
面向对象程序设计

420420

对象数组和对象指针

- ◆ 1、对象数组的定义和使用
- ◆ 2、指向对象的指针

- ▶ 可以简单的理解：类就是自定义的数据类型，而对象就是类的实例。因此也可以构造对象数组和对象的指针。

30.1 对象数组的定义和使用

- ▶ 将具有相同类类型的对象有序地集合在一起便构成了对象数组，以一维对象数组为例，其定义形式为：

```
类名 对象数组名[常量表达式];
```

- ▶ 一维对象数组有时也称为对象向量，它的每个元素都是相同类类型的对象。
- ▶ 例如表示平面上若干个点的，可以这样定义：

```
Point points[100]; //表示100个点
```

30.1 对象数组的定义和使用

- ▶ 关于对象数组的说明：
 - ▶ (1) 在建立对象数组时，需要调用构造函数。如果对象数组有100个元素，就需要调用100次构造函数。
 - ▶ (2) 如果对象数组所属类有带参数的构造函数时，可用初始化列表按顺序调用构造函数，使用复制初始化来初始化每个数组元素。

```
Point P[3]={Point(1,2),Point(5,6),Point(7,8)}; //三个实参
```

- ▶ (3) 如果对象数组所属类有单个参数的构造函数时，定义数组时可以直接在初值列表中提供实参。

```
Student S[5]={20,21,19,20,19}; //Student类只有一个数据成员
```

- ▶ (4) 对象数组创建时若没有初始化，则其所属类要么有合成默认构造函数（此时无其他的构造函数），要么定义无参数的构造函数或全部参数为默认参数的构造函数（此时编译器不再合成默认构造函数）。
- ▶ (5) 对象数组的初始化式究竟是什么形式，本质上取决于所属类的构造函数。因此，需要明晰初始化实参与构造函数形参的对应关系，避免出现歧义性。
- ▶ (6) 如果对象数组所属类含有析构函数，那末每当建立对象数组时，按每个元素的排列顺序调用构造函数；每当撤销数组时，按相反的顺序调用析构函数。

30.2 指向对象的指针

- ▶ 在建立对象时，编译器会为每一个对象分配一定的存储空间，以存放其成员。对象内存单元的起始地址就是对象的指针。可以定义一个指针变量，用来存放对象的指针。指向类对象的指针变量的定义形式为：

```
类名 *对象指针变量名=初值;
```

►例如:

```
class Time {  
public:  
    Time(int h=0,int m=0,int s=0):  
        hour(h),minute(m),second(s) { } //构造函数  
    void set(int h=0,int m=0,int s=0)  
        { hour=h,minute=m,second=s; }  
    int hour, minute, second; //公有的数据成员  
};  
Time now(12,0,0), *pt; //指向对象的指针变量  
pt=&now; //指向对象
```


30.2 指向对象的指针

- ▶ 可以通过对象指针访问对象和对象的成员。如：

```
pt->set(13,13,0);  
pt->hour=1;
```

对象数组和对象指针

◆ 3、类成员指针

◆ 4、this指针

- 对象的成员要占用存储空间，因此也有地址，可以定义指向对象成员的指针变量，一般形式为：

```
数据成员类型 *指针变量名=初值;
```

- 例如：

```
int *ptr=&now.hour; //指向对象数据成员的指针变量
```

- ▶ C++比C语言有更严格的静态类型，更加强调类型安全和编译时检查。
- ▶ 因此，C++的指针被分成数据指针、函数指针、数据成员指针、成员函数指针四种，而且不能随便相互转换。其中前两种是C语言的，称为普通指针（ordinary pointer）；后两种是C++专门为类扩展的，称为成员指针（pointer to member）。
- ▶ 成员指针与类的类型和成员的类型相关，它只应用于类的非静态成员。由于静态类成员不是任何对象的组成部分，所以静态成员指针可用普通指针。

▶ 1. 数据成员指针

▶ 定义数据成员指针的一般形式为：

数据成员类型 类名::*****指针变量名 = 成员地址初值;

