

# C++期末复习总结

# 基本数据类型

Type	Size	数值范围
无值型void	0 byte	无值域
布尔型bool	1 byte	true false
有符号短整型short [int] /signed short [int]	2 byte	-32768~32767
无符号短整型unsigned short [int]	2 byte	0~65535
有符号整型int /signed [int]	4 byte	-2147483648~2147483647
无符号整型unsigned [int]	4 byte	0~4294967295
有符号长整型long [int]/signed long [int]	4 byte	-2147483648~2147483647
无符号长整型unsigned long [int]	4 byte	0~4294967295
long long	8 byte	0~18446744073709552000
有符号字符型char/signed char	1 byte	-128~127
无符号字符型unsigned char	1 byte	0~255
宽字符型wchar_t (unsigned short.)	2 byte	0~65535
单精度浮点型float	4 byte	-3.4E-38~3.4E+38
双精度浮点型double	8 byte	1.7E-308~1.7E+308
long double	8 byte	



# 函数的参数传递

## 一. 值传递

1. 利用值传递方式，实际上是把实参的内容复制到形参中，实参和形参是存放在两个不同的内存空间中。在函数体内对形参的一切修改对实参都没有影响
2. 如果形参是类的对象，利用值传递的话每次都要调用类的构造函数构造对象，效率比较低

## 二. 指针传递（地址传递）

1. 当进行指针传递的时候，形参是指针变量，实参是一个变量的地址或者是指针变量，调用函数的时候，形参指向实参的地址。
2. 指针传递中，函数体内可以通过形参指针改变实参地址空间的内容。

## 三. 引用传递

1. 引用实际上是某一个变量的别名，和这个变量具有相同的内存空间。
2. 实参把变量传递给形参引用，相当于形参是实参变量的别名，对形参的修改都是直接修改实参。



# 指针与引用

从概念上讲。指针从本质上讲就是存放变量地址的一个变量，在逻辑上是独立的，它可以被改变，包括其所指向的地址的改变和其指向的地址中所存放的数据的改变。

而引用是一个别名，它在逻辑上不是独立的，它的存在具有依附性，所以引用必须在一开始就被初始化，而且其引用的对象在其整个生命周期中是不能被改变的（自始至终只能依附于同一个变量）。



# 指针与引用

在C++中，指针和引用经常用于函数的参数传递，然而，指针传递参数和引用传递参数是有本质上的不同的：

指针传递参数本质上是值传递的方式，它所传递的是一个地址值。值传递过程中，被调函数的形式参数作为被调函数的局部变量处理，即在栈中开辟了内存空间以存放由主调函数放进来的实参的值，从而成为了实参的一个副本。值传递的特点是被调函数对形式参数的任何操作都是作为局部变量进行，不会影响主调函数的实参变量的值。（这里是在说实参指针本身的地址值不会变）

而在引用传递过程中，被调函数的形式参数虽然也作为局部变量在栈中开辟了内存空间，但是这时存放的是由主调函数放进来的实参变量的地址。被调函数对形参的任何操作都被处理成间接寻址，即通过栈中存放的地址访问主调函数中的实参变量。正因为如此，被调函数对形参做的任何操作都影响了主调函数中的实参变量。



# 指针与引用

## ★相同点：

- 都是地址的概念；

指针指向一块内存，它的内容是所指内存的地址；而引用则是某块内存的别名。

## ★不同点：

- 指针是一个实体，而引用仅是个别名；
- 引用只能在定义时被初始化一次，之后不可变；指针可变；引用“从一而终”，指针可以“见异思迁”；
- 引用没有const，指针有const，const的指针不可变；（具体指没有int& const a这种形式，而const int& a是有的，前者指引用本身即别名不可以改变，这是当然的，所以不需要这种形式，后者指引用所指的值不可以改变）
- 引用不能为空，指针可以为空；
- “sizeof (引用)”得到的是所指向的变量(对象)的大小，而“sizeof (指针)”得到的是指针本身的大小；
- 指针和引用的自增(++ )运算意义不一样；
- 引用是类型安全的，而指针不是（引用比指针多了类型检查）



