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：了解你的编译器

付文轩 1911410

年级：2019 级

专业：信息安全

指导教师：王刚

2021 年 9 月 25 日

摘要

在一个代码从完成到运行，这之间包含很多过程，这些过程大致可以分为四步：预处理、编译、汇编和链接。本次实验将在 Ubuntu 环境下进行实验实验相关内容为：以 GCC 为研究对象（或你常用的、熟悉的编译工具），更深入地探究语言处理系统的完整工作过程：预处理器做了什么？编译器做了什么？汇编器做了什么？链接器做了什么？本文将结合教材、PPT、网上相关资料，对如上的四个工作进行探究

关键字：预处理器，编译器，编译器，汇编器，链接器

目录

一、 实验知识准备

(一) 实验准备

1. 虚拟环境配置

本次实验环境为 Ubuntu 18.04，使用工具 VMware 配置此虚拟机，得到的虚拟机如图1所示

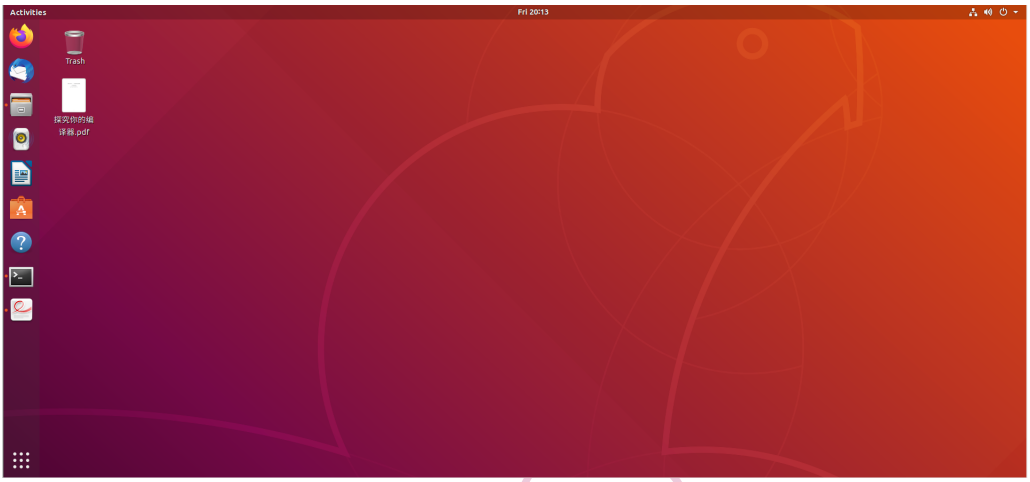

图 1: 虚拟机截图

2. 实验工具安装

本次实验需要用到如：gcc，clang 等工具，安装代码如下：

```
sudo apt install gcc
sudo apt install clang
sudo apt install llvms
sudo apt install -y flex
sudo apt install -y bison
sudo apt install graphviz
sudo apt-get install gawk
```

3. 实验代码编写

以阶乘代码为例，观察不同编写方式对四个阶段的影响

main 函数内声明并分开赋值

```
#include<iostream>
using namespace std;
void main() {
    int i, n, f;

    n = 10;
    i = 2;
    f = 1;
```

```
9
10      while (i <= n) {
11          f = f * i;
12          i = i + 1;
13      }
14      cout << f << endl;
15  }
```

main 函数内声明和赋值同时进行

```
1   #include<iostream>
2   using namespace std;
3   void main() {
4       int n = 10;
5       int i = 2;
6       int f = 1;
7
8       while (i <= n) {
9           f = f * i;
10          i = i + 1;
11      }
12      cout << f << endl;
13  }
```

main 函数外使用静态变量方式声明并赋值

```
1   #include<iostream>
2   using namespace std;
3
4   static int n = 10;
5
6   void main() {
7       int i = 2;
8       int f = 1;
9       while (i <= n) {
10          f = f * i;
11          i = i + 1;
12      }
13      cout << f << endl;
14  }
```

使用宏定义方式

```
1   #include<iostream>
2   #define n 10
3
4   using namespace std;
5
6   void main() {
7       int f = 1;
```

```
    int i = 2;
    while (i <= n) {
        f = f * i;
        i = i + 1;
    }
    cout << f << endl;
}
```

(二) 编译环境简介

在如今，几乎所有程序员都会使用高级语言进行编写，较少的情况下才会使用汇编等语言进行开发，而计算机是无法识别并运行高级语言的，由此就需要一个“翻译”将高级语言转换成汇编语言，进而转换成机器码，使得计算机能够运行程序，实现对应的目的。在这一过程中，编译器就扮演着这样一个“翻译”的角色。其中最广为人知也是使用相对较多的编译器是——gcc。

1. GCC 编译器简介

GCC 是 GNU 项目的编译器组件之一，也是 GNU 最具有代表性的作品。在 GCC 设计之初仅仅作为一个 C 语言的编译器，可是经过十多年的发展，GCC 已经不仅仅能支持 C 语言；它现在还支持 Ada 语言、C++ 语言、Java 语言、Objective C 语言，Pascal 语言、COBOL 语言，以及支持函数式编程和逻辑编程的 Mercury 语言，等等。而 GCC 更为强大的一点是 GCC 允许程序员将编译过程中得到的语法中间表达导出成为数据文件，可以让程序员通过中文件进一步了解到编译代码过程中所经历的过程。

2. GCC 使用方式

GCC 是 Linux 下基于命令行的 C 语言编译器，其基本的使用语法为：gcc [option |filename]

对于编译 C++ 的源程序，其基本语法如为：g++ [option |filename]

其中 option 为 GCC 使用时的选项，而 filename 为需要 GCC 做编译的处理的的文件名。

编译常用参数

–c选项：这是GCC命令的常用选项。–c选项告诉GCC仅把源程序编译为目标代码而不做链接工作，所以采用该选项的编译指令不会生成最终的可执行程序，而是生成一个与源程序文件名相同的以.o为后缀的目标文件。例如一个Test.c的源程序经过下面的编译之后会生成一个Test.o文件
–S选项：使用该选项会生成一个后缀名为.s的汇编语言文件，但是同样不会生成可执行程序。
–e选项：–e选项只对文件进行预处理，预处理的输出结果被送到标准输出（比如显示器）。
–v选项：在Shell的提示符号下键入gcc –v，屏幕上就会显示出目前正在使用的gcc版本的信息

| -x language：强制编译器指定的语言编译器来编译某个源程序。

(三) 各阶段概述

一般高级语言程序编译的过程：预处理、编译、汇编、链接。gcc 在后台实际上也经历了这几个过程，我们可以通过-v 参数查看它的编译细节，如果想看某个具体的编译过程，则可以分别使用-E,-S,-c 和 -O，对应的后台工具则分别为 cpp,cc1,as,ld。下面我们将逐步分析这几个过程以及相关的内容，诸如语法检查、代码调试、汇编语言等

1. 预处理阶段

编译器以 C 文件作为一个单元，首先读这个 C 文件，发现第一句与第二句是包含一个头文件，就会在所有搜索路径中寻找这两个文件，找到之后，就会将相应头文件中再去处理宏，变量，函数声明，嵌套的头文件包含等，检测依赖关系，进行宏替换，看是否有重复定义与声明的情况发生，最后将那些文件中所有的东西全部扫描进这个当前的 C 文件中，形成一个中间 “C 文件”。预处理器主要是对以下几个方面进行处理：

1. 宏定义指令，如 define a b 对于这种伪指令，预编译所要做的是将程序中的所有 a 用 b 替换，但作为字符串常量的 a 则不被替换。还有 undef，则将取消对某个宏的定义，使以后该串的出现不再被替换。
2. 条件编译指令，如 ifdef，ifndef，else，elif，endif 等。这些伪指令的引入使得程序员可以通过定义不同的宏来决定编译程序对哪些代码进行处理。预编译程序将根据有关的文件，将那些不必要的代码过滤掉
3. 头文件包含指令，如 include ”FileName” 或者 include 等。该指令将头文件中的定义统统都加入到它所产生的输出文件中，以供编译程序对之进行处理。
4. 特殊符号，预编译程序可以识别一些特殊的符号。例如在源程序中出现的 LINE 标识将被解释为当前行号（十进制数），FILE 则被解释为当前被编译的 C 源程序的名称。预编译程序对于在源程序中出现的这些串将用合适的值进行替换。
5. 删除注释（删除” // “和”/**/”）
6. 添加行号和文件标识
7. 保留 progma

