****

**MOOC Based Seminar Report**

**On**

**[CS50]**

**Introduction To Computer Science**

BY : **Shreshth Tiwari**

ROLL NO: **1961159**

COURSE: **B.TECH**

BRANCH : **CSE**

UNDER THE GUIDANCE OF

**DR. MEHUL MANU**

**ASSISTANT PROFESSOR**

DURING THE YEAR **2019-2020**

****

**ACKNOWLEDGEMENT**

I would like to give a special thanks to several people who made this all possible.

It is because of their tremendous support that I was able to finish this project with immense success .

I would like to thank each and everyone who helped me in the completion of this project, but firstly and fore mostly I would like to pour my heart out in gratitude and humility towards our MOOC coordinator **Dr. Mehul Manu** who ostensibly supported me in this project whole heartedly.

I consider myself very fortunate and gratified that I was given an opportunity and once again would like to show my appreciation whole heartedly to everyone to make this possible.

shreshth.tiwari.24@gmail.com SHRESHTH TIWARI

****

**BHIMTAL CAMPUS**

**THIS IS TO CERTIFY THAT MR. Shreshth Tiwari HAS SATISFACTORILY PRESENTED MOOC BASED SEMINAR. THE COURSE OF THE MOOC CS 50 (Introduction To Computer Science) IN PARTIAL FULFILMENT OF THE SEMINAR PRESENTATION REQUIREMENT IN 2nd SEMESTER OF B.TECH (CSE) DEGREE COURSE PRESCRIBED BY GRAPHIC ERA HILL UNIVERSITY BHIMTAL CAMPUS DURING THE YEAR** 2019-2020**.**

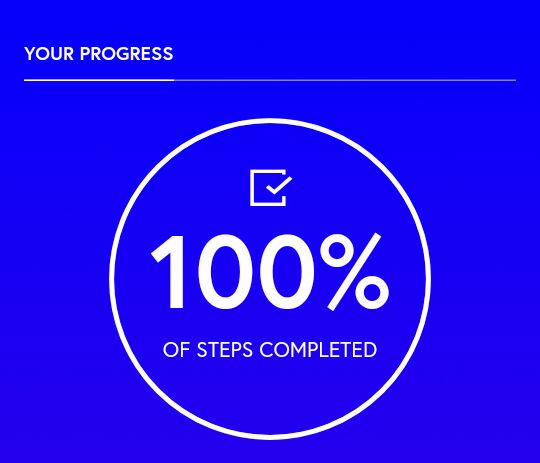
MOOC–Coordinator- HOD-

Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

****

**Copy Of Confirmation Of Course Completed Received**

****



**BHIMTAL CAMPUS**

**WEEK 1**

**CONTENTS**

* Data Types
* Operators
* Conditional Statements
* Loops

Data Types

In the *C* *programming* *language*, **data types** constitute the semantics and characteristics of storage of data elements. They are expressed in the language syntax in form of declarations for memory locations or **variables**. Data types also determine the types of operations or methods of processing of data elements.

There are following datatypes in C.

**int** –

* The **int** data type is used for variables that will store integer values. Eg- -3, -2, -1, 0, 1, 2, 3…
* Integers always take up **4 bytes** of memory (*32 bits*). This means the range of values they can store is necessarily limited to **32 bits** worth of information.
* Now as it turns out, it was long ago decided that we would split up that range of 32 bits into **negative** number and **positive** numbers, each getting half of the range.
* So, the range of values that we represent with an **integer** range from **-231** to (**231-1**) because we also need a spot for **0**.
* And roughly here, this is about **-2 billion** to about **2 billion**.
* So that’s what we can fit in an integer variable.

Now we also have something called **unsigned int**.

**unsigned int**-

* **unsigned int** is not a separate type of variable. Rather, it is what’s called a **qualifier**.
* It modifies the datatype of integer slightly.
* What it effectively does is doubles the **positive** range of the values, that an **integer** can take on at the expense of no longer allowing us to take on **negative** values.
* So, if we have numbers that we know will get higher than **2 billion** but less than **4 billion**, Eg- which is **232**. We might want to use an unsigned int if we know the value will never be **negative**.

**char**-

* **Char** is used for variables that will store single characters. It’s short for **characters**.
* So, characters always take up **1 byte** of memory, which is just **8 bits**.
* This means that they can only fit values in the range of **-27** to (**27-1**).
* Thanks to **ASCII**, it was long ago decided a way to map those **positive numbers** from **0** to **127** to various characters that all exist in our keyboard. Eg- **A** – **65** and **a** – **97**.
* The character **0** for when we actually type the character, not representing the number **0**, is **48**.

**float**-

* **float** is used to represent the **floating-point** numbers or **real** numbers.
* **floating-point** numbers are basically the numbers that have a **decimal point** in them.
* **floating-point** values like **integers** are also contained within **4 bytes** of memory.
* Suffice it to say that we have **32 bits** to work with. And if we have a number like **π**, which has an **integer** part **3**, and a **floating** part or **decimal** part **0.14159**, and so one, we need to be able to represent all of it-

The **integer** part and the **decimal** part.

* **So, what does it mean?**

One thing is that if the decimal part gets longer and longer, if I might not be able to be precise with the decimal part. And that’s really the limitation of a float.

* **Float** have a **precision problem**.

We only have **32 bits** to work with, so we can only be so precise with our **decimal** part. We can’t necessarily have a decimal precise to **100** to **200** digits because we only have **32 bits** to work with.

* Now fortunately there’s another **data type** called **double**.

**double**-

* **double**, like **float**, is used to store **real** numbers, or **floating-point** values.
* The difference is that **double** is **double precision**. They can fit **64 bits** of data, or **8 bytes**.
* **What does that mean?**

It means we can be a lot more precise with the decimal point. Instead of having **π** to the **7** places maybe with float, we can maybe have it to **30** places.

* If that’s important, we might want to use a **double** instead of a **float**. Basically, if we are working on anything where having a really long **decimal** place and a lot of **precision** is important.

**void**-

* **Void** is not a **data type** but it’s a **type**.
* So, **void** is a type. It does exist but it is not a **datatype**.
* We can’t create a variable of type **void** and assign a value to it.
* But **functions**, for example, can have a **void return type**.
* Basically, if we see a function that has a void return type, it means it doesn’t return a value.
* Eg- **printf** is one. It does not actually return anything to us. It prints something to the screen, and It’s basically a side effect of what printf does. But it doesn’t give us a value back.

**How to Work with Variables?**

To create a **variable**, all we need is to do **two** things:

* First, give it a **type**.
* Second, give it a **name**.
* Once you have done that ad slapped a **semicolon** (**;**) at the end of that line.

**You Have Create A variable.**

**Eg-**

*int number;*

*char letter*;

**What have we done here?**

We have created two variables. The first variable’s name is **number** and it is capable of holding **integer** type values, because its type is **int**.

**letter** is another variable that can hold **characters** because its data type is **char**.

**Here’s a small code-**

int number; **//declaration**

number = 17; **//assignment**

char letter; **//declaration**

letter = ’H’; **//assignment**

**It can also be done in one step-**

int number = 17; **//initialization**

char letter = ‘H’; **//initialization**

Operators

C language supports a rich set of built-in operators. An operator is a symbol that tells the compiler to perform a certain mathematical or logical manipulation. Operators are used in programs to manipulate data and variables.

C operators can be classified into following types:

* Arithmetic operators
* Relational operators
* Logical operators
* Bitwise operators
* Assignment operators
* Conditional operators
* Special operators

**Arithmetic** **Operators-**

C supports all the basic arithmetic operators. The following table shows all the basic arithmetic operators.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| **+** | adds two operands. |
| **-** | subtract second operands from first. |
| **\*** | multiply two operand. |
| **/** | divide numerator by denominator. |
| **%** | remainder of division. |
| **++** | Increment operator - increases integer value by one. |
| **--** | Decrement operator - decreases integer value by one. |

**Relational** **Operators-**

The following table shows all relation operators supported by C-

|  |  |
| --- | --- |
| **Operator** | **Description** |
| **==** | Check if two operand are equal. |
| **!=** | Check if two operand are not equal. |
| **>** | Check if operand on the left is greater than operand on the right. |
| **<** | Check operand on the left is smaller than right operand. |
| **>=** | check left operand is greater than or equal to right operand. |
| **<=** | Check if operand on left is smaller than or equal to right operand. |

**Logical** **Operators-**

C language supports following 3 logical operators. Suppose a = 1 and b = 0,

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| **&&** | Logical AND | *(a && b) is false* |
| **||** | Logical OR | *(a || b) is true* |
| **!** | Logical NOT | *(!a) is false* |

**Bitwise** **Operators-**

Bitwise operators perform manipulations of data at **bit level**. These operators also perform **shifting of bits** from right to left. Bitwise operators are not applied to **float** or **double**.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| **&** | Bitwise AND |
| **|** | Bitwise OR |
| **^** | Bitwise exclusive OR |
| **<<** | left shift |
| **>>** | right shift |

Now let’s see truth table for bitwise **&**, **|** and **^**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **a** | **b** | **a & b** | **a | b** | **a ^ b** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

**Assignment** **Operators-**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| **=** | assigns values from right side operands to left side operand | a=b |
| **+=** | adds right operand to the left operand and assign the result to left | a+=b is same as a=a+b |
| **-=** | subtracts right operand from the left operand and assign the result to left operand | a-=b is same as a=a-b |
| **\*=** | Multiply left operand with the right operand and assign the result to left operand | a\*=b is same as a=a\*b |
| **/=** | divides left operand with the right operand and assign the result to left operand | a/=b is same as a=a/b |

**Conditional** **Operator-**

The conditional operators in C language are known by two more names

1. **Ternary Operator**
2. **? : Operator**

It is actually the if condition that we use in C language decision making, but using conditional operator, we turn the if condition statement into a short and simple operator.

The syntax of a conditional operator is :

expression 1 ? expression 2: expression 3

**Explanation:**

* The question mark **"?"** in the syntax represents the **if** part.
* The first expression (expression 1) generally returns either true or false, based on which it is decided whether (expression 2) will be executed or (expression 3)
* If (expression 1) returns true then the expression on the left side of **" : "** i.e (expression 2) is executed.
* If (expression 1) returns false then the expression on the right side of **" : "** i.e (expression 3) is executed.

**Special** **Operators-**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| **sizeof** | Returns the size of an variable | **sizeof(x)** return size of the variable **x** |
| **&** | Returns the address of an variable | **&x ;** return address of the variable **x** |
| **\*** | Pointer to a variable | **\*x ;** will be pointer to a variable **x** |

Conditional Statements

In a 'C' program are executed sequentially. This happens when there is no condition around the statements. If you put some condition for a block of statements the flow of execution might change based on the result evaluated by the condition. This process is referred to as decision making in 'C.' The decision-making statements are also called as control statements.

In 'C' programming conditional statements are possible with the help of the following two constructs:

1. If statement

2. If-else statement

It is also called as branching as a program decides which statement to execute based on the result of the evaluated condition.

**ifStatement-**

It is one of the powerful conditional statement. If statement is responsible for modifying the flow of execution of a program. If statement is always used with a condition. The condition is evaluated first before executing any statement inside the body of If. The syntax for if statement is as follows:

**if (condition)**

**instruction;**

The condition evaluates to either true or false. True is always a non-zero value, and false is a value that contains zero. Instructions can be a single instruction or a code block enclosed by curly braces { }.

**Following program illustrates the use of if construct in 'C' programming:**

#include<stdio.h>

int main()

{

int num1=1;

int num2=2;

if(num1<num2) //test-condition

{

printf("num1 is smaller than num2");

}

return 0;

}

**Output:**

num1 is smaller than num2

1. In the above program, we have initialized two variables with num1, num2 with value as 1, 2 respectively.
2. Then, we have used if with a test-expression to check which number is the smallest and which number is the largest. We have used a relational expression in if construct. Since the value of num1 is smaller than num2, the condition will evaluate to true.
3. Thus it will print the statement inside the block of If. After that, the control will go outside of the block and program will be terminated with a successful result.

**Relational** **Operators-**

C has six relational operators that can be used to formulate a Boolean expression for making a decision and testing conditions, which returns **true** or **false**:

* **<** less than
* **<=** less than or equal to
* **>**  greater than
* **>=** greater than or equal to
* **==** equal to
* **!=** not equal to

Notice that the **equal test** (**==**) is different from the **assignment operator** (**=**) because it is one of the most common problems that a programmer faces by mixing them up.

**For example:**

int x = 41;

x =x+ 1;

if (x == 42)

{

printf("You succeed!");

}

**Output :**

You succeed

Keep in mind that a condition that evaluates to a non-zero value is considered as true.

