C++语言程序设计
第三章 类和对象的使用

王焦乐

http://faculty.hitsz.edu.cn/jlwang

哈尔滨工业大学（深圳）

机电工程与自动化学院

本章主要内容

- 类对象的初始化
- 对象数组
- 对象指针
- 公用数据的保护
- 对象的动态建立和释放
- 对象的赋值和复制
- 不同对象间实现数据共享
- 允许访问私有数据的“朋友”
- 类模板

0、类对象的初始化

- 变量初始化：对变量赋值

```
int a=10;
```

- 错误的类数据成员初始化

```
class Time
{hour=0;minute=0;sec=0;};
```

- 所有成员都是公用的，可以在定义对象时进行初始化

```
class Time
{
  public:  //声明公用成员
    int hour;
    int minute;
    int sec;
};

Time t1={14,56,30};
```

- private 或 protected 需要用成员函数进行初始化
- 若存在大量数据成员，需要更加高效的方法进行初始化

0、类对象的初始化

用构造函数实现数据成员的初始化

- 构造函数（constructor）进行初始化
- 建立对象时自动执行，不需要用户调用
- 函数名与类名相同
- 不具有类型，无返回值

```
#include <iostream>
using namespace std;
class Time
{
public:
  Time()
  {
    hour = 0;
    minute = 0;
    sec = 0;
  }
  void set_time();
  void show_time();
private:
  int hour;
  int minute;
  int sec;
};
```

```
void Time::set_time()
{
  cin >> hour;
  cin >> minute;
  cin >> sec;
}

void Time::show_time()
{
  cout << hour << ":"
       << minute << ":"
       << sec << endl;
}
```

```
int main()
{
  Time t1;
  t1.set_time();
  t1.show_time();
  Time t2;
  t2.show_time();
  return 0;
}
```

0、类对象的初始化

用构造函数实现数据成员的初始化

- 构造函数（constructor）进行初始化
- 建立对象时自动执行，不需要用户调用
- 函数名与类名相同
- 不具有类型，无返回值
- 类内声明类外定义

```
class Time
{
public:
  Time();
  void set_time();
  void show_time();
private:
  int hour;
  int minute;
  int sec;
};
```

```
Time::Time()
{
  hour = 0;
  minute = 0;
  sec = 0;
}
void Time::set_time()
{
  cin >> hour;
  cin >> minute;
  cin >> sec;
}
```

```
void Time::show_time()
{
  cout << hour << ":"
       << minute << ":"
       << sec << endl;
}
```

0、类对象的初始化

用构造函数实现数据成员的初始化

- 构造函数（constructor）进行初始化
- 建立对象时自动执行，不需要用户调用
- 函数名与类名相同
- 不具有类型，无返回值
- 类内声明类外定义

- 不需要也不能被用户调用
- 也可以做其他操作
- 默认构造函数

0、类对象的初始化

带参数的构造函数

- 为不同对象赋不同初值

```
// 构造函数首部一般格式
构造函数名(类型 1 形参 1, 类型 2 形参 2, ...)

// 定义对象一般格式
类名 对象名(实参 1, 实参 2, ...);
```

```
#include <iostream>
using namespace std;
class Box
{
public:
  Box(int,int,int);
  int volume();
private:
  int height;
  int width;
  int length;
};
```

```
int main()
{
  Box box1(12, 25, 30);
  cout << "The volume of box1 is "
       << box1.volume() << endl;
  Box box2(15, 30, 21);
  cout << "The volume of box1 is "
       << box2.volume() << endl;
  return 0;
}
```

```
Box::Box(int w, int h, int len)
{
  height = h;
  width = w;
  length = len;
}

int Box::volume()
{
  return(height * width * length);
}
```

0、类对象的初始化

□ 参数初始化列表

```
// 带有初始化表的构造函数
构造函数名(类型 1 形参 1, 类型 2 形参 2):成员名 1(形参 1),成员名 2(形参 2){}
// 定义对象
类名 对象名(实参 1, 实参 2);
```

```
class Student
{
  public:
    Student(int n, char s,nam[]):num(n),sex(s)
    {strcpy(name,nam);}
  private:
    int num;
    char sex;
    char name[20];
}

Student stud1(10101,'m',"Wang_Li");
```

