

☺ C++程序设计-绩点与寄点

Written by axi

前言

务必阅读前言！

本人在复习 C++的过程中，发现了自身对于一些知识点的不了解，同时考虑到了 C++程序设计课距离考试已经过去了很久，而笔试往往和实际的应用不是那么紧密相关 (比如一些运算符的规律等，实际使用的时候一般会使用可读性更强的写法)，所以打算自己进行复习，同时顺带将自己复习的内容进行归纳总结，假如说进行分享能帮到别人，不胜荣幸。

里面大多数内容较为基础，但是没有最基础的内容，诸如最为基础的创建变量等等操作，以及有较多的有趣的内容以及进阶内容，有的可能稍微有一些超纲，所以可以酌情阅读，并且可以根据自身的兴趣在这个基础上进行进一步的探索，也可能有错误，务必指出。

一部分的内容有我自身编写的一些小测试，假如觉得在看知识点的时候有一些疑惑，可以自己编写程序并且运行，寻找答案，并且欢迎把自身的疑惑以及对答案告知作者本人，从而丰富测试中的内容。

一部分的内容，比如“运算符”，看上去十分简单，但是其中内容较为有深度，建议不要只看标题就一笔带过，而是认真确定其中的内容自身了解，或者认为没有必要了解，再进行跳过。

数据类型

作为较为基础的内容，不提供更多的内容，只是给出表格，供需要的人查看：

关键字	数据类型	占用字节 (1 字节=8 位)
bool	布尔型	1
char	字符型	1
int	整型	4
float	浮点型	4
double	双浮点型	8

关键字	数据类型	占用字节 (1 字节=8 位)
void	无类型	/
unsigned	作为前缀表示无符号	不变
signed	作为前缀表示有符号	不变
short	作为前缀存储范围减半	$\div 2$
long	作为前缀存储范围翻倍	$\times 2$

运算符

运算符作为 C++一开始的内容，其运算的先后顺序、运算的一些特殊性质以及其和数据类型本身的关系，都很有讲究。

⚙️ 运算顺序

以下表格说明了基础运算符的运算优先级，由上到下为由高到低：

运算符类型	内容
逻辑运算符	!
算数运算符	* / %
算数运算符	+-
关系运算符	< > <= >=
关系运算符	== !=
逻辑运算符	&&
逻辑运算符	

这里面值得注意的是，没有提及的 `()` 运算符，可以用于将一部分的运算作为一个整体，但是其**本身并不会调整运算的优先级**，或者说这种调整在计算机中是没有意义的，比如说一个数学表达式 `a = 10 / (1 + 2)`，对于计算机来说，完全没有必要将 `()` 内部的内容优先处理，而是只需要把其作为整体即可。其优先考虑的就是除法部分，但是之后发现存在一个括号，于是要求括号中返回数值，此时括号内的内容才被计算。

以下提出一个思考，众所周知，`!` 具有最高的优先级，那么其对于运算会不会提高其顺序呢，我们做出了这样的一个程序，以下，大概可以理解：

```

int out()
{
    cout << 1 << endl;
    return 0;
}
int in()
{
    cout << 0 << endl;
    return 0;
}
int main()
{
    int c = (in())+!(out());
    system("pause");
}

```

事实上输出结果将会是先 0 后 1，而非我们希望的先 0 后 1 (也就是因为! 导致 out ()先一步执行)，也就是 **!** 运算符改变计算顺序的结果，也就是说! 并不会使得运算优先进行，甚至说我们所提到的一切的所谓优先级，都不是使其提前进行，而是只是**把其作为一个层次性更高的整体**。

返回

对于正常的运算，一般来说所谓的返回值就是那些函数才会考虑的东西，但是我们在这里提及的返回值，主要为一些内容：

1. `a++` 与 `++a`，其不同点在于，前者的返回值是已经寄存的自加之前的值，后者则是返回其当前的值。
2. 赋值的返回值为其变量的值，如 `cout << (a = 1);` 输出的结果为 1。
3. `cout` 与 `cin` 的返回值均为 1。

舍入

在 C++的运算中，舍入的规则为直接舍弃小数点后面内容，同时当进行整型运算的时候，这一步将会被立刻执行，也就是整型除以整型，立刻为整型。

隐式转换

隐式转换指的是在 C++进行运算的过程中，不用加以声明而进行的转换，这种转换一般会向具有更高自由度的数据类型进行转换。比如说 `int a = 10 / 3 + 10 / 3 + 10 / 3;` 结果是 9，而 `int a = 10.0 / 3 + 10.0 / 3 + 10.0 / 3;` 结果则是 10，这是因为 `10.0/3` 在过程中被转化为了双浮点数，之后三个

双浮点数进行的运算得到了 10，而之前的式子中每一个都直接返回了 3，结果是 9。

同时，隐式转换通常用于进行 char 与 int 数据类型之间的转换，char 转换成 int 将会变成其对应的 ascii 码值，而 int 转换为 char 会变为其数字作为 ascii 码对应的字符。

≡ 三目运算符

三目运算符是一种特殊的运算符，其格式为 `a ? b : c`，等价于：

```
if (a)
    return b;
else
    return c;
```

同时这也意味着假如说 a 为 true，c 将不会被执行。

≡ 与或以及真假

对于 `int` 和 `bool` 的隐式转换来说，`bool` 的 `true` 是 `int` 的 1，`false` 是 0，而对于 `int` 来说，非 0 为 `true`，0 为 `false`。

