

# 编译系统原理第一次作业

## 了解你的编译器

姓名：马平川<sup>①</sup>

学号：1511442

学院：软件学院

专业：软件工程

2017 年 9 月 22 日

<sup>①</sup> pika7ma@gmail.com

## 摘 要

编译器是计算机程序编写执行中十分重要的一环，通过探究编译器的完整工作过程我们能够更加深入地了解程序的始终。本文以编译器工作过程为研究课题，重点研究了预处理器、编译器、汇编器和链接器的工作流程，并后续探究了语言间、实现间的差异和语言修饰、语言格式对编译过程的影响，并进一步对编译器其他参数（如优化参数）进行了实验。

**关键词：** 预处理器，编译器，汇编器，链接器，C 语言，C++ 语言



## ABSTRACT

Since the compiler has long been playing a dispensable role in programming field, unpacking the black box can provide us more insights with respect to program from its cradle to its grave. The overall running procedure of compiler is researched in this dissertation, subsequently, detailed internal steps including preprocessor, compiler, assembler and linker are discussed respectively. Then, the impact of language, implementation, decoration and coding style are listed, furthermore, the research also tested other parameters such as the optimization.

**Keywords:** preprocessor, compiler, assembler, linker, C programming language, C++ programming language



## 目 录

<b>第一章 绪 论</b> .....	1
1.1 研究工作的目的及意义 .....	1
1.2 本论文的结构安排 .....	1
<b>第二章 从源代码到可执行文件</b> .....	2
2.1 流程整体把握 .....	2
2.2 拆分理解.....	2
2.2.1 预处理器.....	2
2.2.2 编译器 .....	3
2.2.3 汇编器 .....	4
2.2.4 链接器 .....	4
2.2.5 执行文件.....	5
<b>第三章 不同编译器与不同语言</b> .....	6
3.1 GCC 编译探究.....	6
3.1.1 GCC 简介.....	6
3.1.2 使用 GCC 编译 C 程序 .....	6
3.1.3 使用 GCC 编译 C++ 程序 .....	7
3.2 G++ 编译探究.....	9
3.2.1 G++ 简介 .....	9
3.2.2 G++ 编译过程.....	9
3.2.3 C 和 C++ 编译差异 .....	9
3.2.4 GCC 和 G++ 下 C 编译文件差异 .....	9
<b>第四章 同一功能的不同实现</b> .....	11
4.1 阶乘的循环实现.....	11
4.2 阶乘的递归实现.....	12
4.3 差异对比.....	13
4.3.1 预处理文件 .....	13
4.3.2 汇编文件.....	13
4.3.3 差异分析.....	14
<b>第五章 编程风格的影响</b> .....	15
5.1 环境变量.....	15

5.2 结果差异 .....	15
<b>第六章 编译器优化参数</b> .....	17
6.1 编译器优化参数介绍 .....	17
6.1.1 不优化 .....	17
6.1.2 一级优化 .....	17
6.1.3 二级优化 .....	17
6.1.4 三级优化 .....	17
6.1.5 空间优化 .....	17
6.2 不同优化级别性能对比 .....	17
6.2.1 运行速度比较 .....	17
6.2.2 文件大小对比 .....	19
<b>第七章 警告和调试开关</b> .....	20
7.1 警告开关 .....	20
7.2 调试开关 .....	20
<b>第八章 自动并行化</b> .....	22
8.1 GCC 与 OpenMP .....	22
8.2 自动并行化实践 .....	22
<b>第九章 总结与回顾</b> .....	25
9.1 全文总结 .....	25
9.2 后续工作展望 .....	25
<b>参考文献</b> .....	26

## 第一章 绪 论

### 1.1 研究工作的目的及意义

在计算机发展初期，计算机程序都是由汇编语言（甚至是穿孔纸带）直接写成。不难想到，不仅这种语言的编写费时费力，对每种机型都得更改源码，更加不方便的是，这种编程方式使得程序员通常花费大量时间在没必要的架构和修饰问题上，而非解决问题本身。之后人们发明了高级编程语言，使得语言能够独立于机器存在，自此，编译器便担当了把高级语言翻译成机器语言的工作。而作为计算机中不可或缺的一类特殊程序，研究并了解编译器这一黑箱也成为了现代程序员必备的素养。

本文以探究编译器工作方式为目的，针对编译器对不同程序、不同编译器对同一程序等模式进行了详细实验，不仅为编译系统学习打下基础，更能为今后的程序优化提供经验。本项目全部源码开源于GitHub。

### 1.2 本论文的结构安排

本文的章节结构安排如下：

1. 简单探究预处理器，编译器，汇编器和链接器的工作原理和存在意义
2. 实验不同语言（C 与 C++）编译器工作区别
3. 实验同一功能不同实现的区别
4. 实验不同编程风格和其他修饰性对编译过程的影响
5. 实验编译器其他参数
6. 总结与回顾



## 第二章 从源代码到可执行文件

### 2.1 流程整体把握

现代高级程序的生命从一个高级语言程序文件开始，高级语言能够被人类读懂，但机器却并不能按照其执行，因为实在过于复杂。为了在操作系统上运行这个程序，源文件需要被层层拆解成为机器语言，并最终将机器语言中的指令以可执行程序的格式打包并保存为二进制文件。

如下图所示，这一整套从源代码到可执行文件的过程在现代编译器中便是以预处理器、编译器、汇编器、链接器四过程组成的：

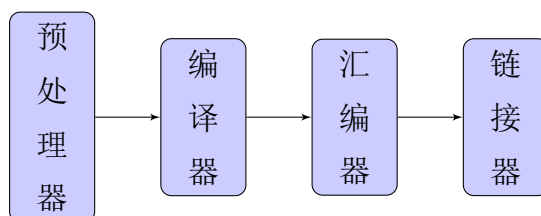


图 2-1 编译工具链整体流程

下面将分解展现各部分的作用。

### 2.2 拆分理解

#### 2.2.1 预处理器

在 GCC 中，预处理器的作用是将高级语言中的预处理（即以 '#' 开头的部分）纳入程序中，其中包括 "#define"、"#include" 等。例如我们编写一段带有预处理指令的程序 `hello_world.c`：

```
1 | #include <stdio.h>
2 | #define HW "Hello World!\n"
3 | int main() {
4 |     printf(HW);
5 |     return 0;
6 | }
```

预处理器的工作便是替换 `HW` 并包入 `stdio.h` 库，其执行完成后的结果实际上还是 C 语言，不过加入了大量代替预处理语句的晦涩语言，其中生成的代替语句中的一部分如下所示：

```
1 | # 1 "hello_world.c"
```

```

2 | # 1 "<built-in>"
3 | # 1 "<command-line>"
4 | # 1 "/usr/include/stdc-predef.h" 1 3 4
5 | # 1 "<command-line>" 2
6 | # 1 "hello_world.c"
7 | # 1 "/usr/include/stdio.h" 1 3 4
8 | # 27 "/usr/include/stdio.h" 3 4
9 | # 1 "/usr/include/features.h" 1 3 4
10 | # 367 "/usr/include/features.h" 3 4