0、类对象的初始化

□ 构造函数重载

```
#include <iostream>
using namespace std;
class Box
{
public:
  Box();
  Box(int h, int w, int len)
      :height(h), width(w), length(len) { }
  int volume();
private:
  int height;
  int width;
  int length;
};
```

```
Box::Box()
{
  height=10;
  width=10;
  length=10;
}

int Box::volume()
{
  return(height*width*length);
}
```

```
int main()
{
  Box box1;
  cout<<"The volume of box1 is "
      <<box1.volume()<<endl;
  Box box2(15,30,25);
  cout<<"The volume of box2 is "
      <<box2.volume()<<endl;
  return 0;
}
```

0、类对象的初始化

□ 构造函数重载

（1）不带形参的构造函数为**默认构造函数（default constructor）**

（2）每个类只有一个默认构造函数，如果是系统自动给的默认构造函数，其函数体是空的。

```
Box box1;
```

错误

```
Box box1();
```

（3）虽然每个类可以包含多个构造函数，但是创建对象时，系统仅执行其中一个。

0、类对象的初始化

□ 构造函数使用默认参数

```
#include <iostream>
using namespace std;
class Box
{
public:
  Box(int h=10,int w=10,
              int len=10){}
  int volume();

private:
  int height;
  int width;
  int length;
};
```

```
Box::Box(int h,int w,int len)
{
  height = h;
  width = w;
  length = len;
}

int Box::volume()
{
  return(height*width*length);
}
```

```
int main()
{
  Box box1;
  cout<<"The volume of box1 is "
      <<box1.volume()<<endl;
  Box box2(15);
  cout<<"The volume of box2 is "
      <<box2.volume()<<endl;
  Box box3(15, 30);
  cout<<"The volume of box3 is "
      <<box3.volume()<<endl;
  Box box4(15, 30, 20);
  cout<<"The volume of box4 is "
      <<box4.volume()<<endl;
  return 0;
}
```

0、类对象的初始化

构造函数使用默认参数

(1) 如果在类外定义构造函数，应该在声明构造函数时指定默认参数值，在定义函数时可以不再指定默认参数值。

```
Box(int =10,int =10,int =10);
```

(2) 在声明构造函数时，形参名可以省略例如：

(3) 全部形参都指定了默认值的构造函数也属于默认构造函数。为了避免歧义，不允许同时定义不带形参的构造函数和全部形参都指定默认值的构造函数。

```
Box();
Box(int h = 10, int w = 10, int len = 10);

Box box1;
```

歧义

(4) 同样为了避免歧义性，如定义了全部形参带默认值的构造函数后，不能再定义重载构造函数。反之亦然。

```
Box();
Box(int ,int =10,int =10);
Box(int, int);
// 若定义对象:
Box box1;          // 正确，调用第一个
Box box2(15);      // 正确，调用第二个
Box box3(15,30);   // 错误，不知调用哪一个
```

歧义

0、类对象的初始化

- 构造函数实现初始化方法归纳

1. 在类中定义构造函数的函数体中
2. 用带参数的构造函数
3. 用初始化列表
4. 使用默认参数
5. 重载构造函数

```
public:
  Time()
  {
    hour = 0;
    minute = 0;
    sec = 0;
  }
```

```
  Box(int h, int w, int l)
  {
    height = h;
    width = w;
    length = l;
  }

Box box1(12,23,30);
```

```
Box(int h, int w, int l):height(h),width(w),length(l){};
```

```
Box(int h=10, int w=10, int l=10):height(h),width(w),length(l){};
```

0、类对象的初始化

析构函数

- ~类名(){}
- 处理对象空间的回收
- 特殊的成员函数，不需要用户调用，释放对象时自动执行
- 构造函数没有返回值
- 没有函数参数
- 默认析构函数

```
int main()
{
  Student stud1(10010,"Wang_li",'f');
  stud1.display();
  Student stud2(10011,"Zhang_fun",'m');
  stud2.display();
  return 0;
}
```

