

嵌入式系统工程师





类与对象高级应用

- 对象数组
- this指针
- 枚举
- 静态成员
- 对象成员
- 友元
- 运算符重载函数

- 对象数组
- this指针
- 枚举
- 静态成员
- 对象成员
- 友元
- 运算符重载函数

3.1 对象数组

➤ 对象数组

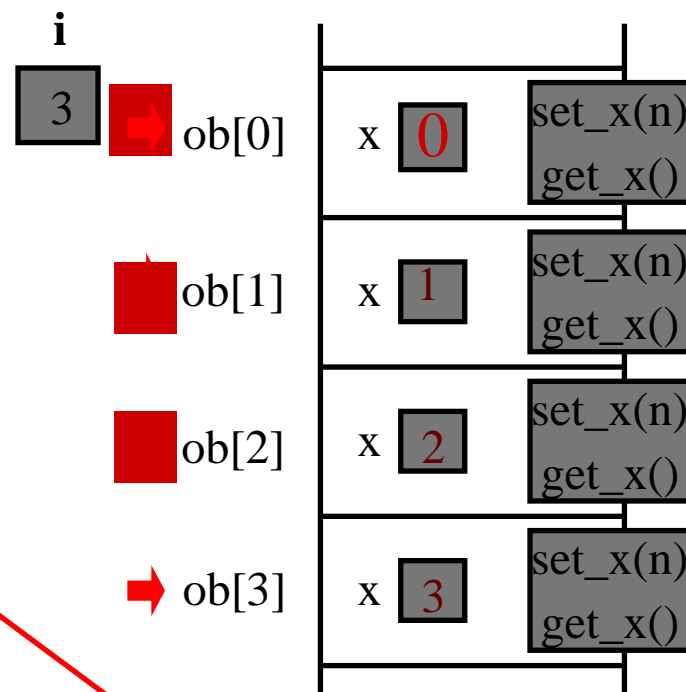
- 对象数组是指每一个数组元素都是一个单独的对象，也就是说，若一个类有若干个对象，我们把这一系列的对象用一个数组来存放

3.1 对象数组

例3.1对象数组的使用

```
#include <iostream.h>
class Exam{
    int x;
public:
    void setX(int n) { x=n; }
    int getX() { return x; }
};

int main() {
    Exam ob[4];    int i;
    for (i=0; i<4; i++)    ob[i].setX(i);
    for (i=0; i<4; i++)    cout<<ob[i].getX()<<endl;
}
```



➤ 对象数组由C++的系统缺省构造函数建立

3.1 对象数组

➤若类中含有构造函数，那么定义对象数组时，也可通过不带参数的构造函数或带有缺省参数的构造函数给对象数组元素赋值

例3.2 对象数组的初始化

- 对象数组
- this指针
- 枚举
- 静态成员
- 对象成员
- 友元
- 运算符重载函数

3.2 this指针

- C++提供了一个特殊的对象指针——this指针
- 它是成员函数所属对象的指针，它指向类对象的地址。成员函数通过这个指针可以知道自己属于哪一个对象。this指针是一种隐含指针，它隐含于每个类的成员函数中

3.2 this “幕后”

this指针“幕后”工作机理。

```
class ABC{
private:
    char a;  int b;
public:
    void init(char ma,int mb)
    {
        a=ma;
        b=mb;
    }
    //...
};

int main()
{
    ABC ob;
    ob.init('x',12);
    //...
}
```

C++在编译过程中做了些简单的转换工作：

- 1、相应地把被调用的成员函数的定义形式进行变化。
- 2、把成员函数的调用形式进行变化。

init(ABC *this,char ma,int mb)
this->a=ma;
this->b=mb;

init(&ob,'x',12);

3.2 this指针

定义形式发生变化

class Exam{ 例3.3 this指针

int x;

public: void load(int val)

{ this->x=val;} //与x=val等价

int getX()

{ return this->x;} //与return x;等价

};

int main()

{ Exam ob, ob1;

ob.load(100);

cout<<ob.getX();

ob1.load(200);

cout<<ob1.getX();

void load(Exam *this,int val);

int getX(Exam *this);

