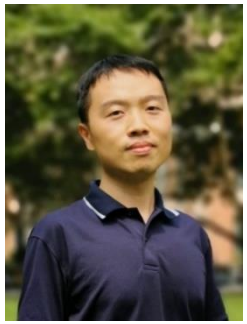
程序设计II



张书航 助理教授

电子邮件：zhangsh52@mail.sysu.edu.cn
个人主页：shuhangz.github.io

李同文 助理教授

电子邮件：litw8@mail.sysu.edu.cn

2022/04/17

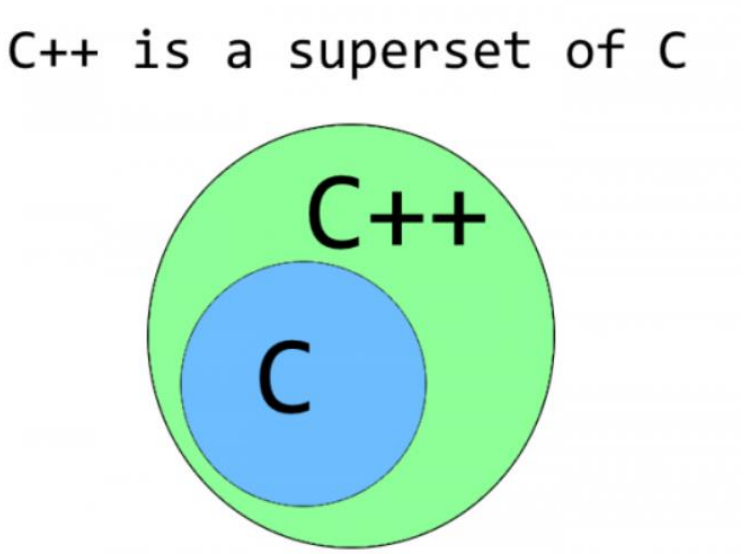
中山大學
SUN YAT-SEN UNIVERSITY

类与对象

程序设计II

回顾：From C to C++

- C++(C plus plus) 从C发展而来
  - “C with classes”
- C++对C的“增强”，表现在六个方面：
  - 类型检查更为严格。
  - 增加了面向对象的机制
  - 增加了模板的机制（Template）
  - 增加了异常处理
  - 增加了运算符重载
  - 增加了标准模板库（STL）



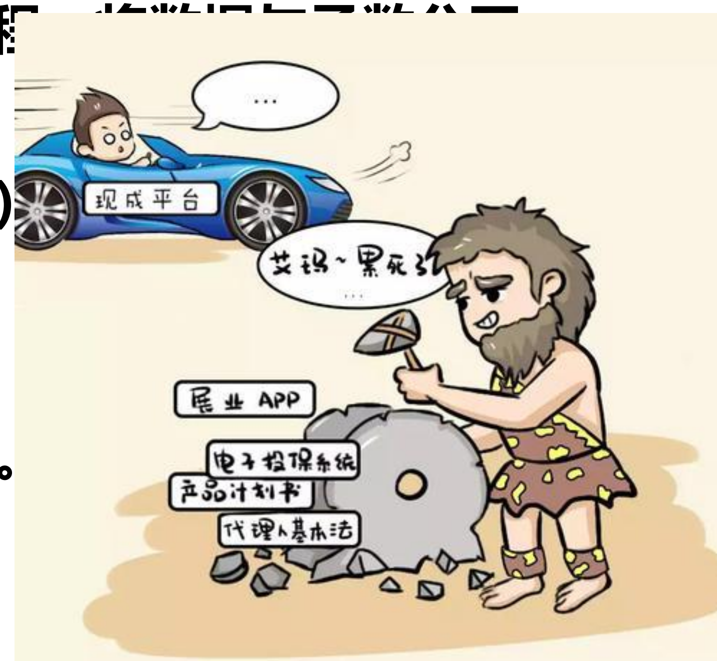
https://durofy.com/10-major-differences-between-c-and-c#prettyPhoto

回顾：面向过程的程序设计

- **重点：**
  - 围绕问题展开，如何实现细节过程，将数据与函数分开。
- **形式：**
  - 主模块+若干个子模块（main（）+子函数）。
- **特点：**
  - 程序=算法+数据结构
  - 自顶向下，逐步求精—功能分解。
- **缺点：**
  - 效率低，程序的可重用性差。
  - 由于数据与操作这些数据的代码（函数）相分离，一旦数据改变，则需要重新编写函数。
  - 程序功能扩充时，需大量修改函数，是手工作坊式的编程。

回顾：面向过程的程序设计

- **重点：**
  - 围绕问题展开，如何实现细节过程
- **形式：**
  - 主模块+若干个子模块（main（）
- **特点：**
  - 程序=算法+数据结构
  - 自顶向下，逐步求精—功能分解。
- **缺点：**
  - 效率低，程序的可重用性差。
  - 由于数据与操作这些数据的代码（函数）相分离，一旦数据改变，则需要重新编写函数。
  - 程序功能扩充时，需大量修改函数，是手工作坊式的编程。



类与对象

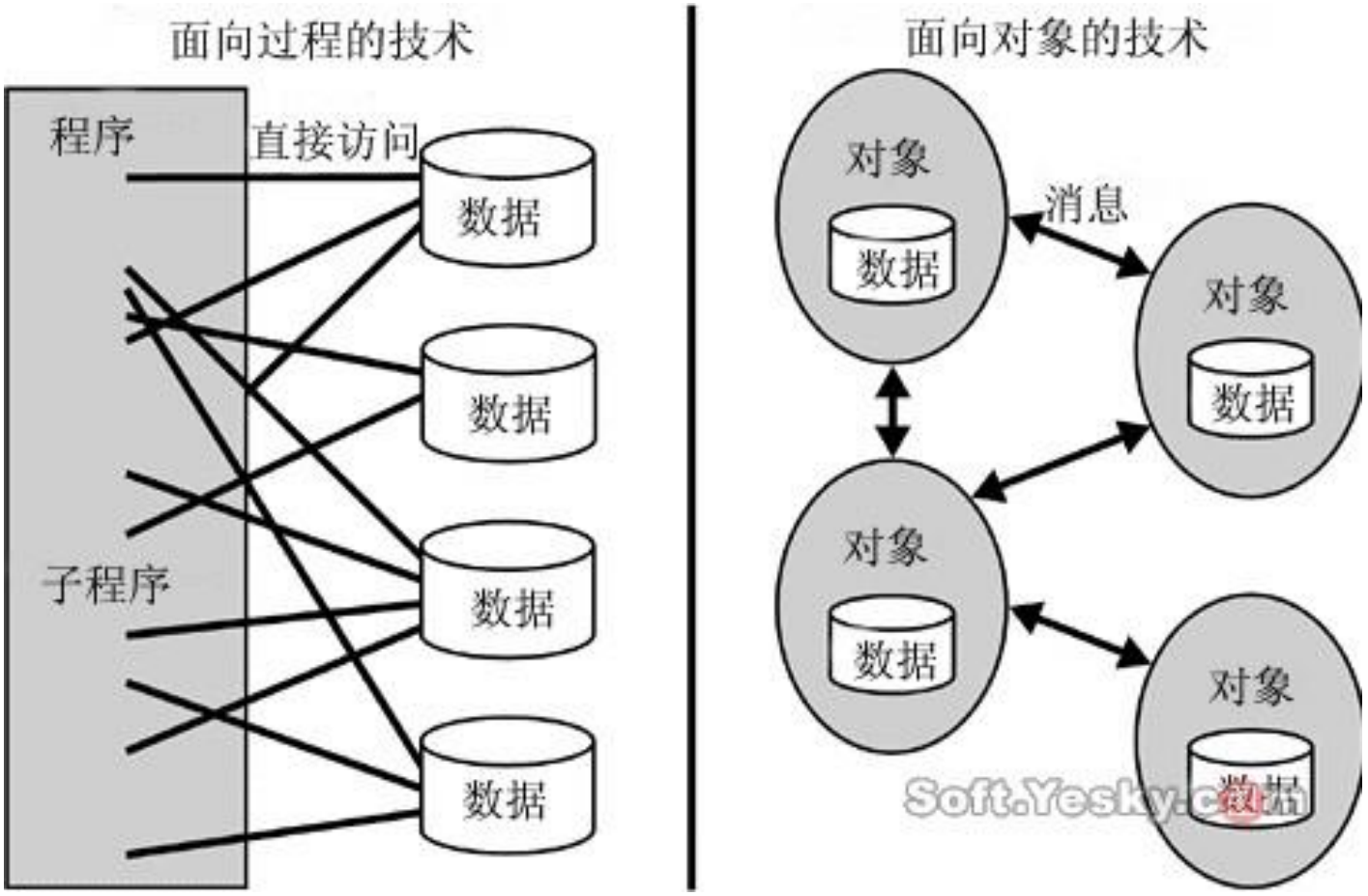
综合课本第16、17章

1 类与对象的基本概念
2 构造函数与析构函数
3 对象数组与对象指针
4 string类
5 向函数传递对象
6 对象的赋值和复制
7 静态成员
8 友元
9 类的组合
10 常类型

