



南开大学
Nankai University

南 开 大 学

计 算 机 学 院

编译系统原理实验报告

了解编译器及 LLVM IR 编程

朱子轩 2110853

专业：计算机科学与技术

刘修铭 2112492

专业：信息安全

年级：2021 级

指导教师：王刚

2023 年 9 月 7 日

摘要

本次实验由朱子轩与刘修铭合作完成，主要通过改进给定的阶乘 C 语言程序代码，在 Ubuntu 虚拟机上对编译器的各个阶段及其功能进行了学习与探索。除了完成实验手册的要求内容外，我们还通过加入注释、引用头文件、宏定义、死代码等部分对编译的预处理部分功能进行验证性探索，加深对于词法分析与语法分析的过程的认识；同时，借助 VS code 的插件，完成了将 CFG 的可视化，进而实现对中间代码生成的多阶段的分析；代码优化部分，我们通过对 O0 O1 O2 等不同优化等级的对比，以程序的运行时间为参考，验证优化效果。除此之外，我们还对 X86、ARM 和 LLVM 的汇编和链接结果分别予以实现，并加以探索。最后，我们编写了一段 LLVM IR 程序，加深对 LLVM IR 中间语言的了解。

关键字：编译器，优化对比，交叉编译，性能测试

目录

一、 分工	1
二、 预备工作及实验平台	2
三、 实验过程	4
(一) 预处理器	4
1. 预处理阶段功能	4
2. 实验验证	4
(二) 编译器	5
1. 词法分析	5
2. 语法分析	8
3. 语义分析和中间代码生成	9
4. 代码优化	13
5. 代码生成	14
(三) 汇编器	14
(四) 链接器加载器	16
(五) 执行	22
(六) LLVM IR 编程	22
四、 总结	23
(一) 第二节	23
(二) 第三节	24

一、 分工

二人均独立完成阶乘 C 语言部分的编译复现，并由刘修铭撰写实验报告，朱子轩予以修改补充。实验所用代码及生成文件均打包附后。

【待补充】LLVM IR 程序部分

NIKU

二、 预备工作及实验平台

按照实验指导，我们选用 Ubuntu 虚拟机配置了实验环境。实验平台如下：

设备名称	lxmliu2002-Ubuntu
系统名称	Ubuntu 22.04.3 LTS
操作系统类型	Linux 64 位
GNOME 版本	42.9
窗口系统	X11
虚拟化	VMware

表 1: 实验平台参数

同时，为了完成后续探索工作，我们修改了给定的阶乘程序。修改后的代码如下：

阶乘

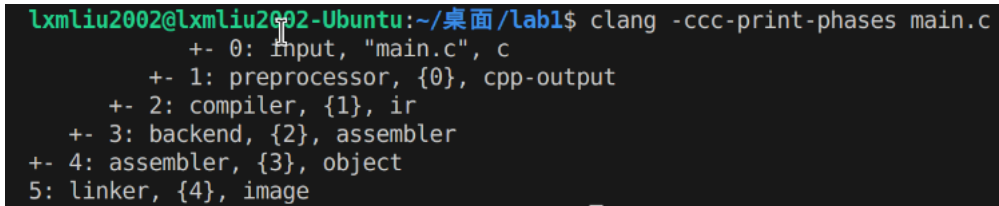
```
1 // 引用头文件
2 #include <stdio.h>
3 #include <time.h>
4 // 宏定义
5 #define MAX 10000
6 int main()
7 {
8     int i, n, f;
9     scanf("%d", &n);
10    i = 2;
11    f = 1;
12    clock_t start, end;
13    double time;
14    start = clock();
15    // 死代码
16    if (i < 1)
17    {
18        printf("i 小于 1\n");
19    }
20    while (i <= n)
21    {
22        f = f * i;
23        i = i + 1;
24    }
25    if (f > MAX)
26    {
27        printf("输出超限\n");
28    }
29    else
30    {
31        printf("%d\n", f);
32    }
```

```
33     end = clock();  
34     time = (double)(end - start) / CLOCKS_PER_SEC;  
35     printf("程序运行时间为: %f\n", time);  
36     return 0;  
37 }
```

NIKU

三、 实验过程

借助 `clang -ccc-print-phases main.c` 指令，我们得到编译的整个流程如图1所示：



```
lxmliu2002@lxmliu2002-Ubuntu:~/桌面/lab1$ clang -ccc-print-phases main.c
+- 0: input, "main.c", c
+- 1: preprocessor, {0}, cpp-output
+- 2: compiler, {1}, ir
+- 3: backend, {2}, assembler
+- 4: assembler, {3}, object
5: linker, {4}, image
```

图 1: 编译各阶段

(一) 预处理器

1. 预处理阶段功能

编译器的预处理阶段是编译过程的第一个阶段，它在实际的编译之前执行，主要完成以下任务：

- 处理和预处理以 `#` 符号开头指令。
- 展开宏定义，将宏名称替换为其定义的文本。
- 根据条件编译指令（如 `#ifdef`、`#ifndef`、`#if`、`#elif` 等）来控制哪些部分的代码应该包含在编译中。
- 移除注释，删除多余的空格、制表符和换行符，以简化源代码。
- 将指定的头文件内容插入到源代码中，从而形成一个整体的源代码文件，以便后续编译阶段使用。

预处理阶段的主要目标是将源代码准备好，以便后续的编译阶段能够进行词法分析、语法分析和生成中间代码。完成预处理后，生成的预处理文件通常会保存为 `.i` 或 `.ii` 扩展名的文件，然后传递给编译器的下一个阶段进行编译。

2. 实验验证

为了验证以上功能，我们将给定的阶乘代码予以改进，实现了对上述功能的验证。

首先，执行 `gcc main.c -E -o main.i` 指令，得到预处理后的 `main.i` 文件。然后观察生成的 `main.i` 文件并将其与 `main.c` 文件进行对比，可以发现：

- 代码的长度增加，由 37 行变为 1067 行，观察可知，引用的头文件已经插入其中。
- 一些无关内容被预处理删除。
- 宏名称已经被替换。

```

1027 # 4 "main.c" 2
1028
1029
1030
1031 # 6 "main.c"
1032 int main()
1033 {
1034     int i, n, f;
1035     scanf("%d", &n);
1036     i = 2;
1037     f = 1;
1038     clock_t start, end;
1039     double time;
1040     start = clock();
1041
1042     if (i < 1)
1043     {
1044         printf("i小于1\n");
1045     }
1046     while (i <= n)
1047     {
1048         f = f * i;
1049         i = i + 1;
1050     }
1051     if (f > 10000)
1052     {
1053         printf("输出超限\n");
1054     }
1055     else
1056     {
1057         printf("%d\n", f);
1058     }
1059     end = clock();
1060     time = (double)(end - start) /
1061 # 34 "main.c" 3 4
1062     ((__clock_t) 1000000);
1063 # 34 "main.c"
1064     printf("程序运行时间为: %f\n", time);
1065     return 0;
1066 }