**For example:**

int present = 1;

if (present)

printf("There is someone present in the classroom \n");

**Output :**

There is someone present in the classroom

**if*-*else Statement-**

The if-else is statement is an extended version of If. The general form of if-else is as follows:

**if (test-expression)**

**{**

**True block of statements**

**}**

**else**

**{**

**False block of statements**

**}**

**statements;**

In this type of a construct, if the value of test-expression is true, then the true block of statements will be executed. If the value of test-expression if false, then the false block of statements will be executed. In any case, after the execution, the control will be automatically transferred to the statements appearing outside the block of If.

**Following programs illustrate the use of the if-else construct:**

We will initialize a variable with some value and write a program to determine if the value is less than ten or greater than ten.

Let's start.

#include<stdio.h>

int main()

{

int num=19;

if(num<10)

{

printf("The value is less than 10");

}

else

{

printf("The value is greater than 10");

}

return 0;

}

**Output:**

The value is greater than 10

1. We have initialized a variable with value 19. We have to find out whether the number is bigger or smaller than 10 using a 'C' program. To do this, we have used the if-else construct.
2. Here we have provided a condition num<10 because we have to compare our value with 10.
3. As you can see the first block is always a true block which means, if the value of test-expression is true then the first block which is If, will be executed.
4. The second block is an else block. This block contains the statements which will be executed if the value of the test-expression becomes false. In our program, the value of num is greater than ten hence the test-condition becomes false and else block is executed. Thus, our output will be from an else block which is "The value is greater than 10". After the if-else, the program will terminate with a successful result.

In 'C' programming we can use multiple if-else constructs within each other which are referred to as nesting of if-else statements.

**Conditional Expression-**

There is another way to express an if-else statement is by introducing the **?:** operator. In a conditional expression the **?:** operator has only one statement associated with the if and the else.

**For example:**

#include <stdio.h>

int main()

{

int y;

int x = 2;

y = (x >= 6) ? 6 : x; // This is equivalent to: if (x >= 5) y = 5; else y = x;

printf("y =%d ",y);

return 0;

}

**Output :**

y =2

**Nested if*-*else Statement-**

When a series of decision is required, nested if-else is used. Nesting means using one if-else construct within another one.

Let's write a program to illustrate the use of nested if-else.

#include<stdio.h>

int main()

{

int num=1;

if(num<10)

{

if(num==1)

{

printf("The value is:%d\n",num);

}

else

{

printf("The value is greater than 1");

}

}

else

{

printf("The value is greater than 10");

}

return 0;

}

**Output:**

The value is:1

**The above program checks if a number is less or greater than 10 and prints the result using nested if-else construct.**

1. Firstly, we have declared a variable num with value as 1. Then we have used if-else construct.
2. In the outer if-else, the condition provided checks if a number is less than 10. If the condition is true then and only then it will execute the inner loop. In this case, the condition is true hence the inner block is processed.
3. In the inner block, we again have a condition that checks if our variable contains the value 1 or not. When a condition is true, then it will process the If block otherwise it will process an else block. In this case, the condition is true hence the If a block is executed and the value is printed on the output screen.
4. The above program will print the value of a variable and exit with success.

**NOTE:** In nested if-else, we have to be careful with the indentation because multiple if-else constructs are involved in this process, so it becomes difficult to figure out individual constructs. Proper indentation makes it easy to read the program.

**Nested else-if Statement-**

Nested else-if is used when multipath decisions are required.

The general syntax of how else-if ladders are constructed in 'C' programming is as follows:

**if (test - expression 1)**

**{**

**statement1;**

**}**

**else if (test - expression 2)**

**{**

**Statement2;**

**}**

**else if (test - expression 3)**

**{**

**Statement3;**

**}**

**else if (test - expression n)**

**{**

**Statement n;**

**}**

**else**

**{**

**default;**

**}**

**Statement x;**

This type of structure is known as the else-if ladder. This chain generally looks like a ladder hence it is also called as an else-if ladder. The test-expressions are evaluated from top to bottom. Whenever a true test-expression if found, statement associated with it is executed. When all the n test-expressions becomes false, then the default else statement is executed.

Let us see the actual working with the help of a program.

#include<stdio.h>

int main()

{

int marks=83;

if(marks>75)

{

printf("First class");

}

else if(marks>65)

{

printf("Second class");

}

else if(marks>55)

{

printf("Third class");

}

else

{

printf("Fourth class");

}

return 0;

}

**Output:**

First class

The above program prints the grade as per the marks scored in a test. We have used the else-if ladder construct in the above program.

1. We have initialized a variable with marks. In the else-if ladder structure, we have provided various conditions.
2. The value from the variable marks will be compared with the first condition since it is true the statement associated with it will be printed on the output screen.
3. If the first test condition turns out false, then it is compared with the second condition.
4. This process will go on until the all expression is evaluated otherwise control will go out of the else-if ladder, and default statement will be printed.

Loops

A **Loop** executes the sequence of statements many times until the stated condition becomes false. A loop consists of two parts, a body of a loop and a control statement. The control statement is a combination of some conditions that direct the body of the loop to execute until the specified condition becomes false. The purpose of the loop is to repeat the same code a number of times.

**Types of Loops**

Depending upon the position of a control statement in a program, a loop is classified into two types:

1. Entry controlled loop

2. Exit controlled loop

In an **entry controlled loop,** a condition is checked before executing the body of a loop. It is also called as a pre-checking loop.

In an **exit controlled loop**, a condition is checked after executing the body of a loop. It is also called as a post-checking loop.

The control conditions must be well defined and specified otherwise the loop will execute an infinite number of times. The loop that does not stop executing and processes the statements number of times is called as an **infinite loop**. An infinite loop is also called as an "**Endless loop**." Following are some characteristics of an infinite loop:

1. No termination condition is specified.

2. The specified conditions never meet.

The specified condition determines whether to execute the loop body or not.

'C' programming language provides us with three types of loop constructs:

1. The while loop

2. The do-while loop

3. The for loop

**while Loop-**

A while loop is the most straightforward looping structure. The basic format of while loop is as follows:

**while (condition)**

**{**

**statements;**

**}**

It is an entry-controlled loop. In while loop, a condition is evaluated before processing a body of the loop. If a condition is true then and only then the body of a loop is executed. After the body of a loop is executed then control again goes back at the beginning, and the condition is checked if it is true, the same process is executed until the condition becomes false. Once the condition becomes false, the control goes out of the loop.

After exiting the loop, the control goes to the statements which are immediately after the loop. The body of a loop can contain more than one statement. If it contains only one statement, then the curly braces are not compulsory. It is a good practice though to use the curly braces even we have a single statement in the body.

In while loop, if the condition is not true, then the body of a loop will not be executed, not even once. It is different in do while loop which we will see shortly.

**Following program illustrates a while loop:**

#include<stdio.h>

#include<conio.h>

int main()

{

int num=1; //initializing the variable

while(num<=10) //while loop with condition

{

printf("%d\n",num);

num++; //incrementing operation

}

return 0;

}

**Output:**

1

2

3

4

5

6

7

8

9

10

The above program illustrates the use of while loop. In the above program, we have printed series of numbers from 1 to 10 using a while loop.

1. We have initialized a variable called num with value 1. We are going to print from 1 to 10 hence the variable is initialized with value 1. If you want to print from 0, then assign the value 0 during initialization.
2. In a while loop, we have provided a condition (num<=10), which means the loop will execute the body until the value of num becomes 10. After that, the loop will be terminated, and control will fall outside the loop.
3. In the body of a loop, we have a print function to print our number and an increment operation to increment the value per execution of a loop. An initial value of num is 1, after the execution, it will become 2, and during the next execution, it will become 3. This process will continue until the value becomes 10 and then it will print the series on console and terminate the loop.

**\n** is used for formatting purposes which means the value will be printed on a new line.

**do-while Loop**

A do-while loop is similar to the while loop except that the condition is always executed after the body of a loop. It is also called an exit-controlled loop.

The basic format of while loop is as follows:

**do**

**{**

**statements**

**} while (expression);**

As we saw in a while loop, the body is executed if and only if the condition is true. In some cases, we have to execute a body of the loop at least once even if the condition is false. This type of operation can be achieved by using a do-while loop.

In the do-while loop, the body of a loop is always executed at least once. After the body is executed, then it checks the condition. If the condition is true, then it will again execute the body of a loop otherwise control is transferred out of the loop.

Similar to the while loop, once the control goes out of the loop the statements which are immediately after the loop is executed.

The critical difference between the while and do-while loop is that in while loop the while is written at the beginning. In do-while loop, the while condition is written at the end and terminates with a semi-colon (;)

**The following program illustrates the working of a do-while loop:**

//We are going to print a table of number 2 using do while loop.

#include<stdio.h>

#include<conio.h>

int main()

{

int num=1; //initializing the variable

do //do-while loop

{

printf("%d\n",2\*num);

num++; //incrementing operation

}while(num<=10);

return 0;

}

**Output:**

2

4

6

8

10

12

14

16

18

20

In the above example, we have printed multiplication table of 2 using a do-while loop. Let's see how the program was able to print the series.

1. First, we have initialized a variable 'num' with value 1. Then we have written a do-while loop.
2. In a loop, we have a print function that will print the series by multiplying the value of num with 2.
3. After each increment, the value of num will increase by 1, and it will be printed on the screen.
4. Initially, the value of num is 1. In a body of a loop, the print function will be executed in this way: 2\*num where num=1, then 2\*1=2 hence the value two will be printed. This will go on until the value of num becomes 10. After that loop will be terminated and a statement which is immediately after the loop will be executed. In this case return 0.

**For loop**

A for loop is a more efficient loop structure in 'C' programming. The general structure of for loop is as follows:

**for (initial value; condition; incrementation or decrementation )**

**{**

**statements;**

**}**

* The initial value of the for loop is performed only once.
* The condition is a Boolean expression that tests and compares the counter to a fixed value after each iteration, stopping the for loop when false is returned.
* The incrementation/decrementation increases (or decreases) the counter by a set value.

**Following program illustrates the use of a simple for loop:**

#include<stdio.h>

int main()

{

int number;

for(number=1;number<=10;number++) //for loop to print 1-10 numbers

{

printf("%d\n",number); //to print the number

}

return 0;

}

**Output:**

1

2

3

4

5

6

7

8

9

10

The above program prints the number series from 1-10 using for loop.

1. We have declared a variable of an int data type to store values.
2. In for loop, in the initialization part, we have assigned value 1 to the variable number. In the condition part, we have specified our condition and then the increment part.
3. In the body of a loop, we have a print function to print the numbers on a new line in the console. We have the value one stored in number, after the first iteration the value will be incremented, and it will become 2. Now the variable number has the value 2. The condition will be rechecked and since the condition is true loop will be executed, and it will print two on the screen. This loop will keep on executing until the value of the variable becomes 10. After that, the loop will be terminated, and a series of 1-10 will be printed on the screen.

In C, the for loop can have multiple expressions separated by commas in each part.

**For example:**

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

{

statements;

}

Also, we can skip the initial value expression, condition and/or increment by adding a semicolon.

**For example:**

int i=0;

int max = 10;

for (; i < max; i++)

{

printf("%d\n", i);

}

Notice that loops can also be nested where there is an outer loop and an inner loop. For each iteration of the outer loop, the inner loop repeats its entire cycle.

Consider the following example, that uses nested for loops output a multiplication table:

#include <stdio.h>

int main() {

int i, j;

int table = 2;

int max = 5;

for (i = 1; i <= table; i++) { // outer loop

for (j = 0; j <= max; j++) { // inner loop

printf("%d x %d = %d\n", i, j, i\*j);

}

printf("\n"); /\* blank line between tables \*/

}}

**Output:**

1 x 0 = 0

1 x 1 = 1

1 x 2 = 2

1 x 3 = 3

1 x 4 = 4

1 x 5 = 5

2 x 0 = 0

2 x 1 = 2

2 x 2 = 4

2 x 3 = 6

2 x 4 = 8

2 x 5 = 10

The nesting of for loops can be done up-to any level. The nested loops should be adequately indented to make code readable. In some versions of 'C,' the nesting is limited up to 15 loops, but some provide more.

The nested loops are mostly used in array applications which we will see in further tutorials.

**break Statement**

The break statement is used mainly in in the switch statement. It is also useful for immediately stopping a loop.

We consider the following program which introduces a break to exit a while loop:

#include <stdio.h>

int main()

{

int num = 5;

while (num > 0)

{

if (num == 3)

break;

printf("%d\n", num);

num--;

}

}

**Output:**

5

4

**continue Statement**

When you want to skip to the next iteration but remain in the loop, you should use the continue statement.

For example:

#include <stdio.h>

int main() {

int nb = 7;

while (nb > 0) {

nb--;

if (nb == 5)

continue;

printf("%d\n", nb);

}}

Output:

6

4

3

2

1

So, the value 5 is skipped.

