数据抽象——类

面向对象的基本概念

- 对象：表示要研究的任何事物
  - 是由数据及其操作所构成的封装体
- 类：是对象的模板
  - 描述了一组具有相同属性和操作的对象
- 方法：指定义于某一特定类上的操作与法则。
- 消息：调用对象的操作，由对象名、消息名和实际变元构成
- 通信：指对象之间的消息传递，是引起面向对象程序进行计算的唯一方式

- 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）。

对象举例

- 学生：张三
  - 对象标识：对象名　student_1
  - 对象属性：
    - name= 张三
    - number=00123456
    - age　= 20
    - major=Physics
    - ……..
  - 对象操作：
    - SelectClass("class_math");
    - …….

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。
- 对栈只能实施两种操作：进栈（增加一个元素）和退栈（删除一个元素），并且这两个操作必须在栈的同一端（称为栈顶）进行。
- 后进先出（Last In First Out，简称LIFO）是栈的一个重要性质。

“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;
//对st进行初始化。
st.top = -1;
//把12放进栈。
st.top++;
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

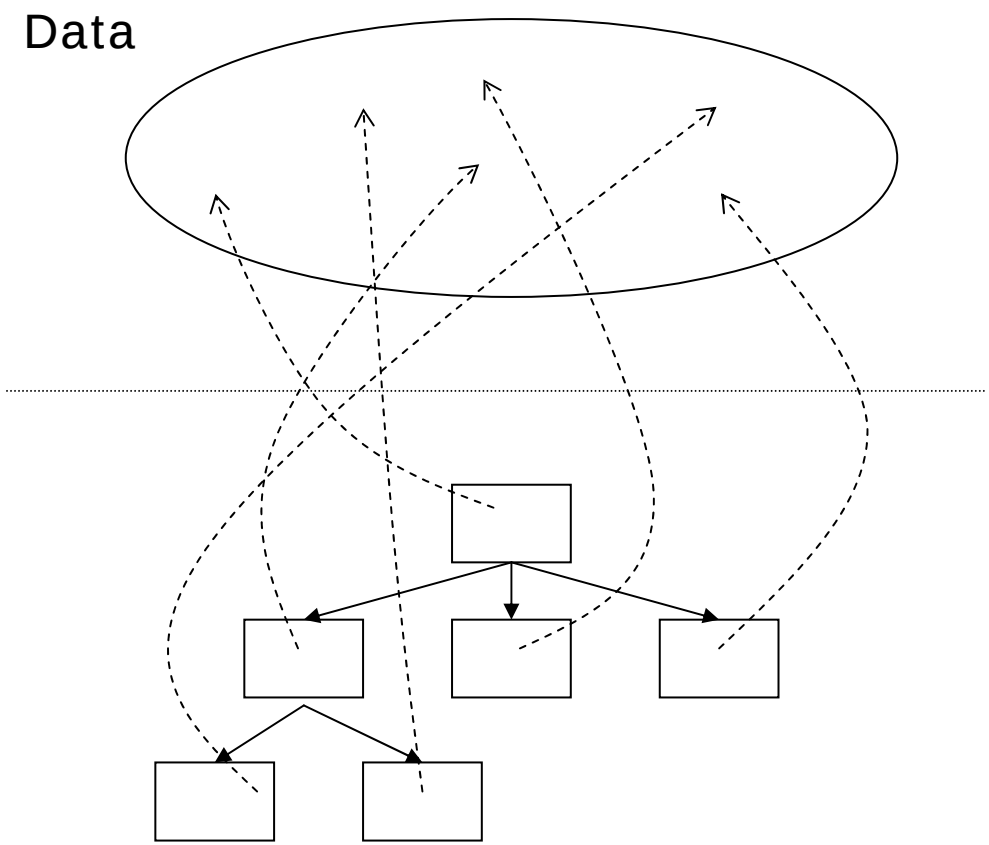
```
#include <iostream>
using namespace std;
//定义栈数据类型
const int STACK_SIZE=100;
class Stack
{  int top;
   int buffer[STACK_SIZE];
 public:
   Stack() { top = -1; }
```

