南开大学

Nankai University

南　开　大　学

网 络 空 间 安 全 学 院

编译原理实验报告

第一次作业

王金晨 1911473

年级：2019 级

专业：信息安全

指导教师：王刚

2021 年 9 月 21 日

摘要

本文主要对我们经常使用的编译器 gcc 与 llvm 进行研究，来探究一个编译器在预处理阶段是如何对引用文件、命名空间、宏定义进行处理；在编译阶段如何进行词法分析、语法分析、语义分析、中间代码生成以及机器无关优化；以及汇编器与链接器加载器的工作探究。此外，还对于不同的优化级别在编译阶段中的各过程的中间结果的区别进行了部分探究。

关键字：预处理, 词法分析, 语法分析, 中间代码生成, 优化

目录

一、 概述

(一) 预处理

在这里我们对编译器如何处理引用文件、静态变量声明定义与宏定义进行探究，推测静态定义与宏定义对文件引用部分无影响，宏定义的处理方式是对宏的替换，代码如下：

对照代码

```
#include<iostream>
using namespace std;
int main()
{
    int a, b, i, n, t;
    a=0;b=1;i=1;
    cin>>n;
    cout<<a<<endl;
    cout<<b<<endl;
    while(i<n)
    {
        t=b;
        b = a + b;
        cout<<b<<endl;
        a=t;
        i=i+1;
    }
    return 0;
}
```

静态定义

```
#include<iostream>
using namespace std;
static int a = 0;
static int b = 1;
static int t = 0;
int main()
{
    int i, n;
    cin>>n;
    cout<<a<<endl;
    cout<<b<<endl;
    while(i<n)
    {
        t=b;
        b = a + b;
        cout<<b<<endl;
        a=t;
        i=i+1;
    }
```

```
    return 0;
}
```

宏定义

```
#include<iostream>
using namespace std;
#define zero int(0)
#define one  int(1)
int main()
{
    int a, b, i, n, t;
    a=zero;b=one;i=one;
    cin>>n;
    cout<<a<<endl;
    cout<<b<<endl;
    while(i<n)
    {
        t=b;
        b = a + b;
        cout<<b<<endl;
        a=t;
        i=i+one;
    }
    return 0;
}
```

使用如下命令进行预处理:

gcc -E test.cpp > test.i

由于预处理结果有几万行，在这里仅显示对照代码预处理后的几十行结果以及经过处理后的 main 函数代码:

pre

```
# 1 "test1.cpp"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "test1.cpp"
# 1 "/usr/include/c++/7/iostream" 1 3
# 36 "/usr/include/c++/7/iostream" 3
# 37 "/usr/include/c++/7/iostream" 3
# 1 "/usr/include/x86_64-linux-gnu/c++/7/bits/c++config.h" 1 3
# 229 "/usr/include/x86_64-linux-gnu/c++/7/bits/c++config.h" 3
# 229 "/usr/include/x86_64-linux-gnu/c++/7/bits/c++config.h" 3
namespace std
{
  typedef long unsigned int size_t;
```

```
  typedef long int ptrdiff_t;
  typedef decltype(nullptr) nullptr_t;
}
...
# 2 "test1.cpp"
using namespace std;
int main()
{
    int a, b, i, n, t;
    a=0;b=1;i=1;
    cin>>n;
    cout<<a<<endl;
    cout<<b<<endl;
    while(i<n)
    {
        t=b;
  b = a + b;
  cout<<b<<endl;
  a=t;
  i=i+1;
    }
    return 0;
}
```