**Which loop to Select?**

Selection of a loop is always a tough task for a programmer, to select a loop do the following steps:

* Analyze the problem and check whether it requires a pre-test or a post-test loop.
* If pre-test is required, use a while or for a loop.
* If post-test is required, use a do-while loop.

**Summary**

* Looping is one of the key concepts on any programming language.
* It executes a block of statements number of times until the condition becomes false.
* Loops are of 2 types: entry-controlled and exit-controlled.
* 'C' programming provides us 1) while 2) do-while and 3) for loop.
* For and while loop is entry-controlled loops.
* Do-while is an exit-controlled loop.



**WEEK 2**

**CONTENTS**

* Functions
* Arrays
* Command Line Arguments
* Scope Rules In C

Functions

A function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The C standard library provides numerous built-in functions that your program can call. For example, **strcat()** to concatenate two strings, **memcpy()** to copy one memory location to another location, and many more functions.

A function can also be referred as a method or a sub-routine or a procedure, etc.

**Defining a Function**

The general form of a function definition in C programming language is as follows −

return\_type function\_name( parameter list )

{

body of the function

}

A function definition in C programming consists of a *function header* and a *function body*. Here are all the parts of a function −

* **Return Type** − A function may return a value. The **return\_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return\_type is the keyword **void**.
* **Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
* **Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
* **Function Body** − The function body contains a collection of statements that define what the function does.

**Example**

Given below is the source code for a function called **max()**. This function takes two parameters num1 and num2 and returns the maximum value between the two −

/\* function returning the max between two numbers \*/

int max(int num1, int num2) {

/\* local variable declaration \*/

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

**Function Declarations**

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts −

return\_type function\_name( parameter list );

For the above defined function max(), the function declaration is as follows −

int max(int num1, int num2);

Parameter names are not important in function declaration only their type is required, so the following is also a valid declaration −

int max(int, int);

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

**Calling a Function**

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value. For example −

#include <stdio.h>

/\* function declaration \*/

int max(int num1, int num2);

int main () {

/\* local variable definition \*/

int a = 100;

int b = 200;

int ret;

/\* calling a function to get max value \*/

ret = max(a, b);

printf( "Max value is : %d\n", ret );

return 0;

}

/\* function returning the max between two numbers \*/

int max(int num1, int num2) {

/\* local variable declaration \*/

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

We have kept max() along with main() and compiled the source code. While running the final executable, it would produce the following result −

Max value is : 200

**Function Arguments**

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways in which arguments can be passed to a function −

|  |  |
| --- | --- |
|  | **Call Type & Description** |
|  | **Call By Value**  This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument. |
|  | **Call By Reference**  This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument. |

By default, C uses **call by value** to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

Arrays

Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

## Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows −

**type arrayName [ arraySize ];**

This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type double, use this statement −

double balance[10];

Here *balance* is a variable array which is sufficient to hold up to 10 double numbers.

## Initializing Arrays

You can initialize an array in C either one by one or using a single statement as follows −

double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ].

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −

double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array −

balance[4] = 50.0;

The above statement assigns the 5th element in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1. Shown below is the pictorial representation of the array we discussed above −

## Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example −

double salary = balance[9];

The above statement will take the 10th element from the array and assign the value to salary variable. The following example Shows how to use all the three above mentioned concepts viz. declaration, assignment, and accessing arrays −

[Live Demo](http://tpcg.io/dhfplr" \t "_blank)

#include <stdio.h>

int main () {

int n[ 10 ]; /\* n is an array of 10 integers \*/

int i,j;

/\* initialize elements of array n to 0 \*/

for ( i = 0; i < 10; i++ ) {

n[ i ] = i + 100; /\* set element at location i to i + 100 \*/

}

/\* output each array element's value \*/

for (j = 0; j < 10; j++ ) {

printf("Element[%d] = %d\n", j, n[j] );

}

return 0;

}

When the above code is compiled and executed, it produces the following result −

Element[0] = 100

Element[1] = 101

Element[2] = 102

Element[3] = 103

Element[4] = 104

Element[5] = 105

Element[6] = 106

Element[7] = 107

Element[8] = 108

Element[9] = 109

## Arrays in Detail

Arrays are important to C and should need a lot more attention. The following important concepts related to array should be clear to a C programmer −

|  |
| --- |
| **Multi Dimensional Array-**  C supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array. |
| **Passing Arrays To Functions-**  You can pass to the function a pointer to an array by specifying the array's name without an index. |
| **Return Array From Function-**  C allows a function to return an array. |
| **Pointer To An Array-**  You can generate a pointer to the first element of an array by simply specifying the array name, without any index. |

Command Line Arguments

Command line argument is a parameter supplied to the program when it is invoked. Command line argument is an important concept in C programming. It is mostly used when you need to control your program from outside. Command line arguments are passed to the main() method.

**Syntax:**

int main(int argc, char \*argv[])

Here argc counts the number of arguments on the command line and argv[ ] is a pointer array which holds pointers of type char which points to the arguments passed to the program.

### Example for Command Line Argument

#include <stdio.h>

#include <conio.h>

int main(int argc, char \*argv[])

{ int i;

if( argc >= 2 )

{ printf("The arguments supplied are:\n");

for(i = 1; i < argc; i++)

{ printf("%s\t", argv[i]);

}

}

else

{ printf("argument list is empty.\n");

}

return 0;

}

Remember that argv[0] holds the name of the program and argv[1] points to the first command line argument and argv[n] gives the last argument. If no argument is supplied, argc will be 1.

Scope Rules In C

Scope of an identifier is the part of the program where the identifier may directly be accessible. In C, all identifiers are **lexically(or statically) scoped**.

There are basically 4 scope rules:

|  |  |
| --- | --- |
| **Scope** | **Meaning** |
| **File Scope** | Scope of a Identifier starts at the beginning of the file and ends at the end of the file. It refers to only those Identifiers that are declared outside of all functions. The Identifiers of File scope are visible all over the file Identifiers having file scope are global |
| **Block Scope** | Scope of a Identifier begins at opening of the block / ‘{‘ and ends at the end of the block / ‘}’. Identifiers with block scope are local to their block |
| **Function Prototype Scope** | Identifiers declared in function prototype are visible within the prototype |
| **Function scope** | Function scope begins at the opening of the function and ends with the closing of it. Function scope is applicable to labels only. A label declared is used as a target to goto statement and both goto and label statement must be in same function |



**WEEK 3**

**CONTENTS**

* Linear Search
* Binary Search
* Bubble Sort
* Selection Sort
* Insertion Sort
* Recursion
* Merge Sort

Linear Search

**Linear search** in C to find whether a number is present in an array. If it's present, then at what location it occurs. It is also known as a sequential search. It is straightforward and works as follows: we compare each element with the element to search until we find it or the list ends.

**Example Programs-**

#include <stdio.h>

int main()  
{ int array[100], search, c, n;

printf("Enter number of elements in array**\n**");  
  scanf("%d", &n);

  printf("Enter %d integer(s)**\n**", n);

  for (c = 0; c < n; c++)  
    scanf("%d", &array[c]);

  printf("Enter a number to search**\n**");  
  scanf("%d", &search);

  for (c = 0; c < n; c++)  
  { if (array[c] == search)    */\* If required element is found \*/*  
    { printf("%d is present at location %d.**\n**", search, c+1);  
      break; }}  
  if (c == n)  
    printf("%d isn't present in the array.**\n**", search);

  return 0;  
}

**Output-**

Enter the number of elements in array

3

Enter 3 numbers

1

3

4

Enter the number to search

4

4 is present at location 3.

**Linear search for multiple occurrences.**

#include <stdio.h>  
   
int main()  
{ int array[100], search, c, n, count = 0;  
     
   printf("Enter number of elements in array\n");  
   scanf("%d", &n);  
     
   printf("Enter %d numbers\n", n);  
     
   for (c = 0; c < n; c++)  
      scanf("%d", &array[c]);  
       
   printf("Enter a number to search\n");  
   scanf("%d", &search);  
     
   for (c = 0; c < n; c++) {  
      if (array[c] == search) {  
         printf("%d is present at location %d.\n", search, c+1);  
         count++; }}  
   if (count == 0)  
      printf("%d isn't present in the array.\n", search);  
   else  
      printf("%d is present %d times in the array.\n", search, count);  
       
   return 0;  
}

**Output-**

Enter the number of elements in array

5

Enter 5 number

1

2

3

2

4

Enter the number to search

2

2 is present at location 2.

2 is present at location 4.

2 is present 2 times in the array.

**Linear search using a function.**

#include <stdio.h>  
   
long linear\_search(long [], long, long);  
   
int main()  
{ long array[100], search, c, n, position;  
   
   printf("Input number of elements in array**\n**");  
   scanf("%ld", &n);  
   
   printf("Input %d numbers**\n**", n);  
   
   for (c = 0; c < n; c++)  
      scanf("%ld", &array[c]);  
   
   printf("Input a number to search**\n**");  
   scanf("%ld", &search);  
   
   position = linear\_search(array, n, search);  
   
   if (position == -1)  
      printf("%d isn't present in the array.**\n**", search);  
   else  
      printf("%d is present at location %d.**\n**", search, position+1);  
   
   return 0; }   
   
long linear\_search(long a[], long n, long find) {  
   long c;  
   
   for (c = 0 ;c < n ; c++ ) {  
      if (a[c] == find)  
         return c; }  
   
   return -1; }

**Output-**

Enter the number of elements in array

2

Enter 2 numbers

1

2

Enter the number to search

1

1 is present at location 1.

**Linear search using pointer.**

#include <stdio.h>  
   
long linear\_search(long [], long, long);  
   
int main()  
{ long array[100], search, c, n, position;  
   
   printf("Input number of elements in array**\n**");  
   scanf("%ld", &n);  
   
   printf("Input %d numbers**\n**", n);  
   
   for (c = 0; c < n; c++)  
      scanf("%ld", &array[c]);  
   
   printf("Input a number to search**\n**");  
   scanf("%ld", &search);  
   
   position = linear\_search(array, n, search);  
   
   if (position == -1)  
      printf("%d isn't present in the array.**\n**", search);  
   else  
      printf("%d is present at location %d.**\n**", search, position+1);  
   
   return 0; }

long linear\_search(long \*p, long n, long find) {  
  long c;  
   
  for (c = 0; c < n; c++) {  
    if (\*(p+c) == find)  
      return c; }  
   
  return -1;}

**Output-**

Input number of elements in array

3

Input 3 numbers

1

2

3

Input a number to search

2

2 is present at location 2.

Binary Search

A binary search is a simplistic algorithm intended for finding the location of an item stored in a sorted list. There are a few variations to the binary search in C program, such as testing for equality and less-than at each step of the algorithm.

Binary search algorithm applies to a sorted array for searching an element. The search starts with comparing the target element with the middle element of the array. If value matches then the position of the element is returned.

In case the target element is less than the middle element (considering the array follows an ascending order) of the array then the second half of the array is discarded and the search continues by dividing the first half.

The process is the same when the target element is greater than the middle element, only, in this case, the first half of the array is discarded before continuing with the search. The iteration repeats until a match for the target element is found.

**Example Program-**

#include <stdio.h>

int main()

{

   int c, first, last, middle, n, search, array[100];

   printf("Enter number of elements:\n");

   scanf("%d",&n);

   printf("Enter %d integers:\n", n);

   for (c = 0; c < n; c++)

      scanf("%d",&array[c]);

   printf("Enter the value to find:\n");

   scanf("%d", &search);

   first = 0;

   last = n - 1;

   middle = (first+last)/2;

   while (first <= last) {

      if (array[middle] < search)

         first = middle + 1;

      else if (array[middle] == search) {

         printf("%d is present at index %d.\n", search, middle+1);

         break;

      }

      else

         last = middle - 1;

      middle = (first + last)/2;

   }

   if (first > last)

      printf("Not found! %d is not present in the list.\n", search);

   return 0;

}

**Output-**

Enter number of elements:

4

Enter 5 integers:

1  
9  
22  
24

Enter the value to find:

24

24 is present at index 4.

Bubble Sort

Bubble sort is also known as sinking sort. This algorithm compares each pair of adjacent items and swaps them if they are in the wrong order, and this same process goes on until no swaps are needed. In the following program we are implementing bubble sort in C language. In this program user would be asked to enter the number of elements along with the element values and then the program would sort them in ascending order by using bubble sorting algorithm logic.

