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：22052313

学　　号：22050324

姓　　名：林灿

完成日期：2024.10.21

一、 实验目的

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

二、 实验内容与实验要求

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。

三、 设计方案与算法描述

本实验暂无涉及到算法等设计。

四、 测试结果

已有一份 c 语言代码，对其进行预处理

```
#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;
}
```

问 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?

#gcc -E sample.c -o sample.i

```
# 1 "sample.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 414 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "sample.c" 2




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

#gcc -DNEG -E sample.c -o sample2.i

```
# 1 "sample.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 414 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "sample.c" 2




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

可以发现，在 main 函数中，M 被替换为了-4，原因是 gcc -Dxx 相当于 #define xx，表示定义宏的意图。在这里相当于定义了 NEG，于是在宏定义的判断中执行的是 #define M -4 此语句。

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。

注意：本人使用 macOS 是 arm 架构，内容与实验手册上有些出入。

sample-32.s:

```
	.section	__TEXT,__text,regular,pure_instructions
	.build_version macos, 13, 0	sdk_version 13, 3
	.syntax unified
	.globl	_main                                   @ -- Begin function main
	.p2align	2
	.code	32                                      @ @main
_main:
@ %bb.0:
	sub	sp, sp, #8
	mov r0, #0
	str	r0, [sp, #4]
	mov r0, #4
	str	r0, [sp]
	ldr	r0, [sp]
	cmp r0, #0
	beq	LBB0_2
	b	LBB0_1
LBB0_1:
	ldr	r0, [sp]
	add	r0, r0, #4
	str	r0, [sp]
	b	LBB0_3
LBB0_2:
	ldr	r0, [sp]
	lsl	r0, r0, #2
	str	r0, [sp]
	b	LBB0_3
LBB0_3:
	mov r0, #0
	add	sp, sp, #8
	bx	lr
                                                    @ -- End function
.subsections_via_symbols
```

sample.s:

```
.section	__TEXT,__text,regular,pure_instructions
	.build_version macos, 13, 0	sdk_version 13, 3
```

```
	.globl	_main                                   ; -- Begin function main
	.p2align	2
_main:                                  ; @main
	.cfi_startproc
; %bb.0:
	sub	sp, sp, #16
	.cfi_def_cfa_offset 16
	str	wzr, [sp, #12]
	mov	w8, #4
	str	w8, [sp, #8]
	ldr	w8, [sp, #8]
	subs	w8, w8, #0
	cset	w8, eq
	tbnz	w8, #0, LBB0_2
	b	LBB0_1
LBB0_1:
	ldr	w8, [sp, #8]
	add	w8, w8, #4
	str	w8, [sp, #8]
	b	LBB0_3
LBB0_2:
	ldr	w8, [sp, #8]
	lsl	w8, w8, #2
	str	w8, [sp, #8]
	b	LBB0_3
LBB0_3:
	mov	w0, #0
	add	sp, sp, #16
	ret
	.cfi_endproc
                                        ; -- End function
.subsections_via_symbols
```

如代码中标红内容所示：

1. 寄存器使用：64 位代码使用 wzr（零寄存器）来存储零值，而 32 位代码使用 mov r0, #0。

2. 条件分支：64 位代码使用 subs 和 cset 指令来进行条件设置和分支，而 32 位代码使用 cmp 和 beq。

3. 函数返回：64 位代码使用 ret 指令来返回。32 位代码使用 bx lr (跳转到链接寄存器)来返回。

反汇编结果：

```
sample.o:	file format mach-o arm64

Disassembly of section __TEXT,__text:

0000000000000000 <ltmp0>:
       0: ff 43 00 d1	sub	sp, sp, #16
       4: ff 0f 00 b9	str	wzr, [sp, #12]
       8: 88 00 80 52	mov	w8, #4
       c: e8 0b 00 b9	str	w8, [sp, #8]
      10: e8 0b 40 b9	ldr	w8, [sp, #8]
      14: 08 01 00 71	subs	w8, w8, #0
      18: e8 17 9f 1a	cset	w8, eq
      1c: c8 00 00 37	tbnz	w8, #0, 0x34 <ltmp0+0x34>
      20: 01 00 00 14	b	0x24 <ltmp0+0x24>
      24: e8 0b 40 b9	ldr	w8, [sp, #8]
      28: 08 11 00 11	add	w8, w8, #4
      2c: e8 0b 00 b9	str	w8, [sp, #8]
      30: 05 00 00 14	b	0x44 <ltmp0+0x44>
      34: e8 0b 40 b9	ldr	w8, [sp, #8]
      38: 08 75 1e 53	lsl	w8, w8, #2
      3c: e8 0b 00 b9	str	w8, [sp, #8]
      40: 01 00 00 14	b	0x44 <ltmp0+0x44>
      44: 00 00 80 52	mov	w0, #0
      48: ff 43 00 91	add	sp, sp, #16
      4c: c0 03 5f d6	ret
```

nm sample.o 结果：

```
0000000000000000 T _main
0000000000000000 t ltmp0
0000000000000050 s ltmp1
```