```

图 2: 预处理结果

根据以上实验结果可知，预处理功能得以验证。

(二) 编译器

1. 词法分析

词法分析阶段的主要目标是将源代码分割成词法单元，以便后续的语法分析阶段能够理解代码的结构和语法。

借助 `clang -E -Xclang -dump-tokens main.c` 命令，我们得到 token 序列：

词法分析结果

1	int 'int '	[StartOfLine]	Loc=<main.c:6:1>
2	identifier 'main'	[LeadingSpace]	Loc=<main.c:6:5>
3	l_paren '('		Loc=<main.c:6:9>
4	r_paren ')'		Loc=<main.c:6:10>
5	l_brace '{'	[StartOfLine]	Loc=<main.c:7:1>
6	int 'int '	[StartOfLine] [LeadingSpace]	Loc=<main.c:8:5>
7	identifier 'i'	[LeadingSpace]	Loc=<main.c:8:9>
8	comma ', '		Loc=<main.c:8:10>
9	identifier 'n'	[LeadingSpace]	Loc=<main.c:8:12>
10	comma ', '		Loc=<main.c:8:13>
11	identifier 'f'	[LeadingSpace]	Loc=<main.c:8:15>
12	semi ';'		Loc=<main.c:8:16>
13	identifier 'scanf'	[StartOfLine] [LeadingSpace]	Loc=<main.c:9:5>
14	l_paren '('		Loc=<main.c:9:10>
15	string_literal '""%d""'		Loc=<main.c:9:11>
16	comma ', '		Loc=<main.c:9:15>
17	amp '&'	[LeadingSpace]	Loc=<main.c:9:17>


```

18 identifier 'n'          Loc=<main.c:9:18>
19 r_paren ')'          Loc=<main.c:9:19>
20 semi ';'            Loc=<main.c:9:20>
21 identifier 'i'      [StartOfLine] [LeadingSpace]  Loc=<main.c:10:5>
22 equal '='          [LeadingSpace] Loc=<main.c:10:7>
23 numeric_constant '2' [LeadingSpace] Loc=<main.c:10:9>
24 semi ';'            Loc=<main.c:10:10>
25 identifier 'f'      [StartOfLine] [LeadingSpace]  Loc=<main.c:11:5>
26 equal '='          [LeadingSpace] Loc=<main.c:11:7>
27 numeric_constant '1' [LeadingSpace] Loc=<main.c:11:9>
28 semi ';'            Loc=<main.c:11:10>
29 identifier 'clock_t' [StartOfLine] [LeadingSpace]  Loc=<main.c:12:5>
30 identifier 'start'   [LeadingSpace] Loc=<main.c:12:13>
31 comma ','           Loc=<main.c:12:18>
32 identifier 'end'     [LeadingSpace] Loc=<main.c:12:20>
33 semi ';'            Loc=<main.c:12:23>
34 double 'double'     [StartOfLine] [LeadingSpace]  Loc=<main.c:13:5>
35 identifier 'time'    [LeadingSpace] Loc=<main.c:13:12>
36 semi ';'            Loc=<main.c:13:16>
37 identifier 'start'   [StartOfLine] [LeadingSpace]  Loc=<main.c:14:5>
38 equal '='          [LeadingSpace] Loc=<main.c:14:11>
39 identifier 'clock'   [LeadingSpace] Loc=<main.c:14:13>
40 l_paren '('          Loc=<main.c:14:18>
41 r_paren ')'          Loc=<main.c:14:19>
42 semi ';'            Loc=<main.c:14:20>
43 if 'if'             [StartOfLine] [LeadingSpace]  Loc=<main.c:16:5>
44 l_paren '('          [LeadingSpace] Loc=<main.c:16:8>
45 identifier 'i'       Loc=<main.c:16:9>
46 less '<'            [LeadingSpace] Loc=<main.c:16:11>
47 numeric_constant '1' [LeadingSpace] Loc=<main.c:16:13>
48 r_paren ')'          Loc=<main.c:16:14>
49 l_brace '{'         [StartOfLine] [LeadingSpace]  Loc=<main.c:17:5>
50 identifier 'printf'   [StartOfLine] [LeadingSpace]  Loc=<main.c:18:9>
51 l_paren '('          Loc=<main.c:18:15>
52 string_literal '"i 小于1\n"' Loc=<main.c:18:16>
53 r_paren ')'          Loc=<main.c:18:28>
54 semi ';'            Loc=<main.c:18:29>
55 r_brace '}'         [StartOfLine] [LeadingSpace]  Loc=<main.c:19:5>
56 while 'while'        [StartOfLine] [LeadingSpace]  Loc=<main.c:20:5>
57 l_paren '('          [LeadingSpace] Loc=<main.c:20:11>
58 identifier 'i'       Loc=<main.c:20:12>
59 lessequal '<='      [LeadingSpace] Loc=<main.c:20:14>
60 identifier 'n'       [LeadingSpace] Loc=<main.c:20:17>
61 r_paren ')'          Loc=<main.c:20:18>
62 l_brace '{'         [StartOfLine] [LeadingSpace]  Loc=<main.c:21:5>
63 identifier 'f'       [StartOfLine] [LeadingSpace]  Loc=<main.c:22:9>
64 equal '='          [LeadingSpace] Loc=<main.c:22:11>
65 identifier 'f'       [LeadingSpace] Loc=<main.c:22:13>

```

```

66 star '*' [LeadingSpace] Loc=<main.c:22:15>
67 identifier 'i' [LeadingSpace] Loc=<main.c:22:17>
68 semi ';' Loc=<main.c:22:18>
69 identifier 'i' [StartOfLine] [LeadingSpace] Loc=<main.c:23:9>
70 equal '=' [LeadingSpace] Loc=<main.c:23:11>
71 identifier 'i' [LeadingSpace] Loc=<main.c:23:13>
72 plus '+' [LeadingSpace] Loc=<main.c:23:15>
73 numeric_constant '1' [LeadingSpace] Loc=<main.c:23:17>
74 semi ';' Loc=<main.c:23:18>
75 r_brace '}' [StartOfLine] [LeadingSpace] Loc=<main.c:24:5>
76 if 'if' [StartOfLine] [LeadingSpace] Loc=<main.c:25:5>
77 l_paren '(' [LeadingSpace] Loc=<main.c:25:8>
78 identifier 'f' Loc=<main.c:25:9>
79 greater '>' [LeadingSpace] Loc=<main.c:25:11>
80 numeric_constant '1000' [LeadingSpace] Loc=<main.c:25:13 <Spelling=
    main.c:5:13>>
81 r_paren ')' Loc=<main.c:25:16>
82 l_brace '{' [StartOfLine] [LeadingSpace] Loc=<main.c:26:5>
83 identifier 'printf' [StartOfLine] [LeadingSpace] Loc=<main.c:27:9>
84 l_paren '(' Loc=<main.c:27:15>
85 string_literal ""输出超限\n"" Loc=<main.c:27:16>
86 r_paren ')' Loc=<main.c:27:32>
87 semi ';' Loc=<main.c:27:33>
88 r_brace '}' [StartOfLine] [LeadingSpace] Loc=<main.c:28:5>
89 else 'else' [StartOfLine] [LeadingSpace] Loc=<main.c:29:5>
90 l_brace '{' [StartOfLine] [LeadingSpace] Loc=<main.c:30:5>
91 identifier 'printf' [StartOfLine] [LeadingSpace] Loc=<main.c:31:9>
92 l_paren '(' Loc=<main.c:31:15>
93 string_literal ""%d\n"" Loc=<main.c:31:16>
94 comma ',' Loc=<main.c:31:22>
95 identifier 'f' [LeadingSpace] Loc=<main.c:31:24>
96 r_paren ')' Loc=<main.c:31:25>
97 semi ';' Loc=<main.c:31:26>
98 r_brace '}' [StartOfLine] [LeadingSpace] Loc=<main.c:32:5>
99 identifier 'end' [StartOfLine] [LeadingSpace] Loc=<main.c:33:5>
100 equal '=' [LeadingSpace] Loc=<main.c:33:9>
101 identifier 'clock' [LeadingSpace] Loc=<main.c:33:11>
102 l_paren '(' Loc=<main.c:33:16>
103 r_paren ')' Loc=<main.c:33:17>
104 semi ';' Loc=<main.c:33:18>
105 identifier 'time' [StartOfLine] [LeadingSpace] Loc=<main.c:34:5>
106 equal '=' [LeadingSpace] Loc=<main.c:34:10>
107 l_paren '(' [LeadingSpace] Loc=<main.c:34:12>
108 double 'double' Loc=<main.c:34:13>
109 r_paren ')' Loc=<main.c:34:19>
110 l_paren '(' Loc=<main.c:34:20>
111 identifier 'end' Loc=<main.c:34:21>
112 minus '-' [LeadingSpace] Loc=<main.c:34:25>

