



南开大学
Nankai University

南 开 大 学

计 算 机 学 院 和 网 络 空 间 安 全 学 院

编译系统原理第一次实验报告

了解编译器及 LLVM IR 编程

姓名：彭钰钊 姜涵

学号：2110756 2113630

年级：2021 级

专业：计算机科学与技术 信息安全

指导教师：王刚

2023 年 9 月 17 日

摘要

人与人之间的关系建立在交流之上，我们和计算机之间的关系也是如此。我们从高级程序设计语言（C++、Python、JAVA）开始学习如何与计算机进行沟通，我们大多会以“Hello World!”向计算机道出我们的第一声问候，随着学习的深入我们逐渐理解计算机为什么能够“听懂”我们的语言并向我们回以问候——汇编语言的存在成为我们使用的高级语言和机器语言转换的桥梁，那么计算机又是如何完成这样一份工作的呢？本学期的编译系统原理将向我们揭开“智慧的”计算机的面纱。事实上，一个完整的程序编译过程可以大致分为四个过程，即预处理、编译、汇编和链接。本文将以 Linux 环境下 GCC 编译器编译 C++ 程序为例，对编译器的这四个工作流程展开深入（chu）入（bu）探索，并进行总结整理，为后续编译系统原理的深入学习打下良好的基础。

在实验分工方面，我们的第一部分工作按语言处理系统的不同阶段进行分工：由姜涵负责预处理器、编译器的部分内容，由彭钰钊负责汇编器、链接器和加载器的部分内容；第二部分工作按照 SysY 的不同语言特性进行分工：由姜涵负责整数运算和循环内容，由彭钰钊负责浮点数和分支的内容。

关键字：编译原理、GCC、预处理器、汇编器、编译器、链接器、LLVM

目录

一、 编译的完整工作过程	1
(一) 预处理器	1
1. 预处理阶段概述	1
2. 预处理阶段的一般流程	1
3. 常用预处理命令	1
4. 预处理器调用	2
(二) 编译器	3
1. 编译阶段概述	3
2. 编译阶段的一般流程	3
3. 词法分析	4
4. 语法分析	4
5. 语义分析	5
6. 中间代码生成	5
7. 代码优化	6
8. 代码生成	7
(三) 汇编器	7
1. 汇编器概述	7
2. 汇编器分析	7
3. 目标文件分析	8
4. 汇编器小结	12
(四) 链接器与加载器	13
1. 链接器概述	13
2. 链接器分析	13
3. 加载器概述	15
二、 LLVM IR 编程	1
(一) SysY 语言概述	1
(二) LLVM IR 概述	1
(三) 整数运算 & 循环分支	1
(四) 浮点数 & 条件分支——学生成绩等级划分程序	3

一、 编译的完整工作过程

(一) 预处理器

1. 预处理阶段概述

预处理是编译过程的第一个阶段，是编译器在实际编译源代码前进行的一系列文本处理操作。预处理阶段的主要目的是对源代码进行一些文本替换和宏展开等操作，以准备好供编译器进一步处理的代码。

预处理阶段的结果是一个经过预处理的源代码文件，这个文件会作为输入传递给编译器的下一个阶段进行进一步的处理和编译。但需要注意，预处理阶段是在编译过程的早期阶段进行的，它只是对源代码进行一些文本处理操作，并没有进行语法检查和语义分析。这意味着一些预处理错误可能会导致编译错误或警告，但并不会在预处理阶段中被发现，需要在后续编译过程中解决。

2. 预处理阶段的一般流程

(1) 头文件包含：编译过程中的源代码通常会使用一些外部定义的功能，这些功能通常由其他文件或库提供。在预处理阶段，所有的头文件指令（如 `#include`）会被替换为对应的头文件内容。

(2) 宏展开：在预处理阶段，可以用 `#define` 定义的宏会被展开为相应的文本。编译器会逐个查找代码中的宏，并将其替换为宏定义中指定的文本。

(3) 条件编译：预处理阶段还会根据条件判断指令（如 `#if`、`#ifdef`、`#ifndef`）来决定编译哪些代码块。根据条件判断的结果，可以选择性地包含或排除代码。(4) 去除注释：注释是用于代码注释和解释的，预处理阶段会将注释从源代码中删除。注释的删除有助于减少编译时的字符数量，提高编译效率。

(5) 常量替换：在预处理阶段，编译器还会将代码中的常量（如数值常量、字符串常量）进行替换，这样可以提高代码的可读性。

(6) 其他预处理指令的处理：预处理阶段还会处理一些特殊的预处理指令 `#error`（生成编译错误信息）、`#warning`（生成编译警告信息）等。

3. 常用预处理命令

预处理的命令通常以 `#` 开头，独占一行，`#` 前不能有非空白符之外的符号。常用的预处理命令：

`#include`：用于包含头文件。

`#define`：用于定义宏。

`#ifdef` / `#ifndef`：用于条件编译。`#if` / `#elif` / `#else` /

`#endif`：用于条件编译。根据表达式的值来决定是否编译某段代码。

`#undef`：用于取消宏定义。

`#warning`：用于生成编译警告信息。

`#error`：用于生成编译错误信息。

`#pragma`：用于向编译器发出特定指令。

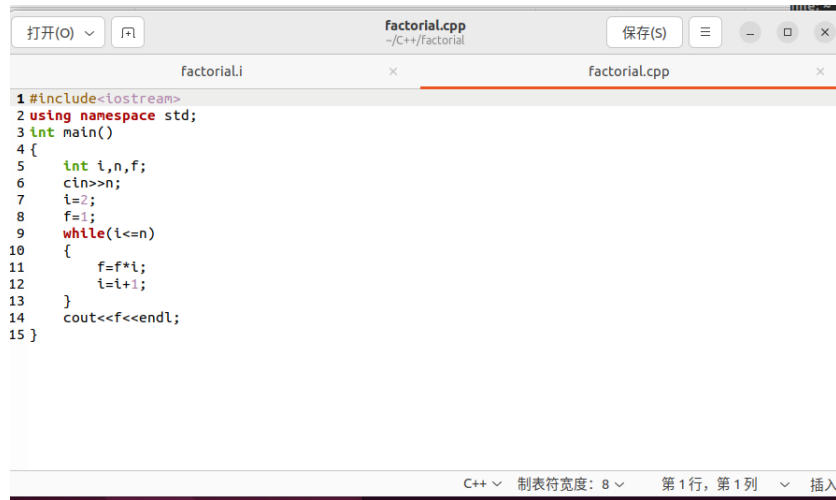
`#line`：用于指定当前行号和源文件名。

`#error`：用于生成编译错误信息。

通过使用这些预处理命令，可以使编译过程更加灵活和可控。

4. 预处理器调用

本次实验用采用 g++ 调用预处理器 (命令: g++ factorial.cpp -E -o factorial.i), 处理文件 factorial.cpp 得到 factorial.i:



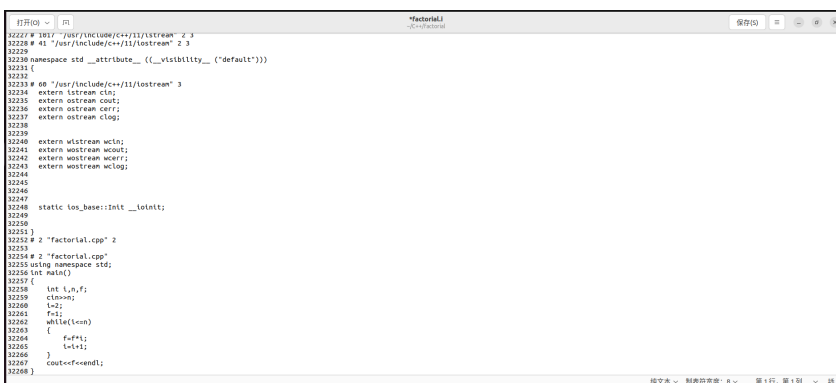
```
1 #include<iostream>
2 using namespace std;
3 int main()
4 {
5     int i,n,f;
6     cin>>n;
7     i=2;
8     f=1;
9     while(i<=n)
10    {
11        f=f*i;
12        i=i+1;
13    }
14    cout<<f<<endl;
15 }
```

