编译原理作业 1：了解编译器

1811431 王鹏

2020 年 10 月 10 日

目录

1 摘要

1.1 摘要

摘要: 修改 gcc/g++ 的编译选项让编译器生成不同的中间结果。简要的说明编译器如何把 C++ 代码最后翻译成机器语言的四个阶段——预处理阶段，编译阶段，汇编阶段，链接阶段，并且主要对其中编译器将预处理之后的程序进行编译生成的汇编码和编译器的一些优化选项进行了理解

关键词:gcc/g++ 编译器，完整编译过程，编译技术，C/C++

2 引言

不管是 C/C++，还是 JAVA，或者是 Python，都是高级语言，然而高级语言是给程序员看的，计算机绝不可能看懂。编译器是一种计算机程序，它会将某种编程语言写成的源代码，转换成另一种编程语言。它主要的目的是将便于人编写，阅读，维护的高级计算机语言所写作的源代码程序，翻译为计算机能解读、运行的低阶机器语言的程序，也就是可执行文件。

编译器把对应的 *.cpp 首先预处理成 *.i 文件（宏的替换以及去掉注释），之后翻译成 *.s 文件 (汇编语言)，汇编器则处理 *.s 生成对应的 *.o 文件 (obj 目标文件)，汇编器是将代码语言翻译成机器语言最后链接器把所有的 *.o 文件链接成一个可执行文件 (.exe)

3 实验内容

环境：Windows Subsystem for Linux(WSL) + Ubuntu20.04
语言：C++
编译器：G++

3.1 预处理器

测试源代码如下：

```cpp
#include<iostream>
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;
    //testminted
}
```

首先，使用 **>g++ -E main.cpp -o main.i**，生成预处理的结果，保存到 main.i 中。main.i 内容：

```
# 1 "main.cpp"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "main.cpp"
# 1 "/usr/include/c++/9/iostream" 1 3
# 36 "/usr/include/c++/9/iostream" 3
# 37 "/usr/include/c++/9/iostream" 3
# 1 "/usr/include/x86_64-linux-gnu/c++/9/bits/c++config.h" 1 3
# 252 "/usr/include/x86_64-linux-gnu/c++/9/bits/c++config.h" 3
# 252 "/usr/include/x86_64-linux-gnu/c++/9/bits/c++config.h" 3
namespace std
...... //特别特别长的头文件宏替换的内容，竟然有两万多行（恐怖如斯
# 2 "main.cpp" 2
# 2 "main.cpp"
using namespace std;     //到这里才是我们写的阶乘 C++ 程序
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;
}
```

猜测前面的一大段应该是替代了 **#include<iostream>** 这一句，完成了导入 iostream 头文件的功能。但还是否有其他部分呢？为了验证猜测的合理性，我们在源程序加上一句 #include<cmath> ，发现产生的 main_plus.i 增加了许多行（28636->33086，而且这些行在 main.i 中是找不到的），新添加的部分代码经过比对，如下所示：

```
# 2 "main.cpp" 2
```

```
# 1 "/usr/include/c++/9/cmath" 1 3
# 39 "/usr/include/c++/9/cmath" 3
# 40 "/usr/include/c++/9/cmath" 3
```

我们可以得出结论预处理会处理以 # 开始的预编译指令，插入 include 指向的文件等，那么联想宏定义即 **#define** 也是以 # 开头的，因此我们把 f_replace **define** 成 f，再把程序里的 f 输出改成 f_replace 尝试，发现程序里的 f_replace 在 i 文件中写成了小 f。

总的来说，**预处理器的作用就是处理源代码中以 # 开头的预编译指令，例如展开所有宏定义，插入 include 指向的文件等，以获得经过预处理的源程序**

3.2 编译器

3.2.1 词法分析

将代码块拆分为 token 流，具体例子如下：

```
Line 1:f = f*i ; //源代码中的内容
. . .
{
    [ 'id ', '1'] ,
    [ '=' ] ,
    [ ' id ' , ' 1 ' ] ,
    [ '* '] ,
    [ 'id ' , '2 ' ] ,
    [ '; ' ]
}
```

两个 identifier 即 f 和 i 分别用 id1 和 id2 表示

3.2.2 AST(抽象语法树)/语法分析

main_1.cpp 源码经过前端解析生成的 AST 信息在 main_1.cpp.original 文件中，之后使用 gcc test.c -fdump-tree-original-raw 命令将 token 流结构化，此时 main_1.cpp 和生成的 AST 如下 (因为带有 **#include<iostream>** 的 AST 文件过长，因此新建了 main_1.cpp 文件再构建 AST)

