DATA-TYPES

BUILT-IN

int

float

double

char

string

bool

USER-DEFINED

struct

enum

union

DERIVED

Array

function

pointer

SCOPE OF A VARIABLE

The scope of a variable is the region in the program where the existence of that variable is valid. For example, consider this analogy - if one person travels to another country illegally, we will not consider that country as its scope because he doesn't have the necessary documents to stay in that country.

Local Variable

Local variables are declared inside the braces of any function and can be assessed only from there.

Global Variable

Global variables are declared outside any function and can be accessed from anywhere.

VARIABLES

TYPE	EXAMPLE	MEANING
int	int i = 5;	Integer (whole number)
float	float f = 45.5;	Decimal numbers (small)
double	double d = 45.4552;	Bigger decimal
char	char ch = ‘Y’;	Single character
string	string s = “Yokshita”	Word or sentence (needs #include <string>)
bool	bool b = true;	true or false

Rules to define a variable

- Variable names in C++ language can range from 1 to 255
- Variable names must start with a letter of the alphabet or an underscore
- After the first letter, variable names can contain letters and numbers
- Variable names are case sensitive
- No spaces and special characters are allowed
- We cannot use reserved keywords as a variable name

C++ OPERATORS

Special symbols that are used to perform actions or operations are known as operators. They could be both unary or binary.
For example, the symbol + is used to perform addition in C++ when put in between two numbers, so it is a binary operator. There are different types of operators.

Arithmetic Operator

Arithmetic operators are used to perform mathematical operations such as addition, subtraction, etc. They could be both binary and unary. A few of the simple arithmetic operators are

OPERATOR	DESCRIPTION
a + b	Adds a and b
a - b	Subtracts b from a
a * b	Multiplies a and b
a / b	Divides a by b
a % b	Modulus of a and b
a++	Post increments a by 1
a--	Post decrements a by 1
++a	Post decrements a by 1
--a	Pre decrements a by 1

Relational Operators

Relational operators are used to check the relationship between two operands and to compare two or more numbers or even expressions in cases. The return type of a relational operator is a Boolean that is, either True or False (1 or 0).

OPERATOR	DESCRIPTION
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Is equal to
!=	Is not equal to

Logical Operators

Logical Operators are used to check whether an expression is true or false. There are three logical operators i.e. AND, OR, and NOT. They can be used to compare Boolean values but are mostly used to compare expressions to see whether they are satisfying or not.

- AND: it returns true when both operands are true or 1.
- OR: it returns true when either operand is true or 1.
- NOT: it is used to reverse the logical state of the operand and is true when the operand is false.

OPERATOR	DESCRIPTION
&&	AND Operator
||	OR Operator
!	NOT Operator

Bitwise Operators

A bitwise operator is used to perform operations at the bit level. To obtain the results, they convert our input values into binary format and then process them using whatever operator they are being used with.

OPERATOR	DESCRIPTION
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise Complement
>>	Shift Right Operator
<<	Shift Left Operator

Assignment Operators

Assignment operators are used to assign values. We will use them in almost every program we develop.

OPERATOR	DESCRIPTION
=	It assigns the right side operand value to the left side operand.
+=	It adds the right operand to the left operand and assigns the result to the left operand.
-=	It subtracts the right operand from the left operand and assigns the result to the left operand.
*=	It multiplies the right operand with the left operand and assigns the result to the left operand.
/=	It divides the left operand with the right operand and assigns the result to the left operand.

Operator Precedence and Associativity

Operator precedence

It helps us determine the precedence of an operator over another while solving an expression. Consider an expression a+b*c. Now, since the multiplication operator's precedence is higher than the precedence of the addition operator, multiplication between a and b is done first and then the addition operation will be performed.

Operator associativity

It helps us to solve an expression; when two or more operators having the same precedence come together in an expression. It helps us decide whether we should start solving the expression containing operators of the same precedence from left to right or from right to left.

C++ MANIPULATORS

In C++ programming, language manipulators are used in the formatting of output. These are helpful in modifying the input and the output stream. They make use of the insertion and extraction operators to modify the output. Here's a list of a few manipulators:

MANIPULATORS	DESCRIPTION
endl	It is used to enter a new line with a flush.
setw(a)	It is used to specify the width of the output.
setprecision(a)	It is used to set the precision of floating-point values.
setbase(a)	It is used to set the base value of a numerical number.

