南 开 大 学

编译原理调研报告

中文题目：gcc 编译过程

外文题目：Report of gcc compiling

学　　号：1712991

姓　　名：周辰霏

专　　业：计算机科学与技术

摘　要

本文主要以gcc/g++编译器为研究对象，以最基本的C++源程序为例，借助MacOS操作系统下Xcode、terminal的编译器程序选项以及Hopper反编译，较为深入的调研了语言处理系统的完整工作流程：预处理→编译→汇编→链接→可执行程序。并对编译器和gcc的发展史、gcc/g++调试选项以及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探究。

关键词：　gcc/g++编译器；反汇编；C++；优化；调试

目录

gcc 编译过程

引　言

编译器可以理解为一种语言处理系统, 它是最早由葛丽丝·霍普提出的一种计算机程序，能将一种编程语言（通常为高级语言）写成的源程序，翻译为另一种编程语言（通常为低级语言）的可执行程序。其主要目的是将便于人编写，阅读和维护的高级语言写成的源程序, 翻译为计算机能解读并运行的低级机器语言写成的程序，即可执行文件。现代编译器主要工作流程是：源代码(source code)→预处理器 (preprocessor)→编译器(compiler)→目标代码(object code)→链接器(Linker) →可执行程序(executables)。

gcc/g++编译器把对应的*.c/*.cpp预处理为*.i文件，再编译成*.s文件(汇编代码), 编译器处理*.s生成对应目标代码*.o文件(obj目标文件)，最后链接器把全部*.o文件链接成可执行文件。

一、背景：编译器发展概况

（一）编译器发展史

编译器的发展史可以大致概括为“五个十年”[1]。

世界上第一个编译器是由葛丽丝·霍普于 1952 年为 A-0 系统编写的，但第一个具备完整功能的编译器却是 1957 年由时任职于 IBM 的约翰·巴克斯领导开发的 FORTRAN 编译器[1]。

1．20世纪60年代

计算机架构师和编译器编写者首先开始考虑**并行性**。面向 STAR-100 的 FORTRAN 编译器添加了用于描述长连续向量操作的语法。TI ASC 拥有了第一个自动向量化编译器，提升了当时编译器分析的最新水平。

2．20世纪70年代

Cray-1 成为第一台商业上大获成功的超级计算机, 其成功很大程度上归功于引入了**向量寄存器**，Cray Research 还开发了一种向量化编译器，它在许多方面与早期的 TI 编译器类似，此外最重要的一项功能是能为程序员提供**编译器反馈**。

开发人员可以启用**编译器优化**标志，让生成的可执行文件运行得更快，Cray 程序员实现了性能、生产力和可移植性这三个目标。[1]

3．20世纪80年代

此时**多处理**已得到广泛实施和使用，32 位单片微处理器的问世推动多处理技术进入了主流。Sequent、Encore 和 SGI 都构建了有一个微处理器、性能出色的系统。

不像大获成功的自动向量化，自动并行化基本上一败涂地。它适用于最内层循环，但要实现大幅的并行加速，通常需要对外层循环进行并行化，外层循环又增添了控制流的复杂性。由此产生了各种针对特定供应商的面向并行循环的指令集。同样促成了消息传递库。

4．20世纪90年代

所有消息传递库都被消息传递接口(MPI)取代，出现了扩展性更强的并行系统，如 Thinking Machines CM-5。

所有针对特定供应商的并行化指令被 OpenMP 取代。其宗旨是以同样的方式彰显“**并行性可以**”。

同时，市面上出现了面向单芯片微处理器的 SIMD 指令集，比如来自 Intel 的 SSE 和 SSE2，此时编译器恢复了已有 25 年历史的向量化技术，以便自动利用 SIMD 指令。

5．2000年以后

众多供应商普遍开始提供多核微处理器。同时，异构 HPC 系统开始出现，附加的处理器大受欢迎，原因是它们可以比 CPU 更快地完成专门操作。它们常常能够以小型机的价格提供大型机的性能。