main_1.cpp:

```
int square(int n){
    return n*n;
}
```

main_1.cpp.004t.original:

```
;; Function int square(int) (null)
;; enabled by -tree-original

@1      return_expr      type: @2       expr: @3
@2      void_type        name: @4       algn: 8
@3      init_expr        type: @5       op 0: @6       op 1: @7
@4      type_decl        name: @8       type: @2       srcp: <built-in>:0
                         note: artificial
@5      integer_type     name: @9       size: @10      algn: 32
                         prec: 32       sign: signed   min : @11
                         max : @12
@6      result_decl      type: @5       scpe: @13      srcp: main_1.cpp:1
                         note: artificial              size: @10
                         algn: 32
@7      mult_expr        type: @5       op 0: @14      op 1: @14
@8      identifier_node  strg: void     lngt: 4
@9      type_decl        name: @15      type: @5       srcp: <built-in>:0
                         note: artificial
@10     integer_cst      type: @16      int: 32
@11     integer_cst      type: @5       int: -2147483648
@12     integer_cst      type: @5       int: 2147483647
@13     function_decl    name: @17      type: @18      scpe: @19
                         srcp: main_1.cpp:1            args: @14
                         link: extern
@14     parm_decl        name: @20      type: @5       scpe: @13
                         srcp: main_1.cpp:1            argt: @5
                         size: @10      algn: 32       used: 1
@15     identifier_node  strg: int      lngt: 3
@16     integer_type     name: @21      size: @22      algn: 128
                         prec: 128      sign: unsigned min : @23
                         max : @24
@17     identifier_node  strg: square   lngt: 6
@18     function_type    size: @25      algn: 8        retn: @5
                         prms: @26
@19     translation_unit_decl name: @27
@20     identifier_node  strg: n        lngt: 1
@21     identifier_node  strg: bitsizetype            lngt: 11
@22     integer_cst      type: @16      int: 128
@23     integer_cst      type: @16      int: 0
@24     integer_cst      type: @16      int: -1
@25     integer_cst      type: @16      int: 8
```

```
@26     tree_list        valu: @5       chan: @28
@27     identifier_node  strg: main_1.cpp            lngt: 10
@28     tree_list        valu: @2
```

粗略理解了一下，编译器对函数 **square** 进行了分析，将语法树用文本的形式呈现。首先是对树中的每一个结点的标号，紧跟着的是结点的类型名。后面带 @ 的部分表示当前结点的子节点，例如如下表示**编号为 5 类型名为 integer_type 有子节点"9" 和"10" 分别代表着变量声明和字节大小 (4 个字节)。子节点"11" 和子节点"12" 分别表示 int 类型的最小值和最大值 ：**

```
@5      integer_type     name: @9       size: @10      algn: 32
                        prec: 32       sign: signed   min : @11
                        max : @12
@11     integer_cst      type: @5       int: -2147483648
@12     integer_cst      type: @5       int: 2147483647
```

再例如如下表示**编号为 7 类型名为 mult_expr 有子节点 5 和 14 和 14，5 表示操作的变量类型为 int 型，而 14 是待操作的两个数，即函数的参数变量**

```
@5      integer_type     name: @9       size: @10      algn: 32
                        prec: 32       sign: signed   min : @11
                        max : @12
@7      mult_expr        type: @5       op 0: @14      op 1: @14
@14     parm_decl        name: @20      type: @5       scpe: @13
                        srcp: main_1.cpp:1            argt: @5
                        size: @10      algn: 32       used: 1
```

因为只找到了 python 的 AST 可视化工具，因此我们可以大致看一下 square 函数的 AST 结构 (大致都一样嘛。。。以下是 python 代码:

```
def square(n):
I    n*n
```

AST 语法树结构：

```
- Program {
  - body: [
    - FunctionDeclaration {
      + id: Identifier {name}
      + params: [1 element]
      - body: BlockStatement {
        - body: [
          + VariableDeclaration {kind, declarations, userCode}
          + VariableDeclaration {kind, declarations, userCode}
          + VariableDeclaration {kind, declarations, userCode}
          + IfStatement {test, consequent}
          - ExpressionStatement {
            - expression: CallExpression {
              - arguments: [
                - Identifier {
                    name: "n"
                  }
                - Identifier {
                    name: "n"
                  }
                ]
              - callee: MemberExpression {
                + object: MemberExpression {object, property, computed, userCode}
                - property: Identifier = $node {
                    name: "multiply"
                    userCode: false
                  }
                  computed: false
                  userCode: false
```

3.2.3 语义分析

```
g++ -O0 -fdump-tree-all-graph main.cpp
```

选取 main.cpp.011t.cfg 和相应的.dot 文件做分析：

```
main ()
{
struct basic_ostream & D.44831;
int f;
int n;
int i;
```

