C++编程知识点

标签（空格分隔）： C++

这里总结一些在进行C++编程练习中需要记录的知识点。

1. 指针和引用

- 野指针是指向未分配或者已释放的内存地址
- 使用`free`释放掉一个指针内容后，必须手动设置指针为`NULL`，否则会产生野指针。

1）两个指针之间的运算

只有指向同一个数组的两个指针变量之间才能进行运算，否则运算毫无意义。

(1) 指针变量的相减

两指针变量相减所得之差是两个指针所指数组元素之间相差的元素个数，实际上是两个指针值**(**地址**)**相减之差再除以该数组元素的长度**(**字节数**).**

例如pf1和pf2是指向同一浮点数组的两个指针变量，设pf1的值为2010H，pf2的值为2000H，而浮点数组每个元素占4个字节，所以pf1-pf2的结果为(2000H-2010H)/4=4，表示pf1和 pf2之间相差4个元素。

> 注意：两个指针变量不能进行加法运算。例如，pf1+pf2是什么意思呢？毫无实际意义。

(2) 两指针变量进行关系运算
指向同一数组的两指针变量进行关系运算可表示它们所指数组元素之间的关系。例如：

- pf1 = pf2 表示pf1和pf2指向同一数组元素；
- pf1 > pf2 表示pf1处于高地址位置；
- pf1 < pf2 表示pf2处于低地址位置。

指针变量还可以与0比较。设p为指针变量，则p==0表明p是空指针，它不指向任何变量；p!=0表示p不是空指针。

空指针是由对指针变量赋予0值而得到的。例如：

```
#define NULL 0
int *p = NULL;
```

对指针变量赋0值和不赋值是不同的。指针变量未赋值时，值是随机的，是垃圾值，不能使用的，否则将造成意外错误。而指针变量赋**0**值后，则可以使用，只是它不指向具体的变量而已。

2）常量指针和指针常量

- const在*的左测，指针所指向的内容不可变，即***p**不可变，是常量指针。
- const在*的右侧，指针不可变，即**p++**不被允许，是一个指针常量。

const 限定一个对象为只读属性。
先从一级指针说起吧：
（1）const char p　限定变量p为只读。这样如p=2这样的赋值操作就是错误的。
（2）const char *p　p为一个指向char类型的指针，const只限定p指向的对象为只读。这样，p=&a或 p++等操作

都是合法的，但如*p=4这样的操作就错了，因为企图改写这个已经被限定为只读属性的对象。
（3）char *const p 限定此指针为只读，这样p=&a或 p++等操作都是不合法的。而*p=3这样的操作合法，因为并没有限定其最终对象为只读。
（4）const char *const p 两者皆限定为只读，不能改写。
有了以上的对比，再来看二级指针问题：
（1）const char *p p为一个指向指针的指针，*const*限定其最终对象为只读，显然这最终对象也是为*char*类型的变量。故像*p=3这样的赋值是错误的，而像*p=？ p++这样的操作合法。
（2）const char * const *p 限定最终对象和 p指向的指针为只读。这样 *p=?的操作也是错的。
（3）const char * const * const p 全部限定为只读，都不可以改写。

3）指针数组和数组指针

- 区分int *p[n]; 和int (*p)[n];就要看运算符的优先级了。

  int *p[n]; 中，运算符[]优先级高，先与p结合成为一个数组，再由int*说明这是一个整型指针数组。

  int (*p)[n]; 中()优先级高，首先说明p是一个指针，指向一个整型的一维数组。

  例子：
  int *s[8]; //定义一个指针数组，该数组中每个元素是一个指针，每个指针指向哪里就需要程序中后续再定义了。

  int (*s)[8]; //定义一个数组指针，该指针指向含8个元素的一维数组（数组中每个元素是int型）。
- 对于一个数组，如`int a[10];`，对其数组名进行加减，如果有取地址符和没有是有区别的：

```
// 有取地址符，相当于每次增加整个数组的长度的倍数
&a + i = a + i*sizeof(a);
// 没有使用取地址符，只是数组名，则增加数组中元素的长度的倍数
a + i = a + i*sizeof(a[0]);
```

- 对于二维指针，假设定义一个`int`型的二维指针：

```
int Sec[2][3]={4,6,3,7,2,7};
int **P = Sec;
```

`**p`等价于`Sec[0][0]`，`*p`等价于`Sec[0]`，`*(p+n)`等价于`Sec[n]`，`*((p+n)+m) = Sec[n][m]`。

4）引用

- 引用是变量的别名，在声明的时候就必须初始化
- 引用传递不可以改变原变量的地址，但可以改变原变量的内容

5）引用和指针的区别

本质上，引用是别名，指针是地址，具体来说是：

- 指针可以在运行时改变其所指向的值，引用一旦和某个对象绑定就不再改变；
- 从内存上看，指针会分配内存区域，而引用不会，它仅仅是一个别名；
- 在参数传递时，引用会做类型检查，而指针不会
- 引用不能为空，即必须初始化，而指针可以为空

---

2. 类和函数

- > 假定CSomething是一个类，执行下面这些语句之后，内存里创建了个CSomething对象
  >
  > `CSomething a()`；// 没有创建对象，这里不是使用默认构造函数，而是定义了一个函数，在C++ Primer393页中有说明。
  >
  > `CSomething b(2);` //使用一个参数的构造函数，创建了一个对象。
  >
  > `CSomething c[3];` //使用无参构造函数，创建了**3**个对象。
  >
  > `CSomething &ra=b;` //ra引用b，没有创建新对象。
  >
  > `CSomething d=b;` //使用拷贝构造函数，创建了一个新的对象d。
  >
  > `CSomething *pA = c;` //创建指针，指向对象c，没有构造新对象。
  >
  > `CSomething *p = new CSomething(4);` //新建一个对象。

1） 虚函数

作用：简单讲即实现多态。

> 基类定义了虚函数，子类可以重写该函数，当子类重新定义了父类的虚函数后，父类指针根据赋给它的不同的子类指针，动态地调用属于子类的该函数，且这样的函数调用是无法在编译器期间确认的，而是在运行期确认，也叫做迟绑定。

底层实现原理：**C++**中虚函数使用虚函数表和 虚函数表指针实现，虚函数表是一个类的虚函数的地址表，用于索引类本身以及父类的虚函数的地 址，假如子类的虚函数重写了父类的虚函数，则对应在虚函数表中会把对应的虚函数替换为子类的 虚函数的地址；虚函数表指针存在于每个对象中（通常出于效率考虑，会放在对象的开始地址处）， 它指向对象所在类的虚函数表的地址；在多继承环境下，会存在多个虚函数表指针，分别指向对应不同基类的虚函数表。