```

```

113 identifier 'start'      [LeadingSpace] Loc=<main.c:34:27>
114 r_paren ')'            Loc=<main.c:34:32>
115 slash '/'             [LeadingSpace] Loc=<main.c:34:34>
116 l_paren '('           [LeadingSpace] Loc=<main.c:34:36 <Spelling=/usr/include/
      x86_64-linux-gnu/bits/time.h:34:25>>
117 l_paren '('           Loc=<main.c:34:36 <Spelling=/usr/include/x86_64-linux
      -gnu/bits/time.h:34:26>>
118 identifier '__clock_t' Loc=<main.c:34:36 <Spelling=/usr/include/
      x86_64-linux-gnu/bits/time.h:34:27>>
119 r_paren ')'            Loc=<main.c:34:36 <Spelling=/usr/include/x86_64-linux
      -gnu/bits/time.h:34:36>>
120 numeric_constant '1000000' [LeadingSpace] Loc=<main.c:34:36 <Spelling=/
      usr/include/x86_64-linux-gnu/bits/time.h:34:38>>
121 r_paren ')'            Loc=<main.c:34:36 <Spelling=/usr/include/x86_64-linux
      -gnu/bits/time.h:34:45>>
122 semi ';'              Loc=<main.c:34:50>
123 identifier 'printf'    [StartOfLine] [LeadingSpace] Loc=<main.c:35:5>
124 l_paren '('           Loc=<main.c:35:11>
125 string_literal '"程序运行时间为: %f\n"' Loc=<main.c:35:12>
126 comma ','             Loc=<main.c:35:42>
127 identifier 'time'      [LeadingSpace] Loc=<main.c:35:44>
128 r_paren ')'            Loc=<main.c:35:48>
129 semi ';'              Loc=<main.c:35:49>
130 return 'return'        [StartOfLine] [LeadingSpace] Loc=<main.c:36:5>
131 numeric_constant '0'   [LeadingSpace] Loc=<main.c:36:12>
132 semi ';'              Loc=<main.c:36:13>
133 r_brace '}'            [StartOfLine] Loc=<main.c:37:1>
134 eof ''                 Loc=<main.c:37:2>

```

观察可知，在词法分析阶段，源程序中的字符串被扫描并分解，识别成为一个个单词，并被写明其类型，便于后续的语法分析。

2. 语法分析

将词法分析生成的词法单元来构建抽象语法树 (Abstract Syntax Tree, 即 AST)。通过 clang -E -Xclang -ast-dump main.c 命令获得相应的 AST，如图3所示：

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <unistd.h>
5  #include <sys/types.h>
6  #include <sys/stat.h>
7  #include <fcntl.h>
8  #include <sys/mman.h>
9  #include <sys/time.h>
10 #include <sys/resource.h>
11 #include <sys/wait.h>
12 #include <sys/queue.h>
13 #include <sys/uio.h>
14 #include <sys/xattr.h>
15 #include <sys/zfs.h>
16 #include <sys/zfs_ioctl.h>
17 #include <sys/zfs_refint.h>
18 #include <sys/zfs_vfs.h>
19 #include <sys/zfs_zfs.h>
20 #include <sys/zfs_zfs_ioctl.h>
21 #include <sys/zfs_zfs_ioctl.h>
22 #include <sys/zfs_zfs_ioctl.h>
23 #include <sys/zfs_zfs_ioctl.h>
24 #include <sys/zfs_zfs_ioctl.h>
25 #include <sys/zfs_zfs_ioctl.h>
26 #include <sys/zfs_zfs_ioctl.h>
27 #include <sys/zfs_zfs_ioctl.h>
28 #include <sys/zfs_zfs_ioctl.h>
29 #include <sys/zfs_zfs_ioctl.h>
30 #include <sys/zfs_zfs_ioctl.h>
31 #include <sys/zfs_zfs_ioctl.h>
32 #include <sys/zfs_zfs_ioctl.h>
33 #include <sys/zfs_zfs_ioctl.h>
34 #include <sys/zfs_zfs_ioctl.h>
35 #include <sys/zfs_zfs_ioctl.h>
36 #include <sys/zfs_zfs_ioctl.h>
37 #include <sys/zfs_zfs_ioctl.h>
38 #include <sys/zfs_zfs_ioctl.h>
39 #include <sys/zfs_zfs_ioctl.h>
40 #include <sys/zfs_zfs_ioctl.h>

```

图 3: 语法分析结果

观察可知，他将语法结构加以识别，构成了一颗层次分明的语法树，对于后续的语义分析提供便利。

同时，为了检验其检验语法的功能，我们编写了一段错误程序，以此检测其功能。

错误代码

```

1 int main()
2 {
3     int i;
4     i *;
5 }

```

输入上述命令后，我们可以得到其语法分析的结果。

错误代码语法分析结果

```

1 test.c:4:8: error: expected expression
2     i *;
3     ^
4 1 error generated.

```

观察可以看到，在其语法树的前面，会有语法的错误提醒，从而验证语法分析检查代码语法的功能。

3. 语义分析和中间代码生成

语义分析是一个关键的步骤，负责理解代码的含义并检查其语法正确性以及语义是否合法，同时，深度分析并理解代码，确保其能正常执行。

利用 `clang -S -emit-llvm main.c` 命令生成 LLVM IR 中间代码。

LLVM IR 中间代码生成

```

1 ; ModuleID = 'main.c'
2 source_filename = "main.c"
3 target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"

```

