CHARUTAR VIDYA MANDAL'S
SEMCOM
Vallabh Vidyanagar

Faculty Name: Ami D. Trivedi
Class: FYBCA
Subject: US01CBCA01 (Fundamentals of Computer Programming Using C)

*UNIT 4: (Strings, User-Defined Functions and Command-line arguments)

ONE-DIMENSIONAL CHARACTER ARRAY (STRING)

A **string** is a sequence of characters which is treated as a single data item. Any group of characters (except double quote sign) defined between double quotations marks is a constant string.

Example: "Man is obviously made to think."

Character strings are often used to build meaningful and readable programs. The common operations performed on character strings are:

- Reading and Writing strings.
- Combining strings together.
- Copying one string to another.
- Comparing strings for equality.
- Extracting a portion of a string.

*Storage representation of One Dimensional Character Array (string)

Suppose we store the string constant "WELL DONE" into the string variable name. Each character of the string is treated as an element of the array name and we may declare the variable name as

char name[10];

and the computer reserves ten storage locations as shown below:

name[0]	name[1]	name[2]	name[3]	name[4]	name[5]	name[6]	name[7]	name[8]	name[9]

When we assign values to array element, it will look as follows:

'W'	'E'	'L'	'L'	' '	'D'	'O'	'N'	'E'	'\0'
name[0]	name[1]	name[2]	name[3]	name[4]	name[5]	name[6]	name[7]	name[8]	name[9]

*Declaration of One Dimensional character array (string)

C does not support string as a data type. The C language treats character strings simply as arrays of characters. So, a string variable in C is any valid C variable name and it is always declared as an array of characters.

The general form of declaration of a string variable is:

char string_name [size] ;

The size represents the maximum number of characters in the string_name.

char name[10]; Declares **name** as a character array (string) variable that can hold maximum of 10 characters (including null character).

When compiler assigns a character string to a character array, it automatically appends a **null character ('\0')** at the end of the string. So, the size should be equal to maximum number of characters in the string plus one.

***Initialization of One Dimensional character Array (string)**

An array can be initialized at either of the following stages:

1) At compile time (at time of declaration of an array)
2) At run time (input form user and with assignment statement)

1) Compile time initialization of One Dimensional character array

char name[6] = { 'H', 'E', 'L', 'L', 'O', '\0' }; **OR** char name[6] = "HELLO";

Above statements declares name to be an array of 6 characters and initialize it with string "HELLO" ending with null character. Name string has to be 6 characters long because it needs 5 characters to store string HELLO and one character to store null terminator.

- Here, size may be omitted. In such cases, the size will be automatically determined based on number of initialized elements. For example,
  char name[ ] = { 'H', 'E', 'L', 'L', 'O', **'\0'** }; **OR** char name[ ] = "HELLO"; will declare name array to contain six elements including null character.
- We can also declare size larger than the string size in initializer. For example,
  char name[10] = "HELLO"; is allowed. In this case, computer creates a character array of size 10, places the value "HELLO", terminates with null character and initializes all other element will NULL.
- char name[3] = "HELLO" ; is illegal. It will result in compilation error.
- char name[10];
  name = "HELLO"; is not allowed.
- char name1[5] = "GOOD", name2[10];
  name2 = name1; is not allowed. Because array name can not be used as left operand in assignment operator.

2) Run time initialization (input form user and character-by-character copy / with strcpy function)

An array can be explicitly initialized at run time. This approach is usually applied for initializing string as per user input.

Run time initialization can be done by

(1) Taking input from user or
(2) Character-by-character copy / with strcpy function

(1) Taking input from user

Using scanf()

```
char name[10];
printf ("Enter name : ");
scanf ("%s", name);
```

Using gets()

```
char name[10];
printf ("Enter name : ");
gets(name);
```

The problem with scanf function is that it terminates its input on the first white space it finds. (A whitespace includes blanks, tabs, carriage returns, form feeds, and new lines.)

Therefore, if the following line of text is typed in at the terminal,

NEW YORK

then only the string "NEW" will be read into the array name, since the blank space after the word "NEW" will terminate the string.

Note that in the scanf calls of character arrays, the ampersand (&) is not required before the variable name.

To read a string of text containing whitepace, another more convenient method is available. We can use library function gets() available in <stdio.h>. This is a simple function with one string parameter. It does not skip whitespaces.

Note: be careful not to input more character than size of string variable. C does not check array-bounds. It may create problems.

1-D character array (string) also can be inputted with following method:

```
char name[10];
for (i=0; i<5; i++)
      scanf("%c", &name[i]);
```

In this example, for loop will be executed 5 times and asks user to input one character each time. First character will be stored in 0^{th} element of name array, second input in 1^{st} element of name array and so on.

Character array is collection of character type elements. So, we can input individual elements by writing a for loop.