在与或运算中，对于一系列的 `||` 连接在一起的时候，假如说从左到右有一个 `true`，则之后的全部内容不会被执行，一系列 `&&` 连接在一起的时候，假如说从左到右有一个 `false`，则之后的全部内容不会被执行。这被称为 C++ 的短路求值机制，从而优化程序的执行效率。

≡ 进阶内容

在这里讨论以上提到的问题，当 `!` 修饰的是函数的时候，并不会使得函数优先执行。

在这里提到一个概念，也就是运算式和语句的区别，这是一种我个人的定义。

在上述的表格中全部的内容都是运算式，也就是我们正常认知中的运算，但是对于如函数、赋值以及 `a++` 等内容，则都是语句，语句具备较高的优先级。

在这里给出以下的规则，在笔者目前的测试中一直成立，并且通过汇编语言验证过其准确性：

1. 运算遵循从左至右计算的原则，在同一运算符中，语句具有更高的优先级，但是会因为遇到的运算符比当前的运算优先级更高，而将其视为一个整体，每个整体中都按照从左至右进行计算。因为上文提到的赋值运算的返回值，使得我们得以在运算中对一个变量进行赋值，从而即使在不通过汇编语言的情况下也可以判断运算的先后，例子如下：

```
int a = 0;
int b = (a = 10) + (a = 100) / (a = 1000);
cout << a;
// 对于一些同学来说，对于运算优先级的理解是，先执行(a = 100) / (a = 1000)，之后执行加法，这也符合我们在数学运算中的规律，但是正如我们之前谈到的括号相关的规则一样，事实上这种顺序的调整对于计算机没有任何的意义，在我们正常的运算中也是如此，所以事实上的顺序是，a = 10，然后执行加法，发现右侧是一个除法，拥有更高的优先级，所以把其作为一个整体，从左至右执行里面的内容，之后返回这个整体的值，所以输出a的时候，并不是输出的10，而是1000
```

2. 每个运算符会在两侧的“整体”完成各自的运算之后要求其返回值。还是按照上面的例子举例子，以上的 b，应该是 1001。用这一条规矩进行一下解释，首先从左到右执行，找到加号，执行左右两侧的整体，从左到右，先执行 `a = 10`，然后右侧，发现是一个运算优先级更高的内容，一起运算，其中一个 `a = 100` 一个 `a = 1000`，从左到右执行，此时 a 最后的值是 1000，这时候除法开始生效，要求左右返回值，以上提到赋值的返回值是其变量的当前值，也就是 1000，所以除法实际上是 `1000 / 1000`，此时右侧的运算完成，于是加号要求返回值，左侧为赋值，返回当前值 1000，右侧为除法，返回 1，结果为 1001。

3. 关于变量的运算，其值为当前的变量的值

```
int a = 0;
int b = a + (a = 10) + a + (a = 100);
cout << b;
// 其运算顺序为，先执行a + (a = 10)，因为语句的运算优先级高，所以先执行a = 10，然后左侧返回a的值10，右侧返回a的当前值，也是10，得到20，继续向右，该加号左侧已经完成，右侧是a，返回a的当前值10，加在一起为30，然后再向右，左侧已经完成，右侧为100，一共为130
```

举一个有趣的例子：

```
int a = 10;
int c = (a=3)+(++a);
cout << c;
```

不妨思考一下输出的结果。

事实上你可能会这样思考：首先 `a = 3`，返回 3，然后 `++a`，返回 4，加在一起，值为 7，但是事实上值为 8，我们已经了解了规则，先完成左右运算，再返回值，也就是先 `a=3`，再 `++a`，再返回 a 的当前值以及 a 的当前值，也就是两个 4。

看一下汇编语言的这一部分：

```
mov     DWORD PTR [rbp-4], 3
add     DWORD PTR [rbp-4], 1
mov     eax, DWORD PTR [rbp-4]
add     eax, eax
mov     DWORD PTR [rbp-8], eax
```

简单翻译一下，第一行的意思是将 3 这个值赋给 a (可以理解为 a 名称为 `rbp-4`，而其变量值为 `DWORD PTR [rbp-4]`)，然后 a 自加 1，这里执行的是两个括号里面的内容，然后将 a 的值赋给这个叫做 `eax` 的临时变量，从这一步开始才是加号的内容，加号会要求来自两个语句的返回值，而此时才会触发赋值的返回值的规律，也就是返回被赋值的变量的值，也就是 4，而 `++a` 也返回当前变量的值，也就是认为二者都是这个临时变量的值，于是自加自身，再将自身的值赋值给了 `rbp-8`，也就是 b。

值得一提的是，**！并不会改变运算的顺序，对于语句的操作来说也是如此**，这一点在笔者的 Linux Ubuntu 的 g++ 9.4.0 编译器中得以验证，其他待定。

假如将上文中的 `++a` 改为 `a++`，值会为 7，因为其实事实上可以将 `a++` 会寄存之前的值，同时返回值这个寄存的值，而 `++a` 则直接使用当前的值，这一点很重要，这导致 `a++` 自加这一行为不会像赋值一样，产生上述内容中我们觉得的反常识的内容，也就是其自身赋值之后，返回的是最后运算结束之前 a 的值，而不是要求返回值当时的 a 的值。`a++` 自加这一操作的返回指向的不是值本身，而是在进行自加操作之后寄存的内容。自减也是同理。

