1. C++的EOF符号，即文件结束符

通过google得知，在所有的类Unix系统中，Control-D是文件结束符。但是在自己的测试程序中，先输入了一些字符，然后没有输入enter / return的时候，直接就按Control-D发现不好使，需要按两次。

这是因为，Control-D是用在一行的开头的时候才会被认作EOF，当我们输入了一些字符，却没有输入enter/return 时， 第一次按control-D， 前面的字符会被送入程序中处理，这事第一次按control-D的作用，当再次按control-D时，此时，输入流中已经没有东西，第二次按相当于是在一行的开头按了control-D这个时候就顺利结束了

所以如果只想按一次的话，我们可以再输入了一些字符后，先按回车键，然后再按control-D

参考：https://stackoverflow.com/questions/21364313/signal-eof-in-mac-osx-terminal

41down voteaccepted	By default, OS X (formerly Mac OS X) terminals recognize EOF v control-D is pressed at the beginning of a line. In detail, the actual operation is that, when control-D is pressed, a the terminal's input buffer are sent to the running process using th terminal. At the start of a line, no bytes are in the buffer, so the pr told there are zero bytes available, and this acts as an EOF indica This procedure doubles as a method of delivering input to the pro before the end of a line: The user may type some characters and control-D, and the characters will be sent to the process immedia without the usual wait for enter/return to be pressed. After this "se buffered bytes immediately" operation is performed, no bytes are buffer. So, when control-D is pressed a second time, it is the sam beginning of a line (no bytes are sent, and the process is given ze and it acts like an EOF. You can learn more about terminal behavior by using the comma tty" in Terminal. The default line discipline is termios. You can lea about the termios line discipline by using the command "man tern

2. typedef中的陷阱（不可随意替换）

typedef char* test

当我们声明 const test str时，我们不能简单的替换为 const char * str。声明语句中的test 其基本数据类型为char型指针，可是用char*替代之后，数据类型就变成了char，*号变成了声明符的一部分。这样改写的结果是 const char 成了基本数据类型。前后两种的声明完全含义不一样。前者（未替换前）表示的是一个指向char的常量指针，改写后的形式则声明了一个指向const char的指针，且该

指针是可变的
所以正确的替换为 char * const str

3. C++ 中 string.size()函数返回的是一个无符号数。

所以切记在表达式中，当size（）函数的返回值去和一个负数比较时，很大可能会比不过，因为负数会转换为比较大的正数。

补充：所有有size函数的对象，或者模板，比如vector， 其返回的值的类型都是通过size_type定义的，这样的好处在于，在不同的平台上都可以不用修改源码，若是数据格式不支持只需要修改宏定义即可。

4. 使用字符串相加应该注意的问题

由于某些历史原因，以及和C兼容的考虑，C++中字符串字面量，并不是string类型，而string类型相加至少要保证有一个是string类型，否则会出错

5. C++对象用等号赋值，其实是对拷贝了对象，而不是Java中的引用

6. decltype

功能： 返回括号内 表达式或者变量的类型。如果括号内是表达式，则返回表达式的结果对应的类型。其中的几个坑点

6.1 如果表达式是解引用操作，则返回的是引用类型

eg int i=0, *p =i, &r =i;

decltype(*p) c // 错误，c是 int& 必须初始化。因为*p是解引用，即可以赋值，也可以访问该值，所以返回解引用类型

6.2 decltype((var)) 永远得到的是引用类型，因为，变量是作为一种可以作为赋值语句左值的特殊表达式，所以应该回获得引用类型

decltype((i)) d //d 是int& 必须初始化

6.3 decltype 里面若是函数，则返回的是函数类型，要声明函数指针，需要在后面加上*号。

6.4 声明复杂类型，还可以用C++11的新标准后置类型声明 auto f1(int) -> int(*)(int*, int)

7. C++数组

begin() 和 end() 用于返回数组的头指针，和尾指针。相减即为数组的长度。

在遍历多维数组时，使用范围for循环，除了最内层的变量可以不用引用，其他层都需要用，如果不用引用类型，编译器会把遍历的对象当成一个指针，遍历

指针是无法编译通过的。

8. 有负数的取余运算

根据取余运算的定义，如果，m n 是整数且n非0，则 (m/n)*n + m%n = m，隐含的意思是， m%n的运算的结果符号与 m的符号相同。

于是有 m%(-n) = m%n (-m)%n = -(m%n) (-m)%(-n) = (-m)%n = -(m%n)