Character array represents single meaningful information like name, address, designation, remarks etc. So this method of inputting each character individually is rarely used.

(2) character-by-character copy / with strcpy function

C does not allow assigning one string directly to another string with assignment statement. We may assign one string to another sting with the help of strcpy function or we can copy character by character.

```
char name1[10] = "HELLO", name2[10] ;
for (i=0; name1[i] != '\0';  i++)
      name2[i] = name1[i];
name2[i] = '\0';
```

```
char name1[10] = "HELLO", name2[10] ;
strcpy (name2, name1);
```

Note: Here name1's content will be copied in name2.

TWO-DIMENSIONAL CHARACTER ARRAY (TABLE OF STRINGS)

We often use lists of character strings, such as a list of names of students in a class, list of names of employees in an organization, list of places, etc.

A list of names can be treated as a table of strings and a two-dimensional character array can be used to store the entire list. For example, a character array city[30][15] may be used to store a list of 30 city names, each of length not more than 15 characters.

Shown below is a table of five cities:

C	h	a	n	d	i	g	a	r	h	\0
M	a	d	r	a	s	\0				
A	h	m	e	d	a	b	a	d	\0	
H	y	d	e	r	a	b	a	d	\0	
B	o	m	b	a	y	\0				

This table can be conveniently stored in a character array city by using the following declaration:

char city[30] [15] = { "Chandigarh", "Madras", "Ahmedabad", "Hyderabad", "Bombay" };

To access the name of i^{th} city, we write city[i-1]. Table of strings is treated as a collection of multiple one dimensional arrays.

***Storage representation of Two Dimensional Character Array**

City

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**		
0	C	h	a	n	d	i	g	a	r	h	\0	←	city[0]
1	M	a	d	r	a	s	\0					←	city[1]
2	A	h	m	e	d	a	b	a	d	\0			:
3	H	y	d	e	r	a	b	a	d	\0			:
4	B	o	m	b	a	y	\0					←	city[4]

***Declaration of Two Dimensional character array**

The general form of declaration of a 2-D character array is:

char string_name [row_size][column_size] ;

The row_size represents the maximum number of rows (total number of strings) and column_size represents number of characters in each row.

char city[5][15]; Declares the **city** as a 2-D character array variable that can hold a maximum of 5 city name each with15 characters (including null character).

***Initialization of Two Dimensional character Array (string)**

An array can be initialized at either of the following stages:

1) At compile time (at time of declaration of an array)
2) At run time (input from user)

1) Compile time initialization of Two Dimensional character array

char city[5][15] = { { ‘C’, ‘h’, ‘a’, ‘n’, ‘d’, ‘i’, ‘g’, ‘a’, ‘r’, ‘h’, ‘\0’ } , { ‘M’, ‘a’, ‘d’, ‘r’, ‘a’, ‘s’, ‘\0’ } };

OR char city[5][15] = { “Chandigarh”, ”Madras” };

Above statements

- Declares city to be a 2-D array which is capable of storing 5 city names each with length of 15 characters (including null character).
- And assigns two city names to city[0] and city[1] element.

Here, size may be omitted. In such cases, the size will be automatically determined based on number of initialized strings. For example,

char city[ ][15] = { { ‘C’, ‘h’, ‘a’, ‘n’, ‘d’, ‘i’, ‘g’, ‘a’, ‘r’, ‘h’, ‘\0’ } , { ‘M’, ‘a’, ‘d’, ‘r’, ‘a’, ‘s’, ‘\0’ } };
OR char city[ ][15] = { “Chandigarh”, ”Madras” };

will declare city array to contain two city name each with length of 15 characters (including null character).

2) Run time initialization (input form user)

```
char city[5][10];
int i;
for (i=0; i<5; i++)
    {    printf (“Enter city name : “);
         gets(city[i]);
    }
```

In this example, for loop will be executed 5 times and asks user to input one city name each time. First city name will be stored in 0^{th} element of city array, second city name in 1^{st} element of city array and so on.

Note: instead of gets(), scanf() with %s can also be used.

STRING HANDLING FUNCTIONS

Following are the most commonly used string-handling functions.

	Function	Action
1	strlen( )	Finds the length of a string.
2	strcmp( )	Compare two strings.
3	strcpy( )	Copies one string over another.
4	strcat( )	Concatenates two strings.
5	strrev( )	Reverse a string

Few Useful functions:

6	gets( )	Input a string
7	puts( )	Display a string
8	strlwr()	Converts uppercase to lowecase
9	strupr()	Converts lowercase to uppercase

1. strlen() Function

Header file: <STRING.H>

It is used to find length of a string. i.e. This function counts and returns the number of characters in a string. It takes the form:

n = strlen(string);

Where n is an integer variable, which receives the value of the length of the string. The argument may be a string constant. The counting ends at the first null character.

```
#include <string.h>
#include <stdio.h>
void main()
{  char str1[10] = “HELLO”;
   n = strlen(str1);
   printf(“Length = %d”,n);
}
```

Output:
Length = 5

Note:
1. n= strlen(“HELLO”); is allowed. And it will produce same output.
2. String str1 can be inputted from user also.