- 声明纯虚函数的类是抽象类，不能实例化
- 只包含纯虚函数的抽象类称为接口，接口不能用虚方法和不能包含已经实现的方法，也不能实例化。
- 基类被虚继承才是虚基类
- **virtual** 函数是动态绑定，而缺省参数值却是静态绑定。 意思是你可能会 在**“**调用一个定义于派生类内的**virtual**函数**”**的同时，却使用基类为它所指定的缺省参数值。

  结论：绝不重新定义继承而来的缺省参数值！（可参考《Effective C++》条款**37**）
- 基类的成员函数设为`virtual`，其派生类的相应的函数也会自动变为虚函数。
- 由于类的构造次序是由基类到派生类，所以在构造函数中调用虚函数，这个虚函数不会呈现出多态； 相反，类的析构是从派生类到基类，当调用继承层次中某一层次的类的析构函数时往往意味着其派生类部分已经析构掉，所以也不会呈现出多态
- 静态函数不可以是虚函数

  因为静态成员函数没有this，也就没有存放vptr的地方，同时其函数的指针存放也不同于一般的成员函数，其无法成为一个对象的虚函数的指针以实现由此带来的动态机制。静态是编译时期就必须确定的，虚函数是运行时期确定的。
- 内联函数不能为虚函数；但虚函数可以声明为`inline`，只是编译器会忽略`inline`属性。
- 构造函数，静态成员函数，友元函数都不能是虚函数；只有类的成员函数才能是虚函数；析构函数可以是虚函数，而且通常声明为虚函数，它可以保证释放父类指针时能正确释放子类对象。

2） 组合和继承

“优先使用对象组合，而不是继承**”**是面向对象设计的第二原则。

组合也叫“对象持有”，就是在类中定义另一类型的成员，继承会破坏类的独立性，增加系统的复杂性，一般系统的继承层次不超过3层。组合拥有良好的扩展性，支持动态组合，因此请优先考虑组合方法。

虚继承

虚继承用于解决多继承条件下的菱形继承问题，底层实现原理与编译器相关，一般通过虚基类指针实现，即各对象中只保存一份父类的对象，多继承时通过虚基类指针引用该公共对象，从而避免菱形继承中的二义性问题。

3） 子类型

- 子类型必须是子类继承了父类的所有可继承特性，也即公有继承，才能说是子类型，否则就只是单纯的子类
- 一种类型当它至少提供了另一种类型的行为,则这种类型是另一种类型的子类型
- 子类型关系是不可逆的

4） 内联函数

- 内联函数只适合于只有**1**～**5**行的小函数。对一个含有许多语句的大函数，函数调用和返回的开销 相对来说微不足道，所以也没有必要用内联函数实现。
- 在内联函数内不允许用循环语句和开关语句。否则编译将该函数视同普通函数。
- 如果内联函数定义在调用函数的后面，则编译器会将其当作普通函数调用来看，并不会直接插入到调用处
- 内联函数函数是在编译时直接插入函数代码的

5） 静态变量

- 通常静态数据成员在类声明中声明,在包含类方法的文件中初始化.
- 初始化时使用作用域操作符来指出静态成员所属的类.
- 但如果静态成员是整型或是枚举型**const,**则可以在类声明中初始化
- 静态局部变量存在静态存储区，而局部变量存储在堆栈区，确切的说是栈区
- 静态局部变量有以下特点：
  （1）该变量在全局数据区分配内存；
  （2）静态局部变量在程序执行到该对象的声明处时被首次初始化，即以后的函数调用不再进行初始化；
  （3）静态局部变量一般在声明处初始化，如果没有显式初始化，会被程序自动初始化为0；
  （4）它始终驻留在全局数据区，直到程序运行结束。但其作用域为局部作用域，当定义它的函数或语句块结束时，其作用域随之结束；
- 函数中的静态变量是静态局部变量 ，函数退出后不被释放，在程序运行结束时才释放。只在函数中可访问。
- 静态全局变量的作用域只能是定义它的文件里，不是被其他文件使用。
- 类的静态成员属于整个类，而不是某个对象，可以被类的所有方法访问，子类也可以访问父类的静态成员；
- 所有静态方法只能访问静态成员，不能访问非静态成员，因为静态方法属于整个类，在对象创建之前就已经分配空间，类的非静态成员要在对象创建后才有内存；
- 静态成员可以被任一对象修改，修改后的值可以被所有对象共享。
- Static全局变量和普通全局变量：

  针对：一个工程里有多个cpp文件时

  相同点：存储方式相同，都是静态存储；

  不同点：作用域不同。

普通全局变量---作用域是整个源程序（含有多个源文件），在各个源文件中都有效

Static全局变量----作用域是当前源文件

- **static**关键字的作用：

1. 隐藏：当同时编译多个文件时，所有未加static前缀的全局变量和函数都具有全局可见性。

static可以用作函数和变量的前缀，对于函数来讲，static的作用仅限于隐藏.

1. **static**的第二个作用是保持变量内容的持久：存储在静态数据区的变量会在程序刚开始运行时就完成初始化，也是唯一的一次初始化。 共有两种变量存储在静态存储区：全局变量和**static**变量，只不过和全局变量比起来，static可以控制变量的可见范围，说到底static还是用来隐藏的。虽然这种用法不常见
2. static的第三个作用是默认初始化为**0**（**static**变量）
3. C++中的作用

1）不能将静态成员函数定义为虚函数。

2）静态数据成员是静态存储的，所以必须对它进行初始化。（程序员手动初始化，否则编译时一般不会报错，但是在Link时会报错误）

3）静态数据成员在<定义或说明>时前面加关键字static。

- C++中成员函数不能同时被**static**和**const**同时修饰，因为**static**表示该函数为静态成员函数，为类所有；而**const**是用于修饰成员函数的，两者相矛盾

6） 函数

- 可变参数函数需要由调用者清栈，因为当前函数并不知道要有多少参数被传入，所以必须用**cdcel**
- 函数重载是面向对象程序设计的多态性的实现，就是指同一个函数名对应着不同的函数实现，系统可根据参数的类型、个数来自动完成调用函数的最佳匹配。

  重载函数的参数至少要有一方面不同，表现如下：

  1.函数的参数类型和个数不同；

  2.函数参数的顺序不同。如：`fun(double,int)`和`fun(int,double)`就是两个不同的函数

  重载的函数是在相同的范围，即同一个类中，对于`virtual`关键字是可有可无。不要求返回值类型必须相同。
- 覆盖是指派生类函数覆盖基类函数，特征是：
  1. 不同的范围，分别位于派生类和基类；
  2. 函数名字相同；
  3. 参数相同；
  4. 基类函数必须有`virtual`关键字。
- 隐藏是指派生类的函数屏蔽了同名的基类函数，规则如下：
  1. 如果派生类的函数与基类的函数同名，但是参数不同。此时，不论有无**virtual**关键字，基类的函数将被隐藏（注意别与重载混淆）。
  2. 如果派生类的函数与基类的函数同名，并且参数也相同，但是基类函数没有**virtual** 关键字。此时，基类的函数被隐藏（注意别与覆盖混淆）
- 当函数返回值之后，其函数内部的栈空间均会被销毁；在函数内部，若程序员没有为指针分配空间，则函数退出时，其栈空间也就不存在了；因此，使用数组时，不能返回一个数组；
- 函数不能嵌套定义，也就是函数内部不能定义函数。