2. 编译阶段

此阶段主要充当 “翻译” 角色，将不同语言的高级语言代码翻译成目标汇编代码。本阶段又可以细分成多个小的阶段

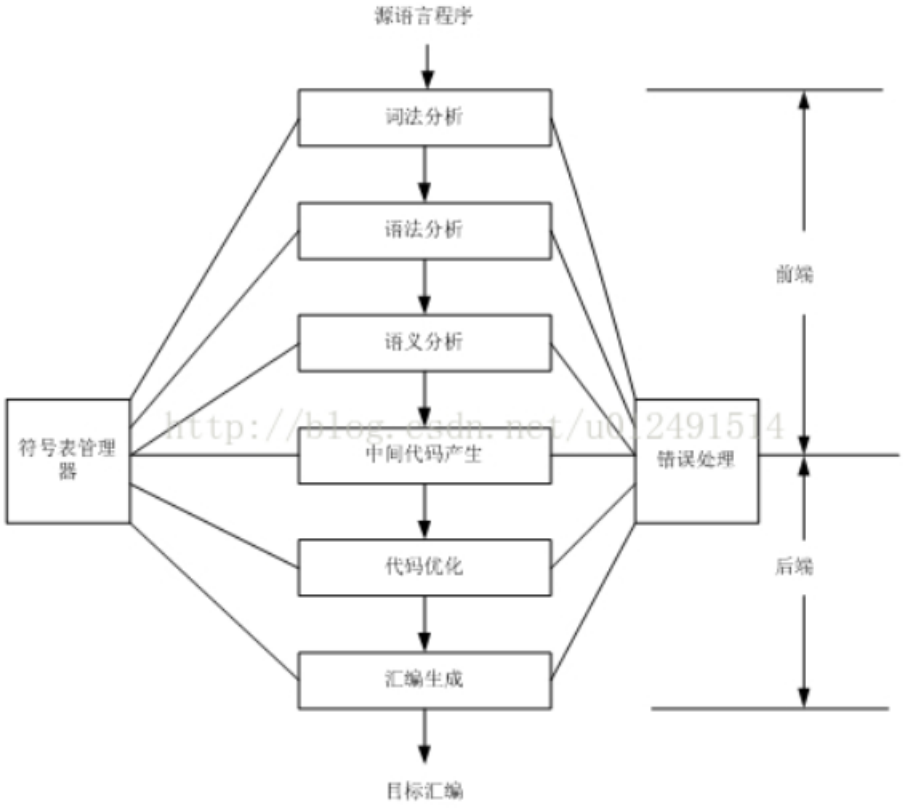

图 2: 编译阶段流程图

此阶段便是本学期课程需要掌握的重点阶段，详细内容就不在这里展开描述了。

3. 汇编阶段

此阶段主要完成将汇编代码翻译成机器码指令，并将这些指令打包形成可重定位的目标文件，[.O] 文件，是二进制文件。
此阶段由汇编器完成。

4. 链接阶段

生成.exe 为后缀的文件，也就是俗称的可执行的二进制文件

1. 合并段（相同段之间）和符号表

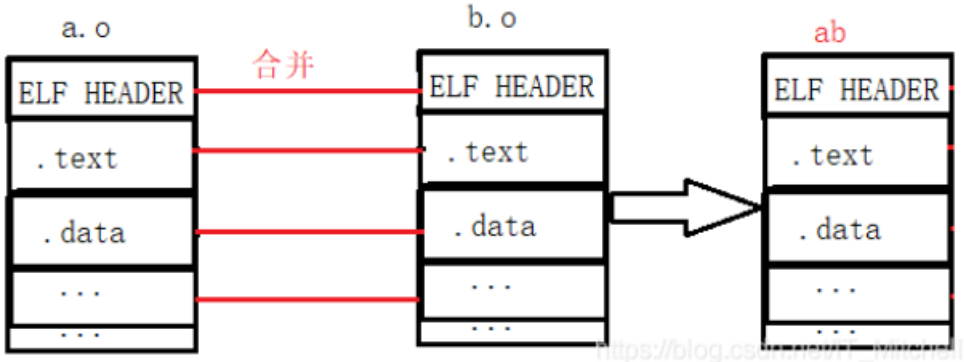

图 3: 合并操作

2. 进行符号解析：在符号引用的地方找到符号定义的地方
3. 分配地址和空间
4. 符号的重定位

二、 实验过程

在这一节中，我们将对不同阶段中不同编写方式的代码进行四个过程的对比。在之后将按照之前代码编写顺序，对四个版本的代码进行分别处理，并以“版本一”“版本二”的方式进行指代。

(一) 预处理阶段

根据预编译阶段的功能：预处理阶段会处理预编译指令，包括绝大多数的开头的指令，如 include define if 等等，对 include 指令会替换对应的头文件，对 define 的宏命令会直接替换相应内容，同时会删除注释，添加行号和文件名标识。我们才将之前阶乘代码以四种方式进行编写，在预处理阶段，我们使用 gcc 对不同版本的源文件进行处理。
常见预处理命令如下

- #define 定义一个预处理宏
- #undef 取消宏的定义
- #include 包含一个文件
- #if 预处理语法中的条件命令
- #ifdef 判断某个宏是否被定义，若定义，则执行后面的语句
- #ifndef 判断某个宏是否被定义，若不被定义，则执行后面的语句
- #else elif endif 与 ifdef 和 ifndef 定义相关
- #pragma 说明编译器的信息
- #warning 显示编译警告信息

- #error 显示编译错误信息

使用命令 gcc test.cpp -E -o test.i 进行预处理
注：在对 cpp 文件进行操作时，如果是 ubuntu 系统，会出现 gcc: error trying to exec 'cc1plus': execvp: 没有那个文件或目录的情况。此时的解决方式是在命令行中执行：sudo apt-get install –reinstall build-essential

1. **生成.i 文件**

得到版本一代码的文件 test1.i 如下：

```
# 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;

}
# 251 "/usr/include/x86_64-linux-gnu/c++/7/bits/c++config.h" 3
namespace std
{
  inline namespace __cxx11 __attribute__((__abi_tag__ ("cxx11"))) { }
}
namespace __gnu_cxx
{
  inline namespace __cxx11 __attribute__((__abi_tag__ ("cxx11"))) { }
}
... ... ... ...
# 2 "test1.cpp" 2

# 2 "test1.cpp"
using namespace std;
```

```
39 void main() {
40     int i, n, f;
41
42     n = 10;
43     i = 2;
44     f = 1;
45
46     while (i <= n) {
47         f = f * i;
48         i = i + 1;
49     }
50     cout << f << endl;
51 }
```

由于生成的.i 文件过长，所以在这里只放了部分片段。
重复上述步骤，得到如图4所示的所有文件

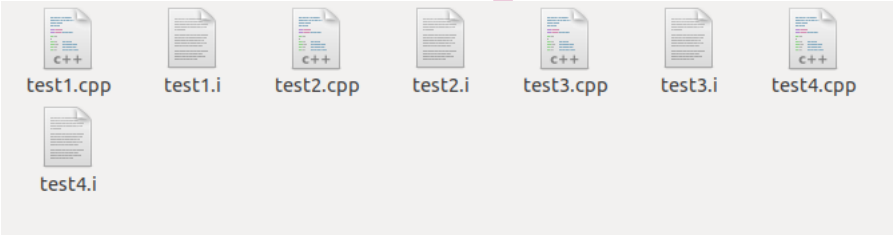

图 4: .i 文件截图

2. 文件对比

随便打开一个.i 文件，可以发现每个文件都有上万行，这样的代码是不可能人工进行对比、寻找差异的，在这里我们使用 Ubuntu 下的 diff 工具，对不同形式代码生成的.i 文件进行对比，其中以版本一作为标准。
从以上的代码中，可以看见预处理阶段对代码中的 include 进行了替换（从代码中第 7 行开始），同时还添加了行号（代码内部的后）和文件名标识（见代码中的第 35、37 行）。

