ICS103 Programming in C

Lecture 9: Functions I

Outline

- Review about Functions
- Types of Functions
  - void Functions with Arguments
    - Actual Arguments & Formal Parameters
    - Writing Modular programs using functions
  - Functions with Input Argument and a Single Result
    - Re-usability of Functions
    - Logical Functions
    - Functions with Multiple Arguments
    - Argument List Correspondence
    - The Function Data Area
    - Testing Functions Using Drivers
- Advantages of Using Function Subprograms
  - Procedural Abstraction
  - Reuse of Functions.

Review about Functions

- In chapter 3, we introduced functions as program modules that perform some operations that contribute towards solving the problem that a C program is designed to solve.
- We learnt how to use functions from the standard C library such as those in <math.h> and <stdio.h>.
- We also learnt the steps involved in defining our own (user-defined) functions, namely:
  - Declare the function prototype before the main function
  - Define the detail implementation of the function after the main function
  - Call the function from the main function where its operation is required
- However, we learnt to write only the simplest type of functions – those that take no argument and return nothing.
- In this Lecture, we shall learn how to write functions that take arguments, those that return a result and those that do both.

Types of Functions

- We use **function arguments** to communicate with the function. There are two types of function arguments:
  - **Input arguments** – ones that are used to pass information from the caller (such as main function) **to** the function.
  - **Output arguments** – ones that return results to the caller **from** the function. [we shall learn about these in the next lecture]
- Types of Functions
  - No input arguments, no value returned – void functions without arguments [already discussed in chapter 3]
  - Input arguments, no value returned - void functions with arguments.
  - Input arguments, single value returned.
  - Input arguments, multiple value returned [next lecture]

void Functions with Input Arguments …

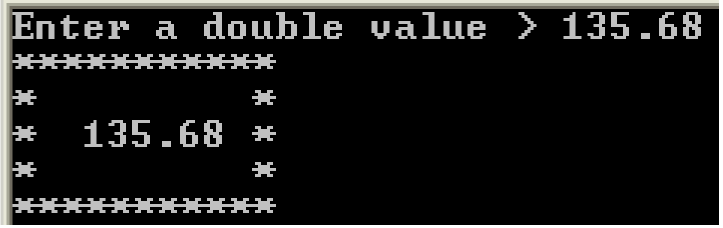
- A function may take one or more arguments as input but returns no result.
- Such functions should be declared as void, but each argument should be declared in the bracket following the function name
- An argument is declared in the same way as variables (its type followed by the name of the argument).

  Example: `void print_rboxed(double rnum);`
- If there are more than one argument, they should be separated by comma.

  Example: `void draw_rectangle(int length, int width);`
- The following function example displays its argument value in a rectangular box.

void Functions with Input Arguments…

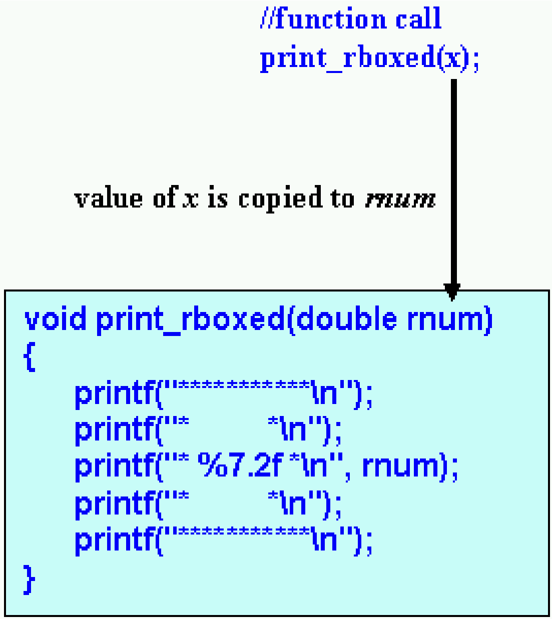
```
/* Uses the print_rboxed function to display a double argument */
#include <stdio.h>

void print_rboxed(double rnum);  //prototype for the function

int main(void) {
   double x;

   printf("Enter a double value > ");
   scanf("%lf", &x);
   print_rboxed(x);  //function call
  return 0;
}

/* Displays a real number in a box.  */
void print_rboxed(double rnum)
{
     printf("***********\n");
     printf("*         *\n");
     printf("* %7.2f *\n", rnum);
     printf("*         *\n");
     printf("***********\n");
}
```



```
Enter a double value > 135.68
***********
*         *
*   135.68 *
*         *
***********
```