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。

输出结果：

```
Apple clang version 14.0.3 (clang-1403.0.22.14.1)
Target: arm64-apple-darwin22.5.0
Thread model: posix
InstalledDir: /Library/Developer/CommandLineTools/usr/bin
 "/Library/Developer/CommandLineTools/usr/bin/clang" -cc1 -triple
arm64-apple-macosx13.0.0 -Wundef-prefix=TARGET_OS_ -Wdeprecated-objc-isa-usage
```

```
-Werror=deprecated-objc-isa-usage -Werror=implicit-function-declaration -E -disable-free -clear-ast-before-backend -disable-llvm-verifier -discard-value-names -main-file-name sample.c -mrelocation-model pic -pic-level 2 -mframe-pointer=non-leaf -fno-strict-return -ffp-contract=on -fno-rounding-math -funwind-tables=1 -fobjc-msgsend-selector-stubs -target-sdk-version=13.3 -fvisibility-inlines-hidden-static-local-var -target-cpu apple-m1 -target-feature +v8.5a -target-feature +crc -target-feature +lse -target-feature +rdm -target-feature +crypto -target-feature +dotprod -target-feature +fp-armv8 -target-feature +neon -target-feature +fp16fml -target-feature +ras -target-feature +rcpc -target-feature +zcm -target-feature +zcz -target-feature +fullfp16 -target-feature +sm4 -target-feature +sha3 -target-feature +sha2 -target-feature +aes -target-abi darwinpcs -fallow-half-arguments-and-returns -debugger-tuning=lldb -target-linker-version 857.1 -v -fcoverage-compilation-dir=/Users/alvin/Homeworks/编译原理/实验/实验一 -resource-dir /Library/Developer/CommandLineTools/usr/lib/clang/14.0.3 -isysroot /Library/Developer/CommandLineTools/SDKs/MacOSX.sdk -I/usr/local/include -internal-isystem /Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/local/include -internal-isystem /Library/Developer/CommandLineTools/usr/lib/clang/14.0.3/include -internal-externc-isystem /Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include -internal-externc-isystem /Library/Developer/CommandLineTools/usr/include -Wno-reorder-init-list -Wno-implicit-int-float-conversion -Wno-c99-designator -Wno-final-dtor-non-final-class -Wno-extra-semi-stmt -Wno-misleading-indentation -Wno-quoted-include-in-framework-header -Wno-implicit-fallthrough -Wno-enum-enum-conversion -Wno-enum-float-conversion -Wno-elaborated-enum-base -Wno-reserved-identifier -Wno-gnu-folding-constant -fdebug-compilation-dir=/Users/alvin/Homeworks/编译原理/实验/实验一 -ferror-limit 19 -stack-protector 1 -fstack-check -mdarwin-stkchk-strong-link -fblocks -fencode-extended-block-signature -fregister-global-dtors-with-atexit -fgnuc-version=4.2.1 -no-opaque-pointers -fmax-type-align=16 -fcommon -fcolor-diagnostics -clang-vendor-feature=+disableNonDependentMemberExprInCurrentInstantiation -fno-odr-hash-protocols -clang-vendor-feature=+enableAggressiveVLAFolding -clang-vendor-feature=+revert09abecef7bbf -clang-vendor-feature=+thisNoAlignAttr -clang-vendor-feature=+thisNoNullAttr -mllvm -disable-aligned-alloc-awareness=1 -D__GCC_HAVE_DWARF2_CFI_ASM=1 -o sample-gcc.i -x c sample.c
clang -cc1 version 14.0.3 (clang-1403.0.22.14.1) default target arm64-apple-darwin22.5.0
ignoring nonexistent directory "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/local/include"
ignoring nonexistent directory "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/Library/Frameworks"
#include "..." search starts here:
#include <...> search starts here:
 /usr/local/include
 /Library/Developer/CommandLineTools/usr/lib/clang/14.0.3/include
 /Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include
```

```
 /Library/Developer/CommandLineTools/usr/include
 /Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/System/Library/Frameworks
(framework directory)
End of search list.
```

经过测试发现，在我的系统中使用 gcc 与 clang 得到的输出是一模一样的，查证资料后，原因在于：

In Apple's version of GCC, both cc and gcc are actually symbolic links to the llvm-gcc compiler. Similarly, c++ and g++ are links to llvm-g++.

使用 man llvm-gcc 后：

llvm-gcc is a C, C++, Objective-C and Objective-C++ compiler. llvm-g++ is a compiler driver for C++. llvm-gcc uses gcc front-end and gcc's command line interface.

因此得出结论，在 macOS 上 gcc 的本质就是 clang。

查阅资料后得知，一般来说，GCC 会更加大型且复杂，GCC 在支持多种平台和架构方面有着更广泛的历史和经验。Clang 在 macOS 和 iOS 平台上特别受欢迎，因为它是 Xcode 开发环境的一部分。但是总的生成没有太大的区别

五、 源代码

```
预处理
gcc -E sample.c -o sample.i
gcc -DNEG -E sample.c -o sample2.i

编译 32 位与 64 位
gcc -S -m32 sample.c -o sample-32.s
gcc -S sample.c -o sample-64.s

输出文件
gcc -c sample.c
objdump -dS sample.o

查看细节
```

```
gcc -v    sample.c -o sample
```

其余所有文件内容均在压缩包文件中