程序运行结果如下：

```
Constructor called.
num: 10010
name: Wang_li
sex: f

Constructor called.
num: 10011
name: Zhang_fun
sex: m

Destructor called.
Destructor called.
```

```
#include <iostream>
#include <string>
using namespace std;
class Student
{
public:
  Student(int n, string nam, char s)
  {
    num = n;
    name = nam;
    sex = s;
    cout << "Constructor called." << endl;
  }

  ~Student()
  {
    cout << "Destructor called." << endl;
  }

  void display()
  {
    cout << "num:" << num << endl;
    cout << "name:" << name << endl;
    cout << "sex:" << sex << endl
         << endl;
  }

private:
  int num;
  string name;
  char sex;
};
```

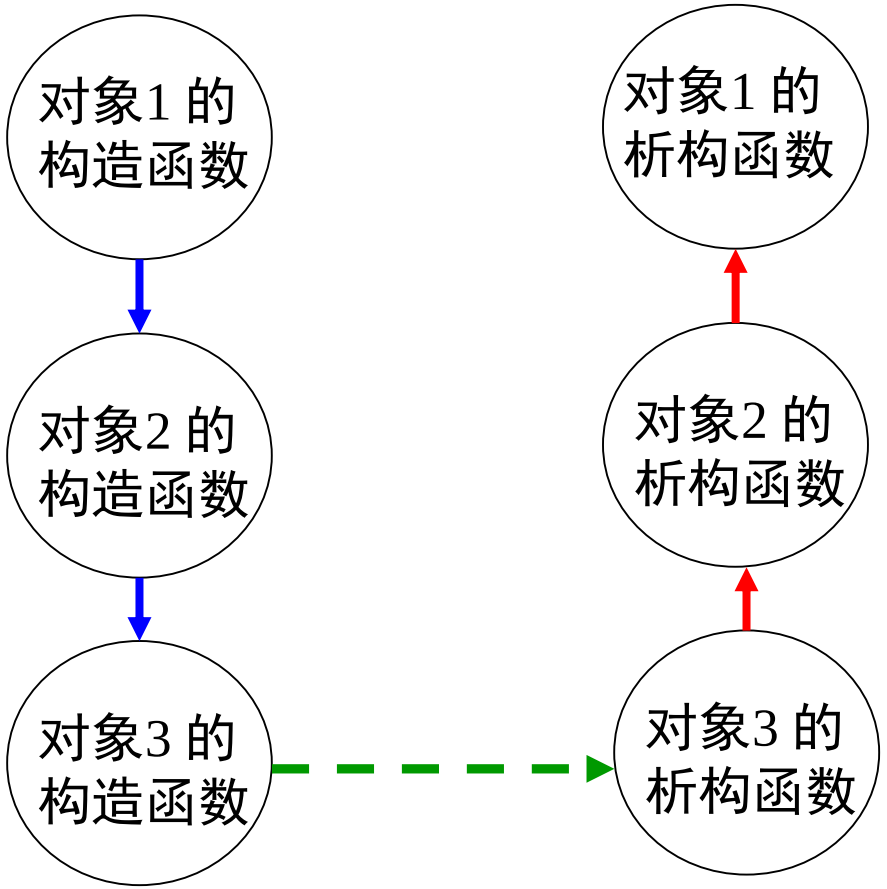
0、类对象的初始化

- 调用构造函数与析构函数的顺序
  - 先构造的后析构，后构造的先析构
  - 栈：先进后出

(1)全局对象（在所有函数之外定义的对象），在所有函数（包括主函数）执行前调用构造。当主函数结束或执行 exit 函数时，调用析构。

(2)局部对象（在函数内定义对象），在创建对象时调用构造函数。如多次调用对象所在的函数，则每次创建对象时都调用构造函数。在函数调用结束时调用析构函数。

(3)静态static局部对象，则在第一次调用该函数建立对象时调用构造函数，但要在主函数结束或调用 exit 函数时才调用析构



1、对象数组

- 类是一种特殊的数据类型，可以定义对象数组。
- 在一个对象数组中各个元素都是同类对象例如一个班级有50个学生，每个学生具有学号、年龄、成绩等属性，可以为这个班级建立一个对象数组，数组包括了50个元素：

```
Student  stud[50];
```

