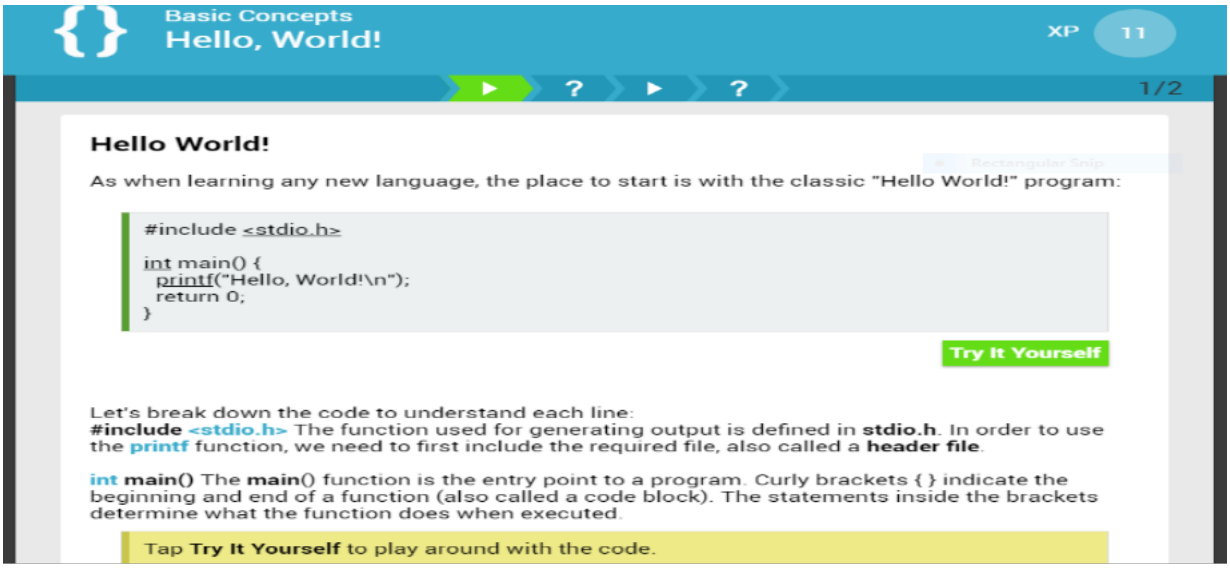
DAILY ASSESSMENT

| | | | |
|---|---|---|---|
| Date: | 18-06-2020 | Name: | Roshni A B |
| Course: | C Programming | USN: | 4AL17EC080 |
| Topic: | Module 1: Basic Concept<br>Module 2: Conditionals & Loops | Semester & Section: | 6TH SEM & 'B' Section |
| Github Repository: | Roshni-online | | |

SESSION DETAILS




Report

C

C is a general-purpose programming language that has been around for nearly 50 years. C has been used to write everything from operating systems (including Windows and many others) to complex programs like the Python interpreter, Git, Oracle database, and more. The versatility of C is by design. It is a low-level language that relates closely to the way machines work while still being easy to learn.

Hello World!

```
#include <stdio.h>
int main() {
printf("Hello, World!\n");
return 0;
}
```

Let's break down the code to understand each line:
#include <stdio.h> The function used for generating output is defined in stdio.h. In order to use the printf function, we need to first include the required file, also called a header file.

int main() The main() function is the entry point to a program. Curly brackets { } indicate the beginning and end of a function (also called a code block). The statements inside the brackets determine what the function does when executed.

The printf function is used to generate output:

```
#include <stdio.h>
int main() {
printf("Hello, World!\n");
return 0;
}
```

Here, we pass the text "Hello World!" to it.
The \n escape sequence outputs a newline character. Escape sequences always begin with a backslash \.
The semicolon ; indicates the end of the statement. Each statement must end with a semicolon.
return 0; This statement terminates the main() function and returns the value 0 to the calling process. The number 0 generally means that our program has successfully executed. Any other number indicates that the program has failed.

