<http://www.cnblogs.com/edacn/archive/2010/06/08/1753956.html>

原文摘自：http://www.cppblog.com/shifan3/archive/2007/01/05/17325.html  
  
  
详解link  
有些人写C/C++(以下假定为C++)程序，对unresolved external link或者duplicated external simbol的错误信息不知所措（因为这样的错误信息不能定位到某一行）。或者对语言的一些部分不知道为什么要（或者不要）这样那样设计。了解本文之后，或许会有一些答案。  
首先看看我们是如何写一个程序的。如果你在使用某种IDE（Visual Studio，Elicpse，Dev C++等），你可能不会发现程序是如何组织起来的（很多人因此而反对初学者使用IDE）。因为使用IDE，你所做的事情，就是在一个项目里新建一系列的.cpp和.h文件，编写好之后在菜单里点击“编译”，就万事大吉了。但其实以前，程序员写程序不是这样的。他们首先要打开一个编辑器，像编写文本文件一样的写好代码，然后在命令行下敲  
cc 1.cpp -o 1.o  
cc 2.cpp -o 2.o  
cc 3.cpp -o 3.o  
这里cc代表某个C/C++编译器，后面紧跟着要编译的cpp文件，并且以-o指定要输出的文件（请原谅我没有使用任何一个流行编译器作为例子）。这样当前目录下就会出现：  
1.o 2.o 3.o  
最后，程序员还要键入  
link 1.o 2.o 3.o -o a.out  
来生成最终的可执行文件a.out。现在的IDE，其实也同样遵照着这个步骤，只不过把一切都自动化了。  
让我们来分析上面的过程，看看能发现什么。  
首先，对源代码进行编译，是对各个cpp文件单独进行的。对于每一次编译，如果排除在cpp文件里include别的cpp文件的情况（这是C++代码编写中极其错误的写法），那么编译器仅仅知道当前要编译的那一个cpp文件，对其他的cpp文件的存在完全不知情。  
其次，每个cpp文件编译后，产生的.o文件，要被一个链接器(link)所读入，才能最终生成可执行文件。  
好了，有了这些感性认识之后，让我们来看看C/C++程序是如何组织的。  
  
首先要知道一些概念：  
编译：编译器对源代码进行编译，是将以文本形式存在的源代码翻译为机器语言形式的目标文件的过程。  
编译单元：对于C++来说，每一个cpp文件就是一个编译单元。从之前的编译过程的演示可以看出，各个编译单元之间是互相不可知的。  
目标文件：由编译所生成的文件，以机器码的形式包含了编译单元里所有的代码和数据，以及一些其他的信息。  
  
下面我们具体看看编译的过程。我们跳过语法分析等，直接来到目标文件的生成。假设我们有一个1.cpp文件  
int n = 1;  
  
void f()  
{  
++n;  
}  
  
它编译出来的目标文件1.o就会有一个区域（假定名称为2进制段），包含了以上数据／函数，其中有n, f，以文件偏移量的形式给出很可能就是：  
偏移量 内容 长度  
0x000 n 4  
0x004 f ??  
注意：这仅仅是猜测，不代表目标文件的真实布局。目标文件的各个数据不一定连续，也不一定按照这个顺序，当然也不一定从0x000开始。  
现在我们看看从0x004开始f函数的内容（在0x86平台下的猜测）：  
0x004 inc DWORD PTR [0x000]  
0x00? ret  
注意n++已经被翻译为：inc DWORD PTR [0x000]，也就是把本单元0x000位置上的一个DWORD(4字节)加1。  
  