Actual Arguments & Formal Parameters

- **Actual argument**: an **expression** used inside the parentheses of a function call.
- **Formal parameter**: An **identifier** that represents a corresponding actual argument in a function definition.
- Actual argument can be any expression that evaluates to a value expected by the function: x, 125.5, x+y, etc.



```
//function call
print_rboxed(x);
```

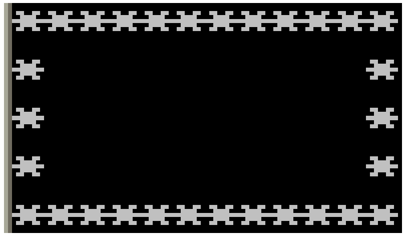
value of *x* is copied to *rnum*

```
void print_rboxed(double rnum)
{
    printf("*************\n");
    printf("*           *\n");
    printf("* %7.2f *\n", rnum);
    printf("*           *\n");
    printf("*************\n");
}
```

- When the function call is encountered at run-time, the expression for the actual argument is first evaluated, the resulting value is assigned to the formal parameter, then the function is executed.
- Arguments make functions more versatile because they enable a function to manipulate different data each time it is called.

Writing Modular programs using functions

- Suppose we wish to write a program that draws a rectangle similar to the following given the *length* and *width*.
- An algorithm for the solution could be:
  - Draw a solid line by printing ‘*’ width times
  - Draw a hollow line (‘*’, width-2 spaces and ‘*’) length – 2 times
  - Draw a solid line by printing * width times



```
************
*          *
*          *
*          *
************
```

- It is possible to write a very long main method to implement the above algorithm.
- However, this will involve many repetitions and the code will be difficult to re-use in another application.
- A better approach is to implement the different components of the solution as functions – this will result in a modular program and re-usable functions.
- The above algorithm could involve three functions, namely, draw_rectangle, draw_solid_line and draw_hollow_line

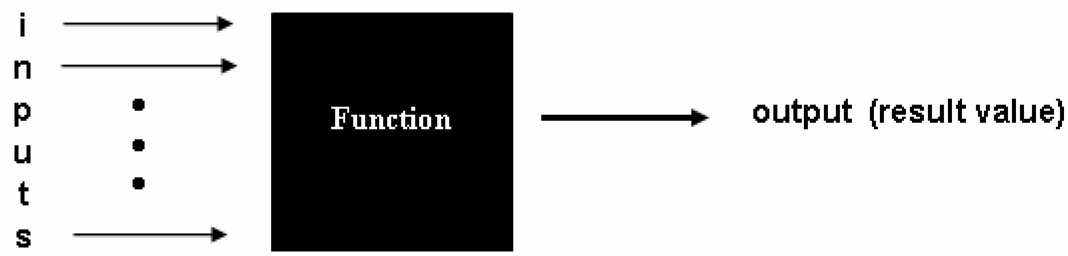
Writing Modular programs using functions …

```
//draws a rectangle using functions
#include <stdio.h>

void draw_solid_line(int size);
void draw_hollow_line(int size);
void draw_rectangle(int len, int wide);

int main(void) {
    int length, width;

    printf("Enter length and width of rectangle >");
    scanf("%d%d", &length, &width);

    draw_rectangle(length, width);

    system("pause");
    return 0;
}

void draw_solid_line(int size) {
     int i;
     for (i=1; i<=size; i++)
         printf("*");
     printf("\n");
}
```

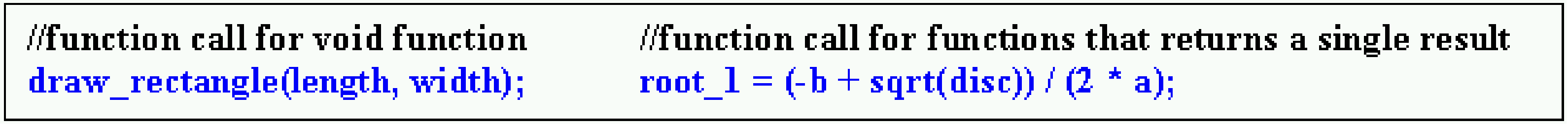
```
void draw_hollow_line(int size) {
     int i;
     printf("*");
     if (size > 2) {
        for (i=1; i<= size-2; i++)
            printf(" ");
     }
     printf("*\n");
}


void draw_rectangle(int len, int wide) {
     int i;
     draw_solid_line(wide);
     if (len > 2) {
        for (i=1; i<=len - 2; i++)
          draw_hollow_line(wide);
     }
     draw_solid_line(wide);
}
```

Functions with Input Argument and a Single Result

- By far, the must common types of functions in C are those that takes one or more arguments and return a single result.

