UNIVERSITY OF MASSACHUSETTS
DARTMOUTH

ECE160: Foundations of Computer Engineering I

Lecture #19 – Exam #2 Review

Instructor: Dr. Liudong Xing
SENG-213C, lxing@umassd.edu
ECE Dept.

Administrative Issues

- Midsemester indicator available in COIN and M: drive *grade.xlsx;* refer to my message sent to your umassd email for explanation and recommendation.

- Lab#8
  - Due **5pm, Wednesday, March 22 (Today)**
- Homework#4
  - Due **9am, Wednesday, March 22 (Today)**

- Exam#2 review session today

Exam #2

- Time: **9:00am ~ 10:30am, Friday, March 24**
- Please arrive at the class on time; no make up time will be given for late arrivals.
- Form:
  - Open book, open notes
  - Calculators are NOT allowed
  - Visual Studio is NOT allowed
- Preparation:
  - Lecture notes #12 - #18 prepared by Dr. Xing (available on class website)
  - Homework #3 - #4
  - Lab #5 - #8

Exam#1: Lectures #2 - #10

- Number systems (L#2)
- Introduction to C programming (L#3)
- Data types and variables (L#4)
- Constants (L#5)
- Formatted input/output (L#6 & 7)
- Expressions (L#8 & 9)
- Two-way selection: if…else (L#10)

Exam#2: Lectures #12 - #18

- Multi-way selection: switch and if-else-if (L#12)
- Loops (L#13)
- Functions (L#14 ~ 17)
- Files I (L#18)

switch statements (L#12, Rules)

```
switch (expression)
{
  case constant-1:
                    statements
                    break;
  case constant-2:
                    statements
                    break;
   case constant-3:
                    statements
                    break;
       ……
    default:
                    statements
                    break;
}
```

- The control expression that *switch* tests must be an integral type, i.e., it can not be a float or a double for example.
- The expression followed by each case label must be a constant expression.
- Two *case* labels can not have the same value.
- However, two cases can have the same statements.
- The *switch* can include at most one *default* label. And it can be coded anywhere, but is traditionally coded last.

Example (1)

- Write a program using *switch* that can convert a numeric score to a letter grade
  - 90 or more → A
  - 80 - 90 → B
  - 70 - 80 → C
  - 60 - 70 → D
  - Below 60 → F

```
float score;
int temp;
char grade;
temp = score/10;
switch (temp)
{
    case 10:    grade = 'A';
                break;
    case 9:     grade = 'A';
                break;
    case 8:     grade = 'B';
                break;
    case 7:     grade = 'C';
                break;
    case 6:     grade = 'D';
                break;
    default:    grade = 'F';
}
```

```
if (expression-1)
    {
      statement-block-1
    }
 else if (expression-2)
    {
      statement-block-2
    }
    ……

 else if (expression-n)
    {
      statement-block-n
    }
else
    {
      statement-block-n+1
    }
```

if-else-if control structure

Example (2)

- Write a program using *if-else-if* that can convert a numeric score to a letter grade
  - 90 or more → A
  - 80 - 90 → B
  - 70 - 80 → C
  - 60 - 70 → D
  - Below 60 → F

```
float score;
char grade;
 if(score >= 90)
            grade = 'A';
 else if(score >= 80)
            grade = 'B';
 else if(score  >= 70)
           grade = 'C';
 else if(score >= 60)
          grade = 'D';
 else
       grade = 'F';
```

Loops (L#13)

- Three *C* loop statements
  - *while* loops
  - *do…while* loops
  - *for* loops

while vs. *do…while*

```
while (expression)
    {
        statement-1
        statement-2
          ……
        statement-n
    }
```

- Pre-test: loop-continuation condition is tested before the loop.
- No semicolon is needed at the end of the while statement!

```
do
{
    statement-1
    statement-2
      ……
    statement-n
} while (expression);
```

- Post-test: loop-continuation condition is tested after the loop.
- Semicolon is needed at the end of the *do…while* statement!!