# 类的定义与实现

Class 类名

```
{  
    public:  
        //公共行为或属性  
    private:  
        //公共行为或属性  
};
```

结束时的分号不能省

1>. 在类定义时定义成员函数  
返回类型 函数名 (形参列表)

```
{  
    //函数体  
}
```

2>. 在类外定义成员函数

返回类型 类名::函数名 (形参列表) 必须通过作用域修饰符限定函数属于哪个类

```
{  
    //函数体  
}
```



# 构造函数与析构函数

## 1. 构造函数的作用

**构造函数**主要用来在创建对象时完成对对象属性的一些初始化等操作, 当创建对象时, 对象会自动调用它的构造函数。一般来说, 构造函数有以下三个方面的作用:

- 给创建的对象建立一个标识符;
- 为对象数据成员开辟内存空间;
- 完成对象数据成员的初始化。

## 2. 默认构造函数

当用户没有显式的去定义构造函数时, 编译器会为类生成一个默认的构造函数, 称为 "**默认构造函数**", 默认构造函数不能完成对象数据成员的初始化, 只能给对象创建一标识符, 并为对象中的数据成员开辟一定的内存空间。

## 3. 构造函数的特点

无论是用户自定义的构造函数还是默认构造函数都主要有以下特点:

- ①. 在对象被创建时自动执行;
- ②. 构造函数的函数名与类名相同;
- ③. 没有返回值类型、也没有返回值;
- ④. 构造函数不能被显式调用。





# 构造函数与析构函数

- 当用户显示的定义了构造函数时，**必须显式定义默认构造函数**。
- 构造函数可以重载
- 在进行构造函数的重载时要注意重载和参数默认的关系要处理好，避免产生代码的二义性导致编译出错。

```
Point(int x = 0, int y = 0)    //默认参数的构造函数
```

```
{  
    xPos = x;  
    yPos = y;  
}
```

```
Point()        //重载一个无参构造函数
```

```
{  
    xPos = 0;  
    yPos = 0;  
}
```



# 构造函数初始化列表与函数体内赋值

- 1.初始化列表**：所有类非静态数据成员都可以在这里初始化，  
所有类静态数据成员都不能在这里初始化
- 2.构造函数体**：对于类非静态数据成员：  
const型成员不能在这里初始化  
引用型成员不能在这里初始化  
没有默认构造函数的成员不能在这里初始化  
对于类静态数据成员：  
可以在这里修改可修改的静态成员，但静态成员必须已经  
在类外部初始 化
- 3.类外初始化**：除一个特例外，所有类static数据成员必须在这里初始化，  
特例是类static const int数据成员可以在这里初始化，也可  
以在 成员的声明处初始化
- 4.类中声明时直接赋值**：类static const int数据成员可以选在这里初始  
化。



# 构造函数初始化列表与函数体内赋值

1 内部数据类型（char，int.....指针等）

2 类中const常量，必须在初始化列表中初始，不能使用赋值的方式初始化

3 无默认构造函数的继承关系中（必须在初始化列表中初始化）



# 数组和指针

指针	数组
保存数据的地址	保存数据
间接访问数据，首先取得指针的内容，把他作为地址，然后从这个地址提取数据。如果指针有一个下表[i],就把指针的内容加上i作为地址，从中提取数据。	直接访问数据，a[i]只是简单的以a+i为地址取得数据。
通常用于动态数据结构	通常用于存储固定数目且数据类型相同的元素
相关的函数为malloc()，free()	隐式分配和删除
通常指向匿名数据	自身即为数据名



# 声明和定义

定义	只能出现在一个地方	确定对象的类型并分配内存，用于创建新的对象。例如： <code>int myArray[100]</code>
声明	可以出现多次	描述对象的类型，用于指代其他地方定义的对象(例如在其他文件里) 例如 <code>extern int myArray[]</code>

声明相当于**普通**的声明：它所说的并非自身，而是描述其他地方创建的对象。  
定义相当于**特殊**的声明：它为对象分配内存。



# 什么时候数组和指针是相同的

规则1 表达式中的数组名（与声明不同）被编译器当做一个指向该数组第一个元素的指针

规则2 下表总是与指针的偏移量相同

规则3 在函数参数的声明中，数组名被编译器当做指向该数组第一个元素的指针

## 具体解读

规则1 “表达式中的数组名”就是指针

就是对数组下表的引用总是可以写成“一个指向数组起始地址的指针加上偏移量”

```
int a[10], *p, i=2;
```

```
p=a; p[i];
```

```
p=a; *(p+i);
```

```
p=a+i; *p;
```