可以看到，gcc 将该 cpp 文件包含的文件编译进来，并对 namespace std 中的定义写入到本文件中。由于文件内容太多，所以我们使用 diff 命令来查看文件间的不同以便快速找到编译器处理 static 与宏命令的地方：其中，test1 是对照文件，test2 是添加了 static 定义的文件，test3 是使用了宏定义的文件。

diff test1 test2

```
1c1
< # 1 "test1.cpp"
---
> # 1 "test2.cpp"
6c6
< # 1 "test1.cpp"
---
> # 1 "test2.cpp"
28152c28152
< # 2 "test1.cpp" 2
---
> # 2 "test2.cpp" 2
28154c28154
< # 2 "test1.cpp"
---
> # 2 "test2.cpp"
28155a28156,28158
```

```
> static int a = 0;
> static int b = 1;
> static int t = 0;
28158,28159c28161
<     int a, b, i, n, t;
<     a=0;b=1;i=1;
---
>     int i, n;
```

diff test1 test3

```
1c1
< # 1 "test1.cpp"
---
> # 1 "test3.cpp"
6c6
< # 1 "test1.cpp"
---
> # 1 "test3.cpp"
28152c28152
< # 2 "test1.cpp" 2
---
> # 2 "test3.cpp" 2
28154c28154
< # 2 "test1.cpp"
---
> # 2 "test3.cpp"
28155a28156,28157
>
>
28159c28161
<     a=0;b=1;i=1;
---
>     a=int(0);b=int(1);i=int(1);
28169c28171
<   i=i+1;
---
>   i=i+int(1);
```

可见，对于 static 定义，并没有进行其他处理，只是由于在编写代码时将 static 定义放在 main 函数开始之前导致的位置不同。对于宏的处理是把使用的宏直接进行文本意义上的替换，并删除宏定义语句。

(二) 编译器

在这一小节我探究了五个方面：词法分析、语法分析、中间代码生成 (以下简称 IR 生成)、机器无关优化、目标代码生成 (汇编代码)。在词法分析上探究了如何使用 gcc 与 llvm 进行 tokens 的扫描，在语法分析上主要探究如何使用上述二编译器生成 AST(抽象语法树)，IR 生成方面主要了解了 IR 的含义以及 CFG 对 IR 的反应，机器无关优化方面主要探究了 llvm 对 IR 的优化，

以及开启不同优化力度后 IR 的变化，目标代码生成则主要探究了 llvm 是如何生成汇编代码的。二编译器还有一个重要阶段是语义分析，该阶段作用为根据 AST 进行正确性上的检查，探究此部分需要阅读源代码，精力不足，不做探究。

1. 词法分析

使用命令：clang -E -Xclang -dump-tokens test.cpp

由于实际上是对预处理后的文件进行词法分析，所以会分析大量的 iostream 等库中的内容，导致词法分析结果过长，这里给出从分析 using name std 这句话开始的部分结果，如下：

```
using 'using'     [StartOfLine]  Loc=<test.cpp:2:1>
namespace 'namespace'     [LeadingSpace] Loc=<test.cpp:2:7>
identifier 'std'          [LeadingSpace] Loc=<test.cpp:2:17>
semi ';'                  Loc=<test.cpp:2:20>
int 'int'         [StartOfLine]  Loc=<test.cpp:3:1>
identifier 'main'         [LeadingSpace] Loc=<test.cpp:3:5>
l_paren '('               Loc=<test.cpp:3:9>
r_paren ')'               Loc=<test.cpp:3:10>
l_brace '{'       [StartOfLine]  Loc=<test.cpp:4:1>
int 'int'         [StartOfLine] [LeadingSpace]    Loc=<test.cpp:5:5>
identifier 'a'    [LeadingSpace] Loc=<test.cpp:5:9>
comma ','                 Loc=<test.cpp:5:10>
identifier 'b'    [LeadingSpace] Loc=<test.cpp:5:12>

equal '='                 Loc=<test.cpp:6:10>
numeric_constant '1'              Loc=<test.cpp:6:11>
semi ';'                  Loc=<test.cpp:6:12>
identifier 'i'            Loc=<test.cpp:6:13>
equal '='                 Loc=<test.cpp:6:14>
numeric_constant '1'              Loc=<test.cpp:6:15>
semi ';'                  Loc=<test.cpp:6:16>
identifier 'cin'          [StartOfLine] [LeadingSpace]    Loc=<test.cpp:7:5>
greatergreater '>>'               Loc=<test.cpp:7:8>
identifier 'n'            Loc=<test.cpp:7:10>
semi ';'                  Loc=<test.cpp:7:11>
identifier 'cout'         [StartOfLine] [LeadingSpace]    Loc=<test.cpp:8:5>
lessless '<<'             Loc=<test.cpp:8:9>
identifier 'a'            Loc=<test.cpp:8:11>
lessless '<<'             Loc=<test.cpp:8:12>
identifier 'endl'                 Loc=<test.cpp:8:14>
semi ';'                  Loc=<test.cpp:8:18>
..
semi ';'                  Loc=<test.cpp:9:18>
while 'while'     [StartOfLine] [LeadingSpace]    Loc=<test.cpp:10:5>
l_paren '('               Loc=<test.cpp:10:10>
identifier 'i'            Loc=<test.cpp:10:11>
less '<'                  Loc=<test.cpp:10:12>
identifier 'n'            Loc=<test.cpp:10:13>

