Functions

Chapter 05

CMPE-112 *Programming Fundamentals*

Lecture Plan

- Two sample programs
- Function Definition
  - *return* Statement
- Function Call
  - Call *by value*
  - Call *by reference*
- Function Prototypes
- Scope of Variables
- External Variables
- Storage Classes
- Recursion

Sample Program (I)

```
/* The program computes n! / m! */
#include  <stdio.h>

int main()
{
  int i, n, m, fact_n, fact_m;

  puts(“\nEnter two numbers”);
  scanf(“%d %d”, &n, &m);

  for (fact_n = 1, i=1; i<=n; i++)      /*  n!  */
    fact_n *= i;

  for (fact_m = 1, i=1; i<=m; i++)   /*  m! */
    fact_m *= i;

  printf(“\nResult = %f\n”, (float)fact_n / fact_m);

  return 0;
}
```

```
/* The program computes n! / m! */
#include  <stdio.h>

int fact( int a )
{
  int i, fff;

  for (fff = 1, i=1; i<=a; i++)      /*  a!  */
    fff *= i;

  return fff;
}

int main()
{
  int n, m;

  puts(“\nEnter two numbers”);
  scanf(“%d %d”, &n, &m);
  printf(“\nResult = %d\n”, (float)fact(n) / fact(m));

  return 0;
}
```

Function Definition (I)

- A *function definition* introduces a new function. The following declarations are made in the function definition:
  - the type of value that the function returns
  - the order and the type of its parameters
  - the statements to be executed when the function is called

```
function_type   function_name (parameter_declarations)
{
   variable_declarations

   function_statements
}
```

- A function that does not return any value, is declared to be of the type *void*. The default type is taken to be *int*.

Function Definition (II)

- *Function_name* is the name of the function being defined
- *Parameter_declarations* specify the types and names of the *parameters* (also called *formal parameters*) of the function, separated by commas

```
double pow( double x, double y )
```

- If a function does not have any parameters, the keyword *void* is used in place of parameter declarations

```
void  terminating_message( void )
```

Function Definition (III)

- The *function body* consists of *variable_declarations* followed by *function_statements,* enclosed in braces
- *Variable_declarations* specify types and names of the variables that are local to the function
- *Function_statements* are executed when the function is called

```
int check_range( int v1, int v2, int x )
{
   int result;

   result = x >= v1 && x <= v2;

   return result;
}
```

Function Definition (IV)

- *A local variable* is one whose value can be accessed only by the function in which it is declared
- Parameters are declared at the top of the function body
- Variable declared local supersede any identically named variables outside the function

```
int abc( int v1, int v2 )
{
   int result;
   …
}
int def( int v1, int v2 )
{
   int result;
   …
}
```

return Statement (I)

- A *return* statement can be of the two following forms

  return *expression*;

  return;
- The type of the expression is converted to the type of the function

```
int truncate( double x )                     int truncate( double x )
{                         equivalent         {
   return x;              ═════════▶            return  (int)x;
}                                            }
```

- The second form is used when the function is of type *void*; otherwise, the value returned is unpredictable
- If there is no *return* statement, the <u>second</u> form is assumed

return Statement (II)

- More than one *return* statement can be used in a function. Each of them terminates the execution of the function, and the rest of the function body is not executed

```
/* The function computes a! */

int fact( int a )
{
   int i, result;

   if ( a < 0 )     return -1;

   if ( a = = 0 )  return 1;

   for (result = 1, i=1; i<=a; i++)      /*  a!  */
      result *= i;

   return  result;
}
```

Function Call (I)

- A function call is an expression of the form

  function_name (argument_list)

  where *function_name* is the name of the function called, and *argument_list* is a comma-separated list of expressions (actual arguments) to the function
- A function call is an expression, its value is the one returned by the function

  z = sqrt(sin(x));
- Parenthesis **must** be present in the function call even when the argument list is empty

  initialize();

Function Call (II)

- The function in which the function call is contained is the ***calling*** function, and the other is a *called* one, e.g. ***main()*** is a calling function and *sin(x)* is a called one
- The called function is executed until a *return* or its *closing brace* is encountered, and the control passes back to the point **after** the function call
- The calling function may ignore the valued returned by a called function. So, the following two statements are both valid

  ```
  z = sin(x);
  sin(y);
  ```

  but the *sine* of *y* is lost, since it is not stored in a variable

Parameter Passing

- Parameters can be passed **from** a calling function **to** a called one in one of the two ways:
  - the called function is provided with the current *values* of the actual argument, and the corresponding formal parameters are assigned these values - **call by value**. So, any change in the value of the parameter *does not* cause a change in the corresponding argument since these are two <u>different</u> locations in the memory
  - the called function is provided with the *addresses* of the actual arguments – **call by reference.** Here any change in the value of the parameter automatically means a change in the corresponding argument since they refer to <u>the same</u> cell in the memory
- C language provides *call by value* parameter passing only, although it allows to pass the address of a variable (called the *pointer* to the variable) as a parameter

Function Prototypes

- Before calling a function, it must be declared with a prototype of its parameters. The general form of a function declaration is as follows

  function_type   function_name (parameter_type_list);

- The *parameter_list* is the comma-separated list of pairs of type and name of the parameters of the function
- Parameter names in the prototype may be omitted
- The prototype of a function must agree with the function definition and its use
- A function definition serves as a prototype for any subsequent call to the function in the same source file

Scope of Variables

- A **block** is a sequence of variable declarations and statements enclosed in braces
- C does not allow a function to be defined inside another function, but it permits nested blocks
- At the beginning of a block some variables may be declared and initialized. The *scope of a variable* declared in a block extends **from** its point of declaration **to** the end of the block
- *Scope* is the part of a code within which a name can be used
- Such a declaration *hides* any identically named variables in the outer blocks. So, a variable name addresses *the latest* declared location in the memory – all previously declared cells are not accessible

Example

- Note the scope of the variable *tmp* used for swapping the values of two variables in the following code

```
if (m < n)   // Swap them
{
    int  tmp = m;
    m = n;
    n = tmp;
}
```

- The variable *tmp* exists only within this *if* statement and is not accessible from any outer block

External Variables

- Sometimes passing values to a function via parameters is difficult is a large number of variables has to be shared
- Variables defined *outside* any function at the same level as function definitions are available to all the functions defined below in the same source file, and they are called **external variables** (or *global variables*)
- If a local variable and a global one have identical names, all references to the name within the function will refer to the local variable
- External variables are useful when
  - *many arguments* are to be passed to a function
  - a function needs to return *more than one* result

Storage Classes

- A variable is of *automatic* storage class if a cell is allocated to it upon entry to a segment of code and deallocated upon exit from this segment
- A variable is of *static* storage class if a cell is allocated to it at the beginning of the program execution and remains allocated until the program execution terminates
- By default, all variables declared within a block are taken to be *auto*, while those declared outside all blocks at the same level as function definitions are always static
- For the explicit declaration a storage class specifier can be used as follows:

```
auto   type  variable1;
static  type  variable2;
```

Recursion

- When a function calls itself (directly, or indirectly) it is called a recursive function

```
/* The recursive function computing n! */

int factorial( int a )
{
  if (a = = 0)
     return  1;       /* Termination condition */
  else
     return  a * factorial(a-1);   /* Recurse */
}
```

```
/* The recursive function computing n! */

int factorial( int a )
{
   return  a = = 0 ? 1 : a * factorial(a-1);
}
```

```
/* The function computes n! */

int fact( int a )
{
  int i, fff;

  for (fff = 1, i=1; i<=a; i++)      /*  a!  */
     fff *= i;

  return  fff;
}
```