- 如果在类的析构函数中调用`delete this`，会发生什么？实验告诉我们，会导致堆栈溢出。原因很简单，`delete`的本质是“为将被释放的内存调用一个或多个析构函数，然后，释放内存”。显然，`delete this`会去调用本对象的析构函数，而析构函数中又调用`delete this`，形成无限递归，造成堆栈溢出，系统崩溃。
- 不建议在构造函数中抛出异常，构造函数抛出异常时，析构函数不会执行，需要手动的去释放内存；
- 构造函数的调用顺序与析构函数的调用顺序相反，这是因为是使用栈实现，先进后出。
- 析构函数不应该抛出异常；当析构函数中会有一些可能发生异常时，那么就必须把这种可能发生的异常完全封装在析构函数内部，决不能让它抛出函数之外。
- 全局对象的构造函数在**main**函数之前调用，析构函数在**main**函数之后调用
- 当派生类中不含对象成员时

  · 在创建派生类对象时,构造函数的执行顺序是：基类的构造函数→派生类的构造函数；

  · 在撤消派生类对象时,析构函数的执行顺序是：派生类的构造函数→基类的构造函数。

  当派生类中含有对象成员时

  · 在定义派生类对象时，构造函数的执行顺序：基类的构造函数→对象成员的构造函数→派生类的构造函数；

  · 在撤消派生类对象时，析构函数的执行顺序：派生类的构造函数→对象成员的构造函数→基类的构造函数。
- 对于只做输入的参数：

  a) 始终用const限制所有指向只输入参数的指针和引用。

  b) 优先通过值来取得原始类型和复制开销比较低的值的对象。

  c) 优先按const的引用取得其他用户定义类型的输入。

  d) 如果函数需要其参数的副本，则可以考虑通过值传递代替通过引用传递
- const对象只能调用const类型成员函数
- 类指针的声明不会调用构造函数，但指向一个类实例会调用构造函数。类的声明也会调用构造函数。
- 成员方法又称为实例方法，静态方法又称为类方法。
- 静态方法中没有`this`指针；
- 类方法可以调用其他类的类方法，也可以调用实例方法。
- 实例方法可以对当前对象的实例变量进行操作，也可以对类变量进行操作，但类方法不能访问实例变量。实例方法必须由实例对象来调用，而类方法除了可由实例对象调用外，还可以由类名直接调用。

  另外，在类方法中不能使用 **this** 或 **super**。 关于类方法的使用，有如下一些限制：

  1 在类方法中不能引用对象变量。

  2 在类方法中不能使用super、this关键字。

  3 类方法不能调用类中的对象方法。

  与类方法相比，实例方法几乎没有什么限制：

  1 实例方法可以引用对象变量（这是显然的），也可以引用类变量。

  2 实例方法中可以使用super、this关键字。

  3 实例方法中可以调用类方法。

7） 继承和访问

（1）基类的私有成员无论什么继承方式，在派生类中均不可以直接访问

（2）在公有继承下，基类的保护成员和公有成员均保持原访问属性

（3）在保护继承方式下，基类的保护和公有成员在派生类的访问属性均为保护属性

（4）在私有继承下，基类的保护和公有成员在派生类中的访问属性均为私有属性

- 赋值运算符重载函数不是不能被派生类继承，而是被派生类的默认“赋值运算符重载函数”给覆盖了。 这就是 **C++** 赋值运算符重载函数不能被派生类继承的真实原因
- 构造函数不能被派生类继承，但可以调用。
- 类成员变量和函数默认情况是私有的，而`struct`是默认公有的。

8）初始化列表

一定需要初始化列表的有三种成员：

- 带有`const`修饰的类成员，如`const int a;`
- 引用成员，如`int &p;`
- 带有引用的类变量，如

```
class A{
  private:
    int &a;
};
class B{
  private:
    A c;
};
// 这里类B的成员c就需要初始化列表进行初始化。
```

此外，对于`static`成员是不允许在类内初始化的，而`static const`成员也不能使用初始化列表。

- 如果父类中没有默认构造方法，那么，在子类中的初始化列表中必须显式的调用基类的有参数构造，否则会编译不通过

9） 多态

- 重载多态和强制多态是 指特定多态。
- 参数多态和包含多态是指通用多态。包含多态是指对基类的**virtual**函数进行重写。

10）运算符

- 只能使用成员函数重载的运算符有：**=**、**()**、**[]**、**->**、**new**、**delete**。
- 单目运算符最好重载为成员函数。
- 对于复合的赋值运算符如**+=**、**-=**、***=**、**/=**、**&=**、**!=**、**~=**、**%=**、**>>=**、**<<=**建议重载为成员函数。
- 对于其它运算符，建议重载为友元函数。
- C语言中，要求运算必须是整型的运算符是**%**。

运算符重载的方法是定义一个重载运算符的函数，在需要执行被重载的运算符时，系统就自动调用该函数，以实现相应的运算。也就是说，运算符重载是通过定义函数实现的。运算符重载实质上是函数的重载。重载运算符的函数一般格式如下：

```
函数类型 operator 运算符名称 (形参表列){
    // 对运算符的重载处理
}
```

- 重载为类成员函数时参数个数=原操作数个数-1（后置++、--除外）
- 重载为友元函数时 参数个数=原操作数个数，且至少应该有一个自定义类型的形参
- 不能重载的运算符有：.（点号），::（域解析符），**?:**（条件语句运算符），**sizeof**（求字节运算符），**typeid**，**static_cast**，**dynamic_cast**，**interpret_cast**（三类类型转换符）
- 逗号运算符：在C语言中，多个表达式可以用逗号分开，其中用逗号分开的表达式的值分别结算，但整个表达式的值是最后一个表达式的值。
- `*`和`++`运算符优先级相同，结合顺序是从左到右，所以`*p++`和`*(p++)`等价，都是先自增指针p，再返回p自增之前所指向的值。

11） 抽象类

抽象类是不完整的，它只能用作基类。在面向对象方法中，抽象类主要用来进行类型隐藏和充当全局变量的角色。

- 抽象类不能实例化。
- 抽象类可以包含抽象方法和抽象访问器。
- 不能用`sealed`修饰符修饰抽象类，因为这两个修饰符的含义是相反的。 采用`sealed`修饰符的类无法继承，而`abstract`修饰符要求对类进行继承。
- 从抽象类派生的非抽象类必须包括继承的所有抽象方法和抽象访问器的实际实现。