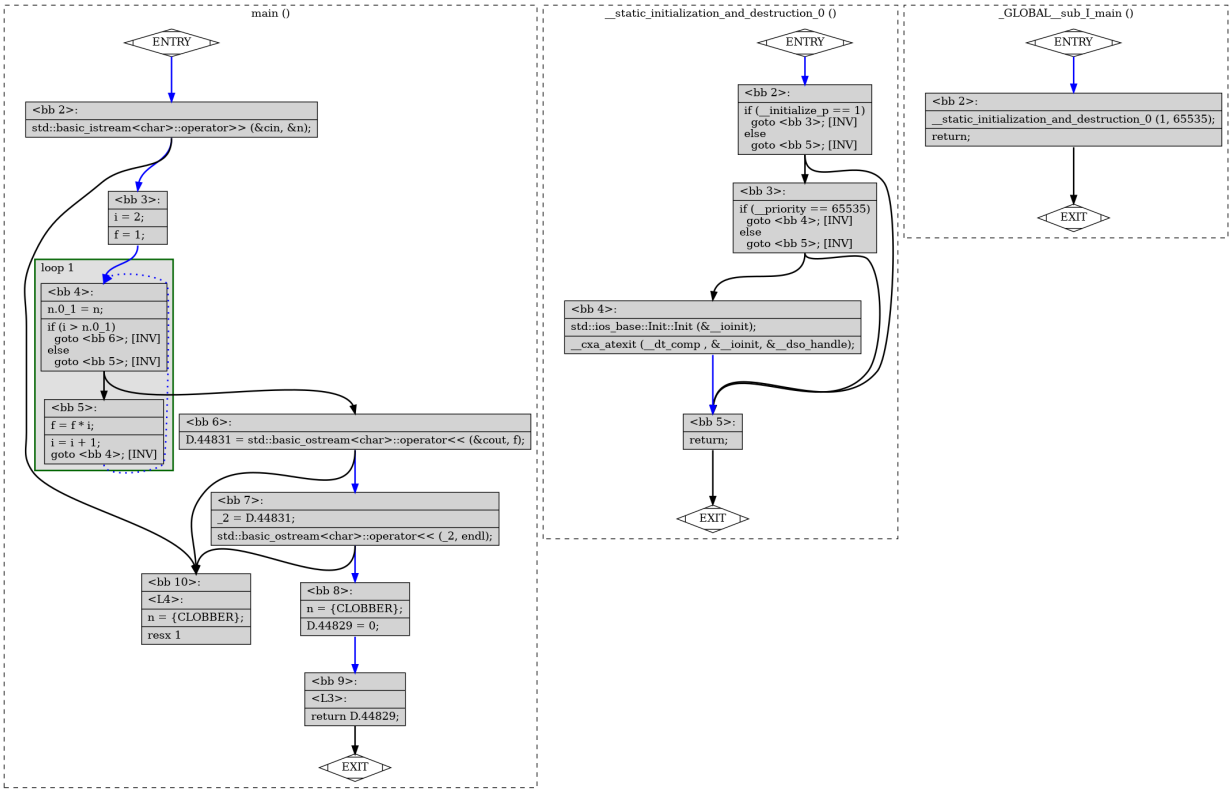
```
int D.44829;
<bb 2> :
std::basic_istream<char>::operator>> (&cin, &n);
<bb 3> :
i = 2;
f = 1;
<bb 4> :
n.0_1 = n;
if (i > n.0_1)
    goto <bb 6>; [INV]
else
    goto <bb 5>; [INV]
<bb 5> :
f = f * i;
i = i + 1;
goto <bb 4>; [INV]
<bb 6> :
D.44831 = std::basic_ostream<char>::operator<< (&cout, f);
<bb 7> :
_2 = D.44831;
std::basic_ostream<char>::operator<< (_2, endl);
<bb 8> :
n = {CLOBBER};
D.44829 = 0;
<bb 9> :
<L3>:
return D.44829;
<bb 10> :
<L4>:
n = {CLOBBER};
resx 1
}
```

可以看到中间代码中，先定义了三个整形变量 `f,n,i`，之后调用了 `operator">>"` 即输入 (操作数为 `n`)，之后进行一个循环对 `f` 和 `i` 进行赋值，最后调用了 `operator"<<"` 即输出并且返回一个值。比较有意思的代码如下：

```
n.0_1 = n;
if (i > n.0_1)
    goto <bb 6>; [INV]
else
    goto <bb 5>; [INV]
```

在中间代码中，用 **if**->goto "case1" **else**->goto "case2" 表示了 while 循环 (判断条件为 i>n),case1 和 case2 分别表示继续循环和跳出循环。

Graphviz 可视化分析 如下：



粗略地理解了一下，总共有三个模块。

模块一：main 也就是我们的主函数，基本与前面叙述的执行过程无差别，因此这里不再赘述。

模块二：_static_initialization_and_destruction(即构造和析构函数): 主要有两个作用：一、呼叫类与预处理/编译的 C++ 源文件中的静态存储的持续时间 (当需要时)。二、注册任何构造通过 atexit 将预处理/编译后的 C++ 源文件中的静态存储持续时间的类的析构函数（需要时）用作退出函数。