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

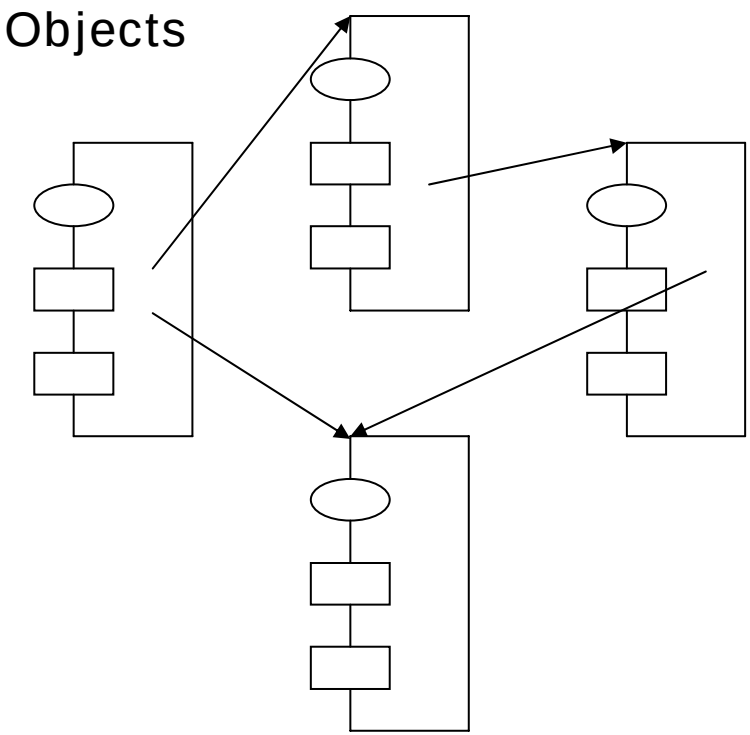
```
//使用栈类型数据
Stack st;  //自动地去调用st.Stack()对st进行初始化。
int x;
st.push(12);  //把12放进栈st。
st.pop(x);  //把栈顶元素退栈并存入变量x。
st.top = -1;  //Error
st.top++;  //Error
st.buffer[st.top] = 12;  //Error
```

- 上述两种解决方案存在下面的几点不同：
  - 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。

为什么要面向对象？



- 过程式程序设计
  - 以功能为中心，强调功能（过程）抽象。
  - 实现了操作的封装，但数据是公开的，数据与对数据的操作分离，数据缺乏保护。
  - 功能易变，程序维护困难。
  - 功能往往针对某个程序而设计，这使得程序功能难以复用。
  - 基于功能分解的解题方式与问题空间缺乏对应。



- 面向对象程序设计
  - 以数据为中心，强调数据抽象。
  - 数据与操作合而为一，实现了数据的封装，加强了数据的保护。
  - 对象相对稳定，有利于程序维护。
  - 对象往往具有通用性，使得程序容易复用。
  - 基于对象/类的解题方式与问题空间有很好的对应。

类和对象

- 对象/类(Object&Class)
  - 对象是由数据（数据成员、成员变量、实例变量、对象的局部变量等）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）描述了对象的特征（数据和操作），它属于类型的范畴 。

类

- 对象构成了面向对象程序的基本计算单位，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

- 类定义中的数据成员描述了类的对象所包含的数据的类型，数据成员的说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
};
```

- 说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）。在说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，这里是声明。
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 &a2; //OK
  B &b2; //OK
   static B b3; //OK
};
```

成员函数

- 成员函数描述了对类定义中的数据成员所能实施的操作。
- 成员函数的定义可以放在类定义中，例如：

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

- 成员函数的定义也可以放在类定义外，例如：

```
class A
{    ...
     void f();
};
void A::f() { ... } //需要用类名受限。
```

- 类成员函数名是可以重载的（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);
   ......
};
```

练习：定义一个时间类

- 要求能够表示时、分、秒
- 定义一个setTime()函数，设置时间
- 定义一个print()函数，打印时间

解答

```
class Time
{
        int hour, minute, second;
public:
        void setTime(int h, int m, int s) //成员函数
        {       hour = h;
                minute= m;
                second = s;
        }
        void print() //成员函数
        {
            cout << “Time is: ” << hour << “:” << minute
   << “:” <<second;
        }
};
```

类成员的访问控制

- 在C++的类定义中，可以用访问控制修饰符public，private或protected来描述对类成员的访问限制。
- Public: 访问不受限制，在程序任何地方都可以访问
- Private: 只能被本类或友元访问
- Protected:只能被本类、派生类和友元访问

例子

```
class A
{
 frind: calss B
 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();
};
```

- class C: public A
- B是A的友元，C是A的派生类
- 问：B可以访问哪些数据和函数？C可以访问哪些数据和函数？

- 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:  //可以有多个访问控制说明
    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;
   }
}
```

- 对比前面栈的实现，可以看到数据封装带来的好处：数据表示的变化不会影响使用者，这是因为类的对外接口没有发生变化。
- 类作用域：类定义构成的作用域，其中的标志符局部于类定义，它们可以与类定义外的全局标志符或其他类定义中的标志符相同。
  - 在类定义外使用类定义中的标志符时，需要通过对象名受限或类名受限
  - 在类定义中使用与局部标志符同名的全局标志符时，要在全局标志符前面加上全局域分辨符(::)