12） 友元

- 友元关系不能被继承。
- 友元关系是单向的，不具有交换性。若类B是类A的友元，类A不一定是类B的友元，要看在类中是否有相应的声明。
- 友元关系不具有传递性。若类B是类A的友元，类C是B的友元，类C不一定是类A的友元，同样要看类中是否有相应的申明

---

3. C语言知识点

1） 标识符

C语言中的标识符有：关键字，预定义标识符，用户标识符。

2） **C**结构体之位域**(**位段**)**

参考文章C结构体之位域(位段)

3）柔性数组

柔性数组**(Flexible Array)**也叫伸缩性数组，也就是变长数组，反映了C语言对精炼代码的极致追求。

这种代码结构产生于对动态结构体的需求， 比如我们需要在结构体中存放一个动态长度的字符串时，就可以用柔性数组。

C99使用不完整类型来实现柔性数组，标准形式如下：

```
struct MyStruct{
  int a;
  double b;
  char c[]; // or char c[0]; 也可以用其他数据类型
}
```

上述结构体中的`c`不占用`MyStruct`的空间，只是作为一个符号地址存在，而且必须是结构体的最后一个成员。

参考文章：柔性数组-读《深度探索C++对象模型》有感

---

4. 线程安全

1） 定义

- 线程安全就是多线程访问时，采用了加锁机制，当一个线程访问该类的某个数据时，进行保护，其他线程不能进行访问直到该线程读取完，其他线程才可使用。不会出现数据不一致或者数据污染。
- 线程不安全就是不提供数据访问保护，有可能出现多个线程先后更改数据造成所得到的数据是脏数据

2） 概念

- 如果你的代码所在的进程中有多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的，就是线程安全的。
- 或者说:一个类或者程序所提供的接口对于线程来说是原子操作或者多个线程之间的切换不会导致该接口的执行结果存在二义性,也就是说我们不用考虑同步的问题。
- 线程安全问题都是由全局变量及静态变量引起的。
- 若每个线程中对全局变量、静态变量只有读操作，而无写操作，一般来说，这个全局变量是线程安全的；若有多个线程同时执行写操作，一般都需要考虑线程同步，否则的话就可能影响线程安全。
- 标准库里面的string在多线程下并不保证是都是安全的，只提供两种安全机制：
  1.多个线程同时读取数据是安全的。
  2.只有一个线程在写数据是安全的。
- 局部变量局部使用是安全的。因为每个**thread**都有自己的运行堆栈，而局部变量是生存在堆栈中,大家不干扰。
- 全局原生变量多线程读写是不安全的，全局变量是在堆(heap)中。
- 函数静态变量多线程读写也是不安全的。
- volatile能保证全局整形变量是多线程安全的么？不能。volatile仅仅是告诫compiler不要对这个变量作优化，每次都要从memory取数值，而不是从register
- InterlockedIncrement保证整型变量自增的原子性。
- POSIX线程标准要求C标准库中的大多数函数具备线程安全性
- 写好多线程安全的法宝就是封装，使数据有保护的被访问到
- 安全性：局部变量 > 成员变量 > 全局变量

3） 多线程

- 实现一个多线程(非MFC)程序, 选择多线程CRT, 创建线程的时候应该用

  _beginthreadex()，因为**_beginthreadex()**比较于 **CreateThread()**有更高的线程安全性，不会造成多个线程共用同一个全局变量的情况

4）线程同步

（1） 进程间通信方法有：文件映射、共享内存、匿名管道、命名管道、邮件槽、剪切板、动态数据交换、对象连接与嵌入、动态连接库、远程过程调用等
（2） 事件、临界区、互斥量、信号量可以实现线程同步

---

5. 其他

1）宏定义

C/C++中，宏定义只是做简单的字符替换。

例如：在 `#define add(a,b) a+b` 中，`a+b` 没有括号，所以 `3*add(4,7)` 实际的替换情况是：`3*4+7=19`；若 `a+b` 有括号，`#define add(a,b) （a+b）` 则结果为：`3*（4+7）=33`；

typedef作用是给已存在的数据类型引入一个别名，语法 `typedef 已有类型名 类型别名`。

所以对于以下例子：

```
#define INT_PTR int*
typedef int*int_ptr;
INT_PTR a,b;
int_ptr c,d;
```

上述定义的四个变量中，只有 `b` 不是指针类型，因为宏定义是直接替换，其 `INT_PTR a,b;` 等价于 `int* a, b;`，即 `a` 是整型指针，而 `b` 是整型变量。而使用了 `typedef`,它是类型定义，将 `int*` q取别名为 `` `int_ptr ``,所以 `int_ptr c, d` 中两个变量都是整型指针。

`#define` 和 `const` 相比有如下劣势：
1. `const` 定义常量是有数据类型的，而 `#define` 宏定义常量却没有
2. `const` 常量有数据类型，而宏常量没有数据类型。编译器可以对 `const` 进行类型安全检查，
   而对后者只进行字符替换，没有类型安全检查，并且在字符替换中可能会产生意料不到的错误
3.有些集成化的调试工具可以对 `const` 常量进行调试，但是不能对宏常量进行调试。

两者的区别如下：

本质上，**define**只是字符串替换，**const**参与编译运行，具体为：

- define不会做类型检查，**const**拥有类型，会执行相应的类型检查
- define仅仅是宏替换，不占用内存，而**const**会占用内存
- **const**内存效率更高，编译器通常将const变量保存在符号表中，而不会分配存储空间，这使得它成 为一个编译期间的常量，没有存储和读取的操作

2）**sizeof**

(1) sizeof 返回的值表示的含义如下（单位字节）：

- 数组 —— 编译时分配的数组空间大小；
- 指针 —— 存储该指针所用的空间大小（存储该指针的地址的长度，是长整型，应该为 4 ）；
- 类型 —— 该类型所占的空间大小；
- 对象 —— 对象的实际占用空间大小；
- 函数 —— 函数的返回类型所占的空间大小。函数的返回类型不能是 void 。

(2) **C**语言**: char a = 'a'; sizeof(char) = 1 sizeof(a) = 1 sizeof('a') = 4**
C++语言**: char a = 'a'; sizeof(char) = 1 sizeof(a) = 1 sizeof('a') = 1**
字符型变量是**1**字节这个没错，奇怪就奇怪在**C**语言认为**'a'**是**4**字节，而**C++**语言认为**'a'**是**1**字节。

原因如下:

- C99标准的规定，'a'叫做整型字符常量(integer　character constant)，被看成是int型，所以在32位机器上占4字节。
- ISO C++标准规定，'a'叫做字符字面量(character literal)，被看成是char型，所以占1字节