```

在 GCC 中，我们可以通过以下命令来使编译器预处理后即停止，不进行编译、汇编及连接：

```
$ gcc -E hello_world.c -o hello_world.i
```

### 2.2.2 编译器

编译器的工作是将预处理器处理过的 "\*.i" 文本文件翻译成为标准的汇编语言以供计算机阅读。如下即为其生成的汇编程序：

```

1 | .file "hello_world.c"
2 | .section .rodata
3 | .LC0:
4 | .string "Hello World!"
5 | .text
6 | .globl main
7 | .type main, @function
8 | main:
9 | .LFB0:
10 | .cfi_startproc
11 | pushq %rbp
12 | .cfi_def_cfa_offset 16
13 | .cfi_offset 6, -16
14 | movq %rsp, %rbp
15 | .cfi_def_cfa_register 6
16 | movl $.LC0, %edi
17 | call puts
18 | movl $0, %eax
19 | popq %rbp
20 | .cfi_def_cfa 7, 8
21 | ret

```

```

22 |         .cfi_endproc
23 | .LFE0:
24 |         .size    main, .-main
25 |         .ident   "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.4) 5.4.0
    |             20160609"
26 |         .section        .note.GNU-stack,"",@progbits

```

我们可以看出，虽然在预处理阶段加入了非常大量的引用库代码，但在生成汇编语言中只会生成程序所需要的，这也是编译器的工作之一。

在 GCC 中，我们可以通过以下命令来使编译器编译后即停止，不进行汇编及连接：

```

|| $ gcc -S hello_world.i -o hello_world.s

```

### 2.2.3 汇编器

在汇编器中，汇编语言被翻译成机器语言指令，并打包成可重定位目标程序。在这一步，我们之前的程序就已经被转换为了机器使用的二进制语言，其生成二进制文件的节选如下所示：

```

|| 0000 005d c348 656c 6c6f 2057 6f72 6c64
|| 2100 0047 4343 3a20 2855 6275 6e74 7520
|| 352e 342e 302d 3675 6275 6e74 7531 7e31
|| 362e 3034 2e34 2920 352e 342e 3020 3230
|| 3136 3036 3039 0000 1400 0000 0000 0000
|| 017a 5200 0178 1001 1b0c 0708 9001 0000
|| 1c00 0000 1c00 0000 0000 0000 1500 0000
|| 0041 0e10 8602 430d 0650 0c07 0800 0000

```

在 GCC 中，我们可以通过以下命令来使编译器编译或汇编源文件，但不进行连接：

```

|| $ gcc -c hello_world.s -o hello_world.o

```

### 2.2.4 链接器

在这里，机器语言距可执行文件只差一部。在链接器中，机器语言与其他机器特殊的代码文件和库文件进行链接，最终创建出可执行目标文件。在 GCC 中，我们可以通过以下命令来指定输出可执行文件：

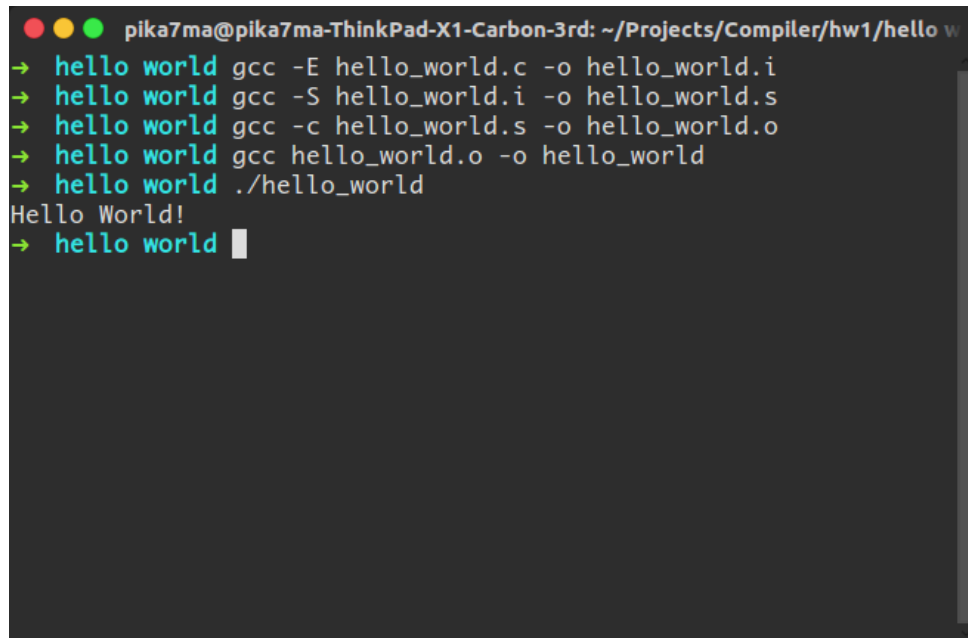
```

|| $ gcc hello_world.o -o hello_world

```

### 2.2.5 执行文件

当我们得到可执行文件后，我们便可以直接运行它。整体流程如图所示：



```
pika7ma@pika7ma-ThinkPad-X1-Carbon-3rd: ~/Projects/Compiler/hw1/hello w
→ hello world gcc -E hello_world.c -o hello_world.i
→ hello world gcc -S hello_world.i -o hello_world.s
→ hello world gcc -c hello_world.s -o hello_world.o
→ hello world gcc hello_world.o -o hello_world
→ hello world ./hello_world
Hello World!
→ hello world
```

图 2-2 从源码到运行

## 第三章 不同编译器与不同语言

### 3.1 GCC 编译探究

#### 3.1.1 GCC 简介

GCC，全称 GNU Compiler Collection，是理查德·马修·斯托曼于 1985 年开始发展的一款编译器。有趣的是其最初只针对 C 语言，全称为 GNU C Compiler，而后来逐渐发展加入了 Ada，Objective-C，C++ 等众多语言，并正式更名为现在的名字。