```

4 target triple = "x86_64-pc-linux-gnu"
5
6 @.str = private unnamed_addr constant [3 x i8] c"%d\00", align 1
7 @.str.1 = private unnamed_addr constant [10 x i8] c"i\E5\B0\8F\E4\BA\8E1\0A
   \00", align 1
8 @.str.2 = private unnamed_addr constant [14 x i8] c"\E8\BE\93\E5\87\BA\E8\B6
   \85\E9\99\90\0A\00", align 1
9 @.str.3 = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
10 @.str.4 = private unnamed_addr constant [28 x i8] c"\E7\A8\8B\E5\BA\8F\E8\BF
   \90\E8\A1\8C\E6\97\B6\E9\97\B4\E4\B8\BA\EF\BC\9A%f\0A\00", align 1
11
12 ; Function Attrs: noline nounwind optnone uwtable
13 define dso_local i32 @main() #0 {
14     %1 = alloca i32, align 4
15     %2 = alloca i32, align 4
16     %3 = alloca i32, align 4
17     %4 = alloca i32, align 4
18     %5 = alloca i64, align 8
19     %6 = alloca i64, align 8
20     %7 = alloca double, align 8
21     store i32 0, i32* %1, align 4
22     %8 = call i32 @__isoc99_scanf(i8* noundef getelementptr inbounds
   ([3 x i8], [3 x i8]* @.str, i64 0, i64 0), i32* noundef %3)
23     store i32 2, i32* %2, align 4
24     store i32 1, i32* %4, align 4
25     %9 = call i64 @clock() #3
26     store i64 %9, i64* %5, align 8
27     %10 = load i32, i32* %2, align 4
28     %11 = icmp slt i32 %10, 1
29     br i1 %11, label %12, label %14
30
31 12:                                     ; preds = %0
32     %13 = call i32 @printf(i8* noundef getelementptr inbounds ([10 x
   i8], [10 x i8]* @.str.1, i64 0, i64 0))
33     br label %14
34
35 14:                                     ; preds = %12, %0
36     br label %15
37
38 15:                                     ; preds = %19, %14
39     %16 = load i32, i32* %2, align 4
40     %17 = load i32, i32* %3, align 4
41     %18 = icmp sle i32 %16, %17
42     br i1 %18, label %19, label %25
43
44 19:                                     ; preds = %15
45     %20 = load i32, i32* %4, align 4
46     %21 = load i32, i32* %2, align 4

```

```

47  %22 = mul nsw i32 %20, %21
48  store i32 %22, i32* %4, align 4
49  %23 = load i32, i32* %2, align 4
50  %24 = add nsw i32 %23, 1
51  store i32 %24, i32* %2, align 4
52  br label %15, !llvm.loop !6
53
54  25:                                     ; preds = %15
55  %26 = load i32, i32* %4, align 4
56  %27 = icmp sgt i32 %26, 10000
57  br i1 %27, label %28, label %30
58
59  28:                                     ; preds = %25
60  %29 = call i32 @__printf(i8* noundef getelementptr inbounds ([14 x
61  i8], [14 x i8]* @.str.2, i64 0, i64 0))
62  br label %33
63
64  30:                                     ; preds = %25
65  %31 = load i32, i32* %4, align 4
66  %32 = call i32 @__printf(i8* noundef getelementptr inbounds ([4 x
67  i8], [4 x i8]* @.str.3, i64 0, i64 0), i32 noundef %31)
68  br label %33
69
70  33:                                     ; preds = %30, %28
71  %34 = call i64 @clock() #3
72  store i64 %34, i64* %6, align 8
73  %35 = load i64, i64* %6, align 8
74  %36 = load i64, i64* %5, align 8
75  %37 = sub nsw i64 %35, %36
76  %38 = sitofp i64 %37 to double
77  %39 = fdiv double %38, 1.000000e+06
78  store double %39, double* %7, align 8
79  %40 = load double, double* %7, align 8
80  %41 = call i32 @__printf(i8* noundef getelementptr inbounds ([28 x
81  i8], [28 x i8]* @.str.4, i64 0, i64 0), double noundef %40)
82  ret i32 0
83
84  }
85
86  declare i32 @__isoc99_scanf(i8* noundef, ...) #1
87
88  ; Function Attrs: nounwind
89  declare i64 @clock() #2
90
91  declare i32 @printf(i8* noundef, ...) #1
92
93  attributes #0 = { noinline nounwind optnone uwtable "frame-pointer"="all" "
94  min-legal-vector-width"="0" "no-trapping-math"="true" "stack-protector-
95  buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+mmx

```

```

    ,+sse,+sse2,+x87" "tune-cpu"="generic" }
90 attributes #1 = { "frame-pointer"="all" "no-trapping-math"="true" "stack-
    protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8
    ,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
91 attributes #2 = { nounwind "frame-pointer"="all" "no-trapping-math"="true" "
    stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features
    "="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
92 attributes #3 = { nounwind }
93
94 !llvm.module.flags = !{!0, !1, !2, !3, !4}
95 !llvm.ident = !{!5}
96
97 !0 = !{i32 1, !"wchar_size", i32 4}
98 !1 = !{i32 7, !"PIC Level", i32 2}
99 !2 = !{i32 7, !"PIE Level", i32 2}
100 !3 = !{i32 7, !"uwtable", i32 1}
101 !4 = !{i32 7, !"frame-pointer", i32 2}
102 !5 = !{"Ubuntu clang version 14.0.0-1ubuntu1.1"}
103 !6 = distinct !{!6, !7}
104 !7 = !{"llvm.loop.mustprogress"}

```

利用 `gcc -fdump-tree-all-graph main.c` 命令可以得到中间代码生成的多阶段输出，此处借助 VS code 中的插件实现对 CFG 的可视化并加以分析。

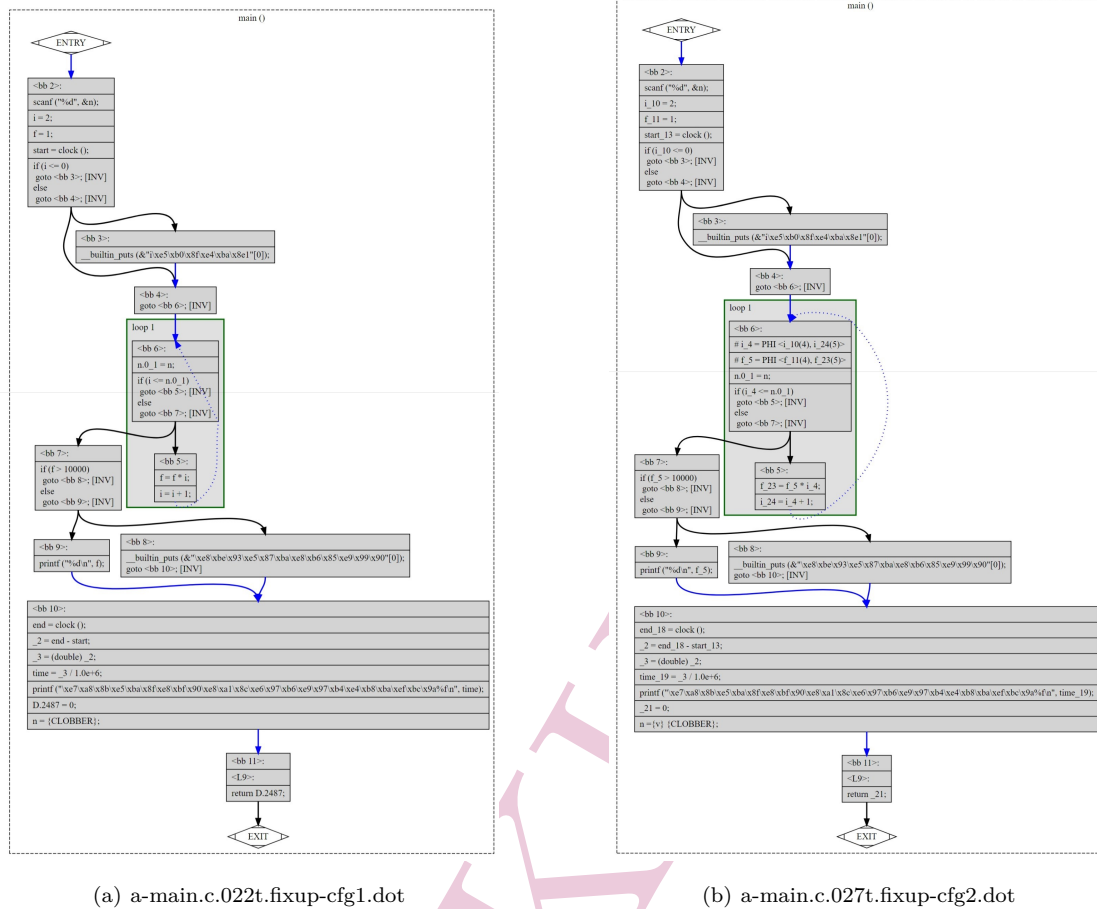


图 4: CFG 可视化

观察生成图片可以发现，控制流、变量名等发生变化，推测其可能为代码生成不同阶段中的不同值。

4. 代码优化

(1) 格式转换

借助 `llvm-as main.ll -o main.bc` 命令得到 LLVM IR 的二进制代码形式。

(2) 优化比较

借助 `opt -S -O0 main.bc -o main-O0.ll` 命令，实现对其 O0 优化。得到中间代码后继续处理，得到可执行文件，运行比较优化性能。

借助 `opt -S -O1 main.bc -o main-O1.ll` 命令，实现对其 O1 优化。得到中间代码后继续处理，得到可执行文件，运行比较优化性能。

借助 `opt -S -O2 main.bc -o main-O2.ll` 命令，实现对其 O2 优化。得到中间代码后继续处理，得到可执行文件，运行比较优化性能。

经过代码比较可知，O2 级优化与 O1 级优化代码相同，故而可知，O1 级优化已经优化完毕。运行三个可执行文件可知，O1 较 O0 有了一定程度优化，但 O2 级优化出现了不稳定情况。但是综合来看优化效果有限，推测可能是因为程序过于简单所导致。


