第六章 数据抽象—对象和类

本章内容

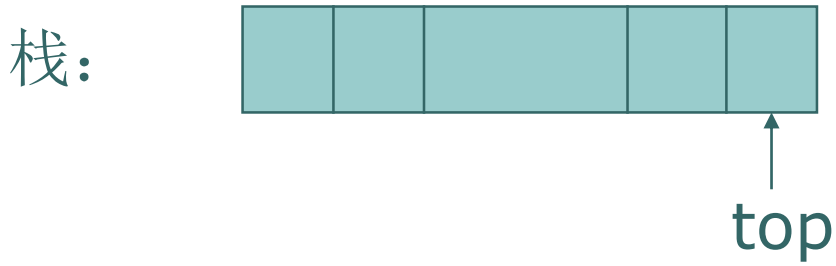
- 面向对象基本思想
- 类和对象
- 对象的初始化和消亡前处理
- const成员
- 静态成员
- 友元

什么是面向对象程序设计

- 把程序构造成由若干对象组成，每个对象由一些数据以及对这些数据所能实施的操作构成；
- 对数据的操作是通过向包含数据的对象发送消息（调用对象的操作）来实现；
- 对象的特征（数据与操作）由相应的类来描述；
- 一个类所描述的对象特征可以从其它的类获得（继承）。

例：栈

- 栈是一种由若干个具有线性次序的元素所构成的复合数据。
- 两种操作：进栈和退栈，并且这两个操作必须在栈的同一端（称为栈顶，top）进行，具有后进先出（Last In First Out，简称LIFO）的重要性质。



“栈”数据的表示及其操作—过程式程序

```
#include <iostream>
using namespace std;
//定义栈数据类型
const int STACK_SIZE=100;

struct Stack
{   int top;
    int buffer[STACK_SIZE];
};

void init(Stack &s)
{   s.top = -1;
}
```

```
bool push(Stack &s, int i)
{   if (s.top == STACK_SIZE-1)
    {       cout << “Stack is overflow.\n”;
            return false;
    }
    else
    {       s.top++;     s.buffer[s.top] = i;
            return true;
    }
}

bool pop(Stack &s, int &i)
{   if (s.top == -1)
    {       cout <<“Stack is empty.\n”;
            return false;
    }
    else
    {       i = s.buffer[s.top];     s.top--;
            return true;
    }
}
```

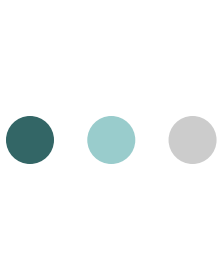
```
//使用栈类型数据
Stack st;
int x;
init(st);  //对st进行初始化。
push(st,12);  //把12放进栈。
pop(st,x);  //把栈顶元素退栈并存入变量x。
或
Stack st;
int x;
st.top = -1; //对st进行初始化。
st.top++; //把12放进栈。
st.buffer[st.top] = 12; //把栈顶元素退栈并存入变量x。
x = st.buffer[st.top];
st.top--;
```

例：“栈”数据的表示及其操作—面向对象程序

```
#include <iostream>
using namespace std;
//定义栈数据类型
const int STACK_SIZE=100;
class Stack
{   int top;
    int buffer[STACK_SIZE];
 public:
    Stack() { top = -1; }
    bool push(int i);
    bool pop(int &i);
}
```

```
bool Stack::push(int i)
{   if (top == STACK_SIZE-1)
    {   cout << “Stack is overflow.\n”;
        return false;
    }
    else
    {   top++;    buffer[top] = i;
        return true;
    }
}

bool Stack::pop(int &i)
{   if (top == -1)
    {   cout << “Stack is empty.\n”;
        return false;
    }
    else
    {   i = buffer[top];   top--;
        return true;
    }
}
```



```
//使用栈类型数据
Stack st;  //自动地去调用st.Stack()对st进行初始化。
int x;
st.push(12);  //把12放进栈st。
st.pop(x);  //把栈顶元素退栈并存入变量x。
st.top = -1;  //Error
st.top ++;  //Error
st.buffer[st.top] = 12;  //Error
```