图 1: 源程序 factorial.cpp



```
1 # 0 "factorial.cpp"
2 # 0 "duilib.h"
3 # 0 "command-line.h"
4 # 0 "usr/include/stdc-predef.h" 1 3 4
5 # 0 "command-line.h" 2
6 # 0 "factorial.cpp"
7 # 1 "usr/include/c++/11/iostream" 1 3
8 # 0 "usr/include/c++/11/iostream" 3
9
10 # 37 "usr/include/c++/11/iostream" 3
11
12 # 1 "usr/include/x86_64-linux-gnu/c++/11/bits/c++config.h" 1 3
13 # 278 "usr/include/x86_64-linux-gnu/c++/11/bits/c++config.h" 3
14
15 # 278 "usr/include/x86_64-linux-gnu/c++/11/bits/c++config.h" 3
16 namespace std
17 {
18     typedef long unsigned int size_t;
19     typedef long int ptrdiff_t;
20
21
22     typedef decltype(nullptr) nullptr_t;
23
24
25 # 300 "usr/include/x86_64-linux-gnu/c++/11/bits/c++config.h" 3
26 namespace std
27 {
28     inline namespace __cxx11 __attribute__((__abi_tag__ ("cxx11"))) {
29
30 namespace __gnu_cxx
31 {
32     inline namespace __cxx11 __attribute__((__abi_tag__ ("cxx11"))) {
33
34 # 386 "usr/include/x86_64-linux-gnu/c++/11/bits/c++config.h" 3
35 # 1 "usr/include/x86_64-linux-gnu/c++/11/bits/os_defines.h" 1 3
36 # 39 "usr/include/x86_64-linux-gnu/c++/11/bits/os_defines.h" 3
37 # 1 "usr/include/features.h" 1 3 4
38 # 392 "usr/include/features.h" 3 4
39 # 1 "usr/include/features-timex.h" 1 3 4
40 # 20 "usr/include/features-timex.h" 3 4
41 # 1 "usr/include/x86_64-linux-gnu/bits/wcharsize.h" 1 3 4
42 # 1 "usr/include/x86_64-linux-gnu/bits/wcharsize.h" 1 3 4
```

图 2: 预处理后的 factorial.i) 文件部分截图



```
32227 # 1 "usr/include/c++/11/iostream" 2 3
32228 # 41 "usr/include/c++/11/iostream" 2 3
32229
32230 namespace std __attribute__((__visibility__ ("default")))
3231 {
3232
3233 # 60 "usr/include/c++/11/iostream" 3
3234 extern istream cin;
3235 extern ostream cout;
3236 extern ostream cerr;
3237 extern ostream clog;
3238
3239
3240 extern wistream wcin;
3241 extern wostream wcout;
3242 extern wostream wcerr;
3243 extern wostream wclog;
3244
3245
3246
3247 static ios_base::Init __init;
3248
3249
3250
3251 }
3252 # 2 "factorial.cpp" 2
3253
3254 # 2 "factorial.cpp" 2
3255 using namespace std;
3256 int main()
3257 {
3258     int i,n,f;
3259     cin>>n;
3260     i=2;
3261     f=1;
3262     while(i<=n)
3263     {
3264         f=f*i;
3265         i=i+1;
3266     }
3267     cout<<f<<endl;
3268 }
```

图 3: 预处理后的 factorial.i) 文件部分截图

通过对比源程序和输出程序, 可以发现, 原本 15 行的代码变成了 32268 行, 这主要是因为预处理将 iostream 库文件的内容复制过来, 使得代码行数变得非常多。

(二) 编译器

1. 编译阶段概述

编译过程中的编译阶段是指在预处理完成后，将经过预处理的源代码经过词法分析、语法分析、语义分析、中间代码生成、代码优化和代码生成等几个主要步骤，最终生成高效、正确的可执行代码。

2. 编译阶段的一般流程

(1) 词法分析：在词法分析阶段，编译器会将源代码分解为词法单元 (token)，即一系列有特定意义的代码片段。这些词法单元包括关键字、标识符、运算符、分隔符等。词法分析器会通过扫描源代码文件并利用正则表达式等方式来识别和提取词法单元。

(2) 语法分析：在语法分析阶段，编译器会根据语法规则对词法单元进行组织和分析，以构建语法树 (syntax tree)。语法树表示源代码的结构和语法关系。语法分析器会使用上下文无关文法 (context-free grammar) 来解析源代码，并采用算法 (如递归下降分析、LR 分析等) 来构建语法树。

(3) 语义分析：在语义分析阶段，编译器会对语法树进行进一步推理和检查，以确定代码的语义是否符合语言规范。语义分析器会检查类型匹配、变量声明和使用、函数调用等语义规则，并进行错误检测。语义分析还包括符号表的构建，用于收集和管理变量、函数等的信息。

(4) 中间代码生成：在中间代码生成阶段，编译器会将语法树转换为一种中间表示形式，以方便后续的优化和代码生成。中间代码通常是一种抽象的、与具体机器相关性较小的表示形式。常见的中间表示有三地址码、静态单赋值形式 (SSA) 等。

(5) 代码优化：在优化阶段，编译器会对中间代码进行一系列的优化操作，以改进代码的性能和效率。优化的目标包括减少代码长度、提高代码执行速度、减少功耗等。常见的优化技术包括常量折叠、公共子表达式消除、循环优化等。

(6) 目标代码生成：在目标代码生成阶段，编译器会将经过优化的中间代码转换为机器代码或类似于目标机器的低级代码。目标代码生成器会将抽象的中间表示映射到具体机器的指令集架构，包括寄存器分配、指令选择和地址计算。生成的目标代码可以是汇编语言代码或二进制机器代码。

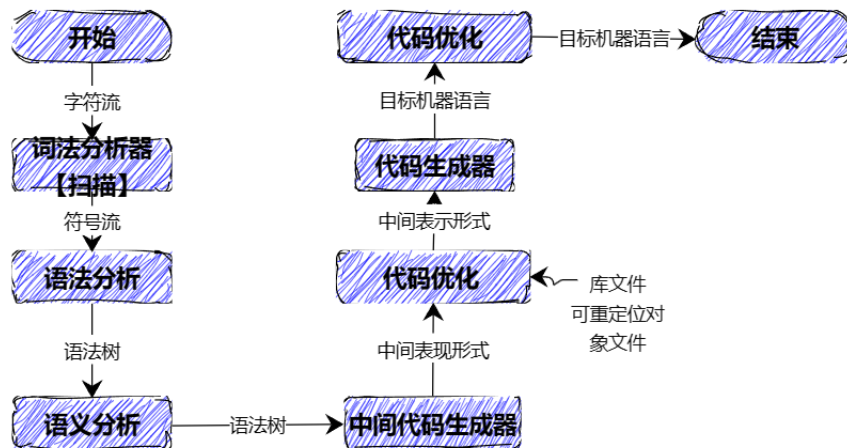


图 4: 编译器的一般流程

3. 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。分词（命令：clang -E -Xclang -dump-tokens factorial.cpp）后结果如下：

```

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

图 5: 词法分析部分输出

查看结果可以发现：程序被按照 C++ 语言逻辑正确分词，并且给予每个词对应的标识（如“lessless”“<<”），使得后续建立语法树等步骤更加方便。

4. 语法分析

该阶段使用语法分析器对词法分析阶段产生的词法单元进行语法分析，采用上下文无关文法构建出语法树。