```

```
plus '+'          [LeadingSpace] Loc=<test.cpp:13:9>
identifier 'b'    [LeadingSpace] Loc=<test.cpp:13:11>
semi ';'                  Loc=<test.cpp:13:12>
identifier 'cout'          [StartOfLine] [LeadingSpace]   Loc=<test.cpp:14:3>
lessless '<<'             Loc=<test.cpp:14:7>
identifier 'b'            Loc=<test.cpp:14:9>
lessless '<<'             Loc=<test.cpp:14:10>
identifier 'endl'                 Loc=<test.cpp:14:12>
semi ';'                  Loc=<test.cpp:14:16>

numeric_constant '1'              Loc=<test.cpp:16:7>
semi ';'                  Loc=<test.cpp:16:8>
r_brace '}'       [StartOfLine] [LeadingSpace]   Loc=<test.cpp:17:5>
return 'return'   [StartOfLine] [LeadingSpace]   Loc=<test.cpp:18:5>
numeric_constant '0'       [LeadingSpace] Loc=<test.cpp:18:12>
semi ';'                  Loc=<test.cpp:18:13>
r_brace '}'       [StartOfLine]   Loc=<test.cpp:19:1>
eof ''           Loc=<test.cpp:21:1>
```

可见。词法分析按照顺序进行扫描，为什么是从 using 开始呢? 因为在实操中，发现这条命令输出了数千行。而 using 之前的部分都是对于预编译结果的解读，而从 using 还是才是对程序正文的解读，可以看到，词法分析首先忽略了所有的空格换行等无含义字符，随后将每个关键词(如 using、std) 视为一个 token，那么在相应的 CFG(上下文无关文法) 中应该有这些终结符的存在，同时，他将每一个变量名字作为一个 token，甚至每个符号，可见这些都是 CFG 中的终结符。