2. strcmp() Function

Header file: <STRING.H>

The strcmp function compares two strings identified by the arguments.

- It has a value 0 if both string arguments are equal.
- If string arguments are not equal then it has the numeric difference between the first nonmatching characters in the strings.

It takes the form: **strcmp (string1, string2);**

string1 and string2 may be string variables or string constants. Examples are:

strcmp(name1,name2);
strcmp(name1,”Good”);
strcmp(“ROM”,”RAM”);

Our requirement is to check whether the strings are equal. If not equal then we want to check that which string is alphabetically above.

For example, the statement **strcmp(“their”,”there”);**

will return a value of –9 which is the numeric difference between ASCII “i” and ASCII “r”. That is, “i” minus “r” in ASCII code is –9. If the value is negative, string1 is alphabetically above string2 (means string1 is smaller than string2).

3. strcpy() Function

Header file: <STRING.H>

The strcpy function works almost like a string-assignment operator. It takes the form

strcpy (string1, string2);

and assigns the contents of string2 to string1. string2 may be a character array variable or a string constant. For example, the statement

strcpy(city, "DELHI"); will assign the string "DELHI" to the string variable city.
strcpy(city1,city2); will assign the contents of the string variable city2 to the string variable city1

The size of the array city1 should be large enough to receive the contents of city2.

For example, consider the following two strings:

	0	1	2	3	4	5
part1	H	E	L	L	O	\0

	0	1	2
part2	H	I	\0

Execution of the statement **strcpy(part1,part2);** will result in:

	0	1	2
part1	H	I	\0

	0	1	2
part2	H	I	\0

4. strcat( ) function

Header file: <STRING.H>

The strcat function joins two strings together. It takes the following form:

strcat (string1, string2);

string1 and string2 are character arrays.

When the function strcat is executed, string2 is appended to string1. It does by removing the null character at the end of string1 and placing string2 from there. The string at string2 remains unchanged.

For example, consider the following two strings:

	0	1	2	3	4	5	6	
part1	V	E	R	Y		\0		

	0	1	2	3	4	5	6	
part2	G	O	O	D	\0			

Execution of the statement **strcat(part1,part2);** will result in:

	0	1	2	3	4	5	6	7	8	9	
part1	V	E	R	Y		G	O	O	D	\0	

	0	1	2	3	4	5	6				
part2	G	O	O	D	\0						

We must take care for size of string in which other string is being appended.

Receiving string must be large enough to accommodate the final string.

strcat function can append a string constant to string variable. For example, **strcat(part1,"GOOD");** is valid.

5. strrev() Function

Header file: <STRING.H>

It reverses all characters of string (except for the terminating null).

Syntax: strrev(string);

For example, it would change hello\0 to olleh\0.

```
#include <string.h>
#include <stdio.h>
void main()
 {
    char s[20]= "string";
    printf ("Before strrev() : %s\n", s);
    strrev (s);
    printf ("After strrev() :  %s\n", s);
 }
```

Output:

```
Before strrev() : string
After strrev() : gnirts
```

6. gets()

Header file: <STDIO.H>

gets() gets a string from standard input device (stdin) i.e. generally keyboard

Syntax: **gets(string);**

gets collects a string of characters terminated by a new line from the standard input stream stdin and store it into s.

gets also allows input strings to contain certain whitespace characters (spaces, tabs).

```
#include <stdio.h>
void main()
{
   char str[80];
   printf("Input a string : ");
   gets(str);
   printf("The string input was : %s\n", str);
}
```

Output:

```
Input a string : How are you
The string input was : How are you
```

7. puts()

Header file: <STDIO.H>

puts: outputs a string to standard output (stdout) i.e. generally monitor and appends a newline character

Syntax: puts(string);

puts copies the null-terminated string s to the standard output stream stdout and appends a newline character.

printf with %s may be replaced by puts() function to print (display) strings.

```
#include <stdio.h>
void main()
{
   char str[30] = "This is an example";
   puts(str);
}
```

Output:

```
This is an example
```

8. strlwr() Function

Header file: <STRING.H>

It converts uppercase letters (A to Z) in given string to lowercase (a to z). No other characters are changed.

Syntax: strlwr(string);

```
#include <stdio.h>
#include <string.h>
void main()
{
   char str[30] = "Borland International";
   printf ("string before strlwr : %s\n", str);
   strupr (str);
   printf ("string after strlwr : %s\n", str);
}
```

Output:

```
string before strlwr : Borland International
string after strlwr : borland international
```

9. strupr() Function

Header file: <STRING.H>

It converts lowercase letters (a to z) in given string to uppercase (A to Z). No other characters are changed.

