



南開大學
Nankai University

南 開 大 學

编译系统实验报告

了解编译器及 LLVM IR 编程

王思宇

年级：2021 级

专业：信息安全

蒋薇

年级：2021 级

专业：计算机科学与技术

2023 年 9 月 15 日

摘要

GCC 是一个流行和强大的编译器, 被广泛用于各种操作系统和平台上, 具有很高的兼容性和可移植性, 可以在多个平台上生成高质量的可执行文件。我们以 GCC、LLVM 为研究对象, 并且结合具体程序代码及实例深入地探究语言处理系统的完整工作过程。本次以一个简单的 C (C++) 源程序为例, 调整编译器的程序选项获得各阶段的输出, 研究它们与源程序的关系。最后编写 LLVM IR 程序, 熟悉 LLVM IR 中间语言。并尽可能地对其实现方式有所了解。

关键字: GCC、编译过程、SysY 编译器、LLVM IR

目录

一、完整编译过程	1
(一) 完整编译过程简述	1
(二) 预处理器功能及具体实现	1
1. 预处理器功能介绍	2
2. 阶乘实例预处理器具体实现	2
3. 与源程序关系	2
(三) 编译器功能功能及实现	3
1. 编译器功能介绍	3
2. 阶乘实例编译器各阶段具体实现	3
(四) 汇编器功能功能及实现	5
1. 汇编器功能介绍	5
2. 阶乘实例汇编器具体实现	5
(五) 链接器功能功能及实现	5
1. 链接器功能介绍	5
2. 阶乘实例链接器具体实现	6
二、SysY 编译器与 LLVM IR 中间语言	6
(一) SysY、LLVM IR 简要介绍	6
1. SysY 介绍	6
2. LLVM IR 介绍	6
(二) test.cpp	7
(三) LLVM IR 程序及关键信息说明	7
(四) LLVM 编译执行	12
三、问题与总结	13

一、完整编译过程

(一) 完整编译过程简述

如图1所示

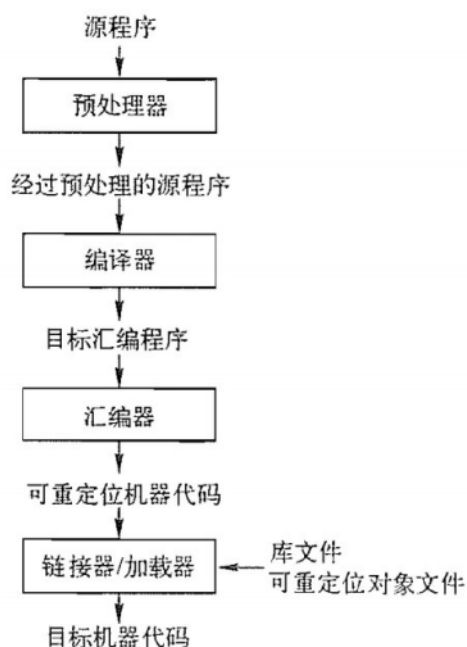


图 1: 编译过程流程图

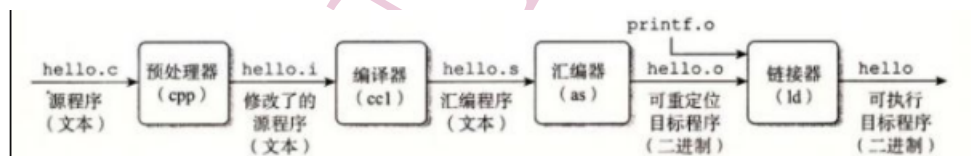


图 2: 程序流程表示

编译器驱动程序读取源程序文件 `main.c`，并把它翻译成一个可执行目标文件 `main`。这个翻译过程分为四个阶段：预处理（Preprocessing）、编译（Compilation）、汇编（Assembly）、链接（Linking）。执行这四个阶段的程序（预处理器、编译器、汇编器、和链接器）一起构成了编译系统。[1]

(二) 预处理器功能及具体实现

编写阶乘实例的 C++ 文件：`main.cpp`，如图所示

```
army@DESKTOP-8P7QBT0:~$ ls
main.cpp  snap  test
army@DESKTOP-8P7QBT0:~$ cat main.cpp
#include<iostream>
using namespace std;
int main(){
    int i, n, f;
    cin >> n; //输入一个n
    i=2;
    f=1;
    while(i<=n){
        f=f*i;
        i=i+1;
    }
    cout<<f<<endl; //最后输出结果
    return 0;
}army@DESKTOP-8P7QBT0:~$
```