- 两种解决方案存在不同：
  - 在方案(1)中，数据的表示对数据的使用者是公开的，对栈的操作可以通过所提供的函数来实现，也可以直接在栈的数据表示上进行；而在方案(2)中，只能通过提供的函数来操作栈。
  - 在方案(1)中，数据和对数据的操作相互独立，数据是作为参数传给对数据进行操作的函数；而在方案(2)中，数据和对数据的操作构成了一个整体，数据操作是数据定义的一部分。
  - 方案(1)需要显式地对栈进行初始化，方案(2)则是隐式地（自动）进行初始化。

面向对象的基本内容

- 对象/类(Object&Class)
  - 对象是由数据及能对其实施的操作（成员函数、方法或消息处理过程等）所构成的封装体，它属于值的范畴。
  - 类（对象的类型）描述了一组具有相同特征的对象，它属于类型的范畴 。
- 继承(Inheritance)
  - 在定义一个类时，可以利用已有类的一些特征描述。
  - 父类（基类）与子类（派生类）
  - 单继承与多继承
  - 代码复用

- 多态性(Polymorphism)，某一论域中的一个元素存在多种解释
  - 一名多用：在同一个作用域中用同一个名字为不同程序实体命名
    - 函数名重载
    - 操作符重载
  - 类属性：指一个程序实体能对多种类型的数据进行操作或描述的特性
    - 类属函数：一个函数能对多种类型的数据进行操作。
    - 类属类型：一个类型可以描述多种类型的数据。

- 面向对象程序设计特有的多态：
  - 对象类型的多态：子类对象既属于子类，也属于父类。
  - 对象标识的多态：父类的引用或指针可以引用或指向子类对象。
  - 消息的多态：一个消息集有多种解释（父类与子类有不同解释）。
- 多态的好处
  - 易于实现程序高层（上层）代码的复用，使得程序扩充变得容易（只要增加底层的具体实现）。
  - 增强语言的可扩充性（操作符重载等）。

- 绑定：确定对多态元素使用的过程，即确定对多态元素的某个使用是多态元素的哪一种形式。
  - 静态绑定（Static Binding，也称前期绑定，Early Binding）：在编译时刻确定。
  - 动态绑定（Dynamic Binding，也称后期绑定或延迟绑定，Late Binding）：在运行时刻确定。

为什么要面向对象?

- 面向对象程序设计的优势 P205-P209
  - 抽象：过程抽象 vs 数据抽象
  - 封装：过程封装 vs 数据封装
  - 模块化：子程序 vs 对象类
  - 软件复用：子程序库 vs 类库和继承机制
  - 软件维护：功能分解 vs 对象和类

类

- 对象构成了面向对象程序的基本单位，而对象的特征则由相应的类来描述。
- C++的类是一种用户自定义类型。
  - 定义形式如下：

    class <类名> { <成员描述> } ;
    其中，类的成员包括：
    数据成员
    成员函数

例：一个日期类的定义

```
class Date
{   public:
        void set(int y, int m, int d) //成员函数
        {       year = y;
                month = m;
                day = d;
        }
        bool is_leap_year() //成员函数
        {       return (year%4 == 0 && year%100 != 0) ||
                                        (year%400==0);
        }
        void print() //成员函数
        {       cout << year << "." << month << "." <<day;
        }
    private:
        int year,month,day; //数据成员
};
```

数据成员

- 数据成员的声明格式与非成员数据的声明格式相同

  例如： class Date //类定义

  ```
  {   ......
      private: //访问控制说明
          int year,month,day;  //数据成员说明  };
  ```

- 声明数据成员时不允许进行初始化（某些静态数据成员除外）。

  例如：class A

  ```
  {   int x=0;     //Error
      const double y=0.0;     //Error
      ......  };
  ```

- 数据成员的类型可以是任意的C++类型（void除外）
- 在声明一个数据成员的类型时，如果未见到相应的类型定义或相应的类型未定义完，则该数据成员的类型只能是这些类型的指针或引用类型（静态成员除外）

例如：
```
class A;  //A是在程序其它地方定义的类
class B
{   A a; //Error，未见A的定义。
    B b; //Error，B还未定义完。
    A *p;  //OK
    B *q;  //OK
    A &aa; //OK
    B &bb; //OK   };
```