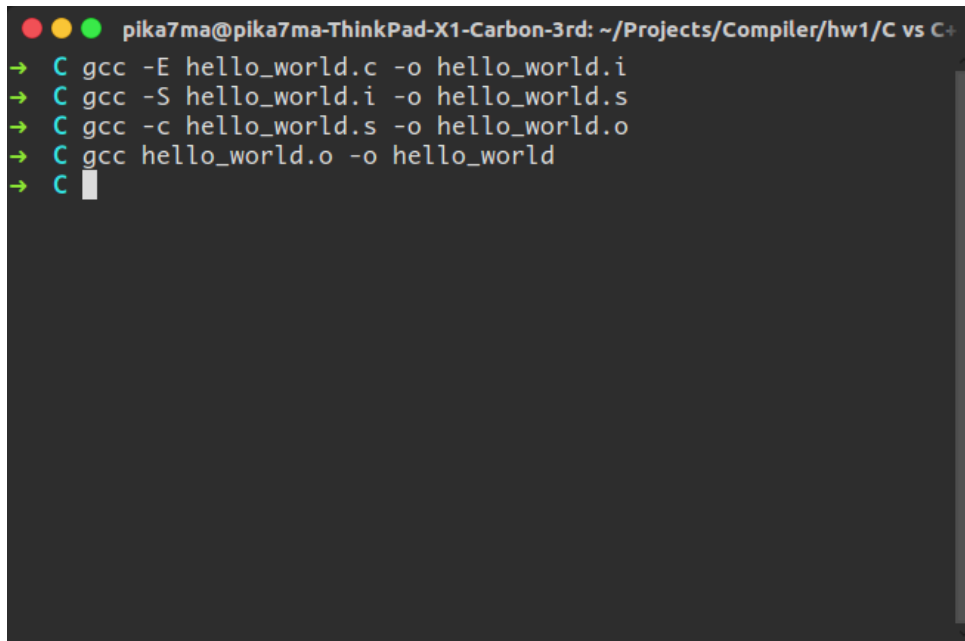
图 3-1 GCC

#### 3.1.2 使用 GCC 编译 C 程序

在这里我们使用最简单的 `hello_world.c` 程序：

```
1 | #include <stdio.h>
2 | int main() {
3 |     printf("Hello World!\n");
4 |     return 0;
5 | }
```

逐步编译过程：

A terminal window with a dark background and light-colored text. The prompt is 'pika7ma@pika7ma-ThinkPad-X1-Carbon-3rd: ~/Projects/Compiler/hw1/C vs C+'. There are five lines of commands, each preceded by a green arrow: 'C gcc -E hello\_world.c -o hello\_world.i', 'C gcc -S hello\_world.i -o hello\_world.s', 'C gcc -c hello\_world.s -o hello\_world.o', 'C gcc hello\_world.o -o hello\_world', and 'C' followed by a cursor. The terminal has a standard Linux-style window title bar with red, yellow, and green buttons.

```
pika7ma@pika7ma-ThinkPad-X1-Carbon-3rd: ~/Projects/Compiler/hw1/C vs C+
→ C gcc -E hello_world.c -o hello_world.i
→ C gcc -S hello_world.i -o hello_world.s
→ C gcc -c hello_world.s -o hello_world.o
→ C gcc hello_world.o -o hello_world
→ C
```

图 3-2 使用 GCC 编译 C

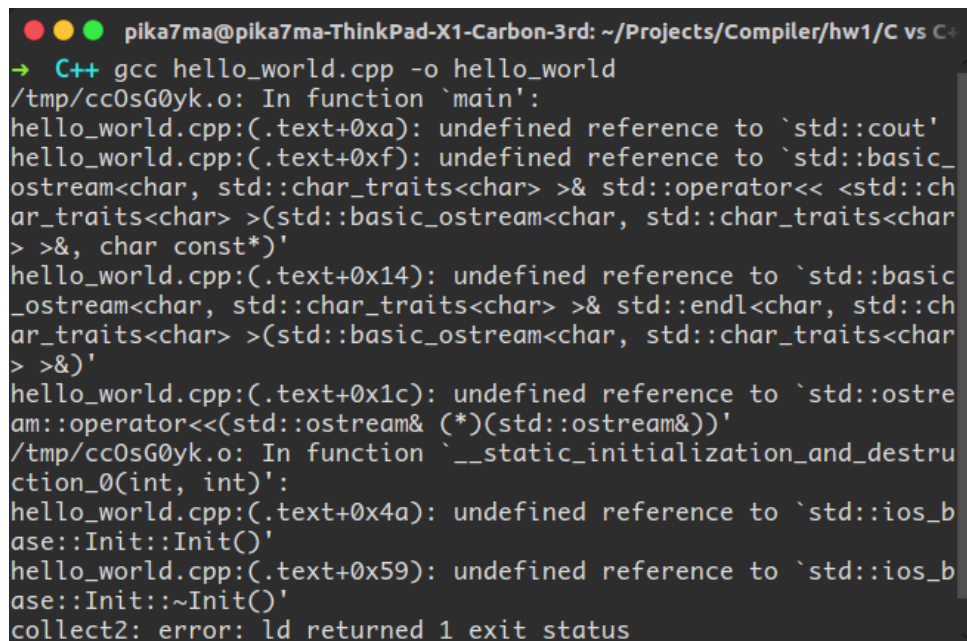
可以看出，GCC 对于 C 语言驾轻就熟，能够快速准确编译出需要的可执行文件。

### 3.1.3 使用 GCC 编译 C++ 程序

在这里我们使用 C++ 版本的 `hello_world.cpp` 程序：

```
1 | #include <iostream>
2 | using namespace std;
3 | int main() {
4 |     cout << "Hello World!" << endl;
5 |     return 0;
6 | }
```

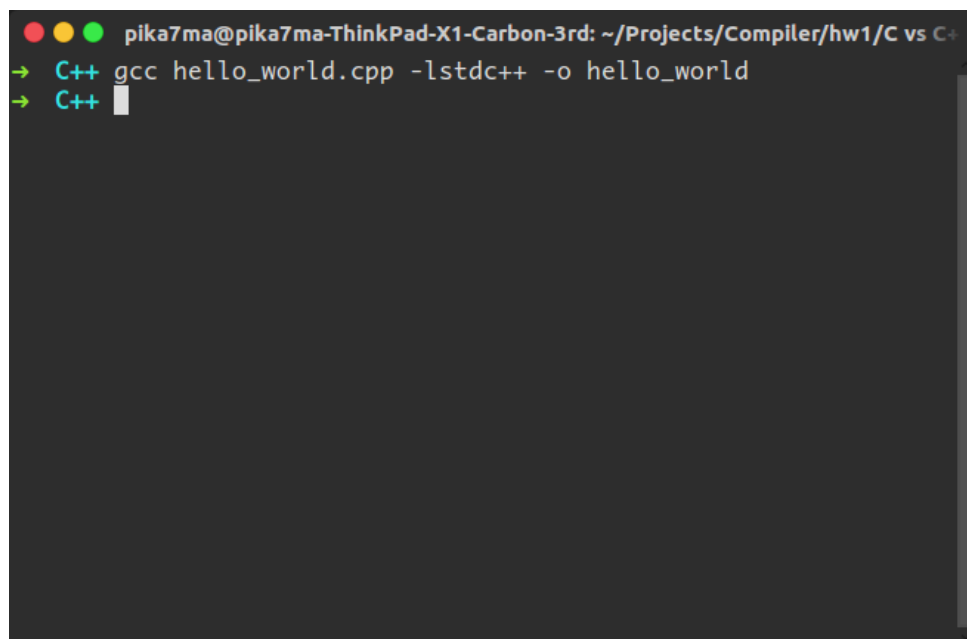
然而，在本应一帆风顺的编译过程中，GCC 报错：



```
pika7ma@pika7ma-ThinkPad-X1-Carbon-3rd: ~/Projects/Compiler/hw1/C vs C++
→ C++ gcc hello_world.cpp -o hello_world
/tmp/cc0sG0yk.o: In function `main':
hello_world.cpp:(.text+0xa): undefined reference to `std::cout'
hello_world.cpp:(.text+0xf): undefined reference to `std::basic_ostream<char, std::char_traits<char> >& std::operator<< (<std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, char const*)'
hello_world.cpp:(.text+0x14): undefined reference to `std::basic_ostream<char, std::char_traits<char> >& std::endl<char, std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&)'
hello_world.cpp:(.text+0x1c): undefined reference to `std::ostream::operator<< (std::ostream& (*)(std::ostream&))'
/tmp/cc0sG0yk.o: In function `__static_initialization_and_destruction_0(int, int)':
hello_world.cpp:(.text+0x4a): undefined reference to `std::ios_base::Init::Init()'
hello_world.cpp:(.text+0x59): undefined reference to `std::ios_base::Init::~Init()'
collect2: error: ld returned 1 exit status
```