Braces are not required if the loop body consists of only one statement

The for Loop

- General expression:

```
for(statement1;statement2;statement3)
{
    loop_body
}
```

  - statement1: contains initial value of control variable
  - statement2: a test expression containing final value of control variable
  - statement3: increments/decrements the control variable
  - Braces are not required if the loop body consists of only one statement
  - The 3 expressions in the *for* structure are optional. The two semicolons are required.
  - Pre-test: loop-continuation condition (statement2) is tested before the loop.

Equivalence

```
x= 2;
while (x < 13)
{
    printf(“%d\n”,x);
    x++;
}
```

```
for(x = 2; x < 13;x++)
{
        printf("%d\n",x);
}
```

```
x =2;
do
{
        printf(“%d\n”,x);
        x++;
} while(x < 13);
```

break/continue

- The *break* and *continue* statements are used in loops to change the flow of control.
- *break* is used to escape from a loop (causes a loop to terminate).
- *continue* is used to skip the remaining statements in the body of a structure and skip to the next iteration.

break vs. continue

```
#include "stdio.h“
void main(void)
{
    int a;
    for(a =1; a <= 7; a++)
    {
        if(a == 4)
            break;
        printf("%d\n", a);
    }
    printf("I got out of the loop at
    a==%d\n",a);
}
```

```
1
2
3
I got out of the loop at a==4
```

```
#include "stdio.h"
void main(void)
{
   int a;
   for(a =1; a <= 7; a++)
   {
        if (a == 4)
            continue;
        printf("%d\n",a);
   }
}
```

```
1
2
3
5
6
7
```

Exam#2: Lectures #12 - #18

- ✓ Multi-way selection: switch and if-else-if (L#12)
- ✓ Loops (L#13)
- **Functions (L#14 ~ 17)**
  - **User defined functions**
  - **Standard library functions**
  - **Recursions**
- Files I (L#18)

Functions (L#14, 15)

- A function is an independent module that somebody calls it in order to perform a specific task
- One reason for defining a function is to avoid writing the same group of C statements over and over again.
- Every C program contains one and only one main()
- Functions must be declared before being used in a program
- Information can be passed between a function and the function that calls it

Function Declarations

- Through the **function prototype** statements

  return_value_type function_name(parameter_list);

  – same as the function header, but with a semicolon at the end

  – Parameter names are not necessary

  ```
  float average_2(int num1, int num2);
  float average_2(int, int);
  ```

  – Make sure that the function prototype matches exactly the function's definition (return type, function name, number, types, and order of arguments)

