chan(Tv& t,int c)
```

```
        ...
}
```

- 如果remote在Tv的前面，由于这个函数中用到了TV类型，在编译器见到这个函数时还没有Tv类型，出现错误
- 如果Tv在remote的前面，remote的方法还没有定义。因此也会出现错误

只要在remote类前先声明Tv，告知编译器Tv是一个存在的类即可,这样的声明叫做**前向声明**

```
class Tv;//前向声明
class Remote
{
...
}

class Tv
{
        friend void Remote::set_chan(Tv& t,int c)
        ...
}
```

如果让整个类成为友元，就不需要前向声明，友元语句已经指出这是一个类

类的成员函数

成员函数描述类中定义的数据成员**所能实施的操作**

成员函数的定义与声明

类内定义成员函数

直接定义。直接在类中定义的函数会被自动定义为**内联函数**

类外定义成员函数

类内需要有**函数的声明**，类外定义函数时要**在函数名前加上对应的类名和域解析符**，如:

```
void date::set(int day,int month)
```

访问对象

创建对象

直接创建

类似**定义变量**的方法，但是用**类定义。**用类定义的变量一般称为对象

间接创建(动态对象)

定义一个指向**某动态对象的指针**，然后用new操作符开辟新的内存空间。

对对象的操作

对对象的操作一般需要通过其类成员**public中的部分**实现。

非动态对象:对象名.类成员

动态对象:对象指针->类成员/(*对象指针).类成员

类外对对象的操作收到**类访问权限的限制**，但如果是**类内的该类对象**，其操作访问**不受类本身访问限制的作用。**

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。

static成员函数也没有**this指针**

下面的实例有助于更好地理解 this 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;
      }
      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
```

当需要返回成员函数所在的**对象或对象的引用**时，可以返回this指针/*this

默认方法与禁用方法

默认方法

如果某个函数在类中不会自动创建，例如在我们定义移动构造函数后默认构造函数就不会被创建，

此时需要**显式声明**这个构造函数需要一个**默认版本**

```
class someclass
{
public:
  someclass(someclass&&);
  someclass()=default;//生成一个someclass的默认版本
  someclass(const someclass&)=default//生成一个复制构造函数的默认版本
```

禁用方法

可以**禁止编译器使用特定方法**，在无意间使用这种方法将会报错

```
class someclass
{
public:
  someclass(someclass&&);
  someclass()=delete;//禁止默认构造此类对象
  void set(double x);
  void set(int x)=delete////告诉编译器不能将int类型的隐式转换
```

构造函数和析构函数

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。**

在没有自定义构造函数的情况下，每个类会提供一个**隐式的默认构造函数**

定义构造函数

例如

```
class line{
        public:
        line(){length=0;}//构造函数的定义
        private:
        double length;
};
```

构造函数的参数**不能与成员同名**，为了区别可以将成员加上前缀"m_".

调用构造函数

创建静态对象

```
A a1;//调用默认构造函数

A a2(1)  A a3(“abcd”)//调用含参数的构造函数

A a[4];//对数组的每个对象都调用默认构造函数

A b[5]={A(),A(1),A(“abcd”),2,”xyz”}//对数组的每个对象调用不同的构造函数,其中2和”xyz”代表调用它们对应的重载构造函数
```

创建动态对象

```
A *p1=new A;//调用默认构造函数

A *p2=new A(2)//调用含参数构造函数

A *p3=new A[20]//创建动态对象数组,创建动态对象数组只能调用各对象的默认构造函数
```

下面的实例有助于更好地理解构造函数的概念：

```
#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
```

若类中含有成员对象，即其他类的成员，在创建对象时**先调用该类成员的构造函数**，再**调用本类的构造函数**

各成员对象的**构造函数调用次序**由它们在**类中声明的顺序决定。**

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：

```
#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);//需要一个初始值

   // 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
```

成员初始化列表

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，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)
{
  ....
}
```

对于**常量数据成员和引用类型初始化**，不能在定义他们时初始化，也不能赋值初始化。必须要通过成员初始化表对这两种数据进行初始化

成员对象初始化列表

当一个类包含其他类对象作为成员变量时，可以使用**成员对象初始化列表**来对这些成员对象进行初始化。格式为：

```
<类名>(<总参数表>):<成员对象1>(<形参表1>), <成员对象2>(<形参表2>), ……
{
        //函数体
}
```

下面是一个示范代码：

```
#include <iostream>

class Engine {
public:
    Engine(int cylinders) : numCylinders(cylinders) {
        std::cout << "Engine constructor called. Cylinders: " << numCylinders << st
d::endl;
    }

private:
    int numCylinders;
};

class Car {
public:
    Car(int year, int cylinders) : manufacturingYear(year), carEngine(cylinders) {
        std::cout << "Car constructor called. Year: " << manufacturingYear << std::
endl;
    }

private:
    int manufacturingYear;
    Engine carEngine;
};

int main() {
    Car myCar(2023, 4);
    return 0;
}
```

在上述示例中，`Car`类包含了一个`Engine`对象作为成员变量。

在`Car`类的构造函数中，使用成员对象初始化列表对`manufacturingYear`和`carEngine`进行初始化。

通过在构造函数参数后使用冒号，然后跟随成员对象的初始化语句，我们可以在构造函数体之前对成员对象进行初始化。

在`main`函数中，创建一个`Car`对象`myCar`时，会调用`Car`类的构造函数。

构造函数首先初始化`manufacturingYear`成员变量为2023，然后初始化`carEngine`成员对象，调用`Engine`类的构造函数，并传递4作为参数。

输出结果将是：

```
Engine constructor called. Cylinders: 4
Car constructor called. Year: 2023
```

这表明在`Car`对象的构造过程中，首先构造`Engine`对象，然后构造`Car`对象，按照成员对象初始化列表中的顺序进行初始化。

使用成员对象初始化列表可以提供更高效的初始化方式，避免了先默认初始化再赋值的过程，同时确保成员对象在构造函数体执行之前已经正确初始化。

拷贝构造函数

可以将同类对象用**于对另一个同类对象初始化**的函数，主要作用即“**拷贝**“。

调用拷贝构造函数的情形：

- 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。

为避免两个对象内的引用变量或指针变量指向同一个变量或空间而造成可能的误改或析构时对同一空间释放两次，

此时需要**自定义拷贝构造函数。**

格式为：

类名 {类名& 变量名};

拷贝构造函数不用引用的话，在调用拷贝构造函数会**先创造一个临时对象**，在赋值给这个临时对象时**又需要**调用拷贝构造函数，从而会陷入死循环。

