8. Complex Programs
ENEE 140

Prof. Tudor Dumitraș
Associate Professor, ECE
University of Maryland, College Park

http://ter.ps/enee140

1

Today's Lecture

- Where we've been
  - Scalar data types (`int, long, float, double, char`)
  - Basic control flow (`while` and `if`)
  - Functions
  - Random number generation
  - Arrays and strings
- Where we're going today
  - Structuring complex programs
  - Enumerations
  - Composite data types: `struct`
  - Command line arguments
  - Truth values
- Where we're going next
  - Then: Control flow

2

2

Review of Arrays

- Arrays are vector data types
  - They can hold multiple values of the same type
- The size of the array must be declared and not exceeded

```
int  a[10];
a[0] = 0;
a[9] = 0;
a[10] = 0;
```

logical error: index out of bounds

- Arrays can be initialized, but not assigned

```
int a[3] = {1, 2, 3}, b[3] = {0, 0, 0};
b = a;
```

syntax error: cannot assign arrays

3

3

Command Line Arguments

- We've seen:

```
cp file1 file2
cal 2014 3
```

UNIX command-line utilities

Command line arguments

- To retrieve the command line arguments in your program

```
int main(int argc, char *argv[])
```

`argc`	Number of arguments provided, including the executable
`argv[0]`	Name of the executable
`argv[i]`	String containing the i^{th} argument

- Example:

`cal 2014 3`   `argc = 3` and `argv = {"cal", "2014", "3"}`

5

5

Structures

- You can create composite types

```
struct point {
  int x;
  int y;
};
struct point a, b;       variables of composite type
a.x = 0;                 accessing members
a.y = 0;
b = a;                   assignment
```

- Manipulating struct variables
  - Can assign them
  - Can access their members
  - Can provide them as parameters to a function (they behave like scalar variables)
  - Can be the return type of a function
  - Cannot compare them (e.g. `b > a`)

6

6

Using Structures in Your Programs

- Structures and functions

```
struct point addpoint (struct point p, int x, int y)
{                        Can pass a structure as a parameter
  struct point temp;

  temp.x = p.x + x;      No conflict between temp.x and x
  temp.y = p.y + y;
  return temp;           Functions can return structures
}
```

- Arrays of structures

```
struct point point_cloud[1000];
point_cloud[0].x = 10;
point_cloud[0].y = 20;
```

- **Good programming practice: when you need two parallel arrays, consider using an array of structures instead**

7

7

typedef

- Create a new type name, for convenient access

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

`typedef struct point Point;`	new composite type
`typedef int Length;`	new scalar type
`Point p = {0, 0};`	variable of type `Point`
`Length l = 1;`	variable of type `Length`

8

8

Truth Values

- The conditions in `while (…)` or `if (…)` can be assigned to variables
  - The type of these variables is integer: **0** is **false** and **1** is **true**
  - In a condition, any integer other than 0 will be accepted as true

`int a = (1==0);`	a is 0
`int b = (a>=0);`	b is 1
`int c = 140;`	
`if (c)`	
`printf("c is true!");`	the printf statement is executed

9

9

enum

- Enumeration constant: list of constant enumeration values

```
enum answer {NO, YES};
```
variables of type answer can take 2 values: NO or YES

```
enum months {JAN=1, FEB, MAR, APR,
     MAY, JUN, JUL, AUG,
     SEP, OCT, NOV, DEC};
```
FEB is 2, MAR is 3, etc.

```
int current_month = FEB;
```

10

10

Header Files

- We've seen

```
#include <stdio.h>
#include <math.h>
```
Header files from the standard library

- A header file includes **function declarations** (prototypes) and **constant definitions** that are shared among multiple C files

```
#include "myheader.h"
```
Include your header file in the C source files

- Must prevent multiple inclusions
  - Wrap everything inside the header in an include guard

```
#ifndef MYHEADER_H_
#define MYHEADER_H_
…

#endif /* MYHEADER_H_*/
```

11

11

Splitting a Program Into Multiple Files

- Another form of modularity
  - Group related functions in one .c source file

- Create one .h header file and multiple .c source files
  - Put all the shared declarations in the header file
  - Put all the function implementations in the source files
  - There must be only one main() function

- Compiling
  - In CLion: add all the `.c` and `.h` files to the same project
  - On the command line: `gcc file1.c file2.c file3.c`
    - Provide all the source files, but not the header file

12

12

Variables With the Same Name

- We've seen

```
void fun()
{
  int a;          variable a declared inside function fun()
  …
}
int main()
{
  int a;          variable a declared inside function main()
  float a;        error: cannot declare another variable named a in main()
  …
}
```

- a from `fun()` and a from `main()` are different variables
  - The same is true for function parameters with the same name

13

13

Variable Scope

- Variable scope (where is the variable visible)
  - Inside the block where it is declared
    - A block is enclosed in `{ }`
  - Can also declare variables at the start of `if`, `while`, `for`, etc. blocks

```
while (condition) {
 int a = 1;
 …
}
```

variable a visible only inside while loop

14

14

Global Variables

- Variables declared outside any function

```
int a;
int main()
{
 …
}
```

global variable

- Global variable scope
  - Globally accessible in all the files compiled and linked together

15

15

Static Variables Declared Outside Any Function

- Declared using keyword `static`

```
static int a;      variable local to current .c file
int main()
{
  …
}
```

- Variable scope
  – Visible only inside the .c file where they are declared
  – Can be used to hold the internal state of a library

16

16

Static Variable Declared Inside A Function

- Initialized only the first time when the block is executed

```
void fun()
{
  static int count_invocations = 0;     static variable
  count_invocations++;
  …
}
```

- Static variables preserve their value across function invocations
  – Same as global variables

- Variable scope
  – Visible only inside the function where they are declared

17

17

Good Programming Practice

- Limit the scope of your variables
  - Declare variables inside functions
  - Use variables local to a .c file to store the internal state of a module
- Avoid global variables
  - They break encapsulation
- Do not include variable declarations in .h files
  - Include only function prototypes and constants defined with `#define`
- Avoid static variables inside a function
  - They cause undefined behavior when the program execution is not sequential

18

18

Review of Lecture

- What did we learn?

19

19

Next Steps

- Next week
  - Control flow
- Assignments for this week
  - Homework: `lab08.pdf` (on http://ter.ps/enee140), due on Friday at 11:59 pm
  - Read **K&R Chapters 2.11, 2.12, 3.4, 3.5, 3.6, 3.7, 3.8, 5.10, 6.2, 6.3, 6.7**
  - Weekly challenge: **check_password_rules.c**
  - **Quiz 6** (due on Sunday at 11:59 pm)

20

20