（二）gcc/g++

gcc是GNU编译器套件(GNU Compiler Collection0，它包括了C、C++、Objective-C、Fortran、Java、Ada、Go语言和D语言的前端，也包括了这些语言的库(如libstdc++、libgcj等等)。GCC的初衷是为GNU操作系统专门编写的一款编译器。GNU系统是彻底的自由软件。此处，“自由”的含义是它尊重用户的自由[2]。

1．部分gcc命令

(在Terminal中键入“gcc -help”可查看)

gcc -v　查看当前gcc版本

gcc -o <file>　输出名为file的文件

gcc -E　只进行预处理

gcc -S　进行预处理以及编译

gcc -c　进行预处理、编译以及汇编

gcc -g　在可执行程序里包含了调试信息,可用 gdb 调试

二、编译的过程

从源代码生成可执行文件共四个阶段：预处理阶段、编译阶段、汇编阶段和链接阶段。预处理器会去掉源代码中的预处理命令,编译器使用预处理器的输出文件生成汇编源文件,汇编器将汇编源文件转化为目标二进制文件，最后链接器把目标文件、操作系统的启动代码以及用到的库文件进行组织，形成最终的可执行代码[3]。过程图解如下:

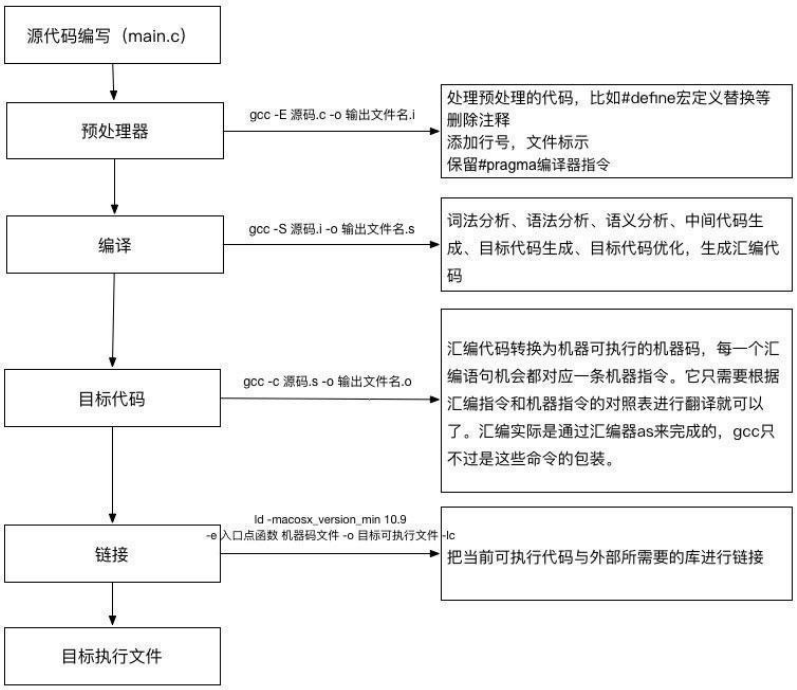


图 1 gcc/g++编译器工作流程

（一）预处理器

预处理器是在编译开始前由编译器调用的独立程序，预编译程序读出源代码，对其中内嵌的指示字进行响应，产生源代码的修改版本，修改后的版本会被编译程序读入[4]。

简单来说，预处理就是将要包含(#include)的文件插入原文件中、将宏定义展开、根据条件编译命令选择要使用的代码，最后将这些代码输出到一个 *.i 文件中等待进一步处理[5]。