成员函数

- 成员函数定义可放在类定义中，也可放到类定义外

例如：class A

```
{    ...
     void f() {...} //建议编译器按内联函数处理。
};
```

例如：class A

```
{    ...
     void f();
};
```

void A::f() { ... } //需要用类名受限。

- 类成员函数是可以重载的（析构函数除外），它遵循一般函数名的重载规则。

例如：class A

```
{     ......
    public:
        void f();
        int f(int i);
        double f(double d);
        ......
};
```

类成员的访问控制

- C++使用控制修饰符来描述对类成员的访问限制。

例如：

```
class A
{    public: //访问不受限制。
        int x;
        void f();
     private: //只能在本类和友元中访问。
        int y;
        void g();
     protected: //只能在本类、派生类和友元中访问。
        int z;
        void h();
};
```

- 在类定义中，可以有多个public、private和protected访问控制说明，C++的默认访问控制是private

例如：

```
class A
{    int m,n; //m,n的访问控制说明为private。
  public:
     int x;   void f();
  private:
     int y;   void g();
  protected:
     int z;   void h();
  public:
     void f1();
};
```

- 类的数据成员和在类的内部使用的成员函数应该指定为private，只有提供给外界使用的成员函数才指定为public。
- public访问控制的成员构成了类与外界的一种接口（interface）。操作一个对象时，只能通过访问对象类中的public成员来实现。
- protected类成员访问控制具有特殊的作用（继承）。

用链表实现栈类Stack

```
#include <iostream>
#include <cstdio>
using namespace std;
class Stack
{   public: //对外的接口
        Stack() { top = NULL; }
        bool push(int i);
        bool pop(int& i);
    private:
        struct Node
        {    int content;
             Node *next;
        } *top;
};
```

```
bool Stack::push(int i)
{   Node *p=new Node;
    if (p == NULL)
    {       cout << "Stack is overflow.\n";
            return false; }
    else
    {       p->content = i;
            p->next = top;  top = p;
            return true; }
}

bool Stack::pop(int& i)
{   if (top == NULL)
    {       cout << "Stack is empty.\n";
            return false; }
    else
    {       Node *p=top;
            top = top->next;
            i = p->content;
            delete p;
            return true; }
}
```

对象

- 类属于类型范畴的程序实体，它存在于静态的程序（运行前的程序）中。而动态的面向对象程序（运行中的程序）则是由对象构成。
- 对象在程序运行时创建，程序的执行是通过对象之间相互发送消息来实现的。当对象接收到一条消息后，它将调用对象类中定义的某个成员函数来处理这条消息。

对象的创建和标识

- 直接方式
  - 定义一个类型为类的变量。
  - 对象通过对象名来标识和访问。
  - 分为全局对象和局部对象。

例如：

```
class A
{ public:
      void f();  void g();
  private:
      int x,y;
};
A a1;  //创建一个A类的对象。
A a2[100]; //创建由100个A类对象组成的数组。
```

- 间接方式（动态对象）
  - 在程序运行时刻，通过new操作或malloc库函数来创建对象。
  - 所创建的对象称为动态对象，其内存空间在程序的堆区中。
  - 动态对象用delete操作或free库函数来撤消。
  - 动态对象通过指针来标识和访问。

例如：**A *p;**

p = new A[100]; //创建一个动态对象数组。 ……

delete []p; //撤消p所指向的动态对象数组。

或 p = (A *)malloc(sizeof(A)*100); ……

free(p);

- 对于动态对象，需注意下面两点：
  - 函数malloc不会调用对象类的构造函数。函数free也不会调对象类的析构函数。
  - 用new创建的动态对象数组只能用默认构造函数进行初始化。delete中的“[]”不能省，否则，只有第一个对象的析构函数会被调用。

对象的操作

- 通过调用对象类中的成员函数来进行操作。

例如：
```
class A
{     int x;
   public:
      void f();  };
int main()
{  A a;      //创建A类的一个局部对象a。
   a.f();    //调用A类的成员函数f对对象a进行操作。
   A *p=new A;   //创建A类的一个动态对象，p指向之。
   p->f();  //调用A类的成员函数f对p所指向的对象进行操作。
   delete p;
   return 0;   }
```