图 3-3 使用 GCC 编译 C++

这主要是因为 GCC 并不会为程序自动链接 C++ 标准库 (Standard C++ Library), 我们需要手动加上 `-lstdc++` 参数才能正确链接:



```
pika7ma@pika7ma-ThinkPad-X1-Carbon-3rd: ~/Projects/Compiler/hw1/C vs C++
→ C++ gcc hello_world.cpp -lstdc++ -o hello_world
→ C++
```

图 3-4 使用带参 GCC 编译 C++

可以看出, GCC 默认还是作为 C 语言编译器存在的。

## 3.2 G++ 编译探究

### 3.2.1 G++ 简介

G++ 是作为 GCC 的 C++ 前端存在的。两者最大的不同在于：G++ 将会把所有 C 语言程序作为 C++ 程序来编译。

### 3.2.2 G++ 编译过程

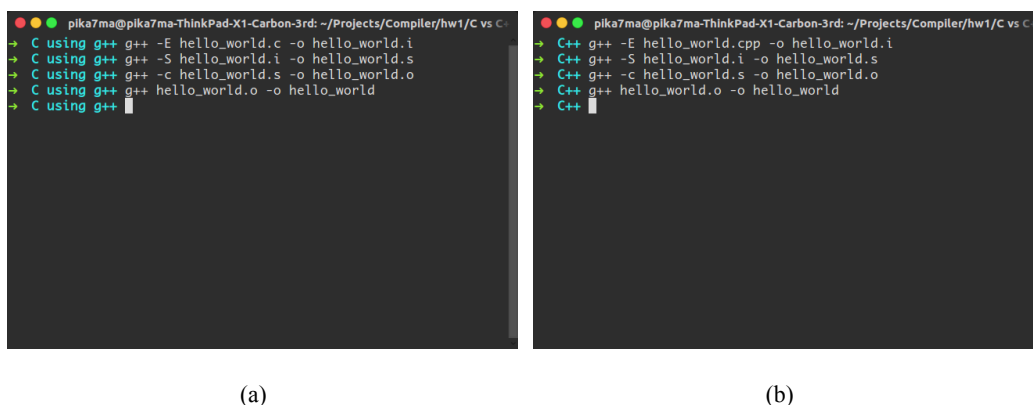


图 3-5 使用 G++ 编译 C 和 C++

可以看出，使用 C++ 后端的 G++ 可以轻松应对 C 和 C++ 程序。

### 3.2.3 C 和 C++ 编译差异

通过对比，虽然实现功能相同，但 C++ 版本的 Hello World! 程序不管是预处理生成文件还是汇编文件都显著长于 C 版本。

### 3.2.4 GCC 和 G++ 下 C 编译文件差异

在预处理文件中，两编译器生成的文件就已经出现了不同，例如其中 G++ 版本中多出了如下语句：

```
1 | # 29 "/usr/include/stdio.h" 3 4
2 | extern "C" {
```

而在汇编文件中，两者完全相同：

```
1 | .file "hello_world.c"
2 | .section .rodata
3 | .LC0:
4 | .string "Hello World!"
5 | .text
```



```
6      .globl  main
7      .type   main, @function
8  main:
9      .LFBO:
10     .cfi_startproc
11     pushq   %rbp
12     .cfi_def_cfa_offset 16
13     .cfi_offset 6, -16
14     movq    %rsp, %rbp
15     .cfi_def_cfa_register 6
16     movl    $.LC0, %edi
17     call    puts
18     movl    $0, %eax
19     popq    %rbp
20     .cfi_def_cfa 7, 8
21     ret
22     .cfi_endproc
23  .LFE0:
24     .size   main, .-main
25     .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.4) 5.4.0
26           20160609"
27     .section        .note.GNU-stack,"",@progbits
```

由此我们可以推断：尽管对 C 来说，使用不同编译器生成的预处理文件有所不同，但其编译器结果是殊途同归的，这也进一步体现了编译器的强大之处。

## 第四章 同一功能的不同实现

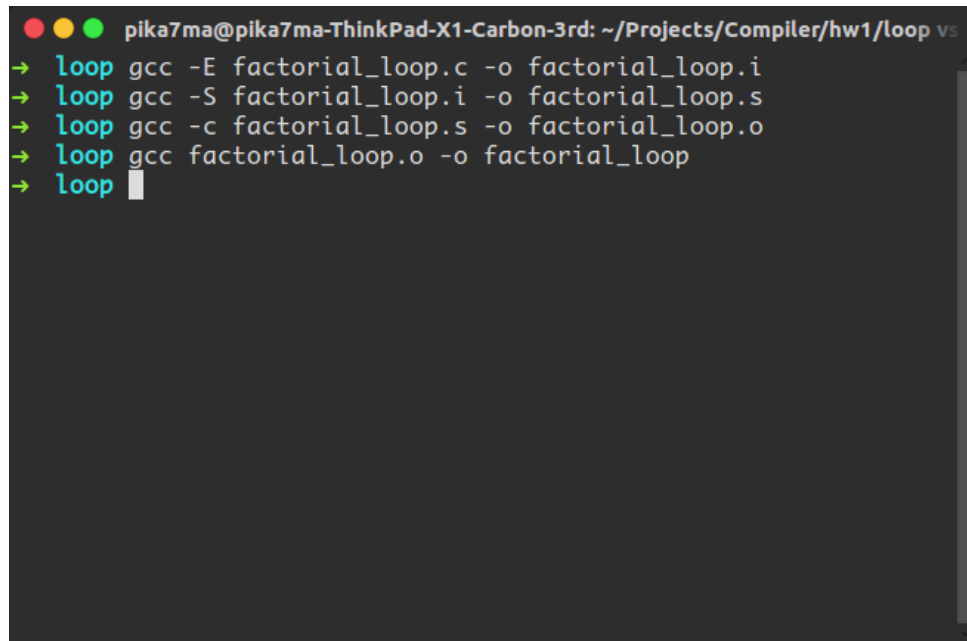
在某些特定问题中，递归和循环能够实现同一功能，如阶乘。本章针对两者差异进行了探究。

### 4.1 阶乘的循环实现

以下是典型的循环阶乘实现方法：

```
1  #include <stdio.h>
2  int main() {
3      int i, n, f;
4      scanf("%d", &n);
5      i = 2;
6      f = 1;
7      while (i <= n) {
8          f *= i;
9          i += 1;
10     }
11     printf("%d\n", f);
12     return 0;
13 }
```