## Data Types

C supports the following basic data types:
int: integer, a whole number.
float: floating point, a number with a fractional part.
double: double-precision floating point value.
char: single character.
The amount of storage required for each of these types varies by platform.
C has a built-in sizeof operator that gives the memory requirements for a particular data type.
For example:

```
#include <stdio.h>
int main() {
printf("int: %ld \n", sizeof(int));
printf("float: %ld \n", sizeof(float));
```

```
printf("double: %ld \n", sizeof(double));
printf("char: %ld \n", sizeof(char));
return 0;
}
```

The program output displays the corresponding size in bytes for each data type.
The printf statements in this program have two arguments. The first is the output string with a format specifier (%ld), while the next argument returns the sizeof value. In the final output, the %ld (for long decimal) is replaced by the value in the second argument.

## Variables

A variable is a name for an area in memory.
The name of a variable (also called the identifier) must begin with either a letter or an underscore and can be composed of letters, digits, and the underscore character.
Variable naming conventions differ, however using lowercase letters with an underscore to separate words is common (snake_case).
Variables must also be declared as a data type before they are used.
The value for a declared variable is changed with an assignment statement.
For example, the following statements declare an integer variable my_var and then assigns it the value 42:

```
int my_var;
my_var = 42;
```

You can also declare and initialize (assign an initial value) a variable in a single statement:

```
int my_var = 42;
```

Let's define variables of different types, do a simple math operation, and output the results:

```
#include <stdio.h>
int main() {
int a, b;
float salary = 56.23;
char letter = 'Z';
a = 8;
b = 34;
int c = a+b;
printf("%d \n", c);
printf("%f \n", salary);
printf("%c \n", letter);
return 0;
}
```

As you can see, you can declare multiple variables on a single line by separating them with a comma. Also, notice the use of format specifiers for float (%f) and char (%c) output.

## Constants

A constant stores a value that cannot be changed from its initial assignment.

By using constants with meaningful names, code is easier to read and understand.
To distinguish constants from variables, a common practice is to use uppercase identifiers.
One way to define a constant is by using the const keyword in a variable declaration:

```
#include <stdio.h>
int main() {
const double PI = 3.14;
printf("%f", PI);
return 0;
}
```

The value of PI cannot be changed during program execution.
For example, another assignment statement, such as PI = 3.141 will generate an error.
Another way to define a constant is with the #define preprocessor directive.
The #define directive uses macros for defining constant values.
For example:

```
#include <stdio.h>
#define PI 3.14
int main() {
printf("%f", PI);
return 0;
}
```

Before compilation, the preprocessor replaces every macro identifier in the code with its corresponding value from the directive. In this case, every occurrence of PI is replaced with 3.14.
The final code sent to the compiler will already have the constant values in place.
The difference between const and #define is that the former uses memory for storage and the latter does not.

## Comments

Comments are explanatory information that you can include in a program to benefit the reader of your code. The compiler ignores comments, so they have no affect on a program.
A comment starts with a slash asterisk /* and ends with an asterisk slash */ and can be anywhere in your code.
Comments can be on the same line as a statement, or they can span several lines.
For example:

```
#include <stdio.h>
/* A simple C program
* Version 1.0
*/
int main() {
/* Output a string */
printf("Hello World!");
```

```
return 0;
}
```

Single-line Comments
C++ introduced a double slash comment // as a way to comment single lines. Some C compilers also support this comment style.
For example:

```
#include <stdio.h>
int main() {
int x = 42; //int for a whole number
//%d is replaced by x
printf("%d", x);
return 0;
}
```

## Arithmetic Operators

C supports arithmetic operators + (addition), - (subtraction), * (multiplication), / (division), and % (modulus division).
Operators are often used to form a numeric expression such as 10 + 5, which in this case contains two operands and the addition operator.
Numeric expressions are often used in assignment statements.
For example:

```
#include <stdio.h>
int main() {
int length = 10;
int width = 5;
int area;
area = length * width;
printf("%d \n", area); /* 50 */
return 0;
}
```

## Operator Precedence

C evaluates a numeric expression based on operator precedence.
The + and – are equal in precedence, as are *, /, and %.
The *, /, and % are performed first in order from left to right and then + and -, also in order from left to right.
You can change the order of operations by using parentheses ( ) to indicate which operations are to be performed first.
For example, the result of 5 + 3 * 2 is 11, where the result of (5 + 3) * 2 is 16.

