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

```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	`extern int x;`	`void foo();`
define	`int x;`	`void foo(){}`
initialize	`int x=6;`	
reference	`y=x;x=1;`	`foo();` (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

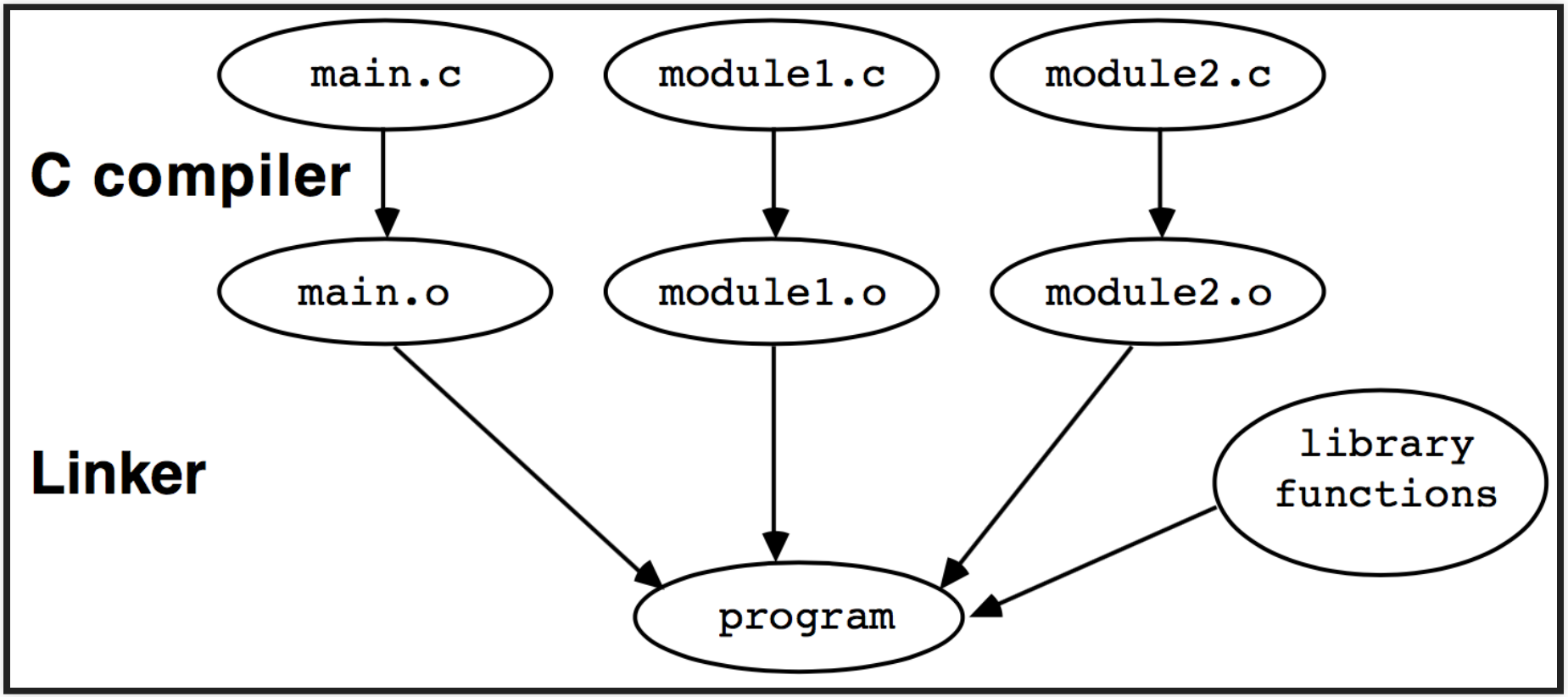
COMPILATION (1)

- 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


main.c
module1.c
module2.c
C compiler
main.o
module1.o
module2.o
Linker
library
functions
program

Linker

COMPILING MULTIPLE C PROGRAMS

In `file1.c`:

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

In `file2.c`:

```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

INCLUD HEADER FILE

In `header1.h`:

```
extern foo();
```

In `file11.c`:

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

```
gcc file11.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
(tab)commands
```

There 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
```

Note there are empty lines btwn. rules.

USE `MAKEFILE` TO LINK (1)

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
```

USE `MAKEFILE` TO LINK (2)

A `Makefile` that does them separately

```
link: file1.o file2.o
    gcc file1.o file2.o

file1.o: file1.c
    gcc -c file1.c

file2.o: file2.c
    gcc -c file2.c
```

```
make
make
```

USE `MAKEFILE` TO LINK (3)

Use default rule to compile individual C file

```
link: file1.o file2.o
    @gcc file1.o file2.o
```

```
make
make
```

- `@` used to hide the command in printout.