(3) `sizeof`求出字符串的字符个数,包括结尾符。`strlen`求出字符串的实际字符,不包括结尾符

(4) `sizeof`求数组时，大小等于`数组元素个数*每个元素的大小`（其中，计算字符串数组是需要计算结束符**'\0'**，这是与strlen的区别，strlen不计算最后的'\0'），但是当数组是函数的形参时会将会降为指针，在**32**位系统中无论什么指针类型都是占**4**个字节,而在**64**位系统，指针则是占**8**个字节。

(5) 对空类或者空结构体 ,对其sizeof操作时候,默认都是 **1**个字节

(6) 对类使用，规则如下：

- 类的大小为类的非静态成员数据的类型大小之和，也就是说静态成员数据不作考虑。
- 普通成员函数与sizeof无关。
- 虚函数由于要维护在虚函数表，所以要占据一个指针大小，也就是4字节。
- 类的总大小也遵守类似class字节对齐的，调整规则。

3）变量和表达式

- 左值只能是变量,不能是表达式
- 逻辑与的话第一个条件为假就不会再判断第二个条件，逻辑或第一个条件为真也就不会再判断第二个条件
- 不能对常量进行自增运算，如已经声明`int a =5;`，则不能出现`++(a++)`，因为`a++=5`得到常量`5`。

4）编译器

- 编译器只要能完成程序语言翻译成机器语言即可，所以任意语言均可实现。
- 使用运算符对数据进行格式输出时，必须要包含**iomanip.h**头文件。
- 处理器为大端模式，表示低地址存储高位；小端模式表示低地址存储低位。例如，80*86是小端模式。
- 不同编辑器中变量类型的字节长度：
    - 32位编译器：

      char ：1个字节

      char*（即指针变量）: 4个字节（32位的寻址空间是2^32, 即32个bit，也就是4个字节。同理64位编译器）

      short int : 2个字节

      int：　4个字节

      unsigned int : 4个字节

      float:　4个字节

      double:　8个字节

      long:　4个字节

      long long:　8个字节

unsigned long: 4个字节

- 64位编译器（加粗的是与32位不同的变量类型）：

  char ：1个字节

  char*(即指针变量**): 8**个字节

  short int : 2个字节

  int： 4个字节

  unsigned int : 4个字节

  float: 4个字节

  double: 8个字节

  long: 8个字节

  long long: 8个字节

  unsigned long: 8个字节

- **makefile**文件保存了编译器和连接器的参数选项,还表述了所有源文件之间的关系**(**源代码文件需要的特定的包含文件**,**可执行文件要求包含的目标文件模块及库等**)**.创建程序(make程序)首先读取makefile文件,然后再激活编译器,汇编器,资源编译器和连接器以便产生最后的输出,最后输出并生成的通常是可执行文件.创建程序利用内置的推理规则来激活编译器,以便通过对特定CPP文件的编译来产生特定的OBJ文件.
  Makefile里主要包含了五个东西：显式规则、隐晦规则、变量定义、文件指示和注释。
  1、显式规则。显式规则说明了，如何生成一个或多的的目标文件。这是由Makefile的书写者明显指出，要生成的文件，文件的依赖文件，生成的命令。

  2、隐晦规则。由于我们的make有自动推导的功能，所以隐晦的规则可以让我们比较粗糙地简略地书写Makefile，这是由make所支持的。
  3、变量的定义。在Makefile中我们要定义一系列的变量，变量一般都是字符串，这个有点你C语言中的宏，当Makefile被执行时，其中的变量都会被扩展到相应的引用位置上。
  4、文件指示。其包括了三个部分，一个是在一个Makefile中引用另一个Makefile，就像C语言中的include一样；另一个是指根据某些情况指定Makefile中的有效部分，就像C语言中的预编译#if一样；还有就是定义一个多行的命令。有关这一部分的内容，我会在后续的部分中讲述。
  5、注释。Makefile中只有行注释，和UNIX的Shell脚本一样，其注释是用“#”字符，这个就像C/C++中的“//”一样。如果你要在你的Makefile中使用“#”字符，可以用反斜框进行转义，如：“#”。
  默认的情况下，**make**命令会在当前目录下按顺序找寻文件名
  为**“GNUmakefile”**、**“makefile”**、**“Makefile”**的文件，找到了解释这个文件。在这三个文件名中，最好使用**“Makefile”**这个文件名，因为，这个文件名第一个字符为大写，这样有一种显目的感觉。最好不要用 “GNUmakefile”，这个文件是GNU的make识别的。有另外一些make只对全小写的“makefile”文件名敏感，但是基本上来说，大多数的make都支持“makefile”和“Makefile”这两种默认文件名。
  在**Makefile**使用**include**关键字可以把别的**Makefile**包含进来，这很像**C**语言的**#include**，被包含的文件会原模原样的放在当前文件的包含位置。include的语法是：
  `include <filename>;` filename可以是当前操作系统Shell的文件模式（可以保含路径和通配符）

5）字符串相关函数

- 对于`strcmp()`函数，比较两个字符串，设这两个字符串为str1，str2：
- 若str1==str2，则返回零；
- 若str1>str2，则返回正数；
- 若str1<str2，则返回负数。
- 对于`strcpy()`函数，复制函数，会自动在复制后的字符串上添加**‘\0’**。
- 对于`strcat(char *dest, const char *src)`函数，连接两个字符串；strcat()会将参数src字符串 拷贝到

参数dest所指的字符串尾。 第一个参数dest要有足够的空间来容纳要拷贝的字符串。

- 字符串格式输出：要输出百分号`%`,必须使用`%%`才可以。
- `string`类的函数`c_str()`返回的是`const char*`，是一个指向C字符串的指针，因此对于两个`string`类型的变量`a, b`,如果使用`a.c_str() == b.c_str()`是对两个指针进行比较。

6）其他函数

- `int abs(int num)`

  正常情况下,

  num为**0**或正数时，函数返回num值；

  当**num**为负数且不是最小的负数时（不要问我最小的int类型负数是多少，上面那个图里面有真相），函数返回num的对应绝对值数，即将内存中该二进制位的符号位取反，并把后面数值位取反加一；

  当**num**为最小的负数时（即0x80000000），由于正数里int类型32位表示不了这个数的绝对值，所以依然返回该负数。

- `memcpy`与`memmove`函数

  `memcpy`与`memmove`的目的都是将**N**个字节的源内存地址的内容拷贝到目标内存地址中。

  但当源内存和目标内存存在重叠时，`memcpy`会出现错误，而`memmove`能正确地实施拷贝，但这也增加了一点点开销。

  因此`memmove`函数可以在源地址和目的地址的位置任意的情况下，在源地址和目的地址的空间大小任意的情况下实现二进制代码块的复制

- `void *memset(void *s, int ch, size_t n);`

  函数解释：将s中前n个字节 （typedef unsigned int size_t ）用ch替换并返回s.

  作用是在一段内存块中填充某个给定的值，它是对较大的结构体或数组进行清零操作的一种最快方法,通常为新申请的内存做初始化工作.