模块三：_GLOBAL_sub_I_main 是一个简单的包装函数，它的功能指针被添加到.init_array 部分。在编译多个源文件时，每个单独的目标文件都会向.init_array 添加初始化函数，并且在调用 main 之前将调用所有这些函数。

3.2.4 中间代码生成

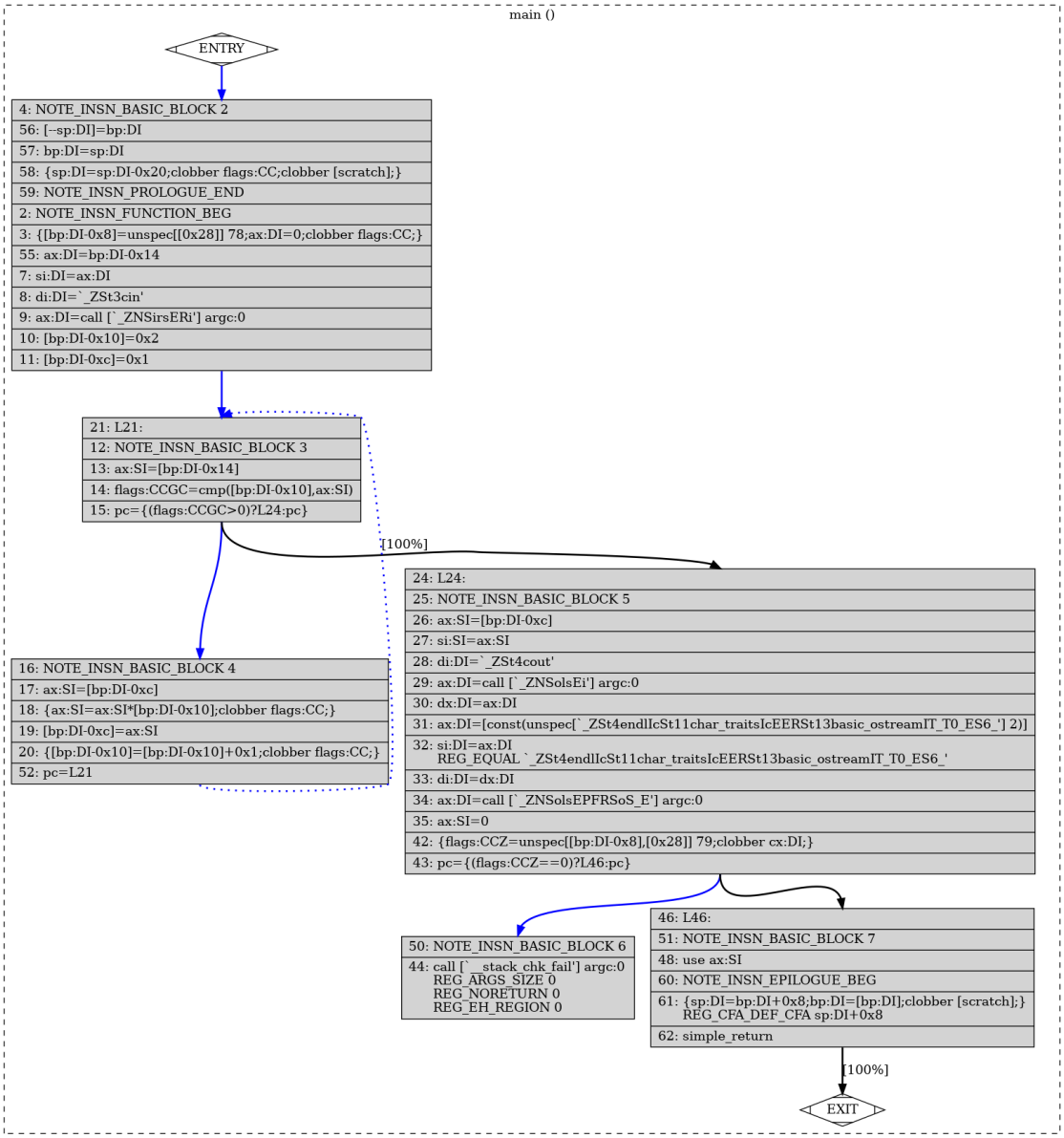
```
1  g++ -O0 -fdump-rtl-all-graph main.cpp
```

对 main.cpp.305r.stack 和对应的 dot 文件进行分析。以及和上一子章节相同的控制流图可视化分析

```
(insn 10 9 11 2 (set (mem/c:SI (plus:DI (reg/f:DI 6 bp)
                (const_int -16 [0xfffffffffffffff0])) [7 i+0 S4 A32])
        (const_int 2 [0x2])) "main.cpp":7:7 67 {*movsi_internal}
    (nil))
(insn 11 10 21 2 (set (mem/c:SI (plus:DI (reg/f:DI 6 bp)
                (const_int -12 [0xfffffffffffffff4])) [7 f+0 S4 A32])
        (const_int 1 [0x1])) "main.cpp":8:7 67 {*movsi_internal}
    (nil))
```