同时值得一提的是，在笔者使用的编译器中，连续的 `||` 以及 `&&` 中，赋值以及自加的表达式将会报错。

另外一点是关于连续的等号或者如+=等操作，此类操作的运算顺序为从右到左。

小测试

请写出一下以下的输出结果：

基础：



```
int a = 10, b = 2;
int d = (a = 0) ? (b++) : 10;
cout << " " << d;
```



```
int a = 0;
if (a || cout << (a = 1) << " " || cout << (a = 10) << " ")
    cout << a + 100;
else
    cout << a;
```



```
int a = 10 + 20 / 3 + 20 / 3 + 20 / 3;
cout << a;
```



```
int a = 10 + 20.0 / 3 + 20.0 / 3 + 20.0 / 3;
cout << a;
```



```
int a = 10 * 11 % 2 + 3 / 2;
cout << a;
```



```
int a = 10 * (11 % 2) + 3 / 2;
cout << a;
```

进阶：



```
int a = 10;
int b = 0;
int c = b + (a++) + !(a = 1);
cout << c;
```



```
int a = 10;
int c = a + (a++) + (a = 1);
cout << c;
```

```
int a = 10;
int c = a + (a++) + !(a = 1);
cout << c;
```

```
int a = 10;
int c = a + (a--) + !(a = 1);
cout << c;
```

```
int a = 10;
int c = (a = 10) + !(a = 0);
cout << c;
```

```
int a = 10;
int c = a + (a = 1) + a + (a = 3) + a;
cout << c;
```

```
int a = 10;
int b = 6;
int c = 12;
int d = a+=b*=c-=6;
cout << 46;
```

答案

基础：

1. 2 10 解析：先执行 `a = 0`，然后返回 `false`，之后执行 `10`，d 为 10，同时 `b++` 不执行，b 为 2。
2. 1 101 解析：先执行 `a`，不是 0，之后执行 `cout << (a = 1) << " "`，然后此时返回值为 `true`，不执行后面内容，同时现在的 a 为 1，之后输出 `a+100`，也就是 101。
3. 28 解析：为 `10+6+6+6`，具体原因见隐式转换中的内容。
4. 30 解析：为 `10+(10+10+10)/3`，为 30。
5. 1 解析：首先执行 `10*11`，结果为 110，执行 `110%2`，结果为 0。加上 `3/2`，结果为 1，一共为 1。

6. 11 解析：首先执行 $11\%2$ ，结果为 1，执行 $10*1$ ，结果为 10。加上加上 $3/2$ ，结果为 1，一共为 11。

进阶：

1. 1 解析：先执行 $a=1$ ，之后执行 $a++$ ，同时记录之前的 a 的值，也就是 1，之后执行加法，为 $0 + 1 + !(1)$ ，加在一起为 1。
2. 22 解析：先执行 $a++$ ，寄存其值 10，同时运算结束的时候 a 为 11，存给 a ，之后执行 $a=1$ ，之后执行加法，各自返回值，为 $11+10+1$ ，也就是 22。
3. 21 解析：先执行 $a++$ ，寄存之前的值 10，同时运算后的值 11 给 a ，之后进行加法，为 $11+10+0$ ，也就是 21。
4. 19 解析：先执行 $a--$ ，寄存之前的值 10，同时运算后的值 9 给 a ，之后进行加法，而 $!(a=1)$ 的值为 0，最后为： $0+1+0$ ，也就是 1。
5. 1 解析：先执行 $a=10$ ，然后执行 $a=0$ ，之后返回二者的值，进行加法，因为赋值返回变量的值，所以为 $0 + 1$ ，也就是 1。
6. 9 解析：依次执行，为 $1+1=2$ ， $2+1=3$ ， $3+3=6$ ， $6+3=9$ 。
7. 46 解析：从右向左计算，结果依次为 6、36、46。

逻辑语句

以下声明几个逻辑语句的要点，不介绍基本的写法：

1. if-else if-else 结构中间不能间隔任何的内容。
2. while 先检测条件再执行，do while 先执行再检测条件。
3. for (a; b; c)，先执行 a ，再检测 b ，再执行内部内容，再执行 c ，再检测 b ，再执行内部内容，在检测，以此类推。
4. switch 之中的 case 之间如果没有添加 break，会一直执行下去，case 的存在只是一个标志位，告诉 switch 应该跳转到这一行，但是并没有作为分割的作用，假如说没有对应的 case，则跳过 switch 或者进入 default (假如存在的话，其含义为 switch 中的 else)。

地址、指针与数组

数据的地址以及指针，是 C++ 学习中重要的一部分，在这里仅进行简要的说明，也就是基础内容，过多的进阶内容将不会涉及（笔者精力有限）。

≡ 数据的地址

地址是变量在内存中储存的值所在的地点，有点像是一个人在酒店里居住的房间号一样，了解了这个房间号，也就可以了解房间中的人到底是谁，而在了解的过程中，自然需要“访问”这一步骤，这在后面会提及。

一个数据有一个地址，这个地址中只储存这一个数据，也就是说二者是一一对应的关系，也就是，假如说知道了地址，就可以知道这个变量的内容。

地址的运算

在了解了地址之后，也可以了解一下地址的运算，这种了解或许需要了解下文中的数组，不过在这里就当作你们已经了解了数组，否则可以先看下面的内容，再回来看。

在 C++ 中，为了地址读取的安全性以及程序编写的便捷性，地址的加法中，如 `&a + 1`，其加 1 并不是在 a 的地址的基础上增加了 1 位，而是增加了 `sizeof (a)` 位，这一点很重要，不过与其举例讲解，我更愿意在后面的测试中向你们展示更多的实例，记住，现在的内容已经足够。