```
class A{
      int x,y;
      public:
            A(){
                  x=y=0;
            }
            void inc(){x++;y++;
            }
};
class B{
      int z;
      A a;
      public:
            B(){z=0;
            }
            B(const B&amp;amp; b):a(b.a){
                  z=b.z;   //1
            }
            void inc(){
                  z++;a.inc();
            }
};
B b1;

b1.inc();
```

```
B b2(b1);//调用拷贝构造函数
```

隐式拷贝构造函数对成员对象初始化时会调用该成员对象的**拷贝构造函数**，但**自定义拷贝构造函数**对成员对象初始化时会调用它的**默认构造函数**。

为了能让这样的成员对象也被赋值，需要**显式调用该成员对象的拷贝构造函数**。如1所示

转移构造函数

当一个将要消亡的对象赋值给另一个对象时，为了省去收回消亡对象空间、为新对象申请空间的时间，

可以用转移构造函数直接将旧对象的内存空间挪用给新对象。如下：

```
#include <iostream>

class MyClass {
public:
    // Default constructor
    MyClass() {
        data = new int[10];
        for (int i = 0; i < 10; i++) {
            data[i] = i;
        }
    }

    // Move constructor
    MyClass(MyClass&& other) {
        data = other.data;
        other.data = nullptr;//将原对象指针置空，避免二次归还空间。
    }

    // Destructor
    ~MyClass() {
        delete[] data;
    }

private:
    int* data;
};

int main() {
    // Create a new instance of MyClass
    MyClass myObj;
```

```
    // Use the move constructor to create a new instance of MyClass
    MyClass myObj2(std::move(myObj));

    // Print the address of the data pointer for each object
    std::cout << "myObj data: " << myObj.data << std::endl;
    std::cout << "myObj2 data: " << myObj2.data << std::endl;

    return 0;
}
```

输出：

```
myObj data: 0x0

myObj2 data: 0x7f8b4bc00010
```

其中，转移构造函数需要用到**“&&”右值引用**类型，这个类型表明被转移的对象是一个即将消亡的对象。

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。**

同样地，在没有定义析构函数前，系统会提供一个隐式的析构函数。但若类中有指针类成员时，考虑到内存的归还问题，需要自定义析构函数

如下：

```
class String{
private:
    char* p;
public:
    String(int n);
    ~String();
};
String::~String(){
    delete[] p;
}
String::String(int n){
    p = new char[n];
```

```
}
```

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，尤其是类中有指针类成员时

下面的实例有助于更好地理解析构函数的概念：

```
#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
```

先调用**本类的析构函数**，再调用**成员对象的析构函数。**

如有多个成员对象，则成员对象**析构函数的调用次序**与它们的**构造函数调用次序相反**

对象消亡的情况

静态对象

函数创建的临时对象在**函数结束**后

main函数结束后main函数中定义的所有对象。数组中的每一个对象都会各自调用析构。

程序关闭后程序的所有全局对象

动态对象

用**指针间接访问**的动态对象需要额外**用delete**

因为指针的消亡是它本身的消亡，而它指向的动态对象需要用delete消亡。

在使用delete之后，指针变量本身不会消亡，只是归还了它先前申请的空间。

运算符重载

大部分 C++ 内置的运算符可以被重载。

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。

运算符重载定义

定义为类成员函数

```
class complex
{
        complex operator + (const complex& c1){}
};
```

声明加法运算符用于把两个复数对象相加，返回最终的复数对象。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。

值得一提的是，此处加法重载可以通过**调用构造函数**返回，如

```
class complex
{
        complex operator + (const complex& c1){return complex(this->r+c1.r,this->i+
c1.i)}
};
```

定义为类的非成员函数

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并声明为友元，如下所示：

```
class complex
{
friend complex operator + (const complex& c1,const complex& c2)
//声明为友元可以访问私有成员
};
complex operator + (const complex& c1,const complex& c2)
```

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** 指针进行访问，如下所示：

```
#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

特殊运算符的重载

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。

重载流提取运算符和流插入运算符可以来操作对象等用户自定义的数据类型。

对输入/输出运算符重载函数要声明为类的友元函数，从而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 <<。

要注意的是，提取运算符的重载，**对象类型不能是一个常量引用**，不然无法正确输入数据

```
#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
```

++ 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。

前缀形式重载调用 operator ++ ()，后缀形式重载调用 operator ++ (int)。

int 在括号内是为了向编译器说明这是一个后缀形式，而不是表示整数。

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。递减运算符（ -- ）类似

```
#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
```

赋值运算符重载

就像其他运算符一样，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。

赋值操作符重载函数应当声明为**类内函数**，以能够访问**类内私有成员**

赋值操作重载函数应**被声明为引用类型**，且参数为**const引用**，以便于**链式赋值**

下面的实例演示了如何重载赋值运算符。

```
#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;
      }
      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
```

注意事项

- 要**考虑自我赋值**的情况。**两个对象相等时不能进行赋值。**
- 要考虑深拷贝操作。避免在进行**赋值操作后**会出现**两个对象中的指针指向同一处地方的情况**；
- 要避免赋值后**原对象内的指针指向的空间**没有指针指向，无法读取造成**内存泄漏。**

移动赋值运算符函数重载

与普通的赋值运算符不同，**转移赋值操作符函数**使用**右值引用**作为参数，并使用**&&**表示。

转移赋值操作符函数的作用是将**一个右值引用绑定的临时对象的资源**（如堆内存）**转移到目标对象上**，

从而避免不必要的资源拷贝和内存分配操作，提高程序性能。

例：

```
class MyClass {
public:
    MyClass& operator=(MyClass&& other) {
        if (this != &other) {
            // 释放当前对象的资源
            // 转移临时对象的资源到当前对象
            // 使临时对象进入“有效但未指定状态”
```

```
        }
        return *this;
    }
};
```

移动复制运算符函数的执行逻辑

1. 释放当前*this的资源
2. 占用other的资源
3. 设置other为默认构造状态
4. 返回*this

赋值运算符与拷贝构造函数的区别

定义格式不同

拷贝构造函数的定义格式为：

ClassName(const ClassName& other);

赋值操作符重载函数的定义格式为：

ClassName& operator=(const ClassName& other);