For example:

```
#include <stdio.h>
int main() {
int a = 6;
int b = 4;
int c = 2;
int result;
result = a - b + c; // 4
printf("%d \n", result);
result = a + b / c; // 8
printf("%d \n", result);
result = (a + b) / c; // 5
printf("%d \n", result);
return 0;
}
```

## Assignment Operators

An assignment statement evaluates the expression on the right side of the equal sign first and then assigns that value to the variable on the left side of the =. This makes it possible to use the same variable on both sides of an assignment statement, which is frequently done in programming.

For example:`int x = 3;`
`x = x + 1; /* x is now 4 */`

To shorten this type of assignment statement, C offers the += assignment operator. The statement above can be written as`x += 1; /* x = x + 1 */`

Many C operators have a corresponding assignment operator. The program below demonstrates the arithmetic assignment operators:

```
int x = 2;
x += 1; // 3
x -= 1; // 2
x *= 3; // 6
x /= 2; // 3
x %= 2; // 1
x += 3 * 2; // 7
```

Look carefully at the last assignment statement. The entire expression on the right is evaluated and then added to x before being assigned to x. You can think of the statement as x = x + (3 * 2).

## Increment & Decrement

Adding 1 to a variable can be done with the increment operator ++. Similarly, the decrement operator -- is used to subtract 1 from a variable.

For example:`z--; /* decrement z by 1 */`

```
y++; /* increment y by 1 */
```
The increment and decrement operators can be used either prefix (before the variable name) or postfix (after the variable name). Which way you use the operator can be important in an assignment statement, as in the following example.

```
z = 3;
x = z--; /* assign 3 to x, then decrement z to 2 */
y = 3;
x = ++y; /* increment y to 4, then assign 4 to x */
```
The prefix form increments/decrements the variable and then uses it in the assignment statement.
The postfix form uses the value of the variable first, before incrementing/decrementing it.

## Conditionals

Conditionals are used to perform different computations or actions depending on whether a condition evaluates to true or false.

## The if Statement

The if statement is called a conditional control structure because it executes statements when an expression is true. For this reason, the if is also known as a decision structure. It takes the form:

```
if (expression)
statements
```
The expression evaluates to either true or false, and statements can be a single statement or a code block enclosed by curly braces { }.
For example:

```
#include <stdio.h>
int main() {
int score = 89;
if (score > 75)
printf("You passed.\n");
return 0;
}
```
In the code above we check whether the score variable is greater than 75, and print a message if the condition is true.

## The if-else Statement

The if statement can include an optional else clause that executes statements when an expression is false.
For example, the following program evaluates the expression and then executes the else clause statement:

```
#include <stdio.h>
int main() {
```

```
int score = 89;
if (score >= 90)
printf("Top 10%%.\n");
else
printf("Less than 90.\n");
return 0;
}
```

## Conditional Expressions

Another way to form an if-else statement is by using the ?: operator in a conditional expression. The ?: operator can have only one statement associated with the if and the else.
For example:

```
#include <stdio.h>
int main() {
int y;
int x = 3;
y = (x >= 5) ? 5 : x;
/* This is equivalent to:
if (x >= 5)
y = 5;
else
y = x;
*/
return 0;
}
```

## Nested if Statements

An if statement can include another if statement to form a nested statement. Nesting an if allows a decision to be based on further requirements.
Consider the following statement:

```
if (profit > 1000)
if (clients > 15)
bonus = 100;
else
bonus = 25;
```

Appropriately indenting nested statements will help clarify the meaning to a reader. However, be sure to understand that an else clause is associated with the closest if unless curly braces { } are used to change the association.
For example:

```
if (profit > 1000) {
if (clients > 15)
```

```
bonus = 100;
}
else
bonus = 25;
```