- 通过对象来访问类的成员要受到访问控制符的限制

```
class A
{   public:
        void f()
        { ..... //允许访问：x,y,f,g,h
        }
    private:
        int x;
        void g()
        { ..... //允许访问：x,y,f,g,h
        }
    protected:
        int y;
        void h()
        { ..... //允许访问：x,y,f,g,h
        }
};
```

```
.....
A a;
a.f();  //OK
a.x = 1;  //Error
a.g();  //Error
a.y = 1;  //Error
a.h();  //Error
```

- 对对象进行赋值

  例如：**Date yesterday,today,some_day;**

  some_day = yesterday; //把yesterday数据成员分别赋值

  //给some_day相应数据成员

- 取对象地址

  例如：**Date *p_date;**

  p_date = &today; //把对象today的地址赋值给指针p_date

- 把对象作为函数实参以及作为函数的返回值等操作。

  例如： **Date f(Date d); //函数f的声明，需要一个Date类的对象**

  //作为参数，返回值为Date类的对象

  又如：**some_day2 = f(yesterday); //调用函数f，把yesterday**

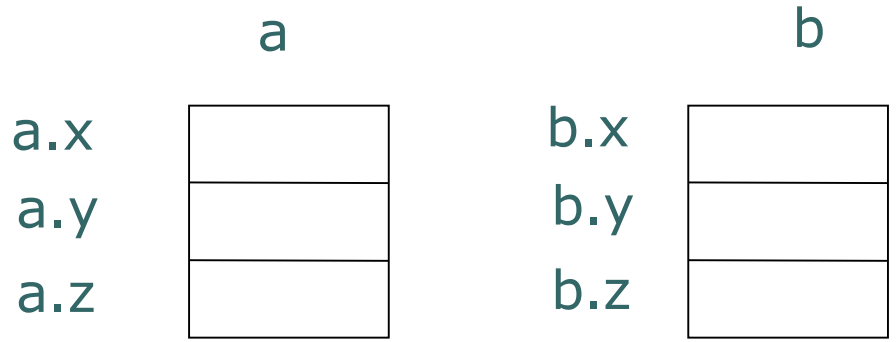
  //作为实参。返回值赋给some_day2

this指针

- 类定义中说明的数据成员（静态数据成员除外）对该类的每个对象都有一个拷贝。

例如：

```
class A
{   public:
        void f();
        void g(int i) { x = i;  f(); };
    private:
        int x,y,z;
};
A a,b;
```



- 类中的成员函数对该类的所有对象只有一个拷贝。
- 每个成员函数都有一个隐藏的形参this，其类型为：

<类名> * const this;

例如，类A的成员函数g的实际形式为：

```
void g(A *const this, int i)
{ this->x = i;
  this->f();
};
```

函数调用 a.g(1)编译成A::g(&a,1);

- 一般情况下，类的成员函数中不必显式使用this指针来访问对象的成员（编译程序会自动加上）。如果成员函数中要把this所指向的对象作为整体来操作（如：取对象的地址），则需要显式地使用this指针。

例如：**void func(A *p)**

```
        { ......
        }
class A
{     int x;
  public:
      void f() {  func(?);  } //“?”应写什么?  func(this);
      void q(int i) { x = i; f(); }
};
......
A a,b;
a.f(); //要求在f中调用func(&a)
b.f(); //要求在f中调用func(&b)
```

对象的初始化 – 构造函数

- 当一个对象被创建时，它将获得一块存储空间，该存储空间用于存储对象的数据成员。在使用对象前，需要对对象存储空间中的数据成员进行初始化。
- C++提供构造函数进行对象初始化。它是类的特殊成员函数，名字与类名相同、无返回值类型。
- 创建对象时，构造函数会自动被调用。

例如：

```
class A
{    int x;
     public:
         A() { x = 0; } //构造函数
 };
A a; //创建对象a：为a分配内存，然后调用a的构造
        函数A()。
```

- 构造函数可以重载。其中，不带参数的（或所有参数都有默认值的）构造函数被称为默认构造函数。

例如：

```
class A
{       int x,y;
    public:
        A() //默认构造函数
        {   x = y = 0; }
        A(int x1)
        {   x = x1; y = 0; }
        A(int x1,int y1)
        {   x = x1; y = y1;}
         ......
};
```