# 什么时候数组和指针是相同的

规则2 c/c++语言把数组下标作为指针的偏移量

规则3 “作为函数参数的数组名”等同于指针

“类型的数组”的形参的声明应该调整为“类型的指针”。在函数形参定义这个特殊情况下，编译器必须把数组形式改写成指向数组第一个元素的指针形式。编译器只向函数传递数组的地址，而不是整个数组的拷贝。

## 数组与指针可交换性的总结

- 1 用 $a[i]$ 这样的形式对数组进行访问总是被编译器“改写”或解释为像 $*(a+i)$ 这样的指针访问。
- 2 指针始终就是指针。绝不能改写成数组。
- 3 在特定的上下文中，也就是它作为函数的参数（只用这种情况），一个数组的声明可以看做一个指针。作为函数参数的数组始终会被编译器修改为指向数组第一个元素的指针。
- 4 当把一个数组定义为函数的参数时，可以选择把它定义为数组也可以把它定义为指针。不管选择哪种方法，在函数内部事实上获得的都是一个指针。
- 5 在其它所用声明中，定义和声明必须相匹配。如果定义了一个数组，在其它文件对它进行声明时也必须把它声明为数组。指针也是如此。



# 常见的指针类型总结

`int p;` //这是一个普通的整型变量

`int *p;` //首先从p 处开始,先与\*结合,所以说明p 是一个指针,然后再与int 结合,说明指针所指向的内容的类型为int 型.所以P 是一个返回整型数据的指针

`int p[3];` //首先从p 处开始,先与[]结合,说明p 是一个数组,然后与int 结合,说明数组里的元素是整型的,所以P 是一个由整型数据组成的数组

`int *p[3];` //首先从p 处开始,先与[]结合,因为其优先级比\*高,所以p 是一个数组,然后再与\*结合,说明数组里的元素是指针类型,然后再与int结合,说明指针所指向的内容的类型是整型的,所以p是一个由指向整型数据的指针所组成的数组.

`int (*p)[3];` //首先从p 处开始,先与\*结合,说明p是一个指针然后再与[]结合(与"()"  
这步可以忽略,只是为了改变优先级),说明指针所指向的内容是一个数组,然后再  
与int 结合,说明数组里的元素是整型的.所以p是一个指向由整型数据组成的数  
组的指针.





# 常见的指针类型总结

`int **p;` //首先从p开始,先与\*结合,说是p 是一个指针,然后再与\*结合,说明指针所指向的元素是指针,然后再与int 结合,说明该被指向的指针所指向的元素是整型数据.

`int p(int);` //从p处起,先与()结合,说明P 是一个函数,然后进入()里分析,说明该函数有一个整型变量的参数然后再与外面的int 结合,说明函数的返回值是一个整型数据

`int (*p)(int);` //从p 处开始,先与指针结合,说明p 是一个指针,然后与()结合,说明指针指向的是一个函数,然后再与()里的int 结合,说明函数有一个int 型的参数,再与最外层的int 结合,说明函数的返回类型是整型,所以p 是一个指向有一个整型参数且返回类型为整型的函数的指针.

`int (*a[10])(int);`//一个有10个指针的数组,该指针指向一个函数,该函数有一个整型参数并返回一个整型数。



# 常见的指针类型总结

`int (*p(int))[3];` //从p 开始,先与()结合,说明p 是一个函数,然后进入()里面,与int 结合,说明函数有一个整型变量参数,然后再与外面的\*结合,说明函数返回的是一个指针,然后到最外面一层,先与[]结合,说明返回的指针指向的是一个数组,然后再与int 结合,说明指针指向的内容是整型数据.所以p是函数, 参数为一个整型数据且返回指向一个包含3个整型变量的数组的指针。