```
| | BinaryOperator 0x167b028 <line:11:9, col:13> 'int' lvalue '='  
| | DeclRefExpr 0x167af78 <col:9> 'int' lvalue Var 0x1677040 'f' 'int'  
| | BinaryOperator 0x167b008 <col:11, col:13> 'int' '*'  
| | ImplicitCastExpr 0x167afd8 <col:11> 'int' <LValueToRValue>  
| | DeclRefExpr 0x167af98 <col:11> 'int' lvalue Var 0x1677040 'f' 'int'  
| | ImplicitCastExpr 0x167aff0 <col:13> 'int' <LValueToRValue>  
| | DeclRefExpr 0x167afb8 <col:13> 'int' lvalue Var 0x1676f40 'i' 'int'
```

图 6: 部分语法树

以图6所示部分语法树为例,可以观察到其使用”|”, ” “, ” - ” 来建立树的基本形状,将对应于源程序第 11 行的 (f=f*i) 逻辑清晰呈现。

此过程像上述片段一样构建出完整的抽象代码树将助于之后的代码生成。

5. 语义分析

该阶段利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。经过语义分析阶段过程后,整个语法树的表达式都被标识了类型,如果有些类型需要做隐式转换,语义分析程序会在语法树中插入相应的转换节点。语义分析器也会对符号表中的符号类型做更新。简单来讲这个阶段就是对运算对象进行类型检查和转换,保证程序的逻辑合法。

6. 中间代码生成

该阶段能够生成虚拟机程序,可以认为是成一个明确的低级或类机器语言的中间表示。

实验过程中通过 g++ 编译器 (-fdump-tree-all-graph 指令)生成多阶段输出,再利用 graphviz 工具进行可视化:

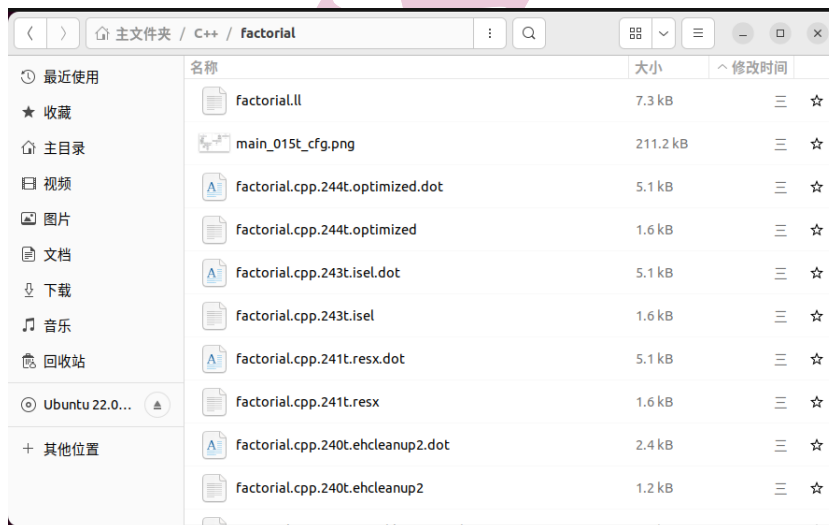


图 7: 中间代码生成多阶段部分输出

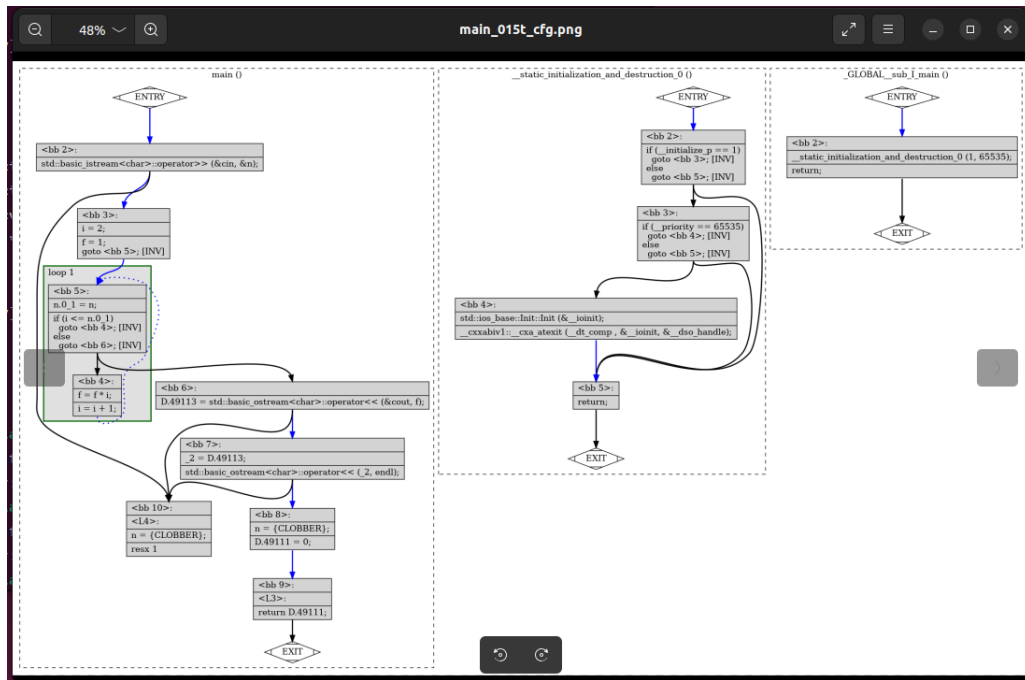


图 8: .dot 文件可视化

LLVM 生成 LLVM IR (命令: `clang -S -emit-llvm factorial.cpp`):

```

46 define dso_local noundef i32 @main() #4 {
47   %1 = alloca i32, align 4
48   %2 = alloca i32, align 4
49   %3 = alloca i32, align 4
50   %4 = alloca i32, align 4
51   store i32 0, i32* %1, align 4
52   %5 = call noundef nonnull align 8 dereferenceable(16) @"class.std::basic_istream"* @_ZN5irsER1(%"class.std::basic_istream"* noundef nonnull align 8 dereferenceable(4) %3)
53   store i32 2, i32* %2, align 4
54   store i32 1, i32* %4, align 4
55   br label %6
56
57 6:
58   %7 = load i32, i32* %2, align 4
59   %8 = load i32, i32* %3, align 4
60   %9 = icmp sle i32 %7, %8
61   br i1 %9, label %10, label %16
62
63 10:
64   %11 = load i32, i32* %4, align 4
65   %12 = load i32, i32* %2, align 4
66   %13 = mul nsw i32 %11, %12
67   store i32 %13, i32* %4, align 4
68   %14 = load i32, i32* %2, align 4
69   %15 = add nsw i32 %14, 1
70   store i32 %15, i32* %2, align 4
71   br label %6, !llvm.loop !6

```

图 9: 部分 factorial.ll 代码

7. 代码优化

该阶段会对生成的.ll 代码进行机器无关的优化, 大致分为三种: Analysis Passes、Transform Passes 和 Utility Passes。

具体优化阶段主要包括 Expand Atomic instructions、Module Verifier、Canonicalize natural loops、Merge contiguous icmps into a memcmp、Expand memcmp() to load/stores、Lower Garbage Collection Instructions、Shadow Stack GC Lowering、ower constant intrinsics、Remove unreachable blocks from the CFG、Constant Hoisting、Partially inline calls to library functions、Instrument function entry/exit with calls to e.g. mcount() (post inlining)、Scalarize Masked Memory Intrinsics、Expand reduction intrinsics、Interleaved Access Pass、Expand indirectbr

instructions、CodeGen Prepare、Exception handling preparation、Safe Stack instrumentation pass、Module Verifier

8. 代码生成

该阶段以中间表示形式作为输入，将其映射到目标语言其部分（命令：g++ factorial.i -S -o factorial.S）：

```

85 .L8:
86     nop
87     leave
88     .cfi_def_cfa 7, 8
89     ret
90     .cfi_endproc
91 .LFE2231:
92     .size    _Z41__static_initialization_and_destruction_0ii, .-_Z41__static_initialization_and_destruction_0ii
93     .type    _GLOBAL__sub_I_main, @function
94 _GLOBAL__sub_I_main:
95 .LFB2232:
96     .cfi_startproc
97     endbr64
98     pushq    %rbp
99     .cfi_def_cfa_offset 16
100    .cfi_offset 6, -16
101    movq     %rsp, %rbp
102    .cfi_def_cfa_register 6
103    movl     $65535, %esi
104    movl     $1, %edi
105    call     _Z41__static_initialization_and_destruction_0ii
106    popq     %rbp
107    .cfi_def_cfa 7, 8
108    ret
109    .cfi_endproc
110 .LFE2232:
111     .size    _GLOBAL__sub_I_main, .-_GLOBAL__sub_I_main
112     .section .init_array,"aw"
113     .align 8
114     .quad    _GLOBAL__sub_I_main
115     .hidden __dso_handle
116     .ident   "GCC: (Ubuntu 11.4.0-1ubuntu1-22.04) 11.4.0"
117     .section .note.GNU-stack,"",@progbits
118     .section .note.gnu.property,"a"
119     .align 8
120     .long    1f - 0f
121     .long    4f - 1f
122     .long    5

```

图 10: 部分 factorial.S 代码

(三) 汇编器

1. 汇编器概述

汇编器 (Assembler) 作为计算机中高级程序设计语言和机器语言之前的桥梁，用于将汇编语言代码转换为机器语言代码。汇编语言是一种相对低级的编程语言，它利用助记符来表示计算机指令和操作码，与高级程序设计语言的翻译过程不同的是，因为汇编程序的每一条语句都近似对应于一条机器指令，所以汇编器所做的就是逐句对照翻译，而不存在编译器中那些复杂的分析过程。

所以**汇编器做了什么？**

我们简单的说，汇编器起到了将高级的汇编语言代码转换为底层的机器语言代码的关键作用。

2. 汇编器分析

在 Linux 终端中输入指令 g++ -c factorial.cpp -o factorial.o；如图11所示