- **void *malloc(size_t size);**
  malloc默认返回的是空指针**void***,需要在**malloc** 前面指定类型，比如`char *p1= (char*) malloc(100)`。其次在函数内部`malloc`的内存需要`free`

- **malloc**需要头文件"stdlib.h"或者"malloc.h"，`malloc/free`只是分配内存/回收内存

- **malloc**与**new**的区别：

1. **malloc**与**free**是**C++/C**语言的标准库函数，**new/delete**是**C++**的运算符。它们都可用于申请动态内存和释放内存。
2. 对于非内部数据类型的对象而言，光用**maloc/free**无法满足动态对象的要求。对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。由于malloc/free是库函数而不是运算符，不在编译器控制权限之内，不能够把执行构造函数和析构函数的任务强加于malloc/free。
3. 因此C++语言需要一个能完成动态内存分配和初始化工作的运算符new，以一个能完成清理与释放内存工作的运算符delete。注意new/delete不是库函数。
4. C++程序经常要调用C函数，而**C**程序只能用**malloc/free**管理动态内存。
5. new可以认为是malloc加构造函数的执行。**new**出来的指针是直接带类型信息的。而**malloc**返回的都是**void**指针。

- for循环的条件判断语句中，如果使用赋值语句或常量值，当值为**0**时，不执行循环体，当值为非**0**时，无限循环

7）编程基础

- 正数的原码、补码和反码都是它本身；负数的最高位是1，表示为负数，它的反码按位取反，然后补码就是反码加1.

  举例说明：

  减法7-3相当于加法 7+（-3）
  被加数7的二进制代码为 0000 0111
  加数-3的二进制代码为 1000 0011
  -3的二进制反码为 1111 1100
  -3的二进制补码为 1111 1101（反码加1）
  即 7-3 相当于7+（-3）=0000 0111+1111 1101=0000 0100=4

- 计算机存储的是补码，对于`signed char a = 0xe0;`，即`1110 0000`,其原码应该就是`1010 0000`,也就`-32`了。然后`unsigned int b = a;`，由于`a`是负值，所以转为32位的补码，也就是`11111111 11111111 11111111 11100000`,也就是`0xfffffe0`.

因此，定点二进制运算器中，减法运算一般通过补码运算的二进制加法器实现

- 下面是求二进制数1的个数的代码实现：

```
// 求二进制数1的个数
int numOfOne(int x){
  int count = 0;
  while(x){
    count++;
    x &= x-1;
  }
  return count;
}
```

而求二进制数0的个数代码如下：

```
int numOfZero(int x){
  int count = 0;
  while(x+1){
    count++;
    x |= x+1;
  }
  return count;
}
```

- 空指针是一个特殊的指针值。空指针是指可以确保没有指向任何一个对象的指针。通常使用宏定义**NULL**来表示空指针常量值。**NULL**就代表系统的**0**地址单元。空指针确保它和任何非空指针进行比较都不会相等，因此经常作为函数发生异常时的返回值使用。
- `#include<filename.h>`:表示只从从标准库文件目录下搜索，对于标准库文件搜索效率快。
  `#include"filename.h"`:表示首先从用户工作目录下开始搜索，对于自定义文件搜索比较快，然后搜索整个磁盘。
- 源码 -> （扫描）-> 标记 -> （语法分析）-> 语法树 -> （语义分析）-> 标识语义后的语法树 -> （源码优化）-> 中间代码 -> （代码生成）-> 目标机器代码 -> （目标代码优化）-> 最终目标代码
- 一般高级语言程序编译的过程：预处理、编译、汇编、链接（参考编译原理 (预处理>编译>汇编>链接)（转）)

    - 预处理：C语言程序从源代码变成可执行程序的第一步，主要是C语言编译器对各种预处理命令进行处理，包括头文件的包含、宏定义的扩展、条件编译的选择等。
    - 编译：编译程序工作时，先分析，后综合，从而得到目标程序。所谓分析，是指词法分析和语法分析；所谓综合是指代码优化，存储分配和代码生成。值得一提的是，大多数的编译程序直接产生机器语言的目标代码，形成可执行的目标文件，但也有的编译程序则先产生汇编语言一级的符号代码文件，然后再调用汇编程序进行翻译加工处理，最后产生可执行的机器语言目标文件。
    - 汇编：把作为中间结果的汇编代码翻译成了机器代码，即目标代码，不过它还不可以运行。
    - 链接：处理可重定位文件，把它们的各种符号引用和符号定义转换为可执行文件中的合适信息**(**一般是虚拟内存地址**)**的过程。链接又分为静态链接和动态链接，前者是程序开发阶段程序员用ld(gcc实际上在后台调用了ld)静态链接器手动链接的过程，而动态链接则是程序运行期间系统调用动态链接器(ld-linux.so)自动链接的过程。
- **XML**数据结构只有一个根结点，但是可以嵌套；**XML**解析分为两种：**SAX**和**DOM**解析。
- 数据传输率（**C**）**=**记录位密度（**D**） **x** 线速度**(V)**
- 通常我们开发的程序有2种模式:Debug模式和Release模式
    1. 在**Debug**模式下,编译器会记录很多调试信息,也可以加入很多测试代码,比如加入断言**assert**， 方便我们程序员测试,以及出现bug时的分析解决
    2. **Release**模式下,就没有上述那些调试信息,而且编译器也会自动优化一些代码,这样生成的程序性能是最优的,但是如果出现问题,就不方便分析测试了
    3. **assert** 含义是断言，它是标准C++的cassert头文件中定义的一个宏，用来判断一个条件表达式的值是否为**ture**,如果不为true, 程序会终止，并且报告出错误，这样就很容易将错误定位

8）面向对象的五个基本原则（**Solid**）

- 单一职责原则（**Single-Resposibility Principle**）：一个类，最好只做一件事，只有一个引起它的变化。单一职责原则可以看做是低耦合、高内聚在面向对象原则上的引申，将职责定义为引起变化的原因，以提高内聚性来减少引起变化的原因。
- 开放封闭原则（**Open-Closed principle**）：软件实体应该是可扩展的，而不可修改的。也就是，对扩展开放，对修改封闭的。
- 里氏替换原则（**Liskov-Substituion Principle**）：子类必须能够替换其基类。这一思想体现为对继承机制的约束规范，只有子类能够替换基类时，才能保证系统在运行期内识别子类，这是保证继承复用的基础。
- 接口隔离原则（**Interface-Segregation Principle**）：使用多个小的专门的接口，而不要使用一个大的总接口
- 依赖倒置原则（**Dependecy-Inversion Principle**）：依赖于抽象。具体而言就是高层模块不依赖于底层模块，二者都同依赖于抽象；抽象不依赖于具体，具体依赖于抽象。