预编译过程主要处理那些源代码文件中以“#”开始的预编译指令。比如

#include、#define 等，主要处理规则如下：

①删除#define，并展开其表示的宏

②处理条件预编译指令，如#if、#ifdef、#elif、#else、#endif 等

③处理#include 预编译指令，递归地将被包含的文件插入到该预编译指令的位置

④删除所有的注释

⑤添加行号和文件名标识，比如 #2 "hello.c" 2

⑥保留所有的 #pragma 编译器指令

经过预编译后的*.i 文件不包含任何宏定义，并且包含的文件也已经被插入到*.i 文件中。所以可以通过查看预编译后的文件来确定宏定义及头文件包含的正确性。

对 hello.c 进行预编译：gcc -E hello.c -o hello .i

1．**对**main.cpp**进行预编译**：

gcc -E main.cpp -o main.i （**或利用**Xcode**中**preprocess**选项**)

```
41080  # 41 "/Library/Developer/CommandLineTools/usr/include/c++/v1/iostream" 2 3
41081  # 45 "/Library/Developer/CommandLineTools/usr/include/c++/v1/iostream" 3
41082
41083
41084  namespace std {inline namespace __1 {
41085
41086
41087  extern __attribute__ ((__visibility__("default"))) istream cin;
41088  extern __attribute__ ((__visibility__("default"))) wistream wcin;
41089
41090
41091  extern __attribute__ ((__visibility__("default"))) ostream cout;
41092  extern __attribute__ ((__visibility__("default"))) wostream wcout;
41093
41094  extern __attribute__ ((__visibility__("default"))) ostream cerr;
41095  extern __attribute__ ((__visibility__("default"))) wostream wcerr;
41096  extern __attribute__ ((__visibility__("default"))) ostream clog;
41097  extern __attribute__ ((__visibility__("default"))) wostream wclog;
41098
41099  } }
41100  # 2 "main.cpp" 2
41101  using namespace std;
41102  int main()
41103  {
41104      int i, n, f;
41105      cin >> n;
41106      i = 2;
41107      f = 1;
41108      while (i <= n)
41109      {
41110          f = f * i;
41111          i = i + 1;
41112      }
41113      cout << f << endl;
41114  }
```

图 2　main.i 部分内容

可以看出多出的 40000 余行是将#include 的头文件部分进行了替换，经过预处理器的这种替换，最后生成了一个没有宏定义、编译指令以及特殊符号的预处理文件*.i。

（二）编译器

编译器对预处理得到的*.i 文件，进行词法分析、语法分析和语义分析，转

换生成中间代码并优化为目标汇编代码：

①词法分析：将字符序列转换为 token 的过程

②语法分析：利用词法分析器生成的 token 构造语法树

③语义分析：使用语法树和符号表中的信息检查源程序是否和语言定义的语义一样；进行类型检查

④中间代码生成：生成一个明确的低级的或类机器语言的中间表示

⑤代码优化：机器无关的代码优化步骤试图改进中间代码，以便生成更好的目标代码

⑥代码生成：将优化后的中间代码映射到目标语言

1. **对**main.i**进行编译**：

gcc -S main.i -o main.s　(**或利用** Xcode **中** assemble **选项**)

```
_main:                                  ## @main
    .cfi_startproc
## %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    subq    $32, %rsp
    movq    __ZNSt3__13cinE@GOTPCREL(%rip), %rdi
    movl    $0, -4(%rbp)
    leaq    -12(%rbp), %rsi
    callq   __ZNSt3__113basic_istreamIcNS_11char_traitsIcEEErsERi
    movl    $2, -8(%rbp)
    movl    $1, -16(%rbp)
    movq    %rax, -24(%rbp)         ## 8-byte Spill
LBB0_1:                                 ## =>This Inner Loop Header: Depth=1
    movl    -8(%rbp), %eax
    cmpl    -12(%rbp), %eax
    jg  LBB0_3
```

图 3　main.s 部分内容(main 函数部分汇编代码)

（三）汇编器

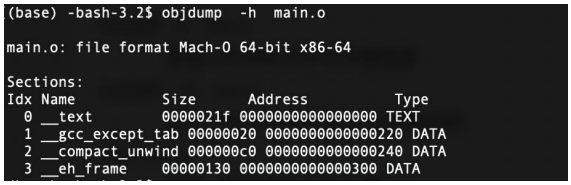
汇编器将汇编源代码汇编为目标机器指令文件，该阶段生成的文件为二进制文件。目标文件由段组成，通常一个目标文件中至少有两个段：代码段和数据段[6]。代码段中包含的是程序指令，一般是可读可执行不可写的;数据段存放程序中用到的全局变量或静态数据，一般数据段是可读可写可执行的。