≡ 指针以及三种指针

要是说地址是酒店中的门牌号，那么指针就是房间的房卡，指针的声明通过特殊的写法 `int* p = &a;`，其中 `int*` 表示 p 是一个 int 类型的指针，int 类型这一点很重要，因为 C++ 中的数据类型各自占用的内存大小是不一样的，而且各自的解码方式也有所区别，假如说将一个 double 类型的变量的地址作为 int 处理，会出现很彻底的错误，而这个等号的后面则表示，指针 p 指向 a 的地址，其中 `&` 被称为取地址符。也就是说对于一个变量 a，`&a` 代表其地址，而同时，对于一个地址或者一个指针 p（指针根据上文所说的，就是一个储存着对应数据类型的变量的地址的量），`*p` 代表其地址中储存的值。

指针是储存地址的变量。

三种指针

指针常量

指针常量是一种特殊的指针，其特点是其指向的地址不能改变，使用的格式为 `int* const p = &a;`，而因为其指向的地址不能改变，所以其创建的时候必须进行初始化，也就是确定其指向。

常量指针

常量指针是一种特殊的指针，其特点是不能通过这个指针来更改其指向的内容，就像是指向了一个常量一样，也就是说当通过这种指针访问数据的时候，数据处于一种只读的状态。使用的格式为 `const int* p = &a;`，值得一提的是，可以通过不适用该指针的任何方法来修改 `a` 的值，常量指针的“常量”效果仅存在于其自身。

双 const

这一种指针没有特殊的名称，但是其格式是 `const int* const p = &a;`，不难猜到，这种指针即不能修改其指向，也不能通过指针改变其指向的地址中储存的值。

类、结构体的指针

这里介绍一种特殊的指针的用法，假如是类或者结构体的指针 `p`，想要访问其中的成员或者方法 `a`，理论来说应该使用 `*p.a`，但是事实上可以通过 `p->a` 直接访问。

数组

数组是一种按照一定的顺序储存数据的数据结构，通过 C++ 直接创建的数组，其地址是连续的。

语法糖

在学习数组的使用之前，先要了解数组的本质，在这里引入一个概念：语法糖。

语法糖指的是不会对于编程本身有实质性的功能增加，但是可以增加代码的可读性，增加程序员的工作效率的一种语法现象，毫无疑问，数组就是一种典型的语法糖。

使用一维数组举例子，对于一维数组 `a[i]`，其使用 `[]` 进行快捷的书写的过程就是一种语法糖的过程，而其本质上其实是 `*(a+i)`，理解这一点是至关重要的，这也能理解诸如对于数组 `a[]` 来说 `a` 的意义，以及一些创建了数组但是却使用指针进行编程，以及如何向函数中传入一个数组等操作。

多维数组

多维数组是一件很好理解的事情，对于大于一维的每一维度的数组中的元素，其储存的都是比其低一维度的数组的首项的地址。这里用二维数组举例子，当我们创建了二维数组 `a[3][3] = {{2,3},{1,3}}`，这里是一个 2×2 的数组，而对于这个数组，`a` 是二维数组的首地址，这个二维数组中一共含有两项，分别是 `a[0]` 以及 `a[1]`，而 `a[0]` 则又是数组 `{2,3}` 的首地址，也就是说 `*a[0]=2`，同理，`a[1]` 是数组 `{1,3}` 的首地址。

这里值得一提的是，可以通过取地址符找到 `a[0]` 的地址，其地址就是 `a[0][0]` 的地址，而 `a[0]` 之中本身存储的也是 `a[0][0]` 的地址，是不是很有趣，也就是说，某种意义上 `a[0]==&a[0]`。

再值得一提的是，不存在“地址的地址”，也就是说不存在我们使用 `&a[0]`，看上去是求了 `a[0][0]` 的地址的地址，但是这里需要把储存地址的容器的地址，和地址的地址这个概念分开。

引用

引用的本质是一种常量指针，声明为 `int &b = a;`，其意思是将 `b` 的地址指向 `a` 的地址，也就是说 `b` 的值和 `a` 的值时时刻刻是同步的，这一点在后续关于函数的传参中会有所体现。

进阶内容

数组的声明

对于数组的声明，除了比较基础的如 `a[2][2] = {{1, 2}, {3, 4}}`；，以及在声明之后不加以初始化的方法 `a[3][3];`，还有以下的技巧：

1. 在一些编译器中，`a[10] = {};` 的效果是让 `a` 中全部的元素都为 0，同样的，`a[10] = {1};` 的效果是让 `a` 中第一个元素为 1，其他的都为 0。
2. 对于数组来说，假如在其创建的时候就进行初始化，其最高维度的长度不需要写明。

地址对齐

在预测地址的时候，结合了结构体这一内容之后，不难发现一个有趣的现象，比如说如下的结构体：

```
struct a
{
    int k 1 = 1;
    int k 2 = 1;
    double m = 1.0;
};
struct b
{
    int k 1 = 1;
    double m = 1.0;
    int k 2 = 1;
};
int main ()
```

```
{
    cout<<sizeof (a)<<" "<<sizeof (b);
    system ("pause");
}
```

