**C++关键字extend**

1 **基本解释**：extern可以置于变量或者函数前，以标示变量或者函数的定义在别的文件中，提示编译器遇到此变量和函数时在其他模块中寻找其定义。此外extern也可用来进行链接指定。

      也就是说extern有两个作用，第一个,当它与"C"一起连用时，如: extern "C" void fun(int a, int b);则告诉编译器在编译fun这个函数名时按着C的规则去翻译相应的函数名而不是C++的，C++的规则在翻译这个函数名时会把fun这个名字变得面目全非，可能是fun@aBc\_int\_int#%$也可能是别的，这要看编译器的"脾气"了(不同的编译器采用的方法不一样)，为什么这么做呢，因为C++支持函数的重载啊，在这里不去过多的论述这个问题，如果你有兴趣可以去网上搜索，相信你可以得到满意的解释!  
    第二，当extern不与"C"在一起修饰变量或函数时，如在头文件中: extern int g\_Int; 它的作用就是声明函数或全局变量的作用范围的关键字，其声明的函数和变量可以在本模块活其他模块中使用，记住它是一个声明不是定义!也就是说B模块(编译单元)要是引用模块(编译单元)A中定义的全局变量或函数时，它只要包含A模块的头文件即可,在编译阶段，模块B虽然找不到该函数或变量，但它不会报错，它会在连接时从模块A生成的目标代码中找到此函数。

2 问题：**extern 变量**  
　　在一个源文件里定义了一个数组：char a[6];  
　　在另外一个文件里用下列语句进行了声明：extern char \*a；  
　　请问，这样可以吗？   
　　答案与分析：  
　　1)、不可以，程序运行时会告诉你非法访问。原因在于，指向类型T的指针并不等价于类型T的数组。extern char \*a声明的是一个指针变量而不是字符数组，因此与实际的定义不同，从而造成运行时非法访问。应该将声明改为**extern char a[ ]**。  
　　2)、例子分析如下，如果a[] = "abcd",则外部变量a=0x61626364 (abcd的ASCII码值)，\*a显然没有意义  
　　显然a指向的空间（0x61626364）没有意义，易出现非法内存访问。  
　　3)、这提示我们，在使用extern时候要严格对应声明时的格式，在实际编程中，这样的错误屡见不鲜。  
　　4)、extern用在变量声明中常常有这样一个作用，你在\*.c文件中声明了一个全局的变量，这个全局的变量如果要被引用，就放在\*.h中并用extern来声明。

3 问题：**当方面修改extern 函数原型**  
　　当函数提供方单方面修改函数原型时，如果使用方不知情继续沿用原来的extern申明，这样编译时编译器不会报错。但是在运行过程中，因为少了或者多了输入参数，往往会照成系统错误，这种情况应该如何解决？  
　　答案与分析：  
　　目前业界针对这种情况的处理没有一个很完美的方案，通常的做法是提供方在自己的xxx\_pub.h中提供对外部接口的声明，然后调用方include该头文件，从而省去extern这一步。以避免这种错误。  
　　宝剑有双锋，对extern的应用，不同的场合应该选择不同的做法。

4 问题：**extern “C”**  
　　在C++环境下使用C函数的时候，常常会出现编译器无法找到obj模块中的C函数定义，从而导致链接失败的情况，应该如何解决这种情况呢？

　　答案与分析：  
　　C++语言在编译的时候为了解决函数的多态问题，会将函数名和参数联合起来生成一个中间的函数名称，而C语言则不会，因此会造成链接时找不到对应函数的情况，此时C函数就需要用extern “C”进行链接指定，这告诉编译器，请保持我的名称，不要给我生成用于链接的中间函数名。  
　　下面是一个标准的写法：  
//在.h文件的头上  
#ifdef \_\_cplusplus  
#if \_\_cplusplus  
extern "C"{  
　#endif  
　#endif /\* \_\_cplusplus \*/   
　…  
　…  
　//.h文件结束的地方  
　#ifdef \_\_cplusplus  
　#if \_\_cplusplus  
}  
#endif  
#endif /\* \_\_cplusplus \*/

5 问题：**extern 函数声明**  
　　常常见extern放在函数的前面成为函数声明的一部分，那么，C语言的关键字extern在函数的声明中起什么作用？  
　　答案与分析：  
　　如果函数的声明中带有关键字extern，仅仅是暗示这个函数可能在别的源文件里定义，没有其它作用。即下述两个函数声明没有明显的区别：  
extern int f(); 和int f();  
　　当然，这样的用处还是有的，就是在程序中取代include “\*.h”来声明函数，在一些复杂的项目中，我比较习惯在所有的函数声明前添加extern修饰。关于这样做的原因和利弊可见下面的这个例子：“用extern修饰的全局变量”