```

lxmliu2002@lxmliu2002-Ubuntu:~/桌面/lab1$ ./main_00
10
输出超限
程序运行时间为：0.000032
lxmliu2002@lxmliu2002-Ubuntu:~/桌面/lab1$ ./main_01
10
输出超限
程序运行时间为：0.000016
lxmliu2002@lxmliu2002-Ubuntu:~/桌面/lab1$ ./main_02
10
输出超限
程序运行时间为：0.000109

```

图 5: 优化结果比较

5. 代码生成

以中间表示形式作为输入，将其映射到目标语言。

利用 `llc main.ll -o main.S` 命令生成目标代码；利用 `gcc main.i -S -o main.S` 命令生成 x86 格式目标代码；利用 `arm-linux-gnueabi-gcc main.i -S -o main.S` 命令生成 arm 格式目标代码。由于篇幅有限，代码文件内容在此不做展示，源文件已放于附件之中。

(三) 汇编器

汇编过程实际上把汇编语言程序代码翻译成目标机器指令的过程，其最终生成的是可重定位的机器代码。分别使用 `gcc main.S -c -o main.o` 命令、`arm-linux-gnueabi-gcc main.S -o main.o` 命令以及 `llc main.bc -filetype=obj -o main.o` 命令实现对 x86 格式、arm 格式以及 LLVM 格式文件进行汇编，得到 main.o 文件。

使用 `objdump -d main_gcc.o` 命令、`arm-linux-gnueabi-objdump -d main_arm.o` 命令对上述文件进行反汇编。由于内容所限，此处仅对 x86 反汇编进行展示。

x86 反汇编结果

```

1 main_gcc.o:      文件格式 elf64-x86-64
2
3 Disassembly of section .text:
4
5 0000000000000000 <main>:
6   0:   f3 0f 1e fa      endbr64
7   4:   55                push   %rbp
8   5:   48 89 e5          mov    %rsp,%rbp
9   8:   48 83 ec 30       sub    $0x30,%rsp
10  c:   64 48 8b 04 25 28 00 mov    %fs:0x28,%rax
11 13:   00 00
12 15:   48 89 45 f8       mov    %rax,-0x8(%rbp)
13 19:   31 c0             xor    %eax,%eax
14 1b:   48 8d 45 d4       lea    -0x2c(%rbp),%rax
15 1f:   48 89 c6          mov    %rax,%rsi
16 22:   48 8d 05 00 00 00 00 lea    0x0(%rip),%rax      # 29 <main+0x29>
17 29:   48 89 c7          mov    %rax,%rdi
18 2c:   b8 00 00 00 00    mov    $0x0,%eax
19 31:   e8 00 00 00 00    call   36 <main+0x36>

```

```

20 36: c7 45 d8 02 00 00 00 movl $0x2,-0x28(%rbp)
21 3d: c7 45 dc 01 00 00 00 movl $0x1,-0x24(%rbp)
22 44: e8 00 00 00 00 call 49 <main+0x49>
23 49: 48 89 45 e0 mov %rax,-0x20(%rbp)
24 4d: 83 7d d8 00 cmpl $0x0,-0x28(%rbp)
25 51: 7f 1f jg 72 <main+0x72>
26 53: 48 8d 05 00 00 00 00 lea 0x0(%rip),%rax # 5a <main+0x5a>
27 5a: 48 89 c7 mov %rax,%rdi
28 5d: e8 00 00 00 00 call 62 <main+0x62>
29 62: eb 0e jmp 72 <main+0x72>
30 64: 8b 45 dc mov -0x24(%rbp),%eax
31 67: 0f af 45 d8 imul -0x28(%rbp),%eax
32 6b: 89 45 dc mov %eax,-0x24(%rbp)
33 6e: 83 45 d8 01 addl $0x1,-0x28(%rbp)
34 72: 8b 45 d4 mov -0x2c(%rbp),%eax
35 75: 39 45 d8 cmp %eax,-0x28(%rbp)
36 78: 7e ea jle 64 <main+0x64>
37 7a: 81 7d dc 10 27 00 00 cmpl $0x2710,-0x24(%rbp)
38 81: 7e 11 jle 94 <main+0x94>
39 83: 48 8d 05 00 00 00 00 lea 0x0(%rip),%rax # 8a <main+0x8a>
40 8a: 48 89 c7 mov %rax,%rdi
41 8d: e8 00 00 00 00 call 92 <main+0x92>
42 92: eb 19 jmp ad <main+0xad>
43 94: 8b 45 dc mov -0x24(%rbp),%eax
44 97: 89 c6 mov %eax,%esi
45 99: 48 8d 05 00 00 00 00 lea 0x0(%rip),%rax # a0 <main+0xa0>
46 a0: 48 89 c7 mov %rax,%rdi
47 a3: b8 00 00 00 00 mov $0x0,%eax
48 a8: e8 00 00 00 00 call ad <main+0xad>
49 ad: e8 00 00 00 00 call b2 <main+0xb2>
50 b2: 48 89 45 e8 mov %rax,-0x18(%rbp)
51 b6: 48 8b 45 e8 mov -0x18(%rbp),%rax
52 ba: 48 2b 45 e0 sub -0x20(%rbp),%rax
53 be: 66 0f ef c0 pxor %xmm0,%xmm0
54 c2: f2 48 0f 2a c0 cvtsi2sd %rax,%xmm0
55 c7: f2 0f 10 0d 00 00 00 movsd 0x0(%rip),%xmm1 # cf <main+0xcf>
>
56 ce: 00
57 cf: f2 0f 5e c1 divsd %xmm1,%xmm0
58 d3: f2 0f 11 45 f0 movsd %xmm0,-0x10(%rbp)
59 d8: 48 8b 45 f0 mov -0x10(%rbp),%rax
60 dc: 66 48 0f 6e c0 movq %rax,%xmm0
61 e1: 48 8d 05 00 00 00 00 lea 0x0(%rip),%rax # e8 <main+0xe8>
62 e8: 48 89 c7 mov %rax,%rdi
63 eb: b8 01 00 00 00 mov $0x1,%eax
64 f0: e8 00 00 00 00 call f5 <main+0xf5>
65 f5: b8 00 00 00 00 mov $0x0,%eax
66 fa: 48 8b 55 f8 mov -0x8(%rbp),%rdx

```