Syntax: strupr(string);

```
#include <stdio.h>
#include <string.h>
void main()
{  char str[30] = "Borland International";
   printf ("string before strupr : %s\n", str);
   strupr (str);
   printf ("string after strupr : %s\n", str);
}
```

Output:

```
string before strupr : Borland International
string after strupr : BORLAND INTERNATIONAL
```

***FUNCTIONS**

C functions can be classified into two categories, namely, **library functions** and **user-defined functions**.

main is an example of user-defined functions. printf and scanf belong to the category of library functions. We have also used other library functions such as sqrt, strcat, gets etc.

The main difference between these two categories is that library functions are not required to be written by us whereas a user-defined function has to be developed by the user at the time of writing a program.

A user defined function can become a part of C program library later on. This is one of the strength of C language.

Need for user-defined functions

main is a specially recognized function in C. Every program must have a main function to indicate where the program has to begin its execution. It is possible to code any program utilizing only main function. This leads to a number of problems.

The program may become too large and complex and as a result the task of debugging, testing, and maintaining becomes difficult.

If a program is divided into functional parts, then each part may be independently coded and later combined into a single unit. These subprograms called 'functions' are much easier to understand, debug, and test.

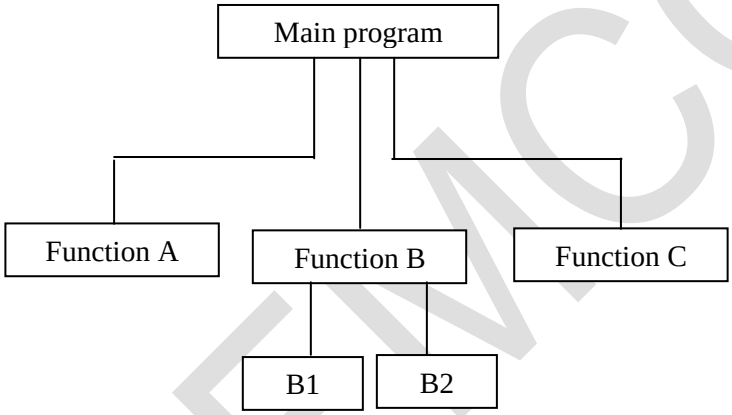
There are times when certain type of operations or calculations is repeated at many points throughout a program. For example, we may use factorial of a number at several points in the program.

In such situations, we may repeat the program statements whenever they are needed. Another approach is to design a function that can be called and used whenever required. This saves time and space.

This sub-sectioning approach clearly results in a number of **advantages**.

1. It facilitates top-down modular programming as shown in fig.. In this programming style, the high level logic of the overall problem is solved first while the details of each lower-level function are addressed later.
2. The length of a source program can be reduced by used functions at appropriate places. This factor is particularly critical with microcomputers where memory space is limited.
3. It is easy to locate and isolate a faulty function for further investigations.
4. A function may be used by many other programs. This means that a C programmer can build on what others have already done, instead of starting over, from scratch.

Top-down modular programming, using functions



A multi-function program

A function is a self-contained block of code that performs a particular task.

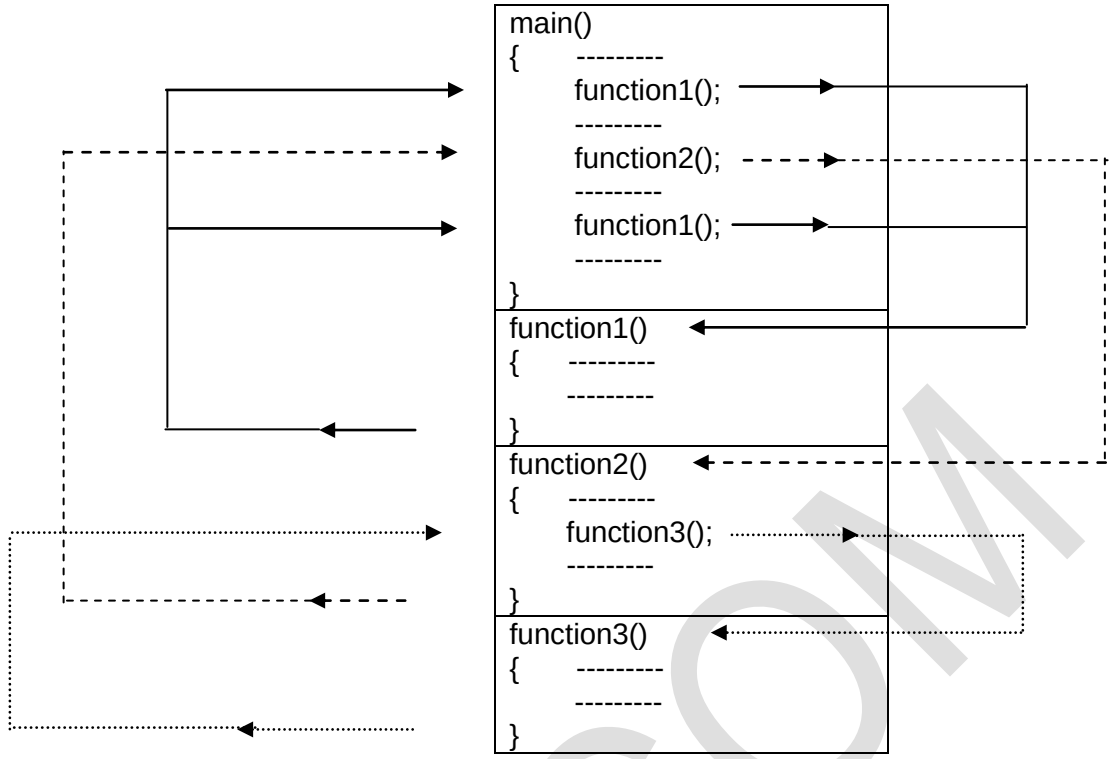
Once a function has been designed, it can be treated as a 'black box' that takes some data from the main program and returns a value. The inner details of operation are invisible to the rest of the program.

