Flowcharts

INTRODUCTION

Here are the steps that may be followed to solve an algorithmic problem:

- Analysing the problem statement means making the objective of the program clear in our minds like what the input is and what is the required output.
- Sometimes the problems are of complex nature, to make them easier to understand, we can break-down the problem into smaller sub-parts.
- In order to save our time in debugging our code, we should first-of-all write down the solution on a paper with basic steps that would help us get a clear intuition of what we are going to do with the problem statement.
- In order to make the solution error-free, the next step is to verify the solution by checking it with a bunch of test cases.
- Now, we clearly know what we are going to do in the code. In this step we will start coding our solution on the compiler.

Basically, in order to structure our solution, we use flowcharts. A flowchart would be a diagrammatic representation of our algorithm - a step-by-step approach to solve our problem.

Flowcharts

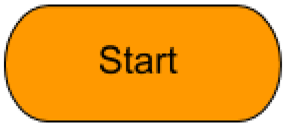
Uses of Flowcharts

- ➔ Used in documentation.
- ➔ Used to communicate one's solution with others, basically used for group projects.

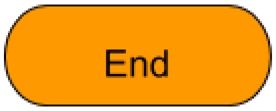
➔ To check out at any step what we are going to do and get a clear explanation of the flow of statements.

Flowchart components

- **Terminators**

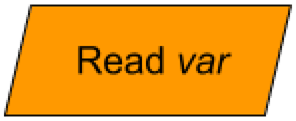


Mainly used to denote the start point of the program.

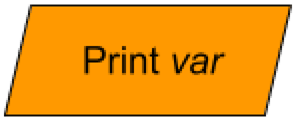


Used to denote the end point of the program.

- **Input/Output**

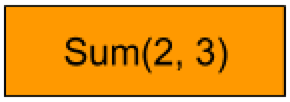


Used for taking input from the user and store it in variable 'var'.



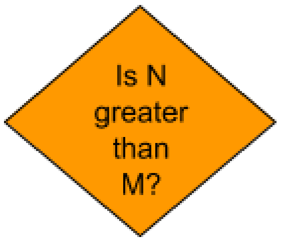
Used to output value stored in variable 'var'.

- **Process**



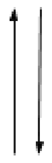
Used to perform the operation(s) in the program. For example: Sum(2, 3) just performs arithmetic summation of the numbers 2 and 3.

- **Decision**



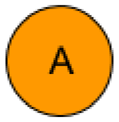
Used to make decision(s) in the program means it depends on some condition and answers in the form of TRUE(for yes) and FALSE(for no).

- **Arrows**



Generally, used to show the flow of the program from one step to another. The head of the arrow shows the next step and the tail shows the previous one.

- **Connector**

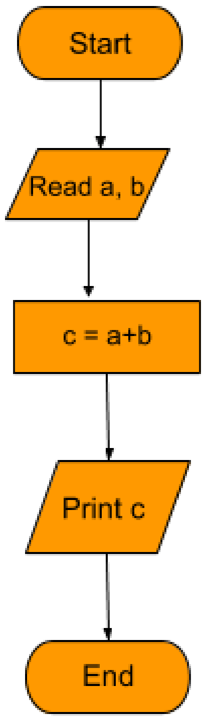


Used to connect different parts of the program and are used in case of break-through. Generally, used for functions(which we will study in our further sections).

Example 1:

Suppose we have to make a flowchart for adding 2 numbers a and b.

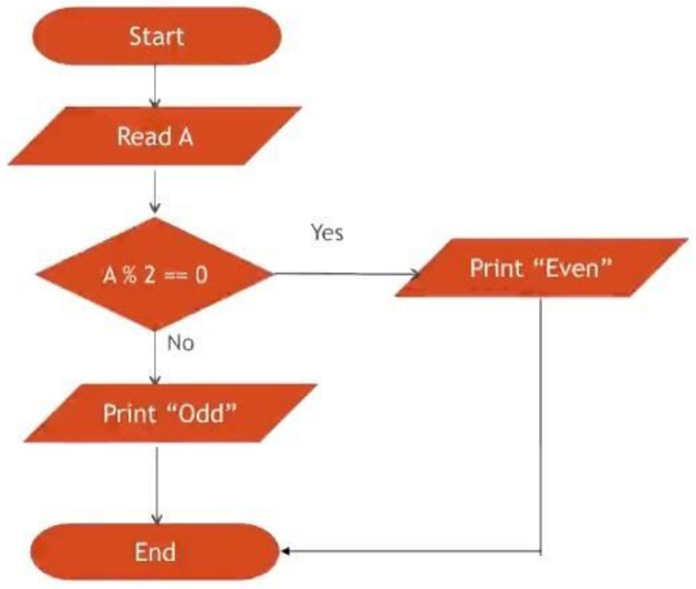
Solution:



Example 2:

Suppose we have to check if a number is even or odd.

