LECTURE 4: C PROGRAMMING

Computer Systems and Networks

Dr. Pallipuram
(vpallipuramkrishnamani@pacific.edu)

University of the Pacific
Deadlines

Lab 2 September 12th, 2019
Lab 3 September 19th, 2019
Today’s Class

- The C compiler and Makefile
- `printf`, `scanf`, and format specifiers
- Arrays: one-dimensional and multi-dimensional
- Introduction to Pointers
A simple code compilation

```bash
unix> gcc -o myprog main.c
unix> ./myprog
```
Why So Many Compilation Steps?

We don’t just care about 1 language or 1 processor family!

- C
- C++
- Objective-C
- Fortran
- Ada
- Others...

GNU Compiler Collection

- x86
- x86-64
- ARM
- PowerPC
- 68000
- MIPS

(and many more!)
When your program has multiple files

unix> gcc main.c file2.c -o MyProg
unix> ./MyProg
Linker + Loader

CALL ProcA
  ...
  ...
CALL ProcC
  ...
  ...
CALL ProcB
  ...
  ...

MyProg.obj
(Main Program)

MyProg.exe

CALL ProcA
  ...
  ...
CALL ProcC
  ...
  ...
CALL ProcB
  ...
  ...

Main Memory

Loader

ProcC.obj

ProcC
  ...
  ...

ProcB.obj

ProcB
  ...
  ...

ProcA.obj

ProcA
  ...
  ...
  ...
  ....
Result: Program binary (saved on disk)
Operating System Goals

Security: OK to run file?

Memory management: Find space and create new virtual memory region for this program

File system: Retrieve program binary code from disk

Loader: Place program binary code into memory

Scheduler: Find CPU time for program to run

Context switch – Program starts running
Problem 1

Without Google search, can you identify the Linux command to link object files.
Makefile

Goal: Compile our program with one command:

```
unix> make
```

A **Makefile** is a **text file** that specifies how to compile your program

- The `make` utility reads the Makefile
- You’ll learn how this file works in Lab 3
An Intermediate Makefile

all: factorial_program
factorial_program: main.o factorial.o output.o
  gcc main.o factorial.o output.o -o factorial_program
main.o: main.c
  gcc -c main.c
factorial.o: factorial.c
  gcc -c factorial.c
output.o: output.c
  gcc -c output.c

clean:
  rm -rf *.o factorial_program
An Advanced Makefile
# The variable CC specifies which compiler will be used.
# (because different unix systems may use different compilers)
CC=gcc

# The variable CFLAGS specifies compiler options
#  -c: Only compile (don't link)
#  -Wall: Enable all warnings about lazy /
dangerous C programming
CFLAGS=-c -Wall

# The final program to build
EXECUTABLE=factorial_program
all: $(EXECUTABLE)

$(EXECUTABLE): main.o factorial.o.o output.o
       $(CC) main.o factorial.o.o output.o.o -o $(EXECUTABLE)

main.o:  main.c
       $(CC) $(CFLAGS) main.c

factorial.o: factorial.c
       $(CC) $(CFLAGS) factorial.c

output.o:  output.c
       $(CC) $(CFLAGS) output.c

clean:
       rm -rf *.o $(EXECUTABLE)
C Tutorial
Print with printf()

```c
printf("This is a string\n");
printf("The integer is %i\n", num);
printf("The floating-point values are %g and %g\n", num1, num2);
```
## Output with printf()

<table>
<thead>
<tr>
<th>Format “Type” Code</th>
<th>Corresponding Variable Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>d or i</td>
<td>int (interpret as signed 2’s comp)</td>
</tr>
<tr>
<td>u</td>
<td>int (interpret as unsigned)</td>
</tr>
<tr>
<td>x</td>
<td>int (print as hexadecimal)</td>
</tr>
<tr>
<td>f or g</td>
<td>float/double</td>
</tr>
<tr>
<td>c</td>
<td>char</td>
</tr>
<tr>
<td>s</td>
<td>string (null-terminated array of chars)</td>
</tr>
<tr>
<td>p</td>
<td>An address to which the pointer points</td>
</tr>
</tbody>
</table>

**Prefix with l or ll (i.e. “long” or “long long” for larger 64-bit data types)**

- Lots of formatting options not listed here…
  - # of digits before / after decimal point?
  - Pad with zeros?
Input with `scanf()`

Input from console

```c
scanf("%d %c", &myint, &mychar)
```

Requires the **address** of the destination variable

- Use the `&` operator to obtain address
Problem 2 – Read the man pages for printf and scanf

**Man(ual) pages exist for common programming functions too**

unix> man printf
unix> man scanf
Arrays
Arrays

Contiguous block of memory

You can have arrays for int, char, float, double, structures...

