面向对象程序设计

Object Oriented Programming

主 讲 教 师：陈洪刚
开 课 时 间：2021年
honggang_chen@yeah. net

CHAPTER 3

CONTENTS

CHAPTER 3

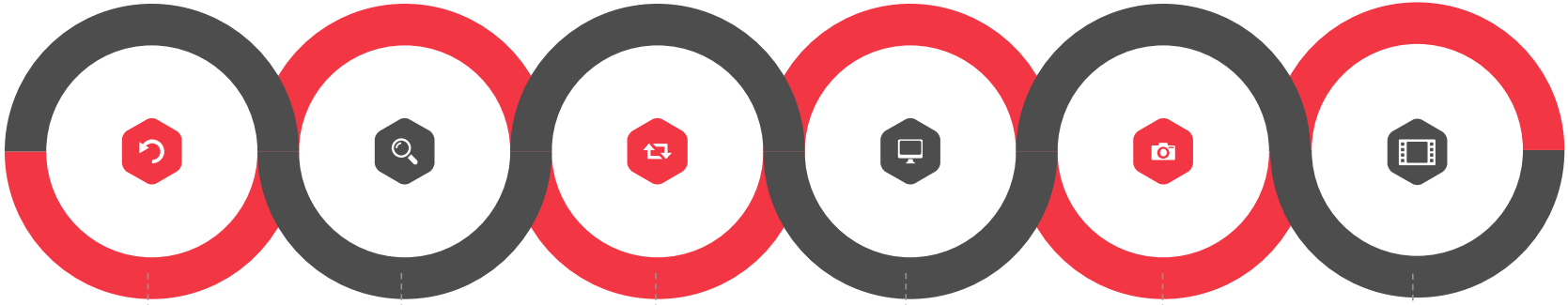
CONTENTS

3.1 利用构造函数对类对象进行初始化

- 如果定义一个变量，而程序未对其进行初始化的话，这个变量的值是不确定的，因为 C和C++ 不会自觉地去为它赋值。

- 如果定义一个对象，而程序未对其数据成员进行初始化的话，这个对象的值也是不确定的。

```
int i;
cout << i << endl;
// i的值可能是任何值
Time t;
t.show_time();
// 对象t的数据成员hour、
// minute、sec可能是任何值
```



对象的初始化

构造函数实现成员的初始化

带参数的构造函数

用参数初始化表对成员初始化

构造函数的重载

使用默认参数的构造函数

3.1.1 对象的初始化

- 变量初始化方式

```
int i = 10;      // 赋值表达式

int i(10);       // 表达式表

struct Time
{
  int hour, minute, sec;
};

Time t = {14, 56, 30};    // 初始化表
```

```
class  Time
{ public:
            int hour;
            int minute;
            int sec;
};

Time  t={ 14, 56, 30 }; // 定义对象并初始化成员
```

- 在一个大括号内顺序列出各个公有数据成员的值，在两个值之间用逗号分隔。注意这只能用于数据成员都是公有的情况。

3.1.1 对象的初始化

- 在前面的例子里，是用成员函数对对象的数据成员赋初值，如果一个类定义了多个对象，对每个对象都要调用成员函数对数据成员赋初值，那么程序就会变得烦琐，所以用成员函数为数据成员赋初值绝不是一个好方法。

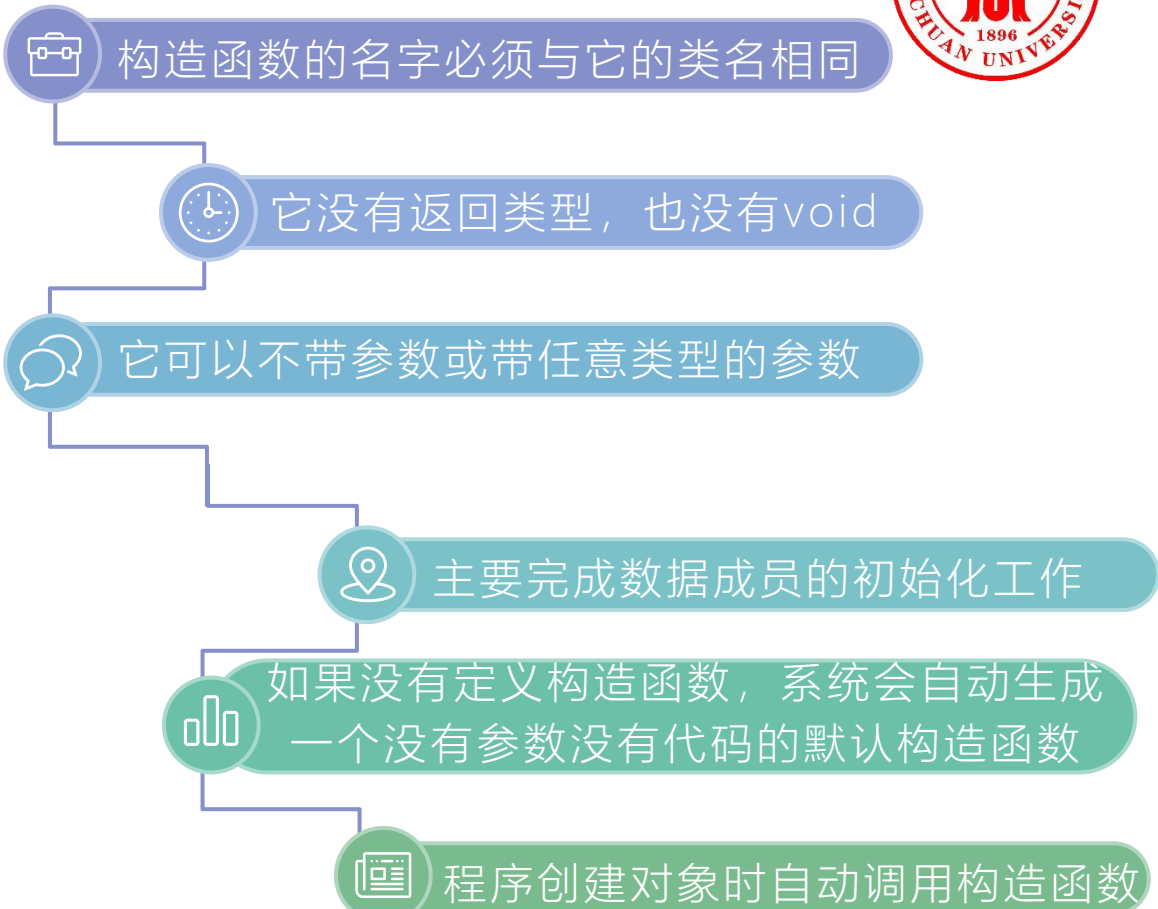
- C++ 提供了构造函数机制，用来为对象的数据成员进行初始化。在前面的学习中一直未讲这个概念，其实如果你未设计构造函数，系统在创建对象时，会自动提供一个默认的构造函数，而它只为对象分配内存空间其他什么也不做。

```
class  Time
{ public:
            int hour;
            int minute;
            int sec;
};

Time  t={ 14, 56, 30 };
// 定义对象并初始化成员
```

```
class    student
    {      int   num;
         public:
           void  setdata()
                 {cin >> num;
                }
     };
st1.setdata();  //调用成员函数
```

