|  |  |  |  |
| --- | --- | --- | --- |
| Date: | **19-06-2020** | Name: | **Varun G Shetty** |
| Course: | **C programming** | USN: | **4AL17EC093** |
| Topic: | * Loops * Functions in C | Semester & Section: | **6th & ‘B’** |
| GitHub Repository: | **Varunshetty4** |  |  |

**REPORT:**

**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**.** A compound Boolean expression is evaluated from**left to right.** Parentheses are used for clarity even though **&& has higher precedence than ||** and will be evaluated first.

**While loop**: The while statement is called a loop structure because it executes statements repeatedly while an expression is true, looping over and over**.**

|  |
| --- |
| while (expression) {  statements } |

The while loop evaluates a condition before the loop is entered, making it possible that the while statements never execute. An **infinite loop** is one that continues indefinitely because the loop condition never evaluates to false. This may cause a run-time error.

**do-while loop:** The do-while loop executes the loop statements before evaluating the expression to determine if the loop should be repeated.

|  |
| --- |
| while (expression) {  statements } |

The **do-while** loop always executes at least once, even if the expression evaluates to false.

**Break and continue**: The break statement was introduced for use in the switch statement. It is also useful for immediately exiting a loop. When you want to remain in the loop, but skip ahead to the next iteration, you use the **continue** statement.

**for loops:** The for statement is a loop structure that executes statements a fixed number of times.

|  |
| --- |
| for (initvalue; condition; increment) {  statements; } |

The **initvalue**is a counter set to an initial value. This part of the **for**loop is performed only once. The **condition**is a Boolean expression that compares the counter to a value before each loop iteration, stopping the loop when false is returned. The **increment**increases (or decreases) the counter by a set value. The **for**loop can **contain multiple expressions separated by commas** in each part.

Ex:for (x = 0, y = num; x < y; i++, y--)

Also, you can skip the **initvalue**, **condition**and/or **increment**.

for (; i < max; i++)

Loops can also be **nested**. When writing a program this way, there is an outer loop and an inner loop. For each iteration of the outer loop the inner loop repeats its entire cycle. A **break**in an **inner loop exits that loop** and execution **continues with the outer loop.  
A continue statement works similarly in nested loops.**

**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. A function is not required to return a value, but a return type must still be in the declaration. In this case, the keyword **void**is used.

When the parameter types and names are included in a declaration, the declaration is called a **function prototype**. For example, the **square**function prototype appears above main():

**int square (int num);**

Our square function **returns an integer** and **takes one parameter of type int**.  Function definitions usually appear after the **main()** function.

|  |
| --- |
| #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);**  **}** |

**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.

Note that in above program even though the value of parameter **x** was changed in **sum\_up**, the value of argument **x** in main() was not changed because only its value was passed into the parameter **x**.The parameters in a function declaration are the **formal parameters**. The values passed to these parameters are the arguments, sometimes called the **actual parameters**.

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.

**Static Variable**: **Static**variables have a **local scope but are not destroyed when a function is exited**. Therefore, a static variable retains its value for the life of the program and can be accessed every time the function is re-entered. A static variable is initialized when declared and requires the prefix **static.**

|  |  |
| --- | --- |
| #include <stdio.h>  void say\_hello();  int main() {  int i;  for (i = 0; i < 5; i++) {  say\_hello();  }  return 0;  }  void say\_hello() {  **static int num\_calls = 1;**  printf("Hello number %d\n", num\_calls);  num\_calls++;  } | >>  Hello number 1  Hello number 2  Hello number 3  Hello number 4  Hello number 5  If you declare as only int instead of static int it will output  Hello number 1  Hello number 1  Hello number 1  Hello number 1  Hello number 1 |

**Recursive Function:** An algorithm for solving a problem may be best implemented using a process called**recursion.** 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.

|  |
| --- |
| int factorial(int num) {  if (num == 1) /\* base case \*/  return (1);  else  return (num \* factorial(num - 1));  } |