Calling Functions

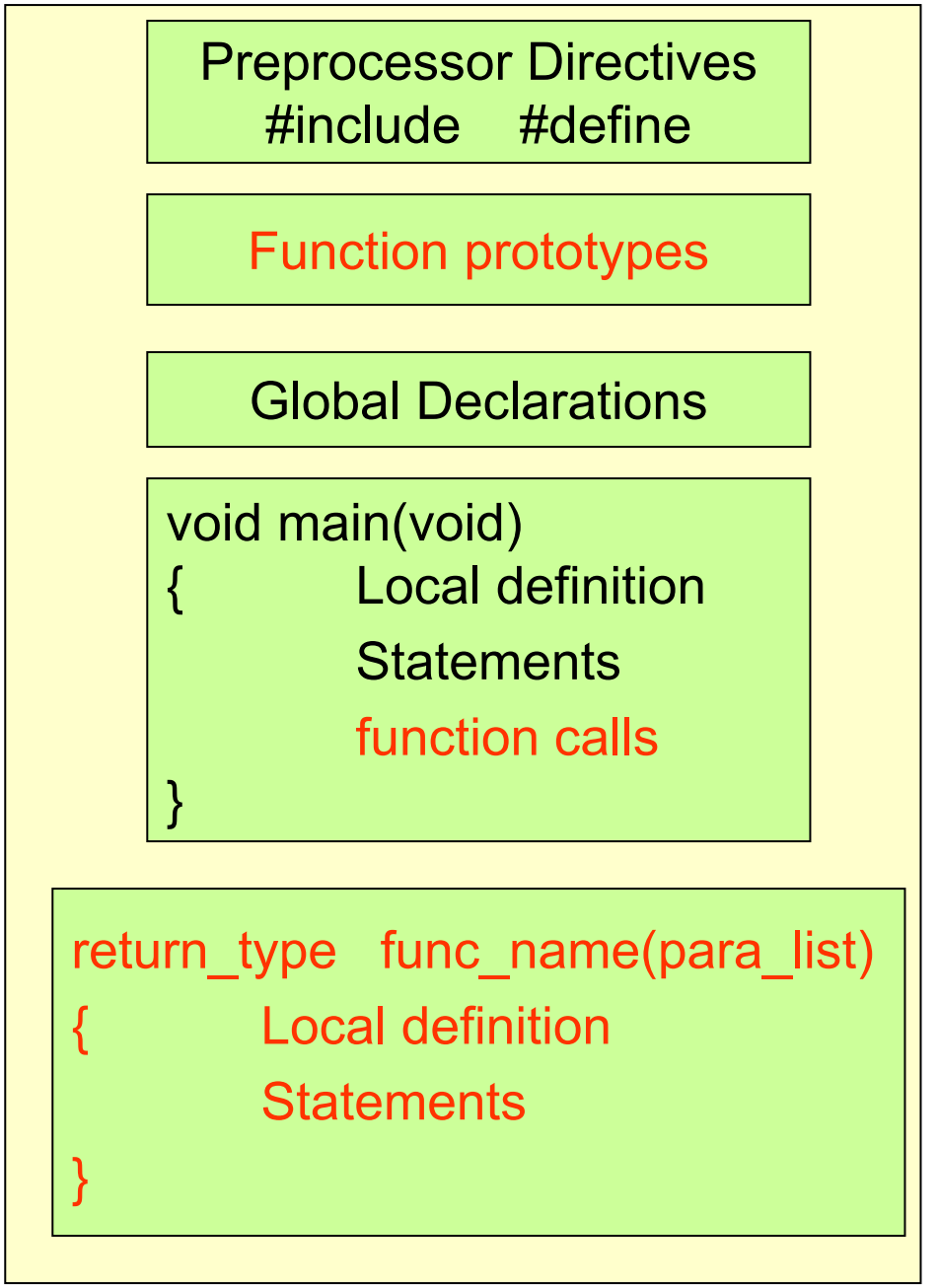
- Format

  function_name(parameter_expression_list)

  – Expression in the list can be and commonly is a single variable or constant
  – Separated by commas
  – Total number of expressions must equal to number of arguments in the function prototype

- When the function has no arguments, remember to put the parentheses when you call them
- A function call *transfers program control* and *passes the values* from the caller to the function

Basic structure of the C programs



Calculator Example

```
#include "stdio.h“
/*function delcaration*/
void add(int f, int g);
void subtract(int f, int g);
void multiply(int f, int g);
void main(void)
{
      char c;
      int a, b;
      printf("Please enter the operation:\n");
      scanf("%c",&c);
      printf("Please enter two integers\n");
      scanf("%d%d",&a,&b);
      switch(c)
      {
            case '+': printf("This is an addition\n");
                      add(a,b);
                      break;
            case '-': printf("This is a subtraction\n");
                      subtract(a,b);
                      break;
            case '*': printf("This is a multiplication\n");
                      multiply(a,b);
                      break;
            default:  printf(" Operation not defined\n");
       }
}
```

```
/*add() function definition*/
void add(int f, int g)
{
      int sum;
      sum = f+g;
      printf("The result is %d\n", sum);
}
/*subtract() function definition*/
void subtract(int f, int g)
{
      int difference;
      difference = f-g;
      printf("The result is %d\n", difference);
}
/*multiply() function definition*/
void multiply(int f, int g)
{
      int product;
      product = f*g;
      printf("The result is %d\n", product);
}
```

Note (Function Definition)!

- DO NOT use a semicolon at the end of the function header definition.
- The function body must be enclosed within a pair of braces!
- Ensure that what you are returning from the function matches the return type of the function.
- The type of each function argument must be individually defined in the parameter list.
- DO NOT define a function inside another function.