MAKEFILE: USING VARIABLES

```
SRCS = file1.c file2.c
OBJS = $(SRCS:.c=.o)
CFLAGS = -g -Iheaders
#LDFLAGS = -L. -lxxx

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

`MAKEFILE`: USING VARIABLES (2)

- A Makefile variable is a text string
- There're standard variables
  - `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`)
  - `CFLAGS` default compile flags

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`

GDB

REFERENCES

- "Reviewing gcc, make, gdb, and Linux Editors", [pdf]
- "Unix Programming Tools", [link]

A BUGGY C PROGRAM

```c
#include<stdio.h> //printf
int array_stack[] = {0,1,2};
int main(){
  int sum; // local variable
  for(int i=0; i<=3; i++){
    sum += array_stack[i];
  }
  printf("sum = %d\n", sum);
  return 0;
}
```

USE GDB TO FIND BUG

- Installing gdb
  - on MacOS: [youtu.be/Vj33vsrDkE80]
  - on Ubuntu: `sudo apt-get install gdb`
- Compile: `gcc -g`
- Run gdb: `gdb a.out`

GDB COMMAND: CONTROL EXECUTION

- CPU executes a C program statement by statement
- Breakpoint: tell where the CPU should stop/pause execution
  - `break/b file:n|fn|file:fn`: breakpoint can be file:line number, function name or file:function name.
  - `disable/enable/delete i`: i is the index of breakpoint
- Stepping: tell CPU to resume the execution
  - `run/r`: run
  - `next/n`: next statement (step over a function call)
  - `continue/c`: continue till breakpoint

GDB COMMAND: EXAMINE RUNTIME

- Examine runtime data
  - `print v`/`p v`: print variable `v`
- Examine code (with `gcc -g`)
  - `list`/`l`
- Examine execution environment: e.g. stack (later)

GDB COMMANDS

functionality	commands
breakpoints	`b`,`disable/enable/delete breakpoi`
stepping	`r`,`s`,`n`,`c`,`finish`,`return`
examine_data	`p/i v`,`display/undisplay`,`watch`,`set`
examine_code	`list`
examine_stack	`bt`,`where`,`info`,`up/down`,`frame`
misc.	`editmode vi`,`b fn if expression`,`h` disassembler, `shell cmd`

DEMO

- Debug the following program using gdb

```c
#include<stdio.h> //printf
int array_stack[] = {0,1,2};
int main(){
  int sum; // local variable
  for(int i=0; i<=3; i++){
    sum += array_stack[i];
  }
  printf("sum = %d\n", sum);
  return 0;
}
```

EXERCISE

- Exercise: Debug the following program using gdb, upload the correct program to BB.

```c
#include<stdio.h>
int main() {
    int x = 5;
    int y = 3;
    int z = x - y;
    int a = x * y;
    int b = a - 7*z;
    b--;
    int c = z + y;
    int d = c / b;
    int e = a + 12;
    int f = e - b;
    printf("%d\n",f);
}
```

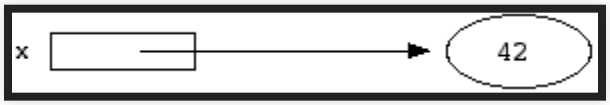
POINTER IN C

REFERENCES

- Pointer Basics: [http://cslibrary.stanford.edu/106/]

POINTER (C SYNTAX)

- a pointer is a variable that stores a reference to something.
  - "something", called pointee, is usually another variable.
- e.g.: a pointer variable named `x` referencing to a "pointee" variable of value `42`.



pointer pointee

POINTER OPERATIONS

- definition/initialization: `int *p1 = p2;`
- dereference: `*p`
- get reference of: `& a`
  - get the *address* (memory location) of variable a

```
#include<stdio.h>
int main(){
    int a = 10;
    int * p = & a;
    int b = *p;
    printf("a=%d,b=%d,*p=%d,p=%p\n",a,b,*p,p);
}
```

LIFE OF A C POINTER/SYMBOL

	pointer	variable	functic
declare	`extern int * p`	`extern int x`	`void`
define	`int *p;`	`int x`	`void`
initialize	`int *p=&a;`	`int x=6`	
	`int*q=malloc(7)`		
reference	`*p=x;x=*p`	`y=x`	`foo()`
destroy	`delete p`		

EXERCISE

- Do the following to complete the code snippet at the bottom. Then compile and execute your program. Submit the completed program to BB.
  1. define two pointers `p1` and `p2`, both pointing to variable `x`.
  2. Use `p1` to update `x`'s value to `5`.
  3. Then use `p2` to read the value of variable `x` and `printf` it on terminal.

```c
#include<stdio.h>
int main(){
    int x = 4;
    // To complete the program below:


}
```