## The if-else if Statement

When a decision among three or more actions is needed, the if-else if statement can be used.
There can be multiple else if clauses and the last else clause is optional.
For example:

```
int score = 89;

if (score >= 90)
printf("%s", "Top 10%\n");
else if (score >= 80)
printf("%s", "Top 20%\n");
else if (score > 75)
printf("%s", "You passed.\n");
else
printf("%s", "You did not pass.\n");
```

## The switch Statement

The switch statement branches program control by matching the result of an expression with a constant case value.
The switch statement often provides a more elegant solution to if-else if and nested if statements.
The switch takes the form:switch (expression) {

```
case val1:
statements
break;
case val2:
statements
break;
default:
statements
}
```

For example, the following program outputs "Three":

```
int num = 3;
switch (num) {
case 1:
printf("One\n");
break;
```

```
case 2:
printf("Two\n");
break;
case 3:
printf("Three\n");
break;
default:
printf("Not 1, 2, or 3.\n");
}
```

## Logical Operators

Logical operators && and || are used to form a compound Boolean expression that tests multiple conditions. A third logical operator is ! used to reverse the state of a Boolean expression.

### The && Operator

The logical AND operator && returns a true result only when both expressions are true.

For example:

```
if (n > 0 && n <= 100)
printf("Range (1 - 100).\n");
```

## The || Operator

The logical OR operator || returns a true result when any one expression or both expressions are true.

For example:

```
if (n == 'x' || n == 'X')
printf("Roman numeral value 10.\n");
```

Any number of expressions can be joined by && and ||.

For example:

```
if (n == 999 || (n > 0 && n <= 100))
printf("Input valid.\n");
```

## The ! Operator

The logical NOT operator ! returns the reverse of its value.

NOT true returns false, and NOT false returns true.

For example:

```
if (!(n == 'x' || n == 'X'))
printf("Roman numeral is not 10.\n");
```

## The while Loop

The while statement is called a loop structure because it executes statements repeatedly while an expression is true, looping over and over again. It takes the form:while (expression)
{
statements
}
The expression evaluates to either true or false, and statements can be a single statement or, more commonly, a code block enclosed by curly braces { }.
For example:

```
#include <stdio.h>
int main() {
int count = 1;
while (count < 8) {
printf("Count = %d\n", count);
count++;
}
return 0;
}
```

## The do-while Loop

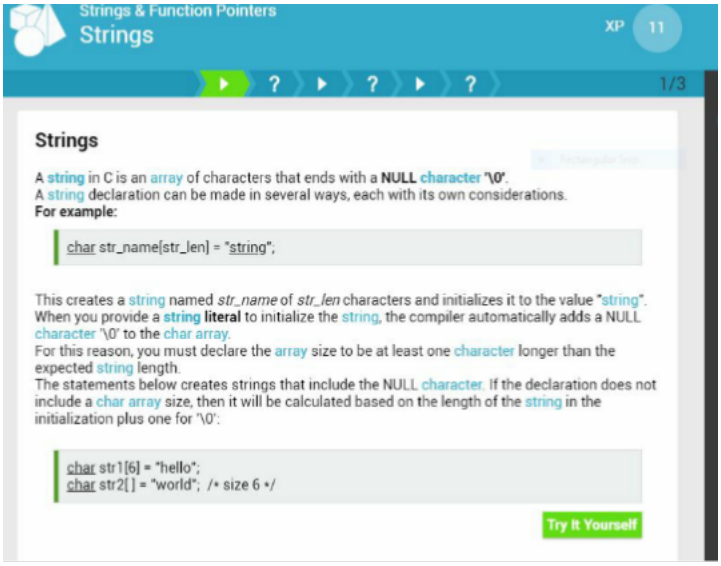
The do-while loop executes the loop statements before evaluating the expression to determine if the loop should be repeated.
It takes the form:do {
statements
} while (expression);
The expression evaluates to either true or false, and statements can be a single statement or a code block enclosed by curly braces { }.
For example:

```
#include <stdio.h>
int main() {
int count = 1;
do {
printf("Count = %d\n", count);
count++;
} while (count < 8);
return 0;
}
```