1. **对**main.s**进行汇编**：

gcc -c main.s -o main.o

由于.o 文件无法直接打开，使用 objdump 命令查看 main.o 的详细信息。

①objdump -h main.o 打印主要段信息



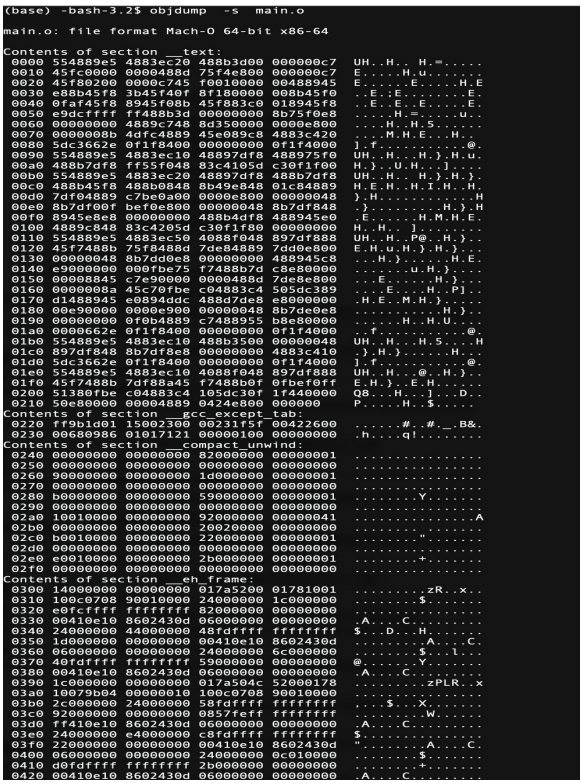
```
(base) -bash-3.2$ objdump  -h  main.o

main.o: file format Mach-O 64-bit x86-64

Sections:
Idx Name          Size      Address          Type
  0 __text        0000021f 0000000000000000 TEXT
  1 __gcc_except_tab 00000020 0000000000000220 DATA
  2 __compact_unwind 000000c0 0000000000000240 DATA
  3 __eh_frame    00000130 0000000000000300 DATA
```