21%-8 = -5; 21/8=2

21%-5 = 1; 21/-5 = -4

9. 运算对象的求值顺序

大多数运算对象都没有规定求值顺序

eg： *test = *test++；

编译器可能会先求左边test的值，也可能先求右边的值，这样就会导致二义型。所以在编程时应该禁止这种写法。

10. 位操作都是用补码进行操作，因为数字在机器中的表示都是用补码。

如-5 &-4

如果用的是原码： 则 10001001&

10001000 =10001000=-4

但是运算结果是-8。

负数的原码到补码的转换是 从右往左走，看到第一个1后，1之后的取反，符号位不变，第1之前的不变。补码到源码也是一样变。

11. 去除const 属性

```cpp
#include <iostream>
#include <string>
#include <cctype>
#include <vector>
using namespace std;
typedef int intarray[4];
int main() {
    const int a = 4;
    const int *p = &a;
    int *test = const_cast<int *>(p);
    *test = 5;
    cout << a << endl;
    cout << *(&a) << endl;
}
```

```
/Users/husterfox/workspace/C++/cmake-build-debug/C__
4
5

Process finished with exit code 0
```

why a=4?

最可能的是编译优化，把a替换成了4. 但是请不要去尝试const 变量值，可能会导致crash

12. **函数重载和覆写**

重载(overloading)是指，函数名字相同，但是参数类型，或者参数个数不同。光返回值类型不同是非法重载。在重载过程中，有底层const和非底层const修饰，可被看作不同的参数类型。底层表示指向的对象可变性。顶层const的有无并不能被视为不同类型。

重载和 const 形参

如 6.2.3 节（第 190 页）介绍的，顶层 const（参见 2.4.3 节，第 57 页）不影响传入函数的对象。一个拥有顶层 const 的形参无法和另一个没有顶层 const 的形参区分开来：

```
Record lookup(Phone);
Record lookup(const Phone);       // 重复声明了 Record lookup(Phone)

Record lookup(Phone*);
Record lookup(Phone* const);      // 重复声明了 Record lookup(Phone*)
```

在这两组函数声明中，每一组的第二个声明和第一个声明是等价的。

另一方面，如果形参是某种类型的指针或引用，则通过区分其指向的是常量对象还是非常量对象可以实现函数重载，此时的 const 是底层的：

```
// 对于接受引用或指针的函数来说，对象是常量还是非常量对应的形参不同
// 定义了 4 个独立的重载函数
Record lookup(Account&);          // 函数作用于 Account 的引用
Record lookup(const Account&);    // 新函数，作用于常量引用

Record lookup(Account*);          // 新函数，作用于指向 Account 的指针
Record lookup(const Account*);    // 新函数，作用于指向常量的指针
```

覆写(override）发生在父子类之间。子类覆写父类的某一函数，该函数与被覆写函数的参数类型和返回值均一样。覆写目的是为了实现和父类不同的功能

13. **构造函数，利用初始化列表赋值，和在构造函数体内的差别?**

利用初始化列表赋值，编译器实际上使用用户指定的值初始化

而在构造函数体内的赋值，编译器会先使用默认初始化的赋值，然后再进行函数体内的赋值，这样效率上会比前者有损失，另外，对于一些类型，如const 类型和 引用类型，只能用初始化列表赋值。所以C++建议程序员用第一种方式。

14. **类成员的初始化顺序**

类成员的初始化顺序只和类成员定义的顺序有关，而与初始化列表中赋值的顺序无关。

15. **类的隐式转换**

比如，当某类的构造参数是单参数，类型为string时，可以直接使用string类型的字符串给另外一个对象进行拷贝操作，编译器会自动调用构造函数，将字符串转换为相应字符串。

隐式转换是默认行为，若不想发生隐式转换，可用explict声明构造函数。多参数构造函数不存在隐式转换。