看上去这两个结构体都是两个 int 一个 double，也就是其大小应该是 $4+4+8=16$ ，但是事实上，b 的大小为 24。这就是地址对齐带来的效果，简单来说，其目的是用空间换时间，但是因为这不是主题，所以不多赘述。

其现象就是，基础数据类型只会从其自身的大小的整数倍的地址开始占用地址，在这里面体现的 b 就是，k1 占用了 0-3 位，而 m 是 double 类型，所以从 8 的倍数开始占用，此时 0 已经被占用，而结构体要求其中数据的地址紧邻，所以从 8 开始占用，占用 8-15 位，之后 int 占用 16-19 位，但是因为整体的结构体也需要考虑地址对齐，而其中最大的数据类型是 double，所以自身大小要为 8 的倍数，也就是 24。

≡ 小测试

```
int a[3][3] = {{1, 2, 3}} ;
int b[][] = {{1, 2}, {2, 4}} ;
int c[3][3] = {1, 2, 3, 4, 5, 6};
int d[3][2] = {{1, 2}, {1, 2}} ;
int e[3][2] = {{1}, {3, 4}, {5, 6}} ;
int f[][3] = {{}, {}} ;
// 以上哪个数组会报错
```

```
struct a
{
    bool b 1;
    bool b 2;
    bool b 3;
    bool b 4;
    bool b 5;
    int i 1;
    double d 1;
    double d 2;
};
struct b
{
    bool b 1;
    bool b 2;
    bool b 3;
    bool b 4;
    bool b 5;
    int i 1;
    double d 1;
```

```
double d 2;  
bool b 6;  
};  
// 以上的结构体的大小 (位)分别是多少
```

```
int a[3][3];  
cout << a[0][0] << " ";  
cout << &a[0][0] + 1 << " ";  
cout << a[0] + 1 << " ";  
cout << a[0] + sizeof (a[0][0]) << " ";  
cout << &a[0] + 1 << " ";  
cout << &a[0] + sizeof (a[0][0]);  
//假如说第一行输出为 0 x 000000, 试问剩下的输出为什么
```

答案

1. b 解析：声明多维数组的时候只有第一个维度可以不声明长度。
2. 32 40 解析：a 首先是五个 bool，占了 0-4 位，之后一位 int，因为地址对齐，占用第 8-11 位，之后两个 double，因为地址对齐，从 16 位开始，占用 16 位，一共 32 位。至于 b，在 32 位的基础上，又多了一位，同时考虑 double 的地址对齐，所以多占用 8 位，为 40。
3. 0 x 000004 0 x 000004 0 x 000010 0 x 00000 c 0 x 000030 解析：对于第一个来说，+1 就是加了 `sizeof (a[0][0])` 位，也就是 4；对于第二个，`a[0]` 等价于 `&a[0][0]`，同理；对于第三个来说，`+sizeof (a[0][0])` 就是加 4，实际上就加了 $4 \times 4 = 16$ 次；对于第四个来说，+1 就是加了 `sizeof (a[0])` 也就是一个 int 的长度为三的数组的大小，也就是 12，所以是 16 进制的 c；对于第五个，也就是加了 $4 \times 12 = 48$ ，也就是 3 个 16，16 进制的 30。

数据的生与死

不知道怎么起标题了，所以随便起了一个，这一章节主要关于代码块、形参与实参以及数据的创建与销毁，其中包括一些后面的面向对象的内容。

代码块

在程序中代码块是指用 `{ }` 框起来的一部分的代码，其中新建的变量的生命周期一般来说只在代码块之中，例外会在后面讲解。

在同一代码块之中，不能用同名的变量，但是在不同代码块中，甚至说在代码块的内外，变量可以同名，虽然在实际的程序设计中不推荐这种写法，但是可以作为考题出现。

不同的代码块中变量同名，因为在进入当前代码块中的时候，别的代码块中的同名变量要不然还没有创建，要不然已经被销毁，自然可以。

而对于在代码块内外的同名变量来说，对于该名称对应的实际的变量，采用就近原则，也就是在当前代码块中假如有该新建变量，则认为该名称为该新建变量，否则为代码块外的变量。

形参与实参

对于函数的传参来说，一般来说对于基础的变量传输，看上去是将其中的内容传了进去，但是实际上函数内部收到的是一份外面的内容的拷贝，也就是说实际上假如说通过直接的传参，尝试改变其中的值，并不会有任何的作用。而且从另一种角度上来说，函数中传入的一切东西都是一种拷贝，这也就带来的实参，也就是实际的参数传入，并且想实际对其造成修改效果的操作，只能通过地址的传入来完成，也就是可以传入指针、地址以及引用，都是可以的，如下：

```
// 值传递，不可以修改实参
void func 1 (int a)
{
    cout << a;
}
// 指针/地址传递，可以修改实参
void func 2 (int* a)
{
    cout << *a;
}
// 引用传递，可以修改实参
void func 3 (int &a)
{
    cout << a;
}
int main ()
{
    int a = 0;
    func 1 (a); // 在函数中也就是 int a = a
    func 2 (&a); // 在函数中也就是 int* a = &a
    func 3 (a); // 在函数中也就是 int &a = a
}
```

≡ 内存四区

提到数据的创建与销毁，就不得不提到内存四区的概念，其中包括代码区（存放二进制代码）、全局区（存放全局变量、静态变量以及常量）、栈区（编译器自动分配释放的变量）、堆区（由程序员分配和释放的变量，不主动释放则在程序结束时释放）。