通过简单的编译过程，我们得到了其可执行文件：



```
pika7ma@pika7ma-ThinkPad-X1-Carbon-3rd: ~/Projects/Compiler/hw1/loop vs
→ loop gcc -E factorial_loop.c -o factorial_loop.i
→ loop gcc -S factorial_loop.i -o factorial_loop.s
→ loop gcc -c factorial_loop.s -o factorial_loop.o
→ loop gcc factorial_loop.o -o factorial_loop
→ loop
```

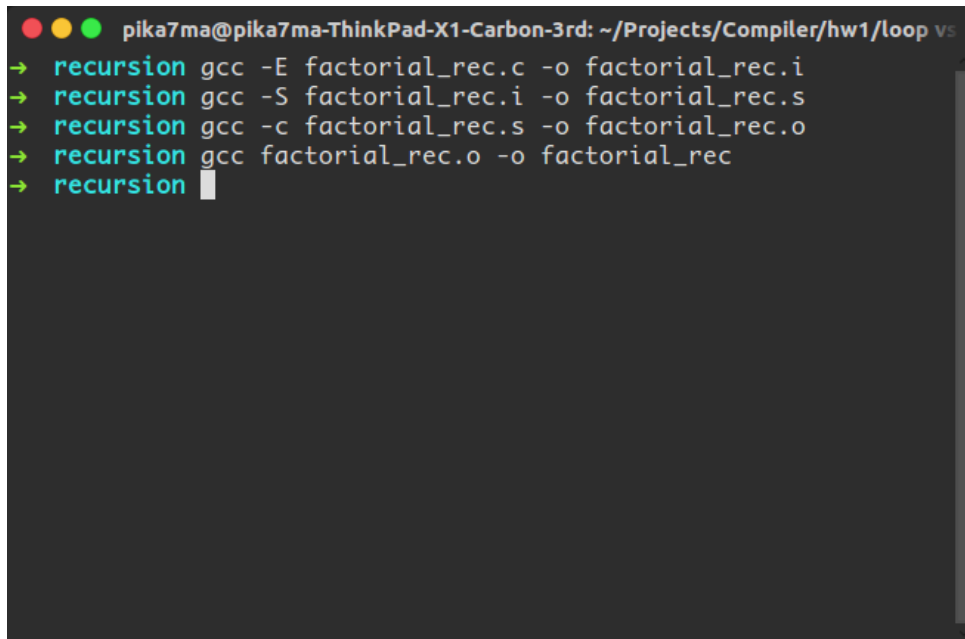
图 4-1 循环阶乘实现

## 4.2 阶乘的递归实现

我们也可以通过递归函数实现阶乘：

```
1 | #include <stdio.h>
2 | int fact(int i) {
3 |     if (i == 1) {
4 |         return 1;
5 |     }
6 |     return i * fact(i - 1);
7 | }
8 | int main() {
9 |     int n, f;
10 |    scanf("%d", &n);
11 |    f = fact(n);
12 |    printf("%d\n", f);
13 | }
```

通过简单的编译过程，我们得到了其可执行文件：



```
pika7ma@pika7ma-ThinkPad-X1-Carbon-3rd: ~/Projects/Compiler/hw1/loop vs
→ recursion gcc -E factorial_rec.c -o factorial_rec.i
→ recursion gcc -S factorial_rec.i -o factorial_rec.s
→ recursion gcc -c factorial_rec.s -o factorial_rec.o
→ recursion gcc factorial_rec.o -o factorial_rec
→ recursion
```

图 4-2 递归阶乘实现

## 4.3 差异对比

### 4.3.1 预处理文件

预处理文件中都被加入了大量的替换语句，由于都只包入了 `iostream` 包，两种实现除了真正代码部分外其他的预处理文件都完全一样。

### 4.3.2 汇编文件

尽管预处理文件极其相似，其汇编部分则截然不同，可以看出循环实现版本的长度要短于递归实现。其中循环实现中可以看到这样的实现：

```
1 | .L3:
2 |     movl    -12(%rbp), %eax
3 |     imull   -16(%rbp), %eax
4 |     movl    %eax, -12(%rbp)
5 |     addl    $1, -16(%rbp)
```

而在递归实现中的主体部分则有这样的代码：

```
1 | .LFB0:
2 |     .cfi_startproc
3 |     pushq   %rbp
4 |     .cfi_def_cfa_offset 16
5 |     .cfi_offset 6, -16
6 |     movq    %rsp, %rbp
```

```
7      .cfi_def_cfa_register 6
8      subq    $16, %rsp
9      movl    %edi, -4(%rbp)
10     cmpl    $1, -4(%rbp)
11     jne     .L2
12     movl    $1, %eax
13     jmp     .L3
14 .L2:
15     movl    -4(%rbp), %eax
16     subl    $1, %eax
17     movl    %eax, %edi
18     call    fact
19     imull   -4(%rbp), %eax
20 .L3:
21     leave
22     .cfi_def_cfa 7, 8
23     ret
24     .cfi_endproc
25 .LFE0:
26     .size   fact, .-fact
27     .section      .rodata
```

### 4.3.3 差异分析

造成二者差异的主要原因在于其不同的编程逻辑。通过此例我们看出预处理过程并不会大幅度改变内部代码，而编译器所做的翻译工作也体现除了它的强大。

## 第五章 编程风格的影响

### 5.1 环境变量

对于比较流行的不同编程风格，在这里选择了比较直观并被大多数人所熟知的差异。

编程风格 A:

```
1 // 使用换行大括号、单行注释、tab 键缩进
2 int main()
3 {
4     return 0;
5 }
```

编程风格 B:

```
1 /* 使用行尾大括号、多行注释、space 键缩进 */
2 int main() {
3     return 0;
4 }
```

### 5.2 结果差异

两段简单的代码都通过了层层翻译成为了可执行文件:



(a)

(b)

图 5-1 编译两种编程风格程序

随后我们发现，在预处理文件中，除了增加了部分头部修饰外，并没有对核心代码进行修改，有趣的是，在这个阶段，所有的注释都被去掉了。而汇编文件中，两者除了 .file 中名字不同外，其他完全相同（下方代码中的 \* 代表不同的名字）:

```
1      .file      "style_*.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     $0, %eax
14     popq     %rbp
15     .cfi_def_cfa 7, 8
16     ret
17     .cfi_endproc
18 .LFE0:
19     .size     main, .-main
20     .ident    "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.4) 5.4.0
21             20160609"
22     .section   .note.GNU-stack,"",@progbits
```

由此我们可以看出：在预处理阶段，预处理器会删除所有的注释，而在编译阶段，编译器会忽略空格、`tab` 等分隔符所带来的影响，大括号的位置也不会影响最终结果。程序员间的编程风格之争可以告一段落了！