1. 版本一和版本二对比，得到的内容如下：

```
1     1c1
2   < # 1 "test1.cpp"
3   ---
4   > # 1 "test2.cpp"
5   6c6
6   < # 1 "test1.cpp"
7   ---
8   > # 1 "test2.cpp"
9   28152c28152
10  < # 2 "test1.cpp" 2
11  ---
12  > # 2 "test2.cpp" 2
```

```
28154c28154
< # 2 "test1.cpp"
---
> # 2 "test2.cpp"
28157,28161c28157,28159
<     int i, n, f;
<
<     n = 10;
<     i = 2;
<     f = 1;
---
>     int n = 10;
>     int i = 2;
>     int f = 1;
```

2. 版本一和版本三对比，得到的内容如下：

```
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"
28156,28157d28155
< void main() {
<     int i, n, f;
28159,28161c28157
<     n = 10;
<     i = 2;
<     f = 1;
---
> static int n = 10;
28162a28159,28161
> void main() {
>     int i = 2;
>     int f = 1;
```

3. 版本一和版本四对比，得到的内容如下：

```
1     1c1
2   < # 1 "test1.cpp"
3   ---
4   > # 1 "test4.cpp"
5   6c6
6   < # 1 "test1.cpp"
7   ---
8   > # 1 "test4.cpp"
9   28152c28152
10  < # 2 "test1.cpp" 2
11  ---
12  > # 2 "test4.cpp" 2
13  28154,28157d28153
14  < # 2 "test1.cpp"
15  < using namespace std;
16  < void main() {
17  <     int i, n, f;
18  28159,28161d28154
19  <     n = 10;
20  <     i = 2;
21  <     f = 1;
22  28163c28156,28162
23  <     while (i <= n) {
24  ---
25  > # 4 "test4.cpp"
26  > using namespace std;
27  >
28  > void main() {
29  >     int f = 1;
30  >     int i = 2;
31  >     while (i <= 10) {
```

从以上的几个版本代码生成的.i 文件对比结果来看，除了因为编写方式不同导致的行号、位置不同以外，其他都没有什么区别。由此也从侧面验证了预处理的作用中对 define 宏命令进行替换的作用。

(二) 编译阶段

根据实验指导书，在这一阶段我们将会使用 llvm 工具进行每一个步骤的执行，采用的指令为 clang；并且根据网上查询到的资料显示，clang 从 2013 年开始就已经支持对 C++ 语言的操作，所以这里无需进行替换，可以直接使用 clang 进行实验。

1. clang 工具简单介绍

使用方式：clang [options] <inputs>
常用参数说明：

1. - Print（但不运行）该编译要运行的命令
2. -fsyntax-only 防止编译器生成代码, 只是语法级别的说明和修改
3. -Xclang 向 clang 编译器传递参数
4. -dump-tokens 运行预处理器, 拆分内部代码段为各种 token
5. -ast-dump 构建抽象语法树 AST, 然后对其进行拆解和调试
6. -S 只运行预处理和编译步骤
7. -fobjc-arc 为 OC 对象生成 retain 和 release 的调用
8. -emit-llvm 使用 LLVM 描述汇编和对象文件
9. -o 输出到目标文件
10. -c 只运行预处理, 编译和汇编步骤

2. 词法分析

首先尝试对版本一的代码进行词法分析，得到非常长的词法分析结果，找到关于函数主体部分的结果如下：

```
using 'using'     [StartOfLine]  Loc=<test1.cpp:2:1>
namespace 'namespace'     [LeadingSpace] Loc=<test1.cpp:2:7>
identifier 'std'          [LeadingSpace] Loc=<test1.cpp:2:17>
semi ';'                  Loc=<test1.cpp:2:20>
void 'void'       [StartOfLine]  Loc=<test1.cpp:3:1>
identifier 'main'         [LeadingSpace] Loc=<test1.cpp:3:6>
l_paren '('               Loc=<test1.cpp:3:10>
r_paren ')'               Loc=<test1.cpp:3:11>
l_brace '{'       [LeadingSpace] Loc=<test1.cpp:3:13>
int 'int'         [StartOfLine] [LeadingSpace]    Loc=<test1.cpp:4:5>
identifier 'i'    [LeadingSpace] Loc=<test1.cpp:4:9>
comma ','                 Loc=<test1.cpp:4:10>
identifier 'n'    [LeadingSpace] Loc=<test1.cpp:4:12>
comma ','                 Loc=<test1.cpp:4:13>
identifier 'f'    [LeadingSpace] Loc=<test1.cpp:4:15>
semi ';'                  Loc=<test1.cpp:4:16>
identifier 'n'    [StartOfLine] [LeadingSpace]    Loc=<test1.cpp:6:5>
equal '='         [LeadingSpace] Loc=<test1.cpp:6:7>
numeric_constant '10'     [LeadingSpace] Loc=<test1.cpp:6:9>
semi ';'                  Loc=<test1.cpp:6:11>
identifier 'i'    [StartOfLine] [LeadingSpace]    Loc=<test1.cpp:7:5>
equal '='         [LeadingSpace] Loc=<test1.cpp:7:7>
numeric_constant '2'      [LeadingSpace] Loc=<test1.cpp:7:9>
semi ';'                  Loc=<test1.cpp:7:10>
identifier 'f'    [StartOfLine] [LeadingSpace]    Loc=<test1.cpp:8:5>
equal '='         [LeadingSpace] Loc=<test1.cpp:8:7>
numeric_constant '1'      [LeadingSpace] Loc=<test1.cpp:8:9>
```

```
semi ';'                Loc=<test1.cpp:8:10>
while 'while'     [StartOfLine] [LeadingSpace]   Loc=<test1.cpp:10:5>
l_paren '('       [LeadingSpace] Loc=<test1.cpp:10:11>
identifier 'i'          Loc=<test1.cpp:10:12>
lessequal '<='    [LeadingSpace] Loc=<test1.cpp:10:14>
identifier 'n'    [LeadingSpace] Loc=<test1.cpp:10:17>
r_paren ')'             Loc=<test1.cpp:10:18>
l_brace '{'       [LeadingSpace] Loc=<test1.cpp:10:20>
identifier 'f'    [StartOfLine] [LeadingSpace]   Loc=<test1.cpp:11:9>
equal '='         [LeadingSpace] Loc=<test1.cpp:11:11>
identifier 'f'    [LeadingSpace] Loc=<test1.cpp:11:13>
star '*'          [LeadingSpace] Loc=<test1.cpp:11:15>
identifier 'i'    [LeadingSpace] Loc=<test1.cpp:11:17>
semi ';'                Loc=<test1.cpp:11:18>
identifier 'i'    [StartOfLine] [LeadingSpace]   Loc=<test1.cpp:12:9>
equal '='         [LeadingSpace] Loc=<test1.cpp:12:11>
identifier 'i'    [LeadingSpace] Loc=<test1.cpp:12:13>
plus '+'          [LeadingSpace] Loc=<test1.cpp:12:15>
numeric_constant '1'      [LeadingSpace] Loc=<test1.cpp:12:17>
semi ';'                Loc=<test1.cpp:12:18>
r_brace '}'       [StartOfLine] [LeadingSpace]   Loc=<test1.cpp:13:5>
identifier 'cout'         [StartOfLine] [LeadingSpace]   Loc=<test1.cpp:14:5>
lessless '<<'     [LeadingSpace] Loc=<test1.cpp:14:10>
identifier 'f'    [LeadingSpace] Loc=<test1.cpp:14:13>
lessless '<<'     [LeadingSpace] Loc=<test1.cpp:14:15>
identifier 'endl'         [LeadingSpace] Loc=<test1.cpp:14:18>
semi ';'                Loc=<test1.cpp:14:22>
r_brace '}'       [StartOfLine]  Loc=<test1.cpp:15:1>
eof ''          Loc=<test1.cpp:15:2>
```