    (1) 在test1.h中有下列声明:  
    #ifndef TEST1H  
    #define TEST1H  
    extern char g\_str[]; // 声明全局变量g\_str  
    void fun1();  
    #endif  
    (2) 在test1.cpp中  
    #include "test1.h"  
        char g\_str[] = "123456"; // 定义全局变量g\_str  
        void fun1() { cout << g\_str << endl; }  
    (3) 以上是test1模块， 它的编译和连接都可以通过,如果我们还有test2模块也想使用g\_str,只需要在原文件中引用就可以了  
    #include "test1.h"

     void fun2()    { cout << g\_str << endl;    }  
    以上test1和test2可以同时编译连接通过，如果你感兴趣的话可以用ultraEdit打开test1.obj,你可以在里面找到"123456"这个字符串,但是你却不能在test2.obj里面找到，这是因为g\_str是整个工程的全局变量，在内存中只存在一份,test2.obj这个编译单元不需要再有一份了，不然会在连接时报告重复定义这个错误!  
    (4) 有些人喜欢把全局变量的声明和定义放在一起，这样可以防止忘记了定义，如把上面test1.h改为  
    extern char g\_str[] = "123456"; // 这个时候相当于没有extern  
    然后把test1.cpp中的g\_str的定义去掉,这个时候再编译连接test1和test2两个模块时，会报连接错误，这是因为你把全局变量g\_str的定义放在了头文件之后，test1.cpp这个模块包含了test1.h所以定义了一次g\_str,而test2.cpp也包含了test1.h所以再一次定义了g\_str,这个时候连接器在连接test1和test2时发现两个g\_str。如果你非要把g\_str的定义放在test1.h中的话，那么就把test2的代码中#include "test1.h"去掉 换成:  
    extern char g\_str[];  
    void fun2()   {  cout << g\_str << endl;   }  
   这个时候编译器就知道g\_str是引自于外部的一个编译模块了，不会在本模块中再重复定义一个出来，但是我想说这样做非常糟糕，因为你由于无法在test2.cpp中使用#include "test1.h",那么test1.h中声明的其他函数你也无法使用了，除非也用都用extern修饰，这样的话你光声明的函数就要一大串，而且头文件的作用就是要给外部提供接口使用的，所以 请记住， **只在头文件中做声明，真理总是这么简单**。

6. extern 和 static

 (1) extern 表明该变量在别的地方已经定义过了,在这里要使用那个变量.  
 (2) static 表示静态的变量，分配内存的时候, 存储在静态区,不存储在栈上面.

    static 作用范围是内部连接的关系, 和extern有点相反.它和对象本身是分开存储的,extern也是分开存储的,但是extern可以被其他的对象用extern 引用,而static 不可以,只允许对象本身用它. 具体差别首先，static与extern是一对“水火不容”的家伙，也就是说extern和static不能同时修饰一个变量；其次，static修饰的全局变量声明与定义同时进行，也就是说当你在头文件中使用static声明了全局变量后，它也同时被定义了；最后，static修饰全局变量的作用域只能是本身的编译单元，也就是说它的“全局”只对本编译单元有效，其他编译单元则看不到它,如:  
    (1) test1.h:  
    #ifndef TEST1H  
    #define TEST1H  
    static char g\_str[] = "123456";   
    void fun1();  
    #endif

    (2) test1.cpp:  
    #include "test1.h"  
    void fun1()  {   cout << g\_str << endl;  }  
    (3) test2.cpp  
    #include "test1.h"  
    void fun2()  {   cout << g\_str << endl;  }  
    以上两个编译单元可以连接成功, 当你打开test1.obj时，你可以在它里面找到字符串"123456",同时你也可以在test2.obj中找到它们，它们之所以可以连接成功而没有报重复定义的错误是因为虽然它们有相同的内容，但是存储的物理地址并不一样，就像是两个不同变量赋了相同的值一样，而这两个变量分别作用于它们各自的编译单元。 也许你比较较真，自己偷偷的跟踪调试上面的代码,结果你发现两个编译单元（test1,test2）的g\_str的内存地址相同，于是你下结论static修饰的变量也可以作用于其他模块，但是我要告诉你，那是你的编译器在欺骗你，大多数编译器都对代码都有优化功能，以达到生成的目标程序更节省内存，执行效率更高，当编译器在连接各个编译单元的时候，它会把相同内容的内存只拷贝一份，比如上面的"123456", 位于两个编译单元中的变量都是同样的内容，那么在连接的时候它在内存中就只会存在一份了，如果你把上面的代码改成下面的样子，你马上就可以拆穿编译器的谎言:  
    (1) test1.cpp:  
    #include "test1.h"  
    void fun1()  
    {  
        g\_str[0] = ''a'';  
        cout << g\_str << endl;  
    }

    (2) test2.cpp  
    #include "test1.h"  
    void fun2()  {  cout << g\_str << endl;  }  
    (3) void main()     {  
        fun1(); // a23456  
        fun2(); // 123456  
    }  
    这个时候你在跟踪代码时，就会发现两个编译单元中的g\_str地址并不相同，因为你在一处修改了它，所以编译器被强行的恢复内存的原貌，在内存中存在了两份拷贝给两个模块中的变量使用。正是因为static有以上的特性，所以一般定义static全局变量时，都把它放在原文件中而不是头文件，这样就不会给其他模块造成不必要的信息污染，同样记住这个原则吧！

7. extern 和const

   C++中const修饰的全局常量据有跟static相同的特性，即它们只能作用于本编译模块中，但是const可以与extern连用来声明该常量可以作用于其他编译模块中, 如extern const char g\_str[];  
    然后在原文件中别忘了定义:     const char g\_str[] = "123456";

    所以当const单独使用时它就与static相同，而当与extern一起合作的时候，它的特性就跟extern的一样了！所以对const我没有什么可以过多的描述，我只是想提醒你，const char\* g\_str = "123456" 与 const char g\_str[] ="123465"是不同的， 前面那个const 修饰的是char \*而不是g\_str,它的g\_str并不是常量，它被看做是一个定义了的全局变量（可以被其他编译单元使用）， 所以如果你像让char\*g\_str遵守const的全局常量的规则，最好这么定义const char\* const g\_str="123456".