```
yuzhao-peng@yuzhao-peng:~/Compiler_Sys/Lab0-1$ g++ -c factorial.cpp -o factorial.o
```

图 11: 终端演示

在 Linux 终端中输入指令 file factorial.o；如图12所示，可以查看文件 factorial.o 的类型

```
yuzhao-peng@yuzhao-peng:~/Compiler_Sys/Lab0-1$ file factorial.o
factorial.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

图 12: 终端演示

现在我们得到了由 factorial.cpp 源文件经过预处理、编译和汇编之后的目标文件 factorial.o 如图13所示，可以查看文件 factorial.o 的类型，接下来我们进一步分析该文件的相关内容。

```
yuzhao-peng@yuzhao-peng:~/Compiler_Sys/Lab0-1$ hexdump -C factorial.o
00000000 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00 |.ELF.....|
00000010 01 00 3e 00 01 00 00 00 00 00 00 00 00 00 00 00 |...>.....|
00000020 00 00 00 00 00 00 00 00 58 07 00 00 00 00 00 00 |.....X.....|
00000030 00 00 00 00 40 00 00 00 00 00 40 00 0f 00 0e 00 |....@....@....|
00000040 f3 0f 1e fa 55 48 89 e5 48 83 ec 20 64 48 8b 04 |....UH..H..dH..|
00000050 25 28 00 00 00 48 89 45 f8 31 c0 48 8d 45 ec 48 |%(...H.E.1.H.E.H|
00000060 89 c6 48 8d 05 00 00 00 00 48 89 c7 e8 00 00 00 |..H.....H.....|
00000070 00 c7 45 f0 02 00 00 00 c7 45 f4 01 00 00 00 eb |..E.....E.....|
00000080 0e 8b 45 f4 0f af 45 f0 89 45 f4 83 45 f0 01 8b |..E...E...E...E...|
00000090 45 ec 39 45 f0 7e ea 8b 45 f4 89 c6 48 8d 05 00 |E.9E...E...H...|
000000a0 00 00 00 48 89 c7 e8 00 00 00 48 8b 15 00 00 |...H.....H....|
000000b0 00 00 48 89 d6 48 89 c7 e8 00 00 00 00 b8 00 00 |..H...H.....|
000000c0 00 00 48 8b 55 f8 64 48 2b 14 25 28 00 00 00 74 |..H.U.dH+.%(...t|
000000d0 05 e8 00 00 00 00 c9 c3 f3 0f 1e fa 55 48 89 e5 |.....UH..|
000000e0 48 83 ec 10 89 7d fc 89 75 f8 83 7d fc 01 75 3b |H....}.u..}.u;|
000000f0 81 7d f8 ff ff 00 00 75 32 48 8d 05 00 00 00 00 |.}.....u2H.....|
00000100 48 89 c7 e8 00 00 00 00 48 8d 05 00 00 00 00 48 |H.....H.....H|
00000110 89 c2 48 8d 05 00 00 00 00 48 89 c6 48 8b 05 00 |..H.....H..H...|
00000120 00 00 00 48 89 c7 e8 00 00 00 90 c9 c3 f3 0f |...H.....|
00000130 1e fa 55 48 89 e5 be ff ff 00 00 bf 01 00 00 00 |..UH.....|
00000140 e8 93 ff ff ff 5d c3 00 00 00 00 00 00 00 00 00 |.....].....|
00000150 00 47 43 43 3a 20 28 55 62 75 6e 74 75 20 31 31 |.GCC: (Ubuntu 11|
00000160 2e 34 2e 30 2d 31 75 62 75 6e 74 75 31 7e 32 32 |.4.0-1ubuntu1-22|
00000170 2e 30 34 29 20 31 31 2e 34 2e 30 00 00 00 00 00 |.04) 11.4.0.....|
00000180 04 00 00 00 10 00 00 00 05 00 00 00 47 4e 55 00 |.....GNU..|
00000190 02 00 00 c0 04 00 00 00 03 00 00 00 00 00 00 00 |.....|
000001a0 14 00 00 00 00 00 00 00 01 7a 52 00 01 78 10 01 |.....zR..x..|
000001b0 1b 0c 07 08 90 01 00 00 1c 00 00 00 1c 00 00 00 |.....|
000001c0 00 00 00 98 00 00 00 00 00 45 0e 10 86 02 43 0d |.....E....C..|
000001d0 06 02 8f 0c 07 08 00 00 1c 00 00 00 3c 00 00 00 |.....<...|
000001e0 00 00 00 00 55 00 00 00 00 4f 00 10 05 03 43 0d |...V...f...C..|
```

图 13: 目标文件内容 16 进制展示 (部分截图)

3. 目标文件分析

目标文件 (Object file) 是存放经过编译器和汇编器处理之后的目标代码¹的文件，也常被称为二进制文件。目标文件作为源代码文件产生程序文件的中间产物，在 Linux 系统中以.o 文件形式存在，在 Windows 系统中以.obj 文件形式存在。在上一小节中查看文件 factorial.o 的类型中可以看见，该文件属于可重定位文件 (relocatable file)，能够在链接的过程中生成可执行文件或共享目标文件。

目标文件与可执行文件的内容和结构很相似，从广义上来说，可以将目标文件和可执行文件看成一种类型的文件，linux 下统称为 ELF 文件格式²。在目标文件中，编译得到的机器指令代

¹一般由机器代码或接近于机器语言的代码组成，包含着机器代码（可直接被计算机中央处理器执行）以及代码在运行时使用的数据和其他调试信息。

²ELF 文件类型又包括以下几类：(1) 可重定位文件 (relocate file) ——编译生成的文件，可被链接生成可执行文件，即 linux 下的.o 以及 windows 下的.obj 文件；(2) 可执行文件 (executable file) ——可以直接执行的程序，最典型的 ELF 文件格式，即 linux 下的/bin/bash 以及 windows 下的 exe 文件；(3) 共享目标文件 (shared object file) ——动态链接文件，可以和其他的可重定位文件和共享目标文件一起链接，生成新的目标文件，即 linux 下的.so 和 windows 下的 DLL 文件；(4) 核心转存文件 (core dump file) ——进程信息存储文件，当进程意外终止时，系统将该进程的地址空间以及终止时的一些信息存到该文件，即 linux 下的 core dump 文件。

码、数据、符号表、调试信息、字符串等都以**段**（有时也称之为**节**）的形式存储。我们可以借助指令 `size factorial.o` 查看.o 文件中各个段所占大小，如图14所示

```
yuzhao-peng@yuzhao-peng:~/Compiler_Sys/Lab0-1$ size factorial.o
text    data    bss     dec     hex filename
415      8       1     424     1a8 factorial.o
```

图 14: 段大小展示

现在对各个段进行简要的解释如下

- 代码段 (.text/.code): 程序源代码编译后的机器指令存放位置
- 数据段 (.data): 初始化不为 0 的全局和静态数据存放位置
- 数据段 (.bss): 未初始化或初始化为 0 的全局和静态数据存放位置
- 其他段