调用时机不同

拷贝构造函数在以下情况会被自动调用：

- 对象**声明时进行初始化**；
- 对象作为**函数参数**传递时；
- 对象作为**函数返回值**时；
- 对象进行**复制初始化**时。

赋值操作符重载函数则是在**对象已经存在**的情况下，将一个已有对象的值赋给另一个已有对象时，才会被调用。

返回值不同

拷贝构造函数没有返回值，因为它的作用是构造一个新的对象并将原对象的值复制到新对象中。

赋值操作符重载函数返回一个引用，因为它的作用是将一个已有对象的值赋给另一个已有对象，并返回被赋值后的对象的引用。

下标运算符 [] 重载

下标操作符 [] 通常用于访问数组元素。重载该运算符用于增强操作 C++ 数组的功能。

重载下标操作符[]时，**返回值类型通常是引用**，以允许使用下标操作符来访问和修改类中的数据成员。

下面的实例演示了如何重载下标运算符 []。

```
#include <iostream>
using namespace std;
const int SIZE = 10;

class safearay
{
   private:
      int arr[SIZE];
   public:
      safearay()
      {
         register int i;
         for(i = 0; i < SIZE; i++)
         {
           arr[i] = i;
         }
      }
      int& operator[](int i)
      {
          if( i >= SIZE )
          {
              cout << "索引超过最大值" <<endl;
              // 返回第一个元素
              return arr[0];
          }
          return arr[i];
      }
};
int main()
{
   safearay A;

   cout << "A[2] 的值为 : " << A[2] <<endl;
   cout << "A[5] 的值为 : " << A[5]<<endl;
   cout << "A[12] 的值为 : " << A[12]<<endl;
```

```
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
$ g++ -o test test.cpp
$ ./test
A[2] 的值为 : 2
A[5] 的值为 : 5
A[12] 的值为 : 索引超过最大值
0
```

函数调用运算符 () 重载

函数调用运算符 () 可以被重载用于类的对象。

重载 () 不是创造了一种新的调用函数的方式，相反地，这是创建一个可以传递任意数目参数的运算符函数。

下面的实例演示了如何重载函数调用运算符 ()。

```
#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;
      }
      // 重载函数调用运算符
      Distance operator()(int a, int b, int c)
      {
         Distance D;
```

```
         // 进行随机计算
         D.feet = a + c + 10;
         D.inches = b + c + 100 ;
         return D;
      }
      // 显示距离的方法
      void displayDistance()
      {
         cout << "F: " << feet <<  " I:" <<  inches << endl;
      }

};
int main()
{
   Distance D1(11, 10), D2;

   cout << "First Distance : ";
   D1.displayDistance();

   D2 = D1(10, 10, 10); // invoke operator()
   cout << "Second Distance :";
   D2.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: 30 I:120
```

类成员访问运算符->重载

类成员访问运算符（ -> ）可以被重载，但它较为麻烦。

它被定义用于为一个类赋予"指针"行为。运算符 -> 必须是一个成员函数。如果使用了 -> 运算符，返回类型必须是指针或者是类的对象。

运算符 -> 通常与指针引用运算符 * 结合使用，用于实现"**智能指针**"的功能。这些指针是行为与正常指针相似的对象，

唯一不同的是，当通过指针访问对象时，它们会执行其他的任务。比如，当指针销毁时，或者当指针指向另一个对象时，会自动删除对象。

间接引用运算符 -> 可被定义为**一个一元后缀运算符**。也就是说，给出一个类：

```
class Ptr{
   //...
   X * operator->();
};
```

类 **Ptr** 的对象可用于访问类 **X** 的成员，使用方式与指针的用法十分相似。例如：

```
void f(Ptr p )
{
   p->m = 10 ; // (p.operator->())->m = 10
}
```

语句 p->m 被解释为 (p.operator->())->m。同样地，下面的实例演示了如何重载类成员访问运算符 ->。

```
#include <iostream>
#include <vector>
using namespace std;

// 假设一个实际的类
class Obj {
   static int i, j;
public:
   void f() const { cout << i++ << endl; }
   void g() const { cout << j++ << endl; }
};

// 静态成员定义
int Obj::i = 10;
int Obj::j = 12;

// 为上面的类实现一个容器
class ObjContainer {
   vector<Obj*> a;
public:
   void add(Obj* obj)
   {
      a.push_back(obj);  // 调用向量的标准方法
   }
   friend class SmartPointer;
```

```
};

// 实现智能指针，用于访问类 Obj 的成员
class SmartPointer {
   ObjContainer oc;
   int index;
public:
   SmartPointer(ObjContainer& objc)
   {
       oc = objc;
       index = 0;
   }
   // 返回值表示列表结束
   bool operator++() // 前缀版本
   {
     if(index >= oc.a.size() - 1) return false;
     if(oc.a[++index] == 0) return false;
     return true;
   }
   bool operator++(int) // 后缀版本
   {
      return operator++();
   }
   // 重载运算符 ->
   Obj* operator->() const
   {
     if(!oc.a[index])
     {
        cout << "Zero value";
        return (Obj*)0;
     }
     return oc.a[index];
   }
};

int main() {
   const int sz = 10;
   Obj o[sz];
   ObjContainer oc;
   for(int i = 0; i < sz; i++)
   {
       oc.add(&o[i]);
   }
   SmartPointer sp(oc); // 创建一个迭代器
   do {
      sp->f(); // 智能指针调用
      sp->g();
   } while(sp++);
```

```
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
10
12
11
13
12
14
13
15
14
16
15
17
16
18
17
19
18
20
19
21
```

new运算符和delete运算符重载

new运算符和delete运算符重载在**类内定义**时，称为局部重载，仅该类对象使用时调用重载的new和delete

在类外定义时，所有new运算符和delete均被重载，称为全局重载。这可能会带来一些风险，并不推荐

重载这两个运算符可以在重载新的运算符功能中添加**异常处理例程**，

同时自定义运算符delete以**用0覆盖被回收的堆内存块**，可以提高应用程序数据的安全性

new操作符函数的原型

```
void* operator new(size_t size);
```

第一个参数必须为size_t size,代表申请空间的字节大小

delete操作符函数的原型

delete操作符必须匹配一个void*类型的参数，函数返回的类型是void，

并且默认情况下，重载后的new和delete操作符函数都是静态成员，因此在函数内部是**无法访问this指针**

```
void operator delete(void*);
```