C++ Basic

INPUT/OUTPUT

C++ language comes with different libraries, which help us in performing input/output operations. In C++, sequences of bytes corresponding to input and output are commonly known as streams. There are two types of streams.

Input stream

In the input stream, the direction of the flow of bytes occurs from the input device (for ex keyboard) to the main memory.

Output stream

In the output stream, the direction of flow of bytes occurs from the main memory to the output device (for ex-display)

Important Points

- **The sign << is called the insertion operator.**
- **The sign >> is called the extraction operator.**
- **cout keyword is used to print.**
- **cin keyword is used to take input at run time.**

CONTROL STRUCTURE

The work of control structures is to give flow and logic to a program. There are three types of basic control structures in C++.

Sequence Structure

Sequence structure refers to the sequence in which program execute instructions one after another.

Selection Structure

Selection structure refers to the execution of instruction according to the selected condition, which can be either true or false. There are two ways to implement selection structures. They are done either by if-else statements or by switch case statements.

Loop Structure

Loop structure refers to the execution of an instruction in a loop until the condition gets false.

C++ If Else

If else statements are used to implement a selection structure. Like any other programming language, C++ also uses the if keyword to implement the decision control instruction.

The condition for the if statement is always enclosed within a pair of parentheses. If the condition is true, then the set of statements following the if statement will execute. And if the condition evaluates to false, then the statement will not execute, instead, the program skips that enclosed part of the code.

An expression in if statements are defined using relational operators. The statement written in an if block will execute when the expression following if evaluates to true. But when the if block is followed by an else block, then when the condition written in the if block turns to be false, the set of statements in the else block will execute.

```
if ( condition )
{
   statements;
}
else
{
   statements;
}
```

SWITCH CASE

The control statement that allows us to make a decision effectively from the number of choices is called a switch, or a switch case-default since these three keywords go together to make up the control statement.
Switch executes that block of code, which matches the case value. If the value does not match with any of the cases, then the default block is executed.

Following is the syntax of switch case-default statements:

```
switch ( integer/character expression )
{
   case {value 1} :
          do this ;
          break;

   case {value 2} :
          do this ;
          break;

   default :
          do this ;
}
```

The expression following the switch can be an integer expression or a character expression. Remember, that case labels should be unique for each of the cases. If it is the same, it may create a problem while executing a program. At the end of the case labels, we always use a colon (:). Each case is associated with a block. A block contains multiple statements that are grouped together for a particular case.

The break keyword in a case block indicates the end of a particular case. If we do not put the break in each case, then even though the specific case is executed, the switch will continue to execute all the cases until the end is reached. The default case is optional. Whenever the expression's value is not matched with any of the cases inside the switch, then the default case will be executed.

C++ Loops

The need to perform an action, again and again, with little or no variations in the details each time they are executed is met by a mechanism known as a loop. This involves repeating some code in the program, either a specified number of times or until a particular condition is satisfied. Loop-controlled instructions are used to perform this repetitive operation efficiently ensuring the program doesn't look redundant at the same time due to the repetitions.

Following are the three types of loops in C++ programming.

- For Loop
- While Loop
- Do While Loop

FOR LOOP

A for loop is a repetition control structure that allows us to efficiently write a loop that will execute a specific number of times. The for-loop statement is very specialized. We use a fo loop when we already know the number of iterations of that particular piece of code we wish to execute. Although, when we do not know about the number of iterations, we use a while loop which is discussed next.

- initialize counter: It will initialize the loop counter value. It is usually i=0.
- test counter: This is the test condition, which if found true, the loop continues, otherwise terminates.
- Increment/decrement counter: Incrementing or decrementing the counter.
- Set of statements: This is the body or the executable part of the for loop or the set of statements that has to repeat itself.

WHILE LOOP

A While loop is also called a pre-tested loop. A while loop allows a piece of code in a program to be executed multiple times, depending upon a given test condition which evaluates to either true or false. The while loop is mostly used in cases where the number of iterations is not known. If the number of iterations is known, then we could also use a for loop as mentioned previously.

The body of a while loop can contain a single statement or a block of statements. The test condition may be any expression that should evaluate as either true or false. The loop iterates while the test condition evaluates to true. When the condition becomes false, it terminates.

SYNTAX

```
while (condition test)
        {
  // Set of statements
        }
```

Do While Loop

A do-while loop is a little different from a normal while loop. A do-while loop, unlike what happens in a while loop, executes the statements inside the body of the loop before checking the test condition.

