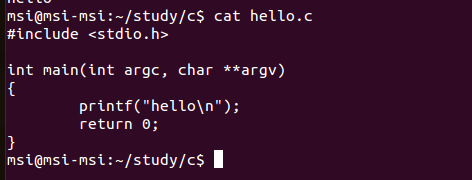
1. main函数启动分析
   1. 概述

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：

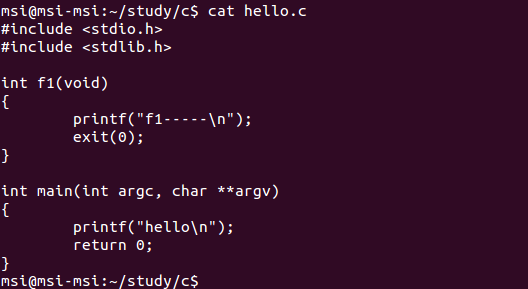


当我们敲下：gcc hello.c时，编译器背着我们干了啥？

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**

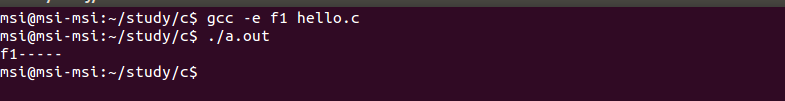
注意到入口地址了吗，它其实才是真正我们的程序被加载到内存中时开始执行的地址，甚至，我们可以手动指定这个地址，我们把代码稍作修改，增加一个f1函数，如下图：



接着，我们在终端再输入以下命令，显式的指定入口地址：

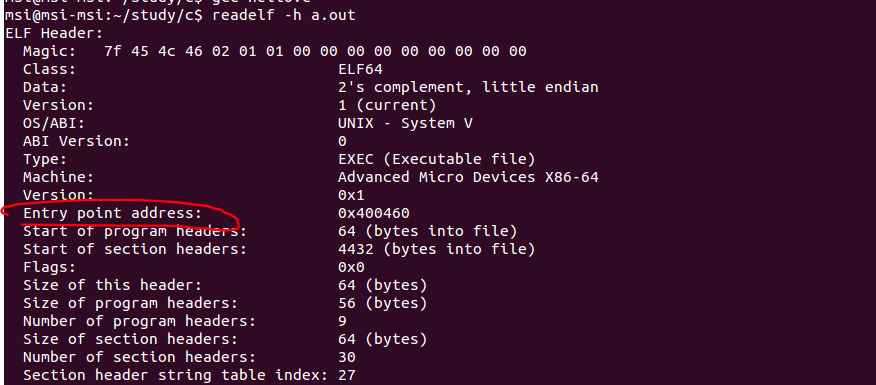
gcc –e f1 hello.c

运行a.out,看截图：

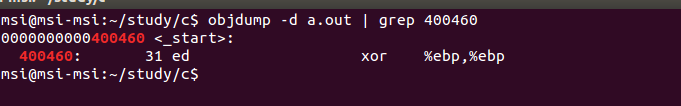


如我们所愿，a.out终于不再从main函数开始了！

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：

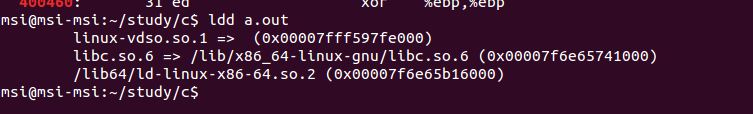


亮点找到了：Entry point address: 0x400460，入口地址，看起来有点像我们的目标，有点小激动啊，我们再看看a.out的代码段，看看这个地址是不是我们的main函数！