```

67 fe: 64 48 2b 14 25 28 00 sub %fs:0x28,%rdx
68 105: 00 00
69 107: 74 05 je 10e <main+0x10e>
70 109: e8 00 00 00 00 call 10e <main+0x10e>
71 10e: c9 leave
72 10f: c3 ret

```

(四) 链接器加载器

由汇编程序生成的目标文件不能够直接执行。大型程序经常被分成多个部分进行编译，因此，可重定位的机器代码有必要和其他可重定位的目标文件以及库文件链接到一起，最终形成真正在机器上运行的代码。进而链接器对该机器代码进行执行生成可执行文件。

通过 `gcc main_gcc.o -o main_gcc` 命令、`gcc main_arm.o -o main_arm` 命令和 `clang main_llvm.o -o main_llvm` 命令生成不同中间代码版本的 `main` 可执行文件。

我们可以修改一下参数实现对链接结果的调整。

- `-o <output>`: 指定输出文件的名称，例如 `-o main` 将生成一个名为 `main` 的可执行文件。
- `-L<path>`: 指定链接器搜索库文件的路径，例如 `-L/usr/lib` 将在 `/usr/lib` 目录下搜索库文件。
- `-l<library>`: 指定链接器需要链接的库文件，例如 `-lm` 将链接数学库。
- `-I<path>`: 指定编译器搜索头文件的路径，例如 `-I/usr/include` 将在 `/usr/include` 目录下搜索头文件。
- `-static`: 静态链接，将所有的库文件都静态链接到可执行文件中，而不是动态链接。
- `-Wl,<option>`: 将 `<option>` 传递给链接器，例如 `-Wl,-rpath,/usr/local/lib` 将设置运行时库搜索路径。
- `-g`: 生成调试信息，方便调试程序。
- `-Wall`: 启用所有警告信息。
- `-O<level>`: 指定优化级别，例如 `-O2` 进行中级优化。
- `-std=<standard>`: 指定使用的 C 语言标准，例如 `-std=c11` 指定使用 C11 标准。

使用 `objdump -d main_gcc` 命令、`arm-linux-gnueabi-objdump -d main_arm` 命令对上述文件进行反汇编。由于内容所限，此处仅对 x86 反汇编进行展示。

x86 反汇编结果

```

1 main_gcc: 文件格式 elf64-x86-64
2
3
4 Disassembly of section .init:
5
6 0000000000001000 <__init>:
7 1000: f3 0f 1e fa endbr64
8 1004: 48 83 ec 08 sub $0x8,%rsp

```

```

9      1008:      48 8b 05 d9 2f 00 00      mov     0x2fd9(%rip),%rax      # 3
      fe8 <__gmon_start__@Base>
10     100f:      48 85 c0                        test    %rax,%rax
11     1012:      74 02                        je      1016 <__init+0x16>
12     1014:      ff d0                        call    *%rax
13     1016:      48 83 c4 08                    add     $0x8,%rsp
14     101a:      c3                        ret
15
16 Disassembly of section .plt:
17
18 0000000000001020 <.plt>:
19     1020:      ff 35 7a 2f 00 00      push    0x2f7a(%rip)      # 3fa0 <
      _GLOBAL_OFFSET_TABLE_+0x8>
20     1026:      f2 ff 25 7b 2f 00 00      bnd jmp *0x2f7b(%rip)      # 3fa8 <
      _GLOBAL_OFFSET_TABLE_+0x10>
21     102d:      0f 1f 00                    nopl    (%rax)
22     1030:      f3 0f 1e fa                endbr64
23     1034:      68 00 00 00 00 00      push    $0x0
24     1039:      f2 e9 e1 ff ff ff      bnd jmp 1020 <__init+0x20>
25     103f:      90                        nop
26     1040:      f3 0f 1e fa                endbr64
27     1044:      68 01 00 00 00 00      push    $0x1
28     1049:      f2 e9 d1 ff ff ff      bnd jmp 1020 <__init+0x20>
29     104f:      90                        nop
30     1050:      f3 0f 1e fa                endbr64
31     1054:      68 02 00 00 00 00      push    $0x2
32     1059:      f2 e9 c1 ff ff ff      bnd jmp 1020 <__init+0x20>
33     105f:      90                        nop
34     1060:      f3 0f 1e fa                endbr64
35     1064:      68 03 00 00 00 00      push    $0x3
36     1069:      f2 e9 b1 ff ff ff      bnd jmp 1020 <__init+0x20>
37     106f:      90                        nop
38     1070:      f3 0f 1e fa                endbr64
39     1074:      68 04 00 00 00 00      push    $0x4
40     1079:      f2 e9 a1 ff ff ff      bnd jmp 1020 <__init+0x20>
41     107f:      90                        nop
42
43 Disassembly of section .plt.got:
44
45 0000000000001080 <__cxa_finalize@plt>:
46     1080:      f3 0f 1e fa                endbr64
47     1084:      f2 ff 25 6d 2f 00 00      bnd jmp *0x2f6d(%rip)      # 3ff8 <
      __cxa_finalize@GLIBC_2.2.5>
48     108b:      0f 1f 44 00 00 00      nopl    0x0(%rax,%rax,1)
49
50 Disassembly of section .plt.sec:
51
52 0000000000001090 <puts@plt>:

```