图 3: 源程序图

1. 预处理器功能介绍

预处理器是编译过程中的一个阶段。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。

2. 阶乘实例预处理器具体实现

对于 gcc，通过添加参数 -E 令 gcc 只进行预处理过程，参数 -o 改变 gcc 输出文件名 (cpp 文件将 gcc 改为 g++)，通过以下命令得到预处理后文件 main.i：

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

```
}army@DESKTOP-8P7QBT0:~$ g++ main.cpp -E -o main.i
army@DESKTOP-8P7QBT0:~$ ls
main.cpp  main.i  snap  test
army@DESKTOP-8P7QBT0:~$
```

图 4: 预处理图

3. 与源程序关系

预处理器根据源代码中的预处理指令（以 # 开头的指令）执行宏替换、条件编译、包含头文件等。

宏替换：将源代码中的宏定义进行展开，将宏名称替换为其定义的代码片段。

条件编译：根据条件编译指令（如 #ifdef、#ifndef、#if、elif 等）判断哪些代码需要包含在最终的编译结果中，哪些代码需要被忽略。

头文件包含：预处理器会根据 #include 指令将指定的头文件内容插入到源文件中。

符号定义与替换：预处理器会处理源代码中的宏定义和预定义符号，将它们替换为对应的值或表达式。

注释删除：预处理器会删除源代码中的注释，使得注释对编译和链接过程没有影响。

如下图可以看到源程序中的 #include 指令已经被替换，注释被删除，添加了行号和文件标识名，：

```
# 2 "main.cpp" 2

# 2 "main.cpp"
using namespace std;
int main(){
    int i, n, f;
    cin >> n;
    i=2;
    f=1;
    while(i<=n){
        f=f*i;
        i=i+1;
    }
    cout<<f<<endl;
    return 0;
}
```

图 5: 预处理图

(三) 编译器功能功能及实现

1. 编译器功能介绍

编译器的主要功能是将预处理器处理过的源程序文件翻译成为标准的汇编语言以供计算机阅读。同时还负责对源代码进行词法、语法、语义分析,进行代码优化,生成目标代码,并进行错误处理。编译器是程序开发中不可或缺的工具,它能够将高级语言的源代码转化为底层机器能够执行的代码,实现程序的编译和执行。编译过程具体分为六步:词法分析、语法分析、语义分析、中间代码生成、代码优化、目标代码生成。

2. 阶乘实例编译器各阶段具体实现

词法分析

将源程序转换为单词序列。对于 LLVM,可以通过以下命令获得 token 序列:

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

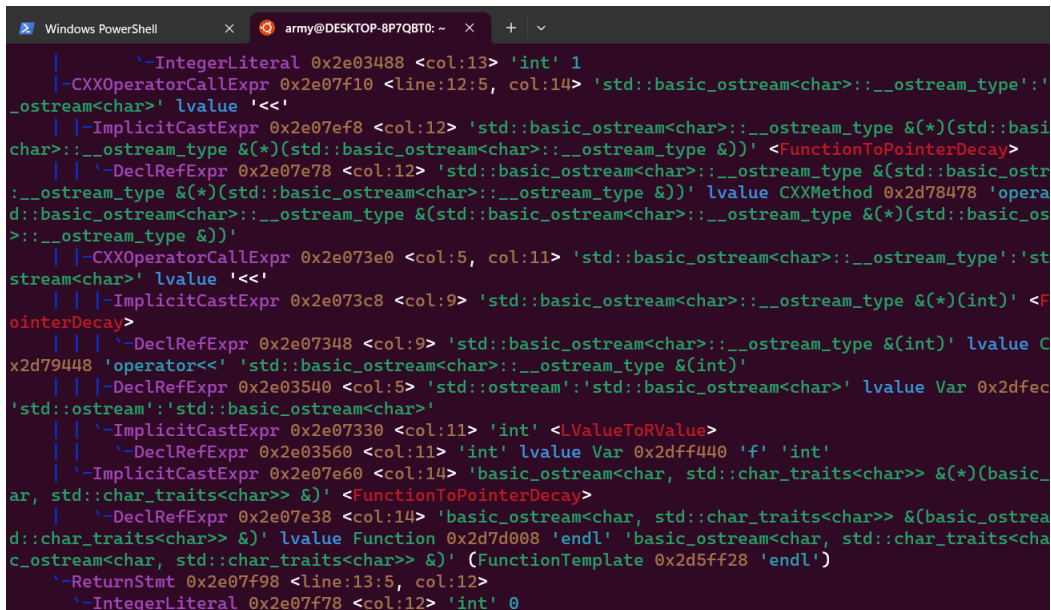
```
numeric_constant '1'          Loc=<main.cpp:7:7>
semi ';'                      Loc=<main.cpp:7:8>
while 'while'                  [StartOfLine] [LeadingSpace] Loc=<main.cpp:8:5>
l_paren '('                   Loc=<main.cpp:8:10>
identifier 'i'                 Loc=<main.cpp:8:11>
lessequal '<='                 Loc=<main.cpp:8:12>
identifier 'n'                 Loc=<main.cpp:8:14>
r_paren ')'                   Loc=<main.cpp:8:15>
l_brace '{'                   Loc=<main.cpp:8:16>
identifier 'f'                 [StartOfLine] [LeadingSpace] Loc=<main.cpp:9:9>
equal '='                     Loc=<main.cpp:9:10>
identifier 'f'                 Loc=<main.cpp:9:11>
star '*'                      Loc=<main.cpp:9:12>
identifier 'i'                 Loc=<main.cpp:9:13>
semi ';'                      Loc=<main.cpp:9:14>
identifier 'i'                 [StartOfLine] [LeadingSpace] Loc=<main.cpp:10:9>
equal '='                     Loc=<main.cpp:10:10>
identifier 'i'                 Loc=<main.cpp:10:11>
plus '+'                      Loc=<main.cpp:10:12>
numeric_constant '1'          Loc=<main.cpp:10:13>
semi ';'                      Loc=<main.cpp:10:14>
r_brace '}'                   [StartOfLine] [LeadingSpace] Loc=<main.cpp:11:5>
identifier 'cout'              [StartOfLine] [LeadingSpace] Loc=<main.cpp:12:5>
lessless '<<'                  Loc=<main.cpp:12:9>
identifier 'f'                 Loc=<main.cpp:12:11>
lessless '<<'                  Loc=<main.cpp:12:12>
```

