



杭州电子科技大学

《编译原理课程实践》

实验报告

题 目： 认识编译器-GCC 相关操作练习

学 院： 计算机学院

专 业： 计算机科学与技术

班 级： 22052312

学 号： 22050201

姓 名： 黄江晔

完成日期： 2024. 10. 24

一、 实验目的

本实验的目的是了解工业界常用的编译器 GCC 和 CLANG，熟悉编译器的安装和使用过程，观察编译器工作过程中生成的中间文件的格式和内容，了解编译器的优化效果，为编译器的学习和构造奠定基础。

二、 实验内容与实验要求

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。

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

三、 设计方案与算法描述

(一) 源程序

`sample.c` 文件内容如下：



```
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(h)
hdu@hdu-vm:~$ nano sample.c
hdu@hdu-vm:~$ cat sample.c
#ifdef NEG
#define M -4
#else
#define M 4
#endif
int main()
{
    int a = M;
    if (a)
        a = a + 4;
    else
        a = a * 4;
    return 0;
}
hdu@hdu-vm:~$
```

该程序涉及的主要语言特征有：

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

(二) 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：

```
hdu@hdu-vm: ~
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
hdu@hdu-vm:~$ gcc -E sample.c -o sample.i
hdu@hdu-vm:~$ cat sample.i
# 1 "sample.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "sample.c"

int main()
{
int a = 4;
if (a)
a = a + 4;
else
a = a * 4;
return 0;
}
```

预处理后的程序文件没有条件编译了，已经根据没有定义 `NEG` 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。

（三）编译得到汇编文件

（1）32 位汇编

`gcc -S -m32 sample.c -o sample-32.s`

```
hdu@hdu-vm:~$ gcc -S -m32 sample.c -o sample-32.s
hdu@hdu-vm:~$ cat sample-32.s
.file "sample.c"
.text
.globl main
.type main, @function
main:
.LFB0:
.cfi_startproc
pushl %ebp
.cfi_def_cfa_offset 8
.cfi_offset 5, -8
movl %esp, %ebp
.cfi_def_cfa_register 5
subl $16, %esp
call __x86.get_pc_thunk.ax
addl $ _GLOBAL_OFFSET_TABLE_, %eax
movl $4, -4(%ebp)
cmpl $0, -4(%ebp)
je .L2
addl $4, -4(%ebp)
jmp .L3
.L2:
sall $2, -4(%ebp)
.L3:
movl $0, %eax
leave
.cfi_restore 5
.cfi_def_cfa 4, 4
ret
.cfi_endproc
.LFE0:
.size main, .-main
.section .text.__x86.get_pc_thunk.ax,"axG",@progbits,__x86.get_pc_thunk.ax,comdat
.globl __x86.get_pc_thunk.ax
.hidden __x86.get_pc_thunk.ax
.type __x86.get_pc_thunk.ax, @function
__x86.get_pc_thunk.ax:
.LFB1:
.cfi_startproc
movl (%esp), %eax
ret
.cfi_endproc
.LFE1:
.ident "GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0"
.section .note.GNU-stack,"",@progbits
```

核心代码：

```

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

    cmpl    $0, -4(%ebp)        #比较a是否为0
    je      .L2                 #是则跳转到.L2
    addl    $4, -4(%ebp)        #不是, 则执行a=a+4
    jmp     .L3                 #跳转到.L3

.L2:
    sall    $2, -4(%ebp)        #将a左移2, 相当于a=a*4

.L3:
    movl    $0, %eax            #将返回值0保存到寄存器eax
    leave   0, 0                #相当于movl %ebp,%esp; popl %ebp
    ret     0                   #返回 (修改eip)

```

(2) 64 位汇编

gcc -S sample.c -o sample.s

```

hdu@hdu-vm:~$ gcc -S sample.c -o sample.s
hdu@hdu-vm:~$ cat sample.s
.file "sample.c"
.text
.globl main
.type main, @function

main:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movl    $4, -4(%rbp)
    cmpl    $0, -4(%rbp)
    je      .L2
    addl    $4, -4(%rbp)
    jmp     .L3
.L2:
    sall    $2, -4(%rbp)
.L3:
    movl    $0, %eax
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   main, .-main
    .ident  "GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0"
    .section .note.GNU-stack,"",@progbits

```

(二) 生成目标文件

gcc -c sample.c

```

hdu@hdu-vm:~$ gcc -c sample.c
hdu@hdu-vm:~$ cat sample.o
ELF 64-bit LSB shared object, Intel x86-64, version 1.0,
  built1: sample.cmain, .symtab, .strtab, .shstrtab, .text, .data, .bss, .comment, .note.GNU-stack, .rela.eh_frame

```

反汇编:

objdump -dS sample.o

```

hdu@hdu-vm:~$ objdump -dS sample.o
sample.o:          文件格式 elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
 0:  55                      push    %rbp
 1:  48 89 e5                mov     %rsp,%rbp
 4:  c7 45 fc 04 00 00 00    movl   $0x4,-0x4(%rbp)
 b:  83 7d fc 00             cmpl   $0x0,-0x4(%rbp)
 f:  74 06                  je     17 <main+0x17>
11:  83 45 fc 04            addl   $0x4,-0x4(%rbp)
15:  eb 04                  jmp     1b <main+0x1b>
17:  c1 65 fc 02           shll   $0x2,-0x4(%rbp)
1b:  b8 00 00 00 00        mov     $0x0,%eax
20:  5d                      pop     %rbp
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。