可以看见对函数正文部分的解读是从 using namespace std 这条语句开始的，其之前的上千行结果应当是对 include 的库导入后的内容进行操作。同时在编写代码时，为了观看方便，在代码中是包含有很多空格、缩进、换行符的，但是从词法分析的结果可以明显看出，其中并没有包括这些，即词法分析会忽略这些没有实际意义的词。然后从 using 开始，以空格进行分割，每个词都形成一个对应的 token。

3. 语法分析

输入指令：clang -E -Xclang -ast-dump test1.cpp，对版本一的简化代码（为了方便观察结果，删除了 include 语句）进行语法分析，得到部分内容的截图如图5：

图 5: 语法分析结果

从截图中已经可以看出一点树状结构了，并且在每个叶节点都标明了行号、列号以及对应的类型值，同时发现非叶节点和我们通常使用的 expr 等还是有些差别的，在这里面使用的是 TypedefDecl、CXXRecordDecl、CompoundStmt 等，这些词也被称为 AST node。

4. 附加阶段：语法树图像生成

使用指令：gcc -fdump-tree-original-raw ./test.c 可以得到 test.c.003t.original 文件 [1]，其内容如下：

```
1
2  ;; Function main (null)
3  ;; enabled by -tree-original
4
5  @1       statement_list   0    : @2       1    : @3
6  @2       bind_expr        type: @4       vars: @5       body: @6
7  @3       return_expr      type: @4       expr: @7
8  @4       void_type        name: @8       algn: 8
9  @5       var_decl         name: @9       type: @10      scpe: @11
10                            srcp: test.c:3                size: @12
11                            algn: 32       used: 1
12 @6       statement_list   0    : @13      1    : @14      2    : @15
```

```
                        3    : @16       4    : @17       5    : @18
                        6    : @19       7    : @20       8    : @21
                        9    : @22       10   : @23       11   : @24
                        12   : @25       13   : @26
@7      modify_expr     type: @10        op 0: @27        op 1: @28
@8      type_decl       name: @29        type: @4
@9      identifier_node strg: i          lngt: 1
@10     integer_type    name: @30        size: @12        algn: 32
                        prec: 32         sign: signed     min : @31
                        max : @32
@11     function_decl   name: @33        type: @34        srcp: test.c:2
                        link: extern
@12     integer_cst     type: @35      int: 32
@13     decl_expr       type: @4
@14     decl_expr       type: @4
@15     decl_expr       type: @4
@16     modify_expr     type: @10        op 0: @36        op 1: @37
@17     modify_expr     type: @10        op 0: @5         op 1: @38
@18     modify_expr     type: @10        op 0: @39        op 1: @40
@19     goto_expr       type: @4         labl: @41
@20     label_expr      type: @4         name: @42
@21     modify_expr     type: @10        op 0: @39        op 1: @43
@22     modify_expr     type: @10        op 0: @5         op 1: @44
@23     label_expr      type: @4         name: @41
@24     cond_expr       type: @4         op 0: @45        op 1: @46
                        op 2: @47
@25     label_expr      type: @4         name: @48
@26     return_expr     type: @4         expr: @49
@27     result_decl     type: @10        scpe: @11        srcp: test.c:2
                        note: artificial                  size: @12
                        algn: 32
@28     integer_cst     type: @10      int: 0
@29     identifier_node strg: void       lngt: 4
@30     type_decl       name: @50        type: @10
@31     integer_cst     type: @10      int: -2147483648
@32     integer_cst     type: @10      int: 2147483647
@33     identifier_node strg: main       lngt: 4
@34     function_type   unql: @51        size: @52        algn: 8
                        retn: @10
@35     integer_type    name: @53        size: @54        algn: 128
                        prec: 128        sign: unsigned   min : @55
                        max : @56
@36     var_decl        name: @57        type: @10        scpe: @11
                        srcp: test.c:3                    size: @12
                        algn: 32         used: 1
@37     integer_cst     type: @10      int: 10
@38     integer_cst     type: @10      int: 2
@39     var_decl        name: @58        type: @10        scpe: @11
```

```
                        srcp: test.c:3                  size: @12
                        algn: 32        used: 1
@40     integer_cst     type: @10       int: 1
@41     label_decl      type: @4        scpe: @11       note: artificial
@42     label_decl      type: @4        scpe: @11       note: artificial
@43     mult_expr       type: @10       op 0: @39       op 1: @5
@44     plus_expr       type: @10       op 0: @5        op 1: @40
@45     le_expr         type: @10       op 0: @5        op 1: @36
@46     goto_expr       type: @4        labl: @42
@47     goto_expr       type: @4        labl: @48
@48     label_decl      type: @4        scpe: @11       note: artificial
@49     modify_expr     type: @10       op 0: @27       op 1: @28
@50     identifier_node strg: int       lngt: 3
@51     function_type   size: @52       algn: 8         retn: @10
@52     integer_cst     type: @35       int: 8
@53     identifier_node strg: bitsizetype               lngt: 11
@54     integer_cst     type: @35       int: 128
@55     integer_cst     type: @35       int: 0
@56     integer_cst     type: @35       int: -1
@57     identifier_node strg: n         lngt: 1
@58     identifier_node strg: f         lngt: 1
```

准备如下两个文件

pre.awk

```
#! /usr/bin/gawk -f
/^[^;]/{
    gsub(/^@/, "~@", $0);
    gsub(/( *):( *)/, ":", $0);
    print;
}
```

treeviz.awk

```
#! /usr/bin/gawk -f
#https://blog.csdn.net/l919898756
BEGIN {RS = "~@"; printf "digraph G {\n node [shape = record];";}
/^[0-9]/{
s = sprintf("%s [label = \"{%s | {", $1, $1);
for(i = 2; i < NF - 1; i++)
    s = s sprintf("%s | ", $i);
    s = s sprintf("%s}}\"];\n", $i);
    $0 = s;
    while (/([a-zA-Z]+):@([0-9]+)/){
        format = sprintf("\\1 \\3\n %s:\\1 -> \\2;", $1);
        $0 = gensub(/([a-zA-Z]+):@([0-9]+)(.*)$/, format, "g");
    };
    printf " %s\n", $0;
}
```

```
END {print "}"}
```

执行命令：./pre.awk test.c.* | ./treeviz.awk > tree.dot，得到语法树的 dot 文件
再执行命令：dot -Tpng tree.dot -o tree.png，得到 AST 如图6：

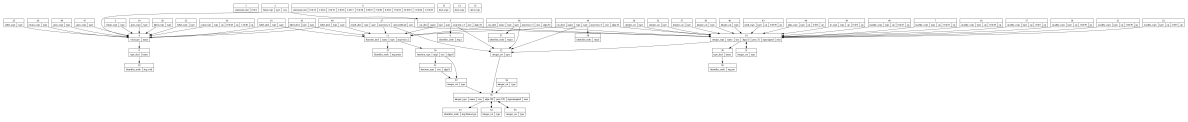

图 6: AST

5. 语义分析

使用命令：clang -fsyntax-only -Xclang -print-decl-contexts test1.cpp 进行语义分析，忽略前面的 include 结果，得到主体的分析结果为：

```
[translation unit] 0x55d485830438
        <typedef> __int128_t
        <typedef> __uint128_t
        <typedef> __NSConstantString
        <typedef> __builtin_ms_va_list
        <typedef> __builtin_va_list
        <using directive>
        [function] main()
           <var> i
           <var> n
           <var> f
```