`int *(*p(int))[3];` //可以先跳过,不看这个类型,过于复杂从P 开始,先与()结合,说明p 是一个函数,然后进入()里面,与int 结合,说明函数有一个整型变量参数,然后再与外面的\*结合,说明函数返回的是一个指针,然后到最外面一层,先与[]结合,说明返回的指针指向的是一个数组,然后再与\*结合,说明数组里的元素是指针,然后再与int 结合,说明指针指向的内容是整型数据.所以p是一个参数为一个整型数据且返回一个指向由3个整型指针变量组成的数组的指针的函数。



# 指针的值和内容

指针的值是指针本身存储的数值，这个值将被编译器当作一个地址，而不是一个一般的数值。在32 位程序里，所有类型的指针的值都是一个32 位整数，因为32 位程序里内存地址全都是32 位长。

指针指向的内容就是从指针的值所代表的那个内存地址开始，长度为sizeof(指针所指向的类型)的一片内存区。

定义指针必须指明指针类型。

指针在内存中的运算依赖指针所指向的数据的类型。



# 指针的运算

## 1) 指针变量加/减 一个整数

例如： $p++$ ， $p--$ ， $p+i$ ， $p-i$ ， $p+-i$ ， $p-=i$ 等。

## 2) 指针变量赋值

将一个变量地址赋给一个指针变量。如：

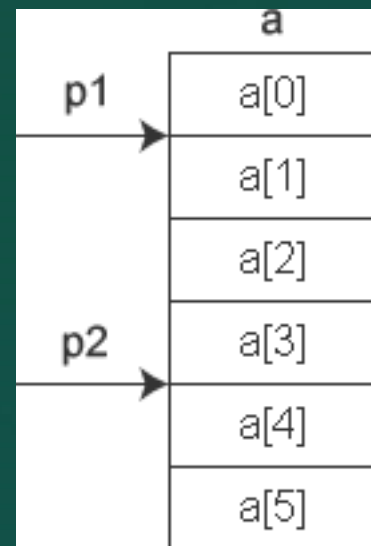
$p = \&a$ ; //将变量a的地址赋给p

$p = \text{array}$ ; //将数组array首元素的地址赋给p

$p = \&\text{array}[i]$ ; //将数组array第i个元素的地址赋给p

$p = \text{max}$ ; //max为已定义的函数，将max的入口地址赋给p

$p1 = p2$ ; //p1和p2都是同类型的指针变量，将p2的值赋给p1



## 3) 两个指针变量可以相减

如果两个指针变量指向同一个数组的元素，则两个指针变量值之差是两个指针之间的元素个数。

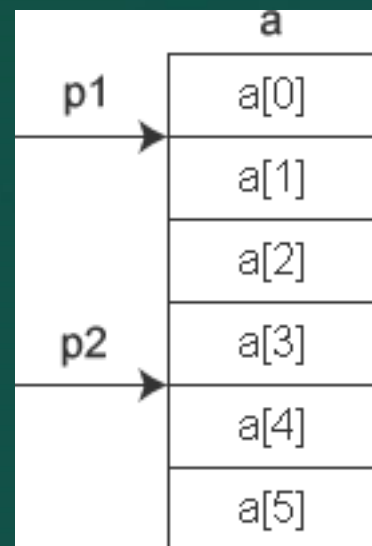
假如p1指向a[1]，p2指向a[4]，则 $p2 - p1 = (a+4) - (a+1) = 4 - 1 = 3$ ，但 $p1 + p2$ 并无实际意义。



# 指针的运算

## 4) 两个指针变量比较

若两个指针指向同一个数组的元素，则可以进行比较。指向前面的元素的指针变量小于指向后面元素的指针变量。如图， $p1 < p2$ ，或者说，表达“ $p1 < p2$ ”的值为真，而“ $p2 < p1$ ”的值为假。注意，如果 $p1$ 和 $p2$ 不指向同一数组则比较无意义。