2. 语法分析

主要探究 gcc 与 llvm 生成的语法树

对于 gcc 使用命令：gcc -fdump-tree-original-raw test.cpp

打开生成的文本文件，发现接近十几万行，而其中按照各个函数为划分构建的语法分析树，这里给出几个有代表性的节点：

```
;; Function std::bad_exception::bad_exception() (null)
;; enabled by -tree-original

@1       must_not_throw_expr type: @2       line: 49       body: @3
@2       void_type           name: @4       algn: 8
@3       statement_list      0    : @5      1    : @6
@4       type_decl           name: @7       type: @2       srcp: <built-in>:0
                             note: artificial
@5       cleanup_point_expr  type: @2       op 0: @8
@6       bind_expr           type: @2       body: @9
@7       identifier_node     strg: void     lngt: 4
@8       expr_stmt           type: @2       expr: @10
@9       statement_list      0    : @11     1    : @12
@10      modify_expr         type: @2       op 0: @13      op 1: @14
@11      cleanup_point_expr  type: @2       op 0: @15
@12      try_catch_expr      type: @2
```

```
@13     indirect_ref    type: @16       op 0: @17
@14     constructor     lngt: 0
@15     expr_stmt       type: @2        line: 49        expr: @18
@16     record_type     size: @19       algn: 64        bfld: @20

@24     addr_expr       type: @35       op 0: @36
@29     function_decl   name: @41       type: @46       scpe: @20
                        srcp: exception:49              note: member
                        accs: pub       note: constructor
                        args: @22       link: extern
@65     identifier_node strg: exception                 lngt: 9
@66     parm_decl       name: @31       type: @67       scpe: @52
                        srcp: exception.h:63            note: artificial
                        argt: @67       size: @19       algn: 64
                        used: 0
@67     pointer_type    qual: c         unql: @35       size: @19
                        algn: 64        ptd : @45
@68     nop_expr        type: @48       op 0: @22
@69     type_decl       name: @74       type: @56       srcp: <built-in>:0
                        note: artificial
@70     function_type   size: @61       algn: 8         retn: @75
@71     method_type     unql: @76       size: @61       algn: 8
                        clas: @45       retn: @2        prms: @77
@75     integer_type    name: @78       size: @79       algn: 32
                        prec: 32        sign: signed    min : @80
                        max : @81
@76     method_type     size: @61       algn: 8         clas: @45
                        retn: @2        prms: @77
@77     tree_list       valu: @35       chan: @82
@79     integer_cst     type: @25       int: 32
@82     tree_list       valu: @84       chan: @73
@83     identifier_node strg: int       lngt: 3
@84     reference_type  size: @19       algn: 64        refd: @85
@85     record_type     qual: c         name: @57       unql: @45
                        size: @19       algn: 64        vfld: @27
                        tag : struct    flds: @27       binf: @59
```

经过分析，每一行为一个节点 @ 加数字即表示一个节点，那么这到底有什么含义呢？经查阅资料，gcc 的 AST 的树节点结构如下：

```
union tree_node
{
  ....
}
```

节点内部内容省略，也就是说，这里的每个节点是一个 union 结构，类似于 C++ 中的类，但只占用一个成员大小的内存，也就是共用。

对于 llvm 使用命令：clang -E -Xclang -ast-dump test.cpp

生成的部分语法树如下：

可以看出一定的树状结构，甚至标出了行号列号，但是由于语法树太长，这里看不到根节点。最顶部一般是 TranslationUnitDecl(一个 Cpp 文件以及那些 include 包括的文件称为翻译单元 (TranslaitonUnit))，TypedefDecl、CXXRecordDecl、CompoundStmt 等称为 AST node，比较常见的有 Stmt、Decl 和 Expr 等。[1]

3. IR 生成

llvm 的 IR 生成，使用命令：clang -S -emit-llvm test.cpp 生成 test.ll 文件除去一些声明与定义外，直接查看对应主函数的 IR，仅给出部分：

```
define i32 @main() #4 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4

  store i32 1, i32* %3, align 4
  store i32 1, i32* %4, align 4
  %7 = call dereferenceable(280) %"class.std::basic_istream"* @_ZNSirsERi(%"
      class.std::basic_istream"* @_ZSt3cin, i32* dereferenceable(4) %5)
  %8 = load i32, i32* %2, align 4
  %9 = call dereferenceable(272) %"class.std::basic_ostream"* @_ZNSolsEi(%"
      class.std::basic_ostream"* @_ZSt4cout, i32 %8)

  %13 = call dereferenceable(272) %"class.std::basic_ostream"*
      @_ZNSolsEPFRSoS_E(%"class.std::basic_ostream"* %12, %"class.std::
      basic_ostream"* (%"class.std::basic_ostream"*)*
      @_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_)
  br label %14

```

```
; <label>:14:                                     ; preds = %18, %0
  %15 = load i32, i32* %4, align 4
  %16 = load i32, i32* %5, align 4
  %17 = icmp slt i32 %15, %16
  br i1 %17, label %18, label %29

; <label>:18:                                     ; preds = %14
  %19 = load i32, i32* %3, align 4
  store i32 %19, i32* %6, align 4
 @_ZNSolsEPFRSoS_E(%"class.std::basic_ostream"* %24, %"class.std::
     basic_ostream"* (%"class.std::basic_ostream"*)*
     @_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_)
  %26 = load i32, i32* %6, align 4
  store i32 %26, i32* %2, align 4
  %27 = load i32, i32* %4, align 4
  %28 = add nsw i32 %27, 1
  store i32 %28, i32* %4, align 4
  br label %14

; <label>:29:                                     ; preds = %14
  ret i32 0
}
```