## 第六章 编译器优化参数

### 6.1 编译器优化参数介绍

在 GCC 编译器中为我们提供了多种编译优化参数，不同的优化参数会影响编译时间和程序运行效率。

#### 6.1.1 不优化

使用参数`-O0`，对文件直接进行编译不做任何处理。

#### 6.1.2 一级优化

使用参数`-O`或`-O1`，目标是在不影响编译速度的前提下，尽量采用一些优化算法降低代码大小和执行速度。

#### 6.1.3 二级优化

使用参数`-O2`，会牺牲一部分编译速度，这里会调用几乎所有的目标配置支持的优化操作，来提高代码的运行效率。

#### 6.1.4 三级优化

使用参数`-O3`，使用众多向量化（`vectorization`）的方法，提高运行效率。

#### 6.1.5 空间优化

使用参数`-Os`，会尽可能压缩目标文件的大小。

### 6.2 不同优化级别性能对比

#### 6.2.1 运行速度比较

在这里我们仅对比`-O0`、`-O1`、`-O2`、`-O3`之间的运行速度差异。

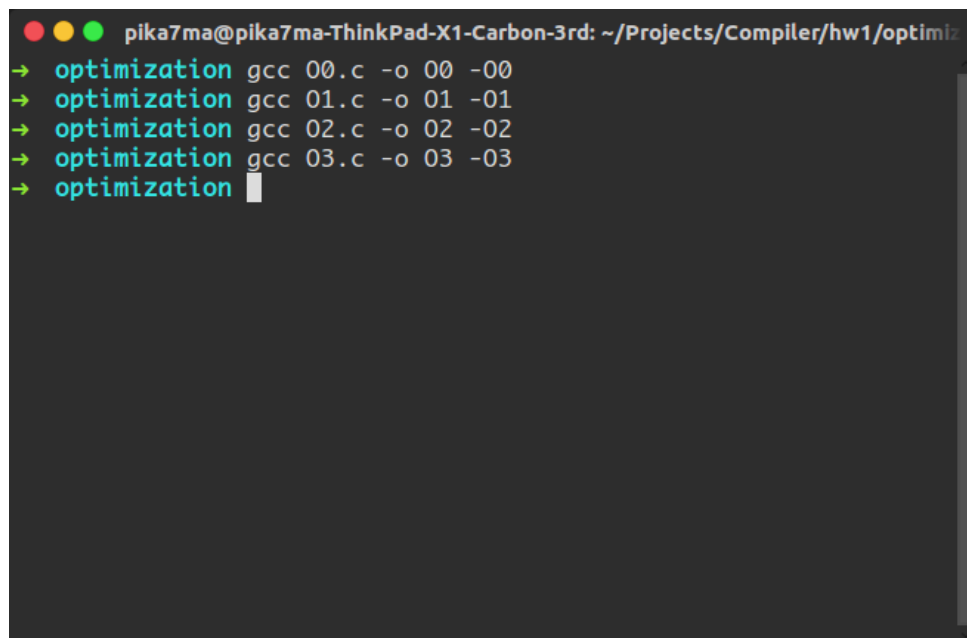
在这里我们使用经典的斐波那契数列函数进行多次重复，以放大优化效果：

```
1 | #include <stdio.h>
2 | #define N 20
3 | #define TIMES 10000000
4 | int main() {
5 |     int time = 0;
6 |     while (time < TIMES) {
```



```
7      int a[N];
8      int i;
9      i = 0;
10     while (i < N) {
11         a[i] = 0;
12         i += 1;
13     }
14     a[0] = 1;
15     a[1] = 1;
16     i = 2;
17     while (i < N) {
18         a[i] = a[i - 1] + a[i - 2];
19         i += 1;
20     }
21     time += 1;
22 }
23 return 0;
24 }
```

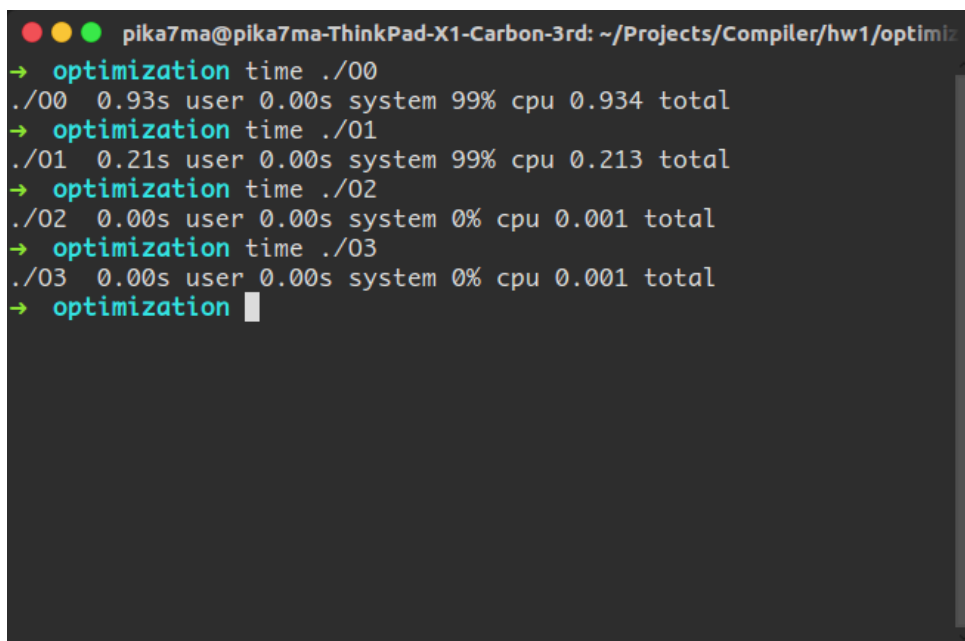
首先我们需要利用不同的优化参数对编译过程进行控制：



```
pika7ma@pika7ma-ThinkPad-X1-Carbon-3rd: ~/Projects/Compiler/hw1/optimiz
→ optimization gcc 00.c -o 00 -O0
→ optimization gcc 01.c -o 01 -O1
→ optimization gcc 02.c -o 02 -O2
→ optimization gcc 03.c -o 03 -O3
→ optimization
```

图 6-1 有优化参数的编译过程

接下来我们需要测试生成可执行文件的运行时间：



```
pika7ma@pika7ma-ThinkPad-X1-Carbon-3rd: ~/Projects/Compiler/hw1/optimiz
→ optimization time ./00
./00 0.93s user 0.00s system 99% cpu 0.934 total
→ optimization time ./01
./01 0.21s user 0.00s system 99% cpu 0.213 total
→ optimization time ./02
./02 0.00s user 0.00s system 0% cpu 0.001 total
→ optimization time ./03
./03 0.00s user 0.00s system 0% cpu 0.001 total
→ optimization
```