图 4　main.o 主要段信息

②objdump -s main.o 将所有段的内容以 16 进制方式打印出来



```
(base) -bash-3.2$ objdump  -s  main.o

main.o: file format Mach-O 64-bit x86-64

Contents of section __text:
 0000 554889e5 4883ec20 488b3d00 000000c7  UH..H.. H.=.....
 0010 45fc0000 00004881 75f4e800 000000c7  E.....H.u.......
 0020 45f80200 0000c745 f0010000 00488945  E......E.....H.E
 0030 e88b45f8 3b45f40f 8f180000 008b45f0  ..E.;E........E.
 0040 0faf45f8 8945f08b 45f883c0 018945f8  ..E..E..E.....E.
 0050 e9dcffff ff488b3d 00000000 8b75f0e8  .....H.=.....u..
 0060 00000000 4889c748 8d350000 0000e800  ....H..H.5......
 0070 0000008b 4dfc4889 45e089c8 4883c420  ....M.H.E...H..
 0080 5dc3662e 0f1f8400 00000000 0f1f4000  ].f...........@.
 0090 554889e5 4883ec10 48897df8 488975f0  UH..H...H.}.H.u.
 00a0 488b7df8 ff55f048 83c4105d c30f1f00  H.}..U.H...]....
 00b0 554889e5 4883ec20 48897df8 488b7df8  UH..H.. H.}.H.}.
 00c0 488b45f8 488b0848 8b49e848 01c84889  H.E.H..H.I.H...H.
 00d0 7df04889 c7be0a00 0000e800 00000048  }.H............H
 00e0 8b7df00f bef0e800 00000048 8b7df848  .}.........H.}.H
 00f0 8945e8e8 00000000 488b4df8 488945e0  .E......H.M.H.E.
 0100 4889c848 83c4205d c30f1f80 00000000  H..H. ]........
 0110 554889e5 4883ec50 4088f048 897df888  UH..H..P@..H.}..
 0120 45f7488b 75f8488d 7de84889 7dd0e800  E.H.u.H.}.H.}...
 0130 00000048 8b7dd0e8 00000000 488945c8  ...H.}......H.E.
 0140 e9000000 000fbe75 f7488b7d c8e80000  .......u.H.}....
 0150 00008845 c7e90000 0000488d 7de8e800  ...E......H.}...
 0160 0000008a 45c70fbe c04883c4 505dc389  ....E....H..P]..
 0170 d1488945 e0894ddc 488d7de8 e8000000  .H.E..M.H.}.....
 0180 00e90000 0000e900 00000048 8b7de0e8  ...........H.}..
 0190 00000000 0f0b4889 c7488955 b8e80000  .......H..H.U...
 01a0 0000662e 0f1f8400 00000000 0f1f4000  ..f...........@.
 01b0 554889e5 4883ec10 488b3500 00000048  UH..H...H.5....H
 01c0 897df848 8b7df8e8 00000000 4883c410  .}.H.}......H...
 01d0 5dc3662e 0f1f8400 00000000 0f1f4000  ].f...........@.
 01e0 554889e5 4883ec10 4088f048 897df888  UH..H...@..H.}..
 01f0 45f7488b 7df88a45 f7488b0f 0fbef0ff  E.H.}..E.H......
 0200 51380fbe c04883c4 105dc30f 1f440000  Q8....H...]...D..
 0210 50e80000 00004889 0424e800 000000    P.....H..$.....
Contents of section __gcc_except_tab:
 0220 ff9b1d01 15002300 00231f5f 00422600  ......#..#._.B&.
 0230 00680986 01017121 00000100 00000000  .h....q!........
Contents of section __compact_unwind:
 0240 00000000 00000000 82000000 00000001  ................
 0250 00000000 00000000 00000000 00000000  ................
 0260 90000000 00000000 1d000000 00000001  ................
 0270 00000000 00000000 00000000 00000000  ................
 0280 b0000000 00000000 59000000 00000001  ........Y.......
 0290 00000000 00000000 00000000 00000000  ................
 02a0 10010000 00000000 92000000 00000041  ...............A
 02b0 00000000 00000000 20020000 00000000  ........ .......
 02c0 b0010000 00000000 22000000 00000001  ........".......
 02d0 00000000 00000000 00000000 00000000  ................
 02e0 e0010000 00000000 2b000000 00000001  ........+.......
 02f0 00000000 00000000 00000000 00000000  ................
Contents of section __eh_frame:
 0300 14000000 00000000 017a5200 01781001  .........zR..x..
 0310 100c0708 90010000 24000000 1c000000  ........$.......
 0320 e0fcffff ffffffff 82000000 00000000  ................
 0330 00410e10 8602430d 06000000 00000000  .A....C.........
 0340 24000000 44000000 48fdffff ffffffff  $...D...H.......
 0350 1d000000 00000000 00410e10 8602430d  .........A....C.
 0360 06000000 00000000 24000000 6c000000  ........$...l...
 0370 40fdffff ffffffff 59000000 00000000  @.......Y.......
 0380 00410e10 8602430d 06000000 00000000  .A....C.........
 0390 1c000000 00000000 017a504c 52000178  .........zPLR..x
 03a0 10079b04 00000010 100c0708 90010000  ................
 03b0 2c000000 24000000 58fdffff ffffffff  ,...$...X.......
 03c0 92000000 00000000 0857feff ffffffff  .........W......
 03d0 ff410e10 8602430d 06000000 00000000  .A....C.........
 03e0 24000000 e4000000 c8fdffff ffffffff  $...............
 03f0 22000000 00000000 00410e10 8602430d  ".........A....C.
 0400 06000000 00000000 24000000 0c010000  ........$.......
 0410 d0fdffff ffffffff 2b000000 00000000  ........+.......
 0420 00410e10 8602430d 06000000 00000000  .A....C.........
```

图 5　main.o　16 进制显示

③size main.o 查看.o 文件中各个段所占大小

```
(base) -bash-3.2$ size main.o
__TEXT   __DATA   __OBJC  others  dec     hex
879      0        0       192     1071    42f
```

图 6　main.o 各段占大小

（四）链接器

链接器是一个程序, 它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[7]。目标文件是包括机器码和链接器可用信息的程序模块，即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。链接器还完成了程序中各目标文件的地址空间的组织[7]。