```

53      1090:      f3 0f 1e fa      endbr64
54      1094:      f2 ff 25 15 2f 00 00      bnd jmp *0x2f15(%rip)      # 3fb0 <
      puts@GLIBC_2.2.5>
55      109b:      0f 1f 44 00 00      nopl 0x0(%rax,%rax,1)
56
57      00000000000010a0 <clock@plt>:
58      10a0:      f3 0f 1e fa      endbr64
59      10a4:      f2 ff 25 0d 2f 00 00      bnd jmp *0x2f0d(%rip)      # 3fb8 <
      clock@GLIBC_2.2.5>
60      10ab:      0f 1f 44 00 00      nopl 0x0(%rax,%rax,1)
61
62      00000000000010b0 <__stack_chk_fail@plt>:
63      10b0:      f3 0f 1e fa      endbr64
64      10b4:      f2 ff 25 05 2f 00 00      bnd jmp *0x2f05(%rip)      # 3fc0 <
      __stack_chk_fail@GLIBC_2.4>
65      10bb:      0f 1f 44 00 00      nopl 0x0(%rax,%rax,1)
66
67      00000000000010c0 <printf@plt>:
68      10c0:      f3 0f 1e fa      endbr64
69      10c4:      f2 ff 25 fd 2e 00 00      bnd jmp *0x2efd(%rip)      # 3fc8 <
      printf@GLIBC_2.2.5>
70      10cb:      0f 1f 44 00 00      nopl 0x0(%rax,%rax,1)
71
72      00000000000010d0 <__isoc99_scanf@plt>:
73      10d0:      f3 0f 1e fa      endbr64
74      10d4:      f2 ff 25 f5 2e 00 00      bnd jmp *0x2ef5(%rip)      # 3fd0 <
      __isoc99_scanf@GLIBC_2.7>
75      10db:      0f 1f 44 00 00      nopl 0x0(%rax,%rax,1)
76
77      Disassembly of section .text:
78
79      00000000000010e0 <_start>:
80      10e0:      f3 0f 1e fa      endbr64
81      10e4:      31 ed      xor %ebp,%ebp
82      10e6:      49 89 d1      mov %rdx,%r9
83      10e9:      5e      pop %rsi
84      10ea:      48 89 e2      mov %rsp,%rdx
85      10ed:      48 83 e4 f0      and $0xfffffffffffffff0,%rsp
86      10f1:      50      push %rax
87      10f2:      54      push %rsp
88      10f3:      45 31 c0      xor %r8d,%r8d
89      10f6:      31 c9      xor %ecx,%ecx
90      10f8:      48 8d 3d ca 00 00 00      lea 0xca(%rip),%rdi      # 11c9
      <main>
91      10ff:      ff 15 d3 2e 00 00      call *0x2ed3(%rip)      # 3fd8 <
      __libc_start_main@GLIBC_2.34>
92      1105:      f4      hlt
93      1106:      66 2e 0f 1f 84 00 00      cs nopw 0x0(%rax,%rax,1)

```

```

94      110d:      00 00 00
95
96      0000000000001110 <deregister_tm_clones>:
97      1110:      48 8d 3d f9 2e 00 00      lea     0x2ef9(%rip),%rdi      #
          4010 <__TMC_END__>
98      1117:      48 8d 05 f2 2e 00 00      lea     0x2ef2(%rip),%rax      #
          4010 <__TMC_END__>
99      111e:      48 39 f8                  cmp     %rdi,%rax
100     1121:      74 15                  je      1138 <deregister_tm_clones+0
          x28>
101     1123:      48 8b 05 b6 2e 00 00      mov     0x2eb6(%rip),%rax      # 3
          fe0 <__ITM_deregisterTMCloneTable@Base>
102     112a:      48 85 c0                  test    %rax,%rax
103     112d:      74 09                  je      1138 <deregister_tm_clones+0
          x28>
104     112f:      ff e0                  jmp     *%rax
105     1131:      0f 1f 80 00 00 00 00      nopl    0x0(%rax)
106     1138:      c3                  ret
107     1139:      0f 1f 80 00 00 00 00      nopl    0x0(%rax)
108
109     0000000000001140 <register_tm_clones>:
110     1140:      48 8d 3d c9 2e 00 00      lea     0x2ec9(%rip),%rdi      #
          4010 <__TMC_END__>
111     1147:      48 8d 35 c2 2e 00 00      lea     0x2ec2(%rip),%rsi      #
          4010 <__TMC_END__>
112     114e:      48 29 fe                  sub     %rdi,%rsi
113     1151:      48 89 f0                  mov     %rsi,%rax
114     1154:      48 c1 ee 3f              shr     $0x3f,%rsi
115     1158:      48 c1 f8 03              sar     $0x3,%rax
116     115c:      48 01 c6                  add     %rax,%rsi
117     115f:      48 d1 fe                  sar     %rsi
118     1162:      74 14                  je      1178 <register_tm_clones+0x38>
119     1164:      48 8b 05 85 2e 00 00      mov     0x2e85(%rip),%rax      # 3
          ff0 <__ITM_registerTMCloneTable@Base>
120     116b:      48 85 c0                  test    %rax,%rax
121     116e:      74 08                  je      1178 <register_tm_clones+0x38>
122     1170:      ff e0                  jmp     *%rax
123     1172:      66 0f 1f 44 00 00      nopw    0x0(%rax,%rax,1)
124     1178:      c3                  ret
125     1179:      0f 1f 80 00 00 00 00      nopl    0x0(%rax)
126
127     0000000000001180 <__do_global_dtors_aux>:
128     1180:      f3 0f 1e fa              endbr64
129     1184:      80 3d 85 2e 00 00 00      cmpb    $0x0,0x2e85(%rip)      #
          4010 <__TMC_END__>
130     118b:      75 2b                  jne     11b8 <__do_global_dtors_aux+0
          x38>
131     118d:      55                  push    %rbp

```

```

132 118e:      48 83 3d 62 2e 00 00    cmpq    $0x0,0x2e62(%rip)          # 3
      ff8 <__cxa_finalize@GLIBC_2.2.5>
133 1195:      00
134 1196:      48 89 e5                mov     %rsp,%rbp
135 1199:      74 0c                je      11a7 <__do_global_dtors_aux+0
      x27>
136 119b:      48 8b 3d 66 2e 00 00    mov     0x2e66(%rip),%rdi          #
      4008 <__dso_handle>
137 11a2:      e8 d9 fe ff ff        call    1080 <__cxa_finalize@plt>
138 11a7:      e8 64 ff ff ff        call    1110 <deregister_tm_clones>
139 11ac:      c6 05 5d 2e 00 00 01    movb    $0x1,0x2e5d(%rip)          #
      4010 <_TMC_END_>
140 11b3:      5d                pop     %rbp
141 11b4:      c3                ret
142 11b5:      0f 1f 00          nopl    (%rax)
143 11b8:      c3                ret
144 11b9:      0f 1f 80 00 00 00 00    nopl    0x0(%rax)
145
146 00000000000011c0 <frame_dummy>:
147 11c0:      f3 0f 1e fa          endbr64
148 11c4:      e9 77 ff ff ff        jmp     1140 <register_tm_clones>
149
150 00000000000011c9 <main>:
151 11c9:      f3 0f 1e fa          endbr64
152 11cd:      55                push    %rbp
153 11ce:      48 89 e5                mov     %rsp,%rbp
154 11d1:      48 83 ec 30          sub     $0x30,%rsp
155 11d5:      64 48 8b 04 25 28 00    mov     %fs:0x28,%rax
156 11dc:      00 00
157 11de:      48 89 45 f8          mov     %rax,-0x8(%rbp)
158 11e2:      31 c0                xor     %eax,%eax
159 11e4:      48 8d 45 d4          lea     -0x2c(%rbp),%rax
160 11e8:      48 89 c6                mov     %rax,%rsi
161 11eb:      48 8d 05 16 0e 00 00    lea     0xe16(%rip),%rax          # 2008
      <_IO_stdin_used+0x8>
162 11f2:      48 89 c7                mov     %rax,%rdi
163 11f5:      b8 00 00 00 00        mov     $0x0,%eax
164 11fa:      e8 d1 fe ff ff        call    10d0 <__isoc99_scanf@plt>
165 11ff:      c7 45 d8 02 00 00 00    movl    $0x2,-0x28(%rbp)
166 1206:      c7 45 dc 01 00 00 00    movl    $0x1,-0x24(%rbp)
167 120d:      e8 8e fe ff ff        call    10a0 <clock@plt>
168 1212:      48 89 45 e0          mov     %rax,-0x20(%rbp)
169 1216:      83 7d d8 00          cmpl    $0x0,-0x28(%rbp)
170 121a:      7f 1f                jg      123b <main+0x72>
171 121c:      48 8d 05 e8 0d 00 00    lea     0xde8(%rip),%rax          # 200b
      <_IO_stdin_used+0xb>
172 1223:      48 89 c7                mov     %rax,%rdi
173 1226:      e8 65 fe ff ff        call    1090 <puts@plt>

```

