

## 任务1：认识编译器-GCC相关操作练习

作业目的

任务描述

编译的主要阶段

源程序

预处理

编译得到汇编文件

1. 32位汇编

2. 64位汇编

生成目标文件

反汇编

全局/外部符号

生成可执行文件

CRT(C Run-Time)库文件

查看编译的详细过程

---

## 任务1：认识编译器-GCC相关操作练习

### 作业目的

练习使用GCC/CLANG编译C程序，理解并运用各种编译选项

查阅GCC相关教程资料，尝试安装gcc环境，或者直接在网络平台<https://www.godbolt.org>，GCC部分编译选项摘录如下：

- `-E` 只执行预处理
- `-c` 编译或汇编源文件，不执行链接
- `-s` 完成编译但不执行汇编，产生汇编文件
- `-o file` 指定输出的文件为file。如果未指定该选项，在Linux下缺省的是将可执行文件存入 `a.out`，对于 `source.suffix` 的目标文件为 `source.o`、汇编文件为 `source.s`，等
- `-m32`，`-m64`，`-m16` 为32位、64位或16位环境产生代码
- - `-m32` 下int，long和指针类型均为32位
  - `-m64` 下int为32位，long和指针类型均为64位
  - `-m16` 与 `-m32` 类似，只是它会在汇编文件开头输出 `.code16gcc` (针对GCC)汇编制导，从而可以按16位模式运行二进制

### 任务描述

本次作业任务：通过对一个简单的C程序示例 `sample.c`，使用不同编译选项进行编译，得到程序的不同表示形式，尝试理解这些形式之间的对应关系，进而理解编译的主要阶段：预处理、编译、汇编、链接。通过实际操作，回答相关问题，将答案整理在answer.pdf的文件中并提交作业网站。

### 编译的主要阶段

## 源程序

sample.c 内容如下：

```
1  #ifdef NEG
2  #define M -4
3  #else
4  #define M 4
5  #endif
6  int main()
7  {
8      int a = M;
9      if (a)
10         a = a + 4;
11     else
12         a = a * 4;
13     return 0;
14 }
```

该程序涉及的主要语言特征有：

- 条件编译(1-5行)：根据是否定义宏NEG,定义不同的M
- 宏定义（第2、4行）以及宏引用（第8行）

## 预处理

在命令行窗口输入 `gcc -E sample.c -o sample.i` 该命令也等同于

`cpp sample.c -o sample.i` 将对 sample.c 进行预处理，生成 sample.i,其内容如下：

```
1  # 1 "sample.c"
2  # 1 "<built-in>" 1
3  # 1 "<built-in>" 3
4  # 368 "<built-in>" 3
5  # 1 "<command line>" 1
6  # 1 "<built-in>" 2
7  # 1 "sample.c" 2
8
9  int main()
10 {
11     int a = 4;
12     if (a)
13         a = a + 4;
14     else
15         a = a * 4;
16     return 0;
17 }
18
```

预处理后的程序文件发生了哪些变化了呢？

- 没有条件编译了，已经根据没有定义NEG,而选择了M定义为4

- 没有宏定义了，所有的宏引用均已经展开，比如第14行原先对宏 `M` 的引用已展开成4

**问题1-1：** 如果在命令行下执行 `gcc -DNEG -E sample.c -o sample.i` 生成的 `sample.i` 与之前的有何区别？

## 编译得到汇编文件

### 1. 32位汇编

在命令行下执行 `gcc -S -m32 sample.c -o sample-32.s` 将对 `sample.i` 进行编译，产生32位汇编代码 `sample-32.s`，内容如下：

将对 `sample.i` 进行编译，产生32位汇编代码 `sample-32.s`，内容如下：

```
1      .file      "sample.c"
2      .text
3      .globl    main
4      .type     main, @function
5  main:
6      .LFB0:
7          .cfi_startproc
8          pushl   %ebp
9          .cfi_def_cfa_offset 8
10         .cfi_offset 5, -8
11         movl    %esp, %ebp
12         .cfi_def_cfa_register 5
13         subl    $16, %esp
14         movl    $4, -4(%ebp)
15         cmpl    $0, -4(%ebp)
16         je      .L2
17         addl    $4, -4(%ebp)
18         jmp     .L3
19  .L2:
20         sall    $2, -4(%ebp)
21  .L3:
22         movl    $0, %eax
23         leave
24         .cfi_restore 5
25         .cfi_def_cfa 4, 4
26         ret
27         .cfi_endproc
28  .LFE0:
29         .size   main, .-main
30         .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.12) 5.4.0 20160609"
31         .section        .note.GNU-stack,"",@progbits
```