调用形式发生变化

ob.load(&ob,100);

ob.getX(&ob);

ob1.load(&ob1,200);

ob1.getX(&ob1);

3.2 this指针的使用

➤ 当形式参数与数据成员同名时:

```
class ABC {  
public:  
    int x;  
    void set(int x)  
    { this->x = x; } //形参与数据成员同名  
}
```

C++对变量的检查顺序:

局部变量、类的数据成员、全局变量

3.2 this指针的使用

➤ 函数返回调用该函数的对象的引用

例如在类中，有设置函数set()和显示函数show()理想情况下，希望能够将一些操作序列连接成一个单独的表达式：

如： `a.set(13, 23).show();`

等价于：

`a.set(13, 23);`

`a.show();` 例3.4 this指针的使用

3.3 向函数传递对象

- 将对象作为参数传递给函数有三种情况，分别为：

```
fun1 (ABC p)      {...}           //值方式  
fun2 (ABC *p)     {...}           //指针方式  
fun3 (ABC &p)     {...}           //引用方式
```

```
int main () {  
    ABC p1, p2, p3;  
  
    .....  
    fun1 (p1); //以“值方式”传递对象， p1 不被修改  
    fun2 (&p2); //以“指针方式”传递对象， p2 被修改  
    fun3 (p3); //以“引用方式”传递对象， p3 被修改  
}
```

3.3 值方式

“值方式”传递对象:

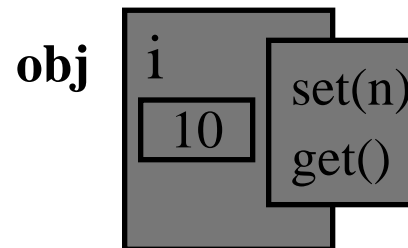
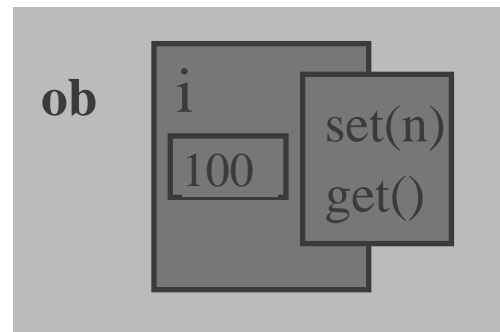
```
class Tr {  
    int i;  
public:  
    Tr(int n) {i=n;}  
    void set(int n) {i=n;}  
    int get() {return i;}  
};  
void Sqr(Tr ob)  
{  
    ob.set(ob.get()*ob.get());  
    cout<<"ob. i="<<ob.get()<<endl;  
}  
  
int main() {  
    Tr obj(10);  
    Sqr(obj);  
    cout<<"obj. i="<<obj.get()<<endl;  
}
```

输出结果:

ob.i=100

obj.i=10

对形参对象的任何修改不会影响到实参对象本身



3.3 指针方式

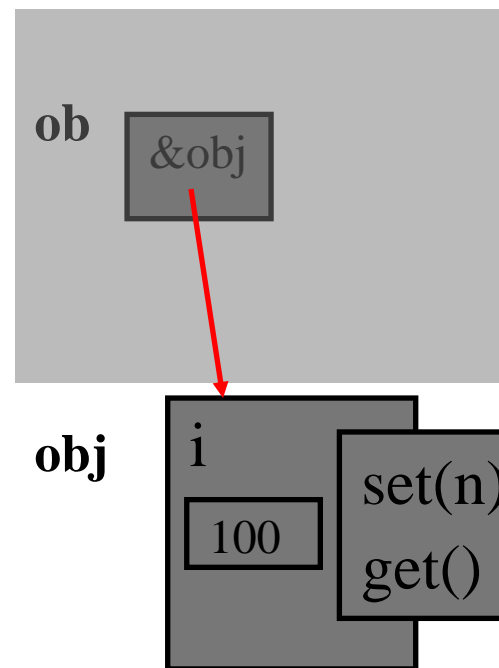
```
class Tr {  
    int i;  
public:  
    Tr(int n) {i=n;}  
    void set(int n) {i=n;}  
    int get() {return i;}  
};  
void Sqr(Tr *ob)  
{ob->set(ob->get() * ob->get());  
  cout<<"ob. i="<<ob->get()<<endl;  
}  
int main() {  
    Tr obj(10);  
    Sqr(&obj);  
    cout<<"obj. i="<<obj.get()<<endl;  
}
```