3.1.2 用构造函数实现数据成员的初始化

- 构造函数用于为对象分配空间和进行初始化，它属于某一个类，可以由系统自动生成。也可以由程序员编写，程序员根据初始化的要求设计构造函数及函数参数。
- 构造函数是一种特殊的成员函数，在程序中不需要写调用语句，在系统建立对象时由系统自觉调用执行。



3.1.2 用构造函数实现数据成员的初始化

例3.1 在例2.3的基础上定义构造成员函数

```
#include <iostream>
using namespace std;
class Time
{
public:
    Time() //  构造函数
    {  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;
}
```

3.1.2 用构造函数实现数据成员的初始化

- 在类Time中定义了构造函数Time，它与所在的类同名。在建立对象时自动执行构造函数，该函数的作用是为对象中的各个数据成员赋初值 0。 注意只有执行构造函数时才为数据成员赋初值。

- 程序运行时首先建立对象t1，并对t1中的数据成员赋初值0，然后执行t1.set_time函数，从键盘输入新值给对象t1的数据成员，再输出t1的数据成员的值。接着建立对象t2，同时对t2中的数据成员赋初值0，最后输出t2的数据成员的初值。

```
class Time
{
public:
    Time() //  构造函数
    {  hour=0;  minute=0;  sec=0;  }
    void set_time();
    void show_time();
private:
    int hour;
    int minute;
    int sec;
};
```

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

3.1.2 用构造函数实现数据成员的初始化

- 程序运行的情况为：

```
10 25 54
10:25:54        // 输出t1的值
0:0:0           // 输出t2的值
```

- 也可以在类内声明构造函数然后在类外定义构造函数。

```
Time();
// 然后在类外定义构造函数：
Time::Time() {
  hour = 0;
  minute = 0;
  sec = 0;
}
```

```
class Time
{
public:
    Time() //  构造函数
    {  hour=0;  minute=0;  sec=0;  }
    void set_time();
    void show_time();
private:
    int hour;
    int minute;
    int sec;
};
```

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

3.1.2 用构造函数实现数据成员的初始化

- 关于构造函数的使用，说明如下：
- （1）什么时候调用构造函数呢？当函数执行到对象定义语句时建立对象，此时就要调用构造函数，对象就有了自己的作用域，对象的生命周期开始了。
- （2）构造函数没有返回值，因此不需要在定义中声明类型。
- （3）构造函数不需要显式地调用，构造函数是在建立对象时由系统自动执行的，且只执行一次。构造函数一般定义为public。
- （4）在构造函数中除了可以对数据成员赋初值，还可以使用其他语句。
- （5）如果用户没有定义构造函数，C++系统会自动生成一个构造函数，而这个函数体是空的，不执行初始化操作。

3.1.3 带参数的构造函数

- 可以采用带参数的构造函数，在调用不同对象的构造函数时，从外边将不同的数据传递给构造函数，实现不同对象的初始化。

- 构造函数的首部的一般格式为：
  构造函数名(类型 形参1，类型 形参2，...)

- 在定义对象时指定实参，定义对象的格式为：
  类名 对象名(实参1，实参2，...);

```
Box(int, int, int);

// 长方体构造函数
Box::Box(int h, int w, int len)
{
  height = h;
  width = w;
  length = len;
}

Box box1(12,25,30);    // 定义对象box1
```

3.1.3 带参数的构造函数

例3.2 有两个长方柱，其长、宽、高分别为：（1）12，25，30（2）15，30，21。编写程序，在类中用带参数的构造函数，计算它们的体积。

分析：可以在类中定义一个计算长方体体积的成员函数计算对象的体积。

```
#include <iostream>
using namespace std;
class Box {
 public:
   Box(int, int, int);
// 函数原型,可不写参数名
   int volume();
 private:
   int height;
   int width;
   int length;
 };
// 长方体构造函数
Box::Box(int h, int w, int len)
{
  height = h;
  width = w;
  length = len;
}
//  计算长方体的体积
int Box::volume()
{
  return(height * width * length);
}
int main()
{
  Box box1(12,25,30);
// 定义对象box1
  cout<< " box1体积=" <<
box1.volume() <<endl;
  Box box2(15,30,21);
// 定义对象box2
  cout<< " box2体积= " <<
box2.volume()<<endl;
  return 0;
}
```

3.1.3 带参数的构造函数

- 构造函数Box有3个参数，分别代表长、宽、高。
- 在主函数中定义对象box1时，指定了实参12，25，30；定义对象box2时，指定了实参15，30，21。
- 然后调用成员函数计算长方体的体积。程序运行的结果如下：

  box1体积= 9000
  box2体积= 9450

- 带形参的构造函数在定义对象时必须指定实参
- 用这种方法可以实现不同对象的初始化

```
Box(int, int, int);

// 长方体构造函数
Box::Box(int h, int w, int len)
{
  height = h;
  width = w;
  length = len;
}

Box box1(12,25,30);    // 定义对象box1
Box box2(15,30,21);    // 定义对象box2
```