**Example Program-**

#include<stdio.h>

int main(){

int count, temp, i, j, number[30];

printf("How many numbers are u going to enter?: ");

scanf("%d",&count);

printf("Enter %d numbers: ",count);

for(i=0;i<count;i++)

scanf("%d",&number[i]);

for(i=count-2;i>=0;i--){

for(j=0;j<=i;j++){

if(number[j]>number[j+1]){

temp=number[j];

number[j]=number[j+1];

number[j+1]=temp;} }}

printf("Sorted elements: ");

for(i=0;i<count;i++)

printf(" %d",number[i]);

return 0;}

Selection Sort

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

**Example Program-**

#include <stdio.h>

#include <stdbool.h>

#define MAX 7

int intArray[MAX] = {4,6,3,2,1,9,7};

void printline(int count) {

int i;

for(i = 0;i < count-1;i++) {

printf("=");}

printf("=\n");}

void display() {

int i;

printf("[");

for(i = 0;i < MAX;i++) {

printf("%d ", intArray[i]);}

printf("]\n");}

void selectionSort() {

int indexMin,i,j;

for(i = 0; i < MAX-1; i++) {

indexMin = i;

for(j = i+1;j < MAX;j++) {

if(intArray[j] < intArray[indexMin]) {

indexMin = j;}}

if(indexMin != i) {

printf("Items swapped: [ %d, %d ]\n" , intArray[i], intArray[indexMin]);

int temp = intArray[indexMin];

intArray[indexMin] = intArray[i];

intArray[i] = temp;}

printf("Iteration %d#:",(i+1));

display();}}

void main() {

printf("Input Array: ");

display();

printline(50); selectionSort();

printf("Output Array: ");display();

printline(50);}

**Output-**

Input Array: [4 6 3 2 1 9 7 ]

==================================================

Items swapped: [ 4, 1 ]

Iteration 1#:[1 6 3 2 4 9 7 ]

Items swapped: [ 6, 2 ]

Iteration 2#:[1 2 3 6 4 9 7 ]

Iteration 3#:[1 2 3 6 4 9 7 ]

Items swapped: [ 6, 4 ]

Iteration 4#:[1 2 3 4 6 9 7 ]

Iteration 5#:[1 2 3 4 6 9 7 ]

Items swapped: [ 9, 7 ]

Iteration 6#:[1 2 3 4 6 7 9 ]

Output Array: [1 2 3 4 6 7 9 ]

==================================================

Insertion Sort

Insertion Sort is a sorting algorithm where the array is sorted by taking one element at a time. The principle behind insertion sort is to take one element, iterate through the sorted array & find its correct position in the sorted array.

## Algorithm for Insertion Sort

* **Step 1** − If the element is the first one, it is already sorted.
* **Step 2** – Move to next element
* **Step 3** − Compare the current element with all elements in the sorted array
* **Step 4** – If the element in the sorted array is smaller than the current element, iterate to the next element. Otherwise, shift all the greater element in the array by one position towards the right
* **Step 5** − Insert the value at the correct position
* **Step 6** − Repeat until the complete list is sorted

**Example Program-**

void insertionSort(int arr[], int n)

{ int i, key, j;

    for (i = 1; i < n; i++) {

        key = arr[i];

        j = i - 1;

   while (j >= 0 && arr[j] > key) {

            arr[j + 1] = arr[j];

            j = j - 1;}

        arr[j + 1] = key; } }

void printArray(int arr[], int n)

{ int i;

    for (i = 0; i < n; i++)

        printf("%d ", arr[i]);

    printf("\n");}

int main()

{ int arr[] = { 12, 11, 13, 5, 6 };

    int n = sizeof(arr) / sizeof(arr[0]);

    insertionSort(arr, n);

    printArray(arr, n);

    return 0;}

**Output-**

5 6 11 12 13

Recursion

Recursion is a programming technique that allows the programmer to express operations in terms of themselves. In C, this takes the form of a function that calls itself. A useful way to think of recursive functions is to imagine them as a process being performed where one of the instructions is to "repeat the process".

**Example Program-**

void count\_to\_ten ( int count )

{

    if ( count < 10 )

       {

           count\_to\_ten( count + 1 );

       }

}

int main()

{

  count\_to\_ten ( 0 );

}

**Output-**

0

1

2

3

4

5

6

7

8

9

10

Merge Sort

**Merge Sort** is a **Divide and Conquer** algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

**Example Program-**

void merge(int arr[], int l, int m, int r)

{

    int i, j, k;

    int n1 = m - l + 1;

    int n2 =  r - m;

    /\* create temp arrays \*/

    int L[n1], R[n2];

    /\* Copy data to temp arrays L[] and R[] \*/

    for (i = 0; i < n1; i++)

        L[i] = arr[l + i];

    for (j = 0; j < n2; j++)

        R[j] = arr[m + 1+ j];

    /\* Merge the temp arrays back into arr[l..r]\*/

    i = 0; // Initial index of first subarray

    j = 0; // Initial index of second subarray

    k = l; // Initial index of merged subarray

    while (i < n1 && j < n2)

    {

        if (L[i] <= R[j])

        {

            arr[k] = L[i];

            i++;

        }

        else

        {

            arr[k] = R[j];

            j++;

        }

        k++;

    }

    /\* Copy the remaining elements of L[], if there

       are any \*/

    while (i < n1)

    {

        arr[k] = L[i];

        i++;

        k++;

    }

    /\* Copy the remaining elements of R[], if there

       are any \*/

    while (j < n2)

    {

        arr[k] = R[j];

        j++;

        k++;

    }

}

/\* l is for left index and r is right index of the

   sub-array of arr to be sorted \*/

void mergeSort(int arr[], int l, int r)

{

    if (l < r)

    {

        // Same as (l+r)/2, but avoids overflow for

        // large l and h

        int m = l+(r-l)/2;

        // Sort first and second halves

        mergeSort(arr, l, m);

        mergeSort(arr, m+1, r);

        merge(arr, l, m, r);

    }

}

/\* UTILITY FUNCTIONS \*/

/\* Function to print an array \*/

void printArray(int A[], int size)

{

    int i;

    for (i=0; i < size; i++)

        printf("%d ", A[i]);

    printf("\n");

}

/\* Driver program to test above functions \*/

int main()

{

    int arr[] = {12, 11, 13, 5, 6, 7};

    int arr\_size = sizeof(arr)/sizeof(arr[0]);

    printf("Given array is \n");

    printArray(arr, arr\_size);

    mergeSort(arr, 0, arr\_size - 1);

    printf("\nSorted array is \n");

    printArray(arr, arr\_size);

    return 0;

}

**Output-**

Given array is

12 11 13 5 6 7

Sorted array is

5 6 7 11 12 13



**WEEK 4**

**CONTENTS**

* Hexadecimal
* Pointers
* Dynamic Memory Allocation
* Call Stacks
* File Pointers

Hexadecimal

**Hexadecimal value** has **16** alphanumeric values from **0** to **9** and **A** to **F**, with the base **16**.

## Representation of Hexadecimal numbers in C programming

In C programming language, a Hexadecimal number is represented by preceding with **"0x"** or **"0X"**, thus the value in Hexadecimal can be written as **"0x64"** (which is equivalent to **100 in Decimal**).

### **Assigning the Hexadecimal number in a variable**

### There is no special type of data type to store Hexadecimal values in C programming, Hexadecimal number is an **integer value** and you can store it in the integral type of data types (char, short or int).

Let suppose, we have two values in Hexadecimal **"64"** (100 in Decimal) and **"FAFA"** (64250 in Decimal).

We are storing **"64"** in an **unsigned char** variable (64 is small value and can be stored with in a Byte) and **"FAFA"** in the int variable.

**Example Format-**

unsigned char a=0x64;

unsigned char b=0xFAFA;

### **Printing the number in Hexadecimal format**

To print integer number in Hexadecimal format, "**%x**" or "**%X**" is used as format specifier in **printf()** statement.

"**%x**" prints the value in **Hexadecimal format** with alphabets in **lowercase** (**a**-**f**).

"**%X**" prints the value in **Hexadecimal format** with alphabets in **uppercase** (**A**-**F**).

#include <stdio.h>

int main()

{

unsigned char a;

int b;

printf("Enter value of a: ");

scanf("%x",&a);

printf("Enter value of b: ");

scanf("%x",&b);

printf("Value of a: Hex: %X, Decimal: %d\n",a,a);

printf("Value of b: Hex: %X, Decimal: %d\n",b,b);

return 0;

}

**Output**

Enter value of a: 64

Enter value of b: FAFA

Value of a: Hex: 64, Decimal: 100

Value of b: Hex: FAFA, Decimal: 64250

Pointers

* Pointers in C language is a variable that stores/points the address of another variable. A Pointer in C is used to allocate memory dynamically i.e. at run time. The pointer variable might be belonging to any of the data type such as int, float, char, double, short etc.
* Pointer Syntax : **data\_type \*var\_name;** Example : **int \*p;  char \*p;**
* Where, **\*** is used to denote that **“p”** is pointer variable and not a normal variable.

# **Key points to remember about pointers in C:**

* Normal variable stores the value whereas pointer variable stores the address of the variable.
* The content of the C pointer always be a whole number i.e. address.
* Always C pointer is initialized to null, i.e. int \*p = null.
* The value of null pointer is 0.
* & symbol is used to get the address of the variable.
* \* symbol is used to get the value of the variable that the pointer is pointing to.
* If a pointer in C is assigned to NULL, it means it is pointing to nothing.
* Two pointers can be subtracted to know how many elements are available between these two pointers.
* But, Pointer addition, multiplication, division are not allowed.
* The size of any pointer is 2 byte (for 16 bit compiler).

**Example-**

#include <stdio.h>

int main()

{

   int \*ptr, q;

   q = 50;

   /\* address of q is assigned to ptr \*/

   ptr = &q;

   /\* display q's value using ptr variable \*/

   printf("%d", \*ptr);

   return 0;

}

**Output-**

50

Dynamic Memory Allocation

As we know, an array is a collection of a fixed number of values. Once the size of an array is declared, we cannot change it.

Sometimes the size of the array we declared may be insufficient. To solve this issue, we can allocate memory manually during run-time. This is known as **dynamic memory allocation** in C programming.

To allocate memory dynamically, library functions are **malloc()**, **calloc()**, **realloc()** and **free()** are used. These functions are defined in the **<stdlib.h>** header file.

## malloc()

The name "malloc" stands for memory allocation.

The malloc() function reserves a block of memory of the specified number of bytes. And, it returns a [pointer](https://www.programiz.com/c-programming/c-pointers" \o "C Pointers) of void which can be casted into pointers of any form.

### **Syntax of malloc()**

ptr = (castType\*) malloc(size);

**Example**

ptr = (float\*) malloc(100 \* sizeof(float));

The above statement allocates 400 bytes of memory. It's because the size of float is 4 bytes. And, the pointer ptr holds the address of the first byte in the allocated memory.

The expression results in a NULL pointer if the memory cannot be allocated.

## calloc()

The name "calloc" stands for contiguous allocation.

The malloc() function allocates memory and leaves the memory uninitialized. Whereas, the calloc() function allocates memory and initializes all bits to zero.

### **Syntax of calloc()**

ptr = (castType\*)calloc(n, size);

**Example:**

ptr = (float\*) calloc(25, sizeof(float));

The above statement allocates contiguous space in memory for 25 elements of type float.

## C free()

Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on their own. You must explicitly use free() to release the space.

**Syntax of free()**

free(ptr);

This statement frees the space allocated in the memory pointed by ptr.

**Example-**

#include <stdio.h>

#include <stdlib.h>

int main()

{

int n, i, \*ptr, sum = 0;

printf("Enter number of elements: ");

scanf("%d", &n);

ptr = (int\*) malloc(n \* sizeof(int));

// if memory cannot be allocated

if(ptr == NULL)

{

printf("Error! memory not allocated.");

exit(0);

}

printf("Enter elements: ");

for(i = 0; i < n; ++i)

{

scanf("%d", ptr + i);

sum += \*(ptr + i);

}

printf("Sum = %d", sum);

// deallocating the memory

free(ptr);

return 0;

}

## realloc()

If the dynamically allocated memory is insufficient or more than required, you can change the size of previously allocated memory using the realloc() function.

**Syntax of realloc()**

ptr = realloc(ptr, x);

Here, ptr is reallocated with a new size x.

**Example-**

#include <stdio.h>

#include <stdlib.h>

int main()

{

int \*ptr, i , n1, n2;

printf("Enter size: ");

scanf("%d", &n1);

ptr = (int\*) malloc(n1 \* sizeof(int));

printf("Addresses of previously allocated memory: ");

for(i = 0; i < n1; ++i)

printf("%u\n",ptr + i);

printf("\nEnter the new size: ");

scanf("%d", &n2);

// rellocating the memory

ptr = realloc(ptr, n2 \* sizeof(int));

printf("Addresses of newly allocated memory: ");

for(i = 0; i < n2; ++i)

printf("%u\n", ptr + i);

free(ptr);

return 0;

}

**Output-**

Enter size: 2

Addresses of previously allocated memory:

26855472

26855476

Enter the new size: 4

Addresses of newly allocated memory:

26855472

26855476

26855480

26855484

Call Stacks

a **call stack** is a **stack data structure** that stores information about the active **subroutines** of a **computer program**. This kind of stack is also known as an **execution stack**, **program stack**, **control stack**, **run-time stack**, or **machine stack**, and is often shortened to just "**the stack**". Although maintenance of the call stack is important for the proper functioning of most **software**, the details are normally hidden and automatic in **high level programming languages**. Many computer **instructions sets** provide special instructions for manipulating stacks.