▶例如:

```
class Data { //Data类
public:
    typedef unsigned int index; //类型成员
    char get() const; //成员函数
    char get(index st, index eb) const; //成员函数重载
    string content; //数据成员
    index cursor,top,bottom; //数据成员
};
```

- ▶ 指向content的指针的完全类型是“指向string类型的Data类成员的指针”，即：

```
String Data::*ps=&Data::content; //指向Data::content的成员指针
```

▶ 2. 成员函数指针

▶ 定义成员函数的指针时必须确保在三个方面与它所指函数的类型相匹配：

▶ ① 函数形参的类型和数目，包括成员是否为const。

▶ ② 返回类型。

▶ ③ 所属类的类型。

▶ 定义的一般形式为：

返回类型 (类名::*指针变量名)(形式参数列表)=成员函数地址初值;

▶ 或

返回类型 (类名::*指针变量名)(形式参数列表) **const** =成员函数地址初值;

- ▶ 例如 “char get() const”成员函数的指针可以这样定义和初始化：

```
char (Data::*pmf)() const = &Data::get;  
//pmf指向Data::get() 成员函数的指针
```

- ▶ 可以为成员指针使用类型别名，例如：

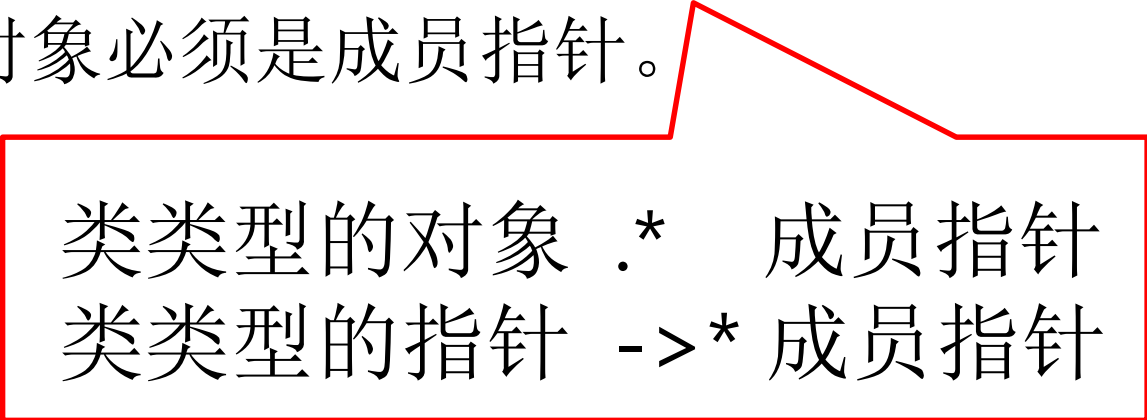
```
typedef char (Data::*GETFUNC)(Data::index,Data::index) const;  
//类型别名GETFUNC
```

- ▶ 这样指向get成员函数的指针的定义可以简化为：

```
GETFUNC pfget = &Data::get; //定义 GETFUNC 型成员函数指针pfget
```

3.使用类成员指针

- ▶ ①通过对象成员指针引用（.*）可以从类对象或引用及成员指针来间接访问类成员，或者通过指针成员指针引用（->*）可以从指向类对象的指针及成员指针访问类成员。
- ▶ 对象成员指针引用运算符（.*）左边的运算对象必须是类类型的对象，指针成员指针引用运算符（->*）左边的运算对象必须是类类型的指针，两个运算符的右边运算对象必须是成员指针。



类类型的对象	.*	成员指针
类类型的指针	->*	成员指针

30.3 类成员指针

表30-1 成员指针间接引用运算符

运算符	功能	目	结合性	用法
.*	对象成员指针引用	双目	自左向右	object.*member_pointer
->*	指针成员指针引用	双目	自左向右	pointer->*member_pointer