All that the program knows about a function is: What goes in and what comes out. Every C program can be designed used a collection of these black boxes.

In following diagram:

- main function has 3 function calls: function1, function2 and again function1. We can call same function as many as times we wish.
- Function call to function1 will jump to function1 and return to calling function (here main) after executing all statements of function1.
- When function2 is called from main, it will jump to function2. From function2, there is a function call to function3. So it will jump to function3. After executing all statements of function3, it will return to function2. Because function2 is a calling function of function3. After executing remaining statements of function2, it will return to main function.

Flow of control in a multi-function program



***ELEMENTS OF USER DEFINED FUNCTION**

To use a user-defined function, we need to create 3 elements related to functions:

- **A.** Function definition
- **B.** Function call
- **C.** Function declaration

A. FUNCTION DEFINITION

Function definition is an independent module that is specially written to implement the requirements of the function.

A function definition is also known as function implementation. It consists of 2 parts:

- **(1) Function header**
- **(2) Function body**

Function header contains 3 elements:	**Function body** contains 3 elements:
1. Function type **2.** Function name **3.** List of parameters	**1.** Local variable declaration **2.** Function statements **3.** A return statement

General form of function definition to implement these two parts is given below:

```
function_type   function-name (parameter  list)
{
   local variable declarations;
   executable statement1;
   executable statement2;
   ……..
   ……..
   return statement;     // optional
}
```

The first line **function_type function-name (parameter list)** is known as function header. And the statements within the opening and closing curly brackets forms function body.

(1) Function Header

Function header consists of 3 parts.

1. Function type (also known as return type)
2. Function name
3. List of parameters (formal parameters)

1. Function type (also known as return type)

The function type will specify the type of the value which is returned to calling function.

- If function is not retuning anything then we have to specify the return type as void.
- If return type is not specified then, it is considered as integer by default.

2. Function name

Function name is any valid C variable name. Name should be appropriate as per the task of function.

3. List of parameters (formal parameters)

Parameter list declares the variables that are used to receive the data sent by calling function. Data received in formal parameters are used by the function to perform processing.

They represent actual input values, so they are called **formal** parameters. Parameters are also known as **arguments**.

Parameter list is separated by comma and enclosed within round brackets.

Example: **int sum (int a, int b) { …… }**

Note:

- Declaration of parameter variables can not be combined. i.e. int sum (int a, b) is illegal.
- To indicate empty parameter list, use void keyword between parenthesis as: **int sum (void)**
  Many compilers accept empty set of parenthesis as: **int sum()**

(2) Function Body

Function body contains 3 elements namely:

1. Local variable declaration **:** It specifies the extra variables needed by the function to carry out processing.
2. Function statements **:** It consists of necessary statements to perform required task.
3. A return statement **:** If a function does not return any value, we can omit the return statement.

A return statement

A function may or may not send any value back to the calling function. A function can send any value to calling function using **return** statement.

Called function can return only one value at the most.

It can take one of following forms:

return; OR **return (expression);**

We can use **return;** statement for void function that does not return any value.

return (expression); is used to return the value of expression. Examples are:

return (p);	return (x+y);	return (5 * 3);

Note:

1. All functions by default return int type data.
2. When a function returns value, it is automatically cast (converted) to function's type. For example, if we return float value but if int return type is mentioned then returned value is truncated to an integer. 7.5 will return 7, i.e. only integer part.

B. FUNCTION CALL

To use function, we need to call it at required place in program. This is known as function call.