- 如果类中未提供任何构造函数，则编译程序在需要时会隐式地为之提供一个默认构造函数。
- 在创建对象时，可以显式地指定调用对象类的某个构造函数。如果没有指定调用何种构造函数，则调用默认构造函数初始化。

```
class A
{       ......
    public:
        A();
        A(int i);
        A(char *p);
};
```

A a1; //调用默认构造函数。可写成A a1=A(); 但不能写成A a1();
(Warning！)

A a2(1); //调用A(int i)。可写成A a2=A(1); 或 A a2=1;

A a3("abcd"); //调A(char *)。可写成A a3=A("abcd"); 或 A a3="abcd";

A a[4]; //调用对象a[0]、a[1]、a[2]、a[3]的默认构造函数。

A b[5]={A(), A(1), A("abcd"), 2, "xyz"}; //调用b[0]的A()、
//b[1]的A(int)、b[2]的A(char *)、b[3]的A(int)和b[4]的A(char *)

A *p1=new A; //调用默认构造函数。

A *p2=new A(2); //调用A(int i)。

A *p3=new A("xyz"); //调用A(char *)。

A *p4=new A[20]; //创建动态对象数组时调用各对象的默认构造函数

成员初始化表

- 对于常量数据成员和引用数据成员（某些静态成员除外），不能在说明它们时初始化，也不能用赋值操作对它们初始化。
- 例如：

```
class A
{       int x;
        const int y=1;   //Error
        int &z=x;     //Error
    public:
        A()
        {   x = 0;    //OK
            y = 1;    //Error
            z = &x;  //Error
         }
};
```

- 采用成员初始化表来进行初始化。

例如：
```
class A
{     int x;
      const int y;
      int& z;
   public:
      A(): z(x),y(1)  //成员初始化表
      { x = 0;}
 };
```

- 成员初始化表中成员初始化的书写次序并不决定它们的初始化次序，数据成员的初始化次序由它们在类定义中的说明次序来决定 。

成员对象

- 对于类的数据成员，其类型可以是另一个类。也就是说，一个对象可以包含另一个对象（称为成员对象）。

例如：

```
class A
{  ...
};
class B
{ ...
  A a; //成员对象
  ...
};
B b; //对象b包含一个成员对象：b.a
```

成员对象的初始化

- 成员对象由成员对象类的构造函数初始化。
- 如果在包含成员对象的类中，没有指出用成员对象类的什么构造函数对成员对象初始化，则调用成员对象类的默认构造函数。
- 可以在类构造函数的成员初始化表中显式指出用成员对象类的某个构造函数对成员对象初始化。

```
class A
{     int x;
  public:
      A() { x = 0; }
      A(int i) { x = i; }
};
class B
{     A a;
       int y;
  public:
      B(int i) { y = i;} //调用A的默认构造函数对a初始化。
      B(int i, int j): a(j) { y = i; } //调用A(int)对a初始化。
};
B b1(1); //b1.y初始化为1，b1.a.x初始化为0
B b2(1,2); //b2.y初始化为1，b2.a.x初始化为2
```

- 创建包含成员对象的类的对象时，先执行成员对象类的构造函数，再执行本身类的构造函数。
- 一个类若包含多个成员对象，这些对象的初始化次序按它们在类中的说明次序（而不是成员初始化表的次序）进行。
- 析构函数的执行次序与构造函数的执行次序正好相反。

拷贝构造函数

- 在创建一个对象时，若用一个同类型的对象对其初始化，这时将会调用一个特殊的构造函数：拷贝构造函数。

例如：

```
class A
{   ......
    public:
        A();  //默认构造函数
        A(const A& a);  //拷贝构造函数
};
```

- 在三种情况下，会调用类的拷贝构造函数：
  - 定义对象

    例如：**A a1;**

    A a2(a1)；//也可写成A a2=a1; 或A a2=A(a1);

    //调用A拷贝构造函数，用a1初始化a2

  - 把对象作为值参数传给函数

    例如：**void f(A x);**

    A a;

    f(a)；//调用f时将创建形参对象x，并调用A的拷贝

    //构造函数，用对象a对其初始化。

- 把对象作为函数的返回值时

例如：**A f()**

{ A a;

.....

