UEE1302
Introduction to Computers and Programming

C_Lecture 05:

Functions (II)

C: How to Program 7th ed.

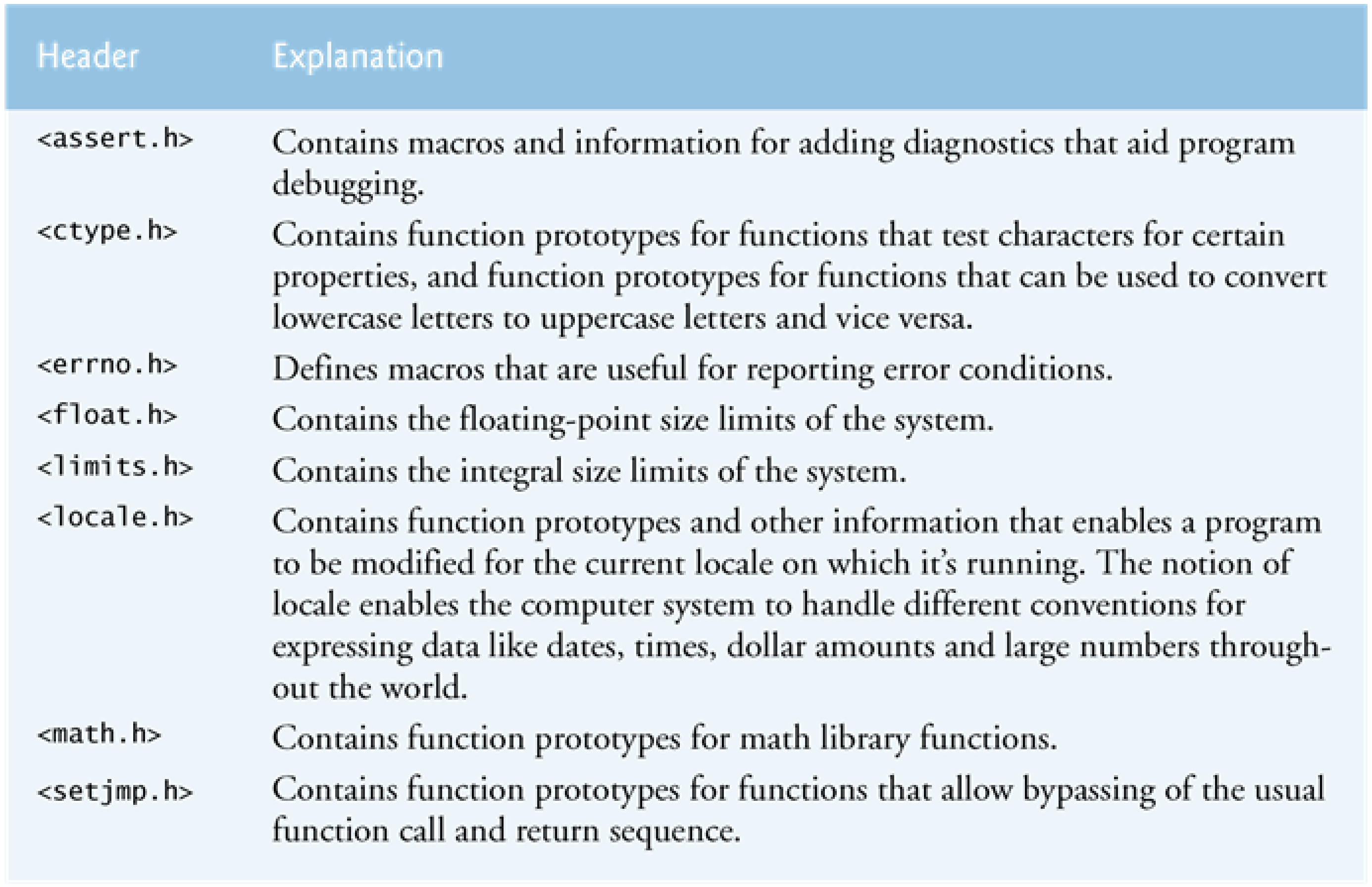
Agenda

- Headers
- Passing Arguments
- Random Number Generation
- Variable Scope
- Variable Storage Class

C Standard Library Header Files

- The C Standard Library is divided into many portions, each with its own header file
- The header files contain the function prototypes for the related functions that form each portion of the library
- The header files also contain definitions of various functions, as well as constants needed by those functions
- A header file “instructs” the compiler on how to interface with library and user-written components