面向对象的方法

- **目的：**
  - 实现软件设计的产业化。
- **观点：**
  - 自然界是由彼此相关并互相通信的实体（对象）所组成。
- **程序设计方法：**
  - 使用面向对象的观点来描述模仿并处理现实问题。
- **要求：**
  - 描述或处理问题时应高度概括、分类、和抽象。
- **特点**
  - 开发时间短，效率高，可靠性高，所开发的程序更强壮
  - 重用，共享，可维护性，精简
  - 适合于大程序长时间的开发工作

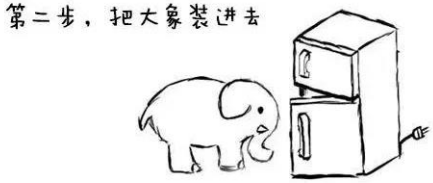
面向对象的基本概念


面向过程的技术
程序
直接访问
数据
数据
数据
数据
子程序
面向对象的技术
对象
数据
消息
对象
数据
对象
数据
对象
数据
Soft.Yesky.com

面向过程vs面向对象

- 问题：如何把大象装进冰箱？

面向过程：
为了把大象装进冰箱，需要3个过程。
1) 把冰箱门打开（得到打开门的冰箱）
2) 把大象装进去（打开门后，得到里面装着大象的冰箱）
3) 把冰箱门关上（打开门、装好大象后，获得关好门的冰箱）
每个过程有一个阶段性的目标，依次完成这些过程，就能把大象装进冰箱。



1:
冰箱开门(冰箱)
冰箱装进(冰箱, 大象)
冰箱关门(冰箱)
==换个写法
(冰箱开门 冰箱)
(冰箱装进 冰箱 大象)
(冰箱关门 冰箱)

2:
冰箱关门(冰箱装进(冰箱开门(冰箱), 大象))
==换个写法
(冰箱关门 (冰箱装进 (冰箱开门 冰箱) 大象))

https://www.zhihu.com/question/27468564/answer/36761503

面向过程vs面向对象

- 问题：如何把大象装进冰箱？

面向对象：

为了把大象装进冰箱，需要做三个动作（或者叫行为）。
每个动作有一个执行者，它就是对象。
1) 冰箱，你给我把门打开
2) 冰箱，你给我把大象装进去（或者说，大象，你给我钻到冰箱里去）
3) 冰箱，你给我把门关上
依次做这些动作，就能把大象装进冰箱。

```
1:
冰箱.开门()
冰箱.装进(大象)
冰箱.关门()
2:
冰箱.开门().装进(大象).关门()
```

https://www.zhihu.com/question/27468564/answer/36761503

结构体与类

1.结构体的扩充：结构体是自定义数据类型中的一种，它可含有不同类型的数据，还可以含有函数。

复数的结构体

```
struct complex{
double real;
double imag;
void init(double r,double
  i)
{ real=r;imag=i;}
double RealComplex()
{return real;}
double ImagComplex()
{return imag;}
double AbsComplex()
{double t;
t=real*real+imag*imag;
return sqrt(t);}
};
```

1.1 结构体与类

- **结构体的成员：**

  数据成员

  函数成员----成员函数

  (如上例中复数的结构体)

结构体的成员分类：

私有成员(private)：结构体内的其它成员访问

公有成员(public)：结构体内、外的其它成员都可以访问

C++结构体中的成员缺省为公有。

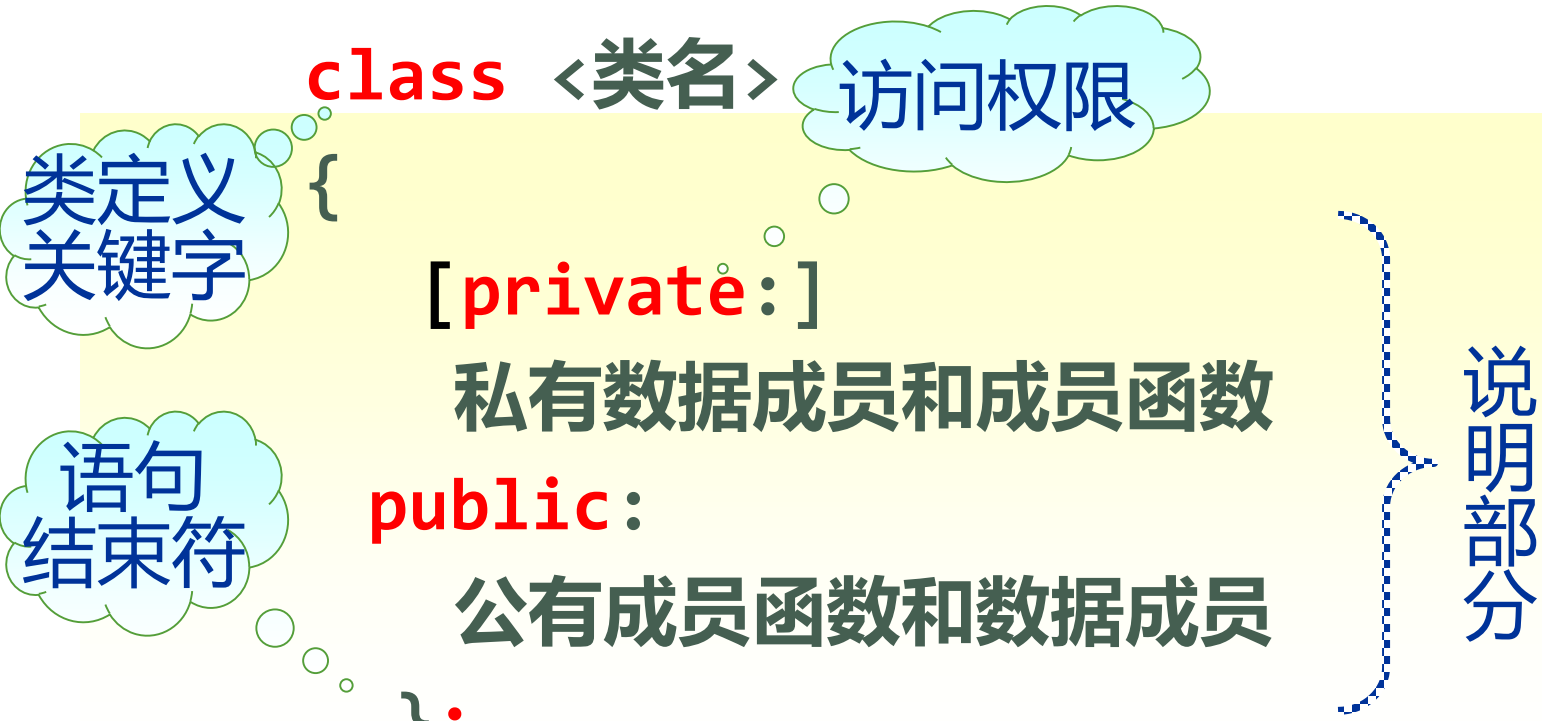
复数的结构体

```
struct complex{
private:
    double real;
    double imag;
public:
    void init(double
  r,double i)
    { real=r;imag=i;}
    double RealComplex()
    {return real;}
    double ImagComplex()
    {return imag;}
    double AbsComplex()
    {
    double t;
    t=real*real+imag*imag;
    return sqrt(t);
    }
    };
```

1 类与对象的基本概念

2. 类的声明

类是C++的一个重要的特性，是比struct更安全有效的数据类型。



1.1 结构体与类

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

class complex{
private:
double real;
double imag;
public:
void init(double r,double
  i)
{ real=r;imag=i;}
double RealComplex()
{return real;}
double ImagComplex()
{return imag;}
double AbsComplex()
{
double t;
t=real*real+imag*imag;
return sqrt(t);}
};
```

1.1 结构体与类

- 结构体和类的区别是：在未指定访问权限时，结构体中的成员被默认为公有的，类中的成员被默认为私有的。在所有其他方面，类和结构体等价。

尽管类和结构体是等价的，但一般在描述数据时使用结构体，在描述数据及对数据的操作时用类。

结构体与类的说明

- **访问权限修饰符**：公有(public)、私有(private)和保护(protected)；
- 访问权限修饰符出现的先后次序无关，并且允许多次出现；
- **缺省访问权限为私有**的；（与结构体不同）
- **公有部分：**一些操作（即成员函数），是提供给用户的接口功能；
- **私有部分：**一些数据成员，通常用来描述该类中的对象的属性；

结构体与类的说明

- 类中的数据成员的类型可以是任意的；包含整型、浮点型、字符型、数组、指针和引用等；**不能用**auto、register或extern进行说明。

- 在类体中**不允许用赋值语句**对所定义的数据成员赋初值