3.1.4 用参数初始化表对数据成员初始化

- C++提供了参数初始化表的方法对数据成员初始化。这种方法不必在构造函数内对数据成员初始化，在函数的首部就能实现数据成员初始化。

- 一般格式：
  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) {　}

- 功能：执行构造函数时，将形参1的值赋予成员1，将形参2的值赋予成员2，形参的值由定义对象时的实参值决定。

- 此时定义对象的格式依然是带实参的形式：
  类名　对象名(实参1，实参2)；

```
// 定义带形参初始化表的构造函数
Box::Box(int h, int w, int len) :
height (h), width(w), length(len) { }

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

Box box1(12,25,30);
```

3.1.5 构造函数的重载

- 构造函数也可以重载。一个类可以有多个同名构造函数，函数参数的个数、参数的类型各不相同。

例3.3：在例3.2的基础上定义两个构造函数。其中一个无参数，另一个有参数。

```
#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 box1(12,25,30);
Box::Box()
{
  height=10;
  width=10;
  length=10;
}

int Box::volume()
{
  return (height * width
              * length);
}
int main()
 {
  Box box1;
  cout<<"box1 体积=
"<<box1.volume()<<endl;

  Box box2(15,30,25);
  cout<<"box2 体积=
"<<box2.volume()<<endl;

  return 0;
 }
```

3.1.5 构造函数的重载

- 例子中定义了两个构造函数，一个无参数另一个带三个参数。系统根据定义对象的格式决定调用哪个构造函数。对象box1没有实参系统为它调用无参数的构造函数；对象box2带三个实参系统为它调用带形参的构造函数。
- 不带形参的构造函数为默认构造函数每个类只有一个默认构造函数，如果是系统自动分配的默认构造函数，其函数体是空的。
- 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。

```
#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 box1(12,25,30);
```

3.1.5 使用默认参数的构造函数

- C++允许在构造函数里为形参指定默认值，如果创建对象时，未给出相应的实参时，系统将用形参的默认值为形参赋值。
- 格式：函数名(类型 形参1=常数，类型 形参2=常数，...);

例3.4：将例3.3中的构造函数改用带默认值的参数，长、宽、高的默认值都是10。

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

int Box::volume()
{
  return (height * width *
length);}
int main()
 {
  Box box1;
  cout<<box1.volume()<<endl;
  Box box2(15);
  cout<<box2.volume()<<endl;
  Box box3(15,30);
  cout<<box3.volume()<<endl;
  Box box4(15,30,20);
  cout<<box4.volume()<<endl;
  return 0;}
```

3.1.5 使用默认参数的构造函数

- 程序运行结果为：

  box1 体积=1000
  box2 体积=1500
  box3 体积=4500
  box4 体积=9000

- 构造函数也可以改写成带参数初始化表的形式。整个函数只需一行，简单方便：

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

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

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

3.1.5 使用默认参数的构造函数

- 在构造函数中使用默认参数提供了建立对象的多种选择，它的作用相当于多个重载构造函数。需要注意：

- （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。

- （2）在声明构造函数时，形参名可以省略例如：Box(int =10,int =10,int =10);

- （3）如果构造函数的所有形参都指定了默认值，在定义对象时，可以指定实参也可不指定实参。由于不指定实参也可以调用构造函数，因此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。

- （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。

3.1.5 使用默认参数的构造函数

- 构造函数：

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

- 定义对象：

```
Box box1;
Box box2(15, 30 );
```

- 这时应该调用哪个构造函数呢？

- 如果构造函数中参数并非都带默认值时，要具体分析情况。

- 如有以下三个原型声明：

- 构造函数：

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

- 定义对象：

```
Box box1;     // 正确，调用第一个
Box box2(15);  //  调用第二个
Box box3(15, 30 ); //不知调用哪一个
```

- 因此不要同时使用重载构造函数和带默认值的构造函数。

3.2 利用析构函数进行清理工作

- 析构函数也是个特殊的成员函数，它的作用与构造函数相反，当对象的生命周期结束时，系统自动调用析构函数，收回对象占用的内存空间。

- 执行析构函数的时机：

① 在一个函数内定义的对象，当这个函数结束时，自动执行析构函数释放对象。

② 静态（static）局部对象要到main函数结束或执行exit命令时才自动执行析构函数释放对象。

③ 全局对象（在函数外定义的对象）当main函数结束或执行exit命令时自动执行析构函数释放对象。

④ 如果用new建立动态对象，用delete时自动执行析构函数释放对象。

3.3 利用析构函数进行清理工作

- 析构函数的特征

① 析构函数名以~符号开始后跟类名

② 析构函数没有数据类型、返回值、形参。由于没有形参所以析构函数不能重载。一个类只有一个析构函数。

③ 如果程序员没有定义析构函数，C++编译系统会自动生成一个析构函数。

- 析构函数除了释放对象（资源）外，还可以执行程序员在最后一次使用对象后希望执行的任何操作。例如输出有关的信息。

```
// 构造函数
Student(int n,string nam,char s)
{num=n;
name=nam;
sex=s;
cout<<"Constructor called."<<endl;}


// 析构函数
~Student()
{ cout<<"Destructor called."<<endl; }
```

3.3 利用析构函数进行清理工作

```
// 例3.5 包含构造函数和析构函数的C++程序
#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;
};
int main()
{Student stud1(10010,"Wang_li",'f');
  stud1.display();
  Student stud2(10011,"Zhang_fun",'m');
  stud2.display();
  return 0;}
```

3.3 利用析构函数进行清理工作

- main函数前声明的类其作用域是全局的。

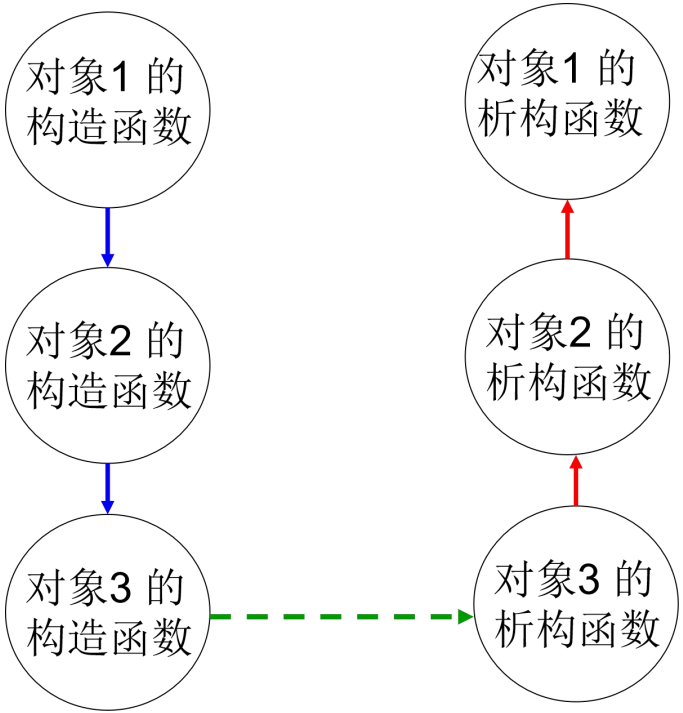
- 程序运行结果如下：

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

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