6. 中间代码生成

使用命令：clang -S -emit-llvm test1.cpp 生成 IR，得到 test1.ll 文件。

```
; ModuleID = 'test1.cpp'
source_filename = "test1.cpp"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

%"class.std::ios_base::Init" = type { i8 }
%"class.std::basic_ostream" = type { i32 (...)**, %"class.std::basic_ios" }
%"class.std::basic_ios" = type { %"class.std::ios_base", %"class.std::
    basic_ostream"*, i8, i8, %"class.std::basic_streambuf"*, %"class.std::
    ctype"*, %"class.std::num_put"*, %"class.std::num_get"* }
%"class.std::ios_base" = type { i32 (...)**, i64, i64, i32, i32, i32, %"
    struct.std::ios_base::_Callback_list"*, %"struct.std::ios_base::_Words",
    [8 x %"struct.std::ios_base::_Words"], i32, %"struct.std::ios_base::
    _Words"*, %"class.std::locale" }
%"struct.std::ios_base::_Callback_list" = type { %"struct.std::ios_base::
    _Callback_list"*, void (i32, %"class.std::ios_base"*, i32)*, i32, i32 }
%"struct.std::ios_base::_Words" = type { i8*, i64 }
%"class.std::locale" = type { %"class.std::locale::_Impl"* }
```

```
%"class.std::locale::_Impl" = type { i32, %"class.std::locale::facet"**, i64, %"class.std::locale::facet"**, i8** }
%"class.std::locale::facet" = type <{ i32 (...)**, i32, [4 x i8] }>
%"class.std::basic_streambuf" = type { i32 (...)**, i8*, i8*, i8*, i8*, i8*, i8*, %"class.std::locale" }
%"class.std::ctype" = type <{ %"class.std::locale::facet.base", [4 x i8], %struct.__locale_struct*, i8, [7 x i8], i32*, i32*, i16*, i8, [256 x i8], [256 x i8], i8, [6 x i8] }>
%"class.std::locale::facet.base" = type <{ i32 (...)**, i32 }>
%struct.__locale_struct = type { [13 x %struct.__locale_data*], i16*, i32*, i32*, [13 x i8*] }
%struct.__locale_data = type opaque
%"class.std::num_put" = type { %"class.std::locale::facet.base", [4 x i8] }
%"class.std::num_get" = type { %"class.std::locale::facet.base", [4 x i8] }

@_ZStL8__ioinit = internal global %"class.std::ios_base::Init" zeroinitializer, align 1
@__dso_handle = external hidden global i8
@_ZSt4cout = external global %"class.std::basic_ostream", align 8
@llvm.global_ctors = appending global [1 x { i32, void ()*, i8* }] [{ i32, void ()*, i8* } { i32 65535, void ()* @_GLOBAL__sub_I_test1.cpp, i8* null }]

; Function Attrs: noinline uwtable
define internal void @__cxx_global_var_init() #0 section ".text.startup" {
  call void @_ZNSt8ios_base4InitC1Ev(%"class.std::ios_base::Init"* @_ZStL8__ioinit)
  %1 = call i32 @__cxa_atexit(void (i8*)* bitcast (void (%"class.std::ios_base::Init"*)* @_ZNSt8ios_base4InitD1Ev to void (i8*)*), i8* getelementptr inbounds (%"class.std::ios_base::Init", %"class.std::ios_base::Init"* @_ZStL8__ioinit, i32 0, i32 0), i8* @__dso_handle) #3
  ret void
}

declare void @_ZNSt8ios_base4InitC1Ev(%"class.std::ios_base::Init"*) unnamed_addr #1

; Function Attrs: nounwind
declare void @_ZNSt8ios_base4InitD1Ev(%"class.std::ios_base::Init"*) unnamed_addr #2

; Function Attrs: nounwind
declare i32 @__cxa_atexit(void (i8*)*, i8*, i8*) #3

; Function Attrs: noinline norecurse optnone uwtable
define i32 @main() #4 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
```

```
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 10, i32* %3, align 4
  store i32 2, i32* %2, align 4
  store i32 1, i32* %4, align 4
  br label %5

; <label>:5:                                      ; preds = %9, %0
  %6 = load i32, i32* %2, align 4
  %7 = load i32, i32* %3, align 4
  %8 = icmp sle i32 %6, %7
  br i1 %8, label %9, label %15

; <label>:9:                                      ; preds = %5
  %10 = load i32, i32* %4, align 4
  %11 = load i32, i32* %2, align 4
  %12 = mul nsw i32 %10, %11
  store i32 %12, i32* %4, align 4
  %13 = load i32, i32* %2, align 4
  %14 = add nsw i32 %13, 1
  store i32 %14, i32* %2, align 4
  br label %5

; <label>:15:                                     ; preds = %5
  %16 = load i32, i32* %4, align 4
  %17 = call dereferenceable(272) %"class.std::basic_ostream"* @_ZNSolsEi(%"
      class.std::basic_ostream"* @_ZSt4cout, i32 %16)
  %18 = call dereferenceable(272) %"class.std::basic_ostream"*
      @_ZNSolsEPFRSoS_E(%"class.std::basic_ostream"* %17, %"class.std::
      basic_ostream"* (%"class.std::basic_ostream"*)*
      @_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_)
  ret i32 0
}

declare dereferenceable(272) %"class.std::basic_ostream"* @_ZNSolsEi(%"class.
    std::basic_ostream"*, i32) #1

declare dereferenceable(272) %"class.std::basic_ostream"* @_ZNSolsEPFRSoS_E
    (%"class.std::basic_ostream"*, %"class.std::basic_ostream"* (%"class.std
    ::basic_ostream"*)*) #1

declare dereferenceable(272) %"class.std::basic_ostream"*
    @_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_(%"class.std::
    basic_ostream"* dereferenceable(272)) #1

; Function Attrs: noinline uwtable
define internal void @_GLOBAL__sub_I_test1.cpp() #0 section ".text.startup" {
```

```
86   call void @__cxx_global_var_init()
87   ret void
88 }
```

可以看见，在 main 函数主题中，出现了很多%，通过分析可以知道，这些% 加数字的形式应当是一个个变量的名称（标识符）。同时注意到，在整体里还有很多的 @，同样在 @ 后也跟着一个赋值形式的语句，由此推断出 @ 也是表示的也是变量。根据 @ 和% 出现的位置可以推断出，以% 开头的变量应当是局部变量，以 @ 开头的变量应当是全局变量。

同时在给变量赋值时，发现有一些不同的语句，如：%10 = load i32, i32* %4, align 4；可以注意到其中有 i32 语句，这个是表示变量的类型应当是 32 位的 int 变量；align 表示对齐，可以理解为当整数不够 32 位时，需要在前面补 0 使其达到 32 位。

这里对于执行速度等上也起到一定程度的优化：当确定某个数的类型时，也就确定了需要在空间中读取的长度，无需关注这个数是否真正占满规定的空间，即使是没有占满，也会通过填充无影响的 0 的方式进行占位，也就省略了需要对某个数的长度进行判断的步骤，达到了一定加速、优化作用（这里是类比数据库中，对 varchar 类型的变量或者是在硬盘上存储时需要按照一定的规则，需要在具体某个变量的值之前存放一个数来表示后面的变量的长度，由此来确定需要往后读多少位）。

利用 graphviz 工具可以将生成的.dot 文件转成.png，进行可视化。使用指令为：dot -Tpng test1.cpp.011t.cfg.dot -o cfg.png，得到 cfg 如图7所示：

