南开大学
Nankai University

南　　开　　大　　学

网 络 空 间 安 全 学 院

编译系统原理实验报告

Lab0　了解编译器

张丛　刘国民

年级：2021 级

专业：信息安全

指导教师：王刚

2023 年 9 月 17 日

摘要

本次实验以阶乘程序为例，详细分析了由 C/C++ 语言源代码生成可执行文件的具体过程，总体上可分为预处理、编译、汇编和链接四个部分。同时还编写和分析了 LLVM IR 程序，熟悉了编译过程中所使用的中间语言。本次实验第一节和第二节由张丛同学负责，第三至五节由刘国民同学负责。

关键字：编译；LLVM；GCC

目录

一、 正文

(一) 第一节 预处理器

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。

若是我们自己创建的头文件则需将 <> 替换成""，此时系统会先从项目文件夹中寻找是否存在相应的头文件，其次会从库文件中寻找，再否则会抛出错误。

如图1所示的 cpp 文件仅有 15 行代码

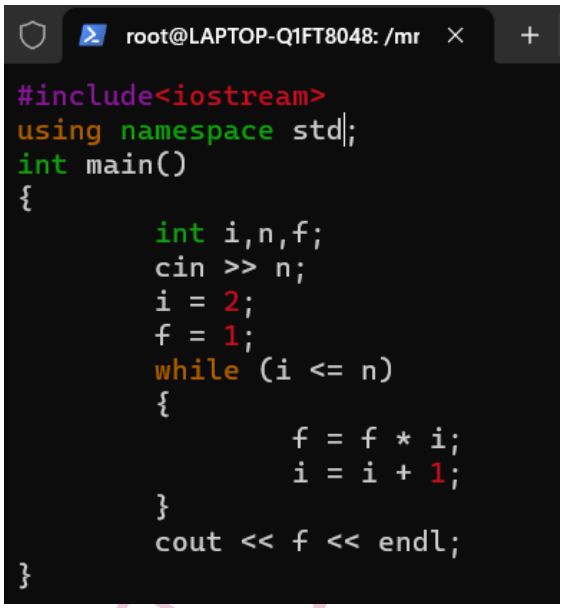

图 1: cpp 代码

在经过以下命令后得到预处理文件:

```
g++ main.cpp -E -o main.i
```

如图2, 得到的预处理文件总共有 32268 行，远多于源文件

图 2: main.i 代码

就图2来说，预处理器预处理了系统头文件和 std 命名空间的模板结构等等。

(二) 第二节 编译器

编译器进行了六个过程：词法分析、语法分析、语义分析、中间代码生成、代码优化、代码生成。

2.1 词法分析 该阶段通过分词器将源程序按一定逻辑进行分词，并且标注出分词结果中每个词相的所属逻辑类别。

使用 LLVM:

```
clang -E -Xclang -dump-tokens main.cpp
```

结果如图3:

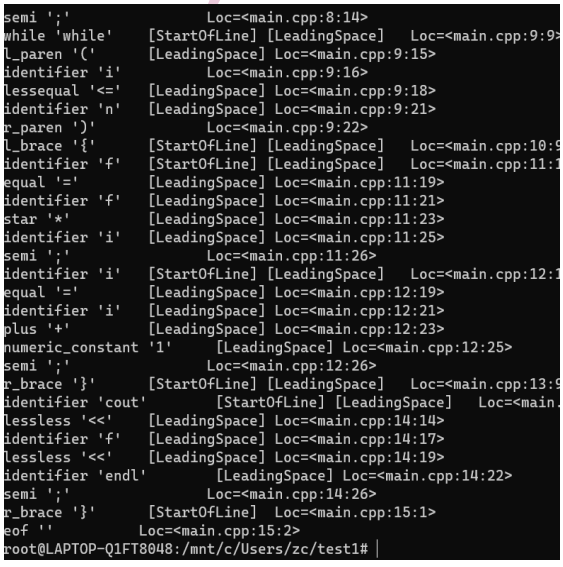

图 3: 词法分析

图3展示了一些词法单元的标识结果，并给出其属性和位置信息。

2.2 语法分析 在此阶段会将词法分析阶段产生的此法单元构建出抽象语法树。

LLVM 可以通过如下命令获得相应的 AST:

```
clang -E -Xclang -ast-dump main.cpp
```

结果如图4:

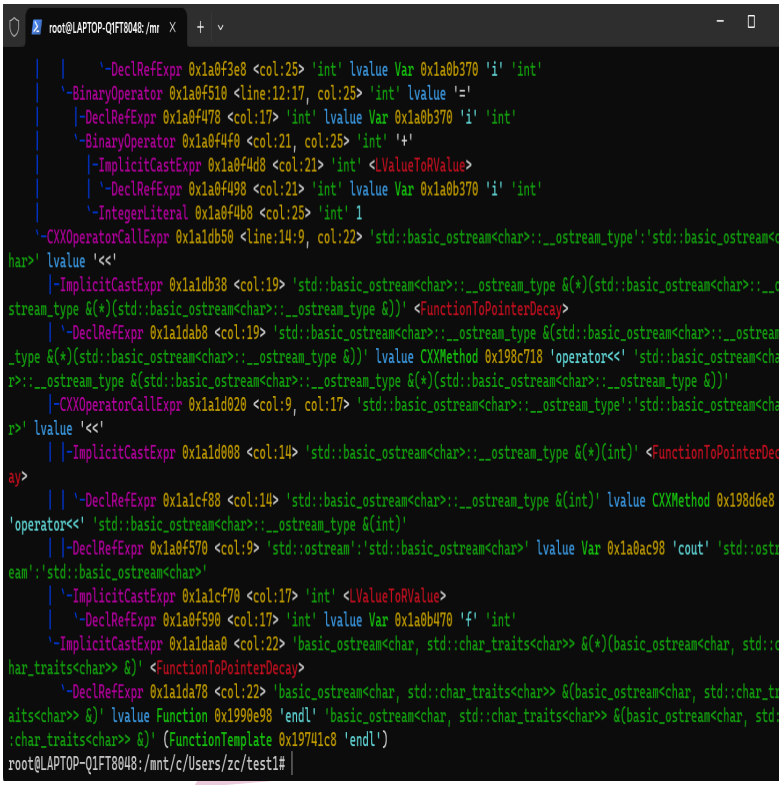

图 4: 语法单元标识结果

2.3 语义分析 使用语法树和符号表中信息来检查源程序是否与语言定义语义一致，进行类型检查等。

以下是语义分析阶段比较重要的部分:

类型检查。语义分析阶段会检查代码中的类型是否匹配，例如将整数值赋给浮点数变量、使用未声明的变量等。这有助于在编译过程中捕获一些常见的错误。

作用域分析。语义分析阶段会确定变量和标识符的可见范围，以及解析变量和函数的引用。这包括检查变量是否在正确的作用域内使用，以及解析函数调用的参数类型和返回类型。

常量折叠。语义分析阶段可能会对一些常量表达式进行求值，以便在后续的优化阶段中进行常量传播和折叠。

错误处理。语义分析阶段会检测并报告代码中的语义错误，例如重复定义的变量、函数调用参数不匹配等。编译器通常会生成有关错误的详细信息，以帮助开发人员定位和修复问题。

2.4 中间代码生成 编译器生成一个明确的低级或类机器语言的中间表示。

若想要可视化抽象语法树，以更好地分析代码依赖关系，可以先通过 g++ 获得抽象语法树的.dot 文件:

```
g++ -fdump-tree-all-graph main.cpp
```

结果如图5:

图 5: 所有.dot 文件

使用 GraphViz 可以为每个函数检索一个打印图 (png 文件)，例如:

```
dot -Tpng a-main.cpp.015t.cfg.dot -o main.png
```

结果如图6:

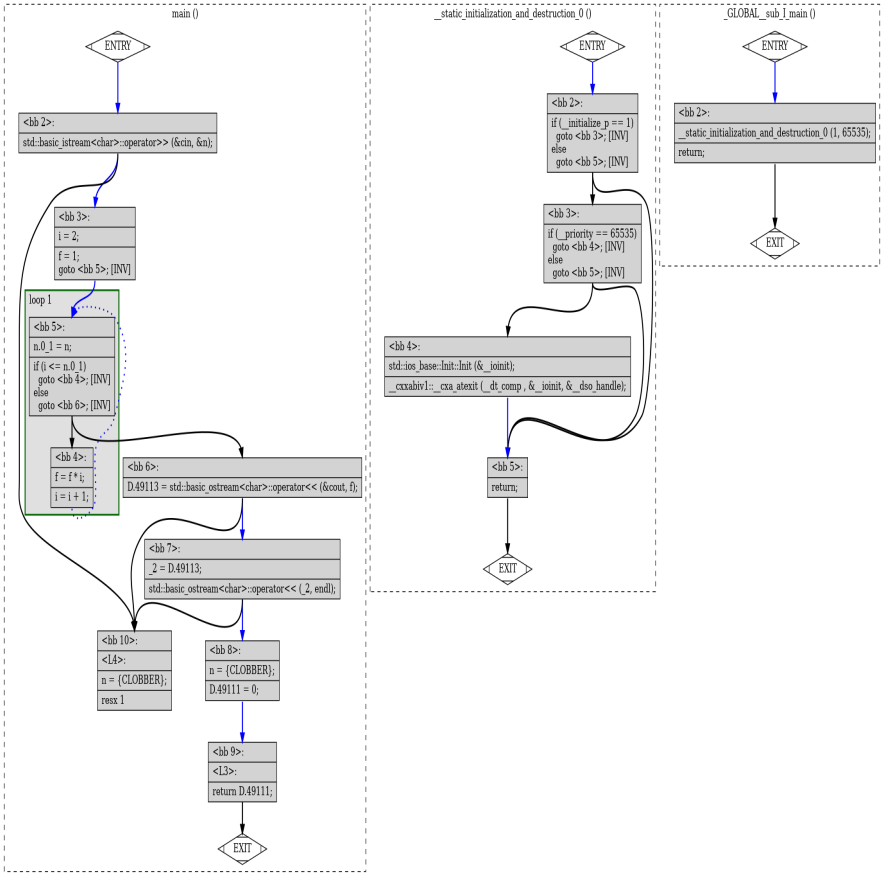

图 6: png 文件

LLVM 可以通过下面的命令生成 LLVM IR:

```
clang -S -emit-llvm main.cpp
```

结果如图7:

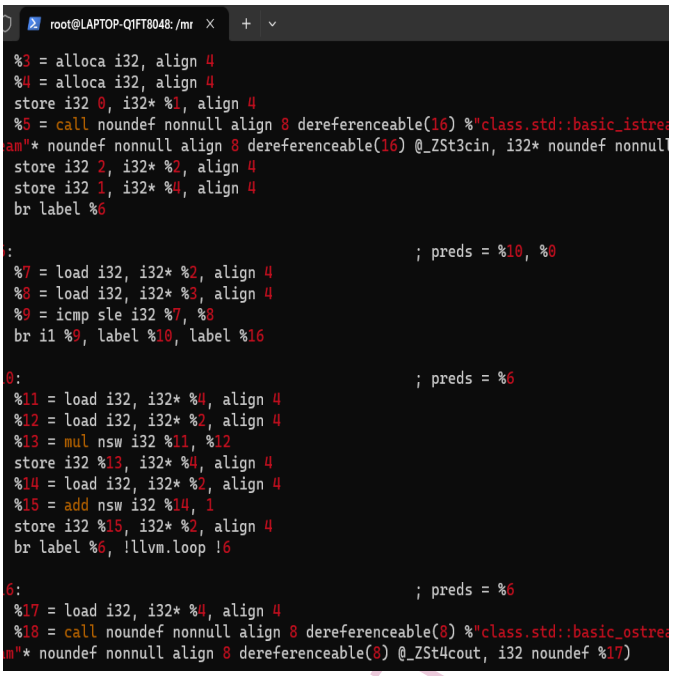

图 7: main.ll

LLVM IR 是 LLVM 编译器的中间表示形式，是一种类似汇编语言的表示形式，它包含了源代码的基本块、指令、变量和函数等信息，使得 LLVM 编译器可以对其进行各种优化和转换。