```
class abc{
   private:
    int b=33;
   public:
    …
};
```

OK

```
class abc{
   private:
    int b;
    b=33;
   public:
    …
};
```

error

1 类与对象的基本概念

1.2 成员函数的定义

有三种定义方式：

第一种方式是在类定义中只给出成员函数的原型，而成员函数体在**类的外部**定义。

其一般格式如下：

```
返回类型 类名 :: 函数名（参数表）{
        //函数体
}
```

双冒号::是域运算符,它主要用于类的成员函数的定义。

示例：定义坐标点类

类型和名

point.h

```
class point{
  private:
    int x, y;
  public:
    void setpoint
(int, int);
    int getx();
    int gety();
};
```

与原型中的一致

point.cpp

成员函数定义

```
void point ::
setpoint
(int a, int b)
{ x=a;y=b;}
int point :: getx()
      {return x;}
int point :: gety()
      {return y;};
```

作用域运算符

1.2 成员函数的定义

第二种方式是将成员函数定义在类的内部，即定义为内联函数。

point.cpp

```
class point {
    int x, y;
    public:
    void setpoint (int a,int b) { x = a;
y = b; }
    int getX (){return x ;}
    int getY (){return y;}
    };
```

1.2 成员函数的定义

第三种方式是将成员函数定义放在类的定义体外，在该成员函数定义前插入inline关键字。

point.h

```
class point{
    int x, y;
public:
void setpoint (int, int);
    int getx();
    int gety();
};
```

point.cpp

```
inline void point ::
setpoint (int a, int b)
{ x=a;y=b;}
inline int point ::
getx()
{return x;}
inline int point ::
gety()
{return y;}
```

说明：内联函数可提高效率，但只适用非常简单的函数。

3.1 类与对象的基本概念

1.3 对象的定义及使用

1．类与对象的关系

为了使用类，还必须定义类的对象。

在定义类时，系统是不会给类分配**存储空间**的，只有定义类的对象时才会给对象分配相应的内存空间。

```
类                    对象;
int                   i (0) ;
```

"对象是类的变量，也叫类的实例。"

1.3 对象的定义及使用

2．对象的定义

(1)在声明类的同时，直接定义对象

```
class point{
private:
 int x, y;
public:
      void setpoint (int, int);
int getx();
int gety();}
op1,op2;
```

1.3 对象的定义及使用

2．对象的定义

- (2)在使用类时定义类对象，定义的格式如下:
- <类名>　<对象名表>;
- 例:
- point　op1,op2;

- point *op=new point;

1.3 对象的定义及使用

3．对象中成员的访问

1）通过对象名访问对象成员

对象名．数据成员名

或　对象名．成员函数名（实参表）

·运算符：表示对象的成员

```
//例：对象的定义及使用
#include <iostream>
using namespace std;
class point{
 private:
    int x, y;
 public:
   void setpoint (int
 a, int b)
 { x=a;y=b;}
   int getx()
      {return x;}
   int gety()
   {return y;}
};
```

```
int main()
{
  point op1,op2;
  int i,j;
  op1.setpoint(1,2);
  op2.setpoint(3,4);
  i=op1.getx();
  j=op1.gety();
  cout<<"op1 i="<<i<<"op1 j="<<j<<endl;
  i=op2.getx();
  j=op2.gety();
  cout<<"op2 i="<<i<<"op2
 j="<<j<<endl;
  return 0;
}
```

1.3 对象的定义及使用

2）通过指向对象的指针访问对象成员

数据成员：<对象指针名>-><数据成员名>

成员函数：<对象指针名>-><成员函数名>(<参数表>)

```
int main()
{
    point ob, * op;
    op=&ob;
    op->setpoint(1,2);
    //…
    return 0;
}
```

->运算符：表示对象指针的成员

1.3 对象的定义及使用

3）通过对象的引用访问对象成员

数据成员：<引用名>.<数据成员名>

成员函数：<引用名>.<成员函数名>(<参数表>)

.运算符：表示引用对象的成员

```
int main()
{
    point ob;
    point &op=ob;
    op.setpoint(1,2);
    //…
    return 0;
}
```

1.4 类的作用域和类成员的访问属性

1.类的作用域

- 在类的声明中的一对大括号所形成的作用域，所有的类成员是属于类作用域的。
- 在类作用域范围内，可以直接使用类成员而不用加所属的类名。
- 类的外部不能直接以单个名字引用数据成员

1.4 类的作用域和类成员的访问属性

```
#include <iostream>
using namespace std;
class abc{
public:
    int i;
    void set (int);
    void disp()
   {cout<<"i= "<<i<<endl;}
};
void abc::set(int si)
{i=si;}
```

```
int fun()
{ return i;}
int main()
{
  abc  ob;
  ob.set(2);
  ob.disp();
   i=1;
  ob.disp();
  return 0;
}
```

1.4 类的作用域和类成员的访问属性

```
#include <iostream>
using namespace std;
class abc{
public:
    int i;
    void set (int);
    void disp()
   {cout<<"i= "<<i<<endl;}
};
void abc::set(int si)
{i=si;}
```

```
int fun(abc op)
{return op.i; }
int main()
{
  abc  ob;
  ob.set(2);
  ob.disp();
  ob.i=1;
  ob.disp();
  cout << fun(ob) <<
endl;
  return 0;
}
```

2.类成员的访问权限

- 类体定义类的成员，它支持两种类型的成员：
  (1)**数据成员**，它们指定了该类对象的内部表示。
  (2)**成员函数**，它们指定该类的操作。
- 类成员有三种不同的访问权限：
  (1)公有(public)成员可以在类外访问。
  (2)私有(private)成员只能被该类的成员函数访问。
  (3)保护(protected)成员只能被该类的成员函数或派生类的成员函数访问。---（后面会谈到）

公有成员的访问

```
class Time{
public:
    int hour;
    int minute;
    int second;

    void set_time(int,int,int);
    void show_time();
};
void Time::set_time(int h,int
   m,int s)
{hour=h;minute=m;second=s;}
```

```
void Time::show_time ()
{
cout<<hour<<':'<<minute<<':'<<second
<<endl;}
int main()
{    Time t1;
     t1.set_time(12,23,34);
     t1.show_time();
     t1.hour=23;
     t1.minute=34;
     t1.second=45;
     t1.show_time();
      return 0; }
```

私有成员的访问

```
class Time{
private:
    int hour;
    int minute;
    int second;
public:
    void set_time(int,int,int);
    void show_time();
};
void Time::set_time(int h,int
   m,int s)
{hour=h;minute=m;second=s;}
```

```
void Time::show_time ()
{
cout<<hour<<':'<<minute<<':'<<second
<<endl;}
int main()
{    Time t1;
     t1.set_time(12,23,34);
     t1.show_time();

     t1.set_time(23,34,45);


      return 0; }
```

2 构造函数与析构函数

构造函数与析构函数的功能：

- 特殊的成员函数；
- 构造函数：在创建对象时，为对象分配存储空间，也可以同时对它的数据成员赋初值。
- 析构函数：在撤销类的对象时，回收存储空间，并做一些善后工作。

2 构造函数与析构函数

2.1 构造函数(construction)

构造函数的特点：

(1) 构造函数的名字必须与类名相同;

(2) 构造函数可以有任意类型的参数，但不能具有返回类型和返回值;

(3) 定义对象时，编译系统会自动地调用构造函数;

(4)构造函数是成员函数，函数体可写在类体内，也可写在类体外；

(5)构造函数是公有的，但不能显式调用。

2.1 构造函数

```
class complex{
private:
   double real;
   double imag;
public:
   complex(double r,double i)
     {real=r;imag=i;}
   double AbsComplex();
}
```

2.1 构造函数例子

```
#include <iostream>
#include <math.h>
class complex{
private:
   double real;
   double imag;
public:
  complex(double
  r,double i)
{ real=r;imag=i;}
double AbsComplex()
{ double t;
   t=real*real+imag*imag;
   return sqrt(t);}
};
int main()
{
   complex A(1.1,2.2);
   cout<<"abs of complex
  A="<<A.AbsComplex()<<endl;
    return 0;
}
```

2 构造函数与析构函数

说明:

(1)构造函数的名字一定和类的名字相同。

(2)构造函数没有返回值；函数也没有类型。

(3)构造函数主要是两个功能：为对象开辟空间，为对象中的数据成员赋初值。

(4)如果没有定义构造函数，编译系统会自动生成一个缺省的构造函数。

```
complex::complex()
{   }
```

只能用来设置对象的存储空间，不能赋初值。

练习：定义时间类Time，实现时间的显示。（要求用构造函数初始化时间对象）

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

class Time{
   int hour;
   int minute;
   int second;
public:
Time (int h,int m,int s)
{ hour=h;
      minute=m;
      second=s;}
   void show_time();
};

void Time::show_time ()
{ cout<<hour<<':'<<minute
<<':'<<second<<endl;}

int main()
{   Time  t1(12,23,34);
     t1.show_time();
     return 0;
   }
```