可以这样建立构造函数：

```
Student :: Student ( int=1001,int=18,int=60 );
```

```
Student stud[n]={
        Student ( 实参1,实参2,实参3 ),
        … …
        Student ( 实参1,实参2,实参3 ),
        };
```

2、对象指针

□ 指向对象的指针

类名 * 对象指针名

Time *pt; **// 定义pt是指向Time类对象的指针**

Time t1; **// 定义Time类对象t1**

pt = &t1; **// 将对象t1的地址赋予pt**

程序在此之后就可以用指针变量访问对象的成员。

(*pt).hour

pt->hour

(*pt). show_time()

pt->show_time()

```
class Time
{
public:
  Time()
  {
    hour = 0;
    minute = 0;
    sec = 0;
  }
  void get_time();
private:
  int hour;
  int minute;
  int sec;
};

void Time::show_time()
{
  cout<<hour<<":"<<minute<<":"<<sec<<endl;
}
```

2、对象指针

□ 指向数据成员的指针

①定义数据成员的指针变量

数据类型 * 指针变量名

②计算公有数据成员的地址

&对象名.成员名

这里的数据类型是数据成员的数据类型。

例：Time t1;

```
int * p1; // 定义一个指向整型数据的指针变量
p1 = & t1.hour; // 假定hour是公有成员
cout<< *p1 << endl;
```

2、对象指针

□ 向普通函数的指针变量

数据类型 (***指针变量名**)(**形参表**);

□ 指向成员函数的指针

①定义指向成员函数的指针变量

数据类型 (**类名**::***指针变量名**)(**形参表**);

数据类型是成员函数的类型。

类名：对象所属的类

变量名：按标识符取名

形参表：成员函数形参表(形参个数、类型)

②取成员函数的地址

&类名::成员函数名

③给指针变量赋初值

指针变量名= **& 类名**::成员函数名；

④用指针变量调用成员函数

(对象名.***指针变量名**)(**实参表**);

对象名：是指定调用成员函数的对象。

*：明确其后的是一个指针变量。

实参表：与成员函数的形参表对应，如无形参可省略

```
int main() {
  Time t1(10,13,56);
  int *p1=&t1.hour;   //   定义指向成员的指针p1
  cout<<*p1<<endl;
  t1.get_time();       //    调用成员函数
  Time *p2=&t1;        //    定义指向对象t1的指针p2
  p2->get_time();      //    用对象指针调用成员函数
  void (Time::*p3)(); //   定义指向成员函数的指针
  p3=&Time::get_time; //  给成员函数的指针赋值
  (t1.*p3)();  //  用指向成员函数的指针调用成员函数
  return 0;
}
```

3、共用数据的保护

- 数据共享
  - 实参与形参、变量与其引用、数据及其指针
- **const**: 既要使数据在一定范围共享，又要保证其不被任意修改
  - 类型名 const 变量名
  - const 类型名 变量名
  - int const a　　int const &a
  - const int a　　const int &a
  - int const *p　　f(int const a)
  - const int *p　　f(int const &a)
  - int * const p　　f(int const *a)　　f(int* const a)
  - 初始化阶段
    - const类型数据必须初始化
    - int const 与 int可以相互赋值
    - 引用 与 变量
    - 指针 与 变量
  - 语句调用阶段
    - const修饰的变量不能改变值

3、共用数据的保护

□ 常对象

类名 const 对象名(实参表)

- 常成员函数是常对象对外接口

如： const Time t1(10,15,36);

t1.get_time(); // 错误，不能调用

为了访问常对象中的数据成员，要定义常成员函数：

void get_time() const

- 常成员函数可以访问，但不允许修改 常对象的数据成员

- 在常对象中要修改某个数据成员，**可变的数据成员**

格式： mutable 类型 数据成员；

在定义数据成员时加**mutable**后，将数据成员声明为可变的数据成员，就可以用声明为const的成员函数修改它的值。

3、共用数据的保护

□ 常数据成员

- 通过构造函数的参数初始化列表来初始化

格式： **const** **类型** 数据成员名

例：
```
const   int  hour;
Time :: Time( int h)
{    hour = h; ...}   //      错误
```
应该写成：
```
Time :: Time( int h ) : hour (h) {}
```