图 6: 词法分析图

语法分析

将词法分析生成的词法单元来构建抽象语法树 (Abstract Syntax Tree, 即 AST)。对于 LLVM 可以通过如下命令获得相应的 AST:

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



```

|   '-IntegerLiteral 0x2e03488 <col:13> 'int' 1
|   |-CXXOperatorCallExpr 0x2e07f10 <line:12:5, col:14> 'std::basic_ostream<char>::__ostream_type':
_ostream<char>' lvalue '<<'
|   |   |-ImplicitCastExpr 0x2e07ef8 <col:12> 'std::basic_ostream<char>::__ostream_type &(*) (std::basi
char>::__ostream_type &(*) (std::basic_ostream<char>::__ostream_type &))' <FunctionToPointerDecay>
|   |   |   |-DeclRefExpr 0x2e07e78 <col:12> 'std::basic_ostream<char>::__ostream_type &(std::basic_ostr
__ostream_type &(*) (std::basic_ostream<char>::__ostream_type &))' lvalue CXXMethod 0x2d78478 'opera
d::basic_ostream<char>::__ostream_type &(std::basic_ostream<char>::__ostream_type &(*) (std::basic_os
>::__ostream_type &))'
|   |   |   |-CXXOperatorCallExpr 0x2e073e0 <col:5, col:11> 'std::basic_ostream<char>::__ostream_type': 'st
stream<char>' lvalue '<<'
|   |   |   |   |-ImplicitCastExpr 0x2e073c8 <col:9> 'std::basic_ostream<char>::__ostream_type &(*) (int)' <F
ointerDecay>
|   |   |   |   |   |-DeclRefExpr 0x2e07348 <col:9> 'std::basic_ostream<char>::__ostream_type &(int)' lvalue C
x2d79448 'operator<<' 'std::basic_ostream<char>::__ostream_type &(int)'
|   |   |   |   |   |-DeclRefExpr 0x2e03540 <col:5> 'std::ostream': 'std::basic_ostream<char>' lvalue Var 0x2dfec
'std::ostream': 'std::basic_ostream<char>'
|   |   |   |   |   |-ImplicitCastExpr 0x2e07330 <col:11> 'int' <LValueToRValue>
|   |   |   |   |   |-DeclRefExpr 0x2e03560 <col:11> 'int' lvalue Var 0x2dff440 'f' 'int'
|   |   |   |   |   |-ImplicitCastExpr 0x2e07e60 <col:14> 'basic_ostream<char, std::char_traits<char>> &(*) (basic_
ar, std::char_traits<char>> &)' <FunctionToPointerDecay>
|   |   |   |   |   |-DeclRefExpr 0x2e07e38 <col:14> 'basic_ostream<char, std::char_traits<char>> &(basic_ostrea
d::char_traits<char>> &)' lvalue Function 0x2d7d008 'endl' 'basic_ostream<char, std::char_traits<cha
c_ostream<char, std::char_traits<char>> &)' (FunctionTemplate 0x2d5ff28 'endl')
|   |   |   |   |-ReturnStmt 0x2e07f98 <line:13:5, col:12>
|   |   |   |   |-IntegerLiteral 0x2e07f78 <col:12> 'int' 0