构造函数可以**不带参数**，用固定值对对象进行初始化。

```
class ab{
      int a;
 public:
       ab();
       //…
  };
ab::ab()
{
   cout<<"Initialized\n";
   a=10;
}
```

不带参数的构造函数

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

用成员初始化列表对数据成员初始化

```
class A{
      int i;
      char j;
      float f;
 public:
      A(int I,char J,float F):i(I),j(J),f(F)
      {}
};
```

用成员初始化列表对数据成员初始化

- 带有成员初始化列表的构造函数的一般形式为：

  类名::构造函数名([参数表])[:成员初始化列表)]

  {

  构造函数体

  }

- 成员初始化列表的一般形式为：

  数据成员名1(初始值1),数据成员名2(初始值2),……

```
A(int I,char J,float F):i(I),j(J),f(F)
{}
```

2 构造函数与析构函数

2.3默认参数的构造函数

```
class complex{
private:
   double real;
   double imag;
public:
   complex(double
   r=0.0,double i=0.0) ;
  //含有缺省参数构造函数
double AbsComplex();
};
```

```
complex::complex(double
   r,double i)
{ real=r;imag=i;}
double complex ::AbsComplex()
{
    double t;
    t=real*real+imag*imag;
    return sqrt(t);
}
```

2 构造函数与析构函数

2.3默认参数的构造函数

```
main()
{
   complex s1;   //按缺省值定义对象
   complex s2(1.1); //只传递一个参数
   complex s3(1.1,2.2); //传递两个参数
   return 0;
}
```

2.4 重载构造函数

- 构造函数可以重载，C++根据声明中的参数选择合适的构造函数。

```
class A{
   public:
     A();
     A(int);
     A(int,char);
     A(float,char);
};
```

```
void main()
{
   A x;
   A y(10);
   A z(10, 'z');
   A w(4.4, 'w');
}
```

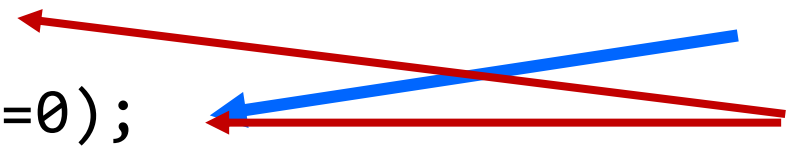
```
//例 计时器
#include <iostream>
using namespace std;
class timer{
   int seconds;
public:
 timer()
{ seconds=0;}
 timer(char *t)
{ seconds=atoi(t);}
 timer(int t)
{ seconds=t;}
 timer(int min,int sec)
{ seconds=min*60+sec;}
 int get_time()
{return seconds;}
};
int main()
{
  timer a,b(10),c("20"),d(1,10);
cout<<"seconds1="<<a.get_time()<<endl;
cout<<"seconds2="<<b.get_time()<<endl;
cout<<"seconds3="<<c.get_time()<<endl;
cout<<"seconds4="<<d.get_time()<<endl;
  return 0;
}
```

2.4 重载构造函数

- 说明：重载没有参数和带缺省参数的构造函数时，有可能产生二义性。

```
class x{
  public:
   x();
   x(int i=0);
};
```

```
void main()
{
   x m(10);
   x n;
}
```



error: call of overloaded 'x()' is ambiguous

2.4 重载构造函数

- 说明：重载没有参数和带缺省参数的构造函数时，有可能产生二义性。

参数数目引发的歧义

```
int get(){
return 5;
}
int get(int a = 5){
return a;
};}
//调用get()
//不给参数和有默认参数会造成歧义。
```

error: call of overloaded 'x()' is ambiguous

2 构造函数与析构函数

- 构造函数是一个有着特殊名字，在对象创建时被自动调用的一种函数，它的功能就是为类的对象分配存储空间并对对象的数据成员进行初始化。
- 析构函数?

2.5 析构函数(destruction)

析构函数：用来释放对象，在对象删除前做一些清理工作。

(1)析构函数的名字与类名相同，并在其前加"~"字符；

(2)析构函数没有参数；没有返回值，不能重载；

一个类中只能定义一个析构函数；

(3)析构函数在对象存在的函数体结束时或使用delete运算符释放new运算符创建的对象时被自动调用；

```
class complex{
private:
  double real;
   double imag;
public:
  complex(double r=0.0,double
 i=0.0);
 ~complex();
  double AbsComplex();
};
complex::complex(double
 r,double i)
{   cout<<"constructing..."<<
 endl;
  real=r;  imag=i; }
complex::~complex()
{
cout<<"destructing..."<<endl;
}
double complex::AbsComplex()
{//....
}
main()
{
complex  A(1.1,2.2);
//...
return 0;
}
```

声明构造函数

声明析构函数

定义构造函数

定义析构函数

析构函数

- 在对象的生存期结束的时刻系统自动调用它，然后再释放此对象所属的空间。
- 如果程序中未定义析构函数，编译器将自动产生一个默认的析构函数。

```
complex::~complex()
        {  }
```

析构函数

- 默认的析构函数不涉及用户申请的内存的释放等清理工作，如需要要显式地定义。

```
class string_data{
      char *str;
  public:
       string_data(char *s)
       {
              str=new char[strlen(s)+1];
              strcpy(str,s);
       }
       ~string_data()
       {delete str;}
       //…
};
```

显式析构函数

构造函数和析构函数特性

- 相似点：
  - 构造函数名称与类名相同，析构函数名称由类名前加"~"构成。
  - 构造函数和析构函数没有返回值。
  - 构造函数和析构函数一般必须定义为类的公有成员。
- 区别
  - 构造函数可以带默认参数，可以重载；
  - 析构函数参数表为空，不能重载；
  - 析构函数的调用顺序与构造顺序相反。

对象数组与对象指针

对象数组

数组是常用的自定义复杂类型，用于声明同类型的一组数据，对象数组就是每一个数组元素都是类对象。

- 声明：

  类名　　数组名[元素个数];

- 访问方法：　通过下标访问

  数组名[下标].成员名

```
//对象数组 例
#include <iostream>
using namespace std;
class exam{
    int x;
 public:
    void set_x(int n)
   { x=n;}
   int get_x()
   {return x;}
};

int main()
{
   exam ob[4];
   int i;
for(i=0;i<4;i++)
   ob[i].set_x(i);
for(i=0;i<4;i++)
   cout<<ob[i].get_x()<<'
';
   cout<<endl;
   return 0;
}
```

对象数组与对象指针

对象数组

•对象数组的初始化：

由于类对象的创建需要调用构造函数，类数组的初始化有额外要求：

（1）在没有为数组给出初始值时，数组元素用缺省值初始化。

应该在类中定义一个无参或带缺省参数的构造函数

若无自定义的构造函数，则各元素状态都不定。

对象数组初始化例子

```
#include <iostream>
using namespace std;
class point{
  private:
    int x, y;
 public:
    point()
   { x=5;y=5;}
point(int a, int b)
   { x=a;y=b;}
int getx()
{return x;}
int gety()
{return y;}
};
```

```
 int main()
{
  point op(3,4);
  cout<<"op x="<<op.getx()<<endl;
  cout<<"op y="<<op.gety()<<endl;
  point op_array[20];
  cout<<"op_array[15]
 x="<<op_array[15].getx()<<endl;
  cout<<"op_array[15]
 y="<<op_array[15].gety()<<endl;
  return 0;

}
```

对象数组初始化例子

```
#include <iostream>
using names
class point
  private:
    int x, y;
 public:
    point()
   { x=5;y=5;}
point(int a, int b)
   { x=a;y=b;}
int getx
{return
int gety
{return y;}
};
```

```
int main()

 point op(3,4);
 cout<<"op x="<<op.getx()<<endl;
 cout<<"op y="<<op.gety()<<endl;
 point op_array[20];
 cout<<"op_array[15]
x="<<op_array[15].getx()<<endl;
 cout<<"op_array[15]
y="<<op_array[15].gety()<<endl;
 return 0;

}
```

构造函数
(不带参数)

构造函数
(带参数)

(2)在数组定义时给出初始值，要求类中必须有自定义的有参构造函数。

格式：通过初始值表赋初值

```
#include <iostream>
using namespace std;
class exam{
    int x;
 public:
   exam(int n)
    { x=n;}
   int get_x()
    {return x;}
};
int main()
{
  exam ob[4]={11,22,33,44};
  int i;
  for(i=0;i<4;i++)
      cout<<ob[i].get_x()<<'
  ';
  cout<<endl;
  return 0;
}
```

构造函数(带参数)