- \$.rodata: 存放只读数据, 跟.rodata 段一样
- \$.comment: 存放编译器版本信息
- \$.debug: 存放调试信息
- \$.dynamic: 存放动态链接信息
- \$.hash: 符号哈希表
- \$.line: 存放调试用的行号表
- \$.note: 存放额外的编译器信息
- \$.strtab: 存放 ELF 文件中用到的各种字符串
- \$.symtab: 符号表
- \$.shstrtab: 段名表
- \$.plt&.got: 动态链接的跳转表和全局入口表
- \$.init&.fini: 程序初始化与终结代码段
- \$.rel.text: 重定位表

```

yuzhao-peng@yuzhao-peng: ~/Compiler_Sys/Lab0-1$ objdump -s factorial.o
factorial.o: 文件格式 elf64-x86-64

Contents of section .text:
0000  f30f1efa 554889e5 4883ec20 6448b004  ....UH...dH..
0010  25280000 00488945 f831c048 8d45ec48  %(...H.E.I.H.E.H
0020  89c648bd 05000000 004889c7 e8000000  ..H.....E.....
0030  00c745f0 02000000 c745f401 000000eb  ..E.....E.....
0040  0e8b45f4 0faf45f0 8945f4b3 45f001b0  ..E...E...E...
0050  45ec3945 f07ea8b 45f489c6 48bd0500  E.9E...E...H...
0060  00000048 89c7e809 00000048 8b150000  ..H.....H....
0070  00004889 d6400c7 e8000000 00000000  ..H.....E.....
0080  0000488b 55f8a448 2b142328 00000074  ..H.U.dH...%(...t
0090  05e80000 0000c9c3 f30f1efa 554889e5  ....UH.....
00a0  4883ec10 897dfc09 75f8b37d fc01753b  H...).u...).u;
00b0  017df8ff f0000075 3248bd05 00000000  .).....u2H.....
00c0  4889c7e8 00000000 48bd0500 00000048  H.....H.....H
00d0  89c248bd 05000000 004889c6 48bd0500  ..H.....H...H...
00e0  00000048 89c7e800 00000090 c9c3f30f  ..H.....E.....
00f0  1efa5548 89e5efff ff0000bf 01000000  ..UH.....E.....
0100  e893ffff ff5dc3  ....J].

Contents of section .init_array:
0000 00000000 00000000 .....

Contents of section .comment:
0000 00474343 3a202855 0275e774 75203131  .GCC: (Ubuntu 11
0010 2e342e30 2d317562 756e7475 317e3232  .4.0-subunit1-22
0020 2e303429 2031312e 342e3000  .04) 11.4.0.
Contents of section .note.gnu.property:
0000 04000000 10000000 05000000 474e5500  ....GNU.
0010 020000c0 04000000 03000000 00000000 .....

Contents of section .eh_frame:
0000 14000000 00000000 017a5200 01781001  ....2R...X...
0010 1b0c0708 90010000 1c000000 1c000000  ....
0020 00000000 90000000 00450e10 860243bd  ....E...C...
0030 00232ffc 07000000 1c000000 3c000000  ....C...
0040 00000000 56000000 00450e10 860243bd  ....V...E...C...
0050 0024ddc 07000000 1c000000 3c000000  ..H.....E...
0060 00000000 19000000 00450e10 860243bd  ....E...C...
0070 06500c07 00000000  .P.....

```

图 15: 目标文件内容展示

根据15我们可以看见所有的非空段 (section) 的内容, 同时我们可以对照代码段的反汇编代码稍作分析 (如16图)。

```

0000000000000000 <main>:
0: f3 0f 1e fa      endbr64
4: 55                push  %rbp
5: 48 b9 e5          mov   %rsp,%rbp
8: 48 83 ec 20       sub   $0x20,%rsp
c: 64 48 8b 04 25 28 00 mov   %fs:(0x28,%rax)
13: 00 00
15: 48 b9 45 f8       mov   %rax,-0x8(%rbp)
19: 31 c0             xor   %eax,%eax
1b: 48 bd 45 ec       lea   -0x14(%rbp),%rax
1f: 48 b9 c6          mov   %rax,%rsi
22: 48 bd 05 00 00 00 00 lea   0x0(%rip),%rax # 29 <main+0x29>
29: 48 b9 c7          mov   %rax,%rdi
2c: e8 00 00 00 00    call  11 <main+0x11>
31: c7 45 f0 02 00 00 movl  $0x2,-0x10(%rbp)
38: c7 45 f4 01 00 00 movl  $0x1,-0xc(%rbp)
3f: eb 0e            jmp   4f <main+0x4f>
41: 8b 45 f4          mov   -0xc(%rbp),%eax
44: 0f af 45 f0       imul  -0x10(%rbp),%eax
48: 89 45 f4          mov   %eax,-0xc(%rbp)
4b: 83 45 f0 01       addl  $0x1,-0x10(%rbp)
4f: 8b 45 ec          mov   -0x14(%rbp),%eax
52: 39 45 f0          cmp   %eax,-0x10(%rbp)
56: 7e e2            jle   64 <main+0x64>
57: 8b 45 f4          mov   -0xc(%rbp),%eax
5a: 89 c5             mov   %eax,%rsi
5c: 48 bd 05 00 00 00 00 lea   0x0(%rip),%rax # 63 <main+0x63>
63: 48 b9 c7          mov   %rax,%rdi
66: e8 00 00 00 00    call  6b <main+0x6b>
6b: 48 b9 15 00 00 00 00 mov   0x0(%rip),%rdx # 72 <main+0x72>
72: 48 b9 d6          mov   %rdx,%rsi
75: 48 b9 c7          mov   %rax,%rdi
78: e8 00 00 00 00    call  7d <main+0x7d>
7d: b8 00 00 00 00    mov   $0x0,%eax
82: 48 b9 55 f8       mov   -0x8(%rbp),%rdx
86: 64 48 2b 14 25 28 00 sub   %fs:(0x28,%rdx)
8d: 00 00
8f: 74 05            je    96 <main+0x96>
91: e8 00 00 00 00    call  96 <main+0x96>
96: c9               leave
97: c3               ret

```

图 16: 代码段内容展示

到目前为止我们对汇编器输出的目标文件反汇编代码已经做了相对细致的分析, 那么汇编器的工作我们是否已经了解详尽了呢? 让我们回过头来看一下编译器生成的汇编代码, 对比之后再回来回答这个问题吧。下面是通过指令 `g++ factorial.i -S -o factorial.S` 生成的汇编代码中关于 `main` 函数部分:

```

1  main:
2  .LFB1731:

```

```

3      .cfi_startproc
4      endbr64
5      pushq   %rbp
6      .cfi_def_cfa_offset 16
7      .cfi_offset 6, -16
8      movq    %rsp, %rbp
9      .cfi_def_cfa_register 6
10     subq    $32, %rsp
11     movq    %fs:40, %rax
12     movq    %rax, -8(%rbp)
13     xorl    %eax, %eax
14     leaq    -20(%rbp), %rax
15     movq    %rax, %rsi
16     leaq    _ZSt3cin(%rip), %rax
17     movq    %rax, %rdi
18     call    _ZNSirsERi@PLT
19     movl    $2, -16(%rbp)
20     movl    $1, -12(%rbp)
21     jmp     ^I.L2
22     .L3:
23     movl    -12(%rbp), %eax
24     imull   -16(%rbp), %eax
25     movl    %eax, -12(%rbp)
26     addl    $1, -16(%rbp)
27     .L2:
28     movl    -20(%rbp), %eax
29     cmpl    %eax, -16(%rbp)
30     jle     ^I.L3
31     movl    -12(%rbp), %eax
32     movl    %eax, %esi
33     leaq    _ZSt4cout(%rip), %rax
34     movq    %rax, %rdi
35     call    _ZNSolsEi@PLT
36     movq    _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_@GOTPCREL(%rip), %rdx
37     movq    %rdx, %rsi
38     movq    %rax, %rdi
39     call    _ZNSolsEPFRSoS_E@PLT
40     movl    $0, %eax
41     movq    -8(%rbp), %rdx
42     subq    %fs:40, %rdx
43     je      .L5
44     call    __stack_chk_fail@PLT
45     .L5:
46     leave

```

```

47     .cfi_def_cfa 7, 8
48     ret
49     .cfi_endproc

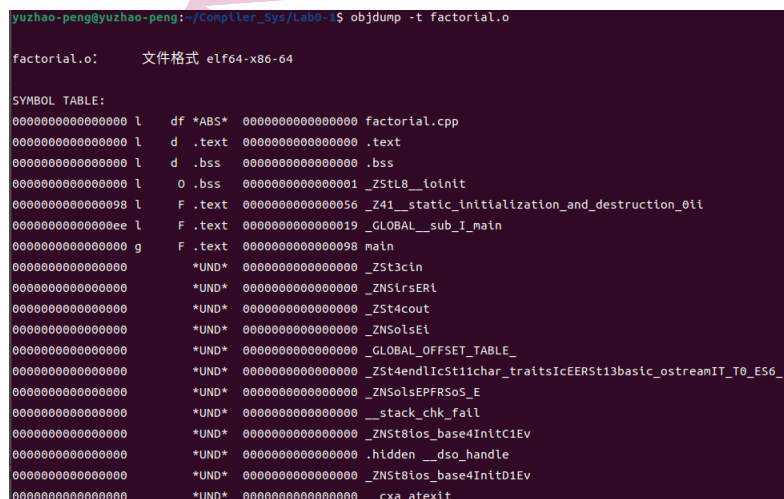
```