通过仔细观察代码可以看出 `1-3` 和第 `3-6` 行是对应源代码变量 `i` 和 `f` 的赋值语句。

Graphviz 可视化分析 如下：



3.2.5 目标代码生成

编译的最后一个阶段（即代码生成）是将预处理后的文件转换成汇编语言，即.S 文件。因此我们执行 **g++ -S main.cpp -fverbose-asm -o main.S**(其中-fverbose-asm 表示在汇编代码文件中加入源 C++ 代码对应的语句)，下面是部分汇编代码：

```
  .file^^I"main.cpp"
  main:
  # main.cpp:4: int main(){
Imovq^^I%fs:40, %rax^^I# MEM[(<address-space-1> long unsigned int *)40B], tmp93
Imovq^^I%rax, -8(%rbp)^^I# tmp93, D.44857
Ixorl^^I%eax, %eax^^I# tmp93
  # main.cpp:6:     cin >> n;
      leaq^^I-20(%rbp), %rax^^I#, tmp87
      movq^^I%rax, %rsi^^I# tmp87,
      leaq^^I_ZSt3cin(%rip), %rdi^^I#,
      call^^I_ZNSirsERi@PLT^^I#
  # main.cpp:7:     i = 2;
      movl^^I$2, -16(%rbp)^^I#, i
  # main.cpp:8:     f = 1;
      movl^^I$1, -12(%rbp)^^I#, f
  .L3:
  # main.cpp:9:     while (i <= n){
      movl^^I-20(%rbp), %eax^^I# n, n.0_1
      cmpl^^I%eax, -16(%rbp)^^I# n.0_1, i
      jg^^I.L2^^I#,
  # main.cpp:10:         f = f * i;
      movl^^I-12(%rbp), %eax^^I# f, tmp89
      imull^^I-16(%rbp), %eax^^I# i, tmp88
      movl^^I%eax, -12(%rbp)^^I# tmp88, f
  # main.cpp:11:         i = i + 1;
      addl^^I$1, -16(%rbp)^^I#, i
  # main.cpp:9:     while (i <= n){
      jmp^^I.L3^^I#
  .L2:
  # main.cpp:13:     cout <<" 结果为: "<< f << endl;
      leaq^^I.LC0(%rip), %rsi^^I#,
      leaq^^I_ZSt4cout(%rip), %rdi^^I#,
      call^^I_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@PLT^^I#
      movq^^I%rax, %rdx^^I#, _2
```

各部分都有对应的注释，例如 C++ 中的语句 **while (i <= n)** 翻译后由一个寄存器操作指令 movl，以及一个比较指令 cmpl，一个跳转指令 jg 组成，等等其他的汇编代码也比较容易理解不再赘述。值

得注意的一点如下 (我在源代码中加入了"cout<<" 结果为""):

```
.LC0:
I.string^^I"\347\273\223\346\236\234\344\270\272\357\274\232"
I.text
I.globl^^Imain
I.type^^Imain, %function
```

可以猜测在汇编代码中,将所有的这种字符串都定义成了全局变量 (.globl)。/347/273/223/346/236/234/344/270/272/357/274/232 应该代表着" 结果为:"。

3.3 汇编器

```
g++ -O0 -c -o main.o main.S
% 我尝试着用文本编辑器打开了.o 文件，发现已经不是我能理解的机器指令了
```

汇编阶段使用汇编器将汇编源代码汇编为目标机器指令文件，该阶段生成的文件为二进制文件。目标文件由段组成，通常一个目标文件中至少有两个段：代码段和数据段。代码段中包含的主要是程序的指令，一般是可读可执行的，但一般确不可写；数据段主要存放程序中要用到的各种全局变量或静态的数据，一般数据段都是可读，可写，可执行的。

汇编代码分析 反汇编的代码大致上与编译阶段生成的汇编代码相同。但是还是有些许区别。例如编译阶段生成的汇编代码中的 offset 为 10 进制，而反汇编生成的代码中为 16 进制。具细如下:

```
mov   -0x14(%rbp),%eax
% 反汇编生成的代码
movl^^I-20(%rbp), %eax^^I# n, n.0_1
% 编译阶段生成的汇编代码
...
%...\dots 还有很多，不再列举
```

常见汇编指令 :

Move：数据传送

Xor：异或指令，这本身是一个逻辑运算指令，但在汇编指令中通常会见到它被用来实现清零功能。用 xor eax,eax 这种操作来实现 mov eax,0，可以使速度更快，占用字节数更少。