对于上述代码块中的内容被销毁的情况，假如说是通过 new 或者 malloc 创建的变量，则会被储存在堆区，而不会被自动销毁等。

≡ 小测试

给出以下代码的输出：

```
int a = 10;
{
    int a = 0;
    cout << a << " ";
}
cout << a;
{
    a = 100;
    cout << a << " ";
}
cout << a;
```

≡ 答案

1. 0 10 100 100 解析：在第一个代码块中，因为就近原则，认为 `cout` 的 `a` 是当前代码块中的 `a`，也就是 0，同时这个赋值不会改变外面的 `a`，之后输出外面的 `a`，值不变，为 10，之后代码块内部没有创建 `a`，也就是指向的 `a` 是外面的 `a`，赋值为 100 并且输出，为 100，之后外面的 `a` 此时为 100，输出，为 100。

面向对象

≡ 基本思想

面向对象的基本思想也就是封装、继承以及多态，也是面向对象的特性，值得一提的是友元函数的出现，在某种程度上破坏了对对象的封装的特性。

对象实现了一种更加美观的将程序抽象化的方法，通过将程序中不同的部分分割为不同的对象，每个对象有着不同的属性 (变量)以及不同的行为 (方法)，从而实现一种更为美观的抽象。

构造与析构

对于一个类来说，其具备一种自定义的初始化方法，被称为构造函数，以及一种在类实例化的对象被删除的时候执行的函数，析构函数。

对于构造函数来说，主要有三种，无参、有参、拷贝，其不是重点，简单给出实例，以及部分说明：

```
class A
{
private:
    int age;
public:
    A ()
    {
        // 无参构造，不写构造函数，编译器自动提供
        this->age = 0;
    }
    A (int i_age)
    {
        // 有参构造，当写有参构造，编译器不再自动提供无参构造
        this->age = i_age;
    }
    A (const A &a)
    {
        // 拷贝构造，不写的时候编译器自动提供
        this->age = a.age;
    }
    ~A ()
    {
        // 析构函数，销毁对象中的内容，一般由编译器执行，但是创建在堆
        // 区，也就是 new 出来的变量需要进行手动的删除，使用 delete 关键字
    }
};
```

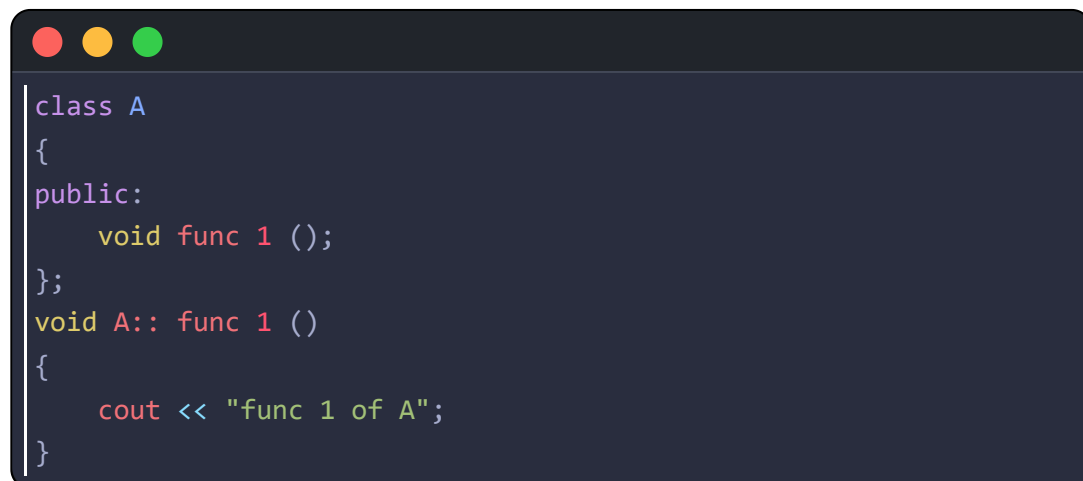
在这里需要考虑一个有趣的问题，也就是深浅拷贝的问题，假如说在 class 中有一个指针变量，其指向的是一个 new 的变量，因此需要手动删除，而这个类实例化了两个对象 a 和 b，b 是通过 a 的拷贝进行的创建，此时 a 与 b 因为都是指针变量，而浅拷贝，也就是编译器提供的构造函数，进行的是简单的赋值操作，也就是二者的指针指向同一内容。这时需要析构 a 与 b，按照特性，a 会被先销毁，之后也就是 a 中的指针指向的地址已经被 delete，而 b 依然会执行析构函数中的这一步骤，但是指针指向的内容已经被删除了，因此

就会报错，因为被删除的东西不能再次被删除，解决这一问题的方法就是重写拷贝构造函数，在拷贝构造函数中复制指针的时候重新 new 一个地址。

≡ 声明与实现

类与对象在本质上不是一种东西，当我们声明一个新的 class 的时候，这只是一个类，但是只有我们 new 出来一个新的对象的时候，这才是对象的实例化，这个被创造出来的个体才被称为对象。

同样的，对于声明和实现，在类中还有另一点得以体现，那就是对于类中方法的声明和实现，可以不放在一起进行，这一点在 C++ 的分文件编写程序过程中使用较多，不多赘述，给一个实例：



```
class A
{
public:
    void func 1 ();
};
void A:: func 1 ()
{
    cout << "func 1 of A";
}
```