输出结果:

ob.i=100

obj.i=100

对形参对象的任何修改
将会影响到实参对象



3.3 引用方式

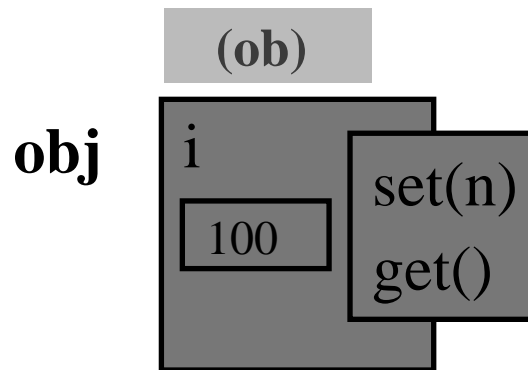
```
class Tr {  
    int i;  
public:  
    Tr(int n) {i=n;}  
    void set(int n) {i=n;}  
    int get() {return i;}  
};  
void Sqr(Tr &ob)  
{ ob.set(ob.get()*ob.get());  
  cout<<"ob. i="<<ob.get()<<endl;  
}  
int main() {  
    Tr obj(10);  
    Sqr(obj);  
    cout<<"obj. i="<<obj.get()<<endl;  
}
```

输出结果:

ob.i=100

obj.i=100

对形参对象的任何修改
将会影响到实参对象



- 对象数组
- this指针
- 枚举
- 静态成员
- 对象成员
- 友元
- 运算符重载函数

- 如果我们希望某些常量只在类中有效，那么使用枚举是一个很好的解决方法。
- 定义枚举类型时，枚举量的值必须是整数，不能是浮点数和字符串。
- 在默认情况下，枚举量的值从0开始，如果没有初始化，那么它的值比前面的值大1。
- 可以创建多个值相同的枚举量

```
enum bits {one=1, two=2, four=4};  
enum bigstep {first, second=100, third};  
enum {zero, null=0, size, num=1}; // 匿名枚举类型
```

```
class A {  
public:  
    // 匿名枚举类型  
    enum {SIZE1=100, SIZE2=200};  
    int array[SIZE1];  
};
```

```
class B {  
public:  
    enum bits {one=1, two=2, four=4};  
    bits flag; // 等价 int flag;  
};
```

例3.5枚举的使用

- 对象数组
- this指针
- 枚举
- 静态成员
- 对象成员
- 友元
- 运算符重载函数

3.4 静态成员

- 在类定义中，它的成员（包括数据成员和成员函数）可以用关键字 `static` 声明为静态的，这些成员称为静态成员
- 静态成员的特性：
 - 不管这个类创建了多少个对象，静态成员只有一个拷贝，这个拷贝被所有属于这个类的对象共享
- 静态成员包括：
 - 静态数据成员
 - 静态成员函数

3.4 静态数据成员

- 在一个类中，若将一个数据成员声明为 `static`，这种成员称为静态数据成员。
(静态数据成员在类内声明，在类外定义)
- 与一般的数据成员不同，无论建立了多少个对象，都只有一个静态数据的拷贝
- 可以认为该静态数据是属于该类的，而不是具体的属于某一个对象

3.4 静态数据成员

► 例3.6 静态数据成员

输出结果:

Student1 count=1 _{s1}

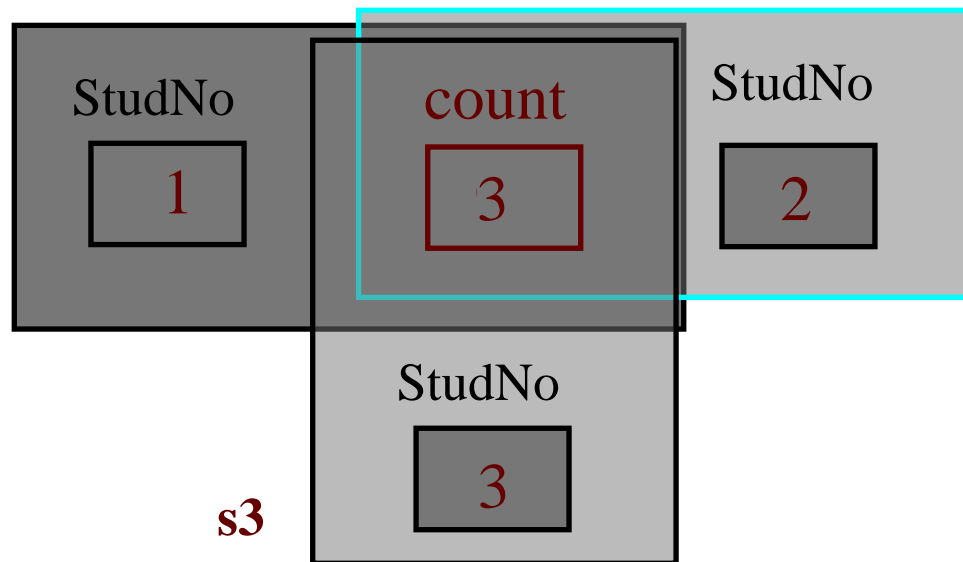
Student1 count=2

Student2 count=2

Student1 count=3

Student2 count=3

Student3 count=3



3.4 静态数据成员

➤ 说明:

- 静态数据成员属于类(准确地说,是属于类中一个对象集合),而不像普通数据成员那样属于某一对对象,因此可以使用“类名::”访问静态的数据成员。但也可以通过“对象名.”访问
- 静态数据成员不能在类中进行初始化,因为在类中不给它分配内存空间,必须在类外的其它地方为它提供定义。一般在main() 开始之前、类的声明之后的特殊地带为它提供定义和初始化。缺省时,静态成员初始为0

3.4 静态数据成员

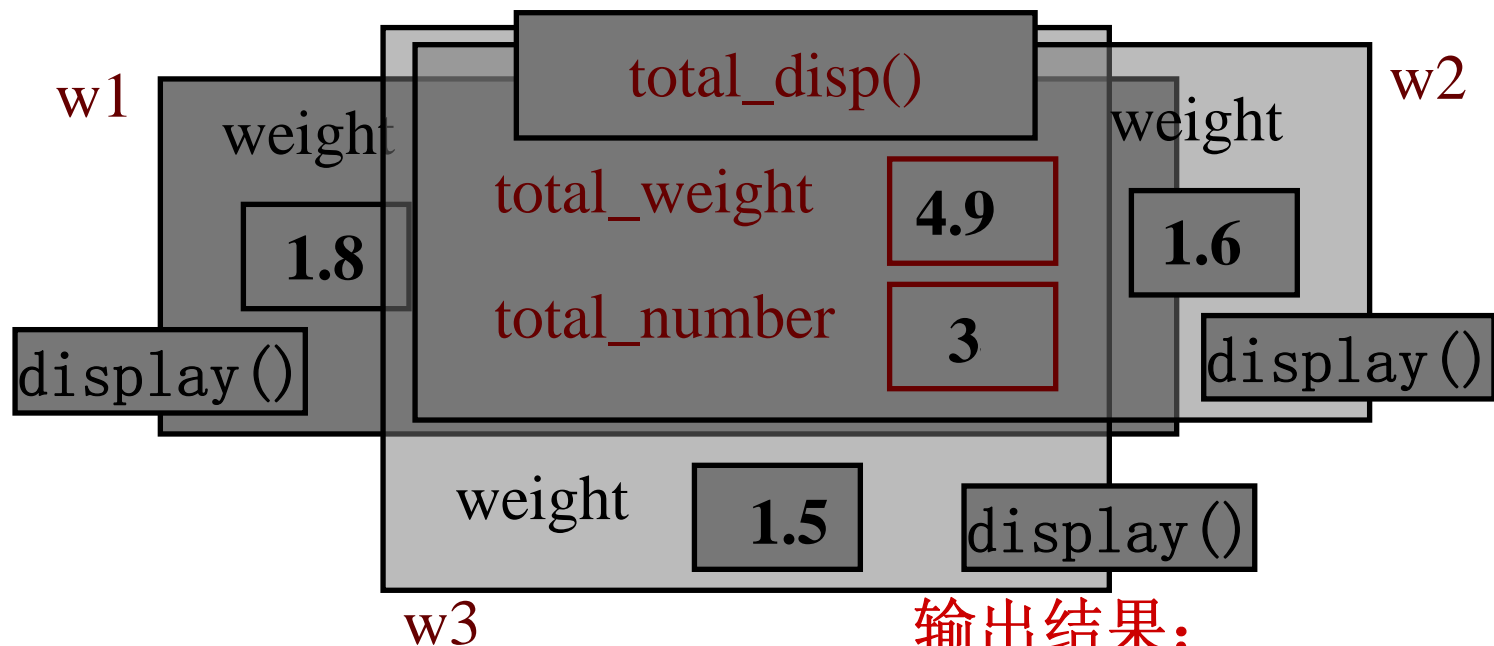
- 静态数据成员与静态变量一样,是在编译时创建并初始化。它在该类的任何对象被建立之前就存在,它可以在程序内部不依赖于任何对象被访问
- C++支持静态数据成员的一个主要原因是可以不必使用全局变量。依赖于全局变量的类几乎都是违反面向对象程序设计的封装原理的。静态数据成员的主要用途是定义类的各个对象所公用的数据,如统计总数、平均数等