练习

- 下面说法错误的是：
  - A 对象是由数据及其操作所构成的封装体
  - B 类描述了一组具有不同属性和操作的对象
  - C 方法是定义于某一特定类上的操作与法则。
  - D 通信是引起面向对象程序进行计算的唯一方式
- 用C++的数组来实现一个栈类时，数组应该定义为___，push和pop方法应该定义为___。
  - A private
  - B protected
  - C public
  - D 都可以

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。
- 对象有那些数据成员以及这些成员的存储安排由类中的数据成员描述决定。

对象的创建和标识

- 直接方式
  - 通过在程序中定义一个类型为类的变量来实现的，其格式与普通变量的定义相同。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]; //创建由对象数组表示的100个A类对象。
```

  - 直接方式创建的对象通过变量名（对象名）来标识和访问。

- 间接方式（动态对象）
  - 在程序运行时刻，通过new操作来创建对象。
  - 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操作来撤消（使之消亡）。
  - 动态对象通过指针来标识和访问。

单个动态对象的创建与撤消

```
A *p;
p = new A;  // 创建一个A类的动态对象。
……
delete p;   // 撤消p所指向的动态对象。
```

动态对象数组的创建与撤消

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。

- 对于动态对象数组，需注意下面的两点：
  - 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

- 对于创建的一个对象，需要通过向它发送消息（调用对象类中定义的成员函数）来对它进行操作，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;
}
```

- 例：通过对象来访问类的成员时要受到类成员访问控制的限制

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

问题：下列的调用是否正确？

```
A a;
(1) a.f();
(2) a.x = 1;
(3) a.g();
(4) a.y = 1;
(5) a.h();
```

（1）正确，（2）-（5）错误

- 可以对对象进行赋值

```
Date yesterday,today,some_day;
some_day = yesterday; //把对象yesterday的数据成员分
                      //别赋值给对象some_day的相应数据成员。
```

- 取对象地址

```
Date *p_date;
p_date = &today; //把对象today的地址赋值给对象指针
                 //p_date。
```

- 把对象作为实参传给函数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，该函数需要一个Date类
                //的对象作为参数，返回值为一个Date类的对象。
some_day = f(yesterday); //调用函数f，把对象yesterday
                         //作为实参。返回值对象赋给对象
    some_day2。
```

对象的初始化

- 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++提供了一种对象初始化的机制：构造函数(Constructor)。构造函数是类的特殊成员函数，它的名字与类名相同、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分配内存空间，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(Default Constructor) 。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。也可写成：A a1=A(); 但不能写成：A a1();
A a2(1); //调用A(int i)。也可写成：A a2=A(1); 或 A a2=1;
A a3("abcd"); //调A(char *)。也可写成：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只能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，也不能采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
      }
};
```

- 对于常量数据成员和引用数据成员，可以在定义构造函数时，在函数头和函数体之间加入一个成员初始化表来对它们进行初始化。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 。

练习

```
class A
{ const x,y;
   int z;
  public:
      A(): x(1),y(2)
     { z=x+y;};
      A(int i): x(1),y(x)
      { z=x+y+i};
    A(char *p): y(x),x(2)
      { z=x+y};
};
```

- 执行以下语句后，z的值？
- (1)A a1;

  a1.z=?
- (2)A a2(3);

  a2.z=?
- (3)A a3(“xyz”);

  a3.z=?

a1.z=3, a2.z=5, a3.z=4

析构函数（ Destructors ）

- 在类中可以定义一个特殊的成员函数：析构函数(Destructor)，它的名字为“~<类名>”，没有返回类型、不带参数、不能被重载。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; //归还资源
     }
};
String s("abcd");
```

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

例：字符串类的实现 P209

```
class String
{ char * str;
  public:
   String() {…} //构造函数
   String(const char* p){…}//自定义构造函数
   ~String(){...}//析构函数
   int length() {...} //取字符串长度
   char &char_at(int i){...} //返回指定位置的字符的引用
   const char *get_str(){return str};//返回字符串指针
   String &append(const cahr *p)
   { char *p1=new char[strlen(str)+strlen(p)+1];//申请资源
     strcpy(p1,str);
     strcat(p1,p);
     delete []str;
     str=p1;
     return *this;
   }
  String &append(const String &s) {return append(s.get_str())};
  String &copy(const String &s) {…}
  int compare(const String &s){…}
  …
}
```

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。
- (1)使用默认构造函数
- (2)使用非默认构造函数：要写在成员初始化表中
- 例如：

```
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，再执行本身类型的构造函数。
- 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，会使用一个对象去初始化另一个同类型的对象：
  - 定义对象时，例如：

```
A a1;
A a2(a1);  //用对象a1对对象a2进行初始化，也可写成：
           //A a2=a1; 或：A a2=A(a1);
```