A call stack is used for several related purposes, but the main reason for having one is to keep track of the point to which each active subroutine should return control when it finishes executing. An active subroutine is one that has been called, but is yet to complete execution, after which control should be handed back to the point of call. Such activations of subroutines may be nested to any level (recursive as a special case), hence the stack structure. For example, if a subroutine DrawSquare calls a subroutine DrawLine from four different places, DrawLine must know where to return when its execution completes. To accomplish this, the **address** following the **instruction** that jumps to DrawLine, the **return address**, is pushed onto the top of the call stack with each call.

**Example-**

#include <limits.h>

#include <stdio.h>

#include <stdlib.h>

struct Stack {

    int top;

    unsigned capacity;

    int\* array;

};

struct Stack\* createStack(unsigned capacity)

{

    struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));

    stack->capacity = capacity;

    stack->top = -1;

    stack->array = (int\*)malloc(stack->capacity \* sizeof(int));

    return stack;

}

int isFull(struct Stack\* stack)

{

    return stack->top == stack->capacity - 1;

}

int isEmpty(struct Stack\* stack)

{

    return stack->top == -1;

}

void push(struct Stack\* stack, int item)

{

    if (isFull(stack))

        return;

    stack->array[++stack->top] = item;

    printf("%d pushed to stack\n", item);

}

int pop(struct Stack\* stack)

{

    if (isEmpty(stack))

        return INT\_MIN;

    return stack->array[stack->top--];

}

int peek(struct Stack\* stack)

{

    if (isEmpty(stack))

        return INT\_MIN;

    return stack->array[stack->top];

}

int main()

{

    struct Stack\* stack = createStack(100);

    push(stack, 10);

    push(stack, 20);

    push(stack, 30);

    printf("%d popped from stack\n", pop(stack));

    return 0;

}

File Pointers

* File pointer is a pointer which is used to handle and keep track on the files being accessed. A new data type called “FILE” is used to declare file pointer. This data type is defined in stdio.h file. File pointer is declared as FILE \*fp. Where, ‘fp’ is a file pointer.
* fopen() function is used to open a file that returns a FILE pointer. Once file is opened, file pointer can be used to perform I/O operations on the file. fclose() function is used to close the file.

Different operations that can be performed on a file are:

1. Creation of a new file (**fopen with attributes as “a” or “a+” or “w” or “w++”)**
2. Opening an existing file (**fopen**)
3. Reading from file (**fscanf or fgets**)
4. Writing to a file (**fprintf or fputs**)
5. Moving to a specific location in a file (**fseek, rewind**)
6. Closing a file (**fclose**)

**Opening or creating file**  
For opening a file, fopen function is used with the required access modes. Some of the commonly used file access modes are mentioned below.

**File opening modes in C:**

1. **“r” –** Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer which points to the first character in it. If the file cannot be opened fopen( ) returns NULL.
2. **“w” –** Searches file. If the file exists, its contents are overwritten. If the file doesn’t exist, a new file is created. Returns NULL, if unable to open file.
3. **“a” –** Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the last character in it. If the file doesn’t exist, a new file is created. Returns NULL, if unable to open file.
4. **“r+” –** Searches file. If is opened successfully fopen( ) loads it into memory and sets up a pointer which points to the first character in it. Returns NULL, if unable to open the file.
5. **“w+” –** Searches file. If the file exists, its contents are overwritten. If the file doesn’t exist a new file is created. Returns NULL, if unable to open file.
6. **“a+” –** Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer which points to the last character in it. If the file doesn’t exist, a new file is created. Returns NULL, if unable to open file.

**Reading from a file –**  
The file read operations can be performed using functions fscanf or fgets. Both the functions performed the same operations as that of scanf and gets but with an additional parameter, the file pointer. So, it depends on you if you want to read the file line by line or character by character.

And the code snippet for reading a file is as:

FILE \* filePointer;

filePointer = fopen(“fileName.txt”, “r”);

fscanf(filePointer, "%s %s %s %d", str1, str2, str3, &year);

**Writing a file –**:

The file write operations can be perfomed by the functions fprintf and fputs with similarities to read operations. The snippet for writing to a file is as :

FILE \*filePointer ;

filePointer = fopen(“fileName.txt”, “w”);

fprintf(filePointer, "%s %s %s %d", "We", "are", "in", 2012);

**Closing a file –**:  
After every successful fie operations, you must always close a file. For closing a file, you have to use fclose function. The snippet for closing a file is given as :

FILE \*filePointer ;

filePointer= fopen(“fileName.txt”, “w”);

---------- Some file Operations -------

fclose(filePointer)

**Example-**

int main( )

{

    FILE \*filePointer ;

    char dataToBeWritten[50]  = "GeeksforGeeks-A Computer Science Portal for Geeks";

    filePointer = fopen("GfgTest.c", "w") ;

    if ( filePointer == NULL )

    {

        printf( "GfgTest.c file failed to open." ) ;

    }

    else

    {

        printf("The file is now opened.\n") ;

        if ( strlen (  dataToBeWritten  ) > 0 )

        {

            fputs(dataToBeWritten, filePointer) ;

            fputs("\n", filePointer) ;

        }

        fclose(filePointer) ;

        printf("Data successfully written in file GfgTest.c\n");

        printf("The file is now closed.") ;

    }

    return 0;

}



**WEEK 5**

**CONTENTS**

* Data Structures
* Singly-Linked Lists
* Hash Tables
* Tries

Data Structures

A **data structure** is a particular way of organizing data in a computer so that it can be used effectively.

Data structures in C are an inevitable part of programs. Computer programs frequently process data, so we require efficient ways in which we can access or manipulate data. Some applications may require modification of data frequently, and in others, new data is continuously added or deleted. So we need efficient ways of accessing data to act on it and build efficient applications.

For example, we can store a list of items having the same data-type using the *array* data structure.

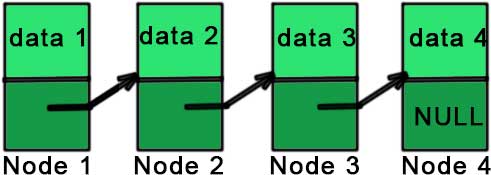


Singly Linked lists

Linked list is one of the most important data structures. We often face situations, where the data is dynamic in nature and number of data can’t be predicted or the number of data keeps changing during program execution. Linked lists are very useful in this type of situations.

The implementation of a linked list in C is done using pointers.

A linked list is made up of many nodes which are connected in nature. Every node is mainly divided into two parts, one part holds the data and the other part is connected to a different node. It is similar to the picture given below.



**Example-**

#include <stdio.h>

#include <stdlib.h>

struct node

{

int data;

struct node \*next;

};

int main()

{

struct node \*prev,\*head,\*p;

int n,i;

printf ("number of elements:");

scanf("%d",&n);

head=NULL;

for(i=0;i<n;i++)

{

p=malloc(sizeof(struct node));

scanf("%d",&p->data);

p->next=NULL;

if(head==NULL)

head=p;

else

prev->next=p;

prev=p;

}

return 0;

}

Hash Tables

Hash Table is a data structure which stores data in an associative manner. In hash table, the data is stored in an array format where each data value has its own unique index value. Access of data becomes very fast, if we know the index of the desired data

**Example-**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <stdbool.h>

#define SIZE 20

struct DataItem {

int data;

int key;};

struct DataItem\* hashArray[SIZE];

struct DataItem\* dummyItem;

struct DataItem\* item;

int hashCode(int key) {

return key % SIZE;}

struct DataItem \*search(int key) {

int hashIndex = hashCode(key);

while(hashArray[hashIndex] != NULL) {

if(hashArray[hashIndex]->key == key)

return hashArray[hashIndex];

++hashIndex;

hashIndex %= SIZE }

return NULL; }

void insert(int key,int data) {

struct DataItem \*item = (struct DataItem\*) malloc(sizeof(struct DataItem));

item->data = data;

item->key = key;

int hashIndex = hashCode(key);

while(hashArray[hashIndex] != NULL && hashArray[hashIndex]->key != -1) {

++hashIndex;

hashIndex %= SIZE }

hashArray[hashIndex] = item;}

struct DataItem\* delete(struct DataItem\* item) {

int key = item->key;

int hashIndex = hashCode(key);

while(hashArray[hashIndex] != NULL) {

if(hashArray[hashIndex]->key == key) {

struct DataItem\* temp = hashArray[hashIndex];

hashArray[hashIndex] = dummyItem;

return temp; }

++hashIndex;

hashIndex %= SIZE; }

return NULL; }

void display() {

int i = 0;

for(i = 0; i<SIZE; i++) {

if(hashArray[i] != NULL)

printf(" (%d,%d)",hashArray[i]->key,hashArray[i]->data);

else

printf(" ~~ "); }

printf("\n");}

int main() {

dummyItem = (struct DataItem\*) malloc(sizeof(struct DataItem));

dummyItem->data = -1;

dummyItem->key = -1;

insert(1, 20);

insert(2, 70);

insert(42, 80);

insert(4, 25);

insert(12, 44);

insert(14, 32);

insert(17, 11);

insert(13, 78);

insert(37, 97);

display();

item = search(37);

if(item != NULL) {

printf("Element found: %d\n", item->data); }

else {

printf("Element not found\n");, }

delete(item);

item = search(37);

if(item != NULL) {

printf("Element found: %d\n", item->data); }

else {

printf("Element not found\n"); }}

## Output-

~~ (1,20) (2,70) (42,80) (4,25) ~~ ~~ ~~ ~~ ~~ ~~ ~~ (12,44) (13,78) (14,32) ~~ ~~ (17,11) (37,97) ~~

Element found: 97

Element not found

Tries

Trie is an efficient information re**Trie**val data structure. Using Trie, search complexities can be brought to optimal limit (key length). If we store keys in binary search tree, a well balanced BST will need time proportional to **M \* log N**, where M is maximum string length and N is number of keys in tree. Using Trie, we can search the key in O(M) time. However the penalty is on Trie storage requirements.

Every node of Trie consists of multiple branches. Each branch represents a possible character of keys. We need to mark the last node of every key as end of word node. A Trie node field isEndOfWord is used to distinguish the node as end of word node.

**Example-**

struct TrieNode  
{  
     struct TrieNode \*children[ALPHABET\_SIZE];

     // isEndOfWord is true if the node  
     // represents end of a word  
     bool isEndOfWord;  
};

Inserting a key into Trie is a simple approach. Every character of the input key is inserted as an individual Trie node. Note that the children is an array of pointers (or references) to next level trie nodes. The key character acts as an index into the array children. If the input key is new or an extension of the existing key, we need to construct non-existing nodes of the key, and mark end of the word for the last node. If the input key is a prefix of the existing key in Trie, we simply mark the last node of the key as the end of a word. The key length determines Trie depth.

earching for a key is similar to insert operation, however, we only compare the characters and move down. The search can terminate due to the end of a string or lack of key in the trie. In the former case, if the *isEndofWord* field of the last node is true, then the key exists in the trie. In the second case, the search terminates without examining all the characters of the key, since the key is not present in the trie.