例如：

```
Data d, *p=&d; //指向对象d的指针p
int Data::*pt = &Data::top; //pt为指向数据成员top的指针
int k = d.top; //d对象里的top成员引用，直接访问对象，直接访问成员
k = d.*pt; //d对象里的pt成员指针引用，直接访问对象，间接访问成员，与上面等价
k = p->top; //p指向对象的top成员，指针成员引用，间接访问对象，直接访问成员
k = p->*pt; //指针成员指针引用，间接访问对象，间接访问成员，与上面等价
char (Data::*pmf)(int,int) const; //pmf为成员函数指针
pmf = &Data::get; //get函数的地址赋给pmf，pmf指向有两个参数的get函数
char c1 = d.get(0,0); //对象d的get成员函数，对象直接调用成员函数，与下面等价
char c2 = (d.*pmf)(0,0); //对象d通过成员函数指针pmf间接调用成员函数
char c3 = (p->*pmf)(0,0); //指针p间接引用对象通过成员函数指针pmf间接调用成员函数
```

- 除了静态成员函数外，每个成员函数都有一个额外的、隐含的形参 **this**。在调用成员函数时，编译器向形参 **this** 传递调用成员函数的对象的地址。例如成员函数：

```
void Point::set(int a,int b) { x=a, y=b; } //Point类里成员函数set定义
```

- 编译器实际上会重写这个函数为：

```
void Point::set(Point* const this,int a,int b)
{ this->x=a,this->y=b; }
```

- ▶ 对应的函数调用:

```
one.set(10,10); //调用成员函数
```

- ▶ 编译器实际上会重写这个函数调用为:

```
Point::set(&one,10,10); //调用成员函数
```

【例30.1】 this指针举例。

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4  class Point
5  {
6  public:
7      Point(int a,int b){x=a;y=b;}    //构造函数
8      void MovePoint(int a,int b){ x=x+a; y=y+b;}
9      void print(){ cout<<"x="<<x<<",y="<<y<<endl;}
10 private:
11     int x,y; //私有数据成员，坐标值
12 };
13 int main()
14 {
15     Point pt1(10,10);
16     pt1.MovePoint(2,2);    pt1.print();
17     return 0;
18 }
```

- ▶ 当对象pt1调用MovePoint(2,2)函数时，即将pt1对象的地址传递给了this指针。这时MovePoint函数的原型变成：

```
void MovePoint( Point *this, int a, int b);
```

- ▶ MovePoint函数实现便成为：

```
void MovePoint(int a, int b) {this->x+=a; this->y+=b;}
```

- ▶ MovePoint函数体等价于：

```
pt1.x+=a; pt1.y+=b;
```


- ▶ 什么时候会用到this指针？
 - ▶ (1) 在类的非静态成员函数中返回类对象本身的时候，直接使用 `return *this;`
 - ▶ (2) 当参数与数据成员名相同时，如 `this->n = n` （不能写成 `n=n`）。

30.4 this指针

【例30.2】 this指针举例。

```
1  class point
2  {
3  public:
4      point(float x,float y)    // 构造函数
5      {
6          this->x=x;           //this->x 表示private中声明的 x;
                                // x 表示构造函数point(float x,float y)中的 x。
7          this->y=y;           //this->y 表示private中声明的 y;
                                // y表示构造函数point(float x,float y)中的 y。
8      }
9  private:
10     float x,y;               // 私有数据成员，坐标值
11 };
```

- ▶ this指针的const限定
- ▶ 假设Point类有getX这样一个非static函数：

```
double Point::getX();
```

- ▶ 编译以后形式如下：

```
double getX(Point *const this);
```

- ▶ 可以看出，this指针的指向不允许改变，所以this指针本身就是const指针。

- ▶ 如果成员函数是常函数也就是下面的定义：

```
double Point::getX() const;
```

- ▶ 编译后会变成：

```
double getX(const Point *const this);
```

- ▶ 可以看出，既不允许改变this指针的指向，也不允许改变this指向的内容。