Some C Standard Library Header Files



Header	Explanation
`<assert.h>`	Contains macros and information for adding diagnostics that aid program debugging.
`<ctype.h>`	Contains function prototypes for functions that test characters for certain properties, and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa.
`<errno.h>`	Defines macros that are useful for reporting error conditions.
`<float.h>`	Contains the floating-point size limits of the system.
`<limits.h>`	Contains the integral size limits of the system.
`<locale.h>`	Contains function prototypes and other information that enables a program to be modified for the current locale on which it's running. The notion of locale enables the computer system to handle different conventions for expressing data like dates, times, dollar amounts and large numbers throughout the world.
`<math.h>`	Contains function prototypes for math library functions.
`<setjmp.h>`	Contains function prototypes for functions that allow bypassing of the usual function call and return sequence.

Programmer-defined Headers

- Programmer-defined headers should also use the .h filename extension
- A programmer-defined header can be included by using the `#include` preprocessor directive
- For example, if the prototype for our square function was located in the header `square.h`, we'd include that header in our program by using the following directive at the top of the program:

```
#include "square.h"
```

Passing Arguments

- Two methods of passing arguments as parameters

- Call-by-value
    - copy of value is passed
    - Changes to the copy do not affect an original variable's value in the caller
- Call-by-reference
    - address of actual argument is passed
    - The caller allows the called function to modify the original variable's value.

Call-by-Value Parameters

- Copy of actual argument (value) passed
  - Considered local variable inside function
- If modified, only local copy (value) changes
  - internally function has no access to actual argument from caller
- This is the default method
  - used in all examples so far

Example of Call-by-Value

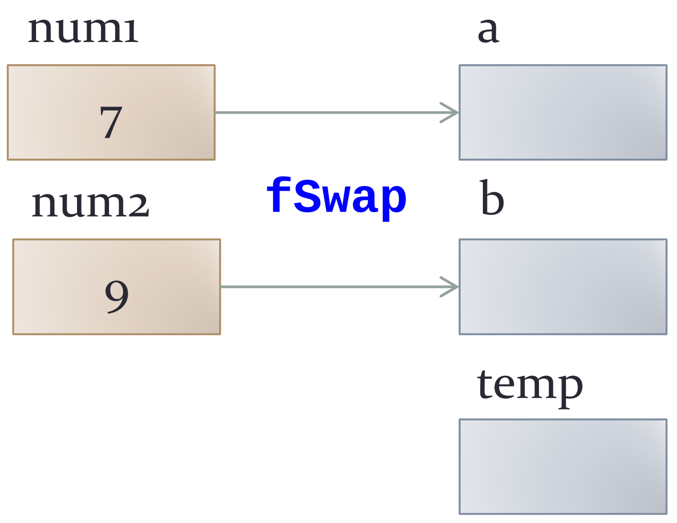
```
#include <stdio.h>

void fSwap(int a, int b);

int main()
{
    int num1 = 7;
    int num2 = 9;
    fSwap(num1, num2);
    printf("%d\n", num1);
    printf("%d\n", num2);

    return 0;
}
```

```
void fSwap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}
```



Call-by-Value Pitfall

- Common mistake
  - Declaring parameters again inside function:

```
double fee(int hrsWorked, int minsWorked)
{
    int quarterHours; // local variable
    int minsWorked;   // NO!
}
```

  - compiler error message "redefinition error…"
- Value arguments ARE like local variables
  - but function gets them automatically

Random Number Generator

- `rand()` picks a randomly chosen number
  - take no arguments
  - return value between 0 & `RAND_MAX`
  - RAND_MAX is *library-dependent* but at least 32767
- App#1: scaling `rand()% r`
  - squeeze random number into smaller range r. ex: `rand()%6`
  - return a random value between 0 & (r-1)
- App#2: shifting `rand()%r + b`
  - shift the starting value from the base b. ex: `rand()%6 + 1`
  - shift range between b & (r-1+b)

Example of Random Number

```
// Fig. 5.11: fig05_11.c
// Shifted, scaled integers produced by 1 + rand()%6.
#include <stdio.h>
#include <stdlib.h>

int main ( void )
{
    int i; // counter
    // loop 20 times
    for ( i = 1; i <= 20; i++)
    {
        printf ("%5d", 1 + (rand() % 6) );
        // if counter is divisible by 5, start a new line
        if ( i % 5 == 0)
            printf( "\n");
    }
    return 0;
}
```

Example of Random Number (cont.)