So even if a condition is false in the first place, the do-while loop would have already run once. A do-while loop is very much similar to a while loop, except for the fact that it is guaranteed to execute the body at least once.

Unlike for and while loops, which test the loop condition first, then execute the code written inside the body of the loop, the do-while loop checks its condition at the end of the loop.

Following is the syntax for using a do-while loop.

SYNTAX

First, the body of the do-while loop is executed once. Only then, the test condition is evaluated. If the test condition returns true, the set of instructions inside the body of the loop is executed again, and the test condition is evaluated. The same process goes on until the test condition becomes false. If the test condition returns false, then the loop terminates.

```
do
{
      statements;
} while (test condition);
```

BREAK STATEMENT

Break statement is used to break the loop or switch case statements execution and brings the control to the next block of code after that particular loop or switch case it was used in.

Break statements are used to bring the program control out of the loop it was encountered in. The break statement is used inside loops or switch statements in C++ language.

One such example to demonstrate how a break statement works is

```
#include <iostream>
using namespace std;

int main()
{
int num = 10;
int i;
for (i = 0; i < num; i++)
{
if (i == 6)
{
break;
}
cout << i << " ";
}

return 0;
}
```

CONTINUE STATEMENT

The continue statement is used inside loops in C++ language. When a continue statement is encountered inside the loop, the control jumps to the beginning of the loop for the next iteration, skipping the execution of statements inside the body of the loop after the continue statement.

It is used to bring the control to the next iteration of the loop. Typically, the continue statement skips some code inside the loop and lets the program move on with the next iteration. It is mainly used for a condition so that we can skip some lines of code for a particular condition.

It forces the next iteration to follow in the loop unlike a break statement, which terminates the loop itself the moment it is encountered.

One such example to demonstrate how a continue statement works is

```
#include <iostream>
using namespace std;

int main()
{
for (int i = 0; i <= 10; i++)
{
if (i < 6)
{
continue;
}
cout << i << " ";
}
return 0;
}
```

Array BASICS

An array is a collection of items that are of the data type stored in contiguous memory locations. And it is also known as a subscript variable.
It can even store the collection of derived data types such as pointers, structures, etc.

An array can be of any dimension. The C++ Language places no limits on the number of dimensions in an array. This means we can create arrays of any number of dimensions. It could be a 2D array or a 3D array or more.

ADVANTAGES OF *Array*

- It is used to represent multiple data items of the same type by using only a single name.
- Accessing any random item at any random position in a given array is very fast in an array.
- There is no case of memory shortage or overflow in the case of arrays since the size is fixed and elements are stored in contiguous memory locations.

Array Operation

Defining an array

1. Without specifying the size of the array:

```
int arr[] = {1, 2, 3};
```

2. With specifying the size of the array

```
int arr[3];
arr[0] = 1, arr[1] = 2, arr[2] = 3;
```

Accessing an array element

An element in an array can easily be accessed through its index number.
An index number is a special type of number which allows us to access variables of arrays. Index number provides a method to access each element of an array in a program. This must be remembered that the index number starts from 0 and not one.

```
#include <iostream>
using namespace std;

int main()
{
int arr[] = {1, 2, 3};
cout << arr[1] << endl;
}
```

Changing an array element

An element in an array can be overwritten using its index number.

```
#include <iostream>
using namespace std;

int main()
{
int arr[] = {1, 2, 3};
arr[2] = 8; //changing the element on index 2
cout << arr[2] << endl;
}
```

Pointers

A pointer is a data type that holds the address of another data type. A pointer itself is a variable that points to any other variable. It can be of type int, char, array, function, or even any other pointer. Pointers in C++ are defined using the '*' (asterisk) operator.

The '&'(ampersand) operator is called the 'address of' operator, and the '*'(asterisk) operator is called the 'value at' dereference operator.

Applications of a Pointer

- Pointers are used to dynamically allocate or deallocate memory.
- Pointers are used to point to several containers such as arrays, or structs, and also for passing addresses of containers to functions.
- Return multiple values from a function
- Rather than passing a copy of a container to a function, we can simply pass its pointer. This helps reduce the memory usage of the program.
- Pointer reduces the code and improves the performance.

OPERATIONS ON POINTERS

ADDRESS OF OPERATOR (&):

& is also known as the Referencing Operator. It is a unary operator. The variable name used along with the Address of operator must be the name of an already defined variable.