链接器处理方式可分为两种：静态链接和动态链接。

1. **动态链接**(gcc/g++**默认使用动态链接**)：

g++ main.cpp -o main_d.o

动态链接使用动态链接库进行链接，生成的程序在执行的时候需要加载所需的动态库才能运行。动态链接生成的程序体积较小，但是必须依赖所需的动态库，否则无法执行[7]。

```
(base) -bash-3.2$ objdump -h main.o

main.o: file format Mach-O 64-bit x86-64

Sections:
Idx Name          Size      Address          Type
  0 __text        0000021f 0000000000000000 TEXT
  1 __gcc_except_tab 00000020 0000000000000220 DATA
  2 __compact_unwind 000000c0 0000000000000240 DATA
  3 __eh_frame    00000130 0000000000000300 DATA
(base) -bash-3.2$ objdump -h main_d.o

main_d.o:       file format Mach-O 64-bit x86-64

Sections:
Idx Name          Size      Address          Type
  0 __text        0000023f 0000000100000c80 TEXT
  1 __stubs       0000003c 0000000100000ec0 TEXT
  2 __stub_helper 00000074 0000000100000efc TEXT
  3 __gcc_except_tab 00000020 0000000100000f70 DATA
  4 __unwind_info 00000068 0000000100000f90 DATA
  5 __nl_symbol_ptr 00000010 0000000100001000 DATA
  6 __got         00000020 0000000100001010 DATA
  7 __la_symbol_ptr 00000050 0000000100001030 DATA
```

图 7　动态链接前后主要段信息对比

```
(base) -bash-3.2$ size main.o
__TEXT  __DATA  __OBJC  others  dec     hex
879     0       0       192     1071    42f
(base) -bash-3.2$ size main_d.o
__TEXT  __DATA  __OBJC  others  dec     hex
4096    4096    0       4294971392      4294979584      100003000
```

图 8　动态链接前后各段占大小对比

16 进制显示动态链接后字符信息更多更紧密，删去了无用信息。

2. **静态链接：**

g++ -static -o main_static.o main.o

静态链接使用已有静态库进行链接，生成的程序包含程序运行所需要的全部库，可以直接运行，一般静态链接生成的程序体积较大。

```
(base) -bash-3.2$ objdump -h main_d.o

main_d.o:       file format Mach-O 64-bit x86-64

Sections:
Idx Name          Size      Address          Type
  0 __text        0000023f 0000000100000c80 TEXT
  1 __stubs       0000003c 0000000100000ec0 TEXT
  2 __stub_helper 00000074 0000000100000efc TEXT
  3 __gcc_except_tab 00000020 0000000100000f70 DATA
  4 __unwind_info 00000068 0000000100000f90 DATA
  5 __nl_symbol_ptr 00000010 0000000100001000 DATA
  6 __got         00000020 0000000100001010 DATA
  7 __la_symbol_ptr 00000050 0000000100001030 DATA
(base) -bash-3.2$ objdump -h main_static.o

main_static.o:  file format COFF-i386

Sections:
Idx Name          Size      Address          Type
  0 .text         0009ff9c 0000000000401000 TEXT DATA
  1 .data         00001ac8 00000000004a1000 DATA
  2 .rdata        0000a2bc 00000000004a3000 DATA
  3 .eh_frame     00038500 00000000004ae000 DATA
  4 .bss          00000000 00000000004e7000 BSS
  5 .idata        00000b24 00000000004e8000 DATA
  6 .CRT          00000018 00000000004e9000 DATA
  7 .tls          00000020 00000000004ea000 DATA
  8 .debug_aranges 000000d8 00000000004eb000 DATA
  9 .debug_info   00014e37 00000000004ec000 DATA
 10 .debug_abbrev 000013b8 0000000000501000 DATA
 11 .debug_line   00001f23 0000000000503000 DATA
 12 .debug_frame  00000038 0000000000505000 DATA
 13 .debug_str    000002ae 0000000000506000 DATA
 14 .debug_loc    0000829a 0000000000507000 DATA
 15 .debug_ranges 00000cd8 0000000000510000 DATA
```

