GCC Tutorial

1. Introduction

This tutorial intends to help you understand how to use gcc in the Linux environment. It will introduce the basic usage of gcc, how to build a program with multiple source files, and important options for using gcc.

2. First example

① Make a hello.c file as in the following:

```
#include <stdio.h>

void print_hello(void);

int main(void)
{
    print_hello();
    return 0;
}

void print_hello(void)
{
    printf("Hello World\n");
    return;
}
```

hello.c

② Compile it using gcc and run the executable file a.out. a.out is the default name of the generated executable file in case you don't explicitly specify the executable name.

```
$ gcc hello.c
$ ./a.out
Hello World
```

③ You can change the name of the executable file using –o option.

```
$ gcc -o hello hello.c
$ ./hello
Hello World
```

④ You can break the compilation process into two steps, which are compile and link. You can use –c option to let gcc generate the object file instead of the executable file. Then you can use the object file to generate the executable file in the second step.

```
$ gcc -c hello.c
$ gcc -o hello hello.o
$ ./hello
Hello World
```

3. Multiple source files

① Make the following files:

```c
#include <stdio.h>
#include "func.h"
int main(void)
{
    print_hello();
    return 0;
}
```

main.c

```c
#include <stdio.h>
#include "func.h"
void print_hello(void)
{
    printf("Hello World\n");
    return;
}
```

func.c

```c
#ifndef __FUNC_H__
#define __FUNC_H__
void print_hello(void);
#endif
```

func.h

② Compile main.c file. It will generate main.o file.

```
$ gcc -c main.c
```

③ Compile func.c file. It will generate func.o file.

```
$ gcc -c func.c
```

④ Link main.o and func.o files to generate the hello executable file. Then run hello.

```
$ gcc -o hello main.o func.o
$ ./hello
$ Hello World
```

※. Note that although there can be multiple c files for a single executable file, there must be only one c file which contains the main function. If you try to link multiple object files with two or more main functions, the compiler will generate a error message. Just give it a try.

4. Advanced gcc options

- -S option generates the assembly code instead of its executable file. The generated assembly code is contained in the corresponding .s file.

```
$ gcc -S hello.c
$ cat hello.s
        .file   "hello.c"
        .section        .rodata
.LC0:
        .string"Hello World!"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $.LC0, %edi
        call    puts
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 4.8.2-19ubuntu1) 4.8.2"
        .section        .note.GNU-stack,"",@progbits
```

Assembly code

- -E option just expands the included header files and macros but do not compile the source file.

```
$ gcc -E main.c
# 1 "main.c"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
…
# 943 "/usr/include/stdio.h" 3 4

# 2 "main.c" 2
# 1 "func.h" 1


void print_hello(void);
# 3 "main.c" 2
int main(void)
{
```

From

stdio.h and func.h

■ -I option indicates the locations where gcc searches for header files

```
$ gcc -c main.c
$ mkdir inc
$ mv func.h inc
$ gcc -c main.c
main.c:1:20: fatal error: func.h: No such file or directory
 #include "header.h"
                   ^
compilation terminated.
$ gcc -c -Iinc main.c
```

※. Note that if you use #include <header.h>, the compiler searches the predefined locations such as /usr/include directory to find the header.h file. Or if you use #include “header”, the compiler searches the current directory. Using –I option, you can put your header files in an arbitrary location and let the compiler know about it. In this case, you may use either <header.h> or “header.h” to specify them. Since <header.h> is more formal than “header.h”, you had better use <header.h>.

■ -D option defines a macro

```
#include <stdio.h>

int main(void)
{
#ifdef HELLO
    printf("Hello World\n");
#else
    printf("Goodbye World\n");
#endif
    return 0;
}
```

test.c

```
$ gcc test.c
$ ./a.out
Goodbye World
$ gcc -DHELLO test.c
$ ./a.out
Hello World
```

※. Note that with –DHELLO option, it’s like putting #define HELLO in the first line of the source code. If you use –DHELLO=3, it’s also like #define HELLO 3 in the first line.

5. Why should I put #ifdef things in header files?

```
#include "a.h"
#include "b.h"

int main(void)
{
    return 0;
}
```

source.c

```
#include "b.h"
```

a.h

```
struct point {
    int x;
    int y;
};
```

b.h

When you try to compile the program, you will encounter an error.

```
$ gcc source.c
b.h:1:8: error: redefinition of ‘struct point’
 struct point {
        ^
In file included from a.h:1:0,
                 from source.c:1:
b.h:1:8: note: originally defined here
 struct point {
        ^
```

You may use gcc –E to investigate the reason.

```
$ gcc -E source.c
# 1 "source.c"
…
struct point {
    int x;
    int y;
};
# 1 "a.h" 2
# 2 "source.c" 2
# 1 "b.h" 1
struct point {
    int x;
    int y;
};
# 3 "source.c" 2

int main(void)
{
    return 0;
`
```

Multiple definitions

To solve this problem, you always have to put #ifndef things in your header files to prevent multiple inclusions. Below is the modified files.

```c
#include "a.h"
#include "b.h"

int main(void)
{
    return 0;
}
```

source.c

```c
#ifndef __A_H__
#define __A_H__

#include "b.h"

#endif
```

a.h

```c
#ifndef __B_H__
#define __B_H__
struct point {
    int x;
    int y;
}
#endif
```

b.h

Let's see why this solve the problem. Blow is the source.c file after including a.h and b.h.