CSC180: Lecture 12

Wael Aboulsaadat

wael@cs.toronto.edu

http://portal.utoronto.ca/

Acknowledgement: These slides are partially based on the slides supplied
with Prof. Savitch book: Problem Solving with C

Programming with Arrays

Returning An Array

- Recall that functions can return a value of type int, double, char, …,
- Functions cannot return arrays
- We learn later how to return a pointer to an array

Programming With Arrays

- The size needed for an array is changeable
  - Often varies from one run of a program to another
  - Is often not known when the program is written

- A common solution to the size problem
  - Declare the array size to be the largest that could be needed
  - Decide how to deal with partially filled arrays

Partially Filled Arrays

- When using arrays that are partially filled
  - Functions dealing with the array may not need to know the declared size of the array, only how many elements are stored in the array
  - A parameter, number_used, may be sufficient to ensure that referenced index values are legal
  - A function such as fill_array(…) needs to know the declared size of the array

Multidimensional Array Parameters

- Recall that the size of an array is not needed when declaring a parameter:
  void display_line(const char a[ ], int size) {…}

- The base type of a multi-dimensional array must be completely specified in the parameter declaration
  - void display_page(const char page[ ] [100],
    int size_dimension_1) {…}

C Program Organization

Program Organization: approach 1

- Simplest Approach

```
float square(float x) {
      return x * x;
}

int factorial(int x) {
      if (x==1) { return x;}
      else { return x * factorial(x-1); }
}

int main( )
{
   ……
}
```

Program Organization: approach 2

- Forward declaration

```
float square(float x);
int factorial(int x);

int main( )
{
   ……
}

float square(float x) {
      return x * x;
}
int factorial(int x) {
      if (x==1) { return x;}
      else { return x * factorial(x-1); }
}
```

Program Organization: approach 3

- Multiple files
- int main() function in a separate file
- Each set of related functions in a separate .c and .h files

Approach 3 could be used to achieve modularity

- **Module:** a unit of organization of a software system
  - *groups* together some functions, data, types, etc.
  - **Example:** various input/output functions in C's standard io library.
- Conceals irrelevant information from user of the function.
- user’s view of a module.

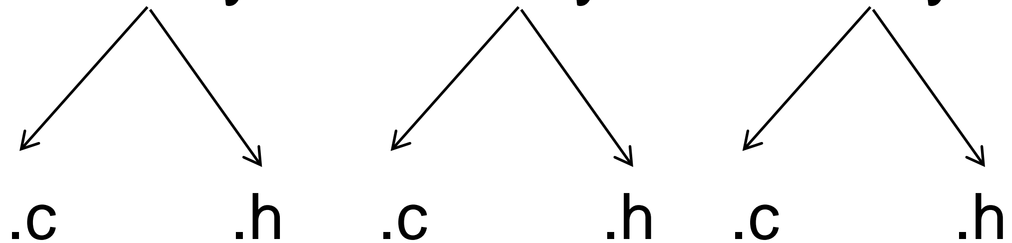
Approach 3 could be used to achieve modularity

- User’s view of a module:
  - describes only what a user needs to know to use the module
  - makes it easier to understand and use
  - describes what services the module provides, but not how it’s able to provide them

Approach 3 could be used to achieve modularity, how?

- Each pair of c & h file are viewed as an independent library

- The program = main + library 1 + library 2 + library 3



Back to "Hello World!"

Include the standard C **header file** so that you can call I/O functions like "printf".

```
#include <stdio.h>

int main() {
   printf("Hello World!\n");
   return 0;
}
```

The entry point of any program is the "main" function. It receives no arguments, and returns an integer.

Last line must return a value before exiting the function.

printf prints its argument string to the screen. A string is denoted by letters within double quotations. '\n' means newline.

Header file

- *stdio.h* is a C standard header file.
- Including it allows you to use functions like *printf*, *fopen*, *getchar* etc.
- There are other common C standard header files.
- For example, *stdlib.h* contains the *rand* function, and *math.h* contains math functions like *sin*, *cos*, *sqrt* etc.