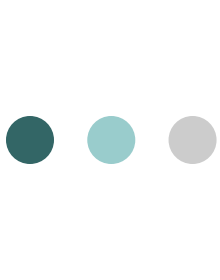
return a; //创建一个A类的临时对象，并调用A的

//拷贝构造函数，用对象a对其初始化。

}

隐式拷贝构造函数

- 如果程序中没有为类提供拷贝构造函数，则编译器将会为其生成一个隐式拷贝构造函数。
- 隐式拷贝构造函数将逐个成员拷贝初始化：
  - 对于普通成员：它采用通常的初始化操作；
  - 对于成员对象：则调用成员对象类的拷贝构造函数来实现成员对象的初始化 。



```
class A
{       int x,y;
    public:
        A() { x=y=0; }
        void inc()
        {  x++;
           y++;
        }
        ..... //其中没有定义拷贝构造函数。
};

A a1;
a1.inc();
A a2(a1); //a2.x初始化为1(a1.x)；a2.y初始化为1(a1.y)
```

自定义拷贝构造函数

- 一般情况下，编译程序提供的隐式拷贝构造函数的行为足以满足要求，类中不需要自定义拷贝构造函数。不过，自定义拷贝构造函数可以对初始化的行为进行控制。

例如：

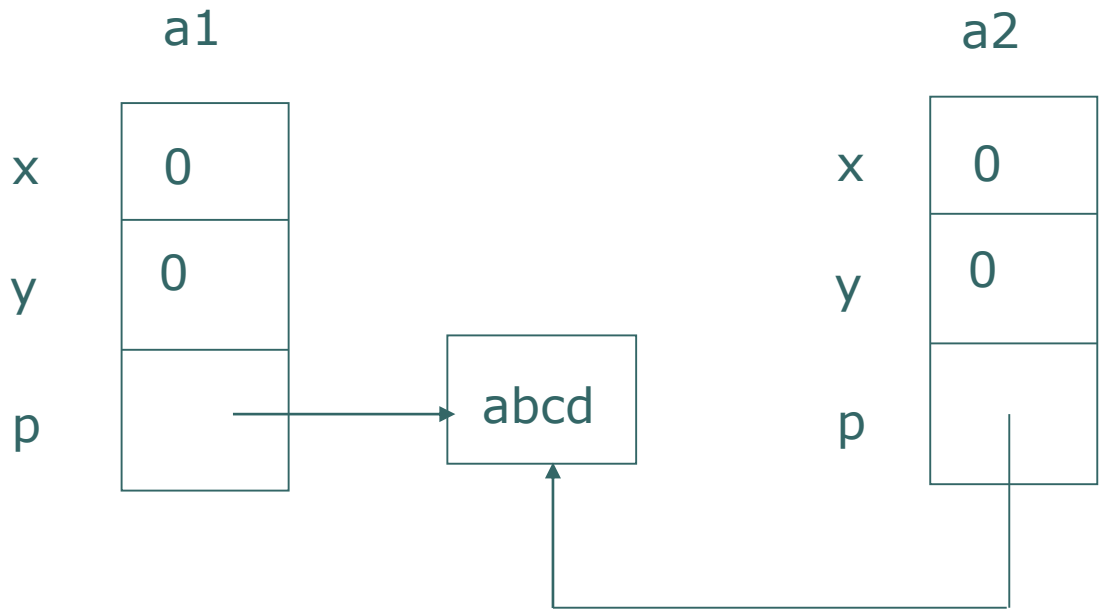
```
class A
{    int  x, y;
  public:
     A() { x = y = 0; }
     A(const A& a)
     { x = a.x+1;
       y = a.y+1; }
};
```

A a1; //a1初始化为：a1.x = 0，a1.y = 0

A a2(a1); //a2初始化为：a2.x = 1，a2.y = 1

```
class A
{       int x, y;
        char *p;
   public:
        A(char *str)
        { x = 0;  y = 0;
          p = new char[strlen(str)+1];
          strcpy(p, str);
        }
        ~A() { delete [] p; }
};
......
A a1("abcd");
A a2(a1);
```