图7展示了一些函数声明和具体实现。

2.5 代码优化 进行与机器无关的代码优化步骤改进中间代码，生成更好的目标代码

通过命令:

```
llc -print-before-all -print-after-all main.ll > main.log 2>&1
```

将生成 LLVM IR 文件”main.ll” 的编译过程中的调试输出，并将结果保存到”log” 文件中。生成的”log” 文件将包含 LLC 工具在编译 LLVM IR 文件期间的调试信息。这些信息可以用于分析和调试编译过程中的问题，如了解 LLVM IR 的转换和优化过程、查看生成的机器码等。

结果如图8:

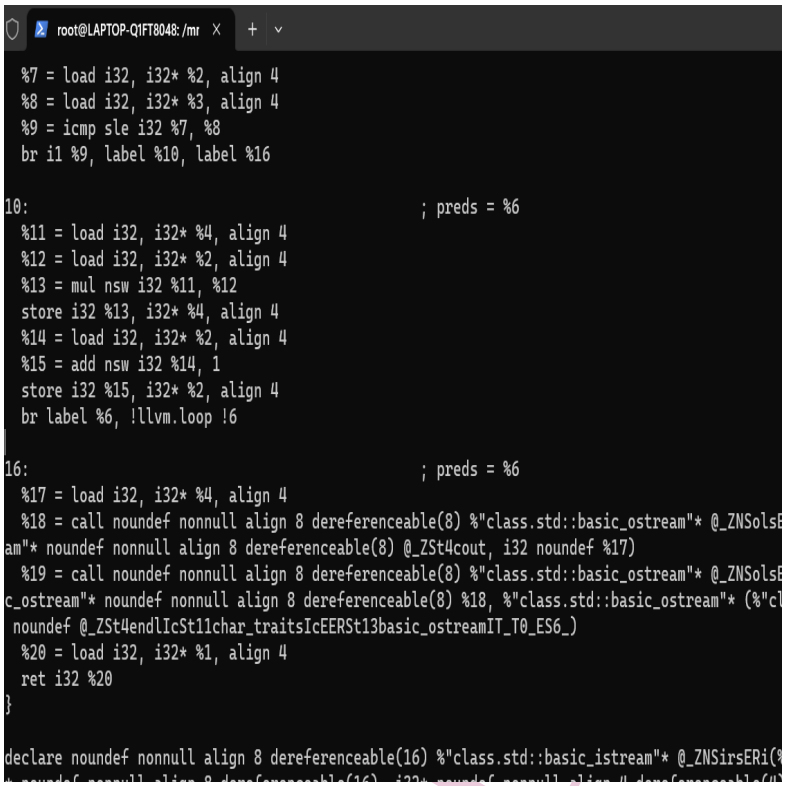

图 8: main.log

可以通过命令:

```
llvm-as main.ll -o main.bc
llvm-dis main.bc -o mian.ll
```

实现 bc 和 ll 这两种 LLVM IR 格式的互转。

得到的 main.bc 如图9:

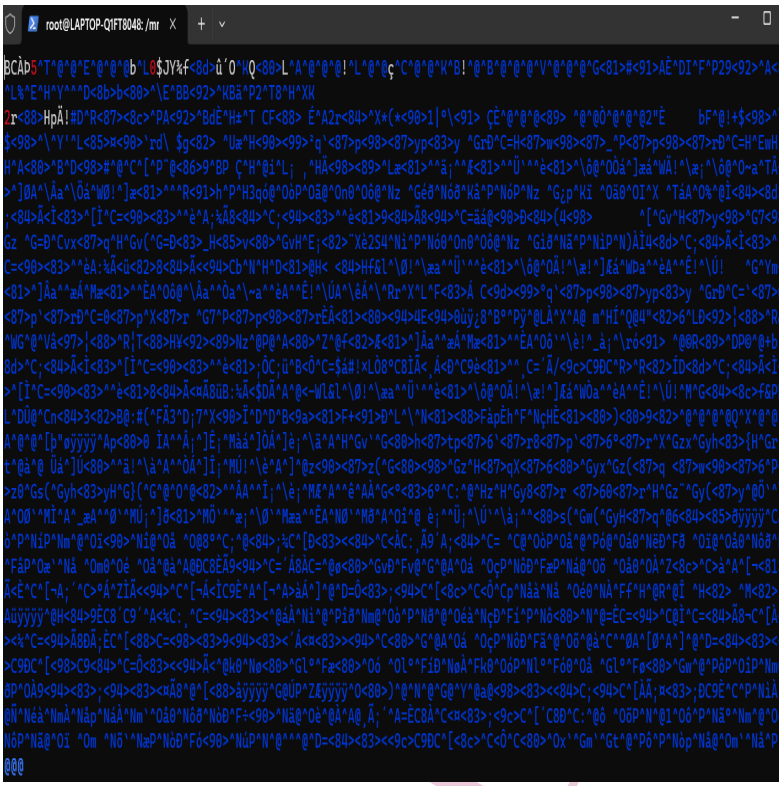

图 9: main.bc

2.6 代码生成 以中间表示形式作为输入，将其映射到目标语言。

使用命令:

```
g++ -S -o main.S main.cpp
```

得到 x86 目标格式代码，如图10:

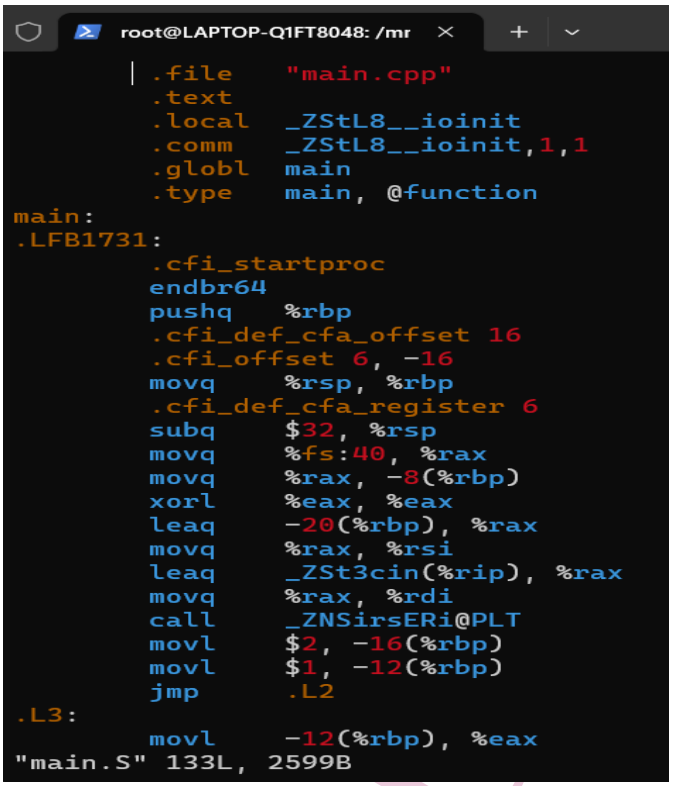

图 10: main-x86.S

使用命令:

```
arm-linux-gnueabihf-g++ main.cpp -S -o main_arm.S
```

得到 arm 目标格式代码，如图11:

图 11: main-arm.S

使用命令:

```
llc main.ll -o main.S
```

LLVM 生成目标代码，如图12:

图 12: main-llvm.S

在代码生成阶段的 x86、ARM 和 LLVM 文件的区别主要体现在指令集架构、指令集、编译器平台、代码优化和性能功耗等方面。选择适合特定需求的目标平台和编译器平台，能获得最佳的性能和效率。

(三) 第三节 汇编器

汇编器实现了从汇编语言翻译成目标机器指令的过程，最终生成的是可重定位的机器码。

在 Linux 终端上，通过以下命令得到生成的.o 目标代码文件：

```
gcc -c jiechen.c -o jiechen_x86.o
```

其中 jiechen.c 和 jiechen_x86.o 是文件名

jiechen.c 源代码如下：

```
#include<stdio.h>
int main(){
    int i, n, f;
    scanf("%d",&n);
    i = 2;
    f = 1;
    while (i <= n) {
        f = f * i;
        i = i + 1;
    }
    printf("%d",f);
    return 0;
```

```
}
```

通过 objdump 工具可以获得目标机器代码的相关内容，输入以下命令得到所有节（section）的信息：

```
objdump -h jiechen_x86.o
```