该函数接收一个必须删除的void *类型的参数。 函数不应该返回任何东西。

delete重载只能有一个

注意：默认情况下，重载的new和delete运算符函数都是静态成员。 因此，他们无权访问此指针。

如下：

```
class Foo {
public:
 Foo() { std::cout << "Foo()" << std::endl; }
 virtual ~Foo() { std::cout << "~Foo()" << std::endl; }

 void* operator new(std::size_t size)
 {
  std::cout << "operator new" << std::endl;
  return std::malloc(size);
 }

 void* operator new(std::size_t size, int num)
 {
  std::cout << "operator new" << std::endl;
  std::cout << "num is " << num << std::endl;
  return std::malloc(size);
 }

 void* operator new (std::size_t size, void* p)
 {
  std::cout << "placement new" << std::endl;
  return p;
 }

 void operator delete(void* ptr)
 {
```

```
  std::cout << "operator delete" << std::endl;
  std::free(ptr);
 }

};

int main()
{
 Foo* m = new(100) Foo;
 Foo* m2 = new(m) Foo;
 std::cout << sizeof(m) << std::endl;
 //delete m2;
 delete m;
 return 0;

}
```

转换函数的重载

其他类型转换为类

可以通过定义构造函数的方式来实现，例如

```
class stonewt
{
private:
        int stone;
        double pds_left;
        double pounds;
public:
        stonewt(double lbs)//用于将double类型数据转为类函数
        {
                pounds=lbs;
        }
        stonewt(int stn,double lbs=0);//用于将int型转为类
        {
                pounds=lbs;stone=stn;
        }
        stonewt();
}

int main()
{
        stonewt mycat;
        mycat=19.6//将double型转为类
```

```
        mycat=stonewt(19.6)//跟上面相同，但是是显式类型转换
}
```

这样的构造函数除了**显式强制类型转换**，还可以用于下面的**隐式转换**

- 用**其他类初始化或赋值**给**类对象**
- 将**其他类数据**传递给接受**类参数的函数**
- **返回值**被声明为类的函数**试图返回一个其他类**时
- 以上情况下，使用可以转换为**double的内置类型**，如int

隐式转换方便，但是并不是十分安全，可以用**explicit关键字**来强制只能使用显式强制类型转换。如下

```
explicit stonewt(double lbs);

mycat=19.6//不允许，必须显式调用
mycat=stonewt(19.6)//可以
```

类转换为其他类

此时需要重载下面这种形式的转换函数

```
operator typename();
```

其中，typename为一个类名

转换函数必须是**类方法**，并且**不能指定返回类型**(typename处已经指定)，也**不能有参数**

将类转为double类型如下

```
class stonewt
{
private:
        int stone;
        double pds_left;
        double pounds;
public:
        stonewt();
        operator int() const//保证不会修改对象内成员的量
```

```
        {
                return int (pounds+0.5);
        }
}

int main()
{
        stonewt mycat;int weight
        mycat=19.6//将double型转为类
        weight=mycat//调用了重载后的类型转换
}
```

继承与多态

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效率的效果。

当创建一个类时，**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**。

继承声明

如下：

```
class derived-class:access-identifier base-class{

};
```

其中，derived-class是**派生类**，

access-identifier是**访问权限修饰符**，即public,private,protected,根据访问权限的不同有**公有继承，私有继承，保护继承，后两者较少被使用**

base-class为**基类**

派生类若有自己特殊的数据成员，最好有自己的**构造函数**

派生类与基类的关系

派生类与基类的转换

- 派生类可以用**基类的公有方法**，派生类**无法直接访问基类的私有成员**，要通过**基类的公有方法访问**
- **指向派生类的指针和引用**可以被用来指向**基类**，反之则不行 (不然可以用这样的指针让基类对象使用派生类的方法)
- 函数参数**若接受基类参数**，也可以**接受派生类参数**(隐式类型转换)

访问权限

	公有继承	保护继承	私有继承
基类公有成员	变成公有	变成保护	变成私有
基类保护成员	变成保护	变成保护	变成私有
基类私有成员	变成私有	变成私有	变成私有

派生类的构造函数

当派生类加入了新的数据成员，基类的构造函数可能无法包含派生类所需要的所有信息。需要定义派生类自己的构造函数。

派生类的构造函数**会首先调用基类的构造函数**，生成一个基类的**临时对象**，再用这个临时对象和自己的数据成员生成一个派生类对象。如下：

```
//这是一个基类
class fruit{
private:
string name;
int guarantee_period;//保质期
public:
fruit(string a,int b):name(a),guarantee_period(b){}
};

//这是一个派生类
class apple:public fruit{
private:
bool isred;
public:
apple(bool c,string a,int b):fruit(a,b){isred=c;}//使用基类的构造函数初始化与基类相同的数据成员
}
```

如果**不指出调用的基类构造函数**，则**默认调用基类默认构造函数**

如果基类使用**默认拷贝构造函数**不会造成问题或基类已有定义的**拷贝构造函数**，也可以这样定义派生类的构造函数

```
apple(bool c,const fruit& a):fruit(a){isred=c;}//1
apple(bool c,const fruit& a):fruit(a),isred(c)//2 广义的初始化列表
```

这里用基类的拷贝构造函数生成一个临时对象

公有继承成员函数属性小结

表 13.1 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

派生类的动态内存分配

基类有动态内存分配而派生类没有

此时**无需**为派生类定义**显式析构函数、拷贝构造函数、赋值运算**，派生类会自动调用基类定义的函数完成

派生类有动态内存分配

```
class baseDMA//基类
{
private:
        char* label;
        int rating;
public:
        baseDMA(const char* l="null",int r = 0);//显式定义构造函数
```

```
        baseDMA(const naseDMA &rs);//拷贝构造函数
        virtual ~baseDMA();//析构函数
        baseDMA &operator=(const baseDMA& rs);//赋值运算
};

class hasDMA:public baseDMA//派生类
{
private:
        char *style;
public:
        hasDMA(const char* l,int r,const char* s):baseDMA(l,r){style=new char[strle
n(s)+1];}
};

baseDMA::~baseDMA(){delete []label;}

baseDMA::baseDMA(const baseDMA& rs)
{
        label=new char[std::strlen(rs.label)+1];
        std::strcpy(label,rs.label);
        rating=rs.rating;
}

baseDMA& baseDMA::operatpr=(const baseDMA& rs)
{
        if(this=&rs)
        return *this;
        label=new char[std::strlen(rs.label)+1];
        std::strcpy(label,rs.label);
        rating=rs.rating;
        return *this;
}
```

此处的派生类用new为style成员变量分配动态内存，下面需要显式定义复制构造函数、析构函数、赋值运算

复制构造函数

派生类的**复制构造函数**只能访问其自身的数据，因此它要**调用基类的复制构造函数**来处理**属于基类部分**的数据

```
hasDMA::hasDMA(const hasDMA& hs):baseDMA(hs)
{
        style=new char[std::strlen(hs.style)+1];
```

```
        std::strcpy(style,hs.style);
}
```

