CS 201
Functions

Debzani Deb

Overview of Today's Lecture

- C Functions
- Types of Functions
  - void Functions without Arguments
  - void Functions with Arguments
  - Functions with arguments
- Advantages of Using Function Subprograms
  - Procedural Abstraction
  - Reuse of Functions.

C Functions

- A C program is a set of functions.
- C functions are just like Java methods – no objects, all public.
- Every C program must contain one function called `main`. This is where the program starts. Function `main` calls other functions.
- So far we wrote C programs with `main` function only.
- We also have seen predefined library functions being used in `main` function.
- Now we are going to create our own functions.


```
#include <stdio.h>
#define PI 3.14159
double find_area(double r);
int main(void){
   double radius, area;

   printf("Enter radius> ");
   scanf("%lf", &radius);

   area = find_area(radius);

   printf("The area is %.2f\n",area );
   return(0);
}
double find_area(double r){
   double a;

   a = PI*r*r;
   return (a);
}
```

Return Type
Function arguments
Function Declaration/Prototype
Function call
Function Header
Local declarations
Executable Statements
Function Body
Function Definition

Enter radius> 10
The area is 314.16

Types of Functions

- We can use **function arguments** to communicate with the function
  - **Input arguments** – ones that used to pass information from the caller to the function.
  - **Output arguments** – ones that return results to the caller from the function.
- Types of Functions
  - No input arguments, no value returned – void functions without arguments
  - Input arguments, no value returned - void functions with arguments.
  - Input arguments, single value returned.
  - Input arguments, multiple value returned.
    - Will cover in chapter 6.

void Functions Without Arguments

- The function just does something without communicating anything back to its caller.
  - Output is normally placed in some place else (e.g. screen)
  - Function Prototypes
  - Function Definitions
  - Placement of Functions in a Program
  - Program Style

Function Prototype (1)

```
/* This program draws a circle in the screen */
#include <stdio.h>
/* Function prototypes */
void draw_circle(void);
int main(void)
{
   draw_circle();
   return (0);
}
/* Draws a circle */
void draw_circle(void) {
   printf("  * *\n");
   printf(" *   *\n");
   printf("  * *\n");
}
```

1st void means no value returned, 2nd void means no input arguments.

Function Prototype (2)

- Like other identifiers in C, a function must be declared before it can be referenced.
- To do this, you can add a **function prototype** before `main` to tell the compiler what functions you are planning to use.
- A function prototype tells the C compiler:
  1. The data type the function will return
     - For example, the `sqrt` function returns a type of double.
  2. The function name
  3. Information about the arguments that the function expects.
     - The `sqrt` function expects a double argument.
- So the function prototype for `sqrt` would be:

```
double sqrt(double);
```


Function Definition (1)

```
/* This program draws a circle in the screen */
#include <stdio.h>
/* Function prototypes */
void draw_circle(void); /* Draws a circle */
int main(void)
{
   draw_circle();
   return (0);
}
/* Draws a circle */
void draw_circle(void) {
   printf("  * *\n");
   printf(" *   *\n");
   printf("  * *\n");
}
```


Function Definition (2)

- The prototype tells the compiler what arguments the function takes and what it returns, but not what it does.
- We define our own functions just like we do the `main` function
  - **Function Header** – The same as the prototype, except it is not ended by the symbol ;
  - **Function Body –** A code block enclosed by {}, containing variable declarations and executable statements.
- In the function body, we define what actually the function does
  - In this case, we call `printf` 3 times to draw a circle.
  - Because it is a void function, we can omit the return statement.
- Control returns to `main` after the circle has been drawn.

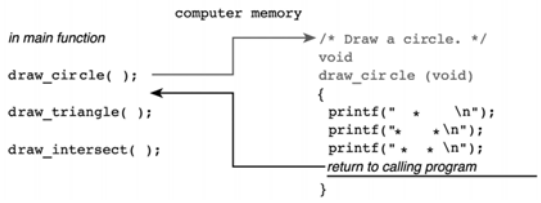
Placement of Functions in a program

- In general, we will declare all of our function prototypes at the beginning (after `#include` or `#define`)
- This is followed by the `main` function
- After that, we define all of our functions.
- However, this is just a convention.
- As long as a function's prototype appears before it is used, it doesn't matter where in the file it is defined.
- The order we define them in does not have any impact on how they are executed

Execution Order of Functions

- Execution order of functions is determined by the order of execution of the **function call** statements.
- Because the prototypes for the function subprograms appear before the `main` function, the compiler processes the function prototypes before it translates the `main` function.
- The information in each prototype enables the compiler to correctly translate a call to that function.
- After compiling the `main` function, the compiler translates each function subprogram.
- At the end of a function, control always returns to the point where it was called.

Figure 3.15 Flow of Control Between the main Function and a Function Subprogram



Program Style

- Each function should begin with a comment that describes its purpose.
- If the function subprograms were more complex, we would include comments on each major algorithm step just as we do in function `main`.
- It is recommended that you put prototypes for all functions at the top, and then define them all after `main`.