- 在有些情况下必须要自定义拷贝构造函数，否则，将会产生设计者未意识到的严重的程序错误。



- 系统提供的隐式拷贝构造函数将会使得a1和a2的成员指针p指向同一块内存区域。
- 它带来的问题是：
  - 如果对一个对象操作之后修改了这块空间的内容，则另一个对象将会受到影响。
  - 当对象a1和a2消亡时，将会分别去调用它们的析构函数，这会使得同一块内存区域将被归还两次，从而导致程序运行异常。

- 解决上面问题的办法是在类A中显式定义一个拷贝构造函数

```
A::A(const A& a)
{  x = a.x;
   y = a.y;
   p = new char[strlen(a.p)+1];
   strcpy(p,a.p);
}
```

- 当类定义包含成员对象时，成员对象的拷贝初始化可由成员对象类的拷贝构造函数来实现。
- 系统提供的隐式拷贝构造函数调用成员对象的拷贝构造函数，而自定义的拷贝构造函数不会去调用成员对象的拷贝构造函数。需要在成员初始化表中显式指出。

例如

```
class A
{ … };
class B
{  int z;
    A a;
    public:
        B();
        B(const B& b): a(b.a) // 调用A的拷贝构造函数用b.a
        { z = b.z;}                    // 对a进行初始化
}
```

析构函数

- 在类中可以定义一个特殊的成员函数：析构函数。它的名字为“~<类名>”，没有返回类型、不带参数。

  例如：
```
class String
{     int len;
      char *str;
   public:
      String(char* s);
      ~String();
}
```

- 一个对象消亡时，系统在收回它的内存空间之前，将会自动调用析构函数。可以在析构函数中完成对象被删除前的一些清理工作（如：归还对象额外申请的资源等）。

```
class String
{      int len;
       char *str;
    public:
       String(char* s)
         {  len = strlen(s);
            str = new char[len+1];
            strcpy(str, s);
         }
       ~String()
         {  delete[] str;
         }
};
void f()
{  String s("abcd"); //调用s的构造函数
    ......
} //调用s的析构函数
```

注意：系统为对象s分配的内存空间只包含len和str本身所需的空间，str所指向的空间不由系统分配，而是由对象自己处理！

const 成员函数

- 在程序运行的不同时刻，一个对象可能会处于不同的状态，其数据成员具有不同的值。
- 可以把类中的成员函数分成两类：
  - 获取对象状态，只获取数据值，不改变数据值。
  - 改变对象状态，改变数据值。

例如 **class Date**

```
{  public:
      void set(int y, int m, int d) //改变对象状态
      int get_day(); //获取对象状态
       int get_month(); //获取对象状态
       int get_year(); //获取对象状态
};
```

- 为了防止在获取对象状态的成员函数中改变数据成员的值，可以把它们说明成const成员函数。

例如：

```
class A
{    int x;
     char *p;
  public:
     void f() const
     { x = 10; //Error
       p = new char[20]; //Error
       strcpy(p,“ABCD”); //不改变p的值，编译器OK
     }
};
```

- 给成员函数加上const修饰符还有一个作用：限制常量对象所能进行的操作。

例如：

```
class A
{     int x, y;
  public:
     void f() const { … … }
     void g() { … … }
};
const  A a; // 只能调用a的const函数
a.f(); // OK
a.g(); // Error
```

- const成员函数放在类外定义时，函数声明和定义都要加上const。

例如：

```
class A
{    … …
     void f() const;
};

void A::f() const // 定义
{    … …
}
```

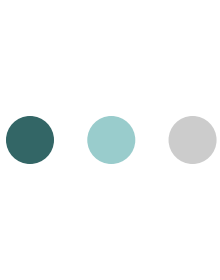
静态成员

- 不同对象需要共享数据。采用全局变量来表示共享数据违背数据抽象与封装原则，数据缺乏保护。静态成员为同一类对象之间的数据共享提供了一种较好的途径。
  - 静态数据成员
  - 静态成员函数

- 静态数据成员对所有对象只有一个拷贝。在类的外部给出它们的定义，进行初始化。

例如：

```
class A
{    int x,y;
     static int shared;  //静态数据成员说明
 public:
     A() { x = y = 0; }
     void increase_all() { x++; y++; shared++; }
     int sum_all() const { return x+y+shared; }
     static int get_shared()
     { return shared; }
};
int A::shared=0;  //静态数据成员的定义
```