```
int main()
{
  Student stud1(10010,"Wang_li",'f');
  stud1.display();

  Student stud2(10011,"Zhang_fun",'m');
  stud2.display();

  return 0;
}
```

3.3 调用构造函数和析构函数的顺序

- 在使用构造函数和析构函数时需要特别注意对它们的调用时间和调用顺序。
- 在一般情况下，调用析构函数的次序与调用构造函数的次序恰好相反：最先调用构造函数的对象，最后调用析构函数。而最后调用构造函数的对象，最先调用析构函数。
- 可简记为：先构造的后析构，后构造的先析构，它相当一个栈，后进先出。



3.3 调用构造函数和析构函数的顺序

- 上面提到，在一般情况下，调用析构函数的次序与调用构造函数的次序相反，这是对同一类存储类别的对象而言的。

- (1)在全局范围中定义的对象（在所有函数之外定义的对象），在文件中的所有函数（包括主函数）执行前调用构造函数。当主函数结束或执行 exit 函数时，调用析构函数。

- (2)如果定义局部自动对象（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。

3.3 调用构造函数和析构函数的顺序

- (3)如果在函数中定义静态局部对象，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 函数时才调用析构函数。例如：

```
void fn()
{
    Student  st1;          //  定义局部自动对象
    static Student st2; //   定义静态局部对象
     …
}
```

- 对象st1是每次调用函数fn时调用构造函数在函数fn结束时调用析构函数。
- 对象st2是第一次调用函数fn时调用构造函数在函数fn结束时并不调用析构函数，在到主函数结束时才调用析构函数。

- 构造函数和析构函数在面向对象程序设计中是相当重要的。本章介绍了最基本的、使用最多的普通构造函数，在3.8节将介绍复制构造函数，在4.7节还要介绍转换构造函数。

3.4 对象数组

- 类是一种特殊的数据类型，它当然是 C++的合法类型，自然就可以定义对象数组。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 std[50];

- 构造函数：Student :: Student (int=1001,int=18,int=60);

- 在建立数组时，同样要调用构造函数。上面的数组有50个元素，要调用50次构造函数。

3.4 对象数组

- 如果构造函数有多个参数，C++ 要求：在等号后的花括号中为每个对象分别写出构造函数并指定实参。格式为：

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

- 假定对象有三个数据成员：学号、年龄、成绩。定义有三个学生的对象数组：

```
Student    std[ 3 ]={
Student(1001, 18, 87),
Student(1002, 19, 76),
Student(1003, 18, 80) };// 构造函数带实参
```

- 在建立对象数组时，分别调用构造函数，对每个对象初始化。每个元素的实参用括号括起来，实参的位置与构造函数形参的位置一一对应，不会混淆。

3.4 对象数组

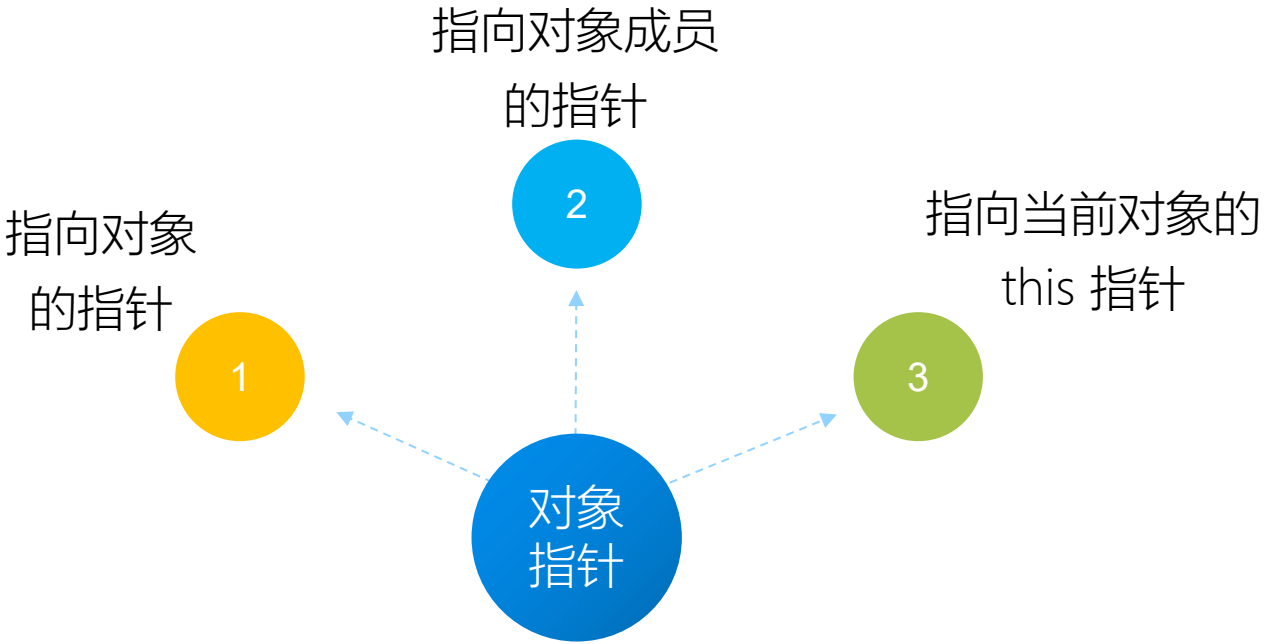
```
// 例3.5 例3.6 对象数组的使用方法。（长方体数组）
#include <iostream>
using namespace std;
class Box
{ public:
    // 带默认参数值和参数表
   Box( int h=10, int w=12, int
len=15 ) : height(h), width(w),
length(len) { }
    int volume();
 private:
   int height;
   int width;
   int length;
};
int Box::volume()
{  return(height*width*length); }

int main()
{
  Box a[3]={  Box(10,12,15),
              Box(15,18,20),
              Box(16,20,26)   };
  cout<<"a[0]的体积是 "<<a[0] .volume()<<endl;
  cout<<"a[1]的体积是 "<<a[1] .volume()<<endl;
  cout<<"a[2]的体积是 "<<a[2] .volume()<<endl;
  return 0;
}
// 每个数组元素是一个对象
```

运行结果如下：