- screen output

```
       6      6      5      5      6
       5      1      1      5      3
       6      6      2      4      2
       6      2      3      4      1
```

`rand()` Function

- Repeatability is an important characteristic of `rand`
  - When debugging, repeatability is essential for proving that corrections to the program work properly
- Function `rand` actually generates *pseudorandom numbers*
- Repeatedly calling `rand` produces a sequence of numbers that appears to be random
  - However, the sequence repeats itself each time the program executes
- Once a program has been thoroughly debugged, it can be conditioned to produce a different sequence of random numbers for each execution

Randomizing with `srand`

- `srand(seed)` uses "`seed`" to alter the sequence of generating random numbers
  - a `void` function
  - require only one argument - the "`seed`"

- `seed` can be any value, including system time
  - EX: `srand(1067); srand(time(NULL));`
  - `time()` returns system time as numeric value in library `<time.h>`

Example of Randomizing

```
// Fig. 5.13: fig05_13.c
// Randomizing die-rolling program.
#include <stdio.h>
#include <stdlib.h>

int main ( void )
{
    int i;
    unsigned seed; // stores the seed entered by the user

    printf ( "Enter seed: ");
    scanf( "%u", &seed ); // note %u for unsigned
    srand( seed ); // seed random number generator
```

Example of Randomizing (cont.)

```
    // loop 10 times
    for ( i = 1; i <= 10; i++)
    {
        printf( "%5d", 1 + (rand() % 6) );
        // if counter is divisible by 5, start a new line
        if ( i % 5 == 0)
            printf( "\n" );
    }
    return 0;
}
```

Example of Randomizing (cont.)

- screen output

```
Enter seed: 67
     6     1     4     6     2
     1     6     1     6     4
```

```
Enter seed: 867
     2     4     6     1     6
     1     1     3     6     2
```

```
Enter seed: 67
     6     1     4     6     2
     1     6     1     6     4
```

Example: A Game of Chance: Craps

- A player rolls two dice
- After the dice have come to rest, the sum of the spots on the two upward faces is calculated
  - If the sum is 7 or 11 on the first throw, the player wins
  - If the sum is 2, 3, or 12 on the first throw (called "craps"), the player loses (i.e., the "house" wins)
  - If the sum is 4, 5, 6, 8, 9, or 10 on the first throw, then that sum becomes the player's "point." To win, you must continue rolling the dice until you "make your point." The player loses by rolling a 7 before making the point

```
/* Fig. 5.10: fig05_10.c
   Craps */
#include <stdio.h>
#include <stdlib.h>
#include <time.h> /* contains prototype for function time */

/* enumeration constants represent game status */
enum Status { CONTINUE, WON, LOST };

int rollDice( void ); /* function prototype */

/* function main begins program execution */
int main( void )
{
   int sum; /* sum of rolled dice */
   int myPoint; /* point earned */

   enum Status gameStatus; /* can contain CONTINUE, WON, or LOST */

   /* randomize random number generator using current time */
   srand( time( NULL ) );

   sum = rollDice(); /* first roll of the dice */
```

Fig. 5.10 | Program to simulate the game of craps. (Part 1 of 4.)

```c

   /* determine game status based on sum of dice */
   switch( sum ) {

      /* win on first roll */
      case 7:
      case 11:
         gameStatus = WON;
         break;

      /* lose on first roll */
      case 2:
      case 3:
      case 12:
         gameStatus = LOST;
         break;

      /* remember point */
      default:
         gameStatus = CONTINUE;
         myPoint = sum;
         printf( "Point is %d\n", myPoint );
         break; /* optional */
   } /* end switch */
```

Fig. 5.10 | Program to simulate the game of craps. (Part 2 of 4.)

```

   /* while game not complete */
   while ( gameStatus == CONTINUE ) {
      sum = rollDice(); /* roll dice again */

      /* determine game status */
      if ( sum == myPoint ) { /* win by making point */
         gameStatus = WON; /* game over, player won */
      } /* end if */
      else {
         if ( sum == 7 ) { /* lose by rolling 7 */
            gameStatus = LOST; /* game over, player lost */
         } /* end if */
      } /* end else */
   } /* end while */

   /* display won or lost message */
   if ( gameStatus == WON ) { /* did player win? */
      printf( "Player wins\n" );
   } /* end if */
   else { /* player lost */
      printf( "Player loses\n" );
   } /* end else */
```