图 6-2 优化后运行时间对比

我们可以明显看出，使用了优化参数后，文件的运行速度大大上升，资源消耗也大大下降。

## 6.2.2 文件大小对比

这里我们使用`-Os`进行压缩编译，生成压缩后的可执行文件`Os`，进行文件大小对比：

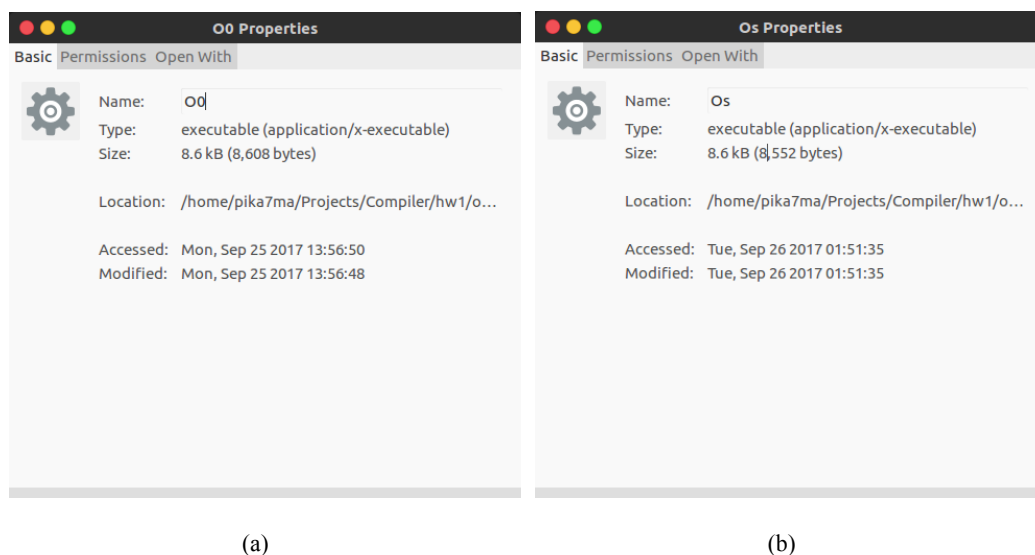


图 6-3 不同编译参数文件大小对比

可以看出，使用压缩优化参数后文件大小有一定的下降。

## 第七章 警告和调试开关

### 7.1 警告开关

在 GCC 中为我们提供了很多对编译器警告的调整参数。如 `-w` 为关闭警告，`-Wall` 为开启全部警告。

使用不标准实现代码作为示例：

```
1 | #include <stdio.h>
2 | void main() { // non-standard implementation: use "void"
   |     instead of "int"
3 |     printf("Hello World!\n");
4 | }
```

编译器警告程度差异如下：



(a)

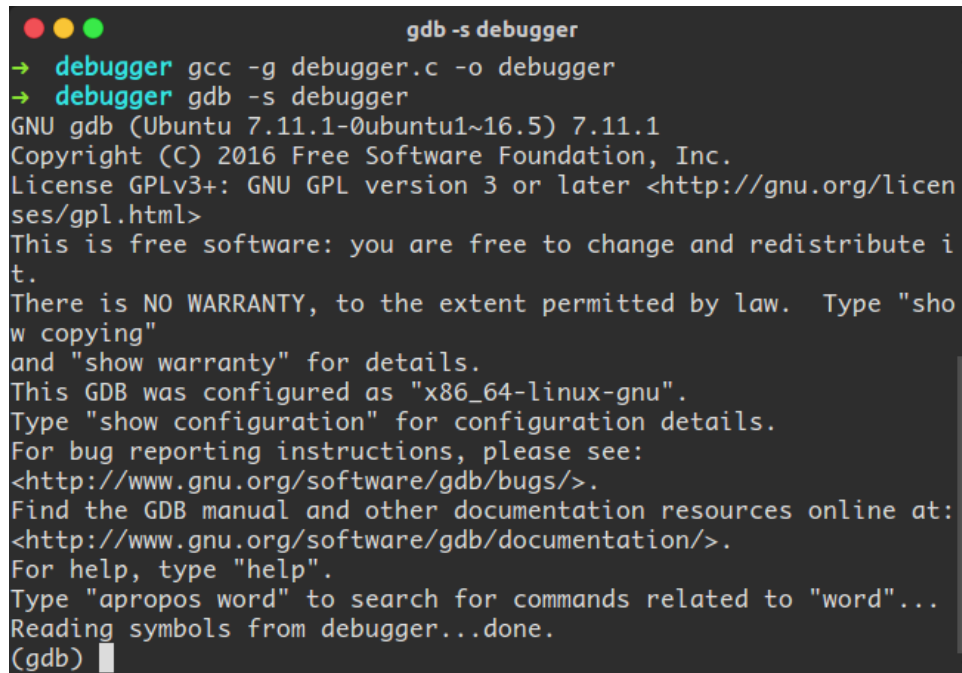
(b)

图 7-1 不同警告级别

在平时编程时最好还是注意一下警告，以免错过某些重要提示信息。

### 7.2 调试开关

为了方便我们调试（debug）程序，GCC 提供了方便的调试参数 `-g`：

A terminal window with a dark background and light-colored text. The title bar shows three colored circles (red, yellow, green) and the text 'gdb -s debugger'. The terminal content shows the execution of 'gcc -g debugger.c -o debugger' and 'gdb -s debugger'. It then displays the GNU GDB version (7.11.1), copyright information (2016 Free Software Foundation, Inc.), license (GPLv3+), and various help messages. The prompt '(gdb)' is visible at the bottom.

```
gdb -s debugger
→ debugger gcc -g debugger.c -o debugger
→ debugger gdb -s debugger
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show
w copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from debugger...done.
(gdb)
```

图 7-2 调试开关

配合 GDB（The GNU Project Debugger）就可以方便地对程序中的运行过程进行调试了。

## 第八章 自动并行化

### 8.1 GCC 与 OpenMP

OpenMP 框架是使用 C、C++ 和 Fortran 进行并发编程的一种强大方法。GCC 自 V4.2 开始支持 OpenMP 2.5 标准，而自 GCC 4.4 开始则支持最新的 OpenMP 3 标准。

为了使程序并行化，我们需要使用 `#pragma omp parallel` 语句来调用 OpenMP 编译指示。为了能够使用这项优化，我们需要在编译阶段指定 `-fopenmp` 选项。在编译期间，GCC 会根据硬件和操作系统配置在运行时生成代码，创建尽可能多的线程。每个线程的起始例程为代码块中位于指令之后的代码。这种行为就是自动并行化，而 OpenMP 本质上由一组功能强大的编译指示组成的。