The function that calls the function is known as calling function. The function which is being called by the calling function is known as called function

A function can be called by simply using function name. Function name followed by pair of parenthesis which contains list of actual parameters (or arguments), if any.

```
void main()
{
    int c;
    c = sum (10, 5);   //Function call
    printf ( "%d", c);
}
```

C. FUNCTION DECLARATION

The calling function should declare the functions which are going to be used later on in the program. This is known as function declaration or **function prototype**.

All functions in C program must be declared, before they are invoked (used). A function declaration consists of four parts.

1. Function type (return type)
2. Function name
3. Parameter list
4. Terminating semicolon

They are mentioned in following format:

Function-type function-name (parameter list) ;

This is similar to function header except the terminating semicolon. For example, **sum** function defined in the previous section can be declared as:

int sum (int a, int b); /* Function prototype */

Above statement can be written as:

int mul (int, int); OR mul (int a, int b); OR mul (int, int);

Points to remember:

1. Parameter list must be separated by comma.
2. Parameter names need not be same in prototype declaration and the function definition.
3. Types of parameters in prototype declaration must match the types of parameters in function definition. Also number of parameters and order must match. When the declared types do not match with the types in function definition, compiler will give error.
4. Use of parameter names in prototype declaration is optional.
5. If the function has no formal parameters, the empty parameter list is written as (void). Writing void is optional in some compilers.
6. The return type in prototype declaration is optional, when function returns int type data.

Parameters used in prototype and function definition are called **formal parameters**. Parameters used in function calls are called **actual parameters**.

A prototype declaration may be placed at two places in a program.

1. Above all functions
2. Inside function definition

When we write prototype declaration above all the functions, the prototype is known as **global prototype**. When we write prototype declaration inside function definition, the prototype is known as **local prototype**.

Note: Prototype declarations are not required if a function definition is placed before a function is used.

*CATEGORY OF FUNCTION

In functions, arguments may be present or not, And the value may be returned or not. Based on this, functions belong to one of the following categories:

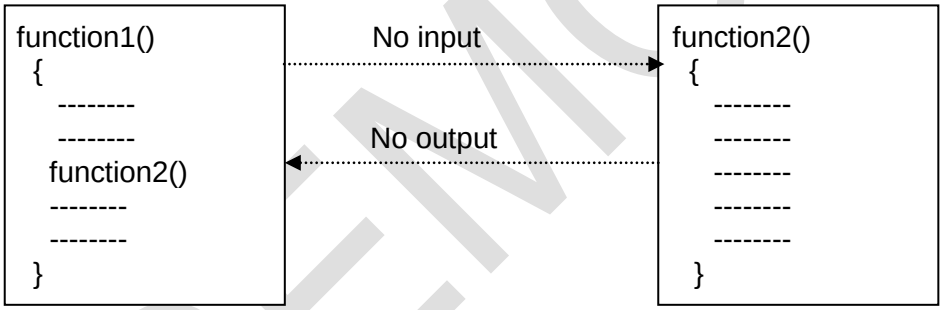
1. Functions with no arguments and no return values
2. Functions with no arguments and one return value
3. Functions with arguments and no return values
4. Functions with arguments and one return value
5. Functions with multiple return values (using pointer)

1. Functions with no arguments and no return values

When a function has no arguments, it does not receive any data from the calling function.

Similarly, when it does not return a value, the calling function does not receive any data from the called function. In effect, there is no data transfer between the calling function and the called function.

No data communication between functions



A function that does not return any value can not be used in an expression. It can be used as an independent statement.

Example:

```
#include <stdio.h>
#include <conio.h>
void si ()
{
  float p,r,s;
  int n;
  printf("Enter p, r and n : ");
  scanf("%f %f %d",&p,&r,&n);
  s=(p * r * n) / 100;
  printf("Simple interest = %.2f",s);
}                                   Cont.
```

```
void main()
{
  clrscr();
  si();
  getch();
}
```

Output:

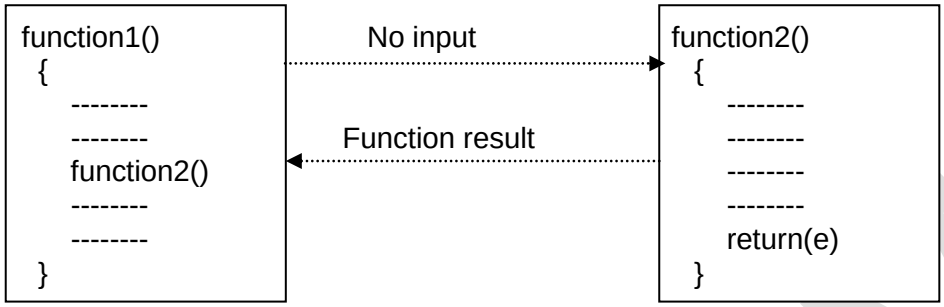
```
Enter p, r and n : 10000 10 5
Simple interest = 5000.00
```

2. Functions with no arguments and one return value

There could be situation where we need to design functions that may not take any arguments but returns a value to the calling function.

e.g getchar() function of <stdio.h>. This function has no arguments but it returns an integer type data that represents a character.