下面如果有另一个2.cpp，如下  
extern int n;  
void g()  
{  
++n;  
}  
那么它的目标文件2.o的2进制段就应该是  
偏移量 内容 长度  
0x000 g ??  
为什么这里没有n的空间（也就是n的定义），因为n被声明为extern，表明n的定义在别的编译单元里。别忘了编译的时候是不可能知道别的编译单元的情况的，故编译器不知道n究竟在何处，所以这个时候g的二进制代码里没有办法填写inc DWORD PTR [???]中的？？？部分。怎么办呢？这个工作就只能交给后来的链接器去处理。为了让链接器知道哪些地方的地址是没有填好的，所以目标文件还要有一个“未解决符号表”，也就是unresolved symbol table. 同样,提供n的定义的目标文件(也就是1.o)也要提供一个“导出符号表”，export symbol table, 来告诉链接器自己可以提供哪些地址。  
让我们理一下思路：现在我们知道，每一个目标文件，除了拥有自己的数据和二进制代码之外，还要至少提供2个表：未解决符号表和导出符号表，分别告诉链接器自己需要什么和能够提供什么。下面的问题是，如何在2个表之间建立对应关系。这里就有一个新的概念：符号。在C/C++中，每一个变量和函数都有自己的符号。例如变量n的符号就是“n”。函数的符号要更加复杂，它需要结合函数名及其参数和调用惯例等，得到一个唯一的字符串。f的符号可能就是"\_f"（根据不同编译器可以有变化）。  
所以，1.o的导出符号表就是  
符号 地址  
n 0x000  
\_f 0x004  
而未解决符号表为空  
2.o的导出符号表为  
符号 地址  
\_g 0x000  
未解决符号表为  
符号 地址   
n 0x001   
这里0x001为从0x000开始的inc DWORD PTR [???]的二进制编码中存储???的起始地址(这里假设inc的机器码的第2－5字节为要+1的绝对地址，需要知道确切情况可查手册)。这个表告诉链接器，在本编译单元0x001的位置上有一个地址，该地址值不明，但是具有符号n。  
链接的时候，链接器在2.o里发现了未解决符号n，那么在查找所有编译单元的时候，在1.o中发现了导出符号n，那么链接器就会将n的地址0x000填写到2.o的0x001的位置上。  
“打住”，可能你就会跳出来指责我了。如果这样做得话，岂不是g的内容就会变成inc DWORD PTR [0x000]，按照之前的理解，这是将本单元的0x000地址的4字节加1，而不是将1.o的对应位置加1。是的，因为每个编译单元的地址都是从0开始的，所以最终拼接起来的时候地址会重复。所以链接器会在拼接的时候对各个单元的地址进行调整。这个例子中，假设2.o的0x00000000地址被定位在可执行文件的0x00001000上，而1.o的0x00000000地址被定位在可执行文件的0x00002000上，那么实际上对链接器来说，1.o的导出符号表其实  
符号 地址  
n 0x000 + 0x2000  
\_f 0x004 + 0x2000  
而未解决符号表为空  
2.o的导出符号表为  
符号 地址  
\_g 0x000 + 0x1000  
未解决符号表为  
符号 地址   
n 0x001 + 0x1000  
所以最终g的代码会变为inc DWORD PTR [0x000 + 0x2000]。  
最后还有一个漏洞，既然最后n的地址变为0x2000了，那么以前f的代码inc DWORD PTR [0x000]就是错误的了。所以目标文件为此还要提供一个表，叫做地址重定向表address redirect table。  
对于1.o来说，它的重定向表为  
地址  
0x005  
这个表不需要符号，当链接器处理这个表的时候，发现地址为0x005的位置上有一个地址需要重定向，那么直接在以0x005开始的4个字节上加上0x2000就可以了。  
让我们总结一下：编译器把一个cpp编译为目标文件的时候，除了要在目标文件里写入cpp里包含的数据和代码，还要至少提供3个表：未解决符号表，导出符号表和地址重定向表。  
未解决符号表提供了所有在该编译单元里引用但是定义并不在本编译单元里的符号及其出现的地址。  
导出符号表提供了本编译单元具有定义，并且愿意提供给其他编译单元使用的符号及其地址。  
地址重定向表提供了本编译单元所有对自身地址的引用的记录。  
链接器进行链接的时候，首先决定各个目标文件在最终可执行文件里的位置。然后访问所有目标文件的地址重定向表，对其中记录的地址进行重定向（即加上该编译单元实际在可执行文件里的起始地址）。然后遍历所有目标文件的未解决符号表，并且在所有的导出符号表里查找匹配的符号，并在未解决符号表中所记录的位置上填写实际的地址（也要加上拥有该符号定义的编译单元实际在可执行文件里的起始地址）。最后把所有的目标文件的内容写在各自的位置上，再作一些别的工作，一个可执行文件就出炉了。  
最终link 1.o 2.o .... 所生成的可执行文件大概是  
0x00000000 ????（别的一些信息）  
....  
0x00001000 inc DWORD PTR [0x00002000] //这里是2.o的开始，也就是g的定义  
0x00001005 ret //假设inc为5个字节，这里是g的结尾  
....  
0x00002000 0x00000001 //这里是1.o的开始，也是n的定义（初始化为1）  
0x00002004 inc DWORD PTR [0x00002000] //这里是f的开始  
0x00002009 ret //假设inc为5个字节，这里是f的结尾  
...  
...  
实际链接的时候更为复杂，因为实际的目标文件里把数据／代码分为好几个区，重定向等要按区进行，但原理是一样的。  
  
  
  