Solution:



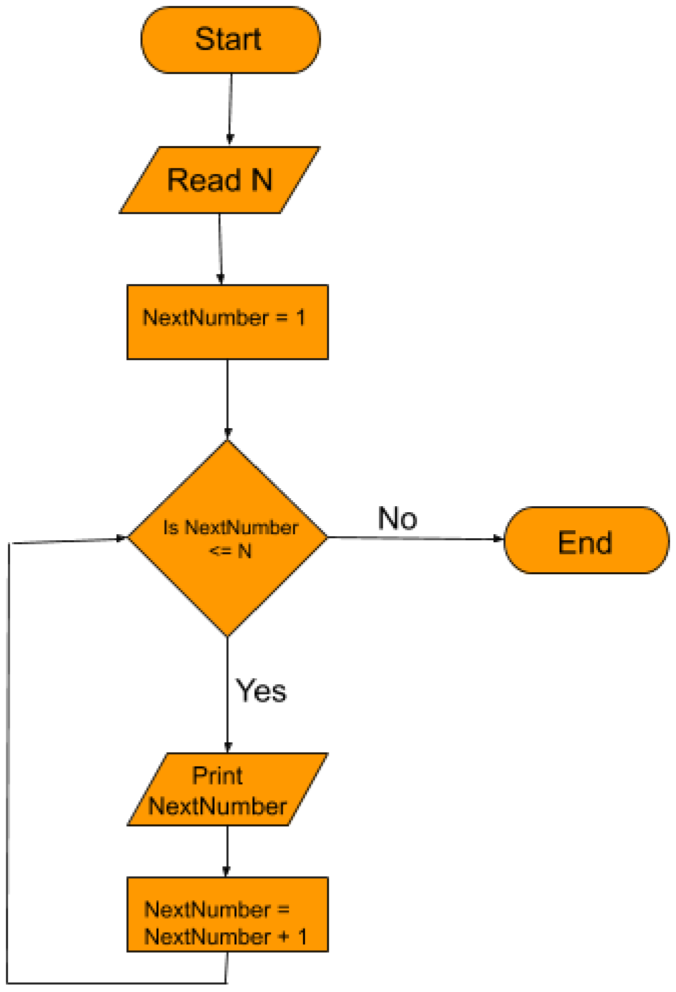
<u>Note:</u> Operator % is used to find the remainder of a number with respect to another number. <u>For example:</u>

- 4 % 3 = 1
- 10 % 5 = 0
- 4 % 2 = 0
- 4 % 4 = 0
- 0 % 1 = 0
- 1 % 0 = undefined (as it leads to division by 0)

Example 3:

To print the numbers less than or equal to n where n is given by the user.

Solution:



Summary

- Flowcharts are the building-block of any program written in any language.
- Different shapes used to have different meanings.
- Every problem can be represented in the form of a flow chart.
- Sometimes, it becomes a bulky process to represent any program using flowchart. In those cases, try to find out the optimal solution to the given problem.

Getting Started

For this course, you will be provided with the in-built compiler of Coding Ninjas. However, if you want to run programs and practice them on your local desktop, there are various compilers out there like Code blocks, VS Code, Dev C++, Atom and many more. We have provided the steps for installing Code Blocks in a separate file.

About Code blocks

Code blocks is an Integrated Development Environment (IDE) for C/C++. To set the path of compiler, follow the given steps:

1. Click on menu **Settings** -> **Compiler.**
2. Then click on the tab **Toolchain Executables.**
3. In the text box under **Compiler's installation directly,** click on the button **Auto Detect**. A pop-up appears.
4. In case, pop-up says, **Couldn't auto detect...,** that means you have downloaded the incorrect setup of code blocks. Uninstall this setup, and install the setup with **MinGW**. Then repeat the above steps.

To create a new file:

1. Follow **File -> New -> Empty File.**
2. Then save that file with extension **.cpp**
3. In order to run and compile the program press F11 or click on the button right next to a play button which says **Build and Run**.

You will get your output window after following step 3.

Looking at the code

C++ code begins with the inclusion of header files. There are many header files available in the C++ programming language which you will discuss while moving ahead with the course.

So, what are these header files?