图 9　动静态链接前后主要段信息对比

```
(base) -bash-3.2$ size main_d.o
__TEXT  __DATA  __OBJC  others  dec     hex
4096    4096    0       4294971392      4294979584      100003000
(base) -bash-3.2$ size main_static.o
   text    data     bss     dec     hex filename
 655260  418338       0 1073598  1061be main_static.o
```

图 10　动静态链接前后各段大小对比

16 进制显示至少上万行，而通过上述两图也可以看出静态链接生成的文件过于冗余复杂，很多部分是没有必要的。

（五）gcc 优化选项

gcc 编译器共有四个优化级别:O0(不进行任何优化)、O1、O2(Os)、O3，优化级别越高，产生的代码的执行效率就越高，相对的编译过程花费时间就会越长。

-O0 关闭所有优化选项；

-O1 最基本的优化等级，在不影响编译速度的前提下，尽量采用一些优化算法降低代码大小和可执行代码的运行速度；

-O2 会比-O1 启用更多优化标记，是较为推荐的优化等级，编译器会牺牲部分编译时间采用几乎所有的目标配置支持的优化算法，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 最高的优化等级，采取很多向量化算法，如利用现代 CPU 中的流水线、Cache 等，虽然最后生成的代码的执行效率很高，但会延长编译代码的时间，并大大增加了编译失败的机会；

-Os 相当于 $O_{2.5}$，即使用全部 O2 优化选项但不缩减文件大小的优化；

综上，-O 后面的的数字越大，产生的代码越优化。随着代码优化级别提高，生成代码的执行效率提高，但编译花费的时间也随之增长。所以我们在优化的时候要做好权衡。

1. 优化

对 main.cpp 进行不同级别的优化后查看编译时间以及编译文件最终大小只有细微的区别，所以利用反汇编查看代码如下：

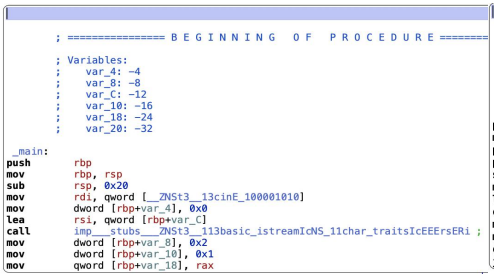


图11 无优化反编译

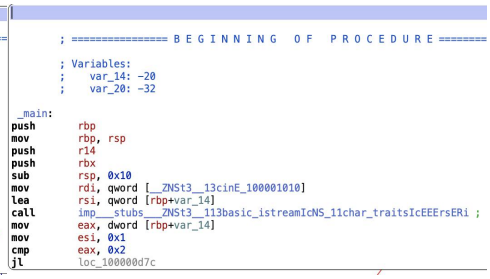


图12 O3优化反编译

由上两图可以看出，虽然代码长度变长了，但是减少了不必要的中间变量，减少了mov的次数，提高了效率；且展开反汇编代码进程查看发现O3优化以后的程序考虑了多种循环跳转，减少了循环判断情况，也提高了效率，寄存器空间也得到了节约。

	O0	O1	O2	O3	Os
编译时间/ms	400	600	600	570	650
程序运行时间	1570	1340	960	920	890

表1 不同优化标记编译运行时间

2. 在程序中加入空循环(细微修改查看变化)

```
6       for(int j=0;j<100;j++)
7       {}
```

图13　空循环样例

在不经过优化的直接编译得到的文件中多出了如下部分对空循环的操作。

```
0000000100000c96        cmp         dword [rbp-0x14], 0x64
0000000100000c9a        jge         _main+51
0000000100000ca0        jmp         _main+37
0000000100000ca5        mov         eax, dword [rbp-0x14]
0000000100000ca8        add         eax, 0x1
0000000100000cab        mov         dword [rbp-0x14], eax
0000000100000cae        jmp         _main+22
0000000100000cb3        mov         rdi, qword [__ZNSt3__13cinE_100001010]
```