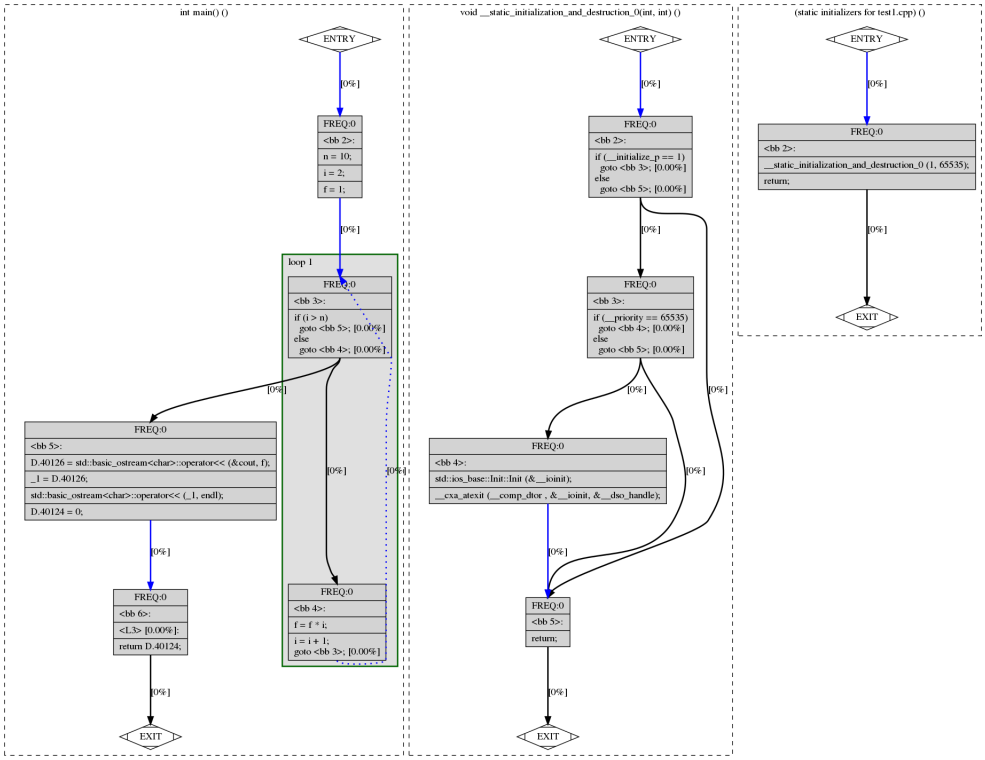

图 7: cfg

7. 机器无关优化

使用命令：llc -print-before-all -print-after-all test1.ll > a.log 2>1，得到 a.log 部分内容如下：

```
# *** IR Dump Before Shrink Wrapping analysis ***:
# Machine code for function main: NoPHIs, TracksLiveness, NoVRegs
Frame Objects:
  fi#0: size=4, align=4, at location [SP+8]
  fi#1: size=4, align=4, at location [SP+8]
  fi#2: size=4, align=4, at location [SP+8]
  fi#3: size=4, align=4, at location [SP+8]

%bb.0: derived from LLVM BB %0
        MOV32mi %stack.0, 1, %noreg, 0, %noreg, 0; mem:ST4[%1]
        MOV32mi %stack.2, 1, %noreg, 0, %noreg, 10; mem:ST4[%3]
        MOV32mi %stack.1, 1, %noreg, 0, %noreg, 2; mem:ST4[%2]
        MOV32mi %stack.3, 1, %noreg, 0, %noreg, 1; mem:ST4[%4]
    Successors according to CFG: %bb.1

%bb.1: derived from LLVM BB %5
    Predecessors according to CFG: %bb.0 %bb.2
        renamable %eax = MOV32rm %stack.1, 1, %noreg, 0, %noreg; mem:LD4[%2]
        CMP32rm killed renamable %eax, %stack.2, 1, %noreg, 0, %noreg,
            implicit-def %eflags; mem:LD4[%3]
        JG_1 %bb.3, implicit killed %eflags
    Successors according to CFG: %bb.3 %bb.2
```

可以看见此时对栈、寄存器的操作变得更加细致，同时在操作时使用了 mov32mi 指令，使得操作位数直接是 32 位，更加高效。

同时对实验手册中删除重定向的效果进行了测试，在删除了重定向后发现，a.log 文件依旧进行了创建，但是 log 文件为空，所有的输出都在命令行中。

8. 代码生成

由于本次实验使用的高级语言为 C++，所以在使用编译器时，使用的是 gcc 的一个前端 g++，因此使用命令：g++ test1.i -S -o test1.S，得到 x86 格式的汇编代码。

x86 格式汇编代码

```
        .file   "test1.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:
```

```
15          .cfi_startproc
16          pushq   %rbp
17          .cfi_def_cfa_offset 16
18          .cfi_offset 6, -16
19          movq    %rsp, %rbp
20          .cfi_def_cfa_register 6
21          movl    $10, -4(%rbp)
22          movl    $2, -12(%rbp)
23          movl    $1, -8(%rbp)
24  .L3:
25          movl    -12(%rbp), %eax
26          cmpl    -4(%rbp), %eax
27          jg      .L2
28          movl    -8(%rbp), %eax
29          imull   -12(%rbp), %eax
30          movl    %eax, -8(%rbp)
31          addl    $1, -12(%rbp)
32          jmp     .L3
33  .L2:
34          movl    $0, %eax
35          popq    %rbp
36          .cfi_def_cfa 7, 8
37          ret
38          .cfi_endproc
39  .LFE1493:
40          .size   main, .-main
41          .type   _Z41__static_initialization_and_destruction_0ii, @function
42  _Z41__static_initialization_and_destruction_0ii:
43  .LFB1974:
44          .cfi_startproc
45          pushq   %rbp
46          .cfi_def_cfa_offset 16
47          .cfi_offset 6, -16
48          movq    %rsp, %rbp
49          .cfi_def_cfa_register 6
50          subq    $16, %rsp
51          movl    %edi, -4(%rbp)
52          movl    %esi, -8(%rbp)
53          cmpl    $1, -4(%rbp)
54          jne     .L7
55          cmpl    $65535, -8(%rbp)
56          jne     .L7
57          leaq    _ZStL8__ioinit(%rip), %rdi
58          call    _ZNSt8ios_base4InitC1Ev@PLT
59          leaq    __dso_handle(%rip), %rdx
60          leaq    _ZStL8__ioinit(%rip), %rsi
61          movq    _ZNSt8ios_base4InitD1Ev@GOTPCREL(%rip), %rax
62          movq    %rax, %rdi
```

```
        call    __cxa_atexit@PLT
.L7:
        nop
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE1974:
        .size   _Z41__static_initialization_and_destruction_0ii, .-
            _Z41__static_initialization_and_destruction_0ii
        .type   _GLOBAL__sub_I_main, @function
_GLOBAL__sub_I_main:
.LFB1975:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $65535, %esi
        movl    $1, %edi
        call    _Z41__static_initialization_and_destruction_0ii
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE1975:
        .size   _GLOBAL__sub_I_main, .-_GLOBAL__sub_I_main
        .section        .init_array,"aw"
        .align 8
        .quad   _GLOBAL__sub_I_main
        .hidden __dso_handle
        .ident  "GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0"
        .section        .note.GNU-stack,"",@progbits
```

另使用命令：arm-linux-gnueabihf-gcc test.i -S -o test_arm.S 对 c 语言格式代码生成 arm 汇编代码，得到结果如下：

arm 格式汇编代码

```
    .arch armv7-a
        .eabi_attribute 28, 1
        .eabi_attribute 20, 1
        .eabi_attribute 21, 1
        .eabi_attribute 23, 3
        .eabi_attribute 24, 1
        .eabi_attribute 25, 1
        .eabi_attribute 26, 2
        .eabi_attribute 30, 6
        .eabi_attribute 34, 1
```

```
        .eabi_attribute 18, 4
        .file   "test.c"
        .text
        .align  1
        .global main
        .syntax unified
        .thumb
        .thumb_func
        .fpu vfpv3-d16
        .type   main, %function
main:
        @ args = 0, pretend = 0, frame = 16
        @ frame_needed = 1, uses_anonymous_args = 0
        @ link register save eliminated.
        push    {r7}
        sub     sp, sp, #20
        add     r7, sp, #0
        movs    r3, #10
        str     r3, [r7, #12]
        movs    r3, #2
        str     r3, [r7, #4]
        movs    r3, #1
        str     r3, [r7, #8]
        b       .L2
.L3:
        ldr     r3, [r7, #8]
        ldr     r2, [r7, #4]
        mul     r3, r2, r3
        str     r3, [r7, #8]
        ldr     r3, [r7, #4]
        adds    r3, r3, #1
        str     r3, [r7, #4]
.L2:
        ldr     r2, [r7, #4]
        ldr     r3, [r7, #12]
        cmp     r2, r3
        ble     .L3
        movs    r3, #0
        mov     r0, r3
        adds    r7, r7, #20
        mov     sp, r7
        @ sp needed
        ldr     r7, [sp], #4
        bx      lr
        .size   main, .-main
        .ident  "GCC: (Ubuntu/Linaro 7.5.0-3ubuntu1~18.04) 7.5.0"
        .section        .note.GNU-stack,"",%progbits
```