# 运算符& \*

C++提供了两个关于地址相关的运算符“\*”和“&”,称为指针运算符,“\*”也称解析。“&”称为取地址运算符,用来得到一个对象的地址。

## 声明语句中

“\*”在声明语句中在被声明变量名之前,表示声明的是指针。

“&”出现在变量声明语句中位于被声明变量的左边时,表示声明的是引用。

## 执行语句中

“\*”出现在执行语句中或声明语句的初始化表达式中作为一元运算符时表示指针所指对象的内容。

“&”出现在赋值语句中是出现在等号右边或在执行语句中作为一元运算符出现时表示取地址。

## 作为二元操作符

“\*”表示两个数相乘。

“&”表示逻辑与操作。



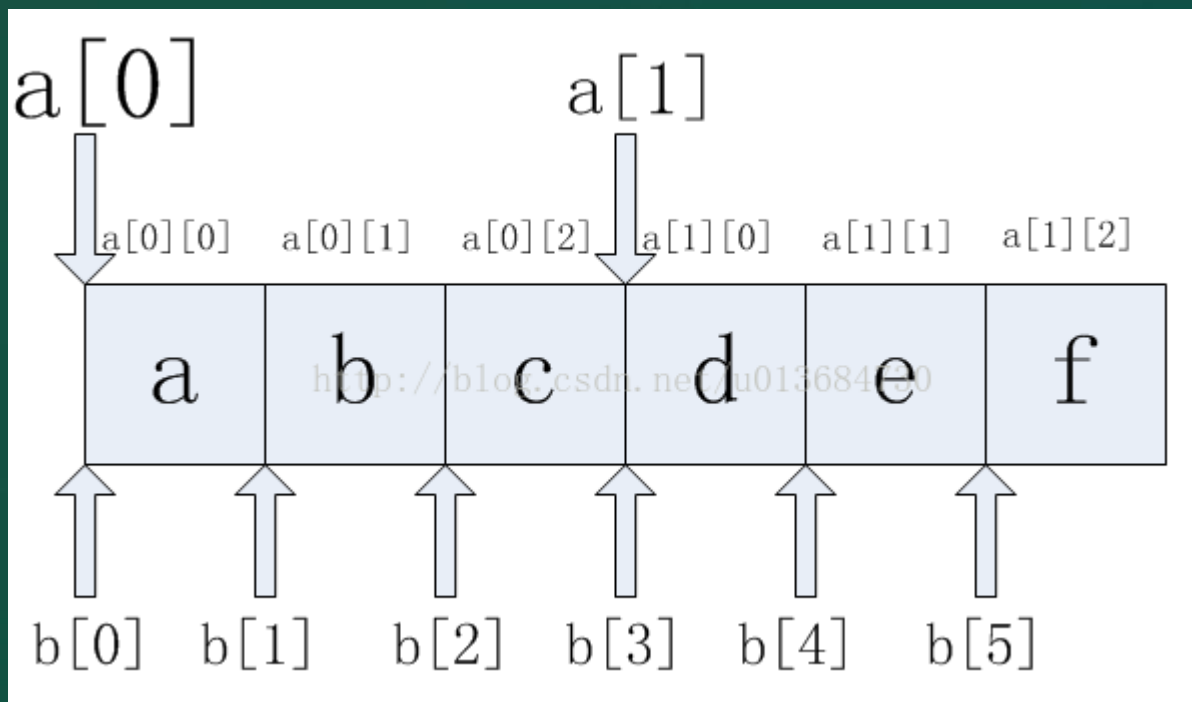
# 指针和数组作为函数参数

实参		匹配的形参	
数组的数组	<code>char c[10][10];</code>	<code>char (*c)[10];</code>	数组指针
指针数组	<code>char *c[10];</code>	<code>char **c;</code>	指针的指针
数组指针	<code>char (*c)[10];</code>	<code>char (*c)[10];</code>	不改变
指针的指针	<code>char **c;</code>	<code>char **c;</code>	不改变



# 二维数组和二维指针

定义了二维数组后，就会在内存中分配一块逻辑上连续的内存块。char c[10][10]，系统就会分配一块100字节的连续内存。也就是说这样的二维数组跟一维数组char c[100]具有相似的内存分布。