分析可知，% 开头的为局部变量，@ 开头的为全局变量，与我们熟知的汇编类似，load 从内存取，大部分指令按照英文意思就可以猜出，可以说是十分易于理解了。

4. 机器无关优化

使用命令：llc -print-before-all -print-after-all a.ll，部分结果如下：

```
main:                                   # @main
        .cfi_startproc
# BB#0:
        pushq   %rbp
.Ltmp3:
        .cfi_def_cfa_offset 16
.Ltmp4:
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
.Ltmp5:
        .cfi_def_cfa_register %rbp
        subq    $32, %rsp
        movabsq $_ZSt3cin, %rdi
        leaq    -20(%rbp), %rsi
        movl    $0, -4(%rbp)
        movl    $0, -8(%rbp)
        movl    $1, -12(%rbp)
        movl    $1, -16(%rbp)
        callq   _ZNSirsERi
```

```
        movabsq $_ZSt4cout, %rdi
        movl    -8(%rbp), %esi
        callq   _ZNSolsEi
        movabsq $_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_,
            %rsi
        movq    %rax, %rdi
        callq   _ZNSolsEPFRSoS_E
        movabsq $_ZSt4cout, %rdi

        movl    -16(%rbp), %eax
        addl    $1, %eax
        movl    %eax, -16(%rbp)
        jmp     .LBB1_1
.LBB1_3:
        xorl    %eax, %eax
        addq    $32, %rsp
        popq    %rbp
        retq
.Lfunc_end1:
        .size   main, .Lfunc_end1-main
        .cfi_endproc
```

可以看到，优化的版本对于堆栈的使用更加高效，并且新增很多标签更加易读，另外，对于多数寄存器的操作，从操作整个 64 位寄存器优化为只操作寄存器的低 32 位，数据传输更快。

5. 汇编代码生成

由于使用的是 C++ 程序，所以使用 g++，g++ 是 gcc 的前端。命令如下：g++ -S test.i -o test.S 展示部分代码：

```
        .file   "test.cpp"
        .text
        .section        .rodata
        .type   _ZStL19piecewise_construct, @object
        .size   _ZStL19piecewise_construct, 1
_ZStL19piecewise_construct:
        .zero   1
        .local  _ZStL8__ioinit
        .comm   _ZStL8__ioinit,1,1
        .text
        .globl  main
        .type   main, @function
main:
.LFB1493:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
```

```
        .cfi_def_cfa_register 6
        subq    $32, %rsp
        movq    %fs:40, %rax
        movq    %rax, -8(%rbp)
        xorl    %eax, %eax

        movq    %rax, %rsi
        movq    %rdx, %rdi
        call    _ZNSolsEPFRSoS_E@PLT
.L3:
        movl    -28(%rbp), %eax
        cmpl    %eax, -16(%rbp)
        jge     .L2
        addl    $1, -16(%rbp)
        jmp     .L3
.L2:
        movl    $0, %eax
        movq    -8(%rbp), %rcx
        xorq    %fs:40, %rcx
        je      .L5
        call    __stack_chk_fail@PLT
.L5:
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

生成了 x86 汇编代码，可以看到，有在 main 之前的入口程序与 main 结束之后的收尾程序。再尝试生成 arm 格式汇编代码，同样只给出部分：

```
        .arch armv7-a
        .eabi_attribute 28, 1
        .fpu vfpv3-d16
        .eabi_attribute 20, 1
...
        .eabi_attribute 18, 4
        .file   "test.cpp"
        .local  _ZStL8__ioinit
...
main:
        .fnstart
.LFB1024:
        @ args = 0, pretend = 0, frame = 24
        @ frame_needed = 1, uses_anonymous_args = 0
        push    {r7, lr}
        .save {r7, lr}
        .pad #24
        sub     sp, sp, #24
        .setfp r7, sp, #0
```