The names of program elements such as variables, functions, classes, and so on must be declared before they can be used. For example, you can't just write x = 42 without first declaring variable 'x' as:

```
int x = 42;
```

The declaration tells the compiler whether the element is an int, a double, a float, a function or a class. Similarly, header files allow us to put declarations in one location and then import them wherever we need them. This can save a lot of typing in multi-file programs. To declare a header file, we use **#include** directive in every .cpp file. This #include is used to ensure that they are not inserted multiple times into a single .cpp file.

Note: # operator is known as **Macros.**

The following are not allowed, or are considered as bad practices while putting header files into the code:

- Built-in-type definitions at namespace or global scope
- non-inline function definitions
- Non-const variable definitions
- Aggregate definitions
- Unnamed namespaces
- Using directives

Now, moving forward to the code:

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

iostream stands for Input/Output stream, meaning this header file is necessary if you want to take input through the user or print output to the screen. This header file contains the definitions for the functions:

- **cin** : used to take input
- **cout** : used to print output

namespace defines which input/output form is to be used. You will understand these better as you progress in the course.

Note: semicolon (;) is used for terminating a C++ statement. ie. different statements in a C++ program are separated by semicolon

main() function:

Look at the following piece of code:

```
int main() {

    Statement 1;
    Statement 2;
    ...
}
```

You can see the highlighted portion above. Let's discuss each portion stepwise.

Starting with the line:

```
int main()
```

- **int** : This is the return-type of the function. You will get this thing clear once you reach the **Functions** topic.

- **main()** : This is the portion of any C++ code inside which all the commands are written and gets executed.
    - This is the line at which the program will begin executing. This statement is similar to the start block of flowcharts.
    - As you will move further in the course, you will get a clear glimpse of what this function is. Till then, just note that you will have to write all the programs inside this block.
- **{}** : all the code written inside the curly braces is said to be in one block, also known as scope of a particular function. Again, these things will be clear when you will study functions.

For now, just understand that this is the format in which we are gonna write our basic C++ code. From time to time as you will move forward with the course, you will get a clear and better understanding.

Declaring a variable:

To declare a variable, we should always know what type of value it should hold, whether it's an integer (int), decimal number (float, double), character value (char). In general, the variable is declared as follows:

```
Datatype variableName = VALUE;
```

Note: Datatype is the type of variable:

- int : Integer value
- float, double : Decimal number
- char : Character values (including special characters)
- bool : Boolean values (true or false)
- long : Contains integer values but with larger size
- short : Contains integer values but with smaller size

Table for datatype and its size in C++: (This can vary from compiler to compiler and system to system depending on the version you are using)

Datatype	Default size
bool	1 byte
char	1 byte
short	2 bytes
int	4 bytes
long	8 bytes
float	4 bytes
double	8 bytes

For example: To declare an integer variable ‘a’ with a value of 5, the structure looks like:

```
int a = 5;
```

Similarly, this way other types of variables can also be declared. There is one more type of variable known as **string** variables which store combinations of characters. You will study that in your further lectures.

Rules for variable names:

- Can’t begin with a number.
- Spaces and special characters except underscore(_) are not allowed.
- C++ keywords (reserved words) must not be used as a variable name.
- C++ is case-sensitive, meaning a variable with name ‘A’ is different from variable with name ‘a’. (Difference in the upper-case and lower-case holds true)

Printing/Providing output:

For printing statements in C++ programs, we use the **cout** statement**.**

For example: If you want to print “Hello World!” (without parenthesis) in your code, we will write it in following way:

```
cout << “Hello World!”;
```

A full view of the basic C++ program is given below for the above example:

Code:

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

int main() {
    cout<<”Hello World!”;
}
```

Output:

```
Hello World!
```

Line separator:

For separating different lines in C++, we use **endl** or **‘\n’**.

For example:

Code:

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

int main() {
    cout<<”Hello World1”<<endl;
    cout<<”Hello World2”<<’\n’;

}
```

Output:

```
Hello World1
Hello World2
```

Taking input from the user:

To take input from the user, we use the **cin** statement.