这也是为什么 `int a[5][]` 不被允许，而 `int a[][4]` 允许的原因






# 二维数组和二维指针

```
#define ROW 2  
#define COL 3
```

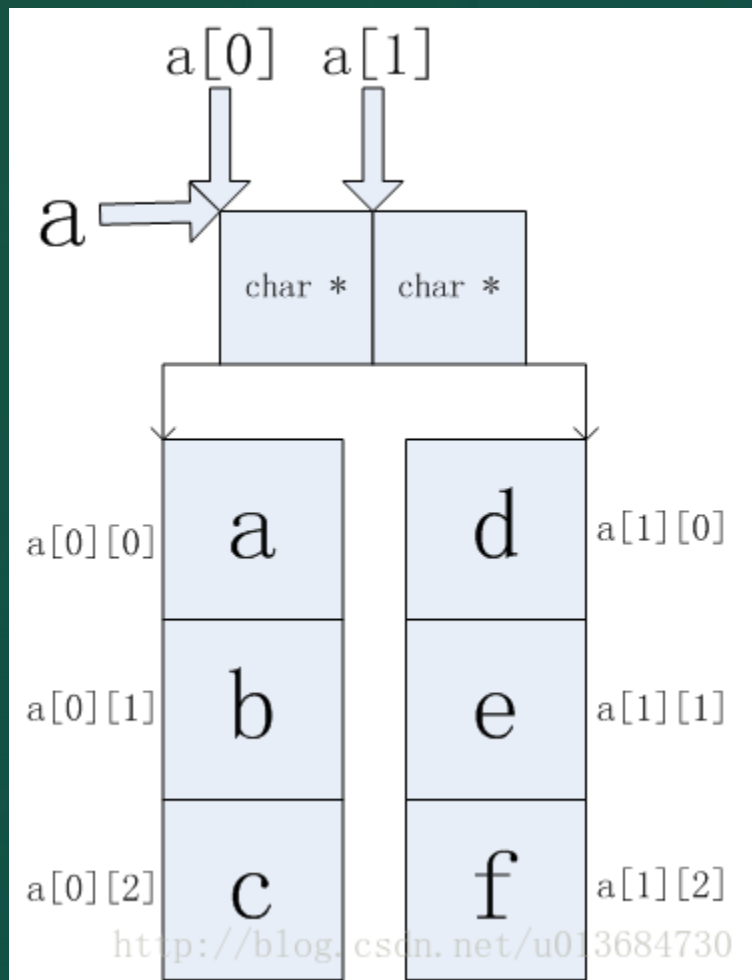
```
union u  
{  
    char a[ROW][COL];  
    char b[ROW*COL];  
}U;
```

```
int main()  
{  
    int i, j;  
    char ch = 'a';  
  
    for (i = 0; i < ROW; i++)  
        for (j = 0; j < COL; j++)  
        {  
            U.a[i][j] = ch;  
            ch++;  
        }  
  
    for (i = 0; i < ROW; i++)  
    {  
        for (j = 0; j < COL; j++)  
            printf("%c - %c\t", U.a[i][j], U.b[i*COL + j]);  
        printf("\n");  
    }  
    return 0;  
}
```