Using & operator along with a variable gives the address number of the variable.

```
#include <iostream>
using namespace std;

int main()
{
    int a = 10;
    cout << "Address of variable a is " << &a << endl;
    return 0;
}
```

OPERATIONS ON POINTERS

INDIRECTION OPERATOR

* is also known as the Dereferencing Operator. It is a unary operator. It takes an address as its argument and returns the content/container whose address is its argument.

```
#include <iostream>
using namespace std;

int main()
{
    int a = 100;
    cout << "Value of variable a stored at address "
<< &a << " is " << (*(&a)) << endl;
    return 0;
}
```

OPERATIONS ON POINTERS

POINTER TO POINTER

Pointer to Pointer is a simple concept, in which we store the address of one pointer to another pointer. This is also known as multiple indirections owing to the operator's name. Here, the first pointer contains the address of the second pointer, which points to the address where the actual variable has its value stored.

```
#include <iostream>
using namespace std;

int main()
{
    int a = 100;
    int *b = &a;
    int **c = &b;
    cout << "Value of variable a is " << a << endl;
    cout << "Address of variable a is " << b << endl;
    cout << "Address of pointer b is " << c << endl;
    return 0;
}
```

OPERATIONS ON POINTERS

ARRAYS AND POINTERS

Storing the address of an array into pointer is different from storing the address of a variable into the pointer. The name of an array itself is the address of the first index of an array. So, to use the (ampersand)& operator with the array name for assigning the address to a pointer is wrong. Instead, we used the array name itself.

```
int marks[] = {99, 100, 38};
int *p = marks;
cout << "The value of marks[0] is " << *p << endl;
```

STRING

A string is an array of characters. Unlike in C, we can define a string variable and not necessarily a character array to store a sequence of characters. Data of the same type are stored in an array, for example, integers can be stored in an integer array, similarly, a group of characters can be stored in a character array or a string variable. A string is a one-dimensional array of characters.

Declaring a string is very simple, the same as declaring a one-dimensional array. It's just that we are considering it as an array of characters.

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    // declare and initialise string
    string str = "Yokshita";
    cout << str << endl;
    return 0;
}
```

Structures

Variables store only one piece of information and arrays of certain data types store the information of the same data type. Variables and arrays can handle a great variety of situations. But quite often, we have to deal with the collection of dissimilar data types. For dealing with cases where there is a requirement to store dissimilar data types, C++ provides a data type called ‘structure’. The structure is a user-defined data type that is available in C++. Structures are used to combine different types of data types, just like an array is used to combine the same type of data types.

Another way to store data of dissimilar data types would have been constructing individual arrays, but that must be unorganized. It is to keep in mind that structure elements too are always stored in contiguous memory locations.

```
#include <iostream>
using namespace std;

struct employee
{
   /* data */
   int eId;
   char favChar;
   int salary;
};

int main()
{
   struct employee Harry;
   return 0;
}
```

Features of Structs

1. We can assign the values of a structure variable to another structure variable of the same type using the assignment operator.

2. Structure can be nested within another structure which means structures can have their members as structures themselves.

3. We can have a pointer pointing to a struct just like the way we can have a pointer pointing to an int, or a pointer pointing to a char variable.

4. We can pass the structure variable to a function. We can pass the individual structure elements or the entire structure variable into the function as an argument. And functions can also return a structure variable.

Accessing struct elements

To access any of the values of a structure's members, we use the dot operator (.). This dot operator is coded between the structure variable name and the structure member that we wish to access.
Before the dot operator, there must always be an already defined structure variable and after the dot operator, there must always be a valid structure element.

```
#include <iostream>
using namespace std;

        struct employee
        {
             /* data */
             int eId;
             char favChar;
             int salary;
         };

 int main()
 {
        struct employee Yoku;
        Yoku.eId = 1;
       Yoku.favChar = 'c';
       Yoku.salary = 120000000;
       cout << "eID of Yokshita is " << Yoku.eId << endl;
       cout<<"favChar of Yokshita is "<<Yoku.favChar<< endl;
       cout << "salary of Yokshita is " << Yoku.salary << endl;
       return 0;
 }
```

Unions

Just like Structures, the union is a user-defined data type. They provide better memory management than structures. All the members in the unions share the same memory location.

The union is a data type that allows different data belonging to different data types to be stored in the same memory locations. One of the advantages of using a union over structures is that it provides an efficient way of reusing the memory location, as only one of its members can be accessed at a time. A union is used in the same way we declare and use a structure. The difference lies just in the way memory is allocated to their members.

Creating a Union element

```
#include <iostream>
using namespace std;

union money
{
    /* data */
    int rice;
    char car;
    float pounds;
};

int main()
{
    union money m1;
    m1.rice = 34;
    cout << m1.rice;
    return 0;
}
```

Enums

Enum or enumeration is a user-defined data type. Enums have named constants that represent integral values. Enums are used to make the program more readable and less complex. It lets us define a fixed set of possible values and later define variables having one of those values.