3、共用数据的保护

□ 常成员函数

数据类型 函数名（**参数表**） **const** {}

- 不能修改数据成员，不能调用**非**常成员函数
- 常对象只能调用常成员函数

数据成员	非 const 成员函数	const 成员函数
非 const 的数据成员	可以引用,也可以改变值	可以引用,但不可以改变值
const 数据成员	可以引用,但不可以改变值	可以引用,但不可以改变值
const 对象的数据成员	不允许引用和改变值	可以引用,但不可以改变值

3、共用数据的保护

□ 指向对象的常指针

- 必须在定义时对其初始化
- 程序运行中不能再修改指针的值

类名 * const 指针变量名 = 对象地址

例：Time t1(10,12,15)，t2;

Time * const p1 = & t1;

在此后，程序中不能修改p1。

例： Time * const p1 = & t2; **// 错误语句**

3、共用数据的保护

□ 指向常变量的指针变量

- 定义指向常变量的指针变量的一般形式为

  const 类型名 *指针变量名;

（1）只能用指向常变量的指针指向常变量

（2）不能通过指向常变量的指针改变变量的值

（3）形参是指向普通变量的指针，则实参应相同

```
const char c[]="boy";
const char *p1;
p1=c;
char *p2=c;   //非法
```

```
char c1='a';
const char *p;
p=&c1;
*p='b'; //非法
c1='b'; //合法
```

用指针变量作形参时形参和实参的对应关系

形 参	实 参	合法否	改变指针所指向的变量的值
指向非 const型变量的指针	非const变量的地址	合法	可以
指向非const 型变量的指针	const 变量的地址	非法	—
指向const型变量的指针	const变量的地址	合法	不可以
指向const型变量的指针	非 const 变量的地址	合法	不可以

3、共用数据的保护

- 指向常对象的指针变量

（1）只能用指向常对象的指针指向常对象

（2）不能通过指向常对象的指针改变对象

（3）指向对象的常指针 VS 指向常对象的指针

（4）指向常对象的指针最常用于形参，用于保护对象不被修改

（5）若希望调用函数时对象不改变，形参应为指向常对象的指针，实参为对象的地址

（6）指向常对象的指针变量本身可以改变

```
const Time *p=&t1;
p=&t2;
```

```
class Time;
Time t1(10,12,15);
const Time *p=&t1;
t1.hour=18;
(*p).hour=18; //非法
```

```
Time * const p;
const Time *p;
```

```
int main()
{
  void fun(const Time *);
  Time t1(10,13,56);
  fun(&t1);
  return 0;
}
void fun(const Time *p)
{
  p->hour=18;   // 错误
  cout<<p->hour<<endl;
}
```

3、共用数据的保护

□ 对象的常引用

■ 作为函数参数

引用作形参时，在函数内修改引用形参也就是修改实参变量。
如果用引用形参又不想让函数修改实参，可以使用常引用机制。
格式： **const 类名 & 形参对象名**

■ 拷贝（复制）构造函数

调用函数 → 实参的拷贝 → 调用复制构造函数 → 时间开销↑

使用常引用、常指针可以不必加实参的拷贝，提高运行效率

3、共用数据的保护

□ const 型数据的小节

表 3.3

形　式	含　义
Time const tl; 或 const Time tl;	tl 是常对象,其值在任何情况下都不能改变
void Time::fun( )const	fun 是 Time 类中的常成员函数,可以引用,但不能修改本类中的数据成员
Time * const p;	p 是指向 Time 对象的常指针,p 的值(即 p 的指向)不能改变
const Time *p;	p 是指向 Time 类常对象的指针,其指向的类对象的值不能通过指针来改变
Time &tl = t;	tl 是 Time 类对象 t 的引用,t 和 tl 指向同一段内存空间

4、对象的动态建立和释放


```
Box *pt = new Box(12,15,18);
std::cout<<pt->height;

delete pt;
```

```
class Box
{
public:
  Box(int h=10,int w=10, int len=10){}
  int volume();

private:
  int height;
  int width;
  int length;
};
```

5、对象的赋值和复制

□ 对象的赋值

- 同类对象之间可以互相赋值。这里所指的对象的值含义是对象中所有数据成员的值。

  格式　　对象1 = 对象2;