现在我们可以来看看几个经典的链接错误了：  
unresolved external link..  
这个很显然，是链接器发现一个未解决符号，但是在导出符号表里没有找到对应的項。  
解决方案么，当然就是在某个编译单元里提供这个符号的定义就行了。（注意，这个符号可以是一个变量，也可以是一个函数），也可以看看是不是有什么该链接的文件没有链接  
duplicated external simbols...  
这个则是导出符号表里出现了重复项，因此链接器无法确定应该使用哪一个。这可能是使用了重复的名称，也可能有别的原因。  
  
  
我们再来看看C/C++语言里针对这一些而提供的特性：  
extern:这是告诉编译器，这个符号在别的编译单元里定义，也就是要把这个符号放到未解决符号表里去。（外部链接）  
  
static:如果该关键字位于全局函数或者变量的声明的前面，表明该编译单元不导出这个函数／变量的符号。因此无法在别的编译单元里使用。（内部链接）。如果是static局部变量，则该变量的存储方式和全局变量一样，但是仍然不导出符号。  
  
默认链接属性：对于函数和变量，模认外部链接，对于const变量，默认内部链接。（可以通过添加extern和static改变链接属性）  
  
外部链接的利弊：外部链接的符号，可以在整个程序范围内使用（因为导出了符号）。但是同时要求其他的编译单元不能导出相同的符号（不然就是duplicated external simbols)  
  
内部链接的利弊：内部链接的符号，不能在别的编译单元内使用。但是不同的编译单元可以拥有同样名称的内部链接符号。  
  
为什么头文件里一般只可以有声明不能有定义：头文件可以被多个编译单元包含，如果头文件里有定义，那么每个包含这个头文件的编译单元就都会对同一个符号进行定义，如果该符号为外部链接，则会导致duplicated external simbols。因此如果头文件里要定义，必须保证定义的符号只能具有内部链接。  
  
为什么常量默认为内部链接，而变量不是：  
这就是为了能够在头文件里如const int n = 0这样的定义常量。由于常量是只读的，因此即使每个编译单元都拥有一份定义也没有关系。如果一个定义于头文件里的变量拥有内部链接，那么如果出现多个编译单元都定义该变量，则其中一个编译单元对该变量进行修改，不会影响其他单元的同一变量，会产生意想不到的后果。  
  
为什么函数默认是外部链接：  
虽然函数是只读的，但是和变量不同，函数在代码编写的时候非常容易变化，如果函数默认具有内部链接，则人们会倾向于把函数定义在头文件里，那么一旦函数被修改，所有包含了该头文件的编译单元都要被重新编译。另外，函数里定义的静态局部变量也将被定义在头文件里。  
  
为什么类的静态变量不可以就地初始化：所谓就地初始化就是类似于这样的情况：  
class A  
{  
static char msg[] = "aha";  
};  
不允许这样做得原因是，由于class的声明通常是在头文件里，如果允许这样做，其实就相当于在头文件里定义了一个非const变量。  
  
在C++里，头文件定义一个const对象会怎么样：  
一般不会怎么样，这个和C里的在头文件里定义const int一样，每一个包含了这个头文件的编译单元都会定义这个对象。但由于该对象是const的，所以没什么影响。但是：有2种情况可能破坏这个局面：  
1。如果涉及到对这个const对象取地址并且依赖于这个地址的唯一性，那么在不同的编译单元里，取到的地址可以不同。（但一般很少这么做）  
2。如果这个对象具有mutable的变量，某个编译单元对其进行修改，则同样不会影响到别的编译单元。  
  
为什么类的静态常量也不可以就地初始化：  
因为这相当于在头文件里定义了const对象。作为例外，int/char等可以进行就地初始化，是因为这些变量可以直接被优化为立即数，就和宏一样。  
  
内联函数：  
C++里的内联函数由于类似于一个宏，因此不存在链接属性问题。  
  
为什么公共使用的内联函数要定义于头文件里：  
因为编译时编译单元之间互相不知道，如果内联函数被定义于.cpp文件中，编译其他使用该函数的编译单元的时候没有办法找到函数的定义，因此无法对函数进行展开。所以说如果内联函数定义于.cpp文件里，那么就只有这个cpp文件可以是用这个函数。  
  
头文件里内联函数被拒绝会怎样：  
如果定义于头文件里的内联函数被拒绝，那么编译器会自动在每个包含了该头文件的编译单元里定义这个函数并且不导出符号。  
  
如果被拒绝的内联函数里定义了静态局部变量，这个变量会被定义于何处：  
早期的编译器会在每个编译单元里定义一个，并因此产生错误的结果，较新的编译器会解决这个问题，手段未知。  
  
为什么export关键字没人实现：  
export要求编译器跨编译单元查找函数定义，使得编译器实现非常困难。