得到如下结果：

```
jiechen_x86.o:     文件格式 elf64-x86-64

节：
Idx Name          Size      VMA               LMA               File off  Algn
0 .text           00000090  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
1 .data           00000000  0000000000000000  0000000000000000  000000d0  2**0
                  CONTENTS, ALLOC, LOAD, DATA
2 .bss            00000000  0000000000000000  0000000000000000  000000d0  2**0
                  ALLOC
3 .rodata         00000003  0000000000000000  0000000000000000  000000d0  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
4 .comment        0000002c  0000000000000000  0000000000000000  000000d3  2**0
                  CONTENTS, READONLY
5 .note.GNU-stack 00000000  0000000000000000  0000000000000000  000000ff
    2**0
                  CONTENTS, READONLY
6 .note.gnu.property 00000020  0000000000000000  0000000000000000  00000100
    2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
7 .eh_frame       00000038  0000000000000000  0000000000000000  00000120  2**3
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
```

字段具体含义：

Idx：节的索引号

Name：节的名称

Size：节的大小（以字节为单位）

VMA (Virtual Memory Address)：节在虚拟内存中的地址

LMA (Load Memory Address)：节在链接时的内存地址

File off：节在文件中的偏移量（以字节为单位）

Algn：节的对齐方式

CONTENTS,ALLOC,LOAD 等为每一个节所具有的属性. 节具体含义：

.text：保存程序的可执行指令代码，即程序的代码区

.data：存储程序的初始化全局变量和静态变量

.bss：存储程序的未初始化全局变量和静态变量

.rodata：包含了程序的只读数据，如字符串常量

.comment：包含了一些编译器的元数据或注释信息

.note.GNU-stack：含有关于目标文件堆栈的特性的信息，例如是否堆栈是可执行的。

.note.gnu.property：包含了一些 GNU 特有的属性信息

.eh_frame：包含了用于异常处理的信息。

由实验结果可以看到，由于 i,n,f 等变量声明在 main 函数中，属于局部变量，保存在栈区，故源程序未声明全局变量，data 节和 bss 节大小为 0 字节。下面尝试在源代码中声明全局变量和常量，重新生成目标代码文件，观察上述三节大小是否变化

对代码（jiechen.c）进行修改：

阶乘

```
#include<stdio.h>
char b ;
int c = 1122;
const int a = 0 ;
int main(){
    int i, n, f;
    scanf("%d",&n);
    i = 2;
    f = 1;
    while (i <= n) {
        f = f * i;
        i = i + 1;
    }
    printf("%d",f);
    return 0;
 }
```

然后通过 objdump 重新查看.bss、.data 和.rodata 节的相关信息：

```
Idx Name          Size      VMA               LMA               File off  Algn
1 .data         00000004  0000000000000000  0000000000000000  000000d0  2**0
                  CONTENTS, ALLOC, LOAD, DATA
2 .bss          00000001  0000000000000000  0000000000000000  000000d0  2**0
                  ALLOC
3 .rodata       00000007  0000000000000000  0000000000000000  000000d0  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
```

可以看到，声明的字符型变量 b 为全局变量，且未完成初始化，故分配在 bss 节中，该节的大小变为 0x00000001，而整形常量 a 初始化为 0，分配在 rodata 节中，该节的大小变为 0x00000007，在原有基础上增加 4 字节，整形变量 c 初始化为 0，分配在 data 节，data 节增加 4 字节，符合实验预期。

同时从实验结果看到，所有节的 VMA 此时均为 0，这是由于目标文件还没有被链接。在链接阶段，链接器会将所有的目标文件和库文件合并到一个单一的可执行文件或库文件中，为每个节分配适当的地址。此时每个节在最终的可执行文件中会有一个唯一的非零的虚拟内存地址。

（四） 第四节 链接器

链接器（linker）负责将多个编译产生的对象文件（object files）和库（libraries）合并成一个单一的可执行文件（executable file）或库文件。通过以下命令将目标代码文件链接成可执行文