DAILY ASSESSMENT

| | | | |
|---|---|---|---|
| Date: | 18-06-2020 | Name: | Roshni A B |
| Course: | C Programming | USN: | 4AL17EC080 |
| Topic: | Module 3: Functions, Array & Pointers<br><br>Module 4: Strings & Function Pointers | Semester & Section: | 6TH SEM & 'B' Section |
| Github Repository: | Roshni-online | | |

SESSION DETAILS




Report

# Functions in C

Functions are central to C programming and are used to accomplish a program solution as a series of subtasks.

By now you know that every C program contains a main() function. And you're familiar with the printf() function.

You can also create your own functions.
A function:
• is a block of code that performs a specific task
• is reusable
• makes a program easier to test
• can be modified without changing the calling program

Even a simple program is easier to understand when main() is broken down into subtasks that are implemented with functions.
For example, it's clear that the goal of this program is to calculate the square of a number:

```
int main() {
int x, result;

x = 5;
result = square(x);
printf("%d squared is %d\n", x, result);

return 0;
}
```

In order to use the square function, we need to declare it.
Declarations usually appear above the main() function and take the form:

```
return_type function_name(parameters);
```

The return_type is the type of value the function sends back to the calling statement.
The function_name is followed by parentheses. Optional parameter names with type declarations are placed inside the parentheses.

## Function Parameters

A function's parameters are used to receive values required by the function. Values are passed to these parameters as arguments through the function call.
By default, arguments are passed by value, which means that a copy of data is given to the parameters of the called function. The actual variable isn't passed into the function, so it won't change.
Arguments passed to a function are matched to parameters by position. Therefore, the first argument is passed to the first parameter, the second to the second parameter, and so on.
The following program demonstrates parameters passed by value:

```
#include <stdio.h>
int sum_up (int x, int y);
```

```
int main() {
int x, y, result;
x = 3;
y = 12;
result = sum_up(x, y);
printf("%d + %d = %d", x, y, result);
return 0;
}
int sum_up (int x, int y) {
x += y;
return(x);
}
```

## Variable Scope

Variable scope refers to the visibility of variables within a program.
Variables declared in a function are local to that block of code and cannot be referred to outside the function.
Variables declared outside all functions are global to the entire program.
For example, constants declared with a #define at the top of a program are visible to the entire program.
The following program uses both local and global variables:

```
#include <stdio.h>
int global1 = 0;
int main() {
int local1, local2;

local1 = 5;
local2 = 10;
global1 = local1 + local2;
printf("%d \n", global1); /* 15 */
return 0;
}
```

## Recursive Functions

An algorithm for solving a problem may be best implemented using a process called recursion. Consider the factorial of a number, which is commonly written as 5! = 5 * 4 * 3 * 2 * 1.
This calculation can also be thought of as repeatedly calculating num * (num -1) until num is 1.
A recursive function is one that calls itself and includes a base case, or exit condition, for ending the recursive calls. In the case of computing a factorial, the base case is num equal to 1.

For example:

```
#include <stdio.h>

//function declaration
int factorial(int num);
int main() {
int x = 5;
printf("The factorial of %d is %d\n", x, factorial(x));
return 0;
}
//function definition
int factorial(int num) {

if (num == 1) /* base case */
return (1);
else
return (num * factorial(num - 1));
}
```

## Arrays in C

An array is a data structure that stores a collection of related values that are all the same type.
Arrays are useful because they can represent related data with one descriptive name rather than using separate variables that each must be named uniquely.
For example, the array test_scores[25] can hold 25 test scores.
An array declaration includes the type of the values it stores, an identifier, and square brackets [ ] with a number that indicates the array size.
For example:int test_scores[25]; /* An array size 25 */
You can also initialize an array when it is declared, as in the following statement:float prices[5] = {3.2, 6.55, 10.49, 1.25, 0.99};
Note that initial values are separated by commas and placed inside curly braces { }.
An array can be partially initialized, as in:float prices[5] = {3.2, 6.55};
Missing values are set to 0.