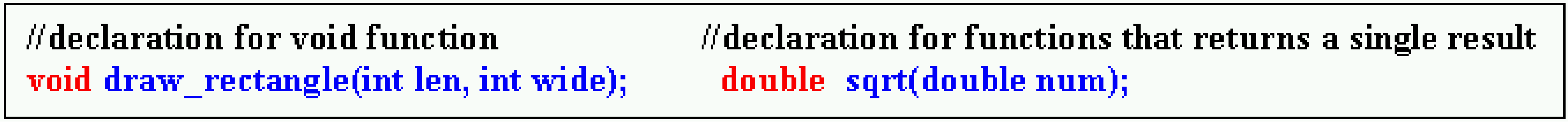


- For example, virtually all the functions in the <math.h> library, *sqrt*, *log*, *abs*, *sin*, *cos*, etc. are in this category.
- Unlike void functions, for which the function call is a statement on its own, functions that return a single result are often called as part of another expression.



```
//function call for void function          //function call for functions that returns a single result
draw_rectangle(length, width);             root_1 = (-b + sqrt(disc)) / (2 * a);
```

- To declare these types of function, instead of void, the function name is preceded by the type of result the function returns (int, double, char, etc.).



```
//declaration for void function                //declaration for functions that returns a single result
void draw_rectangle(int len, int wide);          double  sqrt(double num);
```

Functions with Input Argument and a Single Result…

- Functions that return a single result must have at least one return statement that returns the result to the calling function.

```
/* Computes n! for n greater than or equal to zero */
int factorial (int n) {
    int i,    /* local variables */
       product = 1;

    /* Computes the product n x (n-1) x (n-2) x ... x 2 x 1 */
    for (i = n;  i > 1;  --i) {
      product *= i;
    }

    /* Returns function result */
    return product;
}
```

The complete factorial example

```
/* Computes the factorial of a number  */
#include <stdio.h>
int factorial(int n);

/* shows how to call a user-define function */
int main(void) {
   int num, fact;
   printf("Enter an integer between 0 and 10> ");
   scanf("%d", &num);
   if (num < 0) {
      printf("Factorial not defined for negative
       numbers\n");
   } else if (num <= 7) {
      fact = factorial(num);
      printf("The factorial of %d is %d\n", num, fact);
   } else {
      printf("Number out of range: %d\n", num);
   }

   system("pause");
   return (0);
}
```

```
/*
  Computes n! for n greater than or equal
      to zero
 */
int factorial(int n)
{
      int i,     /* local variables */
          product = 1;

      /* Computes the product
         n x (n-1) x (n-2) x ...  x 2 x 1
       */
      for (i = n;  i > 1;  --i) {
          product *= i;
      }

      /* Returns function result */
      return (product);
}
```

Re-usability of Functions

- One important advantage of using functions is that they are reusable.
  - If we need to write another program that uses the same function, we do not need to re-write the function – just use the one we had before.
- For example, suppose we wish to write a program that computes how many different ways we can choose r items from a total n items.
  - E.g. How many ways we can choose 2 letters from 3 letters (A, B, C).
- From probability theory, this number is given by the combination formula:

$$C(n,r) = \frac{n!}{r!(n-r)!}$$

  - In our example, we have:

$$C(3,2) = \frac{3!}{2!*(3-2)!} = \frac{3!}{2!*1!} = \frac{3*2!}{2!*1} = 3 \quad (AB, AC, BC)$$

Re-usability of Functions …

```
/* Uses the combination functions to computes the number
  of combinations of n items taken r at a time */

#include <stdio.h>
int factorial(int n);
int combinations(int n, int r);

int main(void) {
   int n, r, c;

   printf("Enter total number of components> ");
   scanf("%d", &n);
   printf("Enter number of components selected> ");
   scanf("%d", &r);
   if (r <= n) {
        c = combinations(n, r);
        printf("The number of combinations is %d\n", c);
   } else {
        printf("Components selected cannot exceed total
      number\n");
   }

   system("pause");
   return (0);
}
```

```
/* Uses the factorial function to computes
      the number of  combinations of n items
      taken r at a time.   It assumes n >= r */
int combinations(int n, int r) {
   return factorial(n) / (factorial(r) *
      factorial(n-r));
}


/* Computes n! for n greater than or equal
      to zero */
int factorial(int n) {
     int i,     /* local variables */
         product = 1;


      for (i = n;  i > 1;  --i) {
         product *= i;
     }

     /* Returns function result */
     return (product);
}
```