（1）对象的赋值只对数据成员操作

（2）数据成员中不能含有动态分配的数据成员。

```
int main()
 {
  Box box1(15,30,25),box2;
  cout<<"box1 体积= "<<box1.volume()<<endl;
  box2=box1;
  cout<<"box2 体积= "<<box2.volume()<<endl;
  return 0;
 }
```

运行结果如下：

```
box1 体积= 11250
box2 体积= 11250
```

5、对象的赋值和复制

对象的复制

- 对象赋值的前提是对象1和对象2是已经建立的对象。
- 复制对象是从无到有按照一个对象克隆出另一个对象。
- 创建对象必须调用构造函数，复制对象要调用复制构造函数。

复制对象有两种格式：

（1）类名 对象2(对象1);
　　按对象1复制对象2。

（2）类名 对象2=对象1,对象3=对象1,…;
　　按对象1复制对象2、对象3。

```
Box ::Box ( const  Box & b )
{    height = b.height;
     width = b.width;
     length = b.length;  }
```

复制构造函数只有一个参数，这个参数是本类的对象，且采用引用对象形式，为了防止修改数据，加const限制。构造函数的内容就是将实参对象的数据成员值赋予新对象对应的数据成员，**如果程序中未定义复制构造函数，编译系统将提供默认的复制构造函数，复制类中的数据成员。**

5、对象的赋值和复制

□ 对象的复制

在以下情况调用复制构造函数

（1）在程序里用复制对象格式创建对象

（2）当函数的参数是对象。调用函数时，需要将实参对象复制给形参对象，在此系统将调用复制构造函数。

（3）在函数返回值是类的对象时，需要将函数里的对象复制一个临时对象当作函数值返回。

```
void fun(Box b) { … }
int main()
{
  Box box1(12, 15, 18);
  fun(box1);
  return 0;
}
```

```
Box f()
{
  Box box1(12, 15, 18);
  return box1;
}
int main()
{
  Box box2;
  box2 = f();
}
```

6、不同对象间实现数据共享

- C++用const保护数据对象不被修改，在实际中还需要共享数据，C++ 怎样提供数据共享机制?
- 静态成员、友元实现对象之间、类之间的数据共享。

6、不同对象间实现数据共享

□ 静态数据成员

静态数据成员定义格式：

static 类型 数据成员名

```
class Box
{
public:
  Box(int = 10, int = 10, int = 10);
  int volume();

private:
  static int height;
  int width;
  int length;
};
```

6、不同对象间实现数据共享

□ 静态数据成员

```
#include <iostream>
using namespace std;
class Box
 {public:
   Box(int,int);
   int volume();
   static int height;
   int width;
   int length;
 };
Box::Box(int w,int len)
 {
  width=w;
  length=len;
 }
int Box::volume()
 {return(height*width*length);
 }
int Box::height=10;
```

```
int main()
 {
  Box a(15,20),b(20,30);
  cout<<a.height<<endl;
  cout<<b.height<<endl;
  cout<<Box::height<<endl;
  cout<<a.volume()<<endl;
  return 0;
 }
```

6、不同对象间实现数据共享

□ 静态数据成员

（1）由于一个类的所有对象共享静态数据成员，所以不能用构造函数为静态数据成员初始化，只能在类外专门对其初始化。

格式：

数据类型 类名::静态数据成员名 = 初值;

如果程序未对静态数据成员赋初值，则编译系统自动用**0** 为它赋初值。

（2）既可以用对象名引用静态成员，也可以用类名引用静态成员

（3）静态数据成员在对象外单独开辟内存空间，只要在类中定义了静态成员，即使不定义对象，系统也为静态成员分配内存空间，可以被引用。

（4）在程序开始时为静态成员分配内存空间，直到程序结束才释放内存空间

（5）静态数据成员作用域是它的类的作用域（如果在一个函数内定义类，它的静态数据成员作用域就是这个函数）在此范围内可以用

类名::静态成员名

的形式访问静态数据成员。

6、不同对象间实现数据共享

静态成员函数

- C++ 提供静态成员函数，用它访问静态数据成员，静态成员函数不属于某个对象而属于类。
- 类中的非静态成员函数可以访问类中所有数据成员；而静态成员函数可以直接访问类的静态成员，不能直接访问非静态成员