```
#include <iostream>
 using namespace std;

 enum Meal
 {
       breakfast,
       lunch,
       dinner
  };

 int main()
 {
        Meal m1 = dinner;
        if (m1 == 2)
 {
      cout << "The value of dinner is " << dinner << endl;
  }
 }
```

Functions

Functions are the main part of top-down structured programming. We break the code into small pieces and make functions of that code. Functions could be called multiple or several times to provide reusability and modularity to the C++ program.

Functions are also called procedures or subroutines or methods and they are often defined to perform a specific task. And that makes functions a group of code put together and given a name that can be called anytime without writing the whole code again and again in a program.

Advantages of Functions

1. The use of functions allows us to avoid re-writing the same logic or code over and over again.
2. With the help of functions, we can divide the work among the programmers.
3. We can easily debug or can find bugs in any program using functions.
4. They make code readable and less complex.

Aspects of a function

Declaration — THIS IS WHERE A FUNCTION IS DECLARED TO TELL THE COMPILER ABOUT ITS EXISTENCE.

Definition — A FUNCTION IS DEFINED TO GET SOME TASK EXECUTED. (IT MEANS WHEN WE DEFINE A FUNCTION, WE WRITE THE WHOLE CODE OF THAT FUNCTION AND THIS IS WHERE THE ACTUAL IMPLEMENTATION OF THE FUNCTION IS DONE).

Call — THIS IS WHERE A FUNCTION IS CALLED IN ORDER TO BE USED.

Function Prototype in C++

THE FUNCTION PROTOTYPE IS THE TEMPLATE OF THE FUNCTION WHICH TELLS THE DETAILS OF THE FUNCTION WHICH INCLUDE ITS NAME AND PARAMETERS TO THE COMPILER. FUNCTION PROTOTYPES HELP US TO DEFINE A FUNCTION AFTER THE FUNCTION CALL.

```
// FUNCTION PROTOTYPE
RETURN_DATATYPE FUNCTION_NAME(DATATYPE_1 A,
DATATYPE_2 B);
```

Types of functions

Library functions

LIBRARY FUNCTIONS ARE PRE-DEFINED FUNCTIONS IN C++ LANGUAGE. THESE ARE THE FUNCTIONS THAT ARE INCLUDED IN C++ HEADER FILES PRIOR TO ANY OTHER PART OF THE CODE IN ORDER TO BE USED.
E.G. SQRT(), ABS(), ETC.

User-defined functions

USER-DEFINED FUNCTIONS ARE FUNCTIONS CREATED BY THE PROGRAMMER FOR THE REDUCTION OF THE COMPLEXITY OF A PROGRAM. RATHER, THESE ARE FUNCTIONS THAT THE USER CREATES AS PER THE REQUIREMENTS OF A PROGRAM.
E.G. ANY FUNCTION CREATED BY THE PROGRAMMER.

FUNCTIONS PARAMETERS

A function receives information that is passed to them as a parameter. Parameters act as variables inside the function. Parameters are specified collectively inside the parentheses after the function name. parameters inside the parentheses are comma separated.

We have different names for different parameters.

FORMAL PARAMETERS

THE VARIABLE WHICH IS DECLARED IN THE FUNCTION IS CALLED A FORMAL PARAMETER OR SIMPLY, A PARAMETER. FOR EXAMPLE, VARIABLES A AND B ARE FORMAL PARAMETERS.

```
INT SUM(INT A, INT B){
    //FUNCTION BODY
    }
```

ACTUAL PARAMETERS

THE VALUES WHICH ARE PASSED TO THE FUNCTION ARE CALLED ACTUAL PARAMETERS OR SIMPLY, ARGUMENTS. FOR EXAMPLE, THE VALUES NUM1 AND NUM2 ARE ARGUMENTS.