9）**case**标签

带标签的语句是一种特殊的语句，在语句前面有一个标识符（即标签，下面代码中的**http**）和一个冒号。使用`goto *label*`就可以跳到标签处执行，比如可以在代码中写`goto http`，这样就会执行`cout`语句了。

```
#include<iostream>
using namespace std;
int main()
{
    http://www.csdn.net
    cout<<"hello world!"<<endl;
    return 0;
}
```

case就是一种标签，**case**关键字和它对应的值一起，称为**case**标签。

类中的**public**、**private**、**protect**也是标签，称为成员访问标签。

`case`标签必须是整型常量表达式，具体条件如下：

（1）C++中的const int，注意仅限于C++中的const，C中的const是只读变量，不是常量；

（2）单个字符，如case 'a': 是合法的，因为文字字符是常量，被转成ASCII码，为整型；

（3）使用#define定义的整型，#define定义的一般为常量，比如#define pi 3.14，但是也必须是整型才可以；

（4）使用enum定义的枚举成员。因为枚举成员是const的，且为整型。如果不手动指定枚举值，则默认枚举值为从0开始，依次加1。

具体参考文章 C++中的case标签 。

10） enum

enum中：首元素不赋值的话，默认为0；后一个元素不赋值的话比前一个元素大1

11） 转义字符

转移字符分三种，一般转义字符，八进制转移字符和十六进制转移字符

- 一般转义字符，如'\b'，由两个字符表示，其实代表一个字符，这个代表退格字符
- *八进制转义字符，如**'\007',**三位数字是八进制的，**ASCII**码为**7**的表示响铃
- 十六进制**，如'\xfe'，同样后面数字是所表示意思的Ascii码的十六进制表示，注意一定要有x，大小写都行

12）STL知识

- **STL**一级容器是容器元素本身是基本类型，非组合类型，因此有**vector, deque, list**。

  而**set, multiset**中元素类型是pair;**map, multimap**中元素类型是pair;
- STL中的常用容器包括：顺序性容器（**vector**、**deque**、**list**）、关联容器（**map**、**set**）、容器适配器（**queue**、**stac**）

一、顺序容器：

vector：可变大小数组；

deque：双端队列；

list：双向链表；

forward_list：单向链表；

array：固定大小数组；

string：与vector相似的容器，但专门用于保存字符。

二、关联容器：

按关键字有序保存元素：（底层实现为红黑树）

map：关联数组；保存关键字-值对；

set：关键字即值，即只保存关键字的容器；

multimap：关键字可重复的map；

multiset：关键字可重复的set；

无序集合：

unordered_map：用哈希函数组织的map；

unordered_set：用哈希函数组织的set；

unordered_multimap：哈希组织的map；关键字可以重复出现；

unordered_multiset：哈希组织的set；关键字可以重复出现。

三、其他项：

stack、**queue**、**valarray**、**bitset**

- **STL**容器是线程不安全的
- **std::sort**封装了快速排序算法，是不稳定算法。
- `vector`在执行函数`erase()`后，会指向下一个元素的位置，也就是执行该函数的时候，后面的元素都会向前移动，而迭代器位置没有移动。
- 访问vector中的数据
  使用两种方法来访问vector。
  1、 vector::at()
  2、 vector::operator[]
  operator[]主要是为了与C语言进行兼容。它可以像C语言数组一样操作。但**at()**是我们的首选，因为**at()**进行了边界检查，如果访问超过了**vector**的范围，将抛出一个例外。
- 支持随机访问的就支持`[]`运算，这包括的容器有`vector, deque, map, unordered_map, string`。
- 当使用一个容器的insert或者erase函数通过迭代器插入或删除元素"可能"会导致迭代器失效

  iterator失效主要有两种情况：

  1、iterator变量已经变成了“野指针”，对它进行*,++,--都会引起程序内存操作异常；

  2、iterator所指向的变量已经不是你所以为的那个变量了。

  不同的容器，他们erase()的返回值的内容是不同的，有的会返回被删除元素的下一个的iterator，有的则会返回删除元素的个数。

  对于非结点类，如数组类的容器 **vector,string,deque** 容器标准写法是这样：

```
//vector<int> m_vector;
 for(vector<int>::iterator iter = m_vector.begin(); iter != m_vector.end();)
{
    if(需要删除)
    {
        iter=m_vector.erase(iter);
    }
    else
        ++iter;
}
```

数组型数据结构：该数据结构的元素是分配在连续的内存中，insert和erase操作，都会使得删除点和插入点之后的元素挪位置，所以，插入点和删除掉之后的迭代器全部失效，也就是说insert(*iter)*(或*erase*(iter))，然后在`iter++`，是没有意义的。解决方法：`erase(*iter)`的返回值是下一个有效迭代器的值。`iter=cont.erase(iter)`;

对于结点类容器**(**如**:list,map,set)**是这样**:**

```
//map<int,int> m_map;
for(map<int,int>::iterator iter = m_map.begin(); iter != m_map.end(); )
{
    if(需要删除)
    {
        m_map.erase(iter++);
    }
    else
        ++iter;
}
```

链表型数据结构：对于list型的数据结构，使用了不连续分配的内存，删除运算使指向删除位置的迭代器失效，但是不会失效其他迭代器.解决办法两种，`erase(*iter)`会返回下一个有效迭代器的值，或者`erase(iter++)`.

树形数据结构： 使用红黑树来存储数据，插入不会使得任何迭代器失效；删除运算使指向删除位置的迭代器失效，但是不会失效其他迭代器.erase迭代器只是被删元素的迭代器失效，但是返回值为void，所以采用`erase(iter++)`。

- **std::vector::iterator**重载了`++, *(前置), ==`运算符。

13）内存分配

- 对于堆来讲，生长方向是向上的，也就是向着内存地址增加的方向；对于栈来讲，它的生长方向是向下的，是向着内存地址减小的方向增长。

- ```
  #pragma pack(2)
  class BU
  {
      int number; // 4
      union UBffer
      {
          char buffer[13]; // 13
          int number; // 4
      }ubuf; // union的大小取决于它所有的成员中，占用空间最大的一个成员的大小，并且需要内存对齐，这里因为#pragma pack(2)，所以union的大小为14，如果不写#pragma pack(2)，那么union大小为16【因为与sizeof（int）=4对齐】
      void foo(){} //0
      typedef char*(*f)(void*); //0
      enum{hdd,ssd,blueray}disk; // 4
  }bu;

  因此sizeof（union） = 4+14 +0 +0 +4 = 22
  ```

- 结构体的内存对齐默认是8个字节。

  1、 结构体的大小等于结构体内最大成员大小的整数倍
  2、 结构体内的成员的首地址相对于结构体首地址的偏移量是其类型大小的整数倍，比如说double型成员相对于结构体的首地址的地址偏移量应该是8的倍数。
  3、 为了满足规则1和2编译器会在结构体成员之后进行字节填充！