```

图 7: 语法分析图

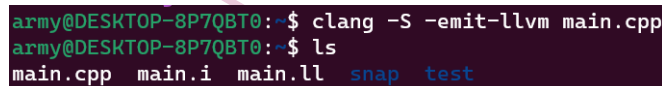
语义分析

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。

中间代码生成

完成上述步骤后, 很多编译器会生成一个明确的低级或类机器语言的中间表示。对于 LLVM 可以通过下面的命令生成 LLVM IR (即 main.ll):

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



```

army@DESKTOP-8P7QBT0:~$ clang -S -emit-llvm main.cpp
army@DESKTOP-8P7QBT0:~$ ls
main.cpp  main.i  main.ll  snap  test

```

图 8: 中间代码生成图

在命令行中输入 `cat main.ll`, 可以打开同目录下的 `main.ll` 文件查看生成的 LLVM IR, 如图是部分 LLVM IR 截图:



```

define dso_local noundef 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
    %5 = call noundef nonnull align 8 dereferenceable(16) @"class.std::basic_istream"*_@_ZNSirsERi(%"class.std::
basic_istream"* noundef nonnull align 8 dereferenceable(16) @_ZSt3cin, i32* noundef nonnull align 4 dereferenc
eable(4) %3)
    store i32 2, i32* %2, align 4
    store i32 1, i32* %4, align 4
    br label %6

6:                                     ; preds = %0
    %7 = load i32, i32* %2, align 4
    %8 = load i32, i32* %3, align 4
    %9 = icmp sle i32 %7, %8
    br i1 %9, label %10, label %16

```

图 9: LLVM IR 生成图

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。可以通过 LLVM 现有的优化 pass 进行代码优化探索。LLVM 可以通过下面的命令生成每个 pass 后生成的 LLVM IR 以观察差别：

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

目标代码生成

以优化后代码的中间表示形式作为输入，将其映射到目标语言。LLVM 生成目标代码可以使用如下命令：

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

```
army@DESKTOP-8P7QBT0:~$ llc -print-before-all -print-after-all a.ll > a.log 2>&1
army@DESKTOP-8P7QBT0:~$ llc main.ll -o main.S
army@DESKTOP-8P7QBT0:~$ ls
a.log  main.S  main.cpp  main.i  main.ll  snap  test
```

图 10: 目标代码生成图

至此就将高级语言转换成为了汇编语言。接下来就是使用汇编器将汇编语言转换为二进制机器码。

(四) 汇编器功能功能及实现

1. 汇编器功能介绍

汇编器的主要功能是将汇编语言代码翻译成机器语言指令，实现汇编过程，从而将高级语言代码或汇编语言代码转化为底层机器能够执行的代码。并将汇编语言程序打包成可重定位目标程序。

2. 阶乘实例汇编器具体实现

详细分析并阐述汇编器处理的结果以及汇编器的具体功能分析

x86 格式汇编可以直接用 gcc 完成汇编器的工作，如使用下面的命令 (cpp 文件将 gcc 换为 g++)：

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

```
army@DESKTOP-8P7QBT0:~$ g++ main.S -c -o main.o
army@DESKTOP-8P7QBT0:~$ ls
a.log  main.S  main.cpp  main.i  main.ll  main.o  snap  test
```

图 11: 汇编器图

(五) 链接器功能功能及实现

1. 链接器功能介绍

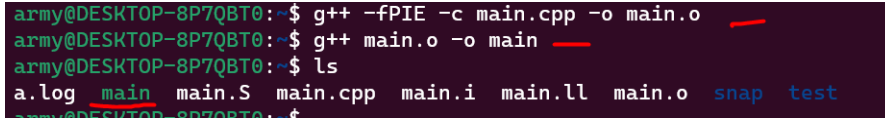
链接器 (linker) 是编译过程中的最后一步，将可重定位的机器代码和相应的一些目标文件合并成一个可执行文件或者库文件。通过解析符号引用、重定位、解决符号冲突等操作，链接器将程序的各个部分有机地结合在一起，使得程序能够正确地执行。由汇编程序生成的目标文件不能够直接执行。大型程序经常被分成多个部分进行编译，因此，可重定位的机器代码有必要和其他可重定位的目标文件以及库文件链接到一起，最终形成真正在机器上运行的代码。这就是链接器的功能。