Call：调用函数，一般函数的参数放在寄存器中；

lea ：取得第二个参数地址后放入到前面的寄存器（第一个参数），然而 lea 也同样可以实现 mov 的操作

jmp：无条件跳转指令

3.4 链接器

链接器做的工作简单而言就是将多个.o 对象文件合成出一个可执行文件。
链接器（Linker）是一个程序，将一个或多个由编译器或汇编器生成的目标文件外加库链接为一个可执行文件。目标文件是包括机器码和链接器可用信息的程序模块。简单的讲，链接器的工作就是解析未定义的符号引用，将目标文件中的占位符替换为符号的地址。链接器还要完成程序中各目标文件的地址空间的组织，这可能涉及重定位工作。

由汇编程序生成的目标文件并不能立即就被执行，其中可能还有许多没有解决的问题。例如，某个源文件中的函数可能引用了另一个源文件中定义的某个符号（如变量或者函数调用等）；在程序中可能调用了某个库文件中的函数，等等。链接程序的主要工作就是将有关的目标文件彼此相连接，将在一个文件中引用的符号同该符号在另外一个文件中的定义连接起来。

链接处理可分为两种：

3.4.1 静态链接

在这种链接方式下，函数的代码将从其所在地静态链接库中被拷贝到最终的可执行程序中。这样该程序在被执行时这些代码将被装入到该进程的虚拟地址空间中。静态链接库实际上是一个目标文件的集合，其中的每个文件含有库中的一个或者一组相关函数的代码

3.4.2 动态链接

在此种方式下，函数的代码被放到称作是动态链接库或共享对象的某个目标文件中。链接程序此时所作的只是在最终的可执行程序中记录下共享对象的名字以及其它少量的登记信息。在此可执行文件被执行时，动态链接库的全部内容将被映射到运行时相应进程的虚地址空间。动态链接程序将根据可执行程序中记录的信息找到相应的函数代码。C/C++ 编译过程对于可执行文件中的函数调用，可分别采用动态链接或静态链接的方法。使用动态链接能够使最终的可执行文件比较短小，并且当共享对象被多个进程使用时能节约一些内存，因为在内存中只需要保存一份此共享对象的代码。但并不是使用动态链接就一定比使用静态链接要优越。在某些情况下动态链接可能带来一些性能上损害。

