4. Functions
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
  - Variables and constants
  - Variable assignment and operators
  - `int`s, `float`s and `char`s
  - Iterating (`while, for`) and branching (`if`)
- Where we're going today
  - Functions
- Where we're going next
  - Integer and floating point arithmetic

2

2

Review: Character Input/Output

- We've seen: reading input character-by-character

```
int c;
c = getchar();
while (c != EOF) {

    …
    c = getchar();

}
```

Why int?

Read next character

Do something with c
Read next character

- Can compare chars and add/subtract offsets

```
c = c +          ;
```
Convert uppercase c to lowercase

```
if (c >=     && c <=     )
```
Check if c is uppercase

```
c = 'D' - 'A';
```
c is

3

3

Formatted Input and Output

- We've seen:

```
int a = 0;
printf("The value of a is %d\n", a);
```

- You can output data with `printf` and read data from the input with `scanf`
- `printf` format specifiers
  - %d: int
  - %f: float, double
  - %c: character
  - %ld: long
  - %E: float, double in scientific notation, e.g. 1.5E3
  - %%: the '%' character
  - See Table 7.1 in K&R for a complete specification
    - Or type `man 3 printf` on the command line
- Read data from the input

```
int a, b;
scanf("%d %d", &a, &b);
```

4

4

Prompting the User for Input

- Print a message indicating the input expected
- Then read the input

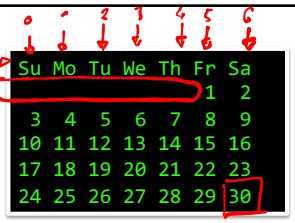
```
int sec;
float gpa;

printf("Enter your section number: ");
scanf("%d", &sec);

printf("Enter your GPA: ");
scanf("%f", &gpa);
```

5

5

A Problem Solved by Programming

- Given a month and a year in the future, print the calendar on a text terminal
  - Example for September 2023




```
print_cal (month, year)
    print "Su Mo Tue We Th Fr Sa"
    new line
    start ← first_day (month, year)
    days ← days_in_month (month, year)
    print_weeks (start, days)
```

6

6

Modularity

- Functions allow you to break down your program's functionality into smaller pieces
- Programs that are made up of many small functions are called **modular**
  - In such programs it's easy to modify one function, without affecting how the rest of the program works
  - Modular code is also easier to read
- Modular programs are the result of **top-down problem solving**
  - Break down the problem you need to solve into smaller sub-problems
  - For each sub-problem, write the prototype of a function that would solve it
  - Write your program by invoking these functions, assuming that they are implemented
  - Then figure out how to implement each function
- **Good programming practice: Writing modular code!**

7

7

Functions: What We've Seen So Far

- Functions allow you to encapsulate computation
  - You don't care **how** a job is done; you know **what** is done
- Examples of functions we've seen so far

```
printf("The value of a is %d\n", a);
```
print an int variable

```
c = getchar();
```
read a character

- You can use these functions in your programs without knowing how they are implemented
- You can also define your own functions
  - Example:

```
int main() {
    …
    return 0;
}
```

8

8

Defining Your Own Functions

- Function **declaration** (prototype)

```
int square(int param);
```

- Function **definition** (implementation)

```
int                                  return type
square(int param)                    function name and parameter list
{
 int result;                         variable declarations
 result = param * param;             statements
 return result;                      return specification
}
```

- Function **invocation** (calling the function in your program)

```
int a = 1+square(2)+square(3);       use the function in an expression
```

what is the value of a? [ ]

- You must declare or define a function before you invoke it

9

9

Function Parameters and Local Variables

- Function **parameters** (arguments)
  - Parameters must have types (e.g. `int`, `float`) and are specified in the function declaration and definition:

```
int pow(int x, int y);        the function takes 2 int parameters
```

  - When you call a function, you must pass as many parameters as in the prototype

```
z = pow(2, 3);                the types must match as well
```

  - Modifying the arguments inside the function **does not** affect the original variables ("call by value" in textbook)
  - The function operates on a copy of the variable

```
int a = 2;
my_function(a);               a is still 2, regardless of what happens in the function
```