通过初始化值表赋值

用带多个参数的构造函数给对象数组初始化

```
#include <iostream>
using namespace std;
class point{
    double x,y;
 public:
   point(double m,double n)
    { x=m;y=n;}
   void
 set_xy(double ,double);
   double get_x();
   double get_y();
};
```

```
int main()
{
  point
 op[4]={point(1.1,1.1),

 point(2.2,2.2),

 point(3.,3.),

 point(4.4,4.4)};
  //……
  return 0;
}
```

对象指针

对象指针在C++中很重要，对象可以直接访问，也可以通过对象指针来访问。

- 声明形式

  类名　*对象指针名;

- 例

```
Point A(5,10);
Point *ptr;
ptr=&A;
```

- 通过指针访问对象成员

  对象指针名->成员名

对象指针

用指针访问单个对象成员

```
#include <iostream>
using namespace std;
class exe{
    int x;
    public:
    void set_x(int a)
     { x=a;}
    void show_x()
     {cout<<x<<endl;}
};
```

```
main()
{
   exe ob,*p;
   ob.set_x(2);
   ob.show_x();
   p=&ob;
   p->show_x();
   return 0;
   }
```

对象

对象指针

指针指向对象

对象指针

用对象指针访问对象数组

```
#include <iostream>
using namespace std;
class exe{
    int x;
 public:
   void set_x(int a)
    { x=a;}
   void show_x()
    {cout<<x<<endl;}
  };
```

```
main()
{
exe ob[2],*p;
ob[0].set_x(10);
ob[1].set_x(20);
p=ob;
p->show_x();
p++;    //指向下一个对象
p->show_x();
return 0;
}
```

指针指向对象数组的第一个元素的地址

注意：当指针加1或减1时，它的增加或减少方式会使指针指向其基本类型的下一元素或上一元素。

this 指针

- 下面的赋值是不允许的：

```
void Date::setMonth( int month)
{
  month = month;
}
```

- 可以用this指针来解决：

```
void Date::setMonth( int month)
{
  this->month = month;
}
```

this 指针

- 该指针是隐含于每一个类的成员函数中的特殊指针;
- 该指针指向正在被某个成员函数操作的对象;
- *this标识调用该成员函数的对象;

  this->成员变量

```
void abc::init(char ma,int mb)
{a=ma;b=mb;}
void abc::init(abc *this,char ma,int mb)
{this->a=ma;this->b=mb;}
```

```
//例 this pointer
#include <iostream>
using namespace std;
class exth{
    int i;
 public:
   void load(int val)     { this->i=val;}
   int get()     {return this->i;}
};
 int main()
{  exth obj;
   obj.load(100);
   cout<<obj.get();
   return 0;
}
```

- 当一个非静态的成员函数被一个对象调用时，对象的地址作为一个**隐含的参数**传给被调用的函数。例如：

```
myDate.setMonth( 3 );
```

可被解释为：

```
setMonth( &myDate, 3 );
```

- 下面的成员函数setMonth，可用**两种方法实现**：
-

```
void Date::setMonth( int mn )//使用隐含的this指针
{
  month = mn;
}
```

-

```
void Date::setMonth( int mn )//显式使用this指针
{
  this->month = mn;
}
```

利用this 指针进行对象复制

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

class Sample{
int  x,y;
public:
 Sample(int i=0,int j=0)
{x=i;y=j;}
void copy(Sample &xy);
void print()
{
 cout<<x<<","<<y<<endl;
}
};
```

```
void Sample:: copy
   (Sample &xy)
{ if(this==&xy)return;
   * this=xy;
};
int main(){
   Sample p1,p2(5,6);
   p1.copy(p2);
   p1.print();
   return 0;
}
```

向函数传递对象

- 对象可以作为函数的参数，既可作值参（不影响实参），也可以作引用参数。
  - 对象作函数参数（传值）
  - 对象指针作函数参数（传址）
  - 对象引用作函数参数（引用）---更简单更直接，应用更广

向函数传递对象例子

```
//例
#include<iostream>
#include<string>
using namespace std;
class tr{
   int i;
 public:
   tr(int n)
    {i=n;}
   void set_i(int n)
    { i=n;}
   int get_i()
    {return i;}
};
void sqr_it (tr ob)
{
  ob.set_i(ob.get_i()*ob.get_i());
  cout<<"copy of obj has i valueof"<<
  ob.get_i();
  cout<<"\n";
}
int main()
{
  tr obj(10);
  sqr_it(obj);
  cout<<"But,obj.i is unchanged in
 main:";
   cout<<obj.get_i();
   return 0;
}
```

向函数传递对象例子(指针)

```
//例
#include<iostream>
#include<string>
using namespace std;
class tr{
   int i;
 public:
   tr(int n)
    {i=n;}
   void set_i(int n)
    { i=n;}
   int get_i()
    {return i;}
};
void sqr_it (tr *ob)
{
  ob->set_i(ob->get_i()*ob->get_i());
  cout<<"copy of obj has i valueof"<<
  ob->get_i();
  cout<<"\n";
}
int main()
{
  tr obj(10);
  sqr_it(&obj);
cout<<"Now, obj.i in main has been
 changed";
   cout<<obj.get_i();
   return 0;
}
```

向函数传递对象例子(引用)

```
//例
#include<iostream>
#include<string>
using namespace std;
class tr{
   int i;
 public:
   tr(int n)
    {i=n;}
   void set_i(int n)
    { i=n;}
   int get_i()
    {return i;}
};
void sqr_it (tr &ob)
{
  ob.set_i(ob.get_i()*ob.get_i());
  cout<<"copy of obj has i valueof"<<
  ob.get_i();
  cout<<"\n";
}
int main()
{
  tr obj(10);
  sqr_it(obj);
cout<<"Now, obj.i in main has been
  changed";
    cout<<obj.get_i();
    return 0;
}
```

对象的赋值和复制

对象的赋值语句

同类对象赋值，所有的数据成员都逐位拷贝。

B=A

```
//例
 #include <iostream>
using namespace std;
class myclass{
    int a, b;
 public:
    void set (int i, int j)
       { a=i;b=j;}
   void show()
       {cout<<a<<" "<<b<<endl;}
};
```

```
 int main()
{
  myclass  o1,o2;
  o1.set(20,5);
  o2=o1;
  o1.show();
  o2.show();
 return 0;
}
```

对象的赋值语句

•说明:

1. 赋值语句,两个对象的类型必须相同，否则编译时出错;
2. 对象的数据相同，对象仍是独立的;
3. 当类中存在指针时，可能产生错误。(多态)
4. 对象的赋值由默认的赋值运算符函数实现。

拷贝构造函数

★ **功能：**

用一个已知的对象来初始化一个被创建的同类对象；

★ **特点：**

- 函数名同类名，无返回类型；
- 只有一个参数，是对某个对象的引用；
- 每个类都必须有一个拷贝初始化构造函数；

<类名>(const <类名>& <引用名>)

拷贝构造函数

1、自定义拷贝构造函数（copy constructor）

拷贝构造函数是一种特殊的构造函数，其形参为本类的对象引用。

```
class 类名
  {  public :
        类名（形参）；//构造函数
        类名（const 类名 &引用名）；
  };
  类名:: 类名（const 类名 &引用名）
  {     函数体     }
```

拷贝构造函数

拷贝构造函数的实现

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

class point{
    int x, y;
  public:
    point(int a, int b)
   { x=a;y=b;}
   point(const point &p)
 { x=2*p.x;
     y=2*p.y;
    }
 void print()
{cout<<x<<"  "<<y<<endl;}
};
```

拷贝构造函数

```
 int main()
{
   point p1(30,40);
   point p2=p1;
   p1.print();
   p2.print();
   return 0;
}
```

"赋值"法调用拷贝构造函数

输出：

30 40

60 80

拷贝构造函数

2、默认拷贝构造函数

- 如果类中没有说明拷贝初始化构造函数，则编译系统自动生成一个默认拷贝初始化构造函数，作为该类的公有成员；
- 这个构造函数执行的功能是：用作为初始值的对象的每个数据成员的值，初始化将要建立的对象的对应数据成员。

默认拷贝构造函数的例子

```
//例
#include <iostream>
using namespace std;
class point{
    int x, y;
  public:
    point(int a, int b)
   { x=a;y=b;}
void print()
{cout<<x<<"
  "<<y<<endl;}
};
```

```
int main()
{
  point p1(30,40);
  point p2(p1);
  point p3=p1;
  p1.print();
  p2.print();
  p3.print();
  return 0;
}
```

输出:

```
30  40
30  40
30  40
```

拷贝构造函数的例子

```
//例
#include <iostream>
using namespace std;
class point{
    int x, y;
  public:
    point(int a, int b)
   { x=a;y=b;}
 point(const point &p)
  { x=2*p.x;
     y=2*p.y;
    }
void print()
{cout<<x<<"
  "<<y<<endl;}
};
```

```
int main()
{
  point p1(30,40);
  point p2(p1);
  point p3=p1;
  p1.print();
  p2.print();
  p3.print();
  return 0;
}
```