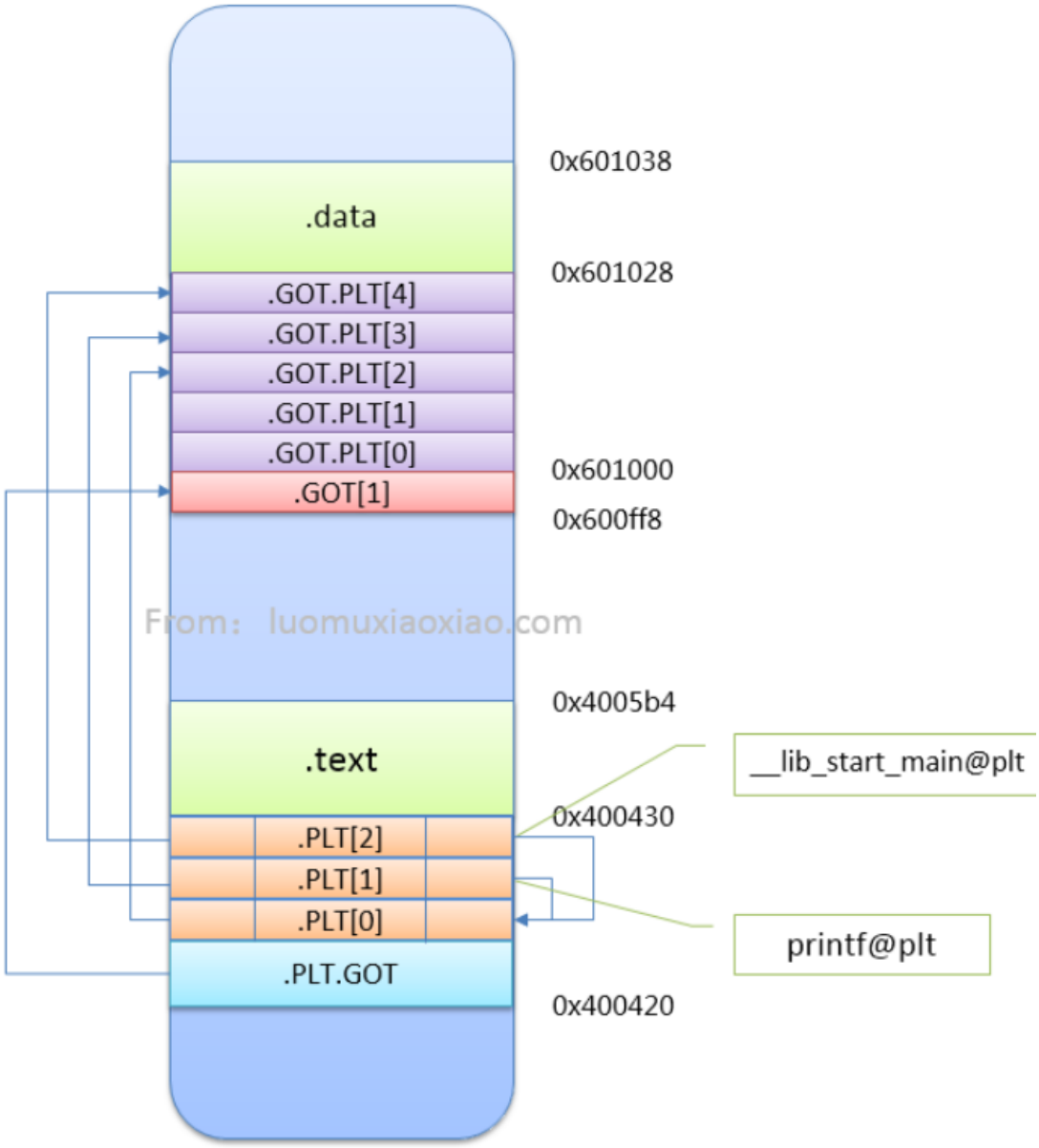
3.4.3 PLT 和 GOT

在执行 **g++ -O0 -o main main.o** 并进行反汇编后，main-anti-exe.S 相比 main-anti-obj.S 又增加了部分代码，框架如下：

```
1    Disassembly of section .text:
2    Disassembly of section .init:
3    Disassembly of section .plt:
4    Disassembly of section .plt.got:
```

GOT 全称 Global Offset Table，即全局偏移量表。它在可执行文件中是一个单独的 section，位于.data section 的前面。每个被目标模块引用的全局符号（函数或者变量）都对应于 GOT 中一个 8 字节的条目。编译器还为 GOT 中每个条目生成一个重定位记录。在加载时，动态链接器会重定位 GOT 中的每个条目，使得它包含正确的目标地址。

PLT 全称 Procedure Linkage Table,即过程链接表。它在可执行文件中也是一个单独的 section，位于.text section 的前面。每个被可执行程序调用的库函数都有它自己的 PLT 条目。每个条目实际上都是一小段可执行的代码。



运行时的PLT和GOT内存视图

3.5 编译器优化 (我的附加探索)

g++/gcc 编译器一共包含三个优化级别-O1、O2、O3，优化级别越高，产生的代码的执行效率就越高，但是编译的过程花费的时间就会越长。

-O0 这个等级关闭所有的优化选项。

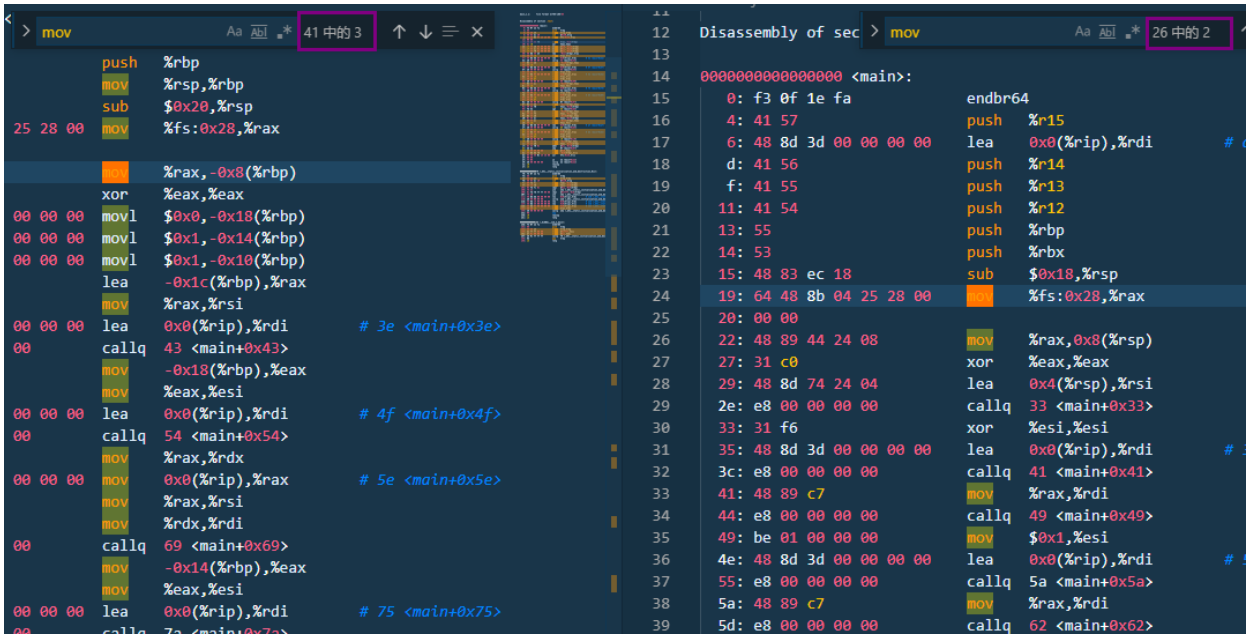
-O1 是最基本的优化等级，编译器会在不花费太多编译时间的同时试图生成更快更小的代码。