```
20          add     r7, sp, #0
21          movw    r3, #:lower16:__stack_chk_guard
22          movt    r3, #:upper16:__stack_chk_guard
23          ldr     r3, [r3]
24          str     r3, [r7, #20]
25          movs    r3, #0
26  ...
27
28          bl      _ZNSolsEi
29          mov     r3, r0
30          movw    r1, #:lower16:
                _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_
31          movt    r1, #:upper16:
                _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_
32          mov     r0, r3
33          bl      _ZNSolsEPFRSoS_E
34          ldr     r3, [r7, #16]
35          str     r3, [r7, #4]
36  ...
37          .fnend
38          .size   main, .-main
39          .align  2
40          .syntax unified
41          .thumb
42          .thumb_func
43          .type   _Z41__static_initialization_and_destruction_0ii, %function
44  _Z41__static_initialization_and_destruction_0ii:
45          .fnstart
```

但是在实验过程中，并不能由之前生成的 test.i 文件直接使用命令 arm-linux-gnueabihf-g++ test.i -S -o test.S 生成汇编代码，需要先使用 arm-linux-gnueabihf-g++ 进行预处理生成对应的 test.i 文件之后才可以，推测是二者对于 C++ 文件的预处理方式不同，根据报错分析可能是符号上或定义上了一些区别。

(三) 汇编器

使用 gcc 对 test.cpp(而不是 arm-linux-gnueabihf-g++) 进行预处理与编译后，使用 gcc test.S -c -o test.o 生成二进制文件，这个时候的二进制文件还不是可执行的，因为他只是在源程序预处理之上所得到的二进制文件，本质上并不包含所引用的库的函数，所以也就没有办法正常执行，使用 objdump 反汇编，给出部分：

```
1   test.o:     文件格式 elf64-x86-64
2
3
4   Disassembly of section .text:
5
6   0000000000000000 <main>:
7      0:   55                      push   %rbp
8      1:   48 89 e5                mov    %rsp,%rbp
```

```
  4:   48 83 ec 20              sub    $0x20,%rsp
  8:   64 48 8b 04 25 28 00     mov    %fs:0x28,%rax
  f:   00 00
 11:   48 89 45 f8              mov    %rax,-0x8(%rbp)
 15:   31 c0                    xor    %eax,%eax
 17:   c7 45 e8 00 00 00 00     movl   $0x0,-0x18(%rbp)
 1e:   c7 45 ec 01 00 00 00     movl   $0x1,-0x14(%rbp)
 25:   c7 45 f0 01 00 00 00     movl   $0x1,-0x10(%rbp)
 2c:   48 8d 45 e4              lea    -0x1c(%rbp),%rax
 30:   48 89 c6                 mov    %rax,%rsi
 33:   48 8d 3d 00 00 00 00     lea    0x0(%rip),%rdi        # 3a <main+0x3a>
 3a:   e8 00 00 00 00           callq  3f <main+0x3f>
 3f:   8b 45 e8                 mov    -0x18(%rbp),%eax
 42:   89 c6                    mov    %eax,%esi
 44:   48 8d 3d 00 00 00 00     lea    0x0(%rip),%rdi        # 4b <main+0x4b>
 4b:   e8 00 00 00 00           callq  50 <main+0x50>
 50:   48 89 c2                 mov    %rax,%rdx
 53:   48 8b 05 00 00 00 00     mov    0x0(%rip),%rax        # 5a <main+0x5a>
 5a:   48 89 c6                 mov    %rax,%rsi
 5d:   48 89 d7                 mov    %rdx,%rdi
 60:   e8 00 00 00 00           callq  65 <main+0x65>
 65:   8b 45 ec                 mov    -0x14(%rbp),%eax
 68:   89 c6                    mov    %eax,%esi
 6a:   48 8d 3d 00 00 00 00     lea    0x0(%rip),%rdi        # 71 <main+0x71>
 71:   e8 00 00 00 00           callq  76 <main+0x76>
 76:   48 89 c2                 mov    %rax,%rdx
 79:   48 8b 05 00 00 00 00     mov    0x0(%rip),%rax        # 80
```

与原来的.S 文件由较大区别，多数体现在对于变量的命名以及一些声明定义等，原因猜测有 2：首先，汇编器在将汇编语言转换成二进制时会进行一定程度的优化；其次，汇编器在翻译时会导致一些信息的丢失，比如节等信息的设置等，会导致反汇编得到的结果不可能完全一样，因为这是信息不对等的。接下来开启优化尝试结果发现反汇编得到的代码一模一样，要么是在这一步不存在优化环节，要么是这个小程序没有优化空间了，至少在 o2 程度上没有优化空间

(四) 链接器加载器

进行链接：使用 g++ test.o -o test，而后使用 objdump 进行反汇编得到，给出部分：