2. 阶乘实例链接器具体实现

对于 g++, 可以使用如下命令将汇编器生成的机器语言生成最终的可执行文件:

```
g++ -fPIE -c main.cpp -o main.o
```

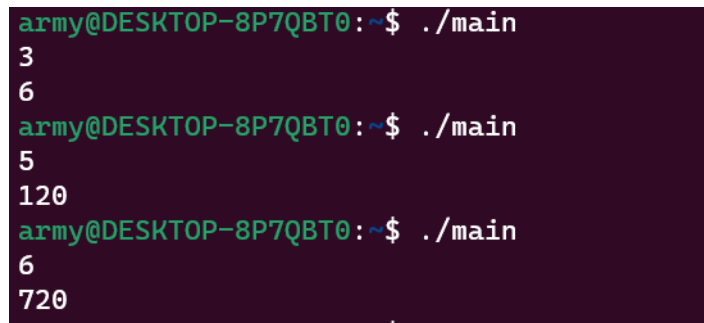
```
g++ main.o -o main
```



```
army@DESKTOP-8P7QBT0:~$ g++ -fPIE -c main.cpp -o main.o
army@DESKTOP-8P7QBT0:~$ g++ main.o -o main
army@DESKTOP-8P7QBT0:~$ ls
a.log  main  main.S  main.cpp  main.i  main.ll  main.o  snap  test
```

图 12: 链接器图

最后, 当我们执行可执行文件时, 便会使用到加载器, 以将二进制文件载入内存。以下是生成的阶乘的可执行文件的测试运行结果:



```
army@DESKTOP-8P7QBT0:~$ ./main
3
6
army@DESKTOP-8P7QBT0:~$ ./main
5
120
army@DESKTOP-8P7QBT0:~$ ./main
6
720
```

图 13: 结果图

二、 SysY 编译器与 LLVM IR 中间语言

目的: 通过编写 LLVM IR 程序, 熟悉 LLVM IR 中间语言, 了解 SysY 编译器特性。

(一) SysY、LLVM IR 简要介绍

1. SysY 介绍

SysY 语言常用于编译原理和计算机体系结构的教学中, 可以用来演示和实践编译器的各个阶段, 包括词法分析、语法分析、语义分析、中间代码生成和代码优化等。同时, SysY 语言也可以作为一种研究对象, 用来设计和实现新的编译器优化技术。SysY 编译器就是针对 SysY 语言的编译器。

2. LLVM IR 介绍

LLVM IR (Intermediate Representation, 中间表示) 是一种与机器无关的中间代码表示形式, 由 LLVM 编译器框架使用。它是一种低级的静态单赋值 (SSA) 形式, 具有简单的指令集和类型系统。

LLVM IR 可以通过 LLVM 工具链中的前端将各种高级语言 (如 C、C++ 等) 编译为 LLVM IR, 然后再通过 LLVM 工具链中的后端将 LLVM IR 编译为目标平台的机器代码。

clang 是 llvm 编译器工具集的前端, 将源代码转换为抽象语法树 (AST), 由后端使用 llvm 编译成平台相关机器代码。

(二) test.cpp

输入 n, 计算 n 到 100 的累加和并输出。

```

fufu@DESKTOP-PCJ99L0: $ cat test.cpp
#include<iostream>
using namespace std;
const int s = 100;

//返回a + b 的结果
int add(int a,int b){
    return a + b;
}

//实现从n到100的累加，如输入n=99,输出199
int main() {
    int i,n,r;
    cin >> n;
    r = 0;
    while(n <=s) {
        r = r + add(n, 0);
        n = n + 1;
    }
    cout << r << endl;
    return 0;
}

```

图 14: test.cpp

(三) LLVM IR 程序及关键信息说明

我们已对 LLVM IR 特性有所了解，结合我们的累加程序代码，我们初步编写 LLVM IR 展示如下：

n-100 累加 LLVM IR

```

1 ; 所有的全局变量都以 @ 为前缀，后面的 global 关键字表明了它是一个全局变量
2 ; 函数定义以 define 开头，i32 标明了函数的返回类型(32位带符号整数类型)，main 表名
   是主函数，@ 是其前缀
3 ; 以 % 开头的符号表示虚拟寄存器，可以把它当作临时变量，称为临时寄存器
4
5 ; ModuleID = 'test.cpp'
6 source_filename = "test.cpp"
7 target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8
   :16:32:64-S128"
8 target triple = "x86_64-pc-linux-gnu"
9
10 %"class.std::ios_base::Init" = type { i8 }
11 %"class.std::basic_istream" = type { i32 (...)** , i64 , %"class.std::basic_ios
   " }
12 %"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"* }