We can design similar function and use it in our programs.



Example:

```
float si ()
{
   float p,r,s;
   int n;
   printf("Enter p, r and n : ");
   scanf("%f %f %d",&p,&r,&n);
   s=(p * r * n) / 100;
   return(s);
}
```

Cont.

```
void main()
{
   float sint;
   clrscr();
   sint=si();
   printf("Simple interest = %.2f",sint);
   getch();
}
```

Output:
Enter p, r and n : 10000 10 5
Simple interest = 5000.00

The called function **si** will be executed line by line in normal fashion until **return(s)**; statement. At this time, float value **s** is passed back to function call in the main. And following indirect assignment occurs at the place of function call:

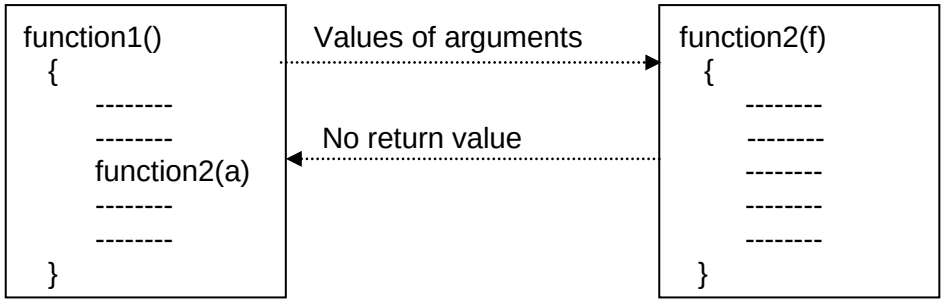
si () = s; The calling statement will assign returned value to **sint**.

3. Functions with arguments and no return values

We can make the calling function to read data from user and pass it to called function. This approach is good. Because the calling function can check the validity of data before handed over to called function.

The nature of data communication between the calling function and the called function with arguments but no return value is shown in figure.

One-way data communication



Example:

```
#include <stdio.h>
#include <conio.h>
void si (float p, float r, int n)
{
   float s;
   s=(p * r * n) / 100;
   printf("Simple interest = %.2f",s);
   getch();
}                                    Cont.
```

```
void main()
{
   float p1,r1;
   int n1;
   clrscr();
   printf("Enter p, r and n : ");
   scanf("%f %f %d",&p1,&r1,&n1);
   si(p1, r1, n1);
}
```

Output:
Enter p, r and n : 10000 10 5
Simple interest = 5000.00

The arguments p, r, and n are called the formal arguments. The arguments p1, r1, and n1 are called the actual arguments.

The calling function sends values of actual arguments to formal arguments using function calls with actual arguments.

For e.g., the function call **si(p1, r1, n1)** will send the values 10000, 10 and 5 to the function **si(p, r, n)** and assign 10000 to p, 10 to r and 5 to n.

The values 10000, 10 and 5 are the actual arguments which become the values of the formal arguments inside the called function.

The **actual and formal arguments** should match in number, type, and order. The values of actual arguments are assigned to the formal arguments on a one to one basis; starting with the first argument.

While the formal arguments must be valid variable names, the actual arguments may be variable names, expressions, or constants. The variables used in actual arguments must be assigned values before the function call is made.

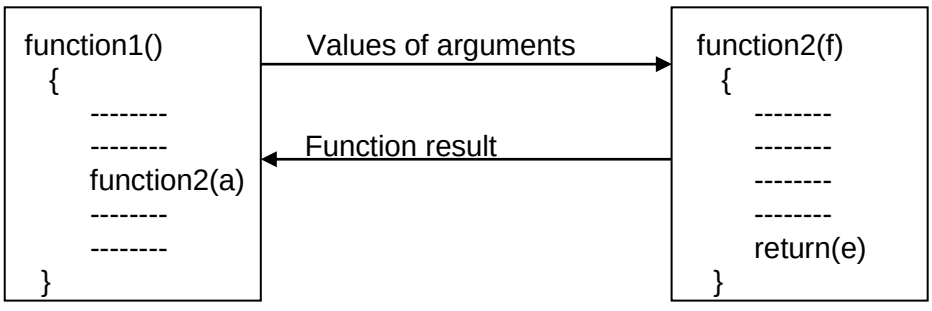
Remember that, when a function call is made, only a copy of the values of actual arguments is passed into the called function. What occurs inside the function will have no effect on the variables used in the actual argument list.