Logical functions

- As we must observed by now, C uses integers to represent logical values.
- Thus, a function that returns a logical result should be declared as type int. In the implementation, the function return 0 for false or any other value for true.
- Example, the following is a logical function that checks if its integer argument is even or not. One important advantage of using functions is that they are reusable.

```
/* Indicates whether or not num is even. Returns 1 if it is, 0 if
not */
int even(int num) {
 int ans;
 ans = ((num % 2) == 0);
 return (ans);
}
```

- The function is used as follows:

```
if (even(n))
    even_nums++;
else
   odd_nums++;
```

Functions with Multiple Arguments

- As we saw with the functions *draw_rectangle (int len, int wide|)* and *combination (int n, int r)*, a function can have multiple arguments.
- Below is another function that multiplies its first argument by 10 raised to the power of its second argument.
- Function call `scale(2.5, 2)`returns the value 250.0

```
/*  Multiplies its first argument by the power of 10 specified
    by its second argument. */

double scale(double x, int n) {
     double scale_factor;     /* local variable */

     scale_factor = pow(10, n);
     return (x * scale_factor);
}
```

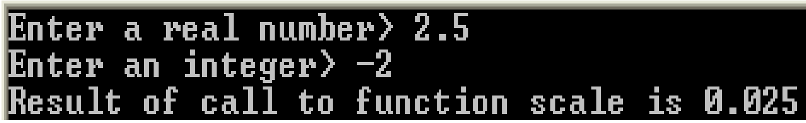
Argument List Correspondence

- When using multiple-argument functions, the number of actual argument used in a function call must be the same as the number of formal parameters listed in the function prototype.
- The order of the actual arguments used in the function call must correspond to the order of the parameters listed in the function prototype.
- Each actual argument must be of a data type that can be assigned to the corresponding formal parameter with no unexpected loss of information.

Testing Function scale

```
/* Tests function scale */
#include <math.h>
#include <stdio.h>
double scale(double x, int n);

int main(void) {
   double num_1;
   int num_2;

   /* Get values for num 1 and num 2 */
   printf("Enter a real number> ");
   scanf("%lf", &num_1);
   printf("Enter an integer> ");
   scanf("%d", &num_2);

   /* Call scale and display result. */
   printf("Result of call to function scale is
%.3f\n",  scale(num_1, num_2));
   system ("pause");
   return (0);}
```

```
double  scale(double x, int n)
{
   double scale_factor;
   scale_factor = pow(10, n);
   return (x * scale_factor);
}
```

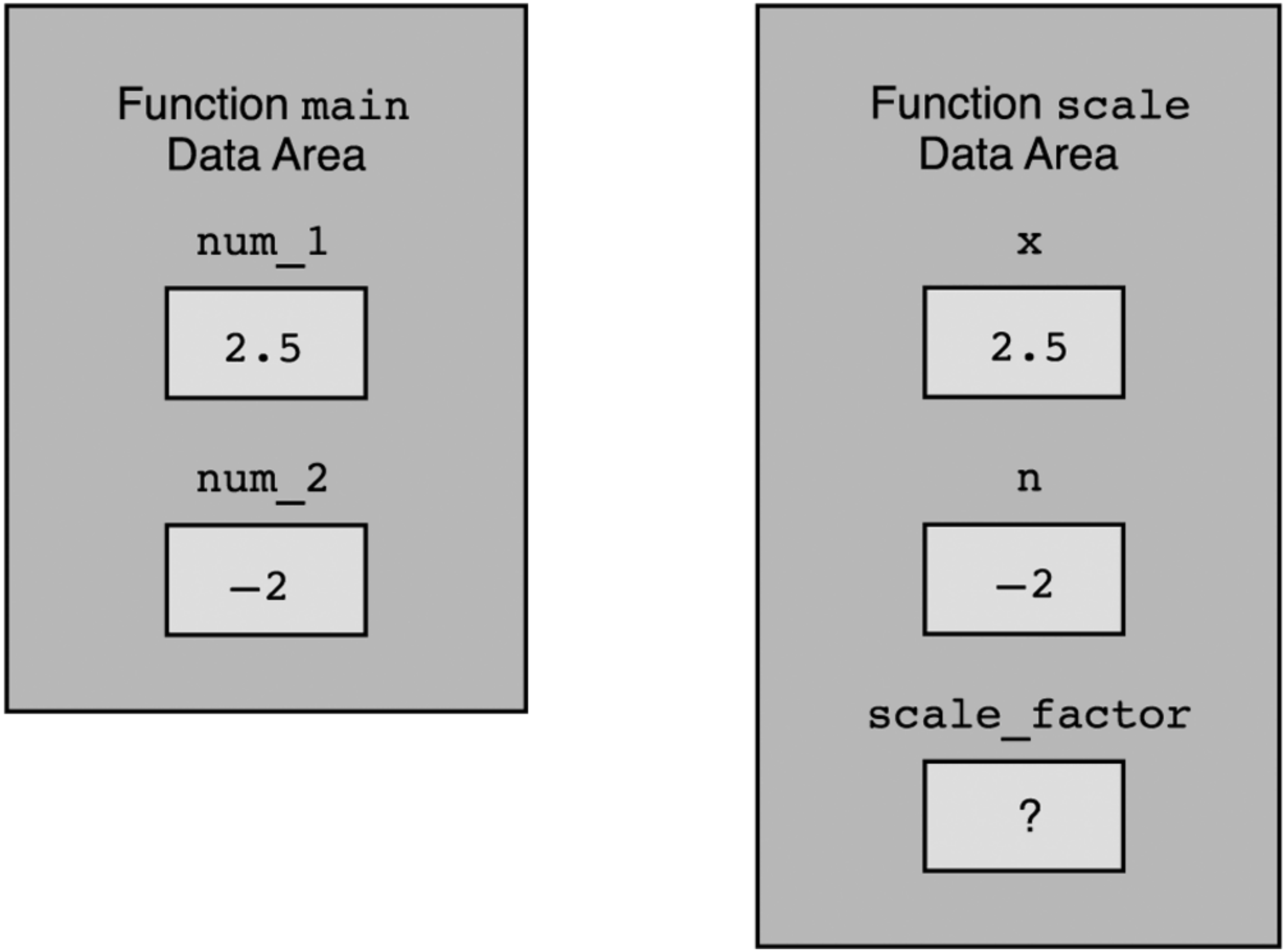


```
Enter a real number> 2.5
Enter an integer> -2
Result of call to function scale is 0.025
```