静态成员函数定义格式：

static 类型 成员函数(形参表){…}

调用公有静态成员函数格式：

类名::成员函数(实参表)

调用方式	静态数据成员	非静态数据成员
静态成员函数	成员名	对象名.成员名
非静态成员函数	成员名	成员名

6、不同对象间实现数据共享

□ 静态成员函数

```
class Student
{
private:
  int num;
  int age;
  float score;
  static float sum;
  static int count;

public:
  Student(int, int, int);
  void total();
  static float average();
};
```

```
Student::Student(int m,int a,int s)
{
  num = m;
  age = a;
  score = s;
}
void Student::total()
{
  sum += score;
  count++;
}
float Student::average()
{
  return (sum / count);
}
float Student::sum = 0;
int Student::count = 0;
```

```
int main()
{
  Student stud[3] = {
      Student(1001, 18, 70),
      Student(1002, 19, 79),
      Student(1005, 20, 98)};
  int n;
  cout << "Please input the number of students:";
  cin >> n;
  for (int i = 0; i < n; i++)
    stud[i].total();
  cout<< "the average score of "
      << n << " student is "
      << Student::average() << endl;
  return 0;
}
```

6、不同对象间实现数据共享

□ 静态成员函数

	静态成员函数	普通成员函数
所有对象共享	Yes	Yes
隐含 this 指针	No	Yes
访问普通成员变量(函数)	No	Yes
访问静态成员变量(函数)	Yes	Yes
通过类名直接调用	Yes	No
通过对象名直接调用	Yes	Yes

7、允许访问私有数据的“朋友”

- 类成员包括：公用（public）、私有（private）、保护（protected）
- 一个例外——友元（friend）
- 友元包括：

  友元函数、友元类

- 友元是C++提供的一种破坏数据封装和数据隐藏的机制。
- 为了确保数据的完整性，及数据封装与隐藏的原则，建议尽量不使用或少使用友元。

7、允许访问私有数据的“朋友”

- 可以访问私有数据的友元函数

1、将普通函数声明为友元函数

2、将另一个类的成员函数声明为友元函数

友元函数声明格式：

friend 类型 类1::成员函数x(类2 &对象);

friend 类型 函数y(类2 &对象);

类1是另一个类的类名，类2是本类的类名。

功能：第一种形式在类2中声明类1的成员函数 x为友元函数。

第二种形式在类2 中声明一个普通函数 y 是友元函数。

注意：友元是单向的，类1是类2的朋友 ≠ 类2是类1的朋友

7、允许访问私有数据的“朋友”

可以访问私有数据的友元函数

```
class Time
{
public:
  Time(int, int, int);
  friend void display(Time &);
private:
  int hour;
  int minute;
  int sec;
};
Time::Time(int h, int m, int s)
{
  hour = h;
  minute = m;
  sec = s;
}
```

```
void display(Time &t)
{
  cout << t.hour << ":"
       << t.minute << ":"
       << t.sec << endl;
}
int main()
{
  Time t1(10, 13, 56);
  display(t1);
  return 0;
}
```

7、允许访问私有数据的“朋友”

提前声明

仅限指针变量和引用

```
class Date;
class Time
{
private:
  int hour;
  int minute;
  int sec;
public:
  Time(int, int, int);
  void display(const Date &);
};
class Date
{
private:
  int month;
  int day;
  int year;
public:
  Date(int, int, int);
  friend void Time::display(const Date &);
};
```

```
Time::Time(int h, int m, int s)
{
  hour = h;
  minute = m;
  sec = s;
}
void Time::display(const Date &da)
{
  cout << da.month << "/" << da.day
       << "/" << da.year cout << endl;
  cout << hour << ":" << minute << ":"
       << sec << endl;
}
Date::Date(int m, int d, int y)
{
  month = m;
  day = d;
  year = y;
}
```

```
int main()
{
  Time t1(10, 13, 56);
  Date d1(12, 25, 2004);
  t1.display(d1);
  return 0;
}
```

7、允许访问私有数据的“朋友”