For example: If you want to input a number from the user:

```
int n;
cin >> n;
```

A full view of the basic C++ program in which you want to take input of 2 numbers and then print the sum of them:

Code:

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

int main() {
    int a, b, sum;
    cin >> a;
    cin >> b;
    sum = a + b;
    cout << “Sum of two numbers: ”;
    cout << sum << endl;
}
```

Input:

```
1
2
```

Output:

```
Sum of two numbers: 3
```

Operators in C++

There are 3 types of operators in C++

- Arithmetic operators
- Relational operators
- Logical operators

Discussing each of them in detail...

Arithmetic operators:

These are used in mathematical operations in the same way as that in algebra.

OPERATOR	DESCRIPTION
+	Add two operands
-	Subtracts second operand from the first
*	Multiplies two operands
/	Divides numerator by denominator
%	Calculates Remainder of division

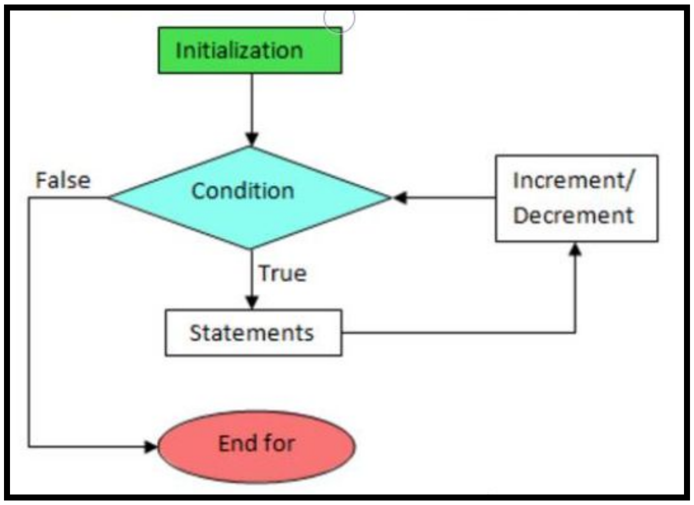
Relational operators:

C++ relational operators specify the relation between two variables by comparing them. Following table shows the relational operators that are supported by C++.

OPERATOR	DESCRIPTION
==	Checks if two operands are equal
!=	Checks if two operands are not equal
>	Checks if operand on the left is greater than operand on the right
<	Checks if operand on the left is lesser than operand on the right
>=	Checks if operand on the left is greater than or equal to operand on the right
<=	Checks if operand on the left is lesser than or equal to operand on the right

Logical operators:

C++ supports 3 types of logical operators. The result of these operators is a boolean value i.e., True(BITWISE '1') or False(BITWISE '0'). Refer the visual representation below:



OPERATOR	DESCRIPTION

&&	Logical AND
||	Logical OR
!	Logical NOT

How is data Stored ?

- **For integers:**

The most commonly used is a signed 32-bit integer type. When you store an integer, its corresponding binary value is stored. There is a separate way of storing positive and negative numbers. For positive numbers the integral value is simply converted into binary value while for negative numbers their 2's complement form is stored.

For negative numbers:

Computers use 2's complement in representing signed integers because:

- There is only one representation for the number zero in 2's complement, instead of two representations in sign-magnitude and 1's complement.
- Positive and negative integers can be treated together in addition and subtraction. Subtraction can be carried out using the **addition logic.**

For example: int i = -4;

Steps to calculate Two's Complement of -4 are as follows:

Step1: Take Binary Equivalent of the positive value (4 in this case)

```
0000 0000 0000 0000 0000 0000 0000 0100
```

Step2: Write 1's complement of binary representation by inverting the bits

```
1111 1111 1111 1111 1111 1111 1111 1011
```

Step3: Find 2's complement by adding 1 to the corresponding 1's complement

```
 1111 1111 1111 1111 1111 1111 1111 1011
+0000 0000 0000 0000 0000 0000 0000 0001
-----------------------------------------------------------------
 1111 1111 1111 1111 1111 1111 1111 1100
```

Thus, integer -4 is represented by the above binary sequence in C++.

- ## For float and double values:

  In C++, any value declared with a decimal point is by-default of type double (which is generally of 8-bytes). If we want to assign a float value (which is generally of 4-bytes), then we must use 'f' or 'F' literal to specify that the current value is "float".

  For example:

```
float var = 10.4f          // float value
Double val = 10.4          // double value
```

- ## For character values:

  Every character has a unique integer value, which is called ASCII value. As we know, systems only understand binary language and thus everything has to be stored in the form of binaries. So, for every character there is a corresponding integer code-ASCII code and the binary equivalent of this code is actually stored in memory when we try to store a character.

  For ASCII values, refer the link below:

  https://ascii.cl/

- ## Adding int to char

When we add int to char, we are basically adding two numbers i.e., one corresponding to the int and the other corresponding to the ASCII code for the character.

For example:

Code:

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

int main () {
    cout<< 'a' + 1;
}
```

Output:

```
98
```

Explanation:

The **ASCII value of 'a' is 97**, so it printed **97+1 = 98** as the output.