# 二维数组和二维指针

使用二维指针表示二维数组的一般动态分配方式：



# 二维数组和二维指针

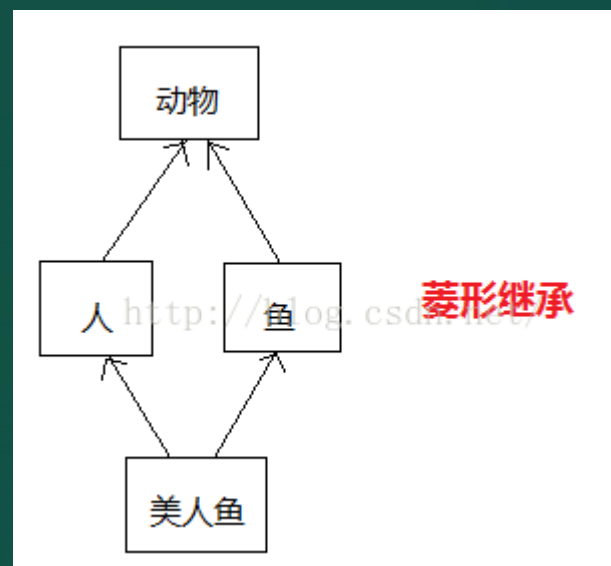
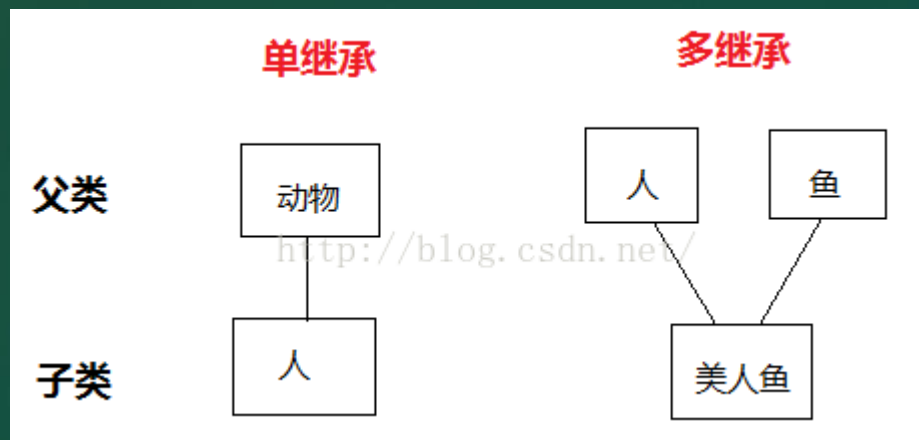
```
char **a;  
int i ,j;  
char ch = 'a';  
a = (char **)malloc(sizeof( char *) * ROW)  
assert(a);  
for (i = 0; i < ROW; i++)  
{  
    a[i] = (char *)malloc(sizeof(char) *COL);  
    assert(a[i]);  
}  
  
for (i = 0; i < ROW; i++)  
    for (j = 0; j < COL; j++)  
    {  
        a[i][j] = ch;  
        ch++;  
    }
```



# 继承和多态

继承关系	父类成员 访问限定符	子类类内 可否访问	子类类外 可否访问	子类成员 访问限定符
public	public	可以	可以	public
	protected	可以	不可以	protected
	private	不可以	不可以	
protected	public	可以	不可以	protected
	protected	可以	不可以	protected
	private	不可以	不可以	
private	public	可以	不可以	private
	protected	可以	不可以	private
	private	不可以	不可以	

# 继承和多态



由单继承和多继承组成的菱形继承有两个问题：

- 1 数据冗余
- 2 访问的二义性



# 虚继承

在继承的时候增加**virtual**关键字，使基类成为**虚基类**。

**解决的问题：**

- 1 数据冗余
- 2 访问的二义性

**带来的好处：**

**时间：**在通过继承类对象访问虚基类对象中的成员（包括数据成员和函数成员）时，都必须通过某种间接引用来完成，这样会增加引用寻址时间（就和虚函数一样），其实就是调整this指针以指向虚基类对象，只不过这个调整是运行时间接完成的。

**空间：**由于共享所以不必要在对象内存中保存多份虚基类子对象的拷贝，这样较之多继承节省空间。虚拟继承与普通继承不同的是，虚拟继承可以防止出现diamond继承时，一个派生类中同时出现了两个基类的子对象。也就是说，为了保证这一点，在虚拟继承情况下，基类子对象的布局是不同于普通继承的。因此，它需要多出一个指向基类子对象的指针。



# 虚函数与多态

虚函数：在类的成员函数之前加virtual关键字。

虚函数重写/覆盖:当派生类与父类的虚函数完全相同时（函数名，参数，返回值都相同,虚析构函数<函数名为类名>除外），子类的这个函数重写/覆盖了父类的函数。

## 多态

静态多态/静态绑定

函数重载：利用函数在编译时重命名标识符来实现。（同一作用域内）

动态多态/动态绑定

使用基类的指针或引用调用 重写虚成员函数。当指针指向基类就调用基类的虚函数，指向派生类就调用派生类的虚函数。（不同作用域内）

动态多态的实现条件：

- 1 满足赋值兼容规则
- 2.子类虚函数重写父类的虚函数（两个类中的虚函数必须完全相同）。
- 3.使用成员函数或者父类的指针/引用来调用父类或子类的虚函数。



谢谢大家