析构函数

类似地，先**借助基类的析构函数**释放**基类的成员**，再用**自身的析构函数**释放自身的成员

```
hasDMA::~hasDMA()
{
        delete []style;
}
```

赋值运算符

参考基类的逻辑

```
hasDMA& hasDMA::operator=(const hasDMA& hs)
{
        if(this==&hs)
        return *this;
        baseDMA::operator=(hs)//调用基类的赋值运算，这里用函数表示法是为了能通过域解析符正确调用基类的赋值运算符
        //*(baseDMA*)this=hs 通过this指针进行初始化 也是可行的
        delete[] style;
        style=new char[std::strlen(hs.style)+1];
        std::strcpy(style,hs.style);
        return *this;
}
```

派生类的友元

如果派生类定义了一个友元函数，为了能使这个函数能够访问基类成员，

要在基类中也定义一个友元函数，并在派生类友元函数中，先将派生类参数进行强制类型转换转换为基类，再调用基类的友元函数

包含、私有继承、保护继承

前面所提到的继承均为**公有继承**，公有继承描述的是**is-a**关系，某物属于某一类的关系

而**包含**、**私有继承**、**保护继承**描述的是has-a关系，某物有某类东西的关系。例如汽车有轮胎，有引擎

- **is-a关系**继承了**接口**和**实现**，派生类可以使用基类的方法也可以重新定义基类的方法
- **has-a关系**只继承了实现**不继承接口**，派生类只能使用基类的方法而不能重新定义

> 例如，轮胎是会磨损的，
>
> 定义一个汽车会磨损是没有意义的，只能说一个汽车的轮胎磨损了。

包含

包含指的是在一个类中，**有一个成员属于其他类**

在类中通过**点运算符**就可以使用这个成员对象的方法

私有继承

私有继承得到的**基类方法**是**私有**的，因此用这样的派生类创建的对象不能使用基类的方法。对外界而言，派生类没有基类的方法。

私有继承可以看作包含了**基类对象**的一个**新类**，不过**基类对象**是**匿名**的，与公有继承相比，它**更类似于包含**

一般而言，如果新类需要**访问保护成员**或要**重新定义虚函数**的话，才会考虑用**私有继承**，否则还是用包含，包含更易懂

私有继承的构造函数

私有继承的构造函数使用**成员初始化表**，类似于公有继承的构造使用成员初始化表

```
class student:private std::string//这里不加访问限定关键词也是可以的，因为继承默认为私有的
{
        public:
        student(const char* str):std::string(str){}
}
```

私有继承调用基类方法

在派生类内，通过使用**类名**和**域解析符**调用基类方法

```
double namelen()const
{
```

```
        return std::string::size();//如果是成员对象的话，会通过点解析符来引用这个方法
}
```

私有继承访问基类对象和基类的友元函数

通过**强制类型转换**得到一个基类对象，这个返回的基类对象就可以用基类对象的方法和友元函数了

```
const string& name()const
{
        return (const string&)*this//利用this指针
}
```

保护继承

私有继承的变体，也属于描述has-a关系，很少用

多重继承

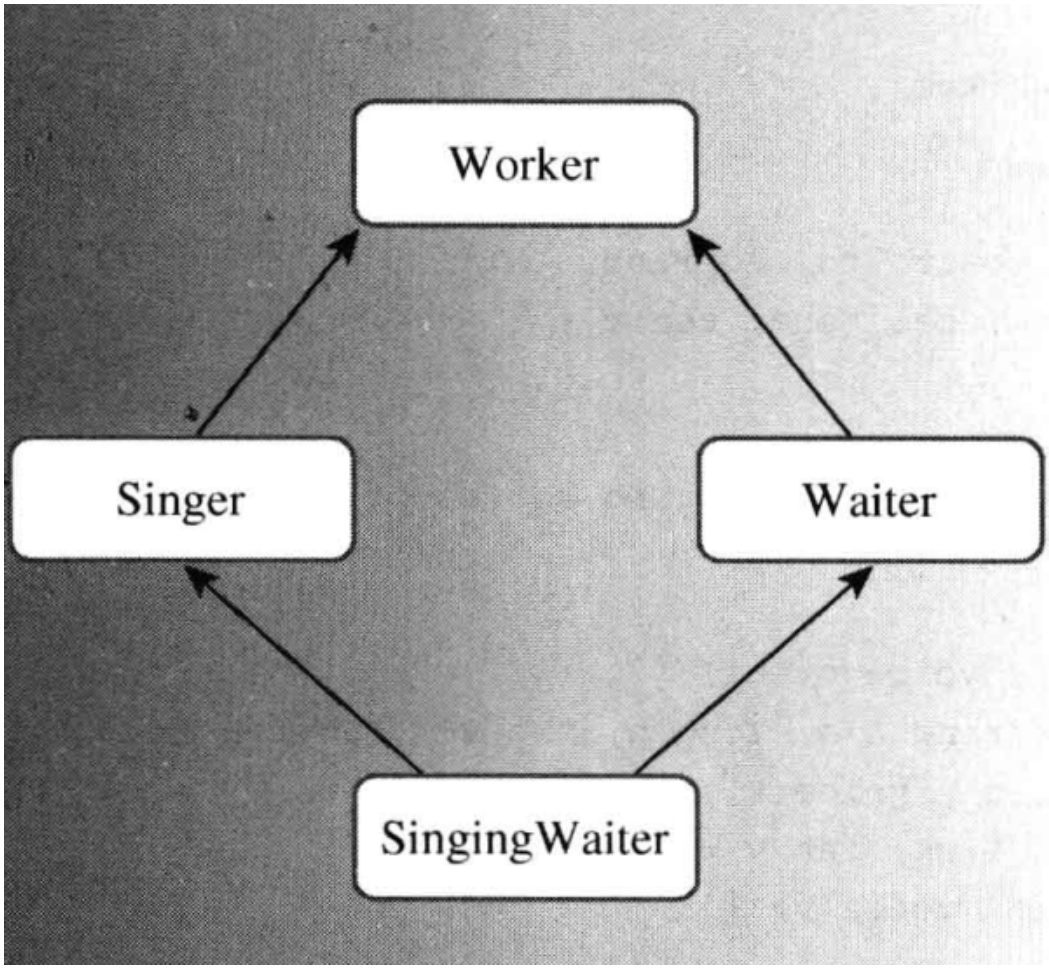
多重继承指的是一个派生类**继承自多个基类**，语法为

```
class classname:access-identifidition classA,access-identifidition classB...
```