上述汇编文件中的核心汇编代码如下：

```
1  main:
2      pushl    %ebp                #保存基址寄存器ebp
3      movl     %esp, %ebp          #把栈顶寄存器的值存入ebp
4      subl     $16, %esp           #在栈顶分配16字节的空间
5      movl     $4, -4(%ebp)        #把立即数4存入局部变量a
```

```

6      cmpl    $0, -4(%ebp)    #比较a是否为0
7      je      .L2            #是则跳转到.L2
8      addl    $4, -4(%ebp)    #不是, 则执行a=a+4
9      jmp     .L3            #跳转到.L3
10     .L2:
11     sall     $2, -4(%ebp)    #将a左移2, 相当于a=a*4
12     .L3:
13     movl     $0, %eax        #将返回值0保存到寄存器eax
14     leave    #相当于movl %ebp,%esp; popl %ebp
15     ret      #返回 (修改eip)

```

## 2. 64位汇编

在命令行下执行 `gcc -S sample.c -o sample.s` 将对 `sample.c` 进行编译, 产生64位汇编代码 `sample.s`, 内容如下:

```

1      .file    "sample.c"
2      .text
3      .globl  main
4      .type   main, @function
5  main:
6  .LFB0:
7      .cfi_startproc
8      pushq   %rbp
9      .cfi_def_cfa_offset 16
10     .cfi_offset 6, -16
11     movq     %rsp, %rbp
12     .cfi_def_cfa_register 6
13     movl     $4, -4(%rbp)
14     cmpl     $0, -4(%rbp)
15     je       .L2
16     addl     $4, -4(%rbp)
17     jmp      .L3
18     .L2:
19     sall     $2, -4(%rbp)
20     .L3:
21     movl     $0, %eax
22     popq     %rbp
23     .cfi_def_cfa 7, 8
24     ret
25     .cfi_endproc
26     .LFE0:
27     .size    main, .-main
28     .ident   "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.12) 5.4.0 20160609"
29     .section .note.GNU-stack,"",@progbits

```

**问题1-2** 请对比 `sample-32.s` 和 `sample.s`, 找出它们的区别, 并上网检索给出产生这些区别的原因。如:

- `pushq` 和 `pushl`

- `rsp` 和 `esp`

## 生成目标文件

在命令行下执行如下命令 `gcc -c sample.c` 或者 `as sample.s -o sample.o` 将产生目标文件，它没有执行链接。

## 反汇编

执行 `objdump -dS sample.o` 可以将目标文件反汇编，输出：

```
1 sample.o:      file format elf64-x86-64
2 Disassembly of section .text:
3 0000000000000000 <main>:
4   0:   55                push    %rbp
5   1:   48 89 e5          mov     %rsp,%rbp
6   4:   c7 45 fc 04 00 00 00 movl    $0x4,-0x4(%rbp)
7   b:   83 7d fc 00       cmpl    $0x0,-0x4(%rbp)
8   f:   74 06            je      17 <main+0x17>
9  11:   83 45 fc 04       addl    $0x4,-0x4(%rbp)
10 15:   eb 04            jmp     1b <main+0x1b>
11 17:   c1 65 fc 02       shll    $0x2,-0x4(%rbp)
12 1b:   b8 00 00 00 00    mov     $0x0,%eax
13 20:   5d                pop     %rbp
14 21:   c3                retq
```

## 全局/外部符号

执行 `nm sample.o` 可以输出该目标文件的全局符号，即：

```
1 0000000000000000 T main
```

## 生成可执行文件

执行如下命令之一 `gcc sample.c -o sample` `gcc sample.s -o sample` `gcc sample.o -o sample` 将产生可执行文件 `sample`。

由 `sample.o` 得到可执行文件是通过调用链接器 `ld` 得到的，但是直接执行 `ld sample.o -o sample` 会产生如下警告

```
1 ld: warning: cannot find entry symbol _start; defaulting to 00000000004001bb
```

这是为什么呢？主要原因是因为没有链接上需要的 `crt` 文件。

## CRT(C Run-Time)库文件

在 `/usr/lib/x86_64-linux-gnu/` 下包含如下几个 `crt*.o` 文件：

- `crt1.o` 包含程序的入口函数 `_start`，它负责调用 `__libc_start_main` 初始化 `libc` 并且调用 `main` 函数进入真正的程序主体
- `crti.o` 包含 `_init()` 函数，该函数在 `main` 函数前运行

- `crt0.o` 包含 `_finit()` 函数, 该函数在 `main` 函数后运行

你可以显式地将目标文件与这些crt文件链接, 来得到可执行文件, 即执行: `ld /usr/lib/x86_64-linux-gnu/crt1.o /usr/lib/x86_64-linux-gnu/crti.o sample.o -lc -o sample` 则可以产生可执行程序, 其中 `-lc` 表示链接C标准库, 其中提供

- `__libc_start_main (main, __libc_csu_init, __libc_csu_fini)`
- `__libc_csu_init` (负责调用 `_init()`)
- `__libc_csu_fini` (负责调用 `_finit()`)

## 查看编译的详细过程

你可以在执行gcc命令时加上 `-v` 选项, 获得所执行的详细命令行及输出。

**问题1-3** 你可以用 `clang` 替换 `gcc`, 重复上面的各步, 比较使用 `clang` 和 `gcc` 分别输出的结果有何异同。