```

```

13 %"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" }
14 %"struct.std::ios_base::_Callback_list" = type { %"struct.std::ios_base::_
    _Callback_list"*, void (i32, %"class.std::ios_base"*, i32)*, i32, i32 }
15 %"struct.std::ios_base::_Words" = type { i8*, i64 }
16 %"class.std::locale" = type { %"class.std::locale::_Impl"* }
17 %"class.std::locale::_Impl" = type { i32, %"class.std::locale::facet"*, i64,
    %"class.std::locale::facet"*, i8** }
18 %"class.std::locale::facet" = type <{ i32 (...)**, i32, [4 x i8] }>
19 %"class.std::basic_ostream" = type { i32 (...)**, %"class.std::basic_ios" }
20 %"class.std::basic_streambuf" = type { i32 (...)**, i8*, i8*, i8*, i8*, i8*,
    i8*, %"class.std::locale" }
21 %"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] }>
22 %"class.std::locale::facet.base" = type <{ i32 (...)**, i32 }>
23 %struct.__locale_struct = type { [13 x %struct.__locale_data*], i16*, i32*,
    i32*, [13 x i8*] }
24 %struct.__locale_data = type opaque
25 %"class.std::num_put" = type { %"class.std::locale::facet.base", [4 x i8] }
26 %"class.std::num_get" = type { %"class.std::locale::facet.base", [4 x i8] }
27
28 @__ZStL8__ioinit = internal global %"class.std::ios_base::Init"
    zeroinitializer, align 1
29 @__dso_handle = external hidden global i8
30 @__ZSt3cin = external global %"class.std::basic_istream", align 8
31 @__ZSt4cout = external global %"class.std::basic_ostream", align 8
32 @llvm.global_ctors = appending global [1 x { i32, void ()*, i8* }] [{ i32,
    void ()*, i8* } { i32 65535, void ()* @_GLOBAL__sub_I_test.cpp, i8* null
    }]
33
34 ; Function Attrs: noinline uwtable
35 define internal void @__cxx_global_var_init() #0 section ".text.startup" {
36     call void @__ZNSt8ios_base4InitC1Ev(%"class.std::ios_base::Init"* noundef
        nonnull align 1 dereferenceable(1) @__ZStL8__ioinit)
37     %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
38     ret void
39 }
40
41 declare void @__ZNSt8ios_base4InitC1Ev(%"class.std::ios_base::Init"* noundef
    nonnull align 1 dereferenceable(1)) unnamed_addr #1
42
43 ; Function Attrs: nounwind

```

```

44 declare void @_ZNSt8ios_base4InitD1Ev(%"class.std::ios_base::Init"* noundef
    nonnull align 1 dereferenceable(1)) unnamed_addr #2
45
46 ; Function Attrs: nounwind
47 declare i32 @__cxa_atexit(void (i8*)*, i8*, i8*) #3
48
49 ; Function Attrs: mustprogress noline nounwind optnone uwtable
50
51 ; add函数
52 define dso_local noundef i32 @_Z3addii(i32 noundef %0, i32 noundef %1) #4 {
53 ; 分配两个虚拟寄存器3, 4; 参数列表中虚拟寄存器0, 1存有a,b的值
54   %3 = alloca i32, align 4
55   %4 = alloca i32, align 4
56
57 ; 0寄存器中存的a值存到3寄存器, 1寄存器中存的b值存到4寄存器
58   store i32 %0, i32* %3, align 4
59   store i32 %1, i32* %4, align 4
60   %5 = load i32, i32* %3, align 4
61   %6 = load i32, i32* %4, align 4
62
63   ; 7寄存器计算a+b的结果并返回
64   %7 = add nsw i32 %5, %6
65   ret i32 %7
66 }
67
68 ; Function Attrs: mustprogress noline norecurse optnone uwtable
69 define dso_local noundef i32 @main() #5 {
70 ; 四个虚拟寄存器
71   %1 = alloca i32, align 4
72   %2 = alloca i32, align 4
73   %3 = alloca i32, align 4
74   %4 = alloca i32, align 4
75
76 ; 0存入1号寄存器
77   store i32 0, i32* %1, align 4
78
79 ; 调用输入函数, 3号寄存器存输入的n
80   %5 = call noundef nonnull align 8 dereferenceable(16) %"class.std::
        basic_istream"* @_ZNSirsERi(%"class.std::basic_istream"* noundef
        nonnull align 8 dereferenceable(16) @_ZSt3cin, i32* noundef nonnull
        align 4 dereferenceable(4) %3)
81
82 ; 0存入4号寄存器, 可知4号寄存器存r
83   store i32 0, i32* %4, align 4
84
85 ; br无条件分支, 跳转下面的标签6
86   br label %6
87