```

hdu@hdu-vm:~$ nm sample.o
0000000000000000 T main

```

(四) 生成可执行文件

执行 `gcc sample.c -o sample` 可得到可执行文件，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，主要原因是因为没有链接上需要的 `crt` 文件。

```

hdu@hdu-vm:~$ ld sample.o -o sample
ld: 警告: 无法找到项目符号 _start; 缺省为 00000000004000b0

```

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，`lcrtn.o` 包含 `_fini()` 函数，该函数在 `main` 函数后运行，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 _fini() )

```

```

hdu@hdu-vm:~$ ld /usr/lib/x86_64-linux-gnu/crt1.o /usr/lib/x86_64-linux-gnu/crti.o sample.o -lc -o sample
hdu@hdu-vm:~$ find -name sample.o
./sample.o

```

四、 测试结果

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？


```
hdu@hdu-vm: ~  
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)  
hdu@hdu-vm:~$ gcc -DNEG -E sample.c -o sample.i  
hdu@hdu-vm:~$ cat sample.i  
# 1 "sample.c"  
# 1 "<built-in>"  
# 1 "<command-line>"  
# 1 "/usr/include/stdc-predef.h" 1 3 4  
# 1 "<command-line>" 2  
# 1 "sample.c"  
  
int main()  
{  
int a = -4;  
if (a)  
a = a + 4;  
else  
a = a * 4;  
return 0;  
}
```

由于 NEG 被定义，预处理器会执行 #ifdef NEG 的分支，并将 M 定义为 -4。

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。

(1) 寄存器使用：

在 32 位代码中，使用的是 32 位寄存器，比如 %ebp 和 %eax。

在 64 位代码中，使用的是 64 位寄存器，比如 %rbp 和 %rax。

(2) 栈指针操作：

32 位代码中，使用 pushl %ebp 和 movl %esp, %ebp 来保存栈帧和移动栈指针。

64 位代码中，使用 pushq %rbp 和 movq %rsp, %rbp，因为 64 位体系下的栈指针和栈基址都是 64 位寄存器。

(3) 地址操作：

32 位系统使用 subl \$16, %esp 调整栈空间。

64 位系统使用 subq \$16, %rsp 做类似操作，但针对 64 位寄存器。

(4) 跳转指令和比较：

在 32 位代码中，比较操作和跳转使用的是 cmpl（32 位比较），而在 64 位代码中，使用的是 cmpq（64 位比较）。

(5) 返回地址和栈恢复：

在 32 位代码中，使用 leave 和 ret 指令来返回。

在 64 位代码中，使用 popq %rbp 和 ret 完成类似的操作，但处理的是 64 位栈。

产生这些区别的原因：

主要是因为编译目标架构不同。32 位程序和 64 位程序在处理器上操作的寄存器宽度不同（32 位和 64 位），导致生成的汇编代码中使用的寄存器、指令集、栈操作等都有差异。

32 位架构（如 x86）只能处理 32 位的寄存器和地址，所以使用 %ebp、%esp 这样的 32 位寄存器。

64 位架构（如 x86-64）能够处理 64 位宽的寄存器，因此使用了 %rbp、%rsp 这样的 64 位寄存器。

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。

```

1  a = a + 4;
2  else
3  a = a * 4;
4  return 0;
5
6  hdughdugu-vim:~$ clang -S -m32 sample.c -o sample-32.s
7  hdughdugu-vim:~$ cat sample-32.s
8
9  .file
10 "sample.c"
11
12 .globl main
13
14 .type main,@function
15
16 main:
17
18 LF00:
19
20 .cfl_startproc
21 pushl %ebp
22 .cfl_def_cfa_offset 8
23 .cfl_offset 5, 8
24 movl %esp, %ebp
25 .cfl_def_cfa_register 5
26 subl $10, %esp
27 call __806.get_pc_thunk.ax
28 addl $,GLOBAL_OFFSET_TABLE__, %eax
29 .L1:
30 movl %eax, %ecx
31 cmpl $0, -4(%ebp)
32 jle .L2
33 addl $4, -4(%ebp)
34 jmp .L3
35
36 .L2:
37
38 .L3:
39 movl $0, %eax
40 leave
41
42 .cfl_restore 5
43 .cfl_def_cfa 4, 4
44 ret
45
46 .cfl_endproc
47
48 LF01:
49
50 .size main, .-main
51 .section
52 .text, __806.get_pc_thunk.ax, "ax",@progbits, __806.get_pc_thunk
53
54 .globl __806.get_pc_thunk.ax
55 .hidden __806.get_pc_thunk.ax
56 .type __806.get_pc_thunk.ax,@function
57
58 __806.get_pc_thunk.ax:
59
60 LF02:
61
62 .cfl_startproc
63 movl (%esp), %eax
64 ret
65
66 .cfl_endproc
67
68 LF03:
69
70 .ident "GCC: (Ubuntu 7.5.0-3ubuntu1-18.04) 7.5.0"
71 .section
72 ".note.GNU-stack","",@progbits
73
74 .ident "clang version 6.0.0-1ubuntu2 (tags/RELEASE_600/final)"
75 .section
76 ".note.GNU-stack","",@progbits

```

clang 生成的汇编代码更简洁，带有更多的优化注释，而 gcc 生成的汇编代码则包含更多细节和调试信息。

clang 和 gcc 对于局部变量和临时值的处理有所不同。clang 更频繁地使用寄存器存储临时值，避免不必要的栈操作，而 gcc 会为更多的局部变量分配栈空间。

五、 源代码

```
#ifndef NEG
#define M -4
#else
#define M 4
#endif
int main()
{
    int a = M;
    if (a)
        a = a + 4;
    else
        a = a * 4;
    return 0;
}
```