FUNCTIONS

Defining and Accessing a Function, Function prototypes

Designed by Parul Khurana, LIECA.

Functions

- A function is a self-contained program segment that carries out some specific, well-defined task.
- Every C program consists of one or more functions. One of these functions must be called main.
- *Execution of the program will always begin* by carrying out the instructions in main.

Functions

- If a program contains multiple functions, their definitions may appear in any order, though they must be independent of one another.
- That is, one function definition cannot be embedded within another.

Functions

- A **function will carry out its intended action** whenever it is accessed (i.e., **whenever the function is "called"**)from some other portion of the program.
- The *same function can be accessed from several different places within a program*. Once the function has carried out its intended action, control will be returned to the point from which the function was accessed.

Functions

- Generally, a function will process information that is passed to it from the calling portion of the program, and **return a single value**.
- Information is passed to the function **via special identifiers called arguments** (**also called parameters**), and returned via the **return statement**. Some functions, however, accept information *but do not return anything (of type void)*, whereas *other functions return multiple values*, with the help of **pointers and arguments**.

Functions

- Defining a Function
  - A function definition has two principal components: the first line (including the argument declarations), and the body of the function.
  - The first line of a function definition contains the **type specification of the value** returned by the function, followed by the **function name**, and (optionally) a set of arguments, separated by commas and enclosed in parentheses.

Functions

- Defining a Function
  - Each argument is preceded by its associated type declaration.
  - An empty pair of parentheses must follow the function name if the function definition does not include any arguments.

Functions

- Defining a Function
  - In general terms, the first line can be written as:
    - data-type name (type 1 arg1 , type 2 arg 2, . . ., type n arg n)
      - Where data-type represents the data type of the item that is returned by the function, name represents the function name, and type 1, type 2, . . ., type n represent the data types of the arguments arg 1, arg 2, . . , arg n.
      - The arguments are called formal arguments, because they represent the names of data items that are transferred into the function from the calling portion of the program. They are also known as **dummy arguments** or **formal parameters**.

Functions

- Defining a Function
  - In general terms, the first line can be written as:
    - data-type name (type 1 arg1 , type 2 arg 2, . . ., type n arg n)
      - **The identifiers used as formal arguments are "local" in the sense that they are not recognized outside of the function**. Hence, the names of the formal arguments need not be the same as the names of actual arguments in the calling portion of the program.
      - **Each formal argument must be of the same data type, however, as the data item it receives from the calling portion of the program.**

Functions

- Defining a Function - Example
  - Consider the function lower_to_upper in the following example:

```
char lower_to_upper( char c1)
 {
    char c2;
    c2 = ( c1 >= 'a' && c1 <= 'z') ? ( 'A' + c1 - 'a' ) : c1;
    return ( c2 );
 }
```

  - **The first line contains the function name, lower_to_upper, followed by the formal argument c1, enclosed in parentheses.**

Functions

- Defining a Function - Example
  - Consider the function lower_to_upper in the following example:

```
char lower_to_upper( char c1)
  {
     char c2;
     c2 = ( c1 >= 'a'  &&  c1 <= 'z')  ?  ( 'A'  + c1  -  'a'  )  :  c1;
     return  ( c2 );
   }
```

  - **The function name is preceded by the data type char, which describes the data item that is returned by the function. In addition, the formal argument c1 is preceded by the data type char.**

Functions

- Defining a Function - Example
  - Consider the function lower_to_upper in the following example:

```
char lower_to_upper( char c1)
 {
    char c2;
    c2 = ( c1 >= 'a' && c1 <= 'z') ? ( 'A' + c1 - 'a' ) : c1;
    return ( c2 );
 }
```

  - **The body of the function begins on the second line, with the declaration of the local char-type variable c2. Following the declaration of c2 is a statement that tests whether c1 represents a lowercase letter and then carries out the conversion.**

Functions

- Defining a Function - Example
  - Consider the function lower_to_upper in the following example:

```
char lower_to_upper( char c1)
 {
    char c2;
    c2 = ( c1 >= 'a' && c1 <= 'z') ? ( 'A' + c1 - 'a' ) : c1;
    return ( c2 );
 }
```

  - **The original character is returned intact if it is not a lowercase letter. Finally, the return statement causes the converted character to be returned to the calling portion of the program.**

Functions

- Defining a Function - return statement
  - Information is returned from the function to the calling portion of the program via the return statement.
  - The return statement also causes the program logic to return to the point from which the function was accessed.