```
a[0]的体积是 1800
a[1]的体积是 5400
a[2]的体积是 8320
```

3.5 对象指针

- 指针的含义是内存单元的地址，可以指向一般的变量，也可以指向对象。



3.5.1 指向对象的指针

- 对象要占据一片连续的内存空间，CPU实际都是按地址访问内存，所以对象在内存的起始地址是CPU确定对象在内存中位置的依据。这个起始地址称为对象指针。
- C++的对象也可以参加取地址运算： &对象名
- 运算的结果是该对象的起始地址，也称对象的指针。要用与对象类型相同的指针变量保存运算的结果。
- C++中定义对象的指针变量与定义其他的指针变量相似，格式如下：
  类名 * 变量名表
- 类名表示对象所属的类。 变量名按标识符规则取名，两个变量名之间用逗号分隔。定义好指针变量后，必须先给赋予合法的地址后才能使用。

3.5.1 指向对象的指针

- 在此基础上，有以下语句：

```
// 定义pt是指向Time类对象的指针
        Time  *pt;
//  定义Time类对象t1
        Time  t1;
// 将对象t1的地址赋予pt
        pt = &t1;
```

- 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 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<<":"<<min
ute<<":"<<sec<<endl;
}
```

3.5.2 指向对象成员的指针

- 对象由成员组成。对象占据的内存区是各个数据成员占据的内存区的总和。
- 对象成员也有地址，即指针。这指针分指向数据成员的指针和指向成员函数的指针。

- 1. 指向对象公有数据成员的指针

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;
```

3.5.2 指向对象成员的指针

- 对象由成员组成。对象占据的内存区是各个数据成员占据的内存区的总和。
- 对象成员也有地址，即指针。这指针分指向数据成员的指针和指向成员函数的指针。

- 1. 指向对象公有数据成员的指针

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;
```

3.5.2 指向对象成员的指针

➢ 2. 指向对象成员函数的指针

① 定义指向成员函数的指针变量： 数据类型（类名::* 变量名）(形参表)；
数据类型-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-指定成员函数的形参表(形参个数、类型)

② 取成员函数的地址： &类名::成员函数名；
在VC++系统中，也可以不写&。为了与C语言一致，建议不要省略&。

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，可以省略实参表。

3.5.2 指向对象成员的指针

例3.7 有关对象指针的使用方法

```
#include <iostream>
using namespace std;
class Time
{
public:
  Time(int,int,int);
  int hour;
  int minute;
  int sec;
  void get_time();
};
Time::Time(int h,int m,int s)
 {  hour  = h;
    minute = m;
    sec = s; }
void Time::get_time()
{cout<<hour<<":"<<minute<<":"<<sec<<endl; }

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;
}
```

3.5.2 指向对象成员的指针

- 程序运行结果为：

```
10              // *p1的值
10:13:56    //  t1.get_time(); 的执行结果
10:13:56    //  p2->get_time(); 的执行结果
10:13:56    //  (t1.*p3)(); 的执行结果
```

- 程序采用了三种方法输出t1的hour,minute,sec的值

- 说明：

(1) 成员函数的起始地址的正确表示是：

&类名::成员函数名。

不要写成： p3=&t1.get_time;

(2) 主函数的第8和9行可以合并写成：

```
void (Time::*p3) = &Time::get_time;
```

```
int main()
 {  Time t1(10,13,56);
    int *p1=&t1.hour;
    cout<<*p1<<endl;
    t1.get_time();
    Time *p2=&t1;
    p2->get_time();
    void (Time::*p3)();
    p3=&Time::get_time;
    (t1.*p3)();
    return 0;
}
```

3.5.3 指向当前对象的this 指针

- 一个类的成员函数只有一个内存拷贝。类中不论哪个对象调用某个成员函数，调用的都是内存中同一个成员函数代码。例如Time类一个成员函数：

```
void Time::get_time() { cout<<hour<<":"<<minute<<":"<<sec<<endl; }
 t1.get_time();
 t2.get_time();
```

- 当不同对象的成员函数访问数据成员时，怎么保证访问的就是指定对象的数据成员？

- 其实每个成员函数中都包含一个特殊的指针，它的名字是 this 。它是指向本类对象的指针，当对象调用成员函数时，它的值就是该对象的起始地址。

3.5.3 指向当前对象的this 指针

- 所以为了区分不同对象访问成员函数，语法要求的调用成员函数的格式是：
  对象名.成员函数名(实参表)