4. 汇编器小结

那么现在我们可以正式回答**汇编器做了什么？**

汇编器作为将汇编语言翻译为机器语言的工具主要做了以下几项工作：

1. **符号解析**：汇编器会解析汇编语言代码中的符号、标签和助记符。符号和标签通常用于表示内存地址或跳转目标，汇编器需要将它们解析为确切的内存地址或偏移量。针对于本实验示例我们可以轻松地发现原汇编代码中 `_ZNSirsERi@PLT` 和 `_ZNSolsEi@PLT` 是与输入和输出相关的函数，而在我们的目标文件的反汇编代码中函数调用的目标以地址的方式表示，而不包含函数名。
2. **生成机器指令**：汇编器会将汇编语言指令翻译成机器语言指令。每个汇编指令都对应着一个特定的机器指令，汇编器负责将其转换为二进制表示形式。
3. **内存分配**：汇编器会为变量、标签和数据分配内存空间。它需要追踪内存的分配和释放，确保每个变量和标签都有合适的内存位置。
4. **生成目标文件**：汇编器通常会生成一个目标文件，其中包含了汇编代码的二进制表示。这个目标文件可以进一步链接到其他目标文件，以创建可执行文件或共享库。
5. **错误检测和报告**：汇编器会检查源代码中的语法错误、拼写错误或其他问题，并生成错误报告，以帮助程序员找到和修复问题。
6. **生成符号表**：汇编器通常会生成一个符号表，其中包含了程序中定义的符号、标签和它们对应的内存地址。这有助于调试和符号解析。对于本实验则，符号表入口信息如图17所示。



```

yuzhao-peng@yuzhao-peng:~/Compiler_Sys/Lab0-15$ objdump -t factorial.o

factorial.o: 文件格式 elf64-x86-64

SYMBOL TABLE:
0000000000000000 l df *ABS* 0000000000000000 factorial.cpp
0000000000000000 l d .text 0000000000000000 .text
0000000000000000 l d .bss 0000000000000000 .bss
0000000000000000 l O .bss 0000000000000001 _ZStL8_toInit
0000000000000098 l F .text 0000000000000056 _Z41_static_initialization_and_destruction_0ii
00000000000000ee l F .text 0000000000000019 _GLOBAL__sub_I_main
0000000000000000 g F .text 0000000000000098 main
0000000000000000 *UND* 0000000000000000 _ZSt3cin
0000000000000000 *UND* 0000000000000000 _ZNSirsERi
0000000000000000 *UND* 0000000000000000 _ZSt4cout
0000000000000000 *UND* 0000000000000000 _ZNSolsEi
0000000000000000 *UND* 0000000000000000 _GLOBAL__OFFSET_TABLE_
0000000000000000 *UND* 0000000000000000 _ZSt4endlcSt11char_traitsIcEERSt13basic_ostreamIT_0_E6_
0000000000000000 *UND* 0000000000000000 _ZNSolsEPFRSo5_E
0000000000000000 *UND* 0000000000000000 __stack_chk_fail
0000000000000000 *UND* 0000000000000000 _ZNSt8ios_base4InitC1Ev
0000000000000000 *UND* 0000000000000000 _hidden __dso_handle
0000000000000000 *UND* 0000000000000000 _ZNSt8ios_base4InitD1Ev
0000000000000000 *UND* 0000000000000000 _cxa_atexit

```

图 17: 符号表入口展示

7. **可重定位代码生成**：汇编器可以生成可重定位的机器语言代码，这意味着生成的代码可以在不同的内存地址上加载和执行。这支持了模块化编程和库的链接。

8. **指令优化**：一些汇编器可以执行一些简单的代码优化，例如删除死代码、减少指令数目或执行常量折叠等。这一部分我们可以有以下几种选择：

\$ 无优化 (-O0)：生成可读性强的汇编代码，不进行优化。

```
g++ -O0 -S source.cpp -o output.s
```

\$ 优化级别 1 (-O1)：进行基本优化，如删除未使用的变量和内联一些函数。

```
g++ -O1 -S source.cpp -o output.s
```

\$ 优化级别 2 (-O2)：中等级别的优化，包括更多的内联、循环展开等。

```
g++ -O2 -S source.cpp -o output.s
```

\$ 优化级别 3 (-O3)：高级别的优化，通常会更激进地进行内联、循环展开、矢量化等优化。

```
g++ -O3 -S source.cpp -o output.s
```

\$ 优化级别 s：优化大小而非速度，通常会更激进地减小生成的二进制文件的大小。

```
g++ -Os -S source.cpp -o output.s
```

\$ 自定义优化选项：您还可以使用自定义的优化选项来控制不同的优化行为。

```
g++ -Ocustom -S source.cpp -o output.s
```

受篇幅限制这些优化的结果不再赘述。

总之，汇编器的主要工作是将汇编语言代码转换为机器语言代码，以便计算机可以执行。这个过程涉及到符号解析、指令翻译、内存分配、错误检测、目标文件生成和其他任务。最终，汇编器的输出是一个二进制文件，可以在计算机上加载和运行。

(四) 链接器与加载器

1. 链接器概述

在上一部分中我们已经得到了由汇编器生成的目标文件 `factorial.o`，虽然我们说“从广义上来说，可以将目标文件和可执行文件看成一种类型的文件”，但是它们细分还是有所区别，而链接器就是将目标文件转换成可执行文件的关键所在。链接器 (Linker) 将一个或多个由编译器或汇编器生成的目标文件与库文件链接在一起，创建一个可执行程序或共享库。

2. 链接器分析

现在让我们通过目标文件 `factorial.o` 生成可执行文件，在终端中输入指令：`g++ factorial.o -o factorial`，如图18所示，便可以得到可执行文件 `factorial`。

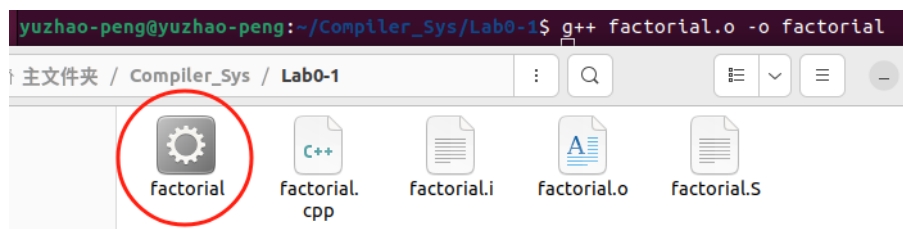


图 18: 生成可执行文件

通过运行可执行文件 `factorial`，我们可以发现能够正常输出结果：


```
yuzhao-peng@yuzhao-peng:~/Compiler_Sys/Lab0-1$ ./factorial
5      输入n=5
120    输出n!=120
yuzhao-peng@yuzhao-peng:~/Compiler_Sys/Lab0-1$ ./factorial
10      输入n=10
3628800 输出n!=3628800
yuzhao-peng@yuzhao-peng:~/Compiler_Sys/Lab0-1$ ./factorial
3      输入n=3
6      输出n!=6
```

图 19: 可执行文件执行

现在我们将进一步探索链接器在将目标文件转换成可执行文件的过程中究竟干了些什么。

过程一：符号解析 (Symbol Resolution)

在目标文件中符号表给链接器提供了两种信息，一个是当前目标文件可以提供给其它目标文件使用的符号，另一个其它目标文件需要提供给当前目标文件使用的符号。

现在给出两个定义：

- 已定义符号集合 D
- 未定义符号集合 U

有了上面两个定义，我们就可以描述链接器执行符号解析的过程：

- 查找当前目标文件的符号表，将已定义符号添加至集合 D 中
- 查找当前目标文件的符号表，将每一个当前目标文件引用的符号与集合 D 中的元素进行对比，如果该符号不在集合 D 中则将其添加到集合 U 中。
- 当所有文件都扫描完成后，如果未定义符号集合 U 不为空，则说明当前输入的目标文件集合中有未定义错误，链接器报错，整个编译过程终止。

简单的说，目标文件所引用的变量都能在其他目标文件中找到唯一定义，那么链接成功。

值得一提的是链接器和汇编器都涉及符号解析，但它们在解决符号引用和符号解析时有一些重要的区别：

(1) 链接器的符号解析解决不同模块之间的符号引用问题，确保程序的各个部分能够正确连接在一起。链接器会查找每个目标文件中引用的符号，并在符号表中查找对应的定义。如果找到了符号的定义，链接器会将引用替换为正确的地址或偏移量，以便程序在运行时能够正确访问这些符号。同时，链接器还会处理全局符号和局部符号的区分，以及公共符号的合并。