虚基类

钻石继承问题

多重继承会带来一个如下图所示的问题



如图所示，在singingwaiter这个类中有两份worker的数据成员，一份是singer的worker，一份是waiter的worker

指针或引用不明确

如果在这样的代码中将singingwater的地址赋给一个worker指针，那么，

worker指针不知道应该获得singingwaiter的singer的worker对象的地址，还是singingwaiter的waiter的worker对象的地址

使用方法不明确

如果worker中有一个虚函数，singer和waiter的版本不一样，那么，

当singingwaiter使用这个函数的时候，编译器将不知道生成哪个版本的代码

虚基类的定义

解决上面的问题，只需要通过某种方法使得在新类被定义的时候，基类的**基类只出现一次**就可以了，为此有了虚基类这个概念

虚基类可以让**多个继承自同个基类的派生类**在被用作基类时，生成的派生类**只继承一份**最开始的基类的成员

虚基类的构造函数由派生链最末端的类调用，虚基类的构造函数优先于非虚基类的构造函数执行

```
class worker
{
...
}

class singer:virtual public worker//将worker用作虚基类
{
private:
int v;
...
}

class waiter:virtual public worker//将worker用作虚基类
{
private:
int p;
...
}

class singingwaiter:public singer,public waiter
{
...
}
```

重新定义构造函数

以上面的类为例，当worker被定义为虚基类时，**信息不能通过中间类进行传递，**下面的构造函数是不可行的

```
singingwaiter(const worker& wk,int p=0,int v=1):waiter(wk,p),singer(wk,v)//不可行
```

如果允许这样的信息传递，其中的wk会有两种途径传递至worker的数据成员，为了能够正确的拷贝，要**显式调用构造函数**

```
singingwaiter(const worker& wk,int p=0,int v=1):worker(wk),waiter(wk,p),singer(wk,
```

```
v)//可行
```

否则worker部分的构造会使用其**默认构造函数**

重新定义方法

现在，在worker，waiter，singer类中均有一个输出数据成员的函数show。为了能让singingwaiter明确使用哪个版本的函数，可以如下定义

```
void singingwaiter::showsinger()
{
        singer::show();
}

void singingwaiter::showwaiter()
{
        waiter::show();
}
```

虚二义性规则

若派生类与基类成员同名，则**优先使用派生类内**的成员

虚二义性规则与访问权限无关，

即使某个类继承自两个继承于相同基类的基类，且其中一个基类的同名函数为私有，在这个类调用此函数时仍然会有二义性

多态

虚函数

在某基类中声明为 **virtual** 并在**一个或多个派生类**中被**重新定义的成员函数**，使得可以**用基类的指针或引用**就可以**调用派生类的函数**，

是**实现多态**的一个重要机制

一般而言，**静态成员函数**，**构造函数**不能声明为虚函数

虚函数的声明

```
virtual type func-name();
```

虚函数的定义

虚函数在声明后，通过**域解析符**来对派生类和基类**各自的方法进行定义**

```
//这是一个基类
class fruit{
private:
string name;
int guarantee_period;//保质期
public:
fruit(string a,int b):name(a),guarantee_period(b){}
virtual bool isgood();//这是一个虚函数，根据水果的保质期判断水果是否是好的
virtual ~fruit();//这是一个虚析构函数，被继承到apple类后若用指针调用则会调用apple类析构函数
};

//这是一个派生类
class apple:public fruit{
private:
bool isred;
public:
apple(bool c,string a,int b):fruit(a,b){isred=c;}//使用基类的构造函数初始化与基类相同的数据成员
virtual bool isgood();//这是一个虚函数，根据保质期和是否是红的苹果判断苹果是否是好的
}
//进行两个类中方法的定义
bool fruit::isgood()
{
        return guarantee_period>12;//
}
bool apple::isgood()
{
        return fruit::isgood()&&isred;//调用了基类的函数，必须要加域解析符
}
```

当基类和派生类**有相同的方法**但**具体定义有差异**时，这样的方法在基类中一般**定义为虚函数。**

如果通过对象来引用类的方法，虚函数的作用并不明显

但如果**通过基类的指针和引用**来用类的方法，虚函数则是必须的

下面的代码中有一个fruit的指针数组，这些指针有指向fruit类的，也有指向apple类的

```
int main()
{
```

```
        fruit* FRUITS[3];
        fruit a1("hhh",9),a2("ggg",24);
        apple a3(false,"ccc",24);
        FRUITS[0]=&a1;
        FRUITS[1]=&a2;
        FRUITS[2]=&a3;//指针数组中前两个是fruit，后一个是apple
        for(int i=0;i<3;i++)
        {
                cout<<FRUITS[i]->isgood()<<endl;//调用每个对象的isgood，如果没有虚函数，由于使用基类指针，这里通过基类指针会调用基类的isgood，返回值为true
        }
        return 0;
}
```

虚函数的工作原理

动态联编与静态联编

静态联编

又称为**早期联编**，程序在运行前就知道这段函数应当被怎样编译。对于**非虚成员函数**通常是**静态联编**，静态联编效率更高

静态联编关注的是**指针或引用**本身的类型

但也有例外，派生类对象的**构造函数**如果**调用虚函数**，会采用**静态联编**

动态联编

又称为**晚期联编**，程序在运行时此知道这段函数应当怎么编译。**虚函数**一般采取这样的方式

动态联编关注的是**指针或引用指向的对象的类型**，如果这是一个**派生类**，那动态联编会把**派生类的函数**绑定到该对象上

动态联编效率低，因此虚函数只在必须使用的时候定义

上面的**联编**也称**绑定**

虚函数表

有虚函数的类中都有一个**隐藏地址成员**，它所存储的地址是**类中虚函数所在的地址。**

如果派生类对基类的虚函数重新定义，则派生类的虚函数表中**重新定义的虚函数**会存储在另一个地址中，继承的虚函数仍在原地址中。

在派生链中，每个**派生类所继承的虚函数**均解释为其**邻近父类的虚函数**

虚函数的不同使用情形

虚析构函数

通过**基类指针**对派生类对象进行delete操作会只调用基类的析构函数，如果派生类对象新增了指针类的数据成员，就需要它能够调用自己的析构函数

此时需要将析构函数声明为虚函数，使得派生类对象在析构时可以**正确调用析构函数**

```
class fruit{
private:
string name;
int guarantee_period;//保质期
public:
fruit(string a,int b):name(a),guarantee_period(b){}
virtual bool isgood();//这是一个虚函数，根据水果的保质期判断水果是否是好的
virtual ~fruit();//这是一个虚析构函数，被继承到apple类后若用指针调用则会调用apple类析构函数
};
```