A good use of void functions – A separate function to display instructions for the user.

```
1.  /*
2.   * Displays instructions to a user of program to compute
3.   * the area and circumference of a circle.
4.   */
5.  void
6.  instruct(void)
7.  {
8.        printf("This program computes the area\n");
9.        printf("and circumference of a circle.\n\n");
10.       printf("To use this program, enter the radius of\n");
11.       printf("the circle after the prompt: Enter radius>\n");
12. }
```

```
This program computes the area
and circumference of a circle.

To use this program, enter the radius of
the circle after the prompt: Enter radius>
```


void Functions with Input Arguments

- In the last section, we used void functions like `draw_circle` to display several lines of program output.
- A void function does not return a result, but we can still pass it arguments.
- For example, we could have a function display its argument value in a more attractive way.
- The effect of the function call `print_rboxed(135.68);`

```
1.  /*
2.   * Displays a real number in a box.
3.   */
4.
5.  void
6.  print_rboxed(double rnum)
7.  {
8.        printf("***********\n");
9.        printf("*          *\n");
10.       printf("* %7.2f *\n", rnum);
11.       printf("*          *\n");
12.       printf("***********\n");
13. }
```

```
***********
*         *
*  135.68 *
*         *
***********
```


Functions with Input Arguments

- Actual Argument & Formal parameter
- Functions with Input Arguments and a Single Result
- Program Style
- Functions with Multiple Arguments
- Argument List Correspondence
- The Function Data Area
- Testing Functions Using Drivers

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 (135.68) is passed into the function and substituted for its formal parameter `rnum`.
- Arguments make functions more versatile because they enable a function to manipulate different data each time it is called.

Functions with Input Arguments and a Single Result

- We can call these functions in expressions just like library functions.
- Let's consider the problem of finding the area and circumference of a circle using functions with just one argument.

```
#define PI 3.14159
/* Compute the circumference of a circle with radius r */
double find_circum(double r) {
   return (2.0 * PI * r);
}
/* Compute the area of a circle with radius r */
double find_area(double r) {
   return (PI * pow(r,2));
}
```

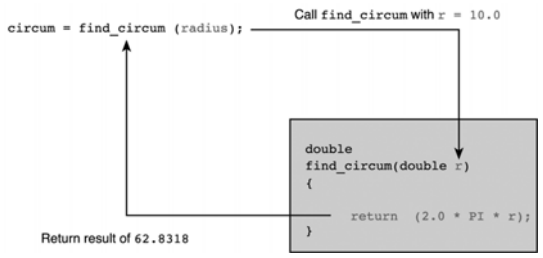

Functions with Input Argument and a Single Result

- Each function heading begins with the word `double`, indicating that the function result is a real number.
- Both function bodies consist of a single return statement.
- When either function executes, the expression in its return statement is evaluated and returned as the function result.
- If we call the function like:

```
area = find_area(5.0)
```

- The value we **returned** from the function will be assigned to `area`.

Figure 3.22 Effect of Executing `circum = find_circum (radius);`



More on Functions

- Make sure that you understand the difference in function calls between void functions and functions that returns a single value.

```
draw_circle();
print_rboxed(135.68);
area = find_area(5.0)
```

- A function call that returns a result must do something with the result, otherwise, the value returned will be lost.

Functions with Multiple Arguments

- We can also define functions with multiple arguments.
- Function call `scale(2.5, 2)` returns the value 250.0

```
1.  /*
2.   * Multiplies its first argument by the power of 10 specified
3.   * by its second argument.
4.   * Pre : x and n are defined and math.h is included.
5.   */
6.  double
7.  scale(double x, int n)
8.  {
9.      double scale_factor;     /* local variable */
10.     scale_factor = pow(10, n);
11.
12.     return (x * scale_factor);
13. }
```


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 or 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.

Figure 3.24 Testing Function scale

```
/*
 * Tests function scale.
 */

#include <math.h>

/* Function prototype */
double scale(double x, int n);

int
main(void)
```

(continued)

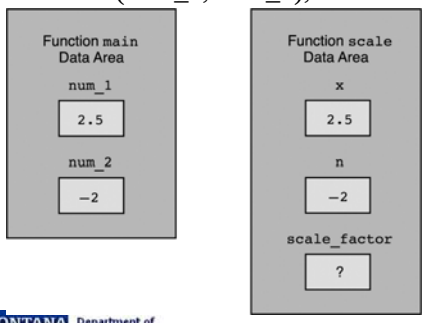
Figure 3.24 Testing Function scale (cont'd)

```
{
    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 %f\n",
           scale(num_1, num_2));     actual arguments

    return (0);
}
                                     information flow
double
scale(double x, int n)               formal parameters
{
    double scale_factor;     /* local variable = 10 to power n */

    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 not still be set next time the function is called.

Figure 3.25 Data Areas After Call scale(num_1, num_2);



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 becomes 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.