Your Own Header files

- You can create your own header files.
- This is to separate the function declarations from the function definitions.

useful.h

```
float square(float x);
int factorial(int x);
```

useful.c

```
#include "useful.h"

float square(float x) {
	return x * x;
}

int factorial(int x) {
	if (x==1) {
		return x;
	} else {
		return x * factorial(x-1);
	}
}
```

Include the header file

- Now, in order for you to use the functions square and factorial in other files, you'll need to include the header file.

main.c

```
#include <stdio.h>
#include “useful.h”

int main() {
	float f;
	int i;

	f = square(3.5);
	i = factorial(10);

	printf(“The numbers are %f %d\n”, f, i);
	return 0;
}
```

Include the header file

- Now, in order for you to use the functions square and factorial in other files, you'll need to include the header file.

main.c

```
#include <stdio.h>
#include “useful.h”

int main() {
	float f;
	int i;

	f = square(3.5);
	i = factorial(10);

	printf(“The numbers are %f %d\n”, f, i);
	return 0;
}
```

Compiling in Unix

- First, make the object files useful.o and main.o
- Then, link the object files together to create the executable.

```
wael@cslin (2)% gcc –c main.c
wael@cslin (3)% gcc –c useful.c
wael@cslin (4)% gcc –o myprog main.o useful.o
wael@cslin (5)% myprog
```

The –c option reads a source file and creates an object file. An object file contains compiled binary code, but is not executable. This line creates the object file main.o

This line creates the object file useful.o

The –o option creates an executable with the immediately following filename, in this case “myprog.” Supply all the input files. In this case, the input files are the object files main.o and useful.o

Run the executable

gcc

- Behavior controlled by command-line switches:

-o *file*	output file for object or executable
-Wall	all warnings – use always!
-c	compile single module (non-main)
-g	insert debugging code (gdb)
-l	library
-E	preprocessor output only

On unix/linux type man gcc to view gcc manual

Compiling in Windows

- Using Microsoft Visual Studio, include all the .c and .h files in the project.
- Visual Studio will automatically consider all the dependencies and generate all the necessary object files.
- Visual Studio will also automatically link the object files to create an executable.

Preprocessor Directives

- There is a *preprocessor* or *pre-compiler* that runs before the compiler itself…
- Any line in your source code that begins with ‘#’ is a preprocessor directive. It gives instructions to the preprocessor.

#include

- The #include instruction tells the preprocessor to append the file named before compiling.
- Filenames that are enclosed in < > are standard C libraries. The preprocessor looks for the files in the standard C library directory (wherever it is installed).
- Filenames that are enclosed in “” are user-written header files and are loaded from the current directory.

useful.h

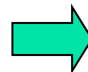
```
float square(float x);
int factorial(int x);
```

useful.c

```
#include "useful.h"

float square(float x) {
      return x * x;
}

int factorial(int x) {
      if (x==1) { return x;}
      else { return x * factorial(x-1); }
}
```



What the compiler actually sees (after preprocessor)

```
float square(float x);
int factorial(int x);

float square(float x) {
      return x * x;
}

int factorial(int x) {
      if (x==1) { return x;}
      else { return x * factorial(x-1); }
}
```

#define

- You can define some symbols to be some other strings.
- For example: #define PI 3.14159
- Thus, when the preprocessor encounters the string PI in the code, it replaces it with 3.14159.
- So, the compiler doesn't see PI at all. It only sees 3.14159.

Source file:

```
#define PI 3.14159

float caclarea(float r) {
    return PI * r * r;
}
```

What the compiler sees:

```
float caclarea(float r) {
    return 3.14159 * r * r;
}
```

#define

- Another common example of the use of #define: “#define ARRAY_LEN 512”
- When you decide to go to a larger array size, simply change the line to: “#define ARRAY_LEN 1024”

Source file:

```
#define ARRAY_LEN 512

float farray[ARRAY_LEN];

void incArray( ) {
    int i;
    for (i=0;i<ARRAY_LEN;i++) {
        farray[i] = farray[i] + 1.0;
    }
}
```

What the compiler sees:

```
float farray[512];

void incArray( ) {
    int i;
    for (i=0;i<512;i++) {
        farray[i] = farray[i] + 1.0;
    }
}
```

Program Comments

- Extensive comments!
  - in .h what inputs each function take and what it does.
  - in .c why is this function implemented this way…

Comments

- *`/* any text until */`*

- Convention for longer comments:

```
/*
 * AverageGrade()
 * Given an array of grades, compute the average.
 */
```

- Avoid **** boxes – hard to edit, usually look ragged.

Hungarian Notation: naming convention

- name of a variable indicates its type or intended use

- E.g.

int	nQuantity,
float	fPrice,
long	lLength,
double	dTemperature,
double[]	darrTemperatures;
unsigned int	unItemsCount,
char	cTaxType,
char[]	carrName;