- **Variables local to the function**
  - You can declare variables inside the function, like you do in `main()`
- Parameters and local variables **cannot be accessed outside the function**

10

Return Values

- The type of the return value is specified in the prototype, before the name of the function

```
int pow(int x, int y);
```
the function returns an int

- It is also possible to write a function that does not return anything

```
void
err_msg(int code)
{
  printf("Encountered an error with code %d\n", code);
}
```
return type is void

function with int parameter

return statement is not needed

11

Putting It All Together

Quiz 3, Question 3

- Consider the following function definition

```
int
fun(int a, int b, int c)
{
  printf("%d\n", a);
  printf("%d\n", b);
  return c;
}
```

- What is the last line printed by the following code snippet?

```
int ret = fun(1,2,3);
printf("%d\n", ret);
```

12

Modularity: Examples

- Example of top-down problem solving
  - You are asked to write a program that prints a Celsius-Fahrenheit conversion table
  - Imagine that you have a function, which takes a float argument representing the temperature in Fahrenheit degrees, and returns a float with the corresponding Celsius value
  - Write the loop that prints the conversion table
  - Then look up the conversion formula and implement the function
- Helper functions
  - In your assignments, you will often be asked to implement functions that provide a certain functionality
  - It is often a good idea to write additional **helper** functions that you use in your program
  - For example, such helper functions may provide functionality that is useful for several tasks

13

13

Mathematical Functions Available in C

- These functions typically accept and return variables of type double

`#include <math.h>`	must include this header to use the math functions (more on this later)
`sin(x);`	sine of x (in radians)
`cos(x);`	cosine of x (in radians)
`exp(x);`	e^x
`log(x);`	natural logarithm of x
`log10(x);`	base 10 logarithm of x
`sqrt(x);`	square root of x

…

14

14

Aside: Manual Pages

- You can get help on most functions from the C standard library using the man command on the GRACE machines

```
man printf      manual page of printf() function
man scanf       manual page of scanf() function
```

15

15

Coding Style

- Programs are meant to be read by humans
  - Code reviews are a common practice in the industry
- Good coding style makes programs more readable
  - Examples of what **not to do**: http://www.ioccc.org/
- There is no "right" coding style
  - Choose a style and be consistent

16

16

Coding Style: Examples

- Explain what the program does in a comment at the top
- Explain what each function does in comments before the function definition
- Use concise, meaningful names for variables and constants
  - If you have many variables, also add short comments describing the purpose of some of the variables
- Follow normal English rules when possible for better readability of your code
  - Write complete sentences in your comments
  - Leave a space after each comma and semicolon (e.g. in printf(), scanf(), for)
  - Leave a space on each side of a binary operator (e.g. =, ==, +)
- Indent code consistently
  - Clion tries to do this automatically
- If you have long, nested {…} blocks, add a comment after the enclosing bracket
  - Indicate which block you are closing (the while block, the if block, etc.)

17

17

Coding Style: More Examples

- Place braces {} in a consistent manner:

```
for ( i = 0; i < 100; i++) {
    statements;
}
```
OR
```
for ( i = 0; i < 100; i++)
{
    statements;
 }
```

- When you prompt the user for input, first print out a message describing what is expected
- Check for errors and corner-cases throughout the program (more about this later)
- Use simple statements as much as possible
  - Avoid statements like `sum = a++ + --b*2`

18

18

Review of Lecture

- What did we learn?

19

Next Steps

- Next lecture
  - Integer and floating point arithmetic
- Assignments for this week
  - Read **K&R Chapters 2.5, 2.7, 2.8, 2.10, B2, B11**
    - Note: some of these chapters refer to strings (e.g. `char s[]`), which we'll cover later
    - For now, think of `s[i]` as a character variable
    - Read man pages for `rand()` and `srand()`; try to understand the implementations on page 46
  - Weekly challenge: **read_divide_ints.c**
  - Homework: **enee140_lab04.pdf**, due on Friday at 11:59 pm
  - No quiz next week

20