```

174 122b:      eb 0e      jmp     123b <main+0x72>
175 122d:      8b 45 dc      mov     -0x24(%rbp),%eax
176 1230:      0f af 45 d8     imul    -0x28(%rbp),%eax
177 1234:      89 45 dc      mov     %eax,-0x24(%rbp)
178 1237:      83 45 d8 01     addl    $0x1,-0x28(%rbp)
179 123b:      8b 45 d4      mov     -0x2c(%rbp),%eax
180 123e:      39 45 d8      cmp     %eax,-0x28(%rbp)
181 1241:      7e ea      jle     122d <main+0x64>
182 1243:      81 7d dc 10 27 00 00  cmpl    $0x2710,-0x24(%rbp)
183 124a:      7e 11      jle     125d <main+0x94>
184 124c:      48 8d 05 c1 0d 00 00  lea     0xdc1(%rip),%rax      # 2014
      <_IO_stdin_used+0x14>
185 1253:      48 89 c7      mov     %rax,%rdi
186 1256:      e8 35 fe ff ff     call    1090 <puts@plt>
187 125b:      eb 19      jmp     1276 <main+0xad>
188 125d:      8b 45 dc      mov     -0x24(%rbp),%eax
189 1260:      89 c6      mov     %eax,%esi
190 1262:      48 8d 05 b8 0d 00 00  lea     0xdb8(%rip),%rax      # 2021
      <_IO_stdin_used+0x21>
191 1269:      48 89 c7      mov     %rax,%rdi
192 126c:      b8 00 00 00 00     mov     $0x0,%eax
193 1271:      e8 4a fe ff ff     call    10c0 <printf@plt>
194 1276:      e8 25 fe ff ff     call    10a0 <clock@plt>
195 127b:      48 89 45 e8      mov     %rax,-0x18(%rbp)
196 127f:      48 8b 45 e8      mov     -0x18(%rbp),%rax
197 1283:      48 2b 45 e0      sub     -0x20(%rbp),%rax
198 1287:      66 0f ef c0      pxor    %xmm0,%xmm0
199 128b:      f2 48 0f 2a c0     cvtsi2sd %rax,%xmm0
200 1290:      f2 0f 10 0d b0 0d 00  movsd   0xdb0(%rip),%xmm1      #
      2048 <_IO_stdin_used+0x48>
201 1297:      00
202 1298:      f2 0f 5e c1      divsd   %xmm1,%xmm0
203 129c:      f2 0f 11 45 f0     movsd   %xmm0,-0x10(%rbp)
204 12a1:      48 8b 45 f0      mov     -0x10(%rbp),%rax
205 12a5:      66 48 0f 6e c0     movq    %rax,%xmm0
206 12aa:      48 8d 05 74 0d 00 00  lea     0xd74(%rip),%rax      # 2025
      <_IO_stdin_used+0x25>
207 12b1:      48 89 c7      mov     %rax,%rdi
208 12b4:      b8 01 00 00 00     mov     $0x1,%eax
209 12b9:      e8 02 fe ff ff     call    10c0 <printf@plt>
210 12be:      b8 00 00 00 00     mov     $0x0,%eax
211 12c3:      48 8b 55 f8      mov     -0x8(%rbp),%rdx
212 12c7:      64 48 2b 14 25 28 00  sub     %fs:0x28,%rdx
213 12ce:      00 00
214 12d0:      74 05      je      12d7 <main+0x10e>
215 12d2:      e8 d9 fd ff ff     call    10b0 <__stack_chk_fail@plt>
216 12d7:      c9      leave
217 12d8:      c3      ret

```



```
218 |
219 | Disassembly of section .fini:
220 |
221 | 00000000000012dc <_fini>:
222 |    12dc:    f3 0f 1e fa    endbr64
223 |    12e0:    48 83 ec 08    sub    $0x8,%rsp
224 |    12e4:    48 83 c4 08    add    $0x8,%rsp
225 |    12e8:    c3            ret
```

将其与上面的反汇编代码进行比较，可以发现其他库文件的反汇编代码，从而验证了链接器的功能。

(五) 执行

在终端中输入./main 即可执行该程序。

```
lxmliu2002@lxmliu2002-Ubuntu:~/桌面/lab1$ ./main
5
120
程序运行时间为：0.000039
```

图 6: 运行结果

(六) LLVM IR 编程

四、 总结

如图7所示



图 7: Caption

表

N/n\Algo	naive-conv	naive-pool	omp-conv	omp-pool
64/2	0.0167	0.01255	0.04142	0.03799
64/4	0.03599	0.0394	0.0458	0.0421

表 2: 性能测试结果 (4 线程)(单位:ms)

带单元格表格

<i>Cost</i>		To				
		<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>
From	<i>B</i>	7	0	1	3	8
	<i>C</i>	8	1	0	2	7
	<i>D</i>	8	3	2	0	5

表 3: 结点 C 距离向量表 (无毒性逆转)

(一) 第二节

伪代码

Algorithm 1 初始化 obj 文件信息——对应 MeshSimplify 类中 readfile 函数,Face 类 calMatrix 函数

Input: obj 文件, 顶点、边、面列表

Output: 是否读取成功

```

1: function CALMATRIX(Face)
2:   normal  $\leftarrow e1 \times e2$ 
3:   normal  $\leftarrow normal / normal.length$ 
4:   temp[]  $\leftarrow normal.x, normal.y, normal.z, normal \cdot Face.v1$ 
5:   Matrix[i][j] = temp[i] * temp[j]
6:   return Matrix
7: end function
8: 根据 obj 的 v 和 f 区分点面信息, 读取并加入列表
9: scale  $\leftarrow$  记录点坐标中距离原点最远的分量, 以便后续 OpenGL 进行显示
10: ori  $\leftarrow$  记录中心点, 便于 OpenGL 显示在中心位置, 避免有的 obj 偏移原点较多
11: 根据三角面片信息, 计算一个面的三条边

```

- 12: 计算每个面的矩阵 $\leftarrow calMatrix$
 13: 将每个面的矩阵加到各点, 由点维护
 14: **return** True

代码

逐列访问平凡算法

```

1  void ord()
2  {
3      double head, tail, freq, head1, tail1, times=0; // timers
4      init(N);
5      QueryPerformanceFrequency((LARGE_INTEGER *)&freq);
6      QueryPerformanceCounter((LARGE_INTEGER *)&head);
7      for (int i=0; i<NN; i++)
8          for (int j=0; j<NN; j++)
9              col_sum[i] += (b[j][i]*a[j]);
10     QueryPerformanceCounter((LARGE_INTEGER *)&tail);
11     cout << "\nordCol:" <<(tail-head)*1000.0 / freq << "ms" << endl;
12 }

```

(二) 第三节

参考文献 [?] [?]

多行公式

$$a + b = a + b \quad (1)$$

$$\frac{a + b}{a - b} \quad (2)$$

行内公式: $\sum_{i=1}^N$

超链接 [YouTube](#)

带标号枚举

1. 1

2. 2

不带标号枚举

- 1
- 2

切换字体大小