`int myarray[5]; //static declaration`

NOTE: Name of the array is the address of the first element

<table>
<thead>
<tr>
<th>address:</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>myarray[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>myarray[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>myarray[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>myarray[3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>myarray[4]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

`printf("%p", myarray); //prints what?`
2-dimensional arrays

```c
int myarray[5][5]; // static declaration
```

Memory map:

<table>
<thead>
<tr>
<th>Address: 4</th>
<th>Address: 8</th>
<th>Address: 12</th>
<th>Address: 16</th>
<th>Address: 20</th>
</tr>
</thead>
<tbody>
<tr>
<td>myarray[0][0]</td>
<td>Address: 8</td>
<td>Address: 12</td>
<td>Address: 16</td>
<td>Address: 20</td>
</tr>
<tr>
<td>Address: 24</td>
<td>Address: 28</td>
<td>Address: 32</td>
<td>Address: 36</td>
<td>Address: 40</td>
</tr>
<tr>
<td>myarray[1][1]</td>
<td>Address: 28</td>
<td>Address: 32</td>
<td>Address: 36</td>
<td>Address: 40</td>
</tr>
<tr>
<td>Address: 44</td>
<td>Address: 48</td>
<td>Address: 52</td>
<td>Address: 56</td>
<td>Address: 60</td>
</tr>
<tr>
<td>myarray[3][2]</td>
<td>Address: 68</td>
<td>Address: 72</td>
<td>Address: 76</td>
<td>Address: 80</td>
</tr>
<tr>
<td>Address: 84</td>
<td>Address: 88</td>
<td>Address: 92</td>
<td>Address: 96</td>
<td>Address: 100</td>
</tr>
</tbody>
</table>
Problem 3: Looping through an array

Consider a 3-D array, int image[256][256][3] for an RGB color image. The first subscript denotes the number of rows, the second subscript denotes the number of columns, and the third subscript denotes the number of color channels. For example, a pixel at row $i$ and column $j$ will have an R value given by $\text{image}[i][j][0]$, G value given by $\text{image}[i][j][1]$, and B value given by $\text{image}[i][j][2]$. Any pixel has a yellow color if its R and G values are 255 and B value is 0. Write a for loop to search for the location of the very first yellow pixel in image. The search should terminate once the yellow pixel is found. Search in row-wise manner.
Pointers
**Pointers** are special variables that hold/store memory addresses of other variables.

When a pointer, say `iptr`, holds the address of an integer variable, say `ivar`, then we say: “`iptr` is an integer pointer that points to `ivar`.”

```c
int ivar = 45;
int *iptr; iptr = &ivar; // `iptr` points to `ivar`
```

<table>
<thead>
<tr>
<th>ivar: 45</th>
<th>iptr: 65536</th>
</tr>
</thead>
<tbody>
<tr>
<td>address: 65536</td>
<td>address: 65520</td>
</tr>
</tbody>
</table>

‘&’ is ‘address of variable’ operator. For example, `&ivar` translates to: “address of variable `ivar`”.

“*” is ‘value at address stored in pointer’ operator. For example, `*iptr` translates to: “value at address stored in pointer `iptr`”.

We can have a ‘multiple’ pointer

Example pointer declaration:
int *iptr; // an integer pointer that will point to an integer

int **dptr; // A double pointer that will point to an integer pointer

int ***tptr; // A triple pointer pointing to a double pointer.

int ****quadptr //
Problem 4

Consider the variables below:

<table>
<thead>
<tr>
<th>Variable Name: ivar</th>
<th>Pointer variable name: iptr</th>
</tr>
</thead>
<tbody>
<tr>
<td>value: 5</td>
<td>value:</td>
</tr>
<tr>
<td>Address: 0xFFABCD</td>
<td>Address: 0xAFAFABAD</td>
</tr>
</tbody>
</table>

```c
int ivar=5;
int *iptr;
iptr = &ivar;
printf("\n %u",ivar);  prints_________
printf("\n %u",&ivar);  prints_________
printf("\n %u",&iptr);  prints_________
printf("\n %u",*iptr);  prints_________
```
## Problem 5

<table>
<thead>
<tr>
<th>Variable Name: ivar</th>
<th>Pointer variable name: iptr</th>
<th>Pointer variable name: dptr</th>
</tr>
</thead>
<tbody>
<tr>
<td>value: 5</td>
<td>value:</td>
<td>value:</td>
</tr>
<tr>
<td>Address: 0xFFABCD</td>
<td>Address: 0xAFABAD</td>
<td>Address: 0xFFACBD</td>
</tr>
</tbody>
</table>

```c
int ivar=5;
int *iptr;
int **dptr;
iptr = &ivar;
dptr=&iptr;
printf("\n %u",dptr);  prints_________
printf("\n %u",iptr);  prints_________
printf("\n %u",*dptr);  prints_________
printf("\n %u",**dptr);  prints_________
printf("\n %u",&dptr);  prints_________
printf("\n %u",*(&iptr));  prints_________
```
Next Class

Pointer basics

Pointers and Arrays
  Dynamic Allocation

Pointers and Structures
  Linked Lists

File I/O in C