Fig. 5.10 | Program to simulate the game of craps. (Part 3 of 4.)

```c

   return 0; /* indicates successful termination */
} /* end main */

/* roll dice, calculate sum and display results */
int rollDice( void )
{
   int die1; /* first die */
   int die2; /* second die */
   int workSum; /* sum of dice */

   die1 = 1 + ( rand() % 6 ); /* pick random die1 value */
   die2 = 1 + ( rand() % 6 ); /* pick random die2 value */
   workSum = die1 + die2; /* sum die1 and die2 */

   /* display results of this roll */
   printf( "Player rolled %d + %d = %d\n", die1, die2, workSum );
   return workSum; /* return sum of dice */
} /* end function rollRice */
```

Fig. 5.10 | Program to simulate the game of craps. (Part 4 of 4.)

```
Player rolled 5 + 6 = 11
Player wins
```

```
Player rolled 4 + 1 = 5
Point is 5
Player rolled 6 + 2 = 8
Player rolled 2 + 1 = 3
Player rolled 3 + 2 = 5
Player wins
```

```
Player rolled 1 + 1 = 2
Player loses
```

```
Player rolled 6 + 4 = 10
Point is 10
Player rolled 3 + 4 = 7
Player loses
```

Fig. 5.11 | Sample runs for the game of craps.

Enumeration Type

- Data type: a set of values together with a set of operations on those values
- To define a new simple data type, called enumeration type
- Syntax for enumeration type

```
enum typename {value1,value2,value3,…}
```

  - value1,value2, … are identifiers called enumerators
  - value1 < value2 < value3
- If a value has been used in one enumeration type => cannot be used by another in the same block

Examples of Enumeration Types

- Ex1:

```
enum colors {brown,blue,red,green};
```

- Ex2:

```
enum standing {FRESHMAN,SOPHOMORE,
               JUNIOR,SENIOR};
```

- Ex3: illegal! Why? Not valid identifiers!

```
enum grades {‘A’,‘B’,‘C’,‘D’,‘F’};
enum year {1st,2nd,3rd,4th,5th};
```

- Ex4: illegal! Why? Repeated enumerators!

```
enum Math {John,Peter,Sean,Joe};
enum Comp {Paul,Judy,Joe,Mary,Jane};
```

enum Manipulation

- Assignment and copy are legal
  - Ex: `sport PopularSport = BASEBALL;`
  - Ex: `sport MySport = PopularSport;`
- Arithmetic operations are NOT allowed!
  - Ex: `ASport = FOOTBALL + 2;`
  - Ex: `BSport = 2 * Volleyball;`
  - Ex: `CSport++; DSport--;`
- Relational operations are legal
  - Ex: `if (ASport > BSport)`

Example of enum Type

```
// example of enum date type
#include <stdio.h>

int main( void )
{
    enum Day {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
    enum Day myDay = Wed;

    if( myDay == Sat || myDay == Sun)
        printf( "Day is a weekend day\n" );
    else if( myDay == Wed)
        prinrf( "We need to meet at EE 634.\n");

    return 0;
}
```

```
We need to meet at ED 634.
```

Variable Scope

- Scope: the section of program where the identifier is valid (known or visible)
- Local variables (local scope): variables are created inside a function or a program component
  - The variable is meaningful only when used in expressions inside the function in which it was declared
- Global variables (global scope): variables created outside any function
  - Can be used by all functions physically placed after the global variable declaration
- Function prototypes should have global scope

Blocks and Scopes

- Declare data inside compound statement
  - called a block (`{ … }`)
  - has a block scope
- Note: all function definitions are blocks!
  - provides local function scope

Nested Scope

- Variables with the same name can be declared in multiple blocks in C/C++
  - totally legal: due to block scope
  - no ambiguity and distinct/unique within its own scope

```
{
    int var = 100;
    …
    {
        int var = x*y;
    }
}
```

Misuse of Globals

- Avoid overuse of globals
  - Too many globals eliminates safeguards provided by C to make functions independent
  - Misuse does not apply to function prototypes
- Prototypes are typically global
- Difficult to track down errors in a large program using globals
  - Global variable can be accessed and changed by any function following the global declaration

Variable Storage Class

- Variable scope: space dimension of variables
  - e.g. local, global
- Storage class: time dimension of variables
  - The length of time that the storage locations are reserved for a variable
  - Lifetime of a variable
  - Four types: `auto, static, extern, register`
  - Ex:

```
auto int num;
static int dist;
```

Variable Storage Class (cont.)