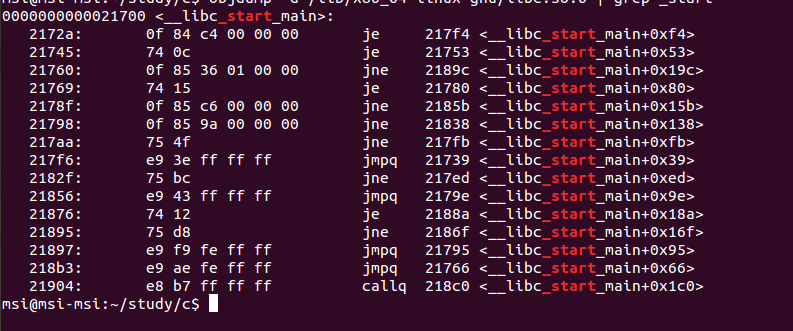


找到了，但似乎与我们的预期不一致，并不是main函数，而是\_start函数，这函数又特么哪里冒出来的？

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。

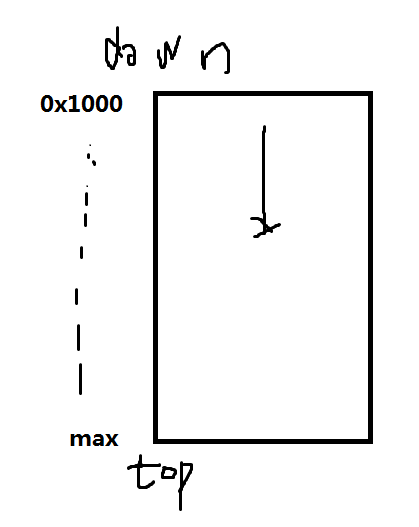
LIFO（后进先出）；

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

栈的top-down由其生长方向决定，与其实际地址高低无关，比如下图，一个以0x1000为起点的向下生长栈：



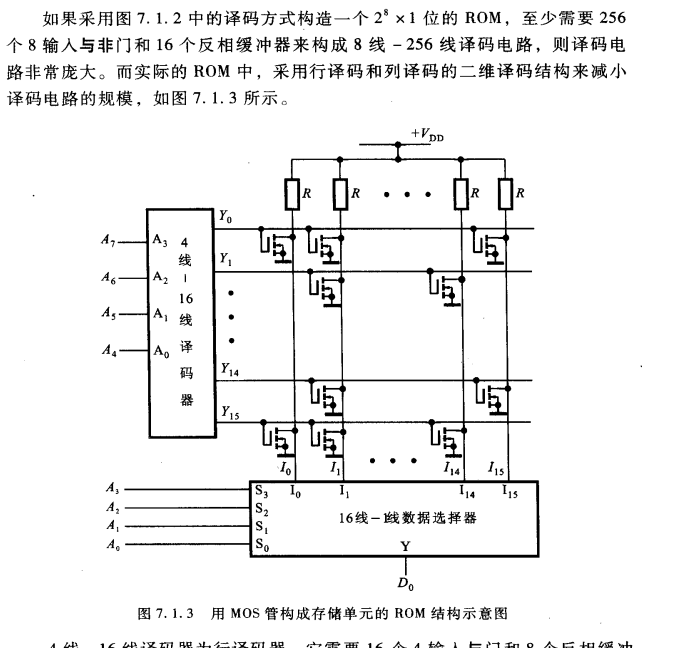
当PUSH操作导致SP指针超过max时，此时叫overflow(上溢)，当POP操作导致SP指针超过0x1000，比如0x1001时，此时叫underflow（下溢），即超过了栈底。（而实际上0x1001在物理上比地址0x1000高）

栈的分类：top-bottom,bottom-top,left-right,right-left

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。



1. 共用体（union）

union，中文名为“共用体”，顾名思义，几种不同数据类型的变量共用一块内存空间，进一步理解就是几个变量的首地址相同。

union的内存分配原则：

1. 首先看最长成员的大小；
2. 其次看最长成员与当前数据对齐方式的关系：**缺了就补**。

共用体一次只能初始化单个元素！

PS:字节对齐主要是针对不同平台取数据的带宽不同，比如，某个平台指定从偶数地址开始取数据，那如果int变量定义在奇数地址，CPU想要获取int变量，则必须要读取两次。

Q1：CPU读/写数据的底层实现与汇编指令关系？

1. 结构体（struct）
   1. 常规结构体数组初始化：

Struct stu stus[] = {

{10,20,30},

{5,6,7},

{1,2,3}

}

* 1. 结构体数组嵌套初始化：

1 #include <stdio.h>

2

3 struct stu\_interesting {

4 int a;

5 int b;

6 };

7

8 struct stu {

9 char a;

10 short b;

11 int c;

12 struct stu\_interesting stu\_int[2];

13 };

14

15 struct stu binbin = {10,20,30};

16

17 int main(int argc, char \*\*argv)

18 {

19 struct stu meimei;

20 struct stu stus[3] = {

21 {

22 .a = 10,

23 .b = 20,

24 .c = 30,

25  **.stu\_int** = { **//当结构体数组的成员还是结构体数组时，以成员数组名字标号，进行初始化，或者采用默认顺序依次初始化**

26 {55,0xaa},

27 {0xaa,55}

28 }

29 },

30

31 {1,2,3,{{1,2},{3,4}}},

32 {11,22,33,{{11,22},{44,55}}}

33 };

* + 1. 嵌套初始化的特别方式：

struct example {

struct addr\_t {

[uint32\_t](http://en.cppreference.com/w/c/types/integer) port;

} addr;

union {

[uint8\_t](http://en.cppreference.com/w/c/types/integer) a8[4];

[uint16\_t](http://en.cppreference.com/w/c/types/integer) a16[2];

} in\_u;

};

struct example ex = { // start of initializer list for struct example

{ // start of initializer list for ex.addr

80 // initialized struct's only member

}, // end of initializer list for ex.addr

{ // start of initializer-list for ex.in\_u

{127,0,0,1} // initializes first element of the union

} };

struct example ex = {80, 127, 0, 0, 1}; // 80 initializes ex.addr.port

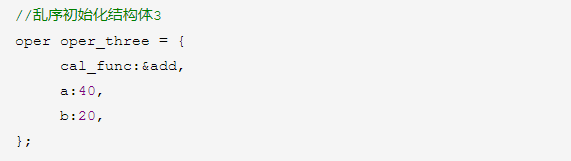
// 127 initializes ex.in\_u.a8[0]

// 0 initializes ex.in\_u.a8[1]

// 0 initializes ex.in\_u.a8[2]

// 1 initializes ex.in\_u.a8[3]

* 1. 结构体初始化的特别方式：



* + 1. 以及这种骚操作：

struct {int n;} s = {[printf](http://en.cppreference.com/w/c/io/fprintf)("a**\n**"), // this may be printed or skipped

.n=[printf](http://en.cppreference.com/w/c/io/fprintf)("b**\n**")}; // always printed

s.n最后的值等于printf的返回值（此处是2）

*（不得不感叹，C语言真是处处埋坑）*