Functions

- Defining a Function - return statement
  - In general terms, the return statement is written as
    - **return expression;**
    - **return;**

❑**The value of the expression is returned to the calling portion of the program.**

❑**The expression is optional. If the expression is omitted, the return statement simply causes control to revert back to the calling portion of the program, without any transfer of information.**

❑**Only one expression can be included in the return statement.**

Functions

- Accessing a Function
  - A function can be accessed (i.e., called) by specifying its name, followed by a list of arguments enclosed in parentheses and separated by commas.
  - If the function call does not require any arguments, an empty pair of parentheses must follow the name of the function.
  - The function call may be a part of a simple expression (such as an assignment statement), or it may be one of the operands within a more complex expression.

Functions

- Accessing a Function
  - The arguments appearing in the function call are referred to as actual arguments, in contrast to the formal arguments that appear in the first line of the function definition.
  - In normal function call, there will be one actual argument for each formal argument.
  - The actual arguments may be expressed as constants, single variables, or more complex expressions.

Functions

- Accessing a Function
  - However, each actual argument must be of same data type as its corresponding formal argument.
  - Remember that it is the value of each actual argument that is transferred into the function and assigned to the corresponding formal argument.

Functions

- Accessing a Function - Example
  - If the function returns a value, the function access is often written as an assignment statement; e.g.,
    - y= polynomial (x);
      - This *function access causes the value returned by the function* to be assigned to the variable y.

Functions

- Accessing a Function - Example
  - On the other hand, if the function does not return anything, the function access appears by itself; e.g.,
    - display (a, b, c);
      - This *function access causes the values of a, b and c to be processed internally (i.e.,* displayed) within the function.

Functions

- Defining a Function - Example
  - **Consider once again the function lower_to_upper in the following example:**

```
char lower_to_upper( char c1)
  {
     char c2;
     c2 = ( c1 >= 'a'  &&  c1 <= 'z')  ?  ( 'A'  + c1  -  'a'  )  :  c1;
     return  ( c2 );
  }
void main ( void)
{
char lower, upper;
printf("Please enter a lowercase character: "); scanf("%c", &lower);
upper = lower_to_upper ( lower );
printf ( "\n the uppercase equivalent is %c \n", upper);
}
```

Within this program, main contains only one call to the programmer-defined function lower_to_upper.

Functions

- Defining a Function - Example
  - **Consider once again the function lower_to_upper in the following example:**

The call is a part of the assignment expression upper = lower_to_upper(lower).

```
char lower_to_upper( char c1)
  {
     char c2;
     c2 = ( c1 >= 'a'  &&  c1 <= 'z') ? ( 'A'  + c1  -  'a'  ) : c1;
     return  ( c2 );
   }
void main ( void)
{
char lower, upper;
printf("Please enter a lowercase character: "); scanf("%c", &lower);
upper = lower_to_upper ( lower );
printf ( "\n the uppercase equivalent is %c \n", upper);
}
```

Functions

- Defining a Function - Example
  - **Consider once again the function lower_to_upper in the following example:**

The function call contains one actual argument, the char-type variable lower.

When the function is accessed , the value of lower to be transferred to the function.

```
char lower_to_upper( char c1)
  {
     char c2;
     c2 = ( c1 >= 'a'  &&  c1 <= 'z')  ?  ( 'A'  + c1  -  'a'  )  :  c1;
     return  ( c2 );
   }
void main ( void)
{
char lower, upper;
printf("Please enter a lowercase character: "); scanf("%c", &lower);
upper = lower_to_upper ( lower );
printf ( "\n the uppercase equivalent is %c \n", upper);
}
```

Functions

- Defining a Function - Example
  - **Consider once again the function lower_to_upper in the following example:**