4. Functions with arguments and one return value

We may want to return result of a function to use it in calling function for further processing. A function should be generally written without ay Input Output operations.

A self-contained and independent function should behave like a ‘black box’ that receives a predefined form of input and outputs a desired value. Such functions will have two-way data communication as shown in fig.

Two-way data communication between functions



Example:

```
#include <stdio.h>
#include <conio.h>
float si (float p, float r, int n)
{
   float s;
   s=(p * r * n) / 100;
   return(s);
}
```

Cont.

```
void main()
{
   float p1,r1,sint;
   int n1;
   clrscr();
   printf("Enter p, r and n : ");
   scanf("%f %f %d",&p1,&r1,&n1);
   sint=si(p1, r1, n1);
   printf("Simple interest = %.2f",sint);
   getch();
}
```

Output:
Enter p, r and n : 10000 10 5
Simple interest = 5000.00

The arguments p, r, and n are called the formal arguments. The arguments p1, r1, and n1 are called the actual arguments.

The calling function sends values of actual arguments to formal arguments using function calls with actual arguments.

For e.g., the function call **si(p1, r1, n1)** will send the values 10000, 10 and 5 to the function **si(p, r, n)** and assign 10000 to p, 10 to r and 5 to n.

The values 10000, 10 and 5 are the actual arguments which become the values of the formal arguments inside the called function.

The called function **si** will be executed line by line in normal fashion until **return(s)**; statement. At this time, float value **s** is passed back to function call in the main. And following indirect assignment occurs at the place of function call:

si (p1, r1, n1) = s;

The calling statement will assign returned value to **sint**.

5. Functions with multiple return values (using pointer)

return statement can return only one value to calling function. But we can get more information from a user defined function. Using arguments we can receive information in function and also send back information to calling function.

The arguments that are used to "send out" information are called **output parameters**.

Address operator (&) and **indirection operator (*)** are used in mechanism of sending more information back. E.g.

```
void sum_diff (int x, int y, int  *s, int *d);
void main()
{
        int x = 20, y = 10, s, d;
        sum_diff (x, y, &s, &d);
}
void sum_diff (int a, int b, int *sum, int *diff)
{
        *sum = a + b;
        *diff = a – b;
}
```

Actual arguments x and y are input arguments, s and d are output arguments. In function call, we pass actual values of x and y but we pass addresses of locations where the values of s and d are stored in memory.

Indirection operator * in sum and diff indicates that these variable used to store addresses and not to store values of variables.

*sum = a + b; will add a and b and store the result in memory location pointed by sum.

Variables *sum and *diff are known as pointers and sum and diff are known as pointer variables. They are declared as int so they can point to locations of int type data.

The use of pointer variables as actual parameters for communicating data between functions is called **"pass by pointers"** or **"call by address"** or **"call by reference"**.

SCOPE AND LIFETIME OF VARIABLES IN FUNCTIONS

The **scope** of variable determines over what part(s) of the program a variable is actually available for use (active). **Longevity** refers to the period during which a variable retains a given value during execution of a program (alive). So longevity has a direct effect on the utility of a given variable.

The variables may also be broadly categorized, depending on the place of their declaration, as internal (local) or external (global). Internal variables are those which are declared within a particular function, while external variables are declared outside of any function.

COMMAND LINE ARGUMENTS

Command line argument is a parameter supplied to a program when the program is called up. This parameter may represent values which will be used by program for processing.

Suppose, we want to execute a program to concate two string. We may pass two strings from command line as:

C: combine hello hi where combine is a program name. Here, user need not require to enter strings during execution.

How to pass these parameters to program?

Every C program should have one main function. This main function indicates beginning of the program.
main function can also take arguments like other library and user defined functions.

main function can take two arguments called **argc** and **argv**. Information of command line is passed to program through these two arguments when main is called up.

argc counts the number of arguments on command line. argv is array of character pointers that point to the command line arguments. The size of array will be equal to value of argc.

For above example, argv[0] -> combine
argv[1] -> hello
argv[2] -> hi

We can use argv[0], argv[1] and so on, with puts or printf to print elements of argv array.

To access command line arguments, we must declare main function with parameters as follows:

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
void main(int argc,char *argv[])
{
  char temp[15];
  clrscr();
  strcpy(temp,argv[1]); // this will copy string hello to empty string temp
  strcat(temp,argv[2]); // this will append string hi at the end of string temp
  puts(temp);
}
```

Note: We need to create combine.exe file for program combine.

Use F9 shortcut key or use Make sub option of compile menu in TURBO C editor. After creating this file, go to command prompt and give command:

C: combine hello hi to get output of program.

Disclaimer

The study material is compiled by Ami D. Trivedi. The basic objective of this material is to supplement teaching and discussion in the classroom in the subject. Students are required to go for extra reading in the subject through library work.