

LECTURE NOTES IN CIS300

YUZHE (RICHARD) TANG

SPRING, 2018

SECTION 2: C/C++ PROGRAMMING

REFERENCES

- "Unix Programming Tools", [[link](#)]
- Computer Systems: A Programmer's Perspective, Randal E. Bryant and David R. O'Hallaron, Chapter 1, [[online pdf](#)]

HELLOWORLD C

```
#include <stdio.h> //preprocessor
int y = 3; //global var. (def. & init.)
//extern int y; //global var. (dec.)
int main() //function (def.)
{
    int x = 0; //local var. (def. & init.), literal,
    printf("helloworld: y = %d\n",y); //function (invocation)
    return 0;
}
```

- printf: format string
- header files

LIFE OF A C CONSTRUCT

	variable	function
declare	<code>extern int x;</code>	<code>void foo();</code>
define	<code>int x;</code>	<code>void foo(){ }</code>
initialize	<code>int x=6;</code>	
reference	<code>y=x;x=1;</code>	<code>foo();</code> (invocation)
destroy		

COMPILATION & EXECUTION: BASICS

- GCC: GNU Compilation Collection
- In your terminal, run the following commands

```
gcc hello.c  
./a.out
```

EXERCISES

- Write a C program that prints out your name. Compile and execute it in Ubuntu. Submit the C program to BB.
- Write a C program that computes the sum of 1,2,3,...,956. Compile and execute the program in Ubuntu. Submit the C program to BB.

GCC

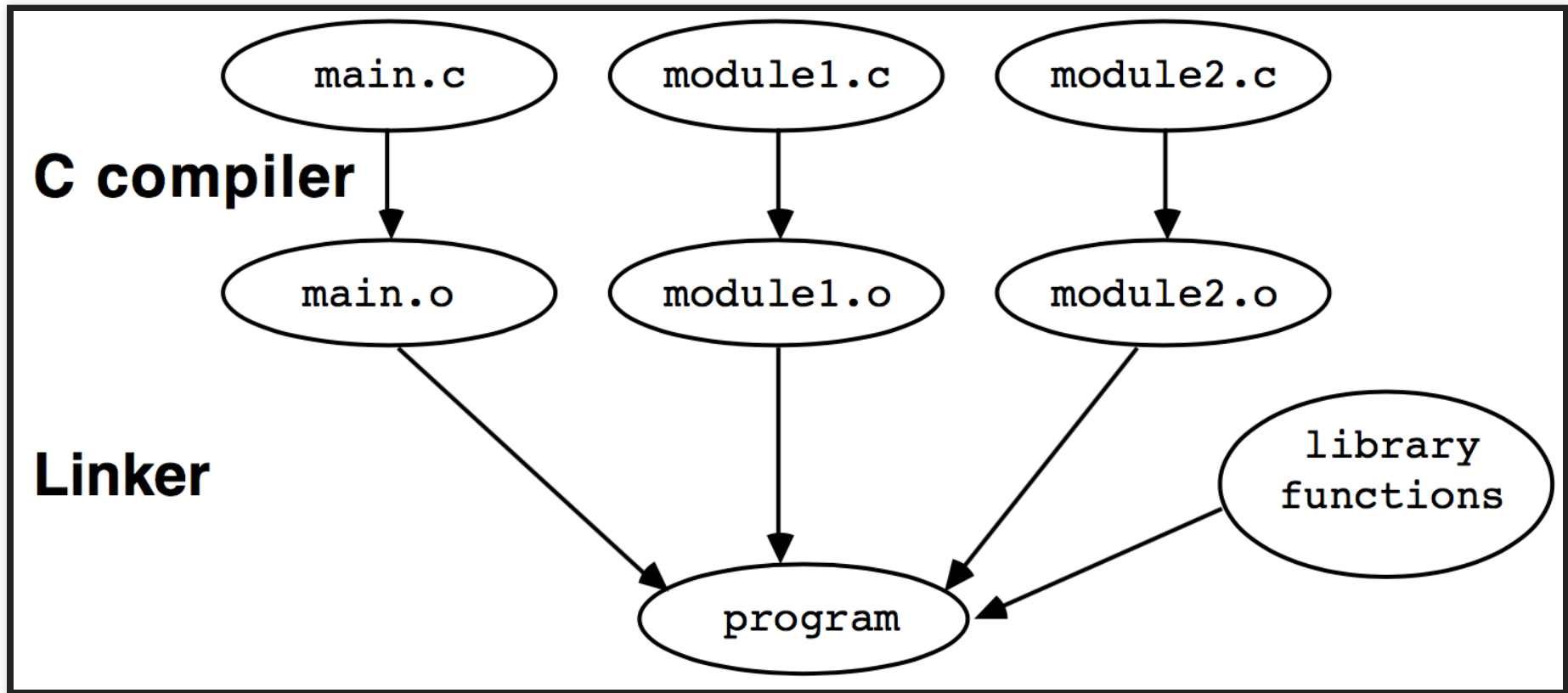
COMPILATION (1)

- Two steps of compilation:
 - *compiling*: text `.c` file to relocatable `.o` (object) file
 - *linking*: multiple relocatable `.o` files to one executable `.o` file
 - *symbol*: reference to link construct (declaration) in one `.o` file to construct (definition) in another `.o` file