输出:

```
30  40
60  80
60  80
```

拷贝构造函数

3、调用拷贝构造函数的3种情况

（1）当用类的一个对象去初始化该类的另一个对象时系统自动调用它实现拷贝赋值。

如上例中的：

```
point p2(p1);
point p2=p1;
```

拷贝构造函数

（2）若函数的形参为类对象，调用函数时，实参赋值给形参，系统自动调用拷贝构造函数。例如：

```
void fun1(point  p)
{   cout<<p.GetX()<<endl;
}
void main()
{   point A(1,2);
    fun1(A); //调用拷贝构造函数
}
```

拷贝构造函数

（3）当函数的返回值是类对象时，系统**并不会调用拷贝构造函数**

。例如：

```
point fun2()
{     point A(1,2);
      return A; //调用构造函数
}
void main()
{
      point  B;
      B=fun2();
}
```

```
#include <iostream>
using namespace std;
class point{
      int  x,y;
public:
 point(int a0,int b=0);
 point(const point &p);
 void print()
 {
      cout<<x<<" "<<y<<endl;
 }
};
point::point(int a,int b)
{
      x=a;
      y=b;
      cout<<"using normal
constructor\n";
}
point::point(const point &p)
{
      x=2*p.x;
      y=2*p.y;
      cout<<"using copy
constructor\n";
```

```
void fun1(point p)
{    p.print();   }
point fun2()
{
    point p4(10,30);
    return p4;
}
int main()
{
    point p1(30,40);
    p1.print();

    point p2(p1);
    p2.print();

    point p3=p1;
    p3.print();

    fun1(p1);
    p2=fun2();
    p2.print();
    return 0;
}
```

拷贝构造函数总结

- 用一个对象初始化时会调用拷贝构造函数，包括用”=”赋值和Copy(copy)这样用类作为构造函数的参数的初始化.
- 作为形参时,也会调用到拷贝构造函数.
- 在作为函数的返回值时,**并不会调用拷贝构造函数**.

静态成员

- **为什么引入静态成员 ？**
- 在程序设计中，应尽量减少全局对象的使用。复杂的程序是由许多程序员共同设计的，全局对象的使用不利于程序的组织与管理。
- 原因之一是全局对象无法控制其可见范围(或可见范围太大)，无法控制谁可否使用它；
- 原因之二是全局对象表示一个有意义的实体，但其所属类型允许的合法操作不一定适用于它所代表的外部对象的行为，即难以保证是否对此对象实施了不合法操作。

```
//引例：point的个数
#include <iostream>
using namespace std;
int  num=0; //全局变量
class point
{   int X,Y;
public:
     point(int x,int y)
     { X=x;Y=y; num++;}
     ~point( ) { num--; }
};
```

```
int main()
{  cout<<num<<endl;
    point p1(1,1);
    cout<<num<<endl;
    point *p2=new point(2,2
    cout<<num<<endl;
    delete p2;
    cout<<num<<endl;
    return 0;
}
```

结果：
0
1
2
1

在C++中的解决方法：

1. 将这个全局对象的声明放到类中，借助类对成员的访问控制能力来限制程序中对这个对象的可访问性——这就是静态数据成员提出的动机。
2. 根据数据抽象的思想对数据进行封装，隐藏数据表示，数据的含义完全由允许的操作来解释——这样又提出了静态成员函数。

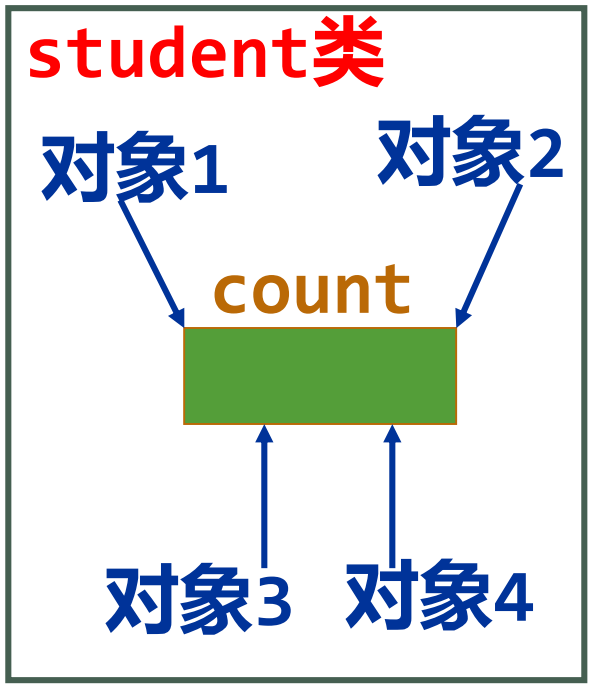
静态数据成员和静态成员函数统称为静态成员，在声明中加关键字static，它们的使用及意义与类中的普通成员有很大区别。

静态成员

静态数据成员

- 要定义静态数据成员，只要在数据成员的定义前增加static关键字。
- 静态数据成员不同于非静态的数据成员，一个类的静态数据成员仅创建和初始化一次，且在程序开始执行的时候创建，然后被该类的所有对象共享
- 非静态的数据成员则随着对象的创建而多次创建和初始化。

```
class Student{
      static int count;
     int student_no;
 public:
   Student()
    { count++;
      student_no=count;
     }
   void print()
    {  cout<<"Student"<<student_no<<" ";
       cout<<"count="<<count<<endl;
    }
};
int Student::count=0;
```



静态成员

- 静态数据成员
  - 用关键字static声明
  - 在类内只作一个"引用性声明"
  - 必须在**类外定义和初始化**，用(::)来指明所属的类
  - 该类的所有对象维护该成员的同一个拷贝，或者说，这个成员对所有的对象都是透明的。所有的对象都可以通过这个成员实现数据共享。

静态数据成员

- 是类的所有对象共享的成员，而不是某个对象的成员。
- 一个类的静态数据成员是用来表示类的属性的成员，而不是对象的属性的成员。
- 可以通过类名访问静态数据成员，也可以通过对象名访问静态数据成员

练习：给出程序运行结果

```
#include<iostream>
using namespace std;
class Sm {
        public:
                static int n;
};
int Sm::n=2;
int main(  )
{     Sm ie1, ie2;
       cout<<ie1.n<<" ";
       ie2.n=3;
       ie1.n=5;
       cout<<ie2.n;
       return 0;
}
```

静态成员函数

1、作用:

操作静态数据成员;

2、定义格式

```
static   返回类型   静态成员函数名(参数表)
```

3、使用格式

<类名>::<静态成员函数名>(<实参表>)

<对象名>.<静态成员函数名>(<实参表>)

<对象指针>-><静态成员函数名>(<实参表>)

静态成员函数例子

```
#include <iostream>
using namespace std;
class point
{   int X,Y;
     static  int  num;
public:
     point(int x,int y)
     { X=x;Y=y; num++;}
     ~point( ) { num--; }
     static int num_obj( );
};
int  point::num_obj( )
{  return num; }
```

```
int point::num=0; //初始化
int main()
{  cout<< point::num_obj()<<endl;
    point p1(1,1);
    cout<< p1.num_obj()<<endl;
    point *p2=new point(2,2);
    cout<< p2->num_obj()<<endl;
    delete p2;
    cout<< point::num_obj()<<endl;
    return 0;
 }
```

静态成员函数

- 静态成员函数也是属于整个类。不论定义多少类的对象，静态成员函数也只有一个拷贝。
- 静态成员函数也要通过static来声明。
- 在程序中可以使用类名或对象名来调用静态成员函数。一般的成员函数只能通过对象名来调用。
- 静态成员函数没有this指针，可以直接引用属于该类的静态数据成员或静态成员函数。一般不访问类中的非静态成员。如需访问，则通过对象名（或对象指针、对象引用）访问该对象的非静态成员。

