CS29206 Systems Programming Laboratory
Spring 2024

Introduction to gcc

Abhijit Das
Pralay Mitra

What you do not know about gcc

- What does the gcc compiler do?
- What are header files? Why should one #include them?
- Why should programs with math functions be compiled with the *–lm* flag?
- What are the compile-time options for gcc?
- How can C programs communicate with the shell?
- What is the C preprocessor?
- How can one write a program in multiple input files?
- What are libraries?
- How can one write one's own libraries?

CS29206 Systems Programming Laboratory
Spring 2024

The Compilation Process and Runtime Loading

Abhijit Das
Pralay Mitra

The four-stage compilation process

Preprocessing This involves the processing of the # directives. Examples:

- The #include'd files are inserted in your code.
- The #define'd macros are literally substituted throughout your code.

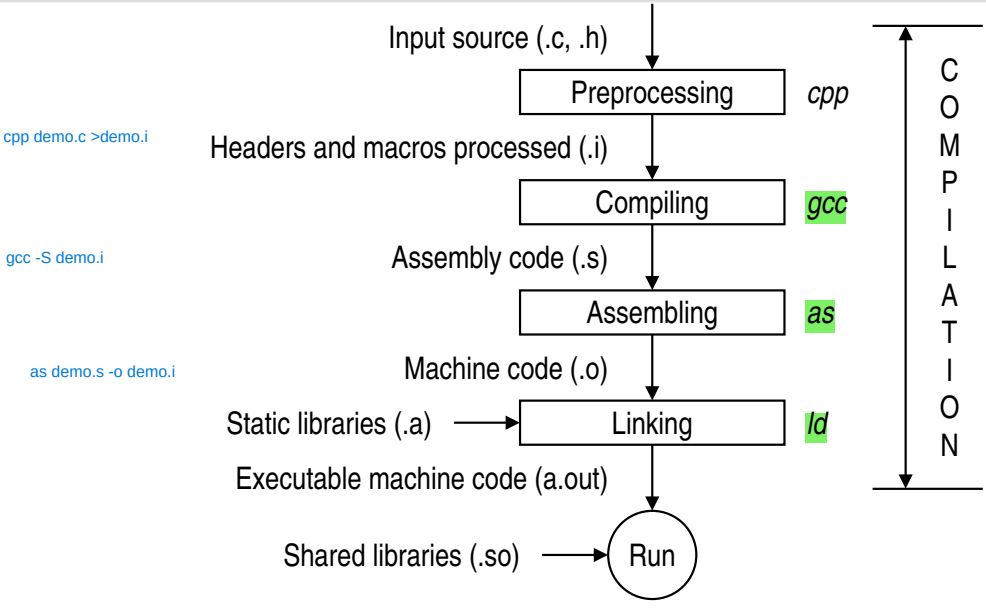
Compiling The input to this process is the preprocessed C file, and the output is an assembly-language code targeted to the architecture of your machine.

Assembling The assembly-language code generated by compiling is converted to a machine code called the object file. The external functions (like printf and sqrt) are still undefined.

Linking The object file(s) is/are eventually converted to an executable file in this process. At this point, the external functions from C runtime library and other libraries are included in the executable file.

Loading Some functions available in shared (or dynamic) libraries are loaded during runtime from shared object files.

The compilation process in a nutshell



An example of the four-stage compilation process

The file demo.c

```
#include <stdio.h>
#include <stdlib.h>

#define TEN 10
#define TWENTY 20

int main ( )
{
    int a, b, c;

    a = TEN;
    b = a + TWENTY;
    c = a * b;
    printf("c = %d\n", c);
    exit(0);
}
```

Preprocessing

The C preprocessor is called *cpp*.

```
$ cpp demo.c > demo.i
$ cat demo.i
...
typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;
...
# 7 "demo.c"
int main ( )
{
    int a, b, c;

    a = 10;
    b = a + 20;
    c = a * b;
    printf("c = %d\n", c);
    exit(0);
}
$
```

Compiling

This needs invoking gcc with the –S flag. A file with extension .s is generated.

```
$ gcc -S demo.i
$ cat demo.s
        .file   "demo.c"
        .text
        .section        .rodata
.LC0:
        .string "c = %d\n"
        .text
        .globl  main
        .type   main, @function
main:
.LFB6:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    $10, -12(%rbp)
        movl    -12(%rbp), %eax
        addl    $20, %eax
        movl    %eax, -8(%rbp)
        movl    -12(%rbp), %eax
        imull   -8(%rbp), %eax
        movl    %eax, -4(%rbp)
        movl    -4(%rbp), %eax
        movl    %eax, %esi
        leaq    .LC0(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        movl    $0, %edi
        call    exit@PLT
        .cfi_endproc
...
$
```

PLT means Procedure Linkage Table.
These functions are for runtime loading.

Assembling

- The assembler is called *as*.
- The symbols in object files are listed by *nm*.

```
$ as demo.s -o demo.o
$ nm demo.o
                 U exit
                 U _GLOBAL_OFFSET_TABLE_
0000000000000000 T main
                 U printf
```

- printf and exit are undefined in this object file.

Linking

- This is done by *ld*.
- This requires many libraries and is complicated.
- gcc does it transparently for you.

```
$ gcc demo.o
$ ./a.out
c = 300
$ nm a.out
...
```

- You get a big list of defined symbols.
- printf and exit are still left undefined.

```
...
                 U exit@@GLIBC_2.2.5
...
                 U printf@@GLIBC_2.2.5
...
```

Runtime loading

- printf and exit are loaded from shared object(s) during runtime.

```
$ ldd a.out
        linux-vdso.so.1 (0x00007ffe80ff2000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f98b5e19000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f98b602d000)
$
```

- If you want these functions to be in your executable, compile with the –static flag.
- This creates a huge a.out.
- You can see printf and exit defined in the executable.

```
$ gcc -static demo.o
$ ldd a.out
        not a dynamic executable
$ nm a.out | grep printf
...
0000000000410bb0 T printf
...
$
```