**Example-**

root

/ \ \

t a b

| | |

h n y

| | \ |

e s y e

/ | |

i r w

| | |

r e e

|

r

**Example Program-**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

#define ARRAY\_SIZE(a) sizeof(a)/sizeof(a[0])

#define ALPHABET\_SIZE (26)

#define CHAR\_TO\_INDEX(c) ((int)c - (int)'a')

struct TrieNode

{

    struct TrieNode \*children[ALPHABET\_SIZE];

    bool isEndOfWord;

};

 struct TrieNode \*getNode(void)

{   struct TrieNode \*pNode = NULL;

     pNode = (struct TrieNode \*)malloc(sizeof(struct TrieNode));

     if (pNode)

    {        int i;

         pNode->isEndOfWord = false;

         for (i = 0; i < ALPHABET\_SIZE; i++)

            pNode->children[i] = NULL;  }

    return pNode; }

void insert(struct TrieNode \*root, const char \*key)

{   int level;

    int length = strlen(key);

    int index;

     struct TrieNode \*pCrawl = root;

   for (level = 0; level < length; level++)

    {  index = CHAR\_TO\_INDEX(key[level]);

        if (!pCrawl->children[index])

          pCrawl->children[index] = getNode();

        pCrawl = pCrawl->children[index];  }

     pCrawl->isEndOfWord = true; }

bool search(struct TrieNode \*root, const char \*key)

{   int level;

    int length = strlen(key);

    int index;

    struct TrieNode \*pCrawl = root;

    for (level = 0; level < length; level++)

    {index = CHAR\_TO\_INDEX(key[level]);

        if (!pCrawl->children[index])

            return false;

        pCrawl = pCrawl->children[index]; }

    return (pCrawl != NULL && pCrawl->isEndOfWord); }

int main()

{  char keys[][8] = {"the", "a", "there", "answer", "any",

                     "by", "bye", "their"};

    char output[][32] = {"Not present in trie", "Present in trie"};

    struct TrieNode \*root = getNode();

    int i;

    for (i = 0; i < ARRAY\_SIZE(keys); i++)

        insert(root, keys[i]);

    printf("%s --- %s\n", "the", output[search(root, "the")] );

    printf("%s --- %s\n", "these", output[search(root, "these")] );

    printf("%s --- %s\n", "their", output[search(root, "their")] );

    printf("%s --- %s\n", "thaw", output[search(root, "thaw")] );

    return 0; }



**WEEK 6**

**CONTENTS**

* Introduction To Python

Python

**Python** is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.

Notes On Python

* Python is an interpreted language that will be used in the context of this class to generate dynamic websites and web applications.
* Some basic Python syntax:
  + Print a string to the screen:
  + print("Hello, world!")
  + Print a format string (variable names enclosed with {} will be replaced by variable values)
  + print(f"Hello, {name}!")
  + Set variable name to the user input returned by input()
  + name = input()
  + Conditional statement:
  + if x > 0:
  + print("x is positive")
  + elif x < 0:
  + print("x is negative")
  + else:
  + print("x is zero")
    - elif and else blocks are optional.
    - Note that indentation in Python is not stylistic, but rather is used to demarcate blocks of code. In this example, the Python interpreter knows where the conditional if block ends and the elif block begins because of the changes in indentation.

### **Datatypes-**

* int : integer value
* float : floating point value
* str : text string
* bool : boolean value (True or False)
* None : empty value
* Note that Python is a weakly typed language.

### **Sequences-**

* Strings:
* name = "Alice"
* print(name[0])
  + Strings are justs sequence of characters, and can be indexed as such.
* Tuples:
* coordinates = (10.0, 20.0)
* print(coordinates[1])
  + Tuples are immutable collections of values under a single name, which can be indexed positionally.
* Lists:
* names = ["Alice", "Bob", "Charlie"]
* print(names[2])
  + Lists are mutable collections of values under a single name, which can be indexed positionally.
  + Indexing out of range raises a Python ‘exception’. In this case, an IndexError, because there is no fourth value in names for Python to return.
* Note that any sequence in Python can contain any number of data types.
* Sets:
* s = set()
* s.add(1)
* s.add(3)
* s.add(5)
  + Sets are unordered collection of unique items. Because they are unordered, they cannot be indexed.
  + s is a set, an unordered collection of unique items
* Dictionaries:
* ages = {"Alice": 22, "Bob": 27}
* print(ages["Alice"])
* ages["Alice"] += 1
  + Dictionaries (or dicts) are like lists, except that they are unordered and their values are indexed by keys.
  + The += operator increments the left-hand side by the right-hand side.

### **Loops-**

for i in range(5):

print(i)

* For-loops iterate over their bodies a limited number of times. In this case, the number of iterations is set by range(5).
* range(5) returns the sequence of numbers starting at 0 through 4. Each value is passed to i once, resulting in the loop running a total of 5 times. i is normally referred to as an iterator variable.

for name in names:

print(name)

* This for-loop iterates over names, which is a list. Every value in the list is assigned, in order, to the iterator name once.

### Functions-

* Python has built-in functions, such as print() and input(), but Python also allows for the creation of user-defined functions
* def square(x):
* return x \* x
  + This is a function called square, which takes a single argument x, and returns the value x \* x.
  + Like loops, the body of a function must be indented.
  + for i in range(10):
  + print("{} squared is {}".format(i, square(i)))
    - This loop, which prints out the results of square with a range of arguments, using an older method for format strings.
* Trying to call a function that hasn’t been defined will raise a NameError exception.

### **Modules-**

* Modules are separate .py files of code, often written by others, used in a new file without rewriting all the old code again. Using modules allows, for example, the use of functions across a program larger than a single file.
* Assuming the square function in the earlier example was saved in functions.py, adding this line atop a new module will allow for the use of square there as well.
* from functions import square
* If, for example, functions.py also included the example loop demonstration of the square function, that loop would be executed every time square was imported from functions, because the Python interpreter reads through the entire functions.py file. To remedy this, code that should only run when their containing file is run directly should be encapsulated in a function, called, for example, main. After, the following should be appended:
* if \_\_name\_\_ == "\_\_main\_\_":
* main()

This should be interpreted as saying ‘if this file is currently being run’, execute main.

### **Classes-**

* A Python class can be thought of as a way to define a new, custom Python data type, somewhat analagous to defining a custom function.
* This creates a new class called Point:
* class Point:
* def \_\_init\_\_(self, x, y):
* self.x = x
* self.y = y
  + The \_\_init\_\_ function is a special function that defines the information needed when a new Point is created. self is always required, which refers to the Point being created, while x and y are its coordinates.
  + self.x and self.y actually do the work of creating x and y attributes for the Point and assigning the m the values passed to \_\_init\_\_.
  + By convention, class names tend to start with a capital letter.
* This instantiates a new Point with x = 3 and y = 5:
* p = Point(3, 5)
  + When this line is run, the \_\_init\_\_ function of the Point class is automatically run.
* To access the x attribute of p, use dot notation:
* print(p.x)

## Flask-

* HTTP (Hypertext Transfer Protocol) is the system the internet uses to interact and communicate between computers and servers. When a URL is entered into a browser, an HTTP request is sent to a server, which interprets the request and sends appropriate HTTP response, which, if all goes as expected, contains the requested information to be displayed by the web browser.
* Having already begun to design websites, the next step is to write the code that takes care of the server-side processing: receiving and interpreting requests, and generating a response for the user.
* Flask a microframework written in Python that makes it easy to get a simple web application up and running with some features that can be useful in the development process.

### **A Simple App-**

* Flask code is generally stored inside application.py, and might look like so:
* from flask import Flask # Import the class `Flask` from the `flask` module, written by someone else.
* app = Flask(\_\_name\_\_) # Instantiate a new web application called `app`, with `\_\_name\_\_` representing the current file
* @app.route("/") # A decorator; when the user goes to the route `/`, exceute the function immediately below
* def index():
* return "Hello, world!"
  + Flask is designed in terms of routes. A route is the part of the URL that determines which page is being requested. The route for the default page is simply /.
* To start up a flask application, run flask run in the directory where application.py is located, with flask being the web server. Flask will print out the URL the server is running on and where the website can be accessed at.
  + flask run produces an error, try running export FLASK\_APP=application.py to make sure it knows to look for application.py as the web server.

### **Fancier Flask and Jinja2-**

@app.route("/<string:name>")

def hello(name):

return f"Hello, {name}!"

* When any string is entered as a route, that will be stored as name, which is can then be used inside the decorated function.
* Since Python code is rendering the website, anything Python is capable of can be used. For example, name can be capitalized before it’s displayed:

name = name.capitalize()

* HTML can also be used inside the return value:

return f"<h1>Hello, {name}!</h1>".

* Inline HTML isn’t that useful, though. Separate HTML files can be used like so:

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route("/")

def index():

return render\_template("index.html")

* index.html and any other template files should be stored in a directory named templates.
* Variables can be defined as Python variables in application.py and used in HTML templates by passing them in as arguments to render\_template. These templates are rendered using a separate templating language called Jinja2:
* In application.py:

headline = "Hello, world!"

return render\_template("index.html", headline=headline)

* In index.html:

<h1>{{ headline }}</h1>

* Jinja2 also allows for conditional statements:

{% if new\_year %}

<h1>Yes! Happy New Year!</h1>

{% else %}

<h1>No.</h1>

{% endif %}

* Loops:

{% for name in names %}

<li>{{ name }}</li>

{% endfor %}

* names should be something that can be looped over, like a Python list, for example.
* If there are multiple routes on the Flask server, then one route can link to another as so:

<a href="{{ url\_for('more') }}">See more...</a>

* more is the name of a function associated with a route.

#### **Template Inheritance-**

* In order to cut down on repetitive HTML amongst many different pages, Jinja2 has a feature called ‘template inheritance’ that uses the idea of blocks to organize content. For examples, have a look at layout.html and index.html in the inheritance/ directory of the same source code.
* Everything in the heading block is placed where indicated in layout.html, and same for body.

#### **Forms-**

* With Flask and Jinja2, the results from HTML forms can now be actually stored and used.
* An HTML form might look like this:
* <form action="" method="post">
* <input type="text" name="name" placeholder="Enter Your Name">
* <button>Submit</button>
* <form>
  + The action attribute lists the route that should be ‘notified’ when the form is submitted. In this case, it’s the URL for a function called hello.
  + The method attribute is how the HTTP request to submit the form should be made. The default method is get, which is what browsers make when a URL is entered. When data is being submitted, however, post should be used.
  + The name attribute of the input, while not new, is now relevant because it can be referenced when the form is submitted.
* The Python code to process this form might look like this:

from flask import Flask, render\_template, request

# some lines omitted here

@app.route("/hello", methods=["POST"])

def hello():

name = request.form.get("name") # take the request the user made, access the form,

# and store the field called `name` in a Python variable also called `name`

return render\_template("hello.html", name=name)

* The route /hello is the same hello listed in the Jinja2 code. This route can also accept the POST method, which is how the form’s data is being submitted. If any other method is used to access this route, a Method Not Allowed error will be raised.
  + If there are multiple request methods that should be allowed, which method is being used can be checked with request.method, which will be equal to, for example, "GET" or "POST".

#### **Sessions-**

* Sessions are how Flask can keep track of data that pertains to a particular user. Let’s take a note-taking app, for example. Users should only be able to see their own notes.
* To use sessions, they must be imported and set up:

from flask import Flask, render\_template, request, session # gives access to a variable called `session`

# which can be used to keep vaules that are specific to a particular user

from flask\_session import Session # an additional extension to sessions which allows them

# to be stored server-side

app.config["SESSION\_PERMANENT"] = False

app.config["SESSION\_TYPE"] = "filesystem"

Session(app)

* Then, assuming there is some HTML form that can submit a note, the note can be stored in a place specific to the user using their session:

@app.route("/", methods=["GET", "POST"])

def index():

if session.get("notes") is None:

session["notes"] = []

if request.method == "POST":

note = request.form.get("note")

session["notes"].append(note)

return render\_template("index.html", notes=session["notes"])

* notes is the list where the notes will be stored. If the user doesn’t have a notes list already (checked with if session.get("notes") is None), then they are given an empty one.
* If a request is submitted via "POST" (that is, through the form), then the note is processed from the form in the same way as before.
* The processed note, now in a Python variable called note, is appended to the notes list. This list is itself inside a dict called session. Every user has a unique session dict, and therefore a unique notes list.
* Finally, the notelist is rendered by passing session["notes"] to render\_template.



**WEEK 7**

**CONTENTS**

* Introduction To SQL

SQL

SQL(Structured Query Language) is a language which is used to interact with relational database management system. It lets us access and manipulate databases. SQL became a standard of the American National Standards Institute (ANSI) in 1986, and of the International Organization for Standardization (ISO) in 1987.

## Database-

## Database is a organized collection of data. For example a database of a college would be having a collection of data such as – 1. Personal records of Students 2. Students performance history 3. Teachers data 4. Financial department data etc.

## Database Management System (DBMS)-

A database management system is a software application which is used for managing different databases. It helps us to create and manage database. With the help of DBMS we take care following tasks –  
1. Data Security  
2. Data Backup  
3. Manages huge amount of data  
4. Data export & import  
5. Serving multiple concurrent database requests  
6. Gives us a way to manage the data using programming languages.

## Types of Databases

There are two types of databases –  
1. Relational Database  
2. Non-relational Database

**Non-relational Database-**

Data is **not** organized in form of tables. Data is stored in form of key & value pairs. The examples of non-relational databases are: JSON and XML.

**We cannot interact with non-relational databases using SQL.**

**Relational Database-**

In relational database, data is organized in form of tables. A table contains rows and columns of data. Table has a unique key to identify each row of the table.

**SQL is used to interact with relational databases**. We often refer relational database as SQL database.

## SQL

1. SQL stands for Structured Query Language, which is a standardised language for interacting with RDBMS (Relational Database Management System). Some of the popular relational database example are: MySQL, Oracle, mariaDB, postgreSQL etc.

2. SQL is used to perform C.R.U.D (Create, Retrieve, Update & Delete) operations on relational databases.

3. SQL can also perform administrative tasks on database such as database security, backup, user management etc.

4. We can create databases and tables inside database using SQL.

## Types of Structured Query Language(SQL)

SQL is basically combination of four different languages, they are –

#### **DQL (Data Query Language)**

DQL is used to fetch the information from the database which is already stored there.