-O2 会比-O1 启用多一些标记，是比较推荐的优化等级。设置了-O2 后，编译器会试图提高代码

性能而不会增大体积和大量占用的编译时间。

-O3 是最高的优化等级，虽然最后生成的代码的执行效率就会很高，但是也存在一定的危险性。这个选项会延长编译代码的时间，并大大增加了编译失败的机会，不太推荐。

可以看到，-O 后面的的数字越大，产生的代码越优化，但是优化级别越高，虽然最后生成的代码的执行效率会越高，但是编译的过程花费的时间就会越长。所以我们需要在执行效率和编译时间之间做出一个权衡。**下面是进行优化之后的斐波那契反汇编代码，发现三个级别不同程度的优化对简单的源码来说差别不大**



通过汇编代码的比较，前面每次调用一个函数之后，先压栈，然后又转到寄存器中，这样消耗了一部分时间。我们使用-O 选项进行优化，优化后生成的汇编代码如下，可以看到函数的返回值并没有经过入栈和出栈的过程。可以明显看出完全优化之后减少了 move 的次数，提高了代码的执行效率。

4 结论

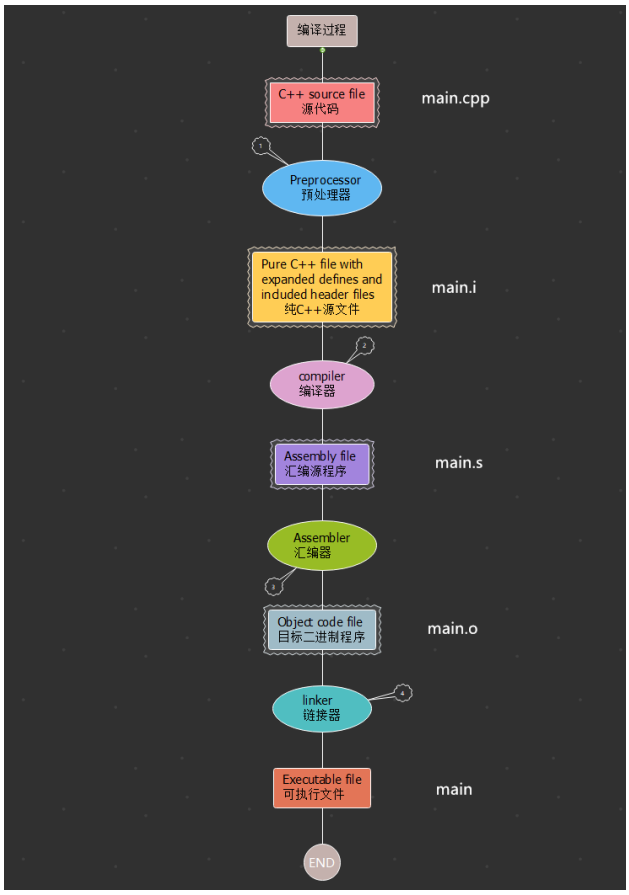
编译流程如下：

预处理 ： 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

（1）宏定义指令，如 #define Name TokenString，#undef 等。对于前一个伪指令，预编译所要做的是将程序中的所有 Name 用 TokenString 替换，但作为字符串常量的 Name 则不被替换。对于后者，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 <FileName> 等。在头文件中一般用伪指令 #define 定义了大量的宏（最常见的是字符常量）



编译　：

词法分析　将单词序列转换为单词序列

语法分析　将词法分析生成的词法单元来构建语法树

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

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。

代码生成　以中间表示形式作为输入，将其映射到目标语言

汇编　汇编过程实际上指把汇编语言代码翻译成目标机器指令的过程。对于被翻译系统处理的每一个 C 语言源程序，都将最终经过这一处理而得到相应的目标文件。目标文件中所存放的也就是与源程序等效的目标的机器语言代码。目标文件由段组成。通常一个目标文件中至少有两个段：代码段：该段中所包含的主要是程序的指令。该段一般是可读和可执行的，但一般却不可写。数据段：主要存放程序中要用到的各种全局变量或静态的数据。一般数据段都是可读，可写，可执行的。

链接器 链接器（Linker）是一个程序，将一个或多个由编译器或汇编器生成的目标文件外加库链接为一个可执行文件。

个人感悟 写到这里是个漫长曲折的过程。我不禁感叹道：在伟大的编译器面前，我要学会谦卑。

参考文献

[1] https://www.cnblogs.com/mickole/articles/3659112.html

[2] 吴元斌.C 语言程序的理解与编译优化 [D].2020-10-02

[3] 编译原理 [M]. 机械工业出版社, (美) 阿霍 (Alfred, 2009

[4] https://blog.csdn.net/elfprincexu/article/details/45043971

[5] https://astexplorer.net/