```
test:     文件格式 elf64-x86-64
Disassembly of section .init:
00000000000007e0 <_init>:
 7e0:   48 83 ec 08             sub    $0x8,%rsp
 7e4:   48 8b 05 fd 07 20 00    mov    0x2007fd(%rip),%rax        # 200fe8 <
    __gmon_start__>
 7eb:   48 85 c0                test   %rax,%rax
 7ee:   74 02                   je     7f2 <_init+0x12>
 7f0:   ff d0                   callq  *%rax
 7f2:   48 83 c4 08             add    $0x8,%rsp
 7f6:   c3                      retq
```

```
Disassembly of section .plt:
0000000000000800 <.plt>:
 800:   ff 35 82 07 20 00        pushq  0x200782(%rip)        # 200f88 <
    _GLOBAL_OFFSET_TABLE_+0x8>
 806:   ff 25 84 07 20 00        jmpq   *0x200784(%rip)       # 200f90 <
    _GLOBAL_OFFSET_TABLE_+0x10>
 80c:   0f 1f 40 00              nopl   0x0(%rax)

0000000000000810 <_ZNSirsERi@plt>:
 810:   ff 25 82 07 20 00        jmpq   *0x200782(%rip)       # 200f98 <
    _ZNSirsERi@GLIBCXX_3.4>
 816:   68 00 00 00 00           pushq  $0x0
 81b:   e9 e0 ff ff ff           jmpq   800 <.plt>

0000000000000820 <__cxa_atexit@plt>:
 820:   ff 25 7a 07 20 00        jmpq   *0x20077a(%rip)       # 200fa0 <
    __cxa_atexit@GLIBC_2.2.5>
 826:   68 01 00 00 00           pushq  $0x1
 82b:   e9 d0 ff ff ff           jmpq   800 <.plt>

...

Disassembly of section .plt.got:

0000000000000870 <__cxa_finalize@plt>:
 870:   ff 25 52 07 20 00        jmpq   *0x200752(%rip)       # 200fc8 <
    __cxa_finalize@GLIBC_2.2.5>
 876:   66 90                    xchg   %ax,%ax
Disassembly of section .text:
0000000000000880 <_start>:
 880:   31 ed                    xor    %ebp,%ebp
 882:   49 89 d1                 mov    %rdx,%r9
 885:   5e                       pop    %rsi
...
 971:   0f 1f 80 00 00 00 00     nopl   0x0(%rax)
 978:   f3 c3                    repz retq
 97a:   66 0f 1f 44 00 00        nopw   0x0(%rax,%rax,1)


000000000000098a <main>:
 98a:   55                       push   %rbp
 98b:   48 89 e5                 mov    %rsp,%rbp
 98e:   48 83 ec 20              sub    $0x20,%rsp
 992:   64 48 8b 04 25 28 00     mov    %fs:0x28,%rax
 999:   00 00
 99b:   48 89 45 f8              mov    %rax,-0x8(%rbp)
 99f:   31 c0                    xor    %eax,%eax
 9a1:   c7 45 e8 00 00 00 00     movl   $0x0,-0x18(%rbp)
```