一般最好**给基类定义一个虚析构函数**，不管它需不需要

虚函数的隐藏

如果在派生类中定义了一个参数列与基类虚函数不同的虚函数，这个虚函数将隐藏基类的虚函数，如

```
class Dwelling
{
public:
        virtual void showperks(int a)const
...
};

class Hovel:public Dwelling
{
public:
        virtual void showperks()const;
};
...
Hovel trump;
trump.showperks()//有效
trump.showperks(5)//无效，基类的虚函数已经被隐藏了
```

只有在基类中定义这个函数的重载版本，才能避免这种情况发生。

不过如果是**返回值类型从返回基类引用或指针变为返回派生引用或指针**的话是可以的，这个规则称为**返回类型协变**

虚说明符override和final

当基类声明了一个虚方法时，派生类如果想**覆盖虚方法**，可以在声明函数时添加**override**

```
virtual void f(char* ch)const override;//这个派生类的方法应当是覆盖原有函数的，也就是说，它的参数列应该与基类的虚方法中的参数列类型相同，不相同则会报错
```

如果只是**想隐藏旧版本**而不想覆盖特定虚方法，可以使用**final**，**禁止派生类重新定义函数**

```
virtual void f(char ch)const final;
```

抽象基类(ABC)

抽象基类是一种**只作为基类而不用于生成对象**的类，

例如要创建兔子和老虎两个类，就可以创建一个动物的抽象基类，显然，创建一个动物的对象并没有什么意义

包含纯虚函数的类就称为**抽象基类**

纯虚函数

纯虚函数的声明

```
virtual func-type func-name()=0//只需在虚函数的后面加上一个"=0"
```

纯虚函数本身不需要被定义，它可以看作是一种**约束规范**，说明所有的派生类都要有这样一个类似功能的函数

> 例如动物都要吃饭，兔子怎样吃饭和老虎怎样吃饭都是可以被定义的，但动物怎样吃饭本身是无法定义的

抽象基类的实例

将上面的fruit类改为抽象基类，如下

```
//这是一个枚举变量
typedef enum color{
Red=0,Yellow
}color;

//这是一个抽象基类
class fruit{
private:
string name;
int guarantee_period;//保质期
color c;
public:
fruit(string a,int b,color c=Red):name(a),guarantee_period(b)
virtual bool isgood()=0;//这是一个纯虚函数，下面的派生类都得有这个方法的实现
virtual ~fruit();//这是一个虚析构函数，被继承到apple类后若用指针调用则会调用apple类析构函数
};

//这是一个apple派生类
class apple:public fruit{
public:
apple(color COLOR,string a,int b):fruit(a,b),c(COLOR)//使用基类的构造函数初始化与基类相同的数据成员
virtual bool isgood();//apple的虚函数，根据保质期和是否是红的苹果判断苹果是否是好的
}

//这是一个banana派生类
class banana:public fruit{
public:
banana(color COLOR,string a,int b):fruit(a,b),c(COLOR)//使用基类的构造函数初始化与基类相同的数据成员
virtual bool isgood();//banana的虚函数，根据保质期和是否是黄的香蕉判断香蕉是否是好的
}

//进行两个类中方法的定义
bool apple::isgood()
{
        return color==Red;
}

bool banana::isgood()
{
        return color==Yellow;
}
```

模板

模板函数

对于**类型不同**，需要**类似操作**的数据，可以编写模板函数，使编译器根据模板**函数所提供的方案**和**传入的参数类型**自动生成所需函数。

是一种**通用的模板函数描述**，其中的泛型可用**任一个具体类型替换。**

通过将类型作为参数传递给模板，可以使编译器生成该类型的函数。要注意的是，泛型函数只是一个方案，并不是具体的函数

模板函数的定义

```
template <typename T>
FunClass Funname(T paremeter1,...)
{

}
```

template和**typename**是模板函数的关键字，

T可以是任何一个**类型名**，要用尖括号圈起来

在下面的函数体中，均用T来代替参数类型

模板函数的形参列表**并不一定需要全是泛型参数**

通常，这样的模板将会放在**头文件中。**

模板函数重载

类似普通的函数重载，可以定义参数列不同的同名模板，例如：

```
template<typename T>
void Swap(T &a, T&b)//模板1

template<typename T>
void Swap(T *a, T *b,int n)//模板2

int main()
{
int i=10,j=20;
...
```

```
Swap(i,j);//调用模板1
int d1={1,2};
int d2={3,4};
Swap(d1,d2,2)//调用模板2
}
```

模板函数的具体化

对于某些特定的类，模板无法处理它们的特殊情况，可以通过**具体化模板函数**，为这些特定的类提供一个特殊的解决方案。

对于给定的变量，非模板函数优先于具体化模板函数，具体化模板函数优先于常规模板函数

隐式实例化

即使用常规模板函数生成函数

显式具体化

例如，有一个构造体为

```
struct job
{
        char name[40];
        double salary;
        int floor;
}
```

已有模板

```
template<typename T>void Swap(T& ,T&)
```

交换函数会直接将两个构造的值全部交换，如果只想交换两个job的salary，可以用如下的显式具体化泛型函数

```
template<>void Swap(job& ,job&)
```

其中typename T被略去，因为函数的参数类型已经说明了泛型所指代的类型

显式实例化

显式实例化和显式具体化相似，

但不同的是显式实例化是**指示编译器生成这样一个函数**，

显式具体化只是一个**生成函数的特殊方案。**以上面的Swap函数为例

```
template void Swap<job>(job&,job&);
```

decltype关键字和后置返回类型

泛型函数有时无法**确定返回值的类型**，例如下面这个函数

```
template<class T1,class T2>
? type ? gt(T1 x,T2 y)
{
        ...
        return x+y;
}
```

当T1和T2均为int时，返回值为int；

当有一方为double时，返回值应为double

为解决这样的问题，C++提供了decltype关键字和后置返回类型

decltype关键字

可以在定义一个变量时，使之类型与另一个变量或一个表达式的**类型相同**，格式如下：

```
class x;
decltype(x) y;
decltype(x+y) z;
decltype(fun(x)) w;//并不会实际调用函数，只查看函数返回值类型
```