3.4 静态成员函数

- 在类定义中，前面有static说明的成员函数称为静态成员函数
- 静态成员函数是一个成员函数，因此可以使用“类名::”和“对象名.”两种方法访问静态成员函数
- 其次，静态成员函数是一种特殊的成员函数，它不属于某一个特定的对象。一般而言，静态成员函数访问的基本上是静态数据成员或全局变量

3.4 静态成员函数

例3.7 使用静态成员函数访问静态数据成员



输出结果:

小猫的重量是:1.8磅

小猫的重量是:1.6磅

小猫的重量是:1.5磅

3只小猫的总重是:4.9磅

3.4 静态成员函数

➤ 说明:

- 静态成员函数可以在类内定义。也可以在类内声明，类外定义。在类外定义时，不要用 `static` 前缀
- 编译系统将静态成员函数限定为内部连接，也就是说，与现行文件相连接的其它文件中的同名函数不会与该函数发生冲突，维护了该函数使用的安全性，这是使用静态成员函数的一个原因
- 使用静态成员函数的另一个原因是，可以用它在建立任何对象之前处理静态数据成员，这是普通成员函数不能实现的功能

3.4 静态成员函数

- 在一般的成员函数中都隐含有一个this指针，用来指向对象自身，而在静态成员函数中没有this指针，因为它不与特定的对象相联系，调用时使用：

类名::静态成员函数名()

如：SmallCat::totalDisp()，当然使用：

对象.静态成员函数名()

也是正确的。如：w1.totalDisp()

- 一般而言，静态成员函数不能访问类中的非静态成员

- 对象数组
- this指针
- 枚举
- 静态成员
- 对象成员
- 友元
- 运算符重载函数

3.5 类对象作为成员

- 在类定义中定义的数据成员一般都是基本的数据类型。但是类中的成员也可以是对象，叫做**对象成员**。使用对象成员时需要注意的问题是**构造函数的定义方式**，即类内部对象的初始化问题
- 含有对象成员类，其构造函数和不含对象成员的构造函数有所不同，例如有以下的类：

```
class X
{
    类名1 成员名1;
    类名2 成员名2;
    .....
};
```


3.5 类对象作为成员

- 一般来说，类X的构造函数的定义形式为：

```
X::X(参数表0):成员名1(参数表1),...,成员名n(参数n表)
```

```
{ //构造函数体  
}
```

对象成员的初始化列表，
一般来自参数表0。

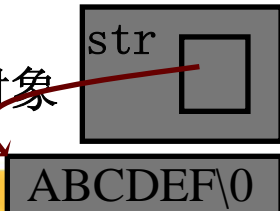
- 当调用构造函数X::X()时，首先按各对象成员在类定义中的顺序依次调用它们的构造函数，对这些对象初始化，最后再执行X::X()的函数体
- 析构函数的调用顺序与此相反