(三) 汇编器

使用指令：gcc test.S -c -o test.o，得到生成好的 test.o 文件，使用 Linux 自带的 objdump 工具打开 test.o，得到的汇编代码如下：

test.o

```
test.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   c7 45 fc 0a 00 00 00    movl   $0xa,-0x4(%rbp)
   b:   c7 45 f4 02 00 00 00    movl   $0x2,-0xc(%rbp)
  12:   c7 45 f8 01 00 00 00    movl   $0x1,-0x8(%rbp)
  19:   eb 0e                   jmp    29 <main+0x29>
  1b:   8b 45 f8                mov    -0x8(%rbp),%eax
  1e:   0f af 45 f4             imul   -0xc(%rbp),%eax
  22:   89 45 f8                mov    %eax,-0x8(%rbp)
  25:   83 45 f4 01             addl   $0x1,-0xc(%rbp)
  29:   8b 45 f4                mov    -0xc(%rbp),%eax
  2c:   3b 45 fc                cmp    -0x4(%rbp),%eax
  2f:   7e ea                   jle    1b <main+0x1b>
  31:   b8 00 00 00 00          mov    $0x0,%eax
  36:   5d                      pop    %rbp
  37:   c3                      retq
```

此时虽然通过工具发现其已经具有了一部分主要功能的汇编代码，但是此时的文件并不能像 exe 那样运行，同时发现在.o 文件中只有函数主体部分，缺少前面的 include 相关代码。此时如果对比一下其他完整的 exe 可执行文件，会发现少了很多内容，如：节信息、变量的声明定义等。由此猜测，此时汇编器生成的结果仅是根据源文件中的函数内容进行一个指令的转换，也就是转换成不包含任何环境的汇编代码。