件（ELF）：

```
gcc -o jiechen_x86 jiechen_x86.o
```

然后在终端打开可执行文件，键入阶乘数字 4，得到结果 24，说明该文件成功正确执行。再通过 objdump 查看文件节的相关信息（只截取 15-26 节信息）：

```
Idx Name          Size      VMA               LMA               File off  Algn
15 .text         00000179  00000000000010a0  00000000000010a0  000010a0  2**4
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
16 .fini         0000000d  000000000000121c  000000000000121c  0000121c  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
17 .rodata       00000007  0000000000002000  0000000000002000  00002000  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
18 .eh_frame_hdr 00000034  0000000000002008  0000000000002008  00002008  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
19 .eh_frame     000000ac  0000000000002040  0000000000002040  00002040  2**3
                 CONTENTS, ALLOC, LOAD, READONLY, D1ATA
20 .init_array   00000008  0000000000003da8  0000000000003da8  00002da8  2**3
                 CONTENTS, ALLOC, LOAD, DATA
21 .fini_array   00000008  0000000000003db0  0000000000003db0  00002db0  2**3
                 CONTENTS, ALLOC, LOAD, DATA
22 .dynamic      000001f0  0000000000003db8  0000000000003db8  00002db8  2**3
                 CONTENTS, ALLOC, LOAD, DATA
23 .got          00000058  0000000000003fa8  0000000000003fa8  00002fa8  2**3
                 CONTENTS, ALLOC, LOAD, DATA
24 .data         00000010  0000000000004000  0000000000004000  00003000  2**3
                 CONTENTS, ALLOC, LOAD, DATA
25 .bss          00000008  0000000000004010  0000000000004010  00003010  2**0
                 ALLOC
26 .comment      0000002b  0000000000000000  0000000000000000  00003010  2**0
                 CONTENTS, READONLY
```

此时发现 VMA 已经非 0，为实际虚拟内存地址。这是因为在链接过程中完成了代码重定位。链接器修改目标文件中的代码和数据，使其对应于分配的地址。其中包括更新指令和数据中的地址引用，以使它们指向正确的位置。确定地址：确定每个符号（如函数和全局变量）在最终程序中的实际地址；更新地址引用：更新代码和数据段中的所有地址引用，使其指向正确的实际地址。

同时发现可执行文件相较于.o 目标代码文件，节的数量有所增加。部分节（如 text 和 data）的大小也增加了。这是因为，运行和启动代码时，为了使程序能够运行，链接器会添加一些额外的代码和数据。这会增加新的节和增加现有节的大小。如：gnu.version 节存储版本控制的相关信息。init 节中存储程序初始化代码。通过以下命令查看可执行文件的反汇编代码：

```
objdump -d jiechen_x86
```

截取以下内容：

```
0000000000001000 <_init>:
00000000000010a0 <_start>:
    10a0:       f3 0f 1e fa             endbr64
    10a4:       31 ed                   xor    %ebp,%ebp
    10a6:       49 89 d1                mov    %rdx,%r9
```

```
    10a9:       5e                      pop    %rsi
    10aa:       48 89 e2                mov    %rsp,%rdx
    10ad:       48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
    10b1:       50                      push   %rax
    10b2:       54                      push   %rsp
    10b3:       45 31 c0                xor    %r8d,%r8d
    10b6:       31 c9                   xor    %ecx,%ecx
    10b8:       48 8d 3d ca 00 00 00    lea    0xca(%rip),%rdi        # 1189
        <main>
    10bf:       ff 15 13 2f 00 00       call   *0x2f13(%rip)
                                        # 3df8<__libc_start_main@GLIBC_2.34>
    10c5:       f4                      hlt
    10c6:       66 2e 0f 1f 84 00 00    cs nopw 0x0(%rax,%rax,1)
    10cd:       00 00 00
0000000000001189 <main>:
  # …… 中间代码省略
1218:    c3                      ret    # <main>函数结束地址
```

程序执行流程：由上可以看到，程序最开始执行 init 节中的代码，完成程序启动前的初始化。然后进入 0x10a0，即 Entry Point 执行启动函数，启动函数最后会调用 main 函数，从而开始执行 main 函数内容。