#### **DDL (Data Definition Language)**

DDL is used to define table schemas.

#### **DCL (Data Control Language)**

DCL is used for user & permission management. It controls the access to the database.

#### **DML (Data Manipulation Language)**

DML is used for inserting, updating and deleting data from the database.

## What is a Query

A Query is a set of instruction given to the database management system, which tells RDBMS what information you would like to get from the database.

**Example**

SELECT employee\_name from EMPLOYEE;

## Using SQL in Web Site

To build a web site that shows data from a database, you will need:

* An RDBMS database program (i.e. MS Access, SQL Server, MySQL)
* To use a server-side scripting language, like PHP or ASP
* To use SQL to get the data you want
* To use HTML / CSS to style the page

## RDBMS

RDBMS stands for Relational Database Management System.

RDBMS is the basis for SQL, and for all modern database systems such as MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access.

The data in RDBMS is stored in database objects called tables. A table is a collection of related data entries and it consists of columns and rows.

Look at the "Customers" table:

### **Example**

SELECT \* FROM Customers;

Every table is broken up into smaller entities called fields. The fields in the Customers table consist of CustomerID, CustomerName, ContactName, Address, City, PostalCode and Country. A field is a column in a table that is designed to maintain specific information about every record in the table.

A record, also called a row, is each individual entry that exists in a table. For example, there are 91 records in the above Customers table. A record is a horizontal entity in a table.

A column is a vertical entity in a table that contains all information associated with a specific field in a table.

## Database Tables

A database most often contains one or more tables. Each table is identified by a name (e.g. "Customers" or "Orders"). Tables contain records (rows) with data.

**Example-**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **ContactName** | **Address** | **City** | **PostalCode** | **Country** |
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | Mexico D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Lulea | S-958 22 | Sweden |

## The table above contains five records (one for each customer) and seven columns (CustomerID, CustomerName, ContactName, Address, City, PostalCode, and Country).

**Some of The Most Important SQL Commands**

* **SELECT** - extracts data from a database
* **UPDATE** - updates data in a database
* **DELETE** - deletes data from a database
* **INSERT INTO** - inserts new data into a database
* **CREATE DATABASE** - creates a new database
* **ALTER DATABASE** - modifies a database
* **CREATE TABLE** - creates a new table
* **ALTER TABLE** - modifies a table
* **DROP TABLE** - deletes a table
* **CREATE INDEX** - creates an index (search key)
* **DROP INDEX** - deletes an index



**WEEK 7**

**CONTENTS**

* Final Project

Final Project

To Make A Software With Any Programming Language

I have Used C++ For Completion Of This Project.

**Source Code-**

#include<bits/stdc++.h>

#include <process.h>

#include <dir.h>

#define systemfile "C:\\Pro-File"

# define max 1000

using namespace std;

int assignrate(int inumber)

{

int cost;

switch(inumber)

{

case 1:case 7:case 23:case 27:case 42:cost=50;break;

case 2:case 26:case 43:cost=40;break;case 3:case 12:case 13:case 15:case 17:case 32:case 37:case 57:cost=20;break;case 4:case 34:case 35:case 39:cost=60;break;

case 5:case 46:cost=35;break;case 6:cost=8;break;case 8:case 18:case 29:case 31:case 40:case 47:case 48:case 49:case 52:case 53:case 55:case 56:cost=10;break;

case 9:case 11:case 24:case 51:case 54:cost=25;break;

case 10:case 21:case 38:cost=55;break;case 14:case 16:case 19:case 22:case 33:case 50:cost=30;break;

case 20:case 30:case 44:cost=15;break;case 25:case 36:cost=80;break;case 28:cost=2;break;case 41:case 45:cost=90;break;

}

return cost;

}

string assignname(int inumber)

{

string iname;

switch(inumber)

{

case 1:{iname="VEG MOMO";break;}case 2:{iname="CHHOLE BHATURE";break;}

case 3:{iname="FRIED RICE";break;}case 4:{iname="BURGER";break;}

case 5:{iname="DOSA";break;}case 6:{iname="SAMOSA";break;}

case 7:{iname="DHOKLA";break;}case 8:{iname="CHAAT";break;}

case 9:{iname="TIKIYA";break;}case 10:{iname="STEAMED MOMO";break;}

case 11:{iname="DAHI VADA";break;}case 12:{iname="SOUP";break;}

case 13:{iname="PARATHA";break;}case 14:{iname="PASTA";break;}

case 15:{iname="MAGGIE";break;}case 16:{iname="JALEBI";break;}

case 17:{iname="CHOLA SAMOSA";break;}case 18:{iname="BUN TIKKI";break;}

case 19:{iname="PAKODE";break;}case 20:{iname="SANDWITCH";break;}

case 21:{iname="UTPAM";break;}case 22:{iname="SPRING ROLLS";break;}

case 23:{iname="THUPPA";break;}case 24:{iname="CHAUMEEN";break;}

case 25:{iname="PIZZA";break;}case 26:{iname="IDLI";break;}

case 27:{iname="DAHI JALEBI";break;}case 28:{iname="PAANI PURI";break;}

case 29:{iname="BUN MAKKHAN";break;}case 30:{iname="BREAD PAKODA";break;}

case 31:{iname="TOAST";break;}case 32:{iname="UPMA";break;}

case 33:{iname="RAS MALAI";break;}case 34:{iname="NONVEG MOMO";break;}

case 35:{iname="CHICKEN";break;}case 36:{iname="MUTTON";break;}

case 37:{iname="FISH RICE";break;}case 38:{iname="NONVEG HOT DOG";break;}

case 39:{iname="NONVEG THUPPA";break;}case 40:{iname="HALF FRY";break;}

case 41:{iname="CHICKEN BURGER";break;}case 42:{iname="FISH CURRY";break;}

case 43:{iname="CHICKEN SOUP";break;}case 44:{iname="BUN OMELET";break;}

case 45:{iname="CHICKEN PIZZA";break;}case 46:{iname="EGG CURRY";break;}

case 47:{iname="TEA";break;}case 48:{iname="FANTA";break;}

case 49:{iname="MIRINDA";break;}case 50:{iname="MILK SHAKE";break;}

case 51:{iname="COFFEE";break;}case 52:{iname="MAZA";break;}

case 53:{iname="FROOTI";break;}case 54:{iname="LASSI";break;}

case 55:{iname="APPY";break;}case 56:{iname="THUMBS UP";break;}

case 57:{iname="MINERAL WATER";break;}

}

return iname;

}

class program\_data

{

private:

char name[500];

char pass[500];

char confirm\_pass[500];

char choice;

char folder\_name[500];

string item\_name;

int item\_number[50],rate[50],quantity[50];

float total[50],net\_total;

public:

void profile();

void itemsbill(int);

void new\_profile();

void locker\_maker();

};

void program\_data::new\_profile()

{

ofstream saveprofile("C:\\Pro-File\\PROFILE.sht");

system("cls");

cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*||PROFILE EDITOR||\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n";

fflush(stdin);

cout<<"===============================================================\n";

cout<<"|| Enter a new name || ";

gets(name);

cout<<"===============================================================\n";

saveprofile<<name;

passregis:

cout<<"===============================================================\n";

cout<<"|| Enter a new password || ";

gets(pass);

cout<<"===============================================================\n";

int q=strlen(pass);

if (q<8)

{

cout<<"====================================================================\n";

cout<<"|| Your password is small, it should atleast have 8 characters....||\n";

cout<<"====================================================================\n";

goto passregis;

}

cout<<"===============================================================\n";

cout<<"|| Confirm your password || ";

gets(confirm\_pass);

cout<<"===============================================================\n";

int i=strcmp(pass,confirm\_pass);

if(i==0)

{

cout<<"===============================\n";

cout<<"|| Successfully Registered... ||";

cout<<"===============================\n";

ofstream savepass("C:\\Pro-File\\KEY.sht");

for(int i=1;i<=sizeof(pass);i++)

{

pass[i]=pass[i]\*pass[i]\*pass[i]\*pass[i]\*pass[i]\*pass[i]\*pass[i];

}

savepass<<pass;

saveprofile.close();

savepass.close();

system("cls");

}

else

{

cout<<"=========================================================================";

cout<<"|| The password and confirm-password don't match, please try again.... ||";

cout<<"=========================================================================";

goto passregis;

}

}

void program\_data::locker\_maker()

{

fflush(stdin);

ofstream makefile("Folder Locker.sht.bat",ios::out);

cout<<"==================\n";

cout<<"|| Instructions ||\n";

cout<<"==================================================================================\n";

cout<<"||The locker is a place where you can safely hide your important files. ||\n";

cout<<"||To lock your files, first click on the locker named as \"Folder Locker.sht\". ||\n";

cout<<"||Then a folder will appear with the name given by you. ||\n";

cout<<"||Put your important files in that folder and again click open the locker. ||\n";

cout<<"||Enter 'Y' if you want to lock the folder and 'N' if you don't want. ||\n";

cout<<"==================================================================================\n";

cout<<"|| Note ||\n";

cout<<"==============================================================================================================================\n";

cout<<"||1|| Don't use spaces, special characters or numbers while writing the folder name ||\n";

cout<<"||2|| Don't use spaces or special characters while writing a password (use only words), or it may generate error!!! ||\n";

cout<<"==============================================================================================================================\n";

cout<<"===============================\n";

cout<<"|| Press ENTER to continue ||\n";

cout<<"===============================\n";

getchar();

cout<<"===============================\n";

cout<<"|| Press ENTER to continue ||\n";

cout<<"===============================\n";

getchar();

system("cls");

cout<<"=========================================================================\n";

cout<<"|| Enter a name for your secure folder-> ";

gets(folder\_name);

cout<<"\n=========================================================================\n";

passregis:

cout<<"=====================================================================\n";

cout<<"|| Enter a password for your locker-> ";

gets(pass);

cout<<"\n=======================================================================\n";

int q=strlen(pass);

if (q<8)

{

cout<<"=================================================================\n";

cout<<"|| Your password is small, it should atleast have 8 characters ||\n";

cout<<"=================================================================\n";

goto passregis;

}

cout<<"==========================================================\n";

cout<<"|| Confirm your password-> ";

gets(confirm\_pass);

cout<<"\n==========================================================\n";

int i=strcmp(pass,confirm\_pass);

if(i==0)

{

makefile<<"cls\n";

makefile<<" @ECHO OFF\n title Secure Folder For Your Files by- Shreshth Tiwari\n";

makefile<<" if EXIST \"Control Panel.{21EC2020-3AEA-1069-A2DD-08002B30309D}\" goto UNLOCK\n";

makefile<<" if NOT EXIST "<<folder\_name<<" goto MDLOCKER\n :CONFIRM\n";

makefile<<" echo Are you sure you want to lock the folder(Y/N)\n";

makefile<<" set/p \"cho=>\"\n if %cho%==Y goto LOCK\n";

makefile<<" if %cho%==y goto LOCK\n if %cho%==n goto END\n";

makefile<<" if %cho%==N goto END\n echo Invalid choice.\n goto CONFIRM\n :LOCK\n";

makefile<<" ren "<<folder\_name<<" \"Control Panel.{21EC2020-3AEA-1069-A2DD-08002B30309D}\"\n";

makefile<<" attrib +h +s \"Control Panel.{21EC2020-3AEA-1069-A2DD-08002B30309D}\"\n";

makefile<<" echo Folder locked\n goto End\n :UNLOCK\n echo Enter the Password to unlock folder\n";

makefile<<" set/p \"pass=>\"\n";

makefile<<" if NOT %pass%== "<<pass<<" goto FAIL\n";

makefile<<" attrib -h -s \"Control Panel.{21EC2020-3AEA-1069-A2DD-08002B30309D}\"\n";

makefile<<" ren \"Control Panel.{21EC2020-3AEA-1069-A2DD-08002B30309D}\" "<<folder\_name<<"\n";

makefile<<" echo Folder Unlocked successfully\n";

makefile<<" goto End\n :FAIL\n echo Invalid password\n";

makefile<<" goto end\n :MDLOCKER\n md "<<folder\_name<<"\n";

makefile<<" echo "<<folder\_name<<" created successfully\n goto End\n :End\n";

cout<<"\n Successfully created The Locker";

cout<<"\n\nPress Enter to continue.....";

getchar();

}

else

{

cout<<"\nThe password and confirm-password don't match, please try again....";

goto passregis;

}

}

void program\_data::profile()

{

ifstream profile("C:\\Pro-File\\PROFILE.sht");

string line;

ifstream savepass("C:\\Pro-File\\KEY.sht");

string checkpass;

if(profile.is\_open()&&savepass.is\_open())