This value is represented by c1 within the function. The value of uppercase equivalent, c2, is then determined and returned to the calling portion of the program, where it is assigned to the char-type variable upper.

```
char lower_to_upper( char c1)
  {
     char c2;
     c2 = ( c1 >= 'a'  &&  c1 <= 'z')  ?  ( 'A'  + c1  -  'a'  )  :  c1;
     return  ( c2 );
  }
void main ( void)
{
char lower, upper;
printf("Please enter a lowercase character: "); scanf("%c", &lower);
upper = lower_to_upper ( lower );
printf ( "\n the uppercase equivalent is %c \n", upper);
}
```

Functions

- Function Prototypes
  - In the previous example, we have examined that the programmer-defined function has always preceded main.
  - Thus, when these programs are compiled, the programmer-defined function will have been defined before the first function access.
  - However, many programmers prefer a " top-down " approach, in which main appears ahead of the programmer-defined function definition.

Functions

- Function Prototypes
  - In such situations the function access (within main) will precede the function definition.
  - This can be confusing to the compiler, unless the compiler is first altered to the fact that the function being accessed will be defined later in the program.
    - **A function prototype is used for this purpose**.

Functions

- Function Prototypes
  - Function prototypes are usually written at the beginning of a program, ahead of any programmer - defined functions (including main).
  - The general form of a function prototype is:
    - **data- type name (type 1 arg 1, type 2 arg 2, ... , type n arg n);**
      - where data - type represents the data type of the item that is returned by the function, name represents the function name, and type 1, type 2, ... , type n represent the data types of the arguments arg 1, arg 2, . . , arg n.
      - Notice that function prototype resembles the first line of a function definition (**though a function prototype ends with a semicolon**).

Functions

- Function Prototypes
  - The names of the arguments within the function prototype need not be declared elsewhere in the program, since these are "dummy" argument names that are recognized only within the prototype.
  - In fact, the **argument names can be omitted**; however, the argument data types are essential.

Functions

- Function Prototypes
  - Function **prototypes are not mandatory** in **C**. They are desirable because they further facilitate error checking between the calls to a function and the corresponding function definition.

Functions

- Function Prototypes - Example
  - **Consider once again the function lower_to_upper in the following example (Revisited):**

Here is a complete program to convert lowercase character into uppercase character.

```
char lower_to_upper( char c1);   / * function prototype */
void main ( void)  {
char lower, upper;
printf("Please enter a lowercase character: "); scanf("%c", &lower);
upper = lower_to_upper ( lower );
printf ( "\n the uppercase equivalent is %c \n", upper);
                        }
 char lower_to_upper( char c1)  {
       char c2;
       c2 = ( c1 >= 'a'  &&  c1 <= 'z')  ?  ( 'A'  + c1  -  'a'  )  :  c1;
       return  ( c2 );                    }
```

Practice Questions

- Predict the result of the following code segment:

```
#include <stdio.h>
int i;
void increment( int i )
{  i++; }

int main()
{
  for( i = 0; i < 10; increment( i ));
  printf("i = %d\n", i);
  return 0;
}
```

Practice Questions

- Predict the result of the following code segment:

```
#include <stdio.h>
void func()
{  int x = 0;
 static int y = 0;
 x++; y++;
 printf("%d --%d\n",x,y); }

int main() {
 func(); func();
 return 0;
}
```

Practice Questions

- Write appropriate function prototypes for the following skeletal outlines:

```
main()
{
int a, b, c;
. . .
c = funct1 ( a, b );
. . .
}
```

```
main()
{
char a,
unsigned int c;
. . .
c = code ( a);
. . .
}
```

```
main()
{
int a,
float b;
double c;
. . .
c = funct2 ( b, a);
. . .
}
```

Practice Questions

- Write a function declared as float power (float m, int n); that returns m^n .
- Write a function declared as void qroot (int a, int b, int c); that prints the roots of a quadratic equation; $ax^2 + bx + c = 0$.
- Write a function declared as void pattern(int n); that displays the n lines of the following pattern:

1
2 3
4 5 6
. . . . .
upto n lines

or

a
b c
d e f
. . . . .
upto n lines