The Function Data Area

- Each time a function call is executed, an area of memory is allocated for storage of that function’s data.
- Included in the function data area are storage cells for its formal parameters and any local variables that may be declared in the function.
- **Local Variables**: variable declarations within a function body.
    - Can only be used from within the function they are declared in – no other function can see them
    - These variables are created only when the function has been activated and become undefined after the call.
- The function data area is always lost when the function terminates.
- It is recreated *empty* when the function is called again.
    - So if you set a local variable value, that value will be reset again next time the function is called.

Data Areas After Call scale(num_1, num_2);



Function `main`
Data Area

`num_1`

`2.5`

`num_2`

`–2`

Function `scale`
Data Area

`x`

`2.5`

`n`

`–2`

`scale_factor`

`?`

Testing Functions Using Drivers

- A function is an independent program module
- As such, it can be tested separately from the program that uses it.
- To run such a test, you should write a short piece of code called `driver` that defines the function arguments, calls the functions, and displays the value returned.
- As long as you do not change the interface, your function can be reused.

Why do we use Functions?

- There are two major reasons:

1. A large problem can be solved easily by breaking it up into several small problems and giving the responsibility of a set of functions to a specific programmer.
    - It is easer to write two 10 line functions than one 20 line one and two smaller functions will be easier to read than one long one.
2. They can simplify programming tasks because existing functions can be reused as the building blocks for new programs.
    - Really useful functions can be bundled into libraries.

Procedural Abstraction

- **Procedural Abstraction –** A programming technique in which a `main` function consists of a sequence of function calls and each function is implemented separately.
- All of the details of the implementation to a particular subproblem is placed in a separate function.
- The main functions become a more abstract outline of what the program does.
  - When you begin writing your program, just write out your algorithm in your main function.
  - Take each step of the algorithm and write a function that performs it for you.
- Focusing on one function at a time is much easier than trying to write the complete program at once.

Reuse of Function Subprograms

- Functions can be executed more than once in a program.
  - Reduces the overall length of the program and the chance of error.
- Once you have written and tested a function, you can use it in other programs or functions.

Common Programming Errors

- Remember to use a #include preprocessor directives for every standard library from which you are using functions.
- Place prototypes for your own function subprogram in the source file preceding the `main` function; place the actual function definitions after the `main` function.
- The acronym **NOT** summarizes the requirements for argument list correspondence.
  - Provide the required **N**umber of arguments
  - Make sure the **O**rder of arguments is correct
  - Make sure each argument is the correct **T**ype or that conversion to the correct type will lose no information.
- Include a statement of purpose on every function you write.
- Also be careful in using functions that are undefined on some range of values.