又如： **A a1,a2;**

shared: 0

	a1		a2
a1.x:	0	**a2.x:**	0
a1.y:	0	**a2.y:**	0

```
a1.increase_all();
cout << a2.get_shared() << ',' << a2.sum_all() <<
endl;  //输出：1,1
```

- 静态成员函数只能访问静态成员，没有隐藏的this指针。

例如：

```
class A
{    int x,y;
     static int shared;  //静态数据成员说明
 public:
     A() { x = y = 0; }
     static  int  set_shared( int i )
     { shared = i; }
     static  int  get_shared()
     { return shared; }
};
```

- 静态成员访问
  - 对象访问

  例如：A a;

  a.set_shared(10);
  - 类名受限访问

  例如：A::get_shared();

例： 实现对某类对象的计数

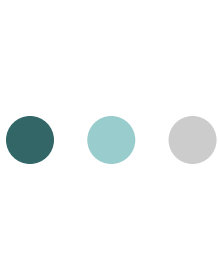
```
class A
{      static int obj_count;
        ……
   public:
       A() { obj_count++; }
       ~A() { obj_count--; }
       static int get_num_of_objects()
       { return obj_count;
        }
       ……
};
int A::obj_count=0;
……
cout << A::get_num_of_objects() << endl;
```

对数据成员的访问效率问题

- 根据数据保护的要求，通常把数据成员说明成private。对private数据成员的访问通常要通过该类的public成员函数来进行，而这有时会降低对数据成员的访问效率。

例如：

```
class Point
{ public:
      Point(double xi, double yi) { x = xi; y = yi; }
      double GetX() const { return x; }
      double GetY() const { return y; }
  private:
      double x, y;
};
```



```
double Distance(const Point& a,const Point& b)
{    double dx=a.GetX() - b.GetX(); //效率不高
     double dy=a.GetY() - b.GetY(); //效率不高
     return sqrt(dx*dx+dy*dy);
}

int main()
{  Point p1(3.0, 5.0), p2(4.0, 6.0);
   double d=Distance(p1, p2);
   cout<<"The distance is "<< d << endl;
   return 0;
}
```

- 一种解决方案：把Distance作为成员函数来定义

```
class Point
{ public:
      Point(double xi, double yi) { x = xi; x = yi; }
      double GetX() const { return x; }
      double GetY() const { return y; }
  private:
      double x, y;
      double Distance(const Point& pt)
      { double dx=x - pt.x; //效率高
        double dy=y - pt.y; //效率高
        return sqrt(dx*dx+dy*dy);
      }
};

A a(1,2), b(3,4);
cout << a.Distance(b);
```

友元

- 为了提高在类的外部对类的数据成员的访问效率，在C++中，可以指定与一个类密切相关的、又不适合作为该类成员的程序实体（某些全局函数、某些其它类或某些其它类的某些成员函数）可以直接访问该类的private和protected成员。这些程序实体称为该类的友元。

例如：
```
class A
{   ......
    friend void func();  //友元函数
    friend class B;  //友元类
    friend void C::f();  //友元类成员函数
};
```

```
class Point
{ public:
      Point(double xi, double yi) { x = xi; x = yi; }
      double GetX() const { return x; }
      double GetY() const { return y; }
  private:
      double x, y;
  friend double Distance(const Point& a, const Point& b);
};
double Distance(const Point& a, const Point& b)
{    double dx=a.x - b.x; //效率高
      double dy=a.y - b.y; //效率高
      return sqrt(dx*dx+dy*dy);
}
```

关于友元的几点说明

- 友元关系不具有对称性。

  例如：假设B是A的友元，如果没有显式指出A是B的友元，则A不是B的友元。

- 友元也不具有传递性。

  例如：假设B是A的友元、C是B的友元，如果没有显式指出C是A的友元，则C不是A的友元。

- 友元是数据保护和数据访问效率之间的一种折衷方案。

- 通常把与类密切相关的、又不适合作为该类成员的程序实体说明成该类的友元。