```
INT SUM(INT A, INT B);

INT MAIN()
    {
INT NUM1 = 5;
INT NUM2 = 6;
 SUM(NUM1,
NUM2);//ACTUAL
PARAMETERS
    }
```

METHODS

NOW, THERE ARE METHODS USING WHICH ARGUMENTS ARE SENT TO THE FUNCTION. THEY ARE,

CALL BY POINTER IN C++

A call by the pointer is a method in C++ to pass the values to the function arguments. In the case of call by pointer, the address of actual parameters is sent to the formal parameter, which means that if we change the values inside the function that will affect the actual values.

```
#INCLUDE <IOSTREAM>
USING NAMESPACE STD;

VOID SWAP(INT *A, INT *B)
{
INT TEMP = *A;
*A = *B;
*B = TEMP;
}

INT MAIN()
{
INT X = 5, Y = 6;
COUT << "THE VALUE OF X IS " << X << " AND
THE VALUE OF Y IS " << Y << ENDL;
SWAP(&X, &Y);
COUT << "THE VALUE OF X IS " << X << " AND
THE VALUE OF Y IS " << Y << ENDL;
}
```

OUTPUT

```
THE VALUE OF X IS 5 AND THE VALUE OF Y IS 6
THE VALUE OF X IS 6 AND THE VALUE OF Y IS 5
}
```

CALL BY REFERENCE IN C++

Call by reference is a method in C++ to pass the values to the function arguments. In the case of call by reference, the reference of actual parameters is sent to the formal parameter, which means that if we change the values inside the function that will affect the actual values.

```
#INCLUDE <IOSTREAM>
USING NAMESPACE STD;

VOID SWAP(INT &A, INT &B)
{
INT TEMP = A;
A = B;
B = TEMP;
}

INT MAIN()
{
INT X = 5, Y = 6;
COUT << "THE VALUE OF X IS " << X << " AND
THE VALUE OF Y IS " << Y << ENDL;
SWAP(X, Y);
COUT << "THE VALUE OF X IS " << X << " AND
THE VALUE OF Y IS " << Y << ENDL;
}
```

OUTPUT

```
THE VALUE OF X IS 5 AND THE VALUE OF Y IS 6
THE VALUE OF X IS 6 AND THE VALUE OF Y IS 5
```

DEFAULT ARGUMENTS IN C++

Default arguments are those values which are used by the function if we don't input our value as parameter. It is recommended to write default arguments after the other arguments.

```
int sum(int a = 5, int b);
```

CONSTANT ARGUMENTS IN C++

Constant arguments are used when you don't want your values to be changed or modified by the function. The const keyword is used to make the parameters non-modifiable.

```
int sum(const int a, int b);
```

Recursion

WHEN A FUNCTION CALLS ITSELF, IT IS CALLED RECURSION AND THE FUNCTION WHICH IS CALLING ITSELF IS CALLED A RECURSIVE FUNCTION. THE RECURSIVE FUNCTION CONSISTS OF A BASE CONDITION AND A RECURSIVE CONDITION.

RECURSIVE FUNCTIONS MUST BE DESIGNED WITH A BASE CASE TO MAKE SURE THE RECURSION STOPS, OTHERWISE, THEY ARE BOUND TO EXECUTE FOREVER AND THAT'S NOT WHAT YOU WANT. THE CASE IN WHICH THE FUNCTION DOESN'T RECUR IS CALLED THE BASE CASE. FOR EXAMPLE, WHEN WE TRY TO FIND THE FACTORIAL OF A NUMBER USING RECURSION, THE CASE WHEN OUR NUMBER BECOMES SMALLER THAN 1 IS THE BASE CASE.

```
int factorial(int n)
{
    if (n == 0 || n == 1)
    {
        return 1;
    }
    return n * factorial(n - 1);
}
```

FUNCTION OVERLOADING

FUNCTION OVERLOADING IS A PROCESS TO MAKE MORE THAN ONE FUNCTION WITH THE SAME NAME BUT DIFFERENT PARAMETERS, NUMBERS, OR SEQUENCES. NOW, THERE ARE A FEW CONDITIONS AND ANY NUMBER OF FUNCTIONS WITH THE SAME NAME FOLLOWING ANY OF THESE ARE CALLED OVERLOADED.

Same name but different data type of parameters	float sum(int a, int b); float sum(float a, float b);
Same name but a different number of parameters	float sum(int a, int b); float sum(int a, int b, int c);
Same name but different parameter sequence	float sum(int a, float b); float sum(float a, int b);