```

```

88 6:                                     ; preds = %9, %0
89 ; 3号寄存器的值n放到7号寄存器
90 %7 = load i32, i32* %3, align 4
91 ; 比较n与100大小
92 %8 = icmp sle i32 %7, 100
93 ; 比较结果为真, 跳转标签9, 比较结果为假, 跳转标签16
94 br i1 %8, label %9, label %16
95
96 9:                                     ; preds = %6
97 ; 4号寄存器r加载到10号寄存器
98 %10 = load i32, i32* %4, align 4
99 ; 3号寄存器n加载到11号寄存器
100 %11 = load i32, i32* %3, align 4
101 ; 调用add函数, 参数为n,0, 返回结果在12号寄存器
102 %12 = call @noundef i32 @_Z3addii(i32 @noundef %11, i32 @noundef 0)
103 ; 13号寄存器为10号寄存器与12号寄存器相加
104 %13 = add nsw i32 %10, %12
105 ; 13号寄存器结果放回4号寄存器r, 实现r = r+add(n,0)
106 store i32 %13, i32* %4, align 4
107 ; 14号寄存器存放3号寄存器n的值
108 %14 = load i32, i32* %3, align 4
109 ; 15号寄存器实现n +1
110 %15 = add nsw i32 %14, 1
111 ; 15号寄存器n+1结果写回3号寄存器的n, 实现n = n+1
112 store i32 %15, i32* %3, align 4
113 ; 跳转标签6循环判定
114 br label %6, !llvm.loop !6
115
116 16:                                    ; preds = %6
117 ; 4号寄存器r加载到17号寄存器
118 %17 = load i32, i32* %4, align 4
119 %18 = call @noundef nonnull align 8 dereferenceable(8) @"class.std::
    basic_ostream"* @_ZNSolsEi(@"class.std::basic_ostream"* @noundef nonnull
    align 8 dereferenceable(8) @_ZSt4cout, i32 @noundef %17)
120
121 ; 调用输出函数, 输出17号寄存器r的值
122 %19 = call @noundef nonnull align 8 dereferenceable(8) @"class.std::
    basic_ostream"* @_ZNSolsEPFRSoS_E(@"class.std::basic_ostream"* @noundef
    nonnull align 8 dereferenceable(8) %18, @"class.std::basic_ostream"* (%
    "class.std::basic_ostream"*)* @noundef
    @_ZSt4endlcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_)
123 ; return 0
124 ret i32 0
125 }
126
127 ; declare为函数声明
128 declare @noundef nonnull align 8 dereferenceable(16) @"class.std::
    basic_istream"* @_ZNSirsERi(@"class.std::basic_istream"* @noundef nonnull

```

```

    align 8 dereferenceable(16), i32* noundef nonnull align 4 dereferenceable
    (4)) #1
129
130 declare noundef nonnull align 8 dereferenceable(8) %"class.std::basic_ostream
    "* @_ZNSolsEi(%"class.std::basic_ostream"* noundef nonnull align 8
    dereferenceable(8), i32 noundef) #1
131
132 declare noundef nonnull align 8 dereferenceable(8) %"class.std::basic_ostream
    "* @_ZNSolsEPFRSoS_E(%"class.std::basic_ostream"* noundef nonnull align 8
    dereferenceable(8), %"class.std::basic_ostream"* (%"class.std::
    basic_ostream"*)* noundef) #1
133
134 declare noundef nonnull align 8 dereferenceable(8) %"class.std::basic_ostream
    "* @_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_(%"class.
    std::basic_ostream"* noundef nonnull align 8 dereferenceable(8)) #1
135
136 ; Function Attrs: noinline uwtable
137 define internal void @_GLOBAL__sub_I_test.cpp() #0 section ".text.startup" {
138     call void @__cxx_global_var_init()
139     ret void
140 }
141
142 attributes #0 = { noinline uwtable "frame-pointer"="all" "min-legal-vector-
    width"="0" "no-trapping-math"="true" "stack-protector-buffer-size"="8" "
    target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
    "tune-cpu"="generic" }
143 attributes #1 = { "frame-pointer"="all" "no-trapping-math"="true" "stack-
    protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8
    ,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
144 attributes #2 = { nounwind "frame-pointer"="all" "no-trapping-math"="true" "
    stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"=
    "+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
145 attributes #3 = { nounwind }
146 attributes #4 = { mustprogress noinline nounwind optnone uwtable "frame-
    pointer"="all" "min-legal-vector-width"="0" "no-trapping-math"="true" "
    stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"=
    "+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
147 attributes #5 = { mustprogress noinline norecurse optnone uwtable "frame-
    pointer"="all" "min-legal-vector-width"="0" "no-trapping-math"="true" "
    stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"=
    "+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
148
149 !llvm.module.flags = !{!0, !1, !2, !3, !4}
150 !llvm.ident = !{!5}
151
152 !0 = !{i32 1, !"wchar_size", i32 4}
153 !1 = !{i32 7, !"PIC Level", i32 2}
154 !2 = !{i32 7, !"PIE Level", i32 2}