## What is a Pointer?

Pointers are very important in C programming because they allow you to easily work with memory locations.
They are fundamental to arrays, strings, and other data structures and algorithms.
A pointer is a variable that contains the address of another variable. In other words, it "points" to the location assigned to a variable and can indirectly access the variable.

Pointers are declared using the * symbol and take the form:pointer_type *identifier
pointer_type is the type of data the pointer will be pointing to. The actual pointer data type is a hexadecimal number, but when declaring a pointer, you must indicate what type of data it will be pointing to.
Asterisk * declares a pointer and should appear next to the identifier used for the pointer variable.
The following program demonstrates variables, pointers, and addresses:

```
int j = 63;
int *p = NULL;
p = &j;
printf("The address of j is %x\n", &j);
printf("p contains address %x\n", p);
printf("The value of j is %d\n", j);
printf("p is pointing to the value %d\n", *p);
```

## Strings

A string in C is an array of characters that ends with a NULL character '\0'.
A string declaration can be made in several ways, each with its own considerations.
For example:char str_name[str_len] = "string";
This creates a string named *str_name* of *str_len* characters and initializes it to the value "string".
When you provide a string literal to initialize the string, the compiler automatically adds a NULL character '\0' to the char array.
For this reason, you must declare the array size to be at least one character longer than the expected string length.
The statements below creates strings that include the NULL character. If the declaration does not include a char array size, then it will be calculated based on the length of the string in the initialization plus one for '\0':

```
char str1[6] = "hello";
char str2[ ] = "world"; /* size 6 */
```

## The sprintf and sscanf Functions

A formatted string can be created with the sprintf() function. This is useful for building a string from other data types.
For example:

```
#include <stdio.h>
int main()
{
char info[100];
char dept[ ] = "HR";
int emp = 75;
```

```
sprintf(info, "The %s dept has %d employees.", dept, emp);
printf("%s\n", info);
return 0;
}
```

## The string.h Library

The string.h library contains numerous string functions.
The statement #include <string.h> at the top of your program gives you access to the following:
strlen(str) Returns the length of the string stored in str, not including the NULL character.
strcat(str1, str2) Appends (concatenates) str2 to the end of str1 and returns a pointer to str1.
strcpy(str1, str2) Copies str2 to str1. This function is useful for assigning a string a new value.
The program below demonstrates string.h functions:

```
#include <stdio.h>
#include <string.h>
int main()
{
char s1[ ] = "The grey fox";
char s2[ ] = " jumped.";
strcat(s1, s2);
printf("%s\n", s1);
printf("Length of s1 is %d\n", strlen(s1));
strcpy(s1, s2);
printf("s1 is now %s \n", s1);
return 0;
}
```

## Converting a String to a Number

Converting a string of number characters to a numeric value is a common task in C programming and is often used to prevent a run-time error.
Reading a string is less error-prone than expecting a numeric value, only to have the user accidentally type an "o" rather than a "0" (zero).

## Function Pointers

Since pointers can point to an address in any memory location, they can also point to the start of executable code.
Pointers to functions, or function pointers, point to executable code for a function in memory. Function pointers can be stored in an array or passed as arguments to other functions.
A function pointer declaration uses the * just as you would with any pointer:return_type

(*func_name)(parameters)
The parentheses around (*func_name) are important. Without them, the compiler will think the function is returning a pointer.
After declaring the function pointer, you must assign it to a function. The following short program declares a function, declares a function pointer, assigns the function pointer to the function, and then calls the function through the pointer:

```
#include <stdio.h>
void say_hello(int num_times); /* function */
int main() {
void (*funptr)(int); /* function pointer */
funptr = say_hello; /* pointer assignment */
funptr(3); /* function call */
return 0;
}
void say_hello(int num_times) {
int k;
for (k = 0; k < num_times; k++)
printf("Hello\n");
}
```