{

system("cls");

while (getline(profile,line))

{

cout<<" =====================================\n";

cout<<" || SHOP OF-:"<<line<<" \n";

cout<<" =====================================\n";

cout<<"\n\n\nHELLO "<<line<<" PLEASE ENTER YOUR PASSWORD SO THAT I CAN START WORKING.....\n\n";

}

while (getline(savepass,checkpass))

{

logincheck:

cout<<"================================================\n";

cout<<"|| Password-> ";

gets(pass);

cout<<"\n================================================\n";

for(int i=1;i<=sizeof(pass);i++)

{

pass[i]=pass[i]\*pass[i]\*pass[i]\*pass[i]\*pass[i]\*pass[i]\*pass[i];

}

if (pass==checkpass)

{

system("cls");

system("color 21");

cout<<"======================================================\n";

cout<<"|| WELCOME HOPE I'LL BE VERY HELPFUL TO YOU ||\n";

cout<<"======================================================\n";

cout<<"|| Press ENTER to continue ||\n";

cout<<"=============================\n";

getchar();

system("cls");

}

else

{

system("cls");

system("color 4E");

cout<<"====================================================\n";

cout<<"|| SORRY YOU ENTERED A WRONG PASSWORD \n";

cout<<"|| EXIT THE PROGRAM NOW IF YOU ARE AN INTRUDER!! \n";

cout<<"====================================================\n";

goto logincheck;

}

}

savepass.close();

profile.close();

}

else

{

cout<<"===========================================================================================\n";

cout<<"|| Seems like you are new or your old profile is deleted. Please first register yourself ||\n";

cout<<"===========================================================================================\n";

new\_profile();

}

}

void program\_data::itemsbill(int number\_of\_items)

{

net\_total=0;

time\_t now=time(0);

char\* date=ctime(&now);

ofstream savedate("LOGS.xls",ios::app);

savedate<<"=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=\n";

savedate<<"----------------------------------\n";

savedate<<date;

savedate<<"----------------------------------\n";

savedate.close();

ofstream savedata("LOGS.xls",ios::app);

ofstream billbuff("C:\\Pro-File\\BILLBUFF.sht");

savedata<<"S.no"<<'\t'<<"Item ID"<<'\t'<<"Item Name"<<'\t'<<"Cost"<<'\t'<<"Quantity"<<'\t'<<"Total";

cout<<"\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*|| ITEMS EDITOR ||\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*";

for(int i=1;i<=number\_of\_items;i++)

{

itemlist:

cout<<'\n';

cout<<"===========\n";

cout<<"|| Press ||\n";

cout<<"===========\n";

cout<<"|| VEG ||\n";

cout<<"===================================================================================\n";

cout<<"||(1) for VEG-MOMO ||(12) for SOUP ||(23) for THUPPA ||\n";

cout<<"||(2) for CHHOLE-BHATURE ||(13) for PARATHA ||(24) for CHAUMEEN ||\n";

cout<<"||(3) for FRIED RICE ||(14) for PASTA ||(25) for PIZZA ||\n";

cout<<"||(4) for BURGER ||(15) for MAGGIE ||(26) for IDLI ||\n";

cout<<"||(5) for DOSA ||(16) for JALEBI ||(27) for DAHI JALEBI ||\n";

cout<<"||(6) for SAMOSA ||(17) for CHHOLA SAMOSA ||(28) for PANI PURI ||\n";

cout<<"||(7) for DHOKLA ||(18) for BUN TIKKI ||(29) for BUN MAKKHAN ||\n";

cout<<"||(8) for CHAAT ||(19) for PAKODE ||(30) for BREAD-PAKODA ||\n";

cout<<"||(9) for TIKIYA ||(20) for SANDWITCH ||(31) for TOAST ||\n";

cout<<"||(10) for STEAMED MOMO ||(21) for UTPAM ||(32) for UPMA ||\n";

cout<<"||(11) for DAHI-VADA ||(22) for SPRING ROLLS ||(33) for RAS MALAI ||\n";

cout<<"===================================================================================\n";

cout<<"|| NON-VEG || \n";

cout<<"===================================================================================\n";

cout<<"||(34) for NONVEG MOMO ||(39) for NONVEG THUPPA ||(43) for CHICKEN SOUP ||\n";

cout<<"||(35) for CHICKEN ||(40) for HALF FRY ||(44) for BUN OMELET ||\n";

cout<<"||(36) for MUTTON ||(41) for CHICKEN BURGER ||(45) for CHICKEN PIZZA ||\n";

cout<<"||(37) for FISH RICE ||(42) for FISH CURRY ||(46) for EGG CURRY ||\n";

cout<<"||(38) for NONVEG HOT DOG || || ||\n";

cout<<"===================================================================================\n";

cout<<"|| BEVARAGES || \n";

cout<<"===================================================================================\n";

cout<<"||(47) for TEA ||(51) for COFFEE ||(55) for APPY ||\n";

cout<<"||(48) for FANTA ||(52) for MAZA ||(56) for THUMBS UP ||\n";

cout<<"||(49) for MIRINDA ||(53) for FROOTI ||(57) for MINERAL WATER ||\n";

cout<<"||(50) for MILK SHAKE ||(54) for LASSI || ||\n";

cout<<"===================================================================================\n";

cout<<'\n'<<i<<". Item Number-: ";

cin>>item\_number[i];

if(item\_number[i]<1||item\_number[i]>57)

{

cout<<"\nITEM NOT FOUND TRY AGAIN...";

goto itemlist;

}

item\_name=assignname(item\_number[i]);

cout<<"\n Item Name-: "<<item\_name;

savedata<<'\n'<<i<<'\t'<<item\_number[i]<<'\t'<<item\_name;

billbuff<<'\n'<<i<<"- "<<"Item Number- "<<item\_number[i]<<'\n'<<" Item Name- "<<item\_name;

rate[i]=assignrate(item\_number[i]);

cout<<"\n Cost of 1 "<<item\_name<<"-: Rs."<<rate[i];

quantityinput:

cout<<"\n Quantity (max=1000)-: ";

cin>>quantity[i];

if(quantity[i]<1||quantity[i]>max)

{

goto quantityinput;

}

cout<<"\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n";

total[i]=rate[i]\*quantity[i];

net\_total+=total[i];

savedata<<'\t'<<"Rs."<<rate[i]<<'\t'<<quantity[i]<<'\t'<<"Rs."<<total[i];

billbuff<<'\n'<<" Cost- "<<"Rs."<<rate[i]<<'\n'<<" Quantity- "<<quantity[i]<<'\n'<<" Total- "<<"Rs."<<total[i]<<"\n-----------------\n";

}

savedata<<"\n\t\t\tNet Total-: Rs."<<net\_total<<"\n\n";

savedata.close();

billbuff.close();

(system("cls"));

cout<<"\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*|| BILL ||\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*";

fstream billbuf("C:\\Pro-File\\BILLBUFF.sht");

char getbill[9999];

if(billbuf.is\_open())

{

while (billbuf.read((char \*)&getbill,sizeof(getbill)));

{

cout<<getbill<<"\n\n";

cout<<"Net Total-: Rs."<<net\_total<<"\n\n";

billbuf<<" ";

billbuf.close();

}

}

system("pause");

}

int main()

{

system("color B0");

int noi,nchoice;

int stat;

stat = mkdir(systemfile);

billsinput:

program\_data data;

cout<<"\t\t\t\t===============================\n";

cout<<"\t\t\t\t|| RESTAURANT BILLING SYSTEM ||\n";

cout<<"\t\t\t\t===============================\n";

cout<<"\t\t\t\t =================================\n";

cout<<"\t\t\t\t || CODED BY \"Shreshth Tiwari\" ||\n";

cout<<"\t\t\t\t =================================\n\n\n";

cout<<"==========\n";

cout<<"|| NOTE ||\n";

cout<<"==========================================================================\n";

cout<<"|| THE PROGRAM WORKS ON NUMBERS. ||\n";

cout<<"|| SO PLEASE DON'T USE ALPHABETS AND OTHER CHARACTERS WHEN NOT NEEDED. ||\n";

cout<<"==========================================================================\n";

cout<<"\t\t\t|| OR IT WILL TERMINATE THE PROGRAM ||\n";

cout<<"\t\t\t======================================\n";

cout<<"=============================\n";

cout<<"|| Press ENTER to continue ||\n";

cout<<"=============================\n";

getchar();

system("cls");

data.profile();

menu:

system("cls");

system("color 21");

cout<<" =================\n";

cout<<" || MAIN MENU ||\n";

cout<<" =================\n\n\n";

cout<<"===========\n";

cout<<"|| Enter ||\n";

cout<<"================================================================================\n";

cout<<"||(1)|| To start billing ||\n";

cout<<"||(2)|| To edit your profile =============== ||\n";

cout<<"||(3)|| To create a folder locker for your secret files. || (special) || ||\n";

cout<<"||(4)|| To clear all billing logs =============== ||\n";

cout<<"||(5)|| To exit ||\n";

cout<<"||(6)|| To exit and restart your system. ||\n";

cout<<"||(7)|| To exit and shut down your system. ||\n";

cout<<"================================================================================\n";

cout<<"|| Choice-> ";

cin>>nchoice;

cout<<"\n================================================================================\n";

if(nchoice==1)

{

noiinput:

system("cls");

cout<<"======================================\n";

cout<<"|| How many items you want to enter ||\n";

cout<<"|| (Use only numbers)(max=1000) ||\n";

cout<<"======================================\n";

cout<<"|| Choice-> ";

cin>>noi;

cout<<"\n====================================\n";

if (noi<1||noi>max)

{

noi=0;

cout<<"=============================\n";

cout<<"|| Error, please try again ||\n";

cout<<"=============================\n";

goto noiinput;

}

cout<<"\nLoading...";

system("cls");

data.itemsbill(noi);

cout<<"\n\n";

cout<<"========================================================\n";

cout<<"|| Want to enter the items again? ||\n";

cout<<"|| (Press '1' for yes or enter any other key to exit) ||\n";

cout<<"========================================================\n";

cout<<"|| Choice-> ";

cin>>nchoice;

cout<<"\n======================================================\n";

if (nchoice==1)

{

goto menu;

}

}

else if(nchoice==2)

{

data.new\_profile();

goto menu;

}

else if(nchoice==3)

{

system("cls");

data.locker\_maker();

goto menu;

}

else if(nchoice==4)

{

system("cls");

system("color 4E");

cout<<"\t=========================================================\n";

cout<<"\t|| Do you really want to clear your logs. ||\n";

cout<<"\t|| [Press (1) For YES or any other number key for NO] ||\n";

cout<<"\t=========================================================\n";

cout<<"\t|| Choice-> ";

cin>>nchoice;

cout<<"\n\t=======================================================\n";

if(nchoice==1)

{

ofstream clearfile("LOGS.xls",ios::trunc);

clearfile<<"";

clearfile.close();

system("cls");

cout<<"Logs cleared\n";

}

else

{

system("cls");

}

goto menu;

}

else if(nchoice==5)

{

system("cls");

system("color 4E");

cout<<"\t======================================================\n";

cout<<"\t|| Do you really want to Exit. ||\n";

cout<<"\t||[Press (1) For YES or any other number key for NO]||\n";

cout<<"\t========================================================\n";

cout<<"\t|| Choice-> ";

cin>>nchoice;

cout<<"\n\t======================================================\n";

if(nchoice==1)

{

exit(0);

}

else

{

system("cls");

}

goto menu;

}

else if(nchoice==6)

{

system("cls");

system("color 4E");

cout<<"\t======================================================\n";

cout<<"\t|| Do you really want to Restart Your System? ||\n";

cout<<"\t|| !! YOU MAY LOSE ANY UNSAVED WORK !! ||\n";

cout<<"\t||[Press (1) For YES or any other number key for NO]||\n";

cout<<"\t======================================================\n";

cout<<"\t|| Choice-> ";

cin>>nchoice;

cout<<"\n\t======================================================\n";

if(nchoice==1)

{

system("C:\\WINDOWS\\System32\\shutdown /r");

exit(0);

}

else

{

system("cls");

}

goto menu;

}

else if(nchoice==7)

{

system("cls");

system("color 4E");

cout<<"\t======================================================\n";

cout<<"\t|| Do you really want to Shut Down Your System? ||\n";

cout<<"\t|| !! YOU MAY LOSE ANY UNSAVED WORK !! ||\n";

cout<<"\t||[Press (1) For YES or any other number key for NO]||\n";

cout<<"\t======================================================\n";

cout<<"\t|| Choice-> ";

cin>>nchoice;

cout<<"\n\t======================================================\n";

if(nchoice==1)

{

system("C:\\WINDOWS\\System32\\shutdown /s");

exit(0);

}

else

{

system("cls");

}

goto menu;

}

else

{

system("cls");

goto menu;

}

}