- 从语法上明确是对象名所指的对象调用成员函数。

- this指针是隐式使用的，在调用成员函数时C++把对象的地址作为实参传递给this指针。例如成员函数定义如下：

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

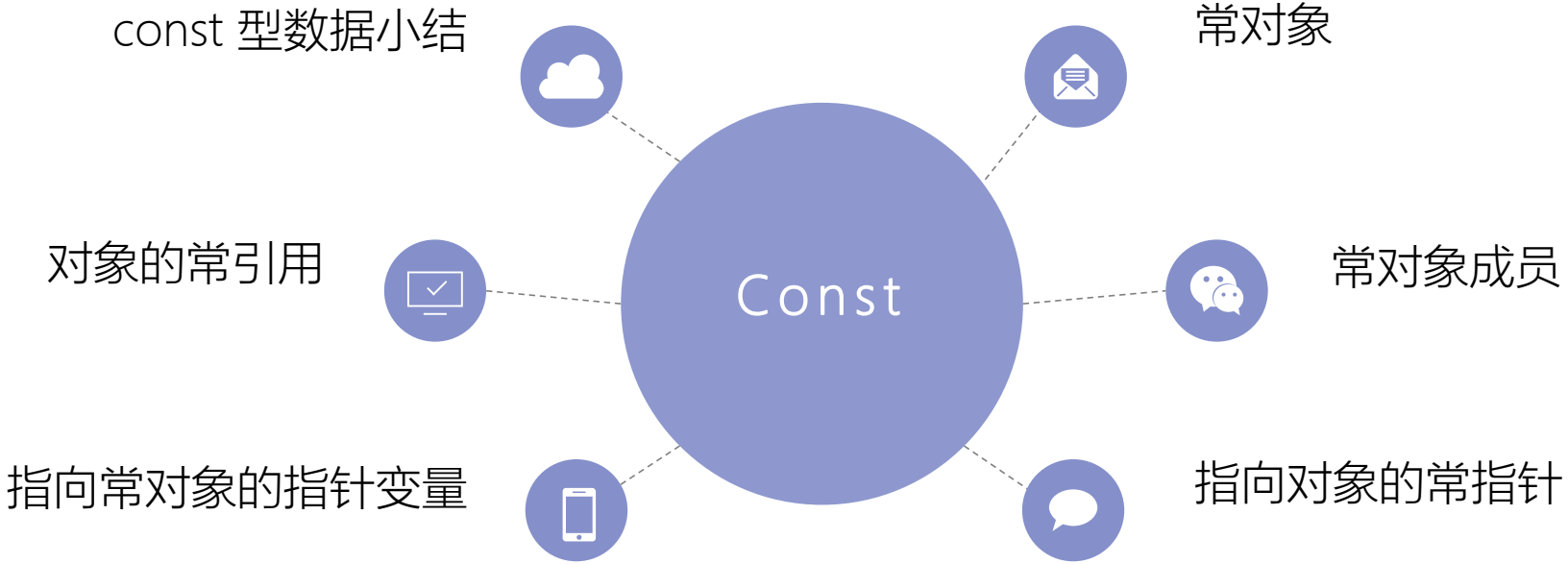
- 对于计算长方体体积的成员函数volume，当对象a 调用它时，就把对象 a地址给 this 指针，编译程序将a 的地址作为实参调用成员函数：a.volume(&a);
  实际上函数是计算：(this->height)*(this->width)*(this->length)
  这时就等价计算： (a.height)*(a.width)*(a.length)

3.5.3 指向当前对象的this 指针

- 可以用 (*this) 表示调用成员函数的对象。(*this) 就是this所指的对象。
- 如前面的计算长方体体积的函数中return语句可以写成：
  return((*this).height* (*this).width * (*this). length);
- 注意this两侧的括号不能省略，不能写成：
  return (*this.height * *this.width * *this. length);
- 根据运算符优先级语句被翻译成：*(this.height)，而this.height是非法的，编译将报错。
- C++ 通过编译程序，在对象调用成员函数时，把对象的地址赋予this 指针，用this 指针指向对象，实现了用同一个成员函数访问不同对象的数据成员。

3.6 共用数据的保护

- 如果既希望数据在一定范围内共享，又不愿它被随意修改，从技术上可以把数据指定为只读型的。C++提供const手段，将数据、对象、成员函数指定为常量，从而实现了只读要求，达到保护数据的目的。



3.6.1 常对象

- 定义格式：const 类名 对象名(实参表)； 或： 类名 const 对象名(实参表)；
- 把对象定义为常对象，对象中的数据成员就是常变量，在定义时必须带实参作为数据成员的初值，在程序中不允许修改常对象的数据成员值。
- 如果一个常对象的成员函数未被定义为常成员函数（除构造函数和析构函数外），则对象不能调用这样的成员函数。

  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
- 如果在常对象中要修改某个数据成员，C++提供了指定可变的数据成员方法。

  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。

3.6.2 常对象成员

- 可以在声明普通对象时将数据成员或成员函数声明为常数据成员或常成员函数。

- 1. 常数据成员

- 定义格式： const 类型 数据成员名
- 功能：将类中的数据成员定义为具有只读的性质。
- 注意只能通过带参数初始表的构造函数对常数据成员进行初始化。

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

- 在类中声明了某个常数据成员后，该类中每个对象的这个数据成员的值都是只读的，而每个对象的这个数据成员的值可以不同，由定义对象时给出。

3.6.2 常对象成员

- 可以在声明普通对象时将数据成员或成员函数声明为常数据成员或常成员函数。
  - 2. 常成员函数
  - 定义格式： 类型 函数名 (形参表) const
  - const 是函数类型的一部分，在声明函数原型和定义函数时都要用const关键字。const是函数类型的一个组成部分，因此在函数的实现部分也要使用关键字const。
  - 常成员函数不能修改对象的数据成员，也不能调用该类中没有由关键字const修饰的成员函数，从而保证了在常成员函数中不会修改数据成员的值。
  - 如果一个对象被说明为常对象，则通过该对象只能调用它的常成员函数。

3.6.2 常对象成员

- 一般成员函数可以访问或修改本类中的非 const数据成员。而常成员函数只能读本类中的数据成员，而不能写它们。

表 3.1

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

- 如果类中有部分数据成员的值要求为只读，可以将它们声明为const，这样成员函数只能读这些数据成员的值，但不能修改它们的值。
- 如果所有数据成员的值为只读，可将对象声明为const，在类中必须声明const 成员函数，常对象只能通过常成员函数读数据成员。
- 常对象不能调用非const成员函数。
- 提醒：如果常对象的成员函数未加const，编译系统将其当作非const成员函数；常成员函数不能调用非const成员函数。

3.6.3 指向对象的常指针

- 如果在定义指向对象的指针时，使用了关键字 const ， 它就是一个常指针，必须在定义时对其初始化。并且在程序运行中不能再修改指针的值。

- 定义格式： 类名 * const 指针变量名 = 对象地址

  例： Time t1(10,12,15)， t2;

  Time * const p1 = & t1;

  Time * const p1 = & t2; // 错误语句, 程序中不能修改p1

- 指向对象的常指针，在程序运行中始终指的是同一个对象。即指针变量的值始终不变，但它所指对象的数据成员值可以修改。当需要将一个指针变量固定地与一个对象相联系时，就可将指针变量指定为const。往往用常指针作为函数的形参，目的是不允许在函数中修改指针变量的值，让它始终指向原来的对象。

3. 6. 4 指向常对象的指针变量

- 指向的对象是常量，不能改。

- 定义格式：const 类型名 *ptr; 或 类型名 const *ptr;