(2) 汇编器需要解析汇编源代码中的符号，包括标签、变量名、函数名等，将符号解析为相应的地址或偏移量，并生成目标文件，其中包含了符号的定义和引用，**但这些引用通常仍然以符号的形式存在，而不是具体的地址**。另外，汇编器不关心不同模块之间的符号引用问题，因为它只负责将一个源文件编译成一个目标文件。符号的最终解析是由链接器完成的。

总之，链接器的符号解析涉及将多个目标文件或共享库合并成一个整体，并确保符号引用在不同模块之间正确连接。而汇编器的符号解析是将汇编源代码转换为目标文件，生成包含符号的定义和引用的中间文件。链接器解决了不同模块之间的符号引用问题，而汇编器只负责生成目标文件，不涉及多模块的符号解析。

过程二：生成库、可执行文件

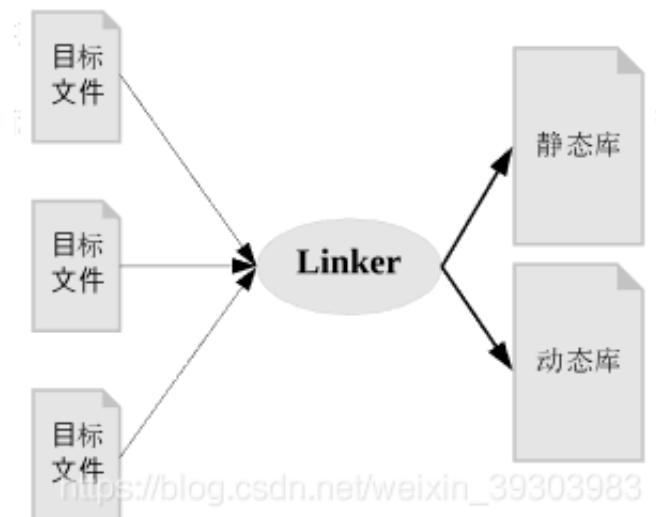


图 20: 生成库

链接器的链接过程包括生成静态库、动态库和可执行文件。生成静态库时，目标文件的代码和数据会被复制到库中；生成动态库时，库的代码和数据不会被复制，而是在运行时加载；生成可执行文件时，链接器会将多个目标文件和库文件合并为一个独立可运行的程序。

过程三：重定向

链接器中一个关键的过程就是重定向，包括地址重定向和符号重定向。

在链接之前的目标文件中代码和数据通常使用相对于模块的起始地址来表示的，而当链接器将多个模块合并为一个可执行文件时，需要将这些相对地址转换为绝对地址，以便在运行时正确访问内存中的代码和数据。

符号重定向是在链接过程中解决符号引用的问题。其中所谓的符号可以是函数、变量、类等。当一个模块引用了另一个模块中定义的符号时，链接器需要查找符号的实际定义，并将引用替换为正确的地址或偏移量。

综合来说，重定向就是将目标文件中的相对地址修正为可执行文件中的绝对地址，同时将模块中的符号引用与实际定义关联起来。

3. 加载器概述

事实上，加载器（Loader）是操作系统的一部分，其主要作用是将可执行程序从存储介质（通常是硬盘或固态硬盘）加载到计算机的内存中，并准备好执行。

加载器是操作系统启动和管理程序的重要组成部分，其任务包括内存分配、地址空间映射、地址重定向、装载、启动程序、动态链接以及错误处理。

但这并非我们在编译系统原理中所研究的范围，因此在这里不做更多的赘述。

以上便是编译完整过程每一步过程的探究，在此再次总结编译的流程：

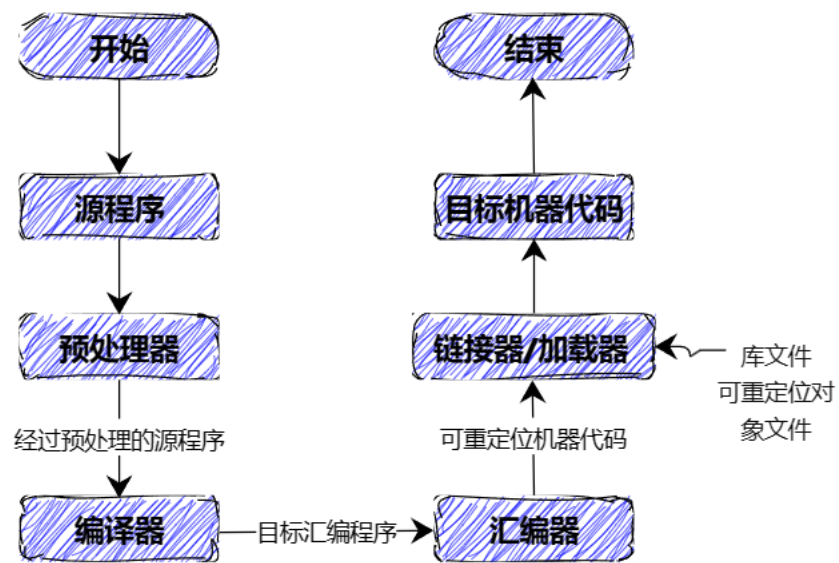


图 21: 编译流程

二、 LLVM IR 编程

(一) SysY 语言概述

SysY 语言是由 C 语言的一个自己扩展而成，其本质上仍然是 C 语言的一个子集。

- 数据类型：支持 int 型（32 位有符号数）和 float 型（32 位单精度浮点数），以及按行优先存储的多维数组类型。此外，SysY 还支持 int 和 float 之间的隐式类型转换。
- 函数实现：基本与 C 语言函数功能类似
- 变量/常量声明：允许在一个语句中声明一个或多个变量或者常量，可以对其初始化，定义后方可使用。
- 语句：包括赋值语句、表达式语句、if 语句、while 语句等。

(二) LLVM IR 概述

LLVM IR (Intermediate Representation) 是相对于 CPU 指令集更为高级、但相对于源程序更为低级的代码中间表示的一种语言，具有类型化、可扩展性和强表现力的特点。LLVM IR 代码存在三种表示形式：在内存中的表示 (BasicBlock、Instruction 等 cpp 类)、二进制代码形式 (用于编译器加载)、可读的汇编语言表示形式。架构如下



图 22: 架构

根据特性我们做了如下 LLVM IR 编程尝试。

(三) 整数运算 & 循环分支

源代码使用本实验测试代码 factorial.cpp;

编写的 LLVM LR 程序:

```

1 str = private unnamed_addr constant [3 x i8] c"%d\00", align 1
2 str.1 = private unnamed_addr constant [4 x i8] c"%d\n\00", align 1
3
4 @main() {
5   try:
6   %n = alloca i32
  
```

```

7 %i = alloca i32
8 %f = alloca i32
9 %1 = alloca i32
10 %2 = alloca i32
11 %3 = alloca i32
12 %4 = alloca i32
13 store i32 0, i32* %f
14 store i32 2, i32* %i
15 store i32 1, i32* %3
16 br label %while.cond
17
18 ile.cond:                                ; preds = %while.cond, %entry
19 %5 = load i32, i32* %i
20 %6 = load i32, i32* %n
21 %7 = icmp sle i32 %5, %6
22 br i1 %7, label %while.body, label %while.end
23
24 ile.body:                                ; preds = %while.cond
25 %8 = load i32, i32* %f
26 %9 = load i32, i32* %i
27 %10 = mul i32 %8, %9
28 store i32 %10, i32* %f
29 %11 = load i32, i32* %i
30 %12 = add i32 %11, 1
31 store i32 %12, i32* %i
32 br label %while.cond
33
34 ile.end:                                ; preds = %while.cond
35 %13 = load i32, i32* %f
36 store i32 %13, i32* %4
37 %14 = load i32, i32* %4
38 %15 = call i32 @i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]*
39 @.str.1, i32 0, i32 0), i32 %14)
40 %16 = load i32, i32* %1
41 %17 = xor i32 %16, -1217177845
42 %18 = call i32 @i8*, ...)* @printf(i8* getelementptr inbounds ([3 x i8], [3 x i8]*
43 @.str, i32 0, i32 0), i32 %17)
44 ret i32 0
45
46
47 clare i32 @printf(i8*, ...)