  - 把对象作为值参数传给函数时，例如：

```
void f(A x);
A a;
f(a);  //调用时将创建形参对象x，用对象a对其初始化。
```

  - 把对象作为返回值时，例如：

```
A f()
{    A a;
     ......
     return a; //创建一个A类的临时对象，用对象a对其初始化。
}
.....
...f()...
```

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。对象的状态由对象的数据成员的值来体现。
- 可以把类中的成员函数分成两类：
  - 获取对象状态
  - 改变对象状态

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对象的状态，可以把它们说明成const成员函数。const成员函数不能改变对象的状态（数据成员的值）。例如：

```
class A
{    int x;
     char *p;
   public:
     ......
     void f() const
     { x = 10; //Error
        p = new char[20]; //Error
        strcpy(p,"ABCD"); //没有改变p的值，编译程序认为OK!
     }
};
```

- 给成员函数加上const修饰符还有一个作用：描述对常量对象所能进行的操作,（对常量对象的成员函数调用合法性进行判断）。即：const对象只能调用const成员函数。例如：

```
class Date
{      int d,m,y;
  public:
       Date();
       Date(int year, int month, int day);
       int get_day() const { return d; }
       int get_month() const { return m; }
       int get_year() const { return y; }
       int set (int year, int month, int day)
       {   y = year; m = month; d = day;
       }
        ...
};
void main()
{ const Date  today(2006,4,3);
   cout << today.get_day() <<  today.get_month()
          << today.get_year();
   today.set (2005,4,13); //Error
}
```

静态成员

- 属于一个类的不同对象有时需要共享数据。
- 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。
- 静态数据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。

在类中，可以把成员说明成静态的。例如：

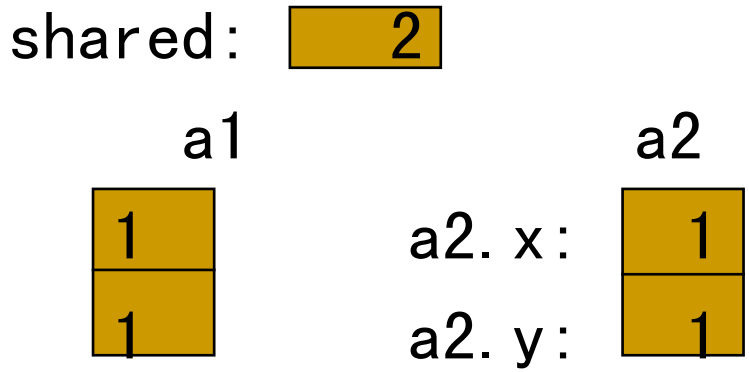
```
class A
{       int x,y;
        static int shared;  //静态数据成员说明
      public:
        A() { x = y = 0; }
        void increase_all() { x++; y++; shared++; }
        int sum() const { return x+y+shared; }
        static int get_shared() //静态成员函数
        { return shared; }
        ......
};
int A::shared=0;  //静态数据成员的定义，必须在类外部给出
A a1,a2;

a1.increase_all();

a2.increase_all();
cout << a2.get_shared() << ',' << a2.sum() << endl;
     //输出：2,4
```

- 类的静态数据成员对该类的所有对象只有一个拷贝。例如：

```
A a1,a2;
```




```
a1.increase_all();
a2.increase_all();
cout << a2.get_shared() << ',' << a2.sum_all()
  << endl;  //输出：2,4
```

- 静态成员函数：只能访问静态成员（包括静态的数据成员和静态的成员函数）。
- 静态成员的访问
  - (1)通过对象访问

```
A a;
a.set_shared();
```

  - (2) 通过类名受限来访问。不必创建对象。例如：

```
cout << A::get_shared();
```

例子：对象进行计数

```
class A
{      static int obj_count;
       ……
   public:
       A() { obj_count++; }
       ~A() { obj_count--; }
       static int get_num_of_objects()
       { return obj_count;
        }
       ……
};
```

下面程序段输出结果是?

```
int A::obj_count=0;
A a;
A b;
A c;
cout << A::get_num_of_objects() << endl;
```

答案 3

友元

- 在C++中，为了提高在类的外部对类的数据成员的访问效率，可以指定某些全局函数（友元函数）、某些其它类（友元类）或某些其它类的某些成员函数（友元类成员函数）可以直接访问该类的private和protected成员。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

- 友元关系具有不对称性。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。
- 通常把与一个类密切相关的、又不适合作为该类成员的程序实体说明成该类的友元。

小结

- 类
  - 概念：类、对象、消息、通信
  - 什么是面向对象程序设计，好处是什么？
  - 如何定义一个简单的类，如日期类，时间类
  - 类成员的三种访问控制
  - 友元的访问权限
- 对象
  - 对象的构造和初始化
  - 构造函数和析构函数
  - Const成员
  - 静态成员