Parameter Passing (L#16)

- Pass by value
  - A copy of the data (argument's value) is passed to the called function.
  - The function can not modify the original variable's value in the caller.

- Pass by reference.
  - The called function can modify the original variable's value in the caller.
  - Any reference to a parameter is the same as a reference to the variable in the calling function
  - It uses the address operator (&) and indirection operator (*).

Example (Pass by Value)

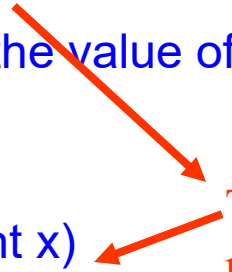
What is the output of the program?

```
#include "stdio.h"
void test(int x);

void main(void)
{
    int a;
    a =2;
    test(a);
    printf("the value of a after call is %d\n", a);
}

void test(int x)
{
    x = x + 5;
}
```

the value of a after call is 2



The value of a is copied into the memory cell reserved for x in the region of memory for test function

Example (Pass by Reference)

```
#include "stdio.h"
void test(int *x);

void main(void)
{
    int a;
    a =2;
    test(&a);
    printf(" the value of a after call is %d\n", a);
}

void test(int *x)
{
    *x = *x + 5;
}
```

In a function prototype or header, * means the variable following * is to hold an address

& means the address of , a copy of the address of variable a is put into memory cell reserved for x in the memory region reserved for the variables of test function

the value of a after call is 7

Standard Library Functions (L#16, 17)

- C has a rich collection of functions whose definitions have been written and are ready to be used in your programs
  - Mathematical functions
  - Random number generation functions: srand(), rand()
  - Character functions
    - Classifying functions: int is…(int testchar);
    - Converting functions: int to….(int oldchar);

Using Standard Library Functions

- To use them, include their prototype declarations in the program

- Their prototypes are grouped into header files
  - Input/output functions (printf, scanf) → stdio.h
  - Mathematical functions → math.h, stdlib.h
  - General utility functions → stdlib.h
  - Etc…

- Use **include** statement to include the header files
  - Example: **#include <stdio.h>**

Mathematical Functions

- double ceil (double number);
  - returns the smallest integral value greater than or equal to a number.
- double floor (double number);
  - returns the largest integral value that is equal or less than a number.
- double fabs(double number);
  - returns the absolute value of a double
- double sqrt(double number);
  - returns the square root of a number.
- double pow (double x, double y);
  - return the value of x raised to the power y, I.e., x^y

rand() and srand()

```
#include "stdlib.h"
#include "stdio.h”
#include "time.h”

void main(void)
{
    int rand1;
    int rand2;

    srand(time(NULL));
    rand1 = rand();
    rand2 = rand();

    printf("The numbers are %d %d\n", rand1, rand2);
}
```

Scaling Random Numbers

- To scale numbers in the range min ~ max, we scale like this:

rand() %((max + 1)-min) + min

```
#include "stdlib.h"
#include "stdio.h"
#include "time.h"

void main(void)
{
    int rand1;
    int rand2;

    srand(time(NULL));
    rand1 = rand()%11;
    rand2 = rand()%11+20;

    printf("The numbers are %d %d\n", rand1, rand2);
}
```

Recursion (L#17)

- Two approaches to writing repetitive algorithms
  - Using loops (for, while, do…while; iterative way)
  - Using recursion
- Recursion is a repetitive process where a function calls itself.
  - Recursive solution involves a two-way journey
    - First, we decompose the problem from top to bottom
    - Then we solve it from bottom to top
  - Base case:
    - The statement that "solves" the problem
    - Every recursive function must have a base case
    - Once the base case has been reached, the solution begins

