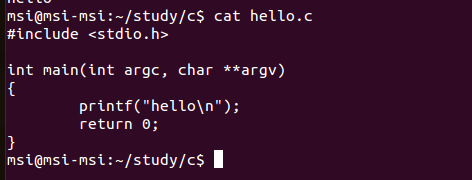
1. main函数启动分析
   1. 概述

从开始学习C语言时，老师和书本都告诉我们C程序是从main函数开始的，刚开始我们一无所知，对此说法奉为圣经，不敢忘记。随着我们的逐渐成长，对程序的理解日益完善，深刻，我们不禁产生了困惑——天下没有无缘无故的爱，也没有无缘无故的关心，那为何我们的C程序偏爱main？难道说，它其实是D&R的私生子名字？

* 1. 分析——编译器偷偷摸摸做的那些事
     1. Linux下的main启动

我们以一个最简单，也是最经典的程序为开始：

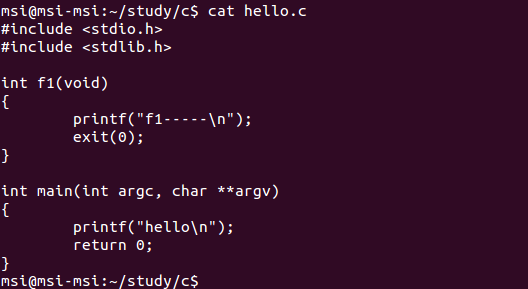


当我们敲下：gcc hello.c时，编译器背着我们干了啥？

编译器至少干了以下几件重要的事情：

1. **指定可执行程序的入口地址（Entry）：**
2. **链接相关库**

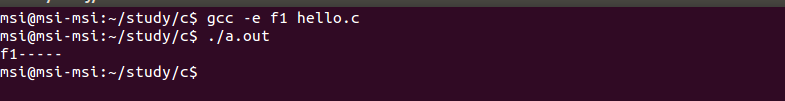
注意到入口地址了吗，它其实才是真正我们的程序被加载到内存中时开始执行的地址，甚至，我们可以手动指定这个地址，我们把代码稍作修改，增加一个f1函数，如下图：



接着，我们在终端再输入以下命令，显式的指定入口地址：

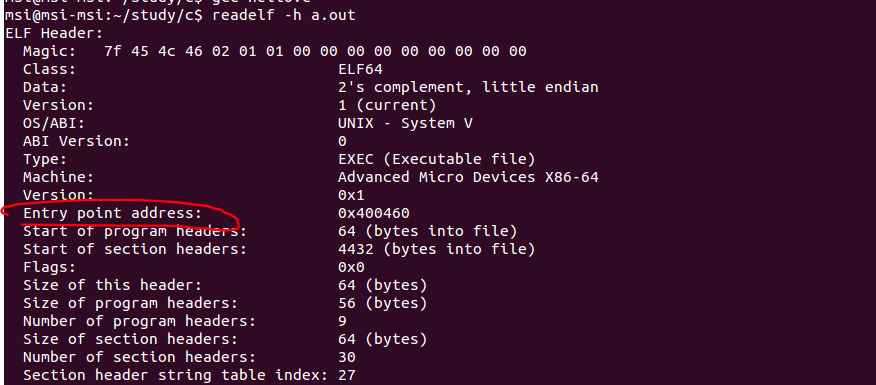
gcc –e f1 hello.c

运行a.out,看截图：

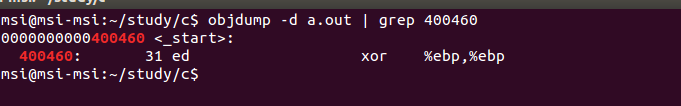


如我们所愿，a.out终于不再从main函数开始了！

现在是不是就大功告成了？编译器的入口地址默认指定为了main函数？没那么简单，别着急，我们继续分析。先用readelf看看a.out(注意，此处没有使用-e选项指定，而是直接编译生成的a.out)的头部信息，找找有什么亮点：

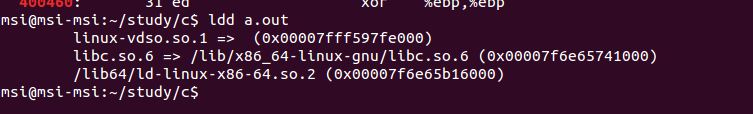


亮点找到了：Entry point address: 0x400460，入口地址，看起来有点像我们的目标，有点小激动啊，我们再看看a.out的代码段，看看这个地址是不是我们的main函数！

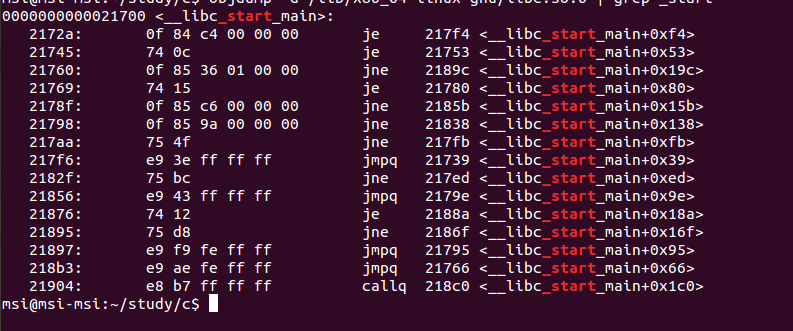


找到了，但似乎与我们的预期不一致，并不是main函数，而是\_start函数，这函数又特么哪里冒出来的？

回过头我们再看看开始说的，编译器偷偷干的事情——链接相关库，我们看看它又背着我们链接了哪些库，如下图：



Linux-vdso.so.1是glibc库Linux下的一种替换方案，ld-linux-x86-64.so.2是link相关的共享库，所以，我们把目标集中在libc.so.6上，即glibc库中，搜索一下里面有没有\_start函数，结果如下：



似乎么有，但是有个\_\_libc\_start\_main的函数，看名字很像启动main的函数啊！赶紧翻开glibc的源码看看：

* + 1. MSVC下的main启动

以Ubuntu12.04x64,gcc4.6.2为例子。

1. 关于硬件stack的理解

栈就是一片memory区域。

LIFO（后进先出）；

SP：栈指针，保存了一个地址，这个地址指向栈顶；

入栈：将数据拷贝到SP指向的地址空间，然后SP+1；

出站：将SP-1，然后将SP指向的空间数据拷贝出来

1. 地址译码的实现

如下图中的4-16线地址译码，实现思想就是2进制和10进制的转换。

寻址原理就是通过二维坐标来定位，换言之，就是二维空间确定一个点的方法。