例3.8 对象作为类的成员

```

class string{
    char *str;
public:
    string(char *s)
    { str=new char[strlen(s)+1];
      strcpy(str,s);
      cout<< "构造string\n ";
    }
    void print(){ cout<<str<<endl;}
    ~string()
    { cout<< "析构string\n ";
      delete str;
    }
};
    
```

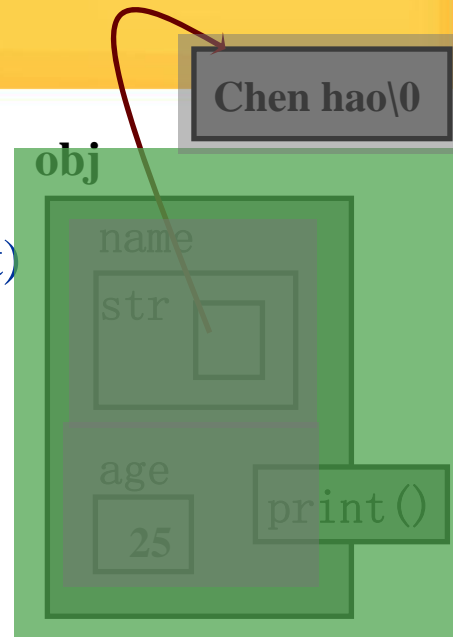
string的对象



```

class girl{
    string name; int age;
public:
    girl(char *st,int ag):name(st)
    { age=ag;
      cout<< "构造girl\n ";
    }
    void print()
    { name.print();
      cout<<"age:"<<age<<endl;
    }
    ~girl(){ cout<< "析构girl\n "; }
};

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



输出结果:

构造string
 构造girl
 Chen hao
 age:25
 析构girl
 析构string

3.5 类对象作为成员

➤ 说明:

- girl类对象在调用构造函数进行初始化的同时，也要对对象成员进行初始化，因为它也是属于此类的成员。因此在写类girl的构造函数时，也缀上了对对象成员的初始化:

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

于是在调用girl的构造函数进行初始化时，也给对象成员name赋上了初值

这里需要注意的是：在定义类girl的构造函数时，必须缀上其对象成员的名字name，而不能缀上类名，即：

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

是错误的，因为在类girl中是类string的对象name作为成员，而不是类string作为其成员

➤ 练习

- 对象数组
- this指针
- 枚举
- 静态成员
- 对象成员
- 友元
- 运算符重载函数

- 类的主要特点之一是数据隐藏，即类的私有成员只能在类定义的范围内使用，也就是说私有成员只能通过它的成员函数来访问
- 但是，有时候需要在类的外部访问类的私有成员。为此，就需要寻找一种途径，在不放弃私有数据安全性的情况下，使得类外部的函数或类能够访问类中的私有成员

- C++中友元作为实现这个要求的手段
- C++中的友元为数据隐藏这堵不透明的墙开了一个小孔，外界可通过这个小孔窥视类内部的秘密，友元是一扇通向私有成员的后门
- 友元可分为：友元函数，友元成员，友元类

3.6 友元函数

- 友元函数不是当前类的成员函数，而是独立于当前类的外部函数，但它可以访问该类的所有对象的成员，包括私有成员和公有成员
- 在类定义中声明友元函数时，需在其函数名前加上关键字 `friend`。此声明可以放在公有部分，也可以放在私有部分。友元函数可以定义在类的内部，也可以定义在类的外部

3.6 友元函数

► 例3.9 使用友元函数

```
class girl{  
    char *name;    int age;  
public:  
    //...  
    friend void disp(girl &); //声明为友元函数  
    //...  
};
```

```
void disp(girl &x) //定义友元函数  
{  
    cout<<"girl\'s name  
    is: "<<x.name<<" , age: "<<x.age<<endl;  
}
```