例：
```
Time t1(10, 12, 15);        // 对象
const Time *p = &t1;   // 指向常量的指针
t1.hour = 18;               // 正确
p->hour = 18;               // 错误
```

3.6.5 对象的常引用

- 前面学过引用是传递参数的有效办法。用引用形参时，形参变量与实参变量是同一个变量，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

3.6.5 对象的常引用

例3.8 对象的引用

```
#include <iostream>
using namespace std;
class Time
 {public:
   Time(int,int,int);
   int hour;
   int minute;
   int sec;
 };
 Time::Time(int h,int m,int s)
 {hour=h;
  minute=m;
  sec=s;
 }
```

```
// 如果不希望在fun里修改t,可以加const
void fun(Time &t)
{t.hour=18;}

int main()
{
Time t1(10,13,56);
 fun(t1);
 cout<<t1.hour<<endl;
 return 0;
}

//  void fun( const Time &t);  //  形参为常引用
// 如果在函数里仍然要修改形参 ， 此时编译程序就会报错。
```

3.6.6 const 型数据的小结

表 3.3

形 式	含 义
Time const t1; 或 const Time t1;	t1 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 &t1 = t;	t1 是 Time 类对象 t 的引用,t 和 t1 指向同一段内存空间

3.7 对象的动态建立和释放

- C++提供了new和delete运算符，实现动态分配、回收内存。它们也可以用来动态建立对象和释放对象。

- 格式： new 类名；
- 功能：在堆里分配内存，建立指定类的一个对象。如果分配成功，将返回动态对象的起始地址（指针）；如不成功，返回0。为了保存这个指针，必须事先建立以类名为类型的指针变量。

- 格式： 类名 * 指针变量名；　　例： Box *pt; pt = new Box;

- 如分配内存成功，就可以用指针变量pt访问动态对象的数据成员。

```
cout << pt -> height;   cout << pt -> volume();
```

- 当不再需要使用动态对象时，必须用delete运算符释放内存。
  格式： delete 指针变量　　(指针变量里存放的是用new运算返回的指针)

3.8 对象的赋值和复制 —— 赋值

- 如果一个类定义了两个或多个对象，则这些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;
- 功能：将对象2值赋予对象1。对象1、对象2都是已建立好的同类对象。
- 说明： （1）对象的赋值只对数据成员操作。
- 说明： （2）数据成员中不能含有动态分配的数据成员。

例3.9 对象的赋值

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

class Box {
public:
  Box(int=10, int=10,
int=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(15,30,25), box2;
  cout << box1.volume() << endl;
  box2 = box1;
  cout <<box2.volume() << endl;
  return 0;
}
```

3.8 对象的赋值和复制 —— 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。C++ 还可以按照一个对象克隆出另一个对象（从无到有）。这就是复制(拷贝)对象。复制对象是创建对象的另一种方法（以前学过的是定义对象）。创建对象必须调用构造函数，复制对象要调用复制(拷贝)构造函数。

- 以Box 类为例，复制构造函数的形式是：

```
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。

3.8 对象的赋值和复制 —— 复制

- 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。

例3.9 用复制对象的方法创建Box类的对象。（用默认复制构造函数。）

```
#include "stdafx.h"
#include <iostream>
using namespace std;
class Box {
public:
  Box(int=10,int=10,int=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(15, 30, 25);
  cout <<  box1.volume() << endl;
  //Box box2 = box1, box3 = box2;
  Box box2(box1), box3(box2);
  cout <<  box2.volume() << endl;
  cout << box3.volume() << endl;
  return 0;
}
```

3.8 对象的赋值和复制 —— 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();
}
```

3.9 静态成员

- 静态成员提供一种共享机制，该共享一般只在类的各对象间共享。比全局共享具有更好的安全性。

- 1. 静态数据成员

- 定义：在一个类中，若将一个数据说明为static，则该数据称为静态数据成员。

  static 类型 数据成员名

- 作用：当作该类类型的全局变量。

- 特殊性：

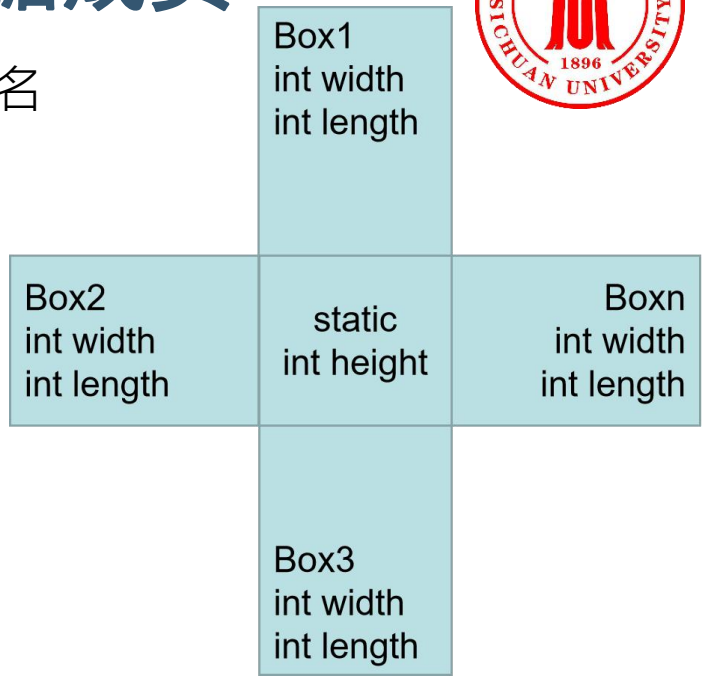
  1）对于非静态数据成员，每个类对象都有自己的拷贝，而静态数据成员对每个类类型只有一个拷贝。

  2）静态数据成员只有一份，由该类类型的所有对象共享访问。

3.9 静态成员 —— 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;};
```



- 静态数据成员的特性：设Box有 n个对象 box1..boxn。这n个对象的height 成员在内存中共享一个整型数据空间。如果某个对象修改了height 成员的值，其他 n-1 个对象的 height 成员值也被改变。从而达到n个对象共享height成员值的目的。

3.9 静态成员 —— 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。如果未对静态数据成员赋初值，则编译系统自动赋初值0。格式：数据类型 类名::静态数据成员名 = 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）静态数据成员也分公有和私有。