Review: factorial(n)

```
long factorial(int n)
{
    int i;
    long fact=1;
    for(i=1; i<= n; i++)
     {
          fact = fact * i;
     }
    return fact;
}
```

```
long factorial(int n)
{
    if (n == 0)
          return 1;
    else
          return(n*factorial(n-1));
}
```

$$factorial(n) = \begin{cases} 1 & \text{if } n = 0 \\ 1 * 2 * \ldots * (n-1) * n & \text{if } n > 0 \end{cases}$$

$$factorial(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * factorial(n-1) & \text{if } n > 0 \end{cases}$$

Iterative Solution

Recursive Solution

Examples of Recursive Functions

- factorial(n) (Lecture#17)
- fibonacci(n) (Lecture#17, HW#4--Problem#4)
- gcd(x,y) (Lab#8)

Exam#2: Lectures #12 - #18

- ✓ Multi-way selection: switch and if-else-if (L#12)
- ✓ Loops (L#13)
- ✓ Functions (L#14 ~ 17)
- **Files I (L#18)**

Files

- A collection of information/related data treated as a unit
- Saved in secondary (auxiliary) memory like disks.
- Using files in C:
  - How to declare a file_pointer (**FILE**)
  - How to open a file (**fopen()**)
  - How to read from a file (**fscanf()**)
  - How to write to a file (**fprintf()**)
  - How to close a file (**fclose()**)

About FILE

- FILE is a C derived data type defined in the C standard header file stdio.h
  - Include file: #include <stdio.h>

- To manipulate a disk file, use the C data type FILE to declare a file_pointer, then use this file_pointer to handle your file

```
FILE *file_pointer;
```

How to Open a file?

- Format:

  file_pointer = **fopen**("file_name", "mode");

- Mode:
  - r: Open file for reading.
  - w: Open text file for writing.
  - a: Open text file for appending.

- fopen() creates a link between a disk file and a file_pointer. Once the link is created we can work with the file_pointer in our program to give us access to the file to which it is linked.

How to Read data from a File?

Using fscanf() :

```
fscanf(file_pointer, "format_string", address_list)
```

- reads the contents of the file indicated by the file_pointer according to the conversion code in format_string.
- contents read are put into the address given by the address_list.

```
FILE *example_ptr;
example_ptr = fopen("Lecture19.txt", "r");
fscanf(example_ptr, "%d%lf", &a, &b)
```

How to Write output to a File

- The output displayed on the screen is lost when the screen scrolls or clears

printf("format_string", data_list)

- To keep a permanent record of the output, write the output to a file

fprintf(file_pointer, "format_string", data_list)

writes the values of data in data_list using the given format_string to a file that is linked to the program using the file_pointer

How to Close a file?

- It's good practice to close files (to free system resources) after they have been used!
- Format/prototype:

```
int fclose(FILE *file_pointer);
```

- Example:

```
fclose(example_ptr);
```

Note: use file_pointer, not the file name to close a file!

A Complete Example (Review)

```
#include "stdafx.h"
int main(void)
{
      FILE *fp;
      int num1=100;
      int num2=200;
      int num3=300;
      int a=0, b=0, c=0;

      fp = fopen(“Xing_file1.txt","w");
      if(!fp)
      {
         printf("I was not able to open file\n");
         return(1);
      }
      fprintf(fp,"%d\n%d\n%d\n", num1, num2, num3);
      if(fclose(fp) == EOF)
     {
       printf("I was not able to close file\n");
       return(2);
      }
```

```
fp = fopen(“Xing_file1.txt",“r");
if(!fp)
      {
         printf("I was not able to open file\n");
         return(1);
      }

fscanf(fp,"%d%d%d",&a,&b,&c);

printf(“a is %d\n b is %d\n c is %d\n",a,b,c);

if(fclose(fp) == EOF)
{
       printf("I was not able to close file\n");
     return(2);
}

}
```

Exam #2

- Time: **9:00am ~ 10:30am, Friday, March 24**
- Please arrive at the class on time; no make up time will be given for late arrivals.
- Form:
  - Open book, open notes
  - Calculators are NOT allowed
  - Visual Studio is NOT allowed
- Preparation:
  - Lecture notes #12 - #18 prepared by Dr. Xing (available on class website)
  - Homework #3 - #4
  - Lab #5 - #8

Good Luck!