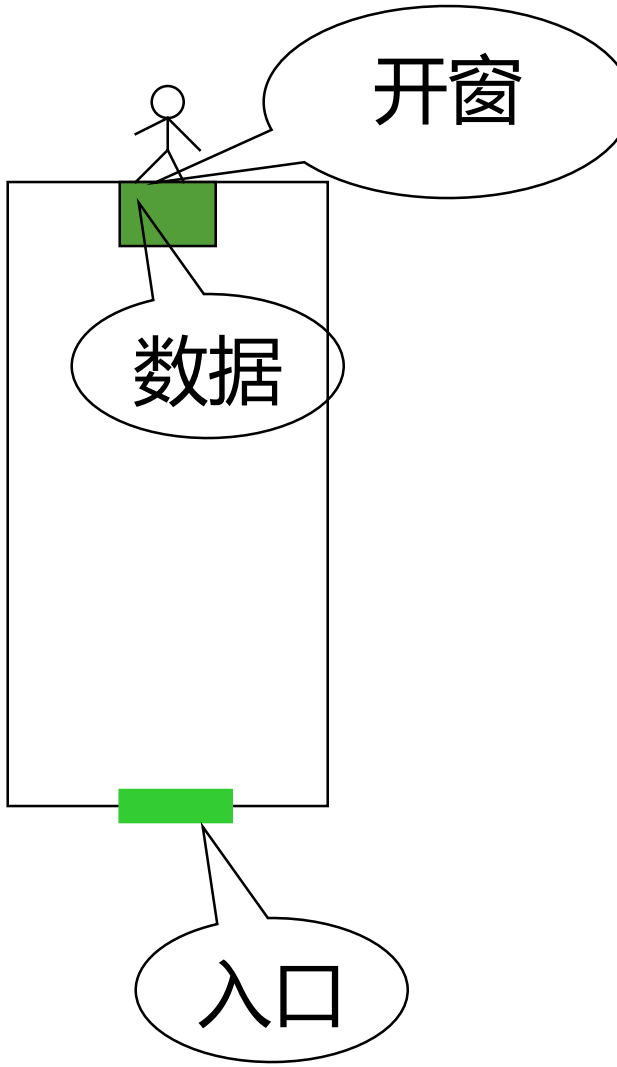
静态成员

- 使用静态成员应注意：
  1. 静态成员函数只能直接访问类的静态数据成员或静态成员函数。
  2. 非静态成员函数可以直接访问类的静态数据成员和非静态数据成员。
  3. 静态数据成员的初始化工作只能在全局数据声明和定义处进行。

友元

为什么要引入友元?

类的封装性(数据隐藏)保护了数据的安全性和软件的可维护性，是一大优点，但也带来了一些不便——模块之间的交互只能通过操作接口来进行。这种限制使得有时效率不高。



"给封装开个窗"

友元

友元：不属于任何类的一般函数、另一个类的成员函数、整个的一个类。

- 友元是C++提供的一种破坏数据封装和数据隐藏的机制。
- 通过将一个模块声明为另一个模块的友元，一个模块能够引用到另一个模块中原本是被隐藏的信息。
- 可以使用友元函数和友元类。
- 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。

友元

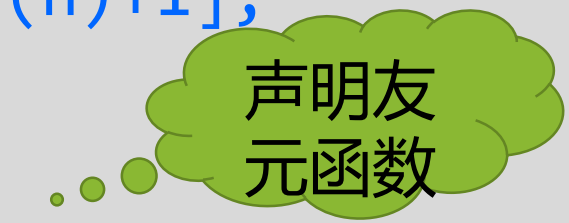
友元函数

1.普通函数声明为类的友元函数

不属于任何类的一般函数,是一种定义在类外部的普通函数，但需要在类体内进行说明(函数前面加friend关键字）；

- 不是成员函数，但可以访问类中的私有成员和公有成员；

```
//例  friend function
#include <iostream>
#include <string.h>
using namespace std;
class girl{
    char *name;
    int age;
public:
   girl(char *n,int d)
   {  name=new char[strlen(n)+1];
      strcpy(name,n);
     age=d; }
    friend void disp(girl &);
    ~girl(){delete name;}
   };
```



```
void disp(girl &x)
{
    cout<<"girl\'s name is:
 "<<x.name<<",age:"<<x.age<<"\n";
}
int main()
{
   girl e("Wang Meng",18);
   disp(e);
   return 0;
}
```

友元函数

- 该类友元函数不是成员函数，定义时不用加"类名::"。
- 普通函数声明为友元函数一般带有一个该类的入口参数。友员不是成员，不能直接引用对象成员的名字，不能用this指针访问它。
- 一个函数可以是多个类的友元函数，只需要在各个类中分别声明

友元

- 定义友元函数通过**直接访问对象的私有数据成员**，避免了函数调用的开销，从而提高了程序运行的效率。
- 虽然友元为我们进行程序设计提供了一定的方便性，但是面向对象的程序设计要求类的接口与类的实现分开，对对象的访问通过其接口函数进行
- 如果直接访问对象的私有成员，就破坏了面向对象程序的信息隐藏和封装特性
- 虽然提供了一些方便，但有可能是得不偿失的，所以，我们要慎用友元。

友元函数

2.成员函数声明为友元函数

该函数是另一个类的成员函数，前加上friend表示该函数是另一类的成员函数，又是本类的友元成员。

```
#include <iostream>
#include <string.h>
using namespace std;

   class girl;
   class boy{
     char *name;
      int age;
    public:
       boy(char *n,int d)
       {
        name=new char[strlen(n)+1];
        strcpy(name,n);
        age=d;
       }
       void disp(girl &);
        ~boy()
        {delete name;}
       };
class girl{
    char *name;
    int age;
public:
   girl(char *n,int d)
   {
    name=new char[strlen(n)+1];
    strcpy(name,n);
    age=d;
   }
    friend void boy::disp(girl &);
    ~girl()
    {delete name;}
};

void boy::disp(girl &x)
{    cout<<"boy\'s name is:
"<<name<<",age:"<<age<<"\n";
     cout<<"girl\'s name is:
"<<x.name<<",age:"<<x.age<<"\n";
}

 int main()
{
     boy b("yao ming",25);
     girl e("wang nan",22);
     b.disp(e);
     return 0;
}
```

向前引用先定义

Girl声明友元函数

“你说你是我的朋友不算，我说你是我的朋友才算“

友元

友元类

在类内说明一个类(该类前面已定义)，前加上 `friend`表示该类是本类的友元类。友元类的所有成员函数可以访问它自己类私有成员又可访问本类的私有成员

- 若一个类为另一个类的友元，则此类的所有成员都能访问对方类的私有成员。
- 定义语法：将友元类名在另一个类中使用 `friend` 修饰说明。

友元类

- 声明方法：在类A的类体中加入声明语句

  `friend 类名B;` 则声明了类B为类A的友元类。

```
class B;
class A {
  //...
  friend  B;
};
```

类B是类A的友元，类B中的所有成员函数都能访问类A的私有成员。

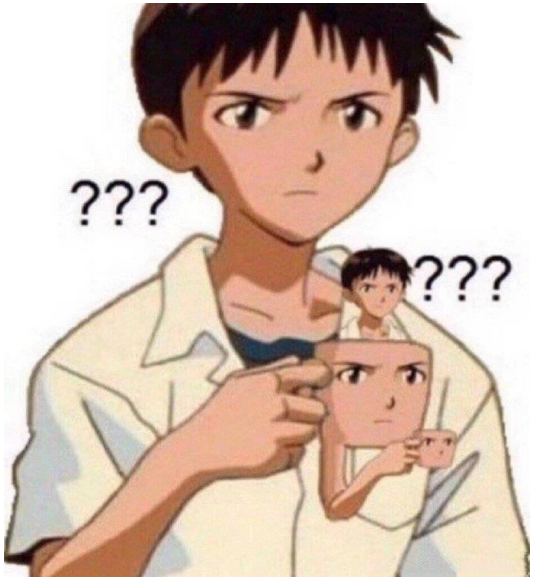
```
#include <iostream>
#include <string.h>
using namespace std;
class girl;
class boy{
    char *name;
    int age;
 public:
   boy(char *n,int d)
   {  name=new char[strlen(n)+1];
      strcpy(name,n);
      age=d; }
   void disp(girl &);
    ~boy()    {delete name;}
};
class girl{
    char *name;
    int age;
    friend boy;
   public:
   girl(char *n,int d)
   {  name=new char[strlen(n)+1];
      strcpy(name,n);
      age=d;
   }
    ~girl()    {delete name;}
};
void boy::disp(girl &x)
     {cout<<"boy\'s name is:
"<<name<<",age:"<<age<<"\n";
     cout<<"girl\'s name is:
"<<x.name<<",age:"<<x.age<<"\n";
    }
int main()
{
     boy b("yao ming",25);
     girl e("wang nan",22);
     b.disp(e);
     return 0;
}
```

友元关系的特点：

- 单向性—不具有交换性
  - “我说你是我的朋友，你可以不说我是你的朋友”
- 无传递性
  - “朋友的朋友不是我的朋友”

类的组合

- C++中，类对象常常可以作为另一个类的成员。即对象成员(子对象)
- 类中的数据成员是另一个类的对象。
- 是软件重用的一种形式。通过这种形式可以将一个复杂对象化为若干简单对象的组合，可以将一个复杂的类化为若干简单类的组合。



"类中有对象"

```
class Point
{private:
   float x,y; //点的坐标
public:
   Point(float h,float v); //构造函数
   float GetX( ); //取X坐标
   float GetY( ); //取Y坐标
   void Draw( ); //在(x,y)处画点
};
//...函数的实现略
```

```
class Line
{
  private:
    Point  p1,p2; //线段的两个端点
  public:
    Line(Point a,Point b); //构造函数
    void Draw( ); //画出线段
};
//...函数的实现略
```