≡ 静态

静态使用 `static` 关键字，可以用于修饰变量、方法乃至类，与其说介绍其使用的方法，本人更加倾向于从静态两个字的命名原因入手。

所谓静态，也就是非动态的，动态我们都了解，是在程序运行的过程中进行操作，那么静态则反之，在程序启动甚至在程序编译的时候就已经被创建了，而在程序结束时被销毁，我们从这里入手，就可以找到静态的特性。

既然在程序初始的时候便已经被创建，也就是说静态成员、方法等是唯一的，与后续的实例化无关的，而同时全部的该类都共用其中唯一的静态，而这些静态的成员与方法，因为一开始便存在，称其存在的地方为静态空间。

至于静态类，则用于制作一些工具类，先行存入内存方便调用。

从这里也不难得出静态的不能使用非静态的，非静态的能使用静态的，因为非静态的在静态产生的时候还没有被创建，必然报错，而非静态随时都可以读写在它之前被创建的内容。

对于 `static` 关键字修饰的成员变量，成员方法，成员属性等，其无需 new 一个出来，而是可以直接类名点出来，如 `Classname. func 1 ()`，无需实例化。

一般来说静态成员是常量或者全局量，静态方法为工具方法，静态类为工具类。

同时在这里介绍常量与静态变量的区别：常量，`const` 关键字所修饰的变量，虽然其与静态变量都可以通过类名点出使用，但是其具有以下不同：

1. `const` 必须初始化，不能修改 `static` 没有这个规则
2. `const` 只能修饰变量、`static` 可以修饰很多
3. `const` 一定是写在访问修饰符后面的，`static` 没有这个要求

友元

友元通过 `friend` 关键字，可以在类中定义，被标记为友元的内容可以访问该类中的 `private` 内容，其中主要有三个种类：

```
// 声明 Person，防止之后的创建不成立，顺序很重要
class Person;
// 创建 func 1，其中希望通过 speakAge 访问 Person 中的私有成员 age，
// 本类作为成员函数的示例。
class func 1
{
public:
    void speakAge (Person p);
};
class Person
{
    friend void speakAge (Person p); // 声明全局函数 speakAge 是友元，该函数可以访问 Person 中私有内容
    friend void func 1::speakAge (Person p); // 声明类 func 1，也就是 func 1 作用域下的 speakAge 是友元，该方法可以访问 Person 中私有内容
    friend class func 2; // 声明类 func 2 是友元类，该类中一切内容可以访问 Person 中私有内容
private:
    int age = 18;
};
// 实现方法，不能在上面直接实现，因为当时 Person 只有声明，没有实现，所以不知道里面有 age 这一内容
void func 1::speakAge (Person p)
{
    cout << p.age << endl;
}
// 全局函数
void speakAge (Person p)
{
    cout << p.age << endl;
}
```

```

}
// 作为友元类的类
class func 2
{
public:
    void speakAge (Person p)
    {
        cout << p.age << endl;
    }
};
int main ()
{
    Person p;
    func 1 f 1;
    func 2 f 2;
    speakAge (p);
    f 1.speakAge (p);
    f 2.speakAge (p);
    system ("pause");
}

```

≡ 重载

应当注意到重载与之后提及的重写有很大的区别，重载的特征在于，其若干函数名称相同，返回类型可以不同，而参数列表必须不同，函数将通过传入的参数判断其对应的参数列表，从而锁定对应的函数。

在这里重点介绍运算符重载的格式，其他的请自行了解，反正也不重要。

运算符重载是对于类来说，诸如加号等一系列运算符没有进行过定义，因此需要进行定义，这一步骤被称为运算符重载，以下给出加号运算符重载以及 `<<` 重载作为示例。

```

class Person
{
public:
    int a = 10;
    int b = 20;
    int operator+(Person &p)
    {
        return this->a + p.a;
    }
};

```

以上代码导致加法返回两个 Person 的 a 的值的和。

```

class Person
{
public:
    int a = 10;
    int b = 20;
    Person operator+(Person &p)
    {
        Person temp;
        temp.a = this->a + p.a;
        temp.b = this->b + p.b;
        return temp;
    }
};

```

以上代码导致加法返回一个 Person，其 a 与 b 为原来两个 Person 的 a 与 b 分别的和。

继承

成员属性

对于对象中的变量以及方法，具有其自身的属性，决定了其调用的访问等级，分别为 `public`、`protected` 以及 `private`，分别意味着在类内外都可以访问、只能在类内访问且不继承给子类以及只能在类内访问但是可以继承给子类。值得一提的是，不进行声明，类中的成员属性均为 `private`。

父与子

继承作为一种面向对象的高级用法，其更好的描述了面向对象对于事物抽象描述并且加以定义的流程，其中继承的语法为 `class Son : 继承属性 father`，实现继承操作的类被称为子类或者派生类，而被继承的则被称为父类或者基类。

其中继承属性指 `public`、`protected` 以及 `private`，意味着将父类中继承的比当前级别更松内容放到哪个级别中，也就是说 `public` 会将 `public` 内容放入 `public`，`protected` 内容放入 `protected`，`protected` 会将 `public` 和 `protected` 内容放入 `protected`，而 `private` 会将 `public` 以及 `protected` 内容放入 `private`，给出一个实例：

```

// 定义一个类，人，其必然拥有一些人具有的属性，如下
class Person
{
public:

```

```

    int age;
    int height;
    int weight;
    string name;
};
// 定义一个类，男性，其继承自 Person，也就是说其具备一切人具备的特征，
// 同时还有一些作为男性的特征，比如说自己是一名男性
class man : public Person
{
public:
    void speak ()
    {
        cout << "I'm a man, my age is" << this->age; // this 指针
        // 指向当前的类，使用->符号，后面填写当前类中的成员或者方法，进行调用
    }
};

```

多态

多态是 C++ 乃至大多数面向对象的程序语言都拥有的一个特性，可以用来增加程序的拓展性，更加灵活的编写程序。

简单讲解一下一个最为基本的多态的使用场景：假如说有以下一个类，Animal，其提供一种方法，叫做 speak，会输出“动物 speak”，而 Animal 是 Cat 以及 Dog 两个类的父类，而我们希望 Cat 以及 Dog 类各自实现一种 speak 的方法，分别输出“猫 speak”以及“狗 speak”。假如说有这样一个场景，希望其中输入一个动物，然后调用其 speak 方法，一种较为复杂的方法是依次实现参数列表中为 Cat 以及 Dog 的方法，进行函数的重载，但是还有另一种解决方案，如下：

```

class Animal
{
public:
    void speak ()
    {
        cout << "Animal Speak";
    }
};
class Cat : public Animal
{
public:
    void speak ()
    {
        cout << "Cat Speak";
    }
};
class Dog : public Animal

```

```

{
public:
    void speak ()
    {
        cout << "Dog Speak";
    }
};
void doSpeak (Animal &animal)
{
    animal.speak ();
}
int main ()
{
    Cat c;
    doSpeak (c);
    system ("pause");
}

```

不难看出，这个程序的执行会将 doSpeak 函数中传入的 Cat 类当作 Animal 类并调用其 speak 方法，这样做的底气在于，因为 Cat 是 Animal 的子类，所以 Cat 中必然包含 Animal 的方法，但是这样做，因为其**静态多态函数地址早绑定**的原因，所以只会输出 Animal Speak，但是可以预见的是，假如说我们预想的，因为 Cat 中重新写了相关的 Speak 函数，假如说有一种方法可以调用子类的方法，而不是父类的方法，必然可以解决我们的需求，而且让整体的程序十分的简单。

VOB

VOB 是多态的常见元素，一般来说多态一定会有这三个元素，来达成其多态的效果，而因为这其中的一些硬性的关键字等，主要出现在其他语言，以及 C++ 更加新的标准中，在 C++98 等中或许没有，但是其依然作为一种概念，规范着多态程序的书写。

VOB，也就是虚函数 (virtual)、重写 (override) 以及父类 (base)，是多态实现的三要素。

首先是 virtual 关键字，对于父类中的方法，添加了 virtual 关键字之后，会将其由本来的函数转化为一种函数指针，之后就可以实现，在调用的时候链接到子类之上。

对于子类中的方法，既然要进行多态操作，也就是要进行完全的对于本来方法的覆盖。不同于函数重载中，对于参数列表的不同，重写的要求更为极端，要求一切与原函数完全一致 (对于协变来说并不是如此，但是因为不在考核范围之类，请对其感兴趣的同学自行了解)，对于一些语言，在子类的重写函数之前需要添加 `override` 关键字，而 c++11 的特性中也添加了 `override` 关键字，作为对于程序的规范，不过这都不在考虑范围内，override 这个单词本身并不必须，但是可以提醒我们对于重写这一点严格的遵守。

最后是 base，这一点在诸如 C# 等语言可以调用父类中本来应该被重写掉的函数，但是在 C++ 中这一点并没有实现，所以这里的 B，只是为了提醒我们其代表着当前子类与父类的某种覆盖关系。

虚、纯虚与抽象

在一些项目的架构中，以及一些设计中，诸如上面的 Animal 案例，虽然我们使用了虚函数，对其进行了改进，但是事实上，并不存在一种没有准确名字的动物，可以用到输出的“Animal Speak”，也就是说，在某种程度上，虽然有这一句话没有问题，但是假如程序真正说出了“Animal Speak”，却恰恰意味着程序出了问题，所以对于一些更为“极端”的设计，当然，也是为了保证程序正常运行没有疏漏的常规操作，存在这样一种函数，其本质上没有任何的实现，所以假如不是通过虚函数链接到了别的函数，而是其本身直接执行，就会报错，甚至在编译阶段，编译器就会给出报错，这种函数就叫做**纯虚函数**，而包含了纯虚函数的类被称为**抽象类**，因为其中有一些方法是尚未被实现的，所以**不能被实例化**，而是只是作为一种程序框架中的抽象的概念而存在。

纯虚函数的写法是，不像一般的具有 `virtual` 的函数一样进行实现，而是写如 `virtual void speak () = 0;`，这样就是一个纯虚函数了。

给出一个完善的使用纯虚函数写的上述 Animal 案例供参考：

```
class Animal
{
public:
    virtual void speak () = 0;
};
class Cat : public Animal
{
public:
    void speak ()
    {
        cout << "Cat Speak";
    }
};
class Dog : public Animal
{
};
void doSpeak (Animal &animal)
{
    animal.speak ();
}
int main ()
{
    Cat c;
```



```
// Dog d; 不能被执行, 因为 Dog 没有实现 speak 方法, 所以为抽象类, 不能被实例化
doSpeak (c);
system ("pause");
}
```

其他

其他的可能会考的内容, 包括文件的读写、枚举与共用体、更多的运算符重载、内联函数、字符串、数组的函数传参、函数指针、模板, 有时间会继续写, 不过还请自行了解。