图14　空循环反汇编代码

而在O1、O2、O3优化后的代码中就省去了上图部分的无用空循环。

（六）gcc 调试选项

gcc 调试选项 gcc -g 表示在编译的时候产生调试信息；

gdb 调试可以实现以下功能：①设置断点 b <行号>②监视程序变量的值 display③程序的单步执行 s④显示、修改变量的值 set var⑤显示、修改寄存器⑥查看程序的堆栈情况 bt⑦远程调试[8]；

1. **简单调试**

g++ -g main.cpp　**(打开编译选项)**
gdb a.out**(启动调试)**
l**(显示当前代码)(一次只显示 10 行，继续键入 l 继续显示)**
b **行号(加断点)**

```
(gdb) l
11          while (i <= n)
12          {
13              f = f * i;
14              i = i + 1;
15          }
16          cout << f << endl;
17      }
(gdb) b 13
Breakpoint 1 at 0x100000c91: file main.cpp, line 13.
```

图 15　在第 13 行加入断点

r**(运行)**
display f　**跟踪变量 f**
s**(逐行执行)**
print **变量名(打印变量当前值)**

r(运行)后输入 5，输入 display f 跟踪 f 的值，按 s 逐行执行，在第三次到达断点时 print f，输出 f 的值为 6,输入 bt 可以看到主函数部分只在 main 函数入口处使用了栈。

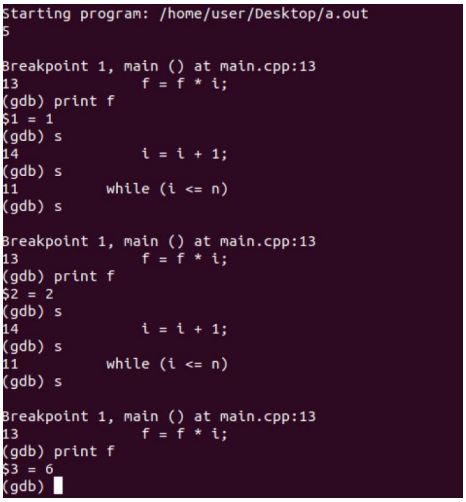


图 16　调试过程

2. **细微修改调试 bug(改变判定条件)**

将原代码 while 循环的判断语句改为>，这导致所有的输入的输出都为 1，利调试选项在容易出错的循环附近——13 行设置断点，逐行执行后发现根本没有进入循环，查看循环条件并更正后再次实验得到正确的结果。

三、结论

编译器前 50 年的发展基本实现了现代编译器所包含的全部功能, 从最早的实现编译功能到后来的调试优化功能和自动并行化等，而 gcc/g++作为其中功能强大的代表，拥有完整的编译器结构和功能。一个 cpp 源代码生成可执行文件需要经过预处理、编译、汇编、链接四个步骤：预处理处理替换以 “#”开头的指令、编译生成汇编文件、汇编生成二进制文件、链接生成可执行文件；同时 gcc 还可以对编译进行调试和优化：调试可以帮助程序员更清楚的了解程序的编译执行步骤以及更好的定位 bug 所在；而优化有不同级别，可通过开关不同的优化标记在不同程度上提高代码执行效率, 但与此同时它也会降低编译效率。

参考文献

[1] 编译器发展史5个十年[EB/OL].https://blog.csdn.net/qq_28260611/article/details/84261057,2018-11-19.

[2] GCC 官网[EB/OL].http://gcc.gnu.org.

[3] 现代编译器基本工作流程[EB/OL].https://www.jianshu.com/p/0134c0db14e7,2016-06-07.

[4] GCC 编译器原理（三）——编译原理三：编译过程--- 预处理[EB/OL].https://www.cnblogs.com/kele-dad/p/9490640.html,2018-08-16.

[5] 张贺. 2.4G 无线传感器网络监控平台设计[D]. 2009.

[6] 最简单的Hopper Disassembler玩转Mac逆向[EB/OL].https://www.jianshu.com/p/c04ac36c6641,2017-09-06.

[7] gcc程序的编译过程和链接原理[EB/OL].https://blog.csdn.net/czg13548930186/article/details/78331692,2017-10-24.

[8] 一个完整的gdb调试过程以及一些常用的命令[EB/OL].https://blog.csdn.net/qq_37941471/article/details/81476942,2017-08-07.