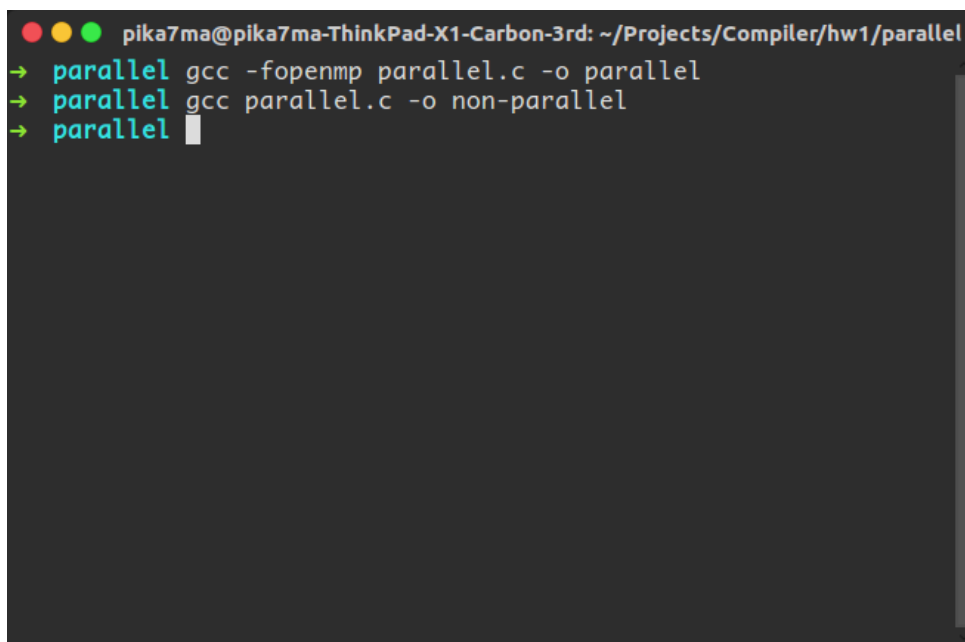
### 8.2 自动并行化实践

在这里我们通过一段简短的可被并行化的代码来进行实验：

```
1  #include <stdio.h>
2  #define NUM 10
3  int main() {
4      int a[NUM];
5      int b[NUM];
6      int c[NUM];
7      #pragma omp parallel for
8      for (int i = 0; i < NUM; ++i) {
9          a[i] = i;
10         b[i] = i;
11     }
12     #pragma omp parallel for
13     for (int i = 0; i < NUM; ++i) {
14         c[i] = a[i] + b[i];
15         printf("%d\n", c[i]);
16     }
17     return 0;
18 }
```

可以看到，我们指定了编译指示 `#pragma omp parallel for`。这句话可以使编译器将 `for` 循环工作负载划分到多个线程中，每个线程都可以在不同的核心上

运行，显著降低总的计算时间。在 GCC 中，我们通过不同的参数来进行自动并行化：



```
pika7ma@pika7ma-ThinkPad-X1-Carbon-3rd: ~/Projects/Compiler/hw1/parallel
→ parallel gcc -fopenmp parallel.c -o parallel
→ parallel gcc parallel.c -o non-parallel
→ parallel
```

图 8-1 自动并行化编译参数

随后执行并对比是否使用自动并行化的文件的输出：



```
(a) pika7ma@pika7ma-ThinkPad-X1-Carbon-3rd: ~/Projects/Compiler/hw1/parallel
→ parallel ./non-parallel
0
2
4
6
8
10
12
14
16
18
→ parallel

(b) pika7ma@pika7ma-ThinkPad-X1-Carbon-3rd: ~/Projects/Compiler/hw1/parallel
→ parallel ./parallel
0
2
4
16
18
6
8
10
12
14
→ parallel
```

(a)

(b)

图 8-2 自动并行化执行结果

对比可以看出，并行化的输出被打乱了，这实际上就是并行化的一个过程。编译器会将 `for` 循环内部拆分成互为独立的众多部分，划分到不同线程，这样做带来的结果就是不同线程的执行顺序由操作系统而非程序员决定，虽然有些不符合直觉，但这样做确实可以节约大量时间并尽可能地利用资源，事实上，在计算机中，以空间换时间的做法比比皆是。为了突出这一差别，我们将循环次数加大并取消输出来更加方便地观察时间差别：

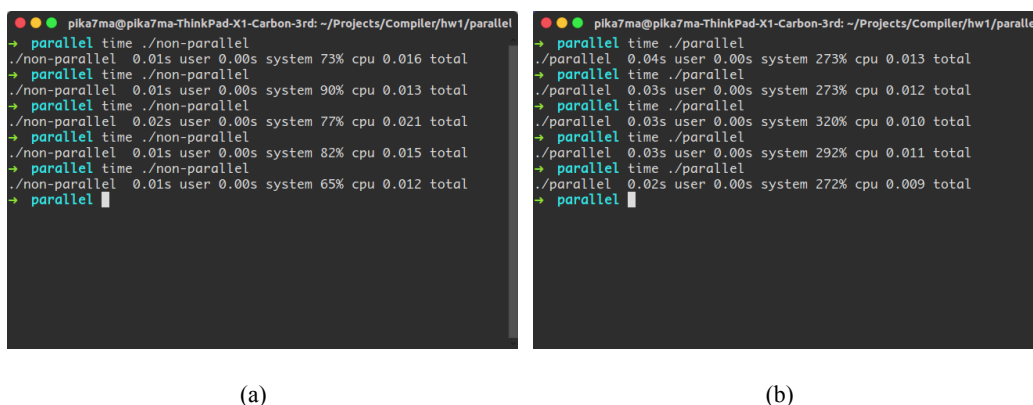


图 8-3 自动并行化性能分析

我们可以看出运行时间的明显差别，总结如下：

编译属性	平均 CPU 利用率	平均时间消耗 s
无自动并行化	77.4%	0.015 4
自动并行化	286.0%	0.011 0

可以看出，自动并行化后程序的平均时间消耗明显降低，降幅达 28%，而平均 CPU 利用率则增长了 369.5%。由于实验需求，在这里特别使用了 4 核心 CPU 来进行实验，而对于硬件，利用率越高则说明程序越能充分利用资源，也意味着优化得更好。

## 第九章 总结与回顾

### 9.1 全文总结

本文以编译器运作流程为研究背景，主要对 GCC 编译器各环节功能意义与 GCC 编译器修饰参数进行了研究。提升了作者自己对编译器流程的认识，并使得作者对原来的编程过程有了新的认识，以后会利用这次学到的经验和方法来对程序进行优化和改进。

### 9.2 后续工作展望

在研究过程中发现了很多作者自身值得改善的地方，仍有以下几点值得进一步学习：

1. 由于软件工程没有开设汇编语言程序设计的课程，对编译器生成的汇编语言的理解不够到位，只能读懂部分代码，以后需要勤加学习。
2. 对  $\text{\LaTeX}$  还不够熟练，在论文的排版编辑上消耗了大量时间，今后还需熟练  $\text{\LaTeX}$  用法。



## 参考文献

- [1] A. V. Aho, M. S. Lam, R. Sethi, et al. Compilers: Principles, techniques, and tools (2nd edition)[M]. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006
- [2] R. M. Stallman, G. DeveloperCommunity. Using the gnu compiler collection: A gnu manual for gcc version 4.3.3[M]. Paramount, CA: CreateSpace, 2009
- [3] D. Kusswurm. Modern x86 assembly language programming: 32-bit, 64-bit, sse, and avx[M]. Berkely, CA, USA: Apress, 2014
- [4] A. Ferrari. x86 assembly guide[EB/OL]. <http://www.cs.virginia.edu/~evans/cs216/guides/x86.html>, January 1, 2017
- [5] A. Sen. 通过 gcc 学习 openmp 框架 [EB/OL]. <https://www.ibm.com/developerworks/cn/aix/library/au-aix-openmp-framework/index.html>, October 29, 2012