```

测试结果:

```
hanjiang@hanjiang-virtual-machine:~/C++/factorial$ ./factorial
3
6
hanjiang@hanjiang-virtual-machine:~/C++/factorial$ ./factorial
7
5040
```

图 23

(四) 浮点数 & 条件分支——学生成绩等级划分程序

```
1  include<iostream>
2  using namespace std;
3
4  void print(float averageScore)
5
6  {
7      if (averageScore >= 90.0) {
8          cout << "Grade: A" << endl;
9      } else if (averageScore >= 80.0) {
10         cout << "Grade: B" << endl;
11     } else if (averageScore >= 70.0) {
12         cout << "Grade: C" << endl;
13     } else if (averageScore >= 60.0) {
14         cout << "Grade: D" << endl;
15     } else {
16         cout << "Grade: F (Fail)" << endl;
17     }
18 }
19
20 int main()
21 {
22     float mathScore, physicsScore, chemistryScore;
23     float averageScore;
24
25     // 输入学生的数学、物理和化学成绩
26     cout << "Enter math score: ";
27     cin >> mathScore;
28
29     cout << "Enter physics score: ";
30     cin >> physicsScore;
31
32     cout << "Enter chemistry score: ";
33     cin >> chemistryScore;
34
35     // 计算平均分
```

```

34  averageScore = (mathScore + physicsScore + chemistryScore) / 3.0;
35
36  // 根据平均分数输出等级
37  cout << "Average score: " << averageScore << endl;
38
39  print(averageScore);
40

```

上述代码体现了若干 SysY 语言的若干特性：

1. 整个文件相当于一个 Module，只有一个文件入口是 main 函数，体现了 SysY 语言中的 CompUnit 特性；

2. 在 main 函数外单独定义一个函数，对应函数定义 FuncDef、函数类型 FuncType、函数形参 FuncFParam 等多个有关函数定义的特性；

3. 函数中有多处声明变量的操作，体现了 SysY 语言声明变量时变量类型确定的特点；4. 自定义函数和 main 函数中的语句逻辑体现了语句块 block 的思想，体现着 SysY 语言的条件分支语句及表达式的特性。

翻译完成的 llvm ir 中间代码

```

1  ; LLVM IR version 8.0.0
2
3  ; 声明标准输入输出库的 printf 和 scanf 函数
4  declare i32 @printf(i8*, ...)
5  declare i32 @scanf(i8*, ...)
6
7  ; 定义全局变量和常量字符串
8  @.str_math = private unnamed_addr constant [16 x i8] c"Enter math score: \00"
9  @.str_physics = private unnamed_addr constant [19 x i8] c"Enter physics score
   : \00"
10 @.str_chemistry = private unnamed_addr constant [20 x i8] c"Enter chemistry
   score: \00"
11 @.str_average = private unnamed_addr constant [17 x i8] c"Average score: %.2f
   \00"
12 @.str_grade = private unnamed_addr constant [8 x i8] c"Grade: \00"
13 @.str_A = private unnamed_addr constant [3 x i8] c"A\00"
14 @.str_B = private unnamed_addr constant [3 x i8] c"B\00"
15 @.str_C = private unnamed_addr constant [3 x i8] c"C\00"
16 @.str_D = private unnamed_addr constant [3 x i8] c"D\00"
17 @.str_F = private unnamed_addr constant [12 x i8] c"F (Fail)\00"
18
19 ; 定义主函数
20 define i32 @main() {
21   entry:
22     ; 分配存储空间来存储变量
23     %mathScore = alloca float
24     %physicsScore = alloca float
25     %chemistryScore = alloca float
26     %averageScore = alloca float

```

```

27
28 ; 输入学生的数学成绩
29 %mathScorePtr = getelementptr inbounds [16 x i8], [16 x i8]* @.str_math,
    i32 0, i32 0
30 call i32 @i8*, ... @printf(i8* %mathScorePtr)
31 %mathScoreInput = alloca float
32 call i32 @i8*, ... @scanf(i8* %mathScorePtr, float* %mathScoreInput)
33 %mathScoreValue = load float, float* %mathScoreInput
34 store float %mathScoreValue, float* %mathScore
35
36 ; 输入学生的物理成绩
37 %physicsScorePtr = getelementptr inbounds [19 x i8], [19 x i8]* @.
    str_physics, i32 0, i32 0
38 call i32 @i8*, ... @printf(i8* %physicsScorePtr)
39 %physicsScoreInput = alloca float
40 call i32 @i8*, ... @scanf(i8* %physicsScorePtr, float* %
    physicsScoreInput)
41 %physicsScoreValue = load float, float* %physicsScoreInput
42 store float %physicsScoreValue, float* %physicsScore
43
44 ; 输入学生的化学成绩
45 %chemistryScorePtr = getelementptr inbounds [20 x i8], [20 x i8]* @.
    str_chemistry, i32 0, i32 0
46 call i32 @i8*, ... @printf(i8* %chemistryScorePtr)
47 %chemistryScoreInput = alloca float
48 call i32 @i8*, ... @scanf(i8* %chemistryScorePtr, float* %
    chemistryScoreInput)
49 %chemistryScoreValue = load float, float* %chemistryScoreInput
50 store float %chemistryScoreValue, float* %chemistryScore
51
52 ; 计算平均分数
53 %sum = fadd float %mathScoreValue, %physicsScoreValue
54 %sum = fadd float %sum, %chemistryScoreValue
55 %averageScoreValue = fdiv float %sum, 3.0
56 store float %averageScoreValue, float* %averageScore
57
58 ; 输出平均分数
59 %averageScorePtr = getelementptr inbounds [17 x i8], [17 x i8]* @.
    str_average, i32 0, i32 0
60 call i32 @i8*, ... @printf(i8* %averageScorePtr, float %
    averageScoreValue)
61
62 ; 调用打印函数来输出等级
63 call void @print(float %averageScoreValue)
64
65 ; 返回 0 表示成功执行
66 ret i32 0
67 }

```



```

68
69 ; 定义打印函数
70 define void @print(float %averageScore) {
71     entry:
72         ; 根据平均分数输出等级
73         %gradePtr = getelementptr inbounds [8 x i8], [8 x i8]* @.str_grade, i32
74             0, i32 0
75         %cmp1 = fcmp oge float %averageScore, 90.0
76         %cmp2 = fcmp oge float %averageScore, 80.0
77         %cmp3 = fcmp oge float %averageScore, 70.0
78         %cmp4 = fcmp oge float %averageScore, 60.0
79
80         ; 使用条件分支来输出等级
81         br i1 %cmp1, label %grade_A, label %check_B
82
83     grade_A:
84         %str_A = getelementptr inbounds [3 x i8], [3 x i8]* @.str_A, i32 0, i32 0
85         call i32 @printf(i8* %str_A)
86         br label %exit
87
88     check_B:
89         br i1 %cmp2, label %grade_B, label %check_C
90
91     grade_B:
92         %str_B = getelementptr inbounds [3 x i8], [3 x i8]* @.str_B, i32 0, i32 0
93         call i32 @printf(i8* %str_B)
94         br label %exit
95
96     check_C:
97         br i1 %cmp3, label %grade_C, label %check_D
98
99     grade_C:
100         %str_C = getelementptr inbounds [3 x i8], [3 x i8]* @.str_C, i32 0, i32 0
101         call i32 @printf(i8* %str_C)
102         br label %exit
103
104     check_D:
105         br i1 %cmp4, label %grade_D, label %grade_F
106
107     grade_D:
108         %str_D = getelementptr inbounds [3 x i8], [3 x i8]* @.str_D, i32 0, i32 0
109         call i32 @printf(i8* %str_D)
110         br label %exit
111
112     grade_F:
113         %str_F = getelementptr inbounds [12 x i8], [12 x i8]* @.str_F, i32 0, i32
114             0
115         call i32 @printf(i8* %str_F)

```

```
114  
115 exit:  
116     ret void  
117 }
```

使用

```
clang -S -fobjc-arc float_if.ll -o float_if.s
```

```
clang -fmodules -c float_if.s -o float_if.o
```

clang float_if.o -o float_if 经过编译程序运行结果如图24所示

```
yuzhao-peng@yuzhao-peng:~/Compiler_Sys/Lab0-1$ ./float_if  
Enter math score: 100  
Enter physics score: 100  
Enter chemistry score: 100  
Average score: 100  
Grade: A  
yuzhao-peng@yuzhao-peng:~/Compiler_Sys/Lab0-1$ ./float_if  
Enter math score: 50  
Enter physics score: 100  
Enter chemistry score: 20  
Average score: 56.6667  
Grade: F (Fail)  
yuzhao-peng@yuzhao-peng:~/Compiler_Sys/Lab0-1$ ./float_if  
Enter math score: 50  
Enter physics score: 50  
Enter chemistry score: 100  
Average score: 66.6667  
Grade: D
```

图 24: 程序测试