- 可以访问私有数据的友元类

在B类中声明A类为友元类的格式：

```
friend A;
```

注意：

（1）友元关系是单向的，不是双向的。

（2）友元关系不能传递。

实际中一般并不把整个类声明友元类，而只是将确有需要的成员函数声明为友元函数。

7、允许访问私有数据的“朋友”

```
class A
{
private:
  int x;
public:
  A(){x=3;}
  friend class B;
};
```

```
class B
{
public:
  void disp1(A temp)
  {
    temp.x++;
    cout<<"disp1:x="<< temp.x <<endl;
  }

  void disp2(A temp)
  {
    temp.x--;
    cout<<"disp2:x="<< temp.x <<endl;
  }
};
```

```
int main()
{
  A a;
  B b;
  b.disp1(a); //友元类访问私有成员
  b.disp2(a);
  return 0;
}
```

8、类模板

- 对于功能相同而只是数据类型不同的函数，不必须定义出所有函数，我们定义一个可对任何类型变量操作的**函数模板**。
- 对于功能相同的类而数据类型不同，不必定义出所有类，只要定义一个可对任何类进行操作的**类模板**。

```
class Compare_int   //  比较两个整数的类
 {private:
    int x,y;
public:
   Compare_int(int a,int b)
    {x=a;y=b;}
   int max()
    {return (x>y)?x:y;}
   int min()
    {return (x<y)?x:y;}
};
```

```
class Compare_float   //  比较两个浮点数的类
 {private:
    float x,y;
public:
   Compare_float(float a,float b)
    {x=a;y=b;}
   float max()
    {return (x>y)?x:y;}
   float min()
    {return (x<y)?x:y;}
};
```

8、类模板

定义类模板的格式：

template < class 类型参数名>

class 类模板名

{ …… }

类型参数名：按标识符取名。如有多个类型参数，每个类型参数都要以class为前导，两个类型参数之间用逗号分隔。

类模板名：按标识符取名。

类模板{ ...}内定义数据成员和成员函数的规则：

用类型参数作为数据类型，用类模板名作为类。

8、类模板

□ 类内定义 VS 类外定义

在类模板外定义成员函数的语法

类型参数 类模板名<类型参数>::成员函数名(形参表) { … … }

```
template<class numtype>
class Compare
{ private:
    numtype x,y;
  public:
    Compare(numtype a,numtype b) //  构造函数
    {x=a;y=b;}
    numtype max()
    {return (x>y)?x:y;}
    numtype min()
    {return (x<y)?x:y;}
};
```

```
template<class numtype>
class Compare
{public:
  Compare(numtype a,numtype b)
  {x=a;y=b;}
  numtype max();
  numtype min();
 private:
  numtype x,y; };

numtype Compare< numtype > ::max()
  {  return (x>y)?x:y;}
numtype Compare< numtype > ::min()
  {  return (x<y)?x:y;}
```

8、类模板

□ 使用类模板

使用类模板时，定义对象的格式：

类模板名　<实际类型名>　对象名;

类模板名　<实际类型名>　对象名(实参表);

例如用类模板Compare定义对象：

Compare <int> cmp2(4,7);

在编译时，编译系统用 int 取代类模板中的类型参数numtype，就把类模板具体化了。

这时Compare <int> 相当于Compare_int类

```
int main()
{
  Compare<int> cmp1(3,7);
  cout<<cmp1.max()<<" is the maximum
integer."<<endl;
  cout<<cmp1.min()<<" is the minimum
integer."<<endl;
  Compare<float> cmp2(45.78,93.6);
  cout<<cmp2.max()<<" is the maximum float
number."<<endl;
  cout<<cmp2.min()<<" is the minimum float
number."<<endl;
  Compare<char> cmp3('a','A');
  cout<<cmp3.max()<<" is the maximum
character."<<endl;
  cout<<cmp3.min()<<" is the maximum
character."<<endl;
  return 0;
}
```

小结

- 类对象的初始化
- 对象数组
- 对象指针
- 公用数据的保护
- 对象的动态建立和释放
- 对象的赋值和复制
- 不同对象间实现数据共享
- 允许访问私有数据的“朋友”
- 类模板