同时，程序调用动态链接库时，会用 dynamic 节保存相关信息。如果是多个.o 文件一起链接成可执行文件，链接器还会将 text 这样的节合并成一个更大的 text 节。如果多个.o 文件中存在符号引用，即解析在一个对象文件中定义而在另一个对象文件中引用的符号（如函数或全局变量）。ELF 文件的首部有 ELF 文件头（类似于 Windows 系统下的 PE 文件头），用于描述文件的总体信息，如：版本，文件类型，机器类型等。

再通过 objdump 反汇编 jiechen_x86.o 发现，反汇编代码中只有 <main> 内容，且大小为 0x8f, 刚好等于可执行文件中 <main> 的大小：0x1218 -0x1189=0x8f，同时反汇编代码未发生改变，即说明链接过程中 main 函数大小和内容未改变。链接实质上是将目标文件、版本信息、程序初始化内容、链接加载动态链接库和文件头等封装在了一起。

（五） 第五节 LLVM 编程

下面对斐波拉契数列程序尝试使用 LLVM 中间语言来实现。直接编写较为困难，结合.c 编译出的.ll 文件一起理解。

```
define dso_local i32 @main() #0 {
  ;需要声明5个变量，同时需要一个恒为0的寄存器，故一共声明6个虚拟寄存器
  %1 = alloca i32, align 4      ;为%1变量分配32bit的空间
  %2 = alloca i32, align 4      ;声明int a
  %3 = alloca i32, align 4      ;声明int b
  %4 = alloca i32, align 4      ;声明int i
  %5 = alloca i32, align 4      ;声明int t
  %6 = alloca i32, align 4      ;声明int n
  store i32 0, i32* %1, align 4       ;恒0寄存器
```

```
  store i32 0, i32* %2, align 4        ;a=0
  store i32 1, i32* %3, align 4        ;b=1
  store i32 1, i32* %4, align 4        ;i=1

  ;调用scanf函数，第二个参数即为%6，即变量n
  %7 = call i32 (i8*, ...) @__isoc99_scanf(i8* noundef getelementptr inbounds
       ([3 x i8], [3 x i8]* @.str, i64 0, i64 0), i32* noundef %6)

  ;为下一步printf做准备
  %8 = load i32, i32* %3, align 4
  ;调用printf
  %9 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([4 x
      i8], [4 x i8]* @.str.1, i64 0, i64 0), i32 noundef %8)
  br label %10    ;无条件跳转到%10

;%10对应while循环的条件判断
10:                                               ; preds = %14, %0
  %11 = load i32, i32* %4, align 4      ;将i加载到内存中
  %12 = load i32, i32* %6, align 4      ;将n加载到内存中
  %13 = icmp slt i32 %11, %12           ;用%13保存比较的布尔值
  br i1 %13, label %14, label %24       ;有条件跳转

;循环体
14:                                               ; preds = %10
  ;对应t=b
  %15 = load i32, i32* %3, align 4
  store i32 %15, i32* %5, align 4

  ;对应b=a+b
  %16 = load i32, i32* %2, align 4
  %17 = load i32, i32* %3, align 4
  %18 = add nsw i32 %16, %17            ;存储加法结果后store回变量b中
  store i32 %18, i32* %3, align 4

  ;对应printf("%d\n",b);
  %19 = load i32, i32* %3, align 4
  %20 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([4 x
      i8], [4 x i8]* @.str.1, i64 0, i64 0), i32 noundef %19)

  ;对应a=t
  %21 = load i32, i32* %5, align 4
  store i32 %21, i32* %2, align 4

  ;对应i=i+1
  %22 = load i32, i32* %4, align 4
  %23 = add nsw i32 %22, 1
  store i32 %23, i32* %4, align 4

```

```
  br label %10, !llvm.loop !6      ;无条件跳转，回到while的判断部分

24:                                                ; preds = %10
  ret i32 0      ;return 0
}
```

采用以下命令，通过 LLVM 将.ll 程序编译成目标程序，进而链接成可执行文件：

```
llvm-as fib.ll -o fib.bc                # 采用二进制形式
llc fib.bc -filetype=obj -o fib.o       # 将LLVM编译成目标代码文件
gcc -o fib fib.o                        # 将目标文件链接成可执行文件
```

打开 fib 后，输入数字 5，得到 1,1,2,3,5 序列。说明程序正确执行。

参考文献