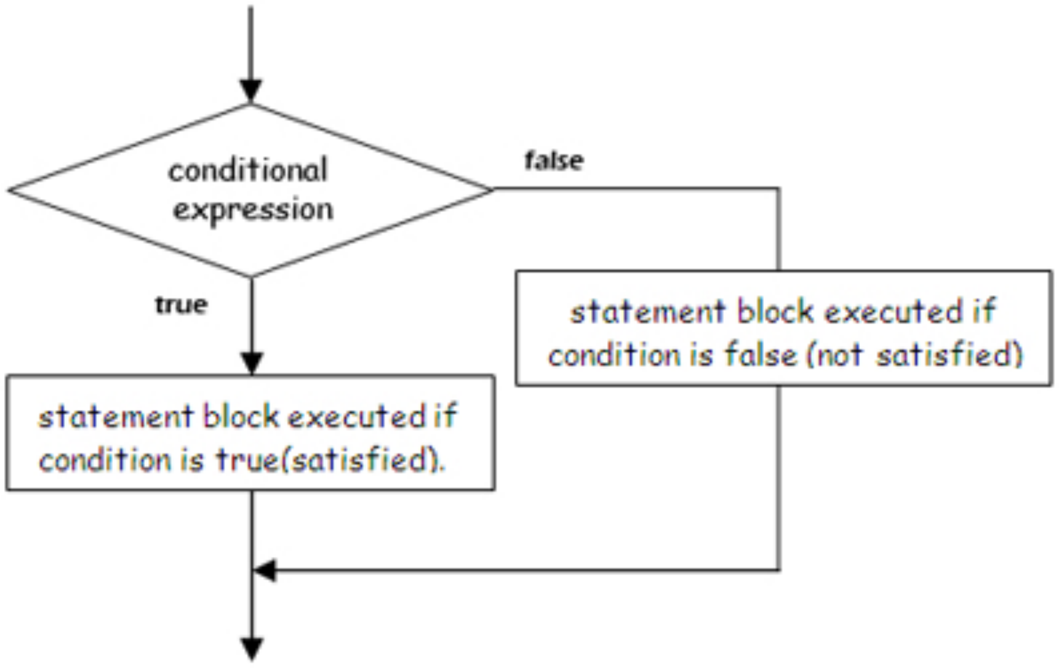
C++ Foundation with Data Structures

Lecture 3 : Conditionals and Loops

Conditional Statements (if else)

Description

Conditionals are used to execute a certain section of code only if some specific condition is fulfilled, and optionally execute other statements if the given condition is false. The result of given conditional expression must be either true or false.


conditional expression
false
true
statement block executed if condition is false (not satisfied)
statement block executed if condition is true(satisfied).

Different variations of this conditional statement are –

- **if statement**
  if statement evaluates the given test expression. If it is evaluated to true, then statements inside the if block will be executed. Otherwise, statements inside if block is skipped.

  Syntax
  ```
  if(test_expression) {
          // Statements to be executed only when test_expression is true
  }
  ```

  Example Code:

```
int main ()
{
        int n=5;
        if ( n<10 )
        {
                cout << "Inside if statement" << endl;
        }
        cout << "Outside if statement" << endl;
}
```

Output

```
Inside if statement
Outside if statement
```

So if the condition given inside if parenthesis is true, then statements inside if block are executed first and then rest of the code. And if the condition evaluates to false, then statements inside if block will be skipped.

- **If – else statement**

  if statement evaluates the given test expression. If it is evaluated to true, then statements inside the if block will be executed. Otherwise, statements inside else block will be executed. After that, rest of the statements will be executed normally.

Syntax

```
if(test_expression) {
        // Statements to be executed when test_expression is true

}
else {
        // Statements to be executed when test_expression is false
}
```

Example Code:

```
int main () {
        int a=10,b=20;
        if (a > b){
                cout<< "a is bigger" <<endl;
```

```
		}
		else {
			cout<< "b is bigger" <<endl;
		}
	}
```
Output

```
b is bigger
```

- **if – else – if**

  Using this we can execute statements based on multiple conditions.

  Syntax

```
	if(test_expression_1) {
		// Statements to be executed only when test_expression_1 is true
	}
	else if(test_expression_2) {
		// Statements to be executed only when test_expression_1 is false and test_expression_2 is true
	}
	else if(test_expression_2) {
		// Statements to be executed only when test_expression_1 & test_expression_2 are false and test_expression_3 is true
	}
	....
	....
	else {
		// Statements to be executed only when all the above test expressions are false
	}
```

Out of all block of statements, only one will be executed based on the given test expression, all others will be skipped. As soon as any expression evaluates to ttue, that block of statement will be executed and rest will be skipped. If none of the expression evaluates to true, then the statements inside else will