```

```

155 | !3 = !{i32 7, !"uhtable", i32 1}
156 | !4 = !{i32 7, !"frame-pointer", i32 2}
157 | !5 = !{"Ubuntu clang version 14.0.0-1ubuntu1.1"}
158 | !6 = distinct !{!6, !7}
159 | !7 = !{"llvm.loop.mustprogress"}

```

通过编写 LLVM IR 程序，我们对 LLVM IR 中间语言的特性有了更深理解，

1、静态单赋值形式 (SSA)：LLVM IR 使用静态单赋值形式，即每个变量只能被赋值一次，这使得程序分析和优化更加容易。

2、类型系统：LLVM IR 具有严格的类型系统，支持整数、浮点数、指针、向量和聚合类型等。

3、中间表示：LLVM IR 是一种中间表示，可以作为源代码和目标代码之间的桥梁。它提供了高级的抽象，同时又具有足够的低级细节，使得编译器可以进行各种优化。

4、低级指令集：LLVM IR 包含一组低级指令，用于执行各种操作，如算术运算、逻辑运算、内存访问等。

5、模块化和可扩展性：LLVM IR 支持模块化编程和可扩展性，允许用户定义自己的指令和数据类型，并且可以在不改变编译器核心的情况下进行扩展。

6、控制流图：LLVM IR 使用控制流图表示程序的控制流，包括基本块、分支和循环等。

7、异常处理：LLVM IR 支持异常处理机制，可以使用异常处理指令来捕获和处理异常。

8、全局变量和函数：LLVM IR 支持全局变量和函数的声明和定义，并提供了一组指令来访问和操作它们。

9、跨平台支持：LLVM IR 是与平台无关的，可以生成适用于不同硬件和操作系统的目标代码。

10、可读性和可调试性：LLVM IR 具有人类可读的文本表示形式，可以方便地进行调试和分析。

(四) LLVM 编译执行

编写完成 LLVM IR 程序之后，我们需要将它编译成目标程序执行验证。如图所示是对编译成功的目标程序进行验证，可以看出，目标程序成功实现了预期的功能。当输入一个 n ，可以计算出从 n 一直累加到 100 的和。与开始编写的 test.cpp 目标功能一致。得出结论：编写的 LLVM IR 程序成功执行。

```

fufu@DESKTOP-PCJ99L0:~$ ls
test test.S test.cpp test.i test.ll test.log test.o
fufu@DESKTOP-PCJ99L0:~$ ./test
99
199
fufu@DESKTOP-PCJ99L0:~$ ./test
98
297
fufu@DESKTOP-PCJ99L0:~$ ./test
1
5050
fufu@DESKTOP-PCJ99L0:~$ ./test
0
5050

```

图 15: LLVM IR 编程结果展示

三、 问题与总结

1. 有时候用到 apt-get 有时候用 apt, 两者具体有什么差异: apt 具有自动解决软件包依赖关系的能力, 可以自动安装或卸载相关的依赖软件包。而 apt-get 需要手动处理依赖关系。

2. 在链接器阶段我们将 main.o 文件生成 main 文件, 使用的命令是 `gcc main.o -o main`, 在这个命令中我们可以尝试加入 `-static` 参数: `-static` 参数指示在链接阶段将所有的库链接为静态库, 而不是使用动态链接库。静态链接库将所有的依赖都包含在可执行文件中, 使得可执行文件在运行时不需要依赖外部的库文件。这意味着生成的可执行文件会比较大, 但是在运行时的依赖性会更少。

3. 在实验过程中, 我们按照语言处理系统的不同阶段划分任务。一位同学负责预处理和编译阶段, 另一位同学负责汇编和链接阶段。最后编写 LLVM IR 程序时, 一起查阅资料进行编写, 最后一起进行验证。实验报告由两人共同负责编写。

MINIBU

参考文献

- [1] 实验指导书. 2023.

NIKU