- 对Union结构体，sizeof的取值不仅考虑**sizeof**最大的成员，还要考虑对齐字节，对齐字节的取值是取成员类型字节最大值与指定对齐字节(32位机器默认是4，64位机器默认是8)两者中的较小值
- **struct**成员类型不可以是它自己。因为会递归定义。理论上这样导致结构体的大小不能被计算（无限大小）。所以在结构体里的成员类型不能是结构体本身。但是成员可以定义为该结构体的指针。因为指针的大小是已知的（随编译器和操作系统而定）。所以可以定义为该结构体的指针，而不是该结构体本身。
- 涉及到内存分配，都得等到运行阶段。
- 类的大小为类的非静态成员数据的类型大小之和，也就是说静态成员数据不作考虑。

  普通成员函数与**sizeof**无关。

  虚函数由于要维护在虚函数表，所以要占据一个指针大小，**32**位系统中就是**4**字节,且同个类的所有虚函数都只需要一个指针指向虚函数表。

  类的总大小也遵守类似**class**字节对齐的，调整规则。

14） Linux 知识

- **BSS**（**Block Started by Symbol**）通常是指用来存放程序中未初始化的全局变量和静态变量的一块内存区域。特点是:可读写的，在程序执行之前BSS段会自动清0。所以，未初始的全局变量在程序执行之前已经成0了。
- fork()系统调用的特性，
  - fork()系统调用是Unix下以自身进程创建子进程的系统调用，一次调用，两次返回，如果返回是0，则是子进程，如果返回值>0，则是父进程（返回值是子进程的pid），这是众为周知的。
- 还有一个很重要的东西是，在fork()的调用处，整个父进程空间会原模原样地复制到子进程中，包括指令，变量值，程序调用栈，环境变量，缓冲区，等等
  ```

15） 类型转换

1. static_cast

`static_cast < type-id > ( expression )`

该运算符把expression转换为type-id类型，但没有运行时类型检查来保证转换的安全性。它主要有如下几种用法：

①**用**于类层次结构中基类（父类）和派生类（子类）之间指针或引用的转换。

进行上行转换（把派生类的指针或引用转换成基类表示）是安全的；

进行下行转换（把基类指针或引用转换成派生类表示）时，由于没有动态类型检查，所以是不安全的。

②**用**于基本数据类型之间的转换，如把int转换成char，把int转换成enum。这种转换的安全性也要开发人员来保证。

③**把**空指针转换成目标类型的空指针。

④**把**任何类型的表达式转换成**void**类型。

注意：**static_cast**不能转换掉**expression**的**const**、**volatile**、或者**__unaligned**属性。

- C++中的**static_cast**执行非多态的转换，用于代替C中通常的转换操作。因此，被做为显式类型转换使用。

  C++中的**reinterpret_cast**主要是将数据从一种类型的转换为另一种类型。所谓“通常为操作数的位模式提供较低层的重新解释”也就是说将数据以二进制存在形式的重新解释。

2. dynamic_cast<>

`dynamic_cast<>`只用于对象的指针和引用，当用于**C++**类继承多态间的转换，分为：

1.子类向基类的向上转型(Up Cast)
2.基类向子类的向下转型(Down Cast)
其中向上转型不需要借助任何特殊的方法，只需用将子类的指针或引用赋给基类的指针或引用即可，**dynamic_cast**向上转型其总是肯定成功的。

而向下转换时要特别注意：**dynamic_cast**操作符，将基类类型的指针或引用安全的转换为派生类的指针或引用。dynamic_cast将一个基类对象指针（或引用）cast到继承类指针，dynamic_cast会根据基类指针是否真正指向继承类指针来做相应处理。这也是dynamic_cast与其他转换不同的地方，dynamic_cast涉及运行时类别检查，如果绑定到引用或指针的对象不是目标类型的对象，则dynamic_cast失败。如果是指针类型失败，则dynamic_cast的返回结果为0，如果是引用类型的失败，则抛出一个bad_cast错误。
注意：**dynamic_cast**在将父类**cast**到子类时，父类必须要有虚函数。因为**dynamic_cast**运行时需要检查**RTTI**信息。只有带虚函数的类运行时才会检查**RTTI**。

3. reinterpret_cast

转换一个指针为其它类型的指针。它也允许从一个指针转换为整数类型,反之亦 然. 这个操作符能够在非相关的类型之间转换. 操作结果只是简单的从一个指针到别的指针的值的 二进制拷贝. 在类型之间指向的内容不做任何类型的检查和转换

4. const_cast

这个转换类型操纵传递对象的**const**属性，或者是设置或者是移除,例如：

```
class C{};
const C* a = new C;
C *b = const_cast(a);
```

6. 代码片段

1）交换两个变量的数值

```
// 不需要额外设定一个变量的交换方法
int x = 5;
int y = 6;
x += y;
// 将 x 赋值给了 y
y = x-y;
// 将 y 赋值给了 x
x -= y;
```

下面是另一种使用异或交换的方式：

```
x ^= y;
y ^= x;
x ^= y;
```

2） 浮点数判断是否为0

浮点数比较是否等于0的方法如下：

```
float a;
if(fabs(a)< FLT_EPSILON)
    // 判断是等于0
    return true;
```

在ANSIC C中定义了`FLT_EPSILON/DBL_EPSILON/LDBL_EPSILON`来用于浮点数与零的比较.

3) 求余操作用位运算实现

x为整型，请用位运算实现x%8

> 当我们求余的时候，相当于除以2的N次冥，也就是相当于把数本身右移N位，但是右移掉的那些位需要通过位运算进行保留；用以上例子来说，X%8即X%2^3,那么就需要右移三次，那么移去的三位需要保留下来，而8=1000，刚好，可以使用0111来保留下来后三位，于是，对于除数都是2的整数次幂的情况，可以使用**x & (2^n-1)**的方法进行与运算，保留下来的最末尾的n位就是余数。
>
> 因此，上述答案是 **x & 7**

4) 实现一个**memcpy**函数

实现代码如下：

```
// 实现memcpy函数
void *Memcpy(void *dst, const void* src, size_t len){
    if (NULL == dst || NULL == src)
        return NULL;

    void *res = dst;
    if (dst < src || (char*)dst > (char*)src + len){
        // 目标地址在源地址前面，或者目标地址在源地址开始的len范围的后面，可以从低地址开始复制
        while (len--){
            *(char*)dst = *(char*)src;
            dst = (char *)dst + 1;
            src = (char *)src + 1;
        }
    }
    else{
        // 源地址在目标地址前面，只能从高地址开始复制
        src = (char*)src + len - 1;
        dst = (char*)dst + len - 1;
        while (len--){
            *(char*)dst = *(char*)src;
            dst = (char *)dst - 1;
            src = (char *)src - 1;
        }
    }
    return res;
}
```

实现复制函数，首先要判断给定两个指针是否为空，然后要注意地址是否由重叠，分情况进行从低地址复制还是从高地址复制。