```
int main() {  
    girl e("Chen Xingwei",18);  
    disp(e); //调用友元函数  
}
```



➤ 说明:

- 友元函数虽然可以访问类对象的私有成员，但它毕竟不是成员函数。因此，在类的外部定义友元函数时，不必像成员函数那样，在函数名前加上“类名::”
- 友元函数一般带有一个该类的入口参数。因为友元函数不是类的成员，所以它不能直接引用对象成员的名称，也不能通过this指针引用对象的成员，它必须通过作为入口参数传递进来的对象名或对象指针来引用该对象的成员

3.6 友元函数

- 当一个函数需要访问多个类时，友元函数非常有用，普通的成员函数只能访问其所属的类，但是多个类的友元函数能够访问相应的所有类的数据

例3.10 使用友元函数访问两个不同的类

- 友元函数通过直接访问对象的私有成员，提高了程序运行的效率。在某些情况下，如运算符被重载时，需要用到友元。但是友元函数破坏了数据的隐蔽性，降低了程序的可维护性，这与面向对象的程序设计思想是背道而驰的，因此使用友元函数应谨慎

3.6 友元成员

- 除了一般的函数可以作为某个类的友元外, 一个类的成员函数也可以作为另一个类的友元, 这种成员函数不仅可以访问自己所在类对象中的私有成员和公有成员, 还可以访问friend声明语句所在类对象中的私有成员和公有成员, 这样能使两个类相互合作、协调工作, 完成某一任务

例3.11 使用友元成员函数访问另一个类

3.6 友元成员

➤ 说明:

- 一个类的成员函数作为另一个类的友元函数时，必须先定义这个类。例如上例中，类boy的成员函数为类girl的友元函数，必须先定义类boy。并且在声明友元函数时，要加上成员函数所在类的类名，如：

```
friend void boy::disp(girl &);
```

- 程序中还要使用“向前引用”，因为函数disp()中将girl &作为参数，而girl要晚一些时候才定义

3.7 友元类

- ▶ 不仅函数可以作为一个类的友元，一个类也可以作为另一个类的友元。这种友元类的声明方法是在另一个类声明中加入语句“friend 类名;”，其中的“类名”即为友元类的类名。此语句可以放在公有部分也可以放在私有部分

- ▶ 例如:

```
class Y{  
    //.....  
};  
class X{  
    //.....  
    friend Y;  
    //.....  
};
```

- 当一个类被声明为另一个类的友元时，它的所有成员函数都成为另一个类的友元函数，这就意味着为友元的类中的所有成员函数都可以访问另一个类的私有成员

例3.12 友元类

3.6 友元类

➤ 说明:

- 友元关系是单向的，不具有交换性（我是你的朋友，不能推断出：你是我的朋友）
- 友元关系也不具有传递性（我是你的朋友，你是他的朋友，不能推断出：我是他的朋友）

- 对象数组
- this指针
- 枚举
- 静态成员
- 对象成员
- 友元
- 运算符重载函数

3.7 运算符重载函数

➤ 作用:

- 使复杂函数的理解更直观, 程序更加简单易懂

➤ 运算符重载函数的形式是:

- 返回类型 `operator` 运算符符号 (参数说明)
{ //函数体的内部实现 }
- 至少有一个参数是自定义类型(数组, 类)
- 如果是单目运算符, 只传一个参数
- 如果是双目运算符, 传两个参数

➤ c++也规定了一些运算符不能够自定义重载

- 例如 `::`、`sizeof`、`..`、`?:`

3.7 运算符重载函数

► 常见运算符重载函数的使用

- 一种是作为类的友元函数进行使用

如: `friend void operator + (Test&, Test&);`

例: 例3.13 作为类的友元函数

- 一种则是作为类的成员函数进行使用

如: `Temp operator + (Temp &t) { ; }`

例: 例3.14 作为类的成员

注意: C++会隐藏第一个参数, 转而取代的是一个
this指针

本章主要介绍了C++中最基本、也是最主要、最重要的概念：类。由于类的引入，使得C++具备了面向对象的程序设计的能力。在程序中使用类，必须深刻理解类的构造函数和析构函数、掌握对象数组与对象指针、理解如何向函数传递对象，了解静态成员、友元等基本概念。



值得信赖的教育品牌

Tel: 400-705-9680 , Email: edu@sunplusapp.com , BBS: bbs.sunplusedu.com