后置返回类型

可以将函数的返回类型置于参数列后，从而在得知形参类型后，可以**确定返回值的类型**，例如：

```
template<class T1,class T2>
auto fun(T1 x, T2 y)->decltype(x+y)
{
        ...
        return x+y;
}
```

其中，auto是一个占位符，指代后面的->decltype(x+y)

模板类

类似模板函数的需要，只要**提供一个基本类型**，就可以根据这个模板类创造所需要的**实例类**

模板类的定义

以定义一个**模板栈**类型为例

```
template <typename T>//类的定义,T是未知类型名
class stack
{
private:
        T items[10];//未知类型的数组
        int top;
public:
        stack();
        bool isempty();
        bool isfull();
        bool push(const T& item);
        bool pop(T& item);
        stack(const stack& st);//此处的stack是一种缩写，没有带上<type>，这种缩写在类内允许，类外声明时要完整写出
        stack& operator=(const stack& st);
};

//下面给出成员函数的定义，但在每个成员函数定义之前，要有模板声明

template <typename T>
stack<T>::stack()//原先定义函数仅需要类名加解析符，现在还要加上<T>
{
        top=0;
}

template <typename T>
bool stack<T>::isempty()
```

```
{
        return top==0;
}

template <typename T>
bool stack<T>::isfull()
{
        return top==10;
}

template <typename T>
bool stack<T>::push(const T& item)
{
        if(top<10)
        {
                items[top++]=item;
                return true;
        }
        else
        return false;
}

template <typename T>
bool stack<T>::pop(T& item)
{
        if(top>0)
        {
                item=items[--top];
                return true;
        }
        else
        return false;
}

template <typename T>
stack<T>(const stack& st)
{
        top=st.top;
        for(int i=0;i<top;i++)
        items[i]=st.items[i];
}

template <typename T>
stack<T>& stack<T>::operator=(const stack& st)//返回类型，解析域名都要完整写出
{
        if(this==&st)
        return *this;
        delete []items;
```

```
        top=st.top;
        items=new T[10];
        for(int i=0;i<top;i++)
        items[i]=st.items[i]
        return *this;
}
```

类型的模板通常被放在**同一个头文件**中

非类型参数声明

模板头中除了含有类型参数外，也可以含有一个**非类型参数**，非类型参数的类型只能是**整型、枚举、引用**或**指针**，使用如下:

```
//下面是一个可变大小的数组模板
template<typename T,int n>//这里的n就是非类型参数
class ArrayTp
{
private:
        T ar[n]//使用n
...
}
```

非类型参数传入时必须是一个**常量表达式**，

而且因为是常量，类内不能对这个常量执行类似自增这样**改变其值**的操作

如每次传入非类型参数不同，会生成不同的类

模板类的使用

使用模板类实例化，只需传入一个**类型参数**即可，以上面的stack为例

```
stack<int>kernels;//创建了一个整型参数的栈
stack<string>colonels;//创建了一个string参数的栈
```

模板类的具体化

隐式实例化

直接用类声明对象时，会进行隐式实例化

显式具体化

类似函数模板的具体化，当需要为**特殊类型实例化**时，可以**显式具体化**，这里有一个表示排序后数组的类模板

```
template<typename T>
class sortedarray
{
...
}
//现在提供一个专为const char*类型使用的模板
template<>class sortedarray<const char*>
{
...
}
```

模板类内的友元

模板类的非模板友元函数

如果在模板类中将一个函数声明为**友元**，这个模板**所有实例化的模板类**均将其作为**友元函数**

```
template<class T>
class hasfriend
{
public:
        friend void counts();//hasfriend<int>,hasfriend<double>均会将其看作友元函数
        friend void report(hasfriend<int>&);//如果友元函数要传入一个参数，它的类型不能是hasfriend这样的模板，因为模板是没有被实现的。只能像这样写一个具体的类
..
};
```

模板类的约束模板友元函数

在**类外声明模板函数**，将**非模板友元函数**修改为模板函数并将它们声明为友元。

```
template <typename T>void counts();//在模板类前先声明
template <typename T>void reports(T &);

template <typename TT>
class hasfriendT
```

```
{
...
        friend void counts<TT>();//利用显式具体化生成一个符合要求的友元函数
        friend void reports<>(hasfriendT<TT>&);//函数参数可以知道用什么类型具体化
}
```

模板类的非约束模板友元函数

在**类内声明模板函数**

```
template<typename T>
class manyfriend
{
...
        template<typename C,typename D>friend void show2(C&,D&);
};
```

模板类的其他特性

嵌套模板

模板类实例化后，其本身可以被当作另一个**类型参数**传入另一个模板。以上面提到的ArrayTp和stack为例

```
ArrayTp<stack<int>> a;//一个以整型为成员的栈的数组
ArrayTp<ArrayTp<int,5>,10>matrix//一个以含五个整形数据的数组为元素的，含十个元素的数组。等同于10×5的矩阵
```

多类型参数

在模板声明中可以加入**多个类型参数**，例

```
template<typename T1,typename T2>
class pair
{
private:
        T1 a;
        T2 b;
public:
..
}
```

```
...
pair<string,int>//实例化一个类
```

默认类型模板参数

在模板函数中，无法提供一个默认的类型参数，只能提供**非类型参数的默认值**，

但模板类两者皆可

```
template <class T1,class T2=int>
class classname
{
...
};
```

成员模板

模板本身可以用作**结构**、**类**、**模板类**的成员，如下代码

```
template<typename T>
class beta
{
private:
        //这是一个嵌套的模板类
        template<typename V>
        class hold
        {
        private:
                V val;
        public:
                hold(V v=0):val(v){}
                void show()const;
        //下面是原类的数据成员
        hold<T>q;//模板对象
public:
        beta(T t,int i):q(t),n(i){};
        template<typename U>
        U blab(U u,T t);//这里的类型U，在需要使用成员方法时，根据传入的参数类型即可确定
}

//下面是嵌套模板类中成员函数的定义，在类外定义时，要依次标明其所属的模板类
template<typename T>
        template<typename U>
                U beta<T>::blab(U u,T t)
```

```
{
        return (q.value*u/t);
}
```

模板类用作参数

在定义的时候将**模板类**当作**类型参数**传入，如下面代码所示

```
template<template<typename T>class thing>//要求传入一个含一个类型参数的模板类
class crab
{
private:
        thing<int>s1;//使用传入的模板类定义成员变量
}
...
crab<stack>nebula;//此处的stack是一个和thing能够匹配的模板类，它和thing一样只要求传入一个类型参数
```