COMPILATION (2)

```
gcc hello.c -o a.out  
gcc -S hello.c -o hello.s #compiler  
gcc -c hello.s -o hello.o #assembler  
gcc hello.o -o a.out #linker
```

- compilation system
 - tools: *gcc/gdb* for compiling and debugging
 - 1. **preprocessor**: from source file to source
 - 2. **compiler**: from source to assembly file
 - *assembly file*
 - 3. **assembler**: from assembly file to relocatable object file
 - 4. **linker**: from multiple objects to an executable object



Linker

COMPILING MULTIPLE C PROGRAMS

In file1.c:

```
#include <stdio.h>
extern void foo();
int main(){
    printf("main();\n");
    foo();
}
```

In file2.c:

```
#include <stdio.h>
void foo(){
    printf("foo();\n");
}
```

COMPILING MULTIPLE C PROGRAMS (2)

```
gcc file1.c file2.c  
# try this?  
gcc file1.c  
gcc file2.c
```

COMPILING MULTIPLE C PROGRAMS (3)

```
gcc -c file1.c # compiler & assembler  
gcc -c file2.c # compiler & assembler  
gcc file1.o file2.o # linker
```

Or

```
gcc -S file1.c # compiler  
gcc -c file1.s # assembler  
gcc -S file2.c # compiler  
gcc -c file2.s # assembler  
gcc file1.o file2.o # linker
```

LINK LIBRARY FILES

```
gcc -S file1.c # compiler  
gcc -c file1.s # assembler  
gcc file1.o file2.o # linker
```

```
mv file2.o ../libfile2.a  
gcc file1.o ../libfile2.a # linker  
gcc file1.o -L.. file2.o # linker  
gcc file1.c -L.. file2.o # linker
```

- Gcc flag: `-Ldir -lmylib` for library to link

INCLUDE HEADER FILE

In header1.h:

```
extern foo();
```

In file1.c:

```
#include <stdio.h>
#include "header1.h"
int main(){
    printf("main();\n");
    foo();
}
```

```
gcc file1.c file2.c
```


INCLUDE HEADER FILE (2)

Header file in another directory

```
mv header1.h ..  
#will this work?  
gcc file11.c file2.c  
gcc -I .. file11.c file2.c
```

- Gcc flag: `-I dir`

GCC FLAGS (SUMMARY)

- `-c` for compile, `-o` for output
- `-Ldir -lmylib` for linking a library
 - search library for unsolved symbols (functions, global variables) when linking
- `-I` for `#include`
 - header file (storing declarations)
- `-Wall, w` for warning
- `-g` for debug (later): `gcc -g file1.c file2.c`
- ref [[link](#)]

EXERCISE

- Write two C files:
 - `filea.c` defines functions `main()` and `bar()`
 - `fileb.c` defines function `foo()`
 - function `main()` calls `foo()`
 - function `foo()` calls `bar()`
 - Compile your program.
 - Submit the program and commands to BB.

MAKE AND MAKEFILE

DOWNLOAD COURSE REPO.

To download course repository, type the following commands

```
sudo apt-get update  
sudo apt-get upgrade  
sudo apt-get install git  
git clone https://github.com/SUCourses/cis300-18spring.git
```

MAKEFILE: DEPENDENCY RULES

- `make` is a tool for project management in shell
- `Makefile` is the configuration file that tells the `make` tool what to do
- A `Makefile` is a series of dependency rules
- Each dep. rule is a IFTTT clause (if-this-then-that)

```
target: files/objects  
    commands
```

It is a **tab** before the commands

HELLOWORLD MAKEFILE

In Makefile (All files are under demos/mar7 dir.)

```
all:
    gcc file1.c file2.c
```

To run it, in shell terminal

```
make
```

(Try change `file.c`, and make it again).

MAKEFILE OF MULTIPLE RULES

```
c:
    gcc file1.c file2.c

exec: c
    ./a.out

clean:
    rm *.o *.out
```


MAKEFILE OF MULTIPLE RULES (2)

Recall how to run compiler, assembler and linker

```
gcc -c file1.c # compiler & assembler  
gcc -c file2.c # compiler & assembler  
gcc file1.o file2.o # linker
```

A Makefile that does them separately

```
assem:  
    gcc -c file1.c  
    gcc -c file2.c  
  
link:  
    @gcc file1.o file2.o  
  
c: assem link  
  
exec: c  
    ./a.out  
  
clean:  
    rm *.o *.out
```

- @ used to hide the command in printout.

COMPACT MAKEFILE: USING VARIABLES

- variable represents strings of text
- standard variable: CC, CFLAGS, LDFLAGS
 - CC is the compiler
 - `OBJS = $(SRCS:.c=.o):`
 - This incantation says that the object files have the same name as the .c files, but with .o extension
 - LDFLAGS library search path (`-L`)

```
SRCS = file2.c file2.c
OBJS = $(SRCS:.c=.o)
```

```
exec: link
      ./a.out
```

```
link: $(OBJS)
      $(CC) $(OBJS)
```

EXERCISE

1. Write a `Makefile` such that `make` always clean `.o` files, recompiles all `.c` files and executes the new `.o` file.
2. Write a `Makefile` such that `make link` will compile a `file.c` file against a library file `libxxx.a`