类的组合

- 构造函数设计（类+对象成员）
  - 原则：不仅要负责对本类中的基本类型成员数据赋初值，也要对对象成员初始化。
  - 当类中出现了子对象（对象成员）时，该类的构造函数要包含对子对象的初始化，通常采用成员初始化列表的方法来初始化子对象；
- 声明形式：

```
类名::类名(对象成员所需的形参，本类成员形参)
        ：对象1(参数)，对象2(参数)，……
{   本类初始化   }
```

成员初始化列表

类的组合

- **构造函数调用顺序：**
  - 先调用**内嵌对象**的构造函数（按内嵌时的声明顺序，先声明者先构造）
  - 然后调用本类的构造函数。（析构函数的调用顺序相反）
- 若调用缺省构造函数（即无形参的），则内嵌对象的初始化也将调用相应的缺省构造函数。

//例

```
#include <stdio.h>
#include <string.h>
#include <iostream>
using namespace std;
class MyString{
private:
  char *str;
public:
   MyString(char *s)
   {  str=new char[strlen(s)+1];
      strcpy(str,s);}
   void print()
   { cout<<str<<endl;}
   ~MyString()
   { delete str;}
};
```

```
class girl{
private:
    MyString name;
    int age;
public:
   girl(char *st,int ag):name(st)
   {   age=ag;   }
   void print()
   {   name.print();
       cout<<"age:"<<age<<endl;}
   ~girl() { }
};
```

子对象name

成员初始化列表

构造函数体

对子对象成员函数的调用

```
int main()
{
      girl obj("chen hao",25);
      obj.print();
      return 0;
}
```

- (1)声明一个含有对象成员的类，首先要创建各成员对象。`MyString name;`
- （2）girl类对象在调用构造函数初始化时，也要对对象成员进行初始化。

```
girl(char *st,int ag):name(st)
```

子对象的总结说明

- **初始化**
  - 子对象必须在**成员初始化列表**中初始化；
  - 建立一个对象时，它的所有子对象一起建立；
  - 先执行子对象构造函数，再执行对象的构造函数体；
- **构造与析构**
  - 析构函数的执行顺序与构造函数的执行顺序相反；
  - 构造函数的调用顺序仅与子对象在类中声明的顺序有关，而与成员初始化列表中给出的对构造函数的调用顺序无关；
  - 构造函数的成员初始化列表中未给出对子对象的调用，则表示使用子对象的缺省构造函数；

常类型

- **常类型**的对象必须进行初始化，而且不能被更新。
- **常引用**：被引用的对象不能被更新。

  const　类型说明符　&引用名
- **常对象**：必须进行初始化，不能被更新。

  类名 const　对象名
- **常对象成员**：常数据成员和常成员函数
- 类型说明符　函数名（参数表）const

Why using const? To protect shared data.

//例 常引用做形参

```
#include<iostream>
int add(const int & i, const int & j);
void main()
{    int a=20,b=30;
      cout<<a<<"+"<<b<<"="<<add(a,b)<<endl;
}
int add(const int & i, const int & j)//常引用做形参，在函数中不能更新i,j所引用的对象。
{   // i=i+20;
     return i+j;
}
```

//例 常对象举例

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

class sample{
     int n;
public:
    int m;
    sample(int i,int j)
    {m=i;n=j;}
    void setvalue(int i)
    {n=i;}
    void display()
    {cout<<"m="<<m<<endl;
     cout<<"n="<<n<<endl;}
};
int main()
{
    sample a(10,20);
    a.setvalue(40);
    a.m=30;
    a.display();
    return 0;
}
```

//例 常对象举例

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

class sample{
      int n;
public:
     int m;
     sample(int i,int j)
     {m=i;n=j;}
     void setvalue(int i)
     {n=i;}
     void display()
     {cout<<"m="<<m<<endl;
      cout<<"n="<<n<<endl;}
};
int main()
{
   sample const a(10,20);
   a.setvalue(40);//error
   a.m=30;//error
   a.display();//error只能
   //调用常成员函数
}
```

用const修饰的对象成员

- 常数据成员
  - 使用const说明的数据成员。
  - 在构造函数中只能通过初始化列表对常数据成员初始化。
  - 任何其他函数不能对常数据成员赋值。

```
#include<iostream>
using namespace std;
class Date{
      const int year;
      const int month;
      const int day;
public:
     Date(int y,int m,int d);
     void showDate();
     const int &r;
};
Date::Date(int y,int m,int
   d):year(y),month(m),day(d),r(year)
{ }
inline void Date::showDate()
{cout<<"r="<<r<<endl;
   cout<<year<<"."<<month<<"."<<day<<endl;}
```

```
int main()
{
     Date date1(2014,5,20);
     date1.showDate();
return 0;
}
```

用const修饰的对象成员

- **常成员函数**
  - 使用const关键字说明的成员函数。格式为：

    类型说明符　函数名（参数表）const;
  - const是函数类型的一个组成部分，因此在实现部分也要带const关键字。 const关键字可以被用于参与对重载函数的区分
  - 常成员函数可以访问常数据成员，也可以访问普通数据成员。
  - 常成员函数不能更新对象的数据成员，也不能调用普通成员函数。
- 通过常对象只能调用它的常成员函数。

```
#include<iostream>
using namespace std;
class Date{
    int year,month,day;
public:
    Date(int y,int m,int d);
    void showDate();
    void showDate() const;
};
Date::Date(int y,int m,int
  d):year(y),month(m),day(d)
{ }
void Date::showDate()
{ cout<<"showDate1:"<<endl;
  cout<<year<<"."<<month<<"."<<day<<endl;}
```

```
void Date::showDate() const
{ cout<<"showDate2:"<<endl;
  cout<<year<<"."<<month<<"."<<day<<endl;}
int main()
{
    Date date1(2011,4,1);
    date1.showDate();
    Date const date2(2011,5,1);
    date2.showDate();
 return 0;
}
```

```
showDate1:
2011.4.1
showDate2:
2011.5.1
```

本章小结

- 类定义的格式
- 对象的定义方法及成员的表示方法
- 对象的创建和释放

  构造函数和析构函数的功能和特点
- 成员函数的特点：

  重载性 内联性 设置参数的默认值
- 静态成员

  静态数据成员和静态成员函数的定义和特点

本章小结

- 友元

  友元函数和友元类的功能及定义方法

- 对象指针

  定义、赋值和用法

- 对象数组

  定义、赋值和元素表示及用法

- 类的组合

  子对象的定义、初始化（成员初始化1列表）和使用

- 常类型：常引用、常对象、常对象成员

练习题

1.写出下面程序的运行结果。

```
#include<iostream>
class example
{
public:
 example(int n)
 {  i=n;
     cout<<"Constructing\n"; }
 ~example()
 { cout<<"Destructing\n"; }
 int get_i()
 {  return i;  }
private:
 int i;
};
```

```
int sqr_it(example o)
{
   return
   o.get_i()*o.get_i();
}
int main()
{
   example x(10);
   cout<<x.get_i()<<endl;
   cout<<sqr_it(x)<<endl;
   return 0;
}
```

练习题

1.写出下面程序的运行结果。

```
#include<iostream>
class example
{
public:
 example(int n)
 {  i=n;
     cout<<"Constructing\n"; }
 ~example()
 { cout<<"Destructing\n"; }
 int get_i()
 {  return i;  }
private:
 int i;
};
```

```
int sqr_it(example &o)
{
   return
   o.get_i()*o.get_i();
}
int main()
{
   example x(10);
   cout<<x.get_i()<<endl;
   cout<<sqr_it(x)<<endl;
   return 0;
}
```

2.写出下面程序的运行结果。

```
#include<iostream>
using namespace std;
class aClass
{
    static int total;
public:
 aClass()
 { total++;  }
 ~aClass()
 {  total--; }
 int get_total()
 { return total; }
};
int aClass::total=0;
int main()
{
   aClass o1,o2,o3;
   cout<<o1.get_total()<<"
   objects in existence\n";
    aClass *p;
   p=new aClass;
   cout<<o1.get_total();
   cout<<" objects in existence
   after allocation\n";
   delete p;
    cout<<o1.get_total();
   cout<<" objects in existence
   after deletion\n";
   return 0;
}
```

3.写出下面程序的运行结果。

```
#include <stdio.h>
#include <string.h>
#include <iostream>
using namespace std;

class Sample {
  int x, y;

 public:
  Sample() { x = y = 0; }
  Sample(int a, int b) {
    x = a;
    y = b;
  }
  ~Sample() {
    if (x == y) {
      cout << "x=y" << endl;
    } else {
      cout << "x !=y" << endl;
    }
  }

  void disp() { cout << "x="
  << x << " , y=" << y <<
  endl; }
};

int main() {
  Sample s1(10, 30), s2;
  s1.disp();
  s2.disp();
  return 0;
}
```

感谢

声明

- 本课程PPT部分内容收集自网上公开资料，如有侵权，请邮件联系我们
- 如您是承担了类似课程的教师，需要原始pptx文件，请邮件联系张书航或李同文
- 联系方式在首页