- Local variable storage classes:
  - `auto`: default class used by C/C++
    - `auto` variables local to the function are alive as long as the function has not returned control to its calling function
    - Variable storage is released back to the OS when return control to its calling function
    - *Run-time initialization*: initialization occurs each time the declaration statement is encountered
  - `static`: the program keeps the variable at its latest value even when the function that declared it finished executing
    - Local static variables remain in existence for the life of the program
    - Initialization (both local and global) is done only once when the program is first compiled
    - All `static` variables are set to zero when no explicit initialization is given

Variable Storage Class (cont.)

- `register`: the variables are stored in registers instead of main memory
  - Same time duration as `auto` variables
  - Increase the execution speed of the program
  - Decrease the size of a compiled C/C++ program
  - Variable will be switched to `auto` if
    - Compiler does not support `register` storage class
    - The declared register variables exceed the computer's register capacity

Example of Variable Storage Class

```
#include <stdio.h>

void test( void );
int main ( void )
{
    int count;
    for (count = 1; count <= 3; count++)
        test();
    return 0;
}
void test(void)
{
    static int num = 0; //or static int num;
    printf ( "num = %d\n", num);
    num++;
    return;
}
```

Example of Variable Storage Class (cont.)

- screen output

```
num = 0
num = 1
num = 2
```

Example of Scoping

```
// Fig. 5.16: fig05_16.c
// A scoping example.
#include <stdio.h>

void useLocal( void );
void useStaticLocal( void );
void useGlobal( void );

int x = 1; // global variable

int main ( void )
{
    int x = 5; // local variable to maim
    printf ( "local x in outer scope of main is %d\n", x);
    {
        int x = 7; // local variable to new scope
        printf ( "local x in inner scope of main is
                   %d\n", x);
    }
```

Example of Scoping (cont.)

```
    printf ( "local x in outer scope of main is %d\n", x);
    useLocal();
    useStaticLocal();
    useGlobal();
    useLocal();
    useStaticLocal();
    useGlobal();

    printf ( "local x in main is %d\n", x);
    return 0;
}
```

Example of Scoping (cont.)

```
void useLocal( void )
{
    int x = 25;
    printf( "\nlocal x in useLocal is %d after entering
             useLocal\n", x);
    x++;
    printf( "\nlocal x in useLocal is %d before exiting
             useLocal\n", x);
}
void useStaticLocal( void )
{
    // initialized only first time useStaticLocal is called
    static int x = 50;
    printf( "\nlocal static x is %d on entering
             useStaticLocal\n", x);
    x++;
    printf( "\nlocal static x is %d on exiting
             useStaticLocal\n", x);
}
```

Example of Scoping (cont.)

```
void useGlobal( void )
{
    printf( "\nglobal x is %d on entering useGlobal\n", x);
    x *= 10;
    printf( "\nglobal x is %d on exiting useGlobal\n", x);
}
```

Example of Variable Storage Class (cont.)

- screen output

```
local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5

local x in useLocal is 25 after entering useLocal
local x in useLocal is 26 before exiting useLocal

local static x is 50 on entering useStaticLocal
local static x is 51 on exiting useStaticLocal

global x is 1 on entering useGlobal
global x is 10 on exiting useGlobal
```

Example of Variable Storage Class (cont.)

- screen output

```
local x in useLocal is 25 after entering useLocal
local x in useLocal is 26 before exiting useLocal

local static x is 51 on entering useStaticLocal
local static x is 52 on exiting useStaticLocal

global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal

local x in main is 5
```

Variable Storage Class (cont.)

- Global variable storage classes: all global variables are initialized at compile time
  - Can not be declared as `auto` or `register`
  - Affect only the scope, not the time duration of global variables
  - `extern`: extend the scope of a existing global variable beyond a single file (e.g. **`extern int vector;`**)
  - `static`: similar to a static local variable but available for the rest of the program
    - Can not be extended beyond a single file
    - Provide a degree of privacy for the static global variable

Extern Global Variables

File1.c

```c
int volts;
float current;
static double power;
int main( void )
{
    fcn1();
    fcn2();
    fcn3();
    fcn4();
}
extern double factor;
//not available in main
int fcn1() {
   …
}
int fcn2() {
   …
}
```

File2.c

```c
double factor;
extern int volts;
int fcn3() {
   …
}
int fcn4() {
    extern float current;
    …
}
```

Summary

- Formal parameter is placeholder, filled in with actual argument in function call
- Call-by-value parameters are local copies in receiving function body
  - actual argument cannot be modified
- Call-by-reference passes memory address of actual argument
  - alternative name for variables
  - actual argument can be modified
  - argument MUST be variable, not constant

Summary (cont.)

- Scope: determines where in a program the variable can be used
- Variable storage class: determines how long the value in a variable will be retained