3.9 静态成员 —— 静态数据成员

例3.10 引用静态数据成员

```
#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;
}
```

程序执行结果：

```
10    10    10    3000
```

3.9 静态成员 —— 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)

引用方式	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

- 静态成员函数不带this指针，所以必须用对象名和成员运算符.访问非静态成员；而普通成员函数有this指针，可以在函数中直接引用成员名。

3.9 静态成员 —— 静态成员函数

- 静态成员函数不带this指针，所以必须用对象名和成员运算符.访问非静态成员；而普通成员函数有this指针，可以在函数中直接引用成员名。

```
class A
    {
    private:
      int x;      // 非静态成员
    public:
      static void f(A &a);  // 静态成员函数
    };
```

```
void A::f(A &a)
{
  cout << x;      //   对x的引用是错误的
  cout << a.x;    //   正确
}
```

3.9 静态成员 —— 静态成员函数

例3.11 关于引用非静态成员和静态成员的具体方法。

```
#include <iostream>
using namespace std;
class Student {
private:
  int num;
  int age;
  float score;
  static float sum;
// 总分，私有，类外不能访问
  static int count;      // 人数
public:
  Student(int, int, float);
  void total();
  static float average();
// 公有，类外可以调用
};
Student::Student(int m, int a,
float 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 = sizeof(stud) /
              sizeof(stud[0]);
  for (int i = 0; i < n; i++)
     stud[i].total();
  cout << n <<
"个学生的平均成绩是";
  cout << Student::average()
 << endl;
  return 0;
}
```

3.10 友元

- 类的私有成员只能被类的成员函数访问，但是有时需要在类的外部访问类的私有成员，C++ 通过友元的手段实现这一特殊要求。
- 友元可以是不属于任何类的一般函数，也可以是另一个类的成员函数，还可以是整个的一个类（这个类中的所有成员函数都可以成为友元函数）。
- 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。

3.10.1 友元函数

- 如果在 A 类外定义一个函数（它可以是另一个类的成员函数，也可以是一个普通函数），在A类中声明该函数是A的友元函数后，这个函数就能访问A类中的所有成员。

- B类是另一个类的类名，A类是本类的类名。则友元函数声明格式：

```
friend  类型 B类::funcx( A类  &对象 );
    //  第一种在B类中声明A类的成员函数funcx为友元函数

friend  类型 funcy( A类  &对象 );
    //  第二种在A类中声明一个普通函数funcy是友元函数
```

- 因为友元不是成员函数，它不属于类，所以它访问对象的成员时必须使用对象名。定义友元函数时形参通常定义为引用对象或对象指针，这样在友元函数内就能访问或修改实参对象了。

3.10.1 友元函数——普通函数

例3.12 将普通函数声明为友元函数 。

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

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;
}
```

3.10.1 友元函数——普通函数

使用友元函数计算两点距离

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

class Point {              //Point类声明
public:                    //外部接口
  Point(int xx = 0, int yy = 0) { X = xx; Y = yy; }
  int GetX() { return X; }
  int GetY() { return Y; }
  friend double Distance(Point &a, Point &b);
    // 友元函数
private:    //私有数据成员
            int X, Y;
};
```

```
double Distance(Point& a, Point& b)
{
  double dx = double(a.X - b.X);
  double dy = double(a.Y - b.Y);
  return sqrt(dx * dx + dy * dy);
}


int main() {
  Point p1(3, 5), p2(4, 6);
  double d = Distance(p1, p2);
  cout<<"The distance is "<<d<<endl;
  return 0;
}
```

3.10.1 友元函数——成员函数

例3.13 将成员函数声明为友元函数

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

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

void Time::display(const Date &da) {
  cout<<da.year"-"<<da.month<<"-"<<da.day<<endl;
  cout<<hour<<":"<<minute<<":"<<sec<<endl;}
```

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

- 注意：友元是单向的，此例中声明Time的成员函数display是Date类的友元，允许它访问Date类的所有成员。但不等于说Date类的成员函数也是Time类的友元。

3.10.2 友元类

- C++允许将一个类声明为另一个类的友元。假定A类是B类的友元类，A类中所有的成员函数都是B类的友元函数。
- 在B类中声明A类为友元类的格式：
  friend A;
- 注意：友元关系是单向的，不是双向的。
- 注意：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。

3.10.2 友元类

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

class B;
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);
    b.disp2(a);
    return 0;
}
```

3.10.2 友元类

```
// 前向声明，类名声明
class Student;

class Teacher
{privated:
      int noOfStudents;
      Student * pList[100];
public:
      void assignGrades(Student& s);
          //    赋成绩
      void adjustHours(Student& s);
          //    调整学时数
};
```

```
class Student
{privated:
      int Hours;
      float gpa;
public:
      friend class Teacher;
};

//  函数定义要在Student类定义之后
void Teacher:: assignGrades(Student& s){...};
void Teacher:: adjustHours(Student& s){...};
```

3.11 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。

- 例如定义比较两个整数的类和比较两个浮点数的类，这两个类做的工作是相似的所以可以用类模板，减少工作量。

```
// 比较两个整数的类
class Compare_int {
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; }
};

// 比较两个浮点数的类
class Compare_float {
...
};
```

3.11 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typename或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。

```
template<class T>
class Compare
{
private:
  T x, y;

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

3.11 类模板

- 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)
  {… … }

- 例如在类模板外定义max和min成员函数：

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

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

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

- 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> 就相当于Compare_int类。

3. 11 类模板

例3.14 声明类模板，实现两个整数、浮点数和字符的比较，求出大数和小数。

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

```
int main()
{
  Compare<int> cmp1(3,7);
  cout<<cmp1.max()<<"是两个整数中的大数."<<endl;
  cout<<cmp1.min()<<"是两个整数中的小数."<<endl;
  Compare<float> cmp2(45.78,93.6);
  cout<<cmp2.max()<<"是两个浮点数中的大数."<<endl;
  cout<<cmp2.min()<<"是两个浮点数中的小数."<<endl;
  Compare<char> cmp3('a','A');
  cout<<cmp3.max()<<"是两个字符中的大者."<<endl;
  cout<<cmp3.min()<<"是两个字符中的小者."<<endl;
  return 0;
}
```

面向对象程序设计

Object Oriented Programming

2021

谢谢大家