```
 9a8:   c7 45 ec 01 00 00 00    movl   $0x1,-0x14(%rbp)
 9af:   c7 45 f0 01 00 00 00    movl   $0x1,-0x10(%rbp)
 9b6:   48 8d 45 e4             lea    -0x1c(%rbp),%rax
...
 a4f:   8b 45 f4                mov    -0xc(%rbp),%eax
 a52:   89 45 e8                mov    %eax,-0x18(%rbp)
 a55:   83 45 f0 01             addl   $0x1,-0x10(%rbp)
 a59:   eb ba                   jmp    a15 <main+0x8b>
 a5b:   b8 00 00 00 00          mov    $0x0,%eax
 a60:   48 8b 4d f8             mov    -0x8(%rbp),%rcx
 a64:   64 48 33 0c 25 28 00    xor    %fs:0x28,%rcx
 a6b:   00 00
 a6d:   74 05                   je     a74 <main+0xea>
 a6f:   e8 cc fd ff ff          callq  840 <__stack_chk_fail@plt>
 a74:   c9                      leaveq
 a75:   c3                      retq

Disassembly of section .fini:

0000000000000b54 <_fini>:
 b54:   48 83 ec 08             sub    $0x8,%rsp
 b58:   48 83 c4 08             add    $0x8,%rsp
 b5c:   c3                      retq
```

可以看到还是有较大不同的，test 文件的反汇编显示 init 程序，而后是 plt 表、start 程序，在 start 程序中为 main 函数分配了栈之后才正式执行 main 函数，并且在 main 函数之后会返回 start 函数并跳转到 fini 函数进行堆栈回收。同时，对于整个程序来说，多出了很多在源代码中没有使用到的函数，比如检查堆栈的函数等，这些是在链接加载过程中会给程序增加一些保护，比如数组越界等检查，来让程序有能力应对针对程序脆弱性的攻击，从而提高程序安全性。

二、 总结

预处理会将包含文件代码插入源文件，导致预处理结果很长，其主要工作就是插入引用的文件或库，并对宏做直接的替换。

编译阶段主要预处理的结果进行词法分析、语法分析、语义分析、优化、代码生成。词法分析采用逐个扫描识别 token，其中会对空格、Tab 键这种排版符号做忽视处理。语法分析主要生成抽象语法树，gcc 与 llvm 生成的语法树采用的节点定义并不相同，相比之下我认为 gcc 更为好读，且可以生成图片便于理解。优化方面可以看到 llvm 的 pass 在内存使用效率、寄存器使用效率、指令执行效率上进行了优化。在代码生成上可以看到由于平台架构不同，对于 x86 与 amd 架构需要不同的编译器来进行汇编代码的生成，原因在于二者汇编代码的不同。

汇编阶段对汇编代码进行处理，将其转为机器码，这个时候的二进制文件是不可执行的，其不具备库函数，也就没办法正常执行，同时也没有被指明在执行时应该被装载到哪里。在这个阶段是无法使用 gcc 对文件进行汇编的，原因在于使用的是 C++ 语言，gcc 无法找到 C++ 语言的库，因而使用 g++ 进行汇编，g++ 与 gcc 并无本质区别，g++ 可以看作对 gcc 的一个 C++ 上的封装，二者在预处理、编译阶段没有区别，只在生成二进制文件的时候产生区别，这是语言特性的原因。

链接阶段就没什么可说的了。

对于 gcc 与 llvm 二者在其他方面没有太大区别，除了灵活性上。llvm 采用统一的中间代码，使得其可以用于各个语言的编译，而不单单是用于 C 语言的 clang，其前端与后端可以自由选择，而不是向 gcc 一样对于不同架构需要开发不同版本。

参考文献

[1] liveforthing. Clang 之语法抽象语法树 AST. https://www.cnblogs.com/zhangke007/p/4714245.html, 2015.