(四) 连接器加载器

使用指令：gcc test.o -o test，之后同样使用 objdump 工具对生成的 test 进行反汇编，得到代码如下：

test

```
test:     file format elf64-x86-64


Disassembly of section .init:

00000000000004b8 <_init>:
 4b8:   48 83 ec 08             sub    $0x8,%rsp
```

```
 4bc:   48 8b 05 25 0b 20 00    mov    0x200b25(%rip),%rax        # 200fe8 <__gmon_start__>
 4c3:   48 85 c0                test   %rax,%rax
 4c6:   74 02                   je     4ca <_init+0x12>
 4c8:   ff d0                   callq  *%rax
 4ca:   48 83 c4 08             add    $0x8,%rsp
 4ce:   c3                      retq

Disassembly of section .plt:

00000000000004d0 <.plt>:
 4d0:   ff 35 f2 0a 20 00       pushq  0x200af2(%rip)        # 200fc8 <_GLOBAL_OFFSET_TABLE_+0x8>
 4d6:   ff 25 f4 0a 20 00       jmpq   *0x200af4(%rip)       # 200fd0 <_GLOBAL_OFFSET_TABLE_+0x10>
 4dc:   0f 1f 40 00             nopl   0x0(%rax)

Disassembly of section .plt.got:

00000000000004e0 <__cxa_finalize@plt>:
 4e0:   ff 25 12 0b 20 00       jmpq   *0x200b12(%rip)       # 200ff8 <__cxa_finalize@GLIBC_2.2.5>
 4e6:   66 90                   xchg   %ax,%ax

Disassembly of section .text:

00000000000004f0 <_start>:
 4f0:   31 ed                   xor    %ebp,%ebp
 4f2:   49 89 d1                mov    %rdx,%r9
 4f5:   5e                      pop    %rsi
 4f6:   48 89 e2                mov    %rsp,%rdx
 4f9:   48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
 4fd:   50                      push   %rax
 4fe:   54                      push   %rsp
 4ff:   4c 8d 05 aa 01 00 00    lea    0x1aa(%rip),%r8        # 6b0 <__libc_csu_fini>
 506:   48 8d 0d 33 01 00 00    lea    0x133(%rip),%rcx        # 640 <__libc_csu_init>
 50d:   48 8d 3d e6 00 00 00    lea    0xe6(%rip),%rdi        # 5fa <main>
 514:   ff 15 c6 0a 20 00       callq  *0x200ac6(%rip)        # 200fe0 <__libc_start_main@GLIBC_2.2.5>
 51a:   f4                      hlt
 51b:   0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)

0000000000000520 <deregister_tm_clones>:
 520:   48 8d 3d e9 0a 20 00    lea    0x200ae9(%rip),%rdi        # 201010 <__TMC_END__>
 527:   55                      push   %rbp
```

```
528:   48 8d 05 e1 0a 20 00     lea    0x200ae1(%rip),%rax        # 201010 <__TMC_END__>
52f:   48 39 f8                 cmp    %rdi,%rax
532:   48 89 e5                 mov    %rsp,%rbp
535:   74 19                    je     550 <deregister_tm_clones+0x30>
537:   48 8b 05 9a 0a 20 00     mov    0x200a9a(%rip),%rax        # 200fd8 <_ITM_deregisterTMCloneTable>
53e:   48 85 c0                 test   %rax,%rax
541:   74 0d                    je     550 <deregister_tm_clones+0x30>
543:   5d                       pop    %rbp
544:   ff e0                    jmpq   *%rax
546:   66 2e 0f 1f 84 00 00     nopw   %cs:0x0(%rax,%rax,1)
54d:   00 00 00
550:   5d                       pop    %rbp
551:   c3                       retq
552:   0f 1f 40 00              nopl   0x0(%rax)
556:   66 2e 0f 1f 84 00 00     nopw   %cs:0x0(%rax,%rax,1)
55d:   00 00 00

0000000000000560 <register_tm_clones>:
560:   48 8d 3d a9 0a 20 00     lea    0x200aa9(%rip),%rdi        # 201010 <__TMC_END__>
567:   48 8d 35 a2 0a 20 00     lea    0x200aa2(%rip),%rsi        # 201010 <__TMC_END__>
56e:   55                       push   %rbp
56f:   48 29 fe                 sub    %rdi,%rsi
572:   48 89 e5                 mov    %rsp,%rbp
575:   48 c1 fe 03              sar    $0x3,%rsi
579:   48 89 f0                 mov    %rsi,%rax
57c:   48 c1 e8 3f              shr    $0x3f,%rax
580:   48 01 c6                 add    %rax,%rsi
583:   48 d1 fe                 sar    %rsi
586:   74 18                    je     5a0 <register_tm_clones+0x40>
588:   48 8b 05 61 0a 20 00     mov    0x200a61(%rip),%rax        # 200ff0 <_ITM_registerTMCloneTable>
58f:   48 85 c0                 test   %rax,%rax
592:   74 0c                    je     5a0 <register_tm_clones+0x40>
594:   5d                       pop    %rbp
595:   ff e0                    jmpq   *%rax
597:   66 0f 1f 84 00 00 00     nopw   0x0(%rax,%rax,1)
59e:   00 00
5a0:   5d                       pop    %rbp
5a1:   c3                       retq
5a2:   0f 1f 40 00              nopl   0x0(%rax)
5a6:   66 2e 0f 1f 84 00 00     nopw   %cs:0x0(%rax,%rax,1)
5ad:   00 00 00

00000000000005b0 <__do_global_dtors_aux>:
```

```
5b0:   80 3d 59 0a 20 00 00    cmpb   $0x0,0x200a59(%rip)        # 201010 <__TMC_END__>
5b7:   75 2f                   jne    5e8 <__do_global_dtors_aux+0x38>
5b9:   48 83 3d 37 0a 20 00    cmpq   $0x0,0x200a37(%rip)        # 200ff8 <__cxa_finalize@GLIBC_2.2.5>
5c0:   00
5c1:   55                      push   %rbp
5c2:   48 89 e5                mov    %rsp,%rbp
5c5:   74 0c                   je     5d3 <__do_global_dtors_aux+0x23>
5c7:   48 8b 3d 3a 0a 20 00    mov    0x200a3a(%rip),%rdi        # 201008 <__dso_handle>
5ce:   e8 0d ff ff ff          callq  4e0 <__cxa_finalize@plt>
5d3:   e8 48 ff ff ff          callq  520 <deregister_tm_clones>
5d8:   c6 05 31 0a 20 00 01    movb   $0x1,0x200a31(%rip)        # 201010 <__TMC_END__>
5df:   5d                      pop    %rbp
5e0:   c3                      retq
5e1:   0f 1f 80 00 00 00 00    nopl   0x0(%rax)
5e8:   f3 c3                   repz retq
5ea:   66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)

00000000000005f0 <frame_dummy>:
5f0:   55                      push   %rbp
5f1:   48 89 e5                mov    %rsp,%rbp
5f4:   5d                      pop    %rbp
5f5:   e9 66 ff ff ff          jmpq   560 <register_tm_clones>

00000000000005fa <main>:
5fa:   55                      push   %rbp
5fb:   48 89 e5                mov    %rsp,%rbp
5fe:   c7 45 fc 0a 00 00 00    movl   $0xa,-0x4(%rbp)
605:   c7 45 f4 02 00 00 00    movl   $0x2,-0xc(%rbp)
60c:   c7 45 f8 01 00 00 00    movl   $0x1,-0x8(%rbp)
613:   eb 0e                   jmp    623 <main+0x29>
615:   8b 45 f8                mov    -0x8(%rbp),%eax
618:   0f af 45 f4             imul   -0xc(%rbp),%eax
61c:   89 45 f8                mov    %eax,-0x8(%rbp)
61f:   83 45 f4 01             addl   $0x1,-0xc(%rbp)
623:   8b 45 f4                mov    -0xc(%rbp),%eax
626:   3b 45 fc                cmp    -0x4(%rbp),%eax
629:   7e ea                   jle    615 <main+0x1b>
62b:   b8 00 00 00 00          mov    $0x0,%eax
630:   5d                      pop    %rbp
631:   c3                      retq
632:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
639:   00 00 00
63c:   0f 1f 40 00             nopl   0x0(%rax)
```

```
0000000000000640 <__libc_csu_init>:
 640:   41 57                   push   %r15
 642:   41 56                   push   %r14
 644:   49 89 d7                mov    %rdx,%r15
 647:   41 55                   push   %r13
 649:   41 54                   push   %r12
 64b:   4c 8d 25 9e 07 20 00    lea    0x20079e(%rip),%r12        # 200df0 <
    __frame_dummy_init_array_entry>
 652:   55                      push   %rbp
 653:   48 8d 2d 9e 07 20 00    lea    0x20079e(%rip),%rbp        # 200df8 <
    __init_array_end>
 65a:   53                      push   %rbx
 65b:   41 89 fd                mov    %edi,%r13d
 65e:   49 89 f6                mov    %rsi,%r14
 661:   4c 29 e5                sub    %r12,%rbp
 664:   48 83 ec 08             sub    $0x8,%rsp
 668:   48 c1 fd 03             sar    $0x3,%rbp
 66c:   e8 47 fe ff ff          callq  4b8 <_init>
 671:   48 85 ed                test   %rbp,%rbp
 674:   74 20                   je     696 <__libc_csu_init+0x56>
 676:   31 db                   xor    %ebx,%ebx
 678:   0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
 67f:   00
 680:   4c 89 fa                mov    %r15,%rdx
 683:   4c 89 f6                mov    %r14,%rsi
 686:   44 89 ef                mov    %r13d,%edi
 689:   41 ff 14 dc             callq  *(%r12,%rbx,8)
 68d:   48 83 c3 01             add    $0x1,%rbx
 691:   48 39 dd                cmp    %rbx,%rbp
 694:   75 ea                   jne    680 <__libc_csu_init+0x40>
 696:   48 83 c4 08             add    $0x8,%rsp
 69a:   5b                      pop    %rbx
 69b:   5d                      pop    %rbp
 69c:   41 5c                   pop    %r12
 69e:   41 5d                   pop    %r13
 6a0:   41 5e                   pop    %r14
 6a2:   41 5f                   pop    %r15
 6a4:   c3                      retq
 6a5:   90                      nop
 6a6:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
 6ad:   00 00 00

00000000000006b0 <__libc_csu_fini>:
 6b0:   f3 c3                   repz retq

Disassembly of section .fini:

00000000000006b4 <_fini>:
```

```
181  6b4:   48 83 ec 08          sub    $0x8,%rsp
182  6b8:   48 83 c4 08          add    $0x8,%rsp
183  6bc:   c3                   retq
```

可以发现链接后生成的文件和链接前的.o 文件差距还是很大的，链接后的文件包含的内容明显比之前多了很多，同时在代码块的第 31 行开始，对 ebp 进行了清空，由此开始进入到函数主体，之后在 115 行开始执行 main 函数。此时的程序才是一个真正完整能执行的状态。

三、 总结

在一个由高级语言编写成的源代码到可以双击执行的过程中，包含了很多子过程。整体可以分为：预处理、编译、汇编、链接。

在预处理阶段时，预处理器的工作是将源代码中的注释删除、将其中的宏等进行替换、标注行号等。从某方面来说是将源码进行压缩、简化，便于后续步骤的进行。

在编译阶段时，又能细分成很多小的阶段。首先是对预处理过后的文件进行词法分析，使得源码能够拆分成一个个更细的部分，对构成源程序的字符流进行扫描和分解，识别出单词，为之后机器能够“读懂”这些语句做一个准备；其次是进行语法分析，对之前词法分析得到的一个个单词序列组成具有一定逻辑顺序的语法树，确定这些输入串能够组成在语法上正确的程序；之后进行语义分析阶段，这一阶段是对语法分析产生的结果进行一个错误排查，看是否存在有语义错误等；在语义分析检查无误之后，需要将此时的语法树构建出一个临时的中间代码，使得其从图像中树的意义上回归到程序的意义上，形成一个虽然无法执行，但是具有逻辑顺序的中间代码；中间代码生成结束后是不会直接将其进行移植、形成目标代码的，而是会对其进行检查，看是否存在可以优化的空间，并对其进行一定程度上的优化，使得后续生成的目标代码能够更加的精简高效；在优化阶段完成后，此时所有的准备工作都已经完成，这个时候就到了最后一个阶段——目标代码生成，把中间代码变换成特定机器上的绝对指令代码或可重定位的指令代码或汇编指令代码。

在汇编阶段时，会对由编译阶段产生的目标代码进行转译。通常情况下，编译器生成的目标代码是面对特定平台的汇编代码，但是此时计算机依旧不认识这些代码、无法执行，此时就需要汇编器将之前得到的汇编代码翻译成计算机可以读取、执行的机器码。但是此时翻译后的机器码依旧是无法执行的，此时的机器码没有装配库函数、没有指明在装载的时候应该装载在什么位置，此时的机器码仅是对源码中的内容进行一个直接的翻译。

在链接阶段时，连接器会把目标文件和库文件链接成可执行文件。

参考文献

[1] Vincent(Hao Li). Gcc 抽象语法树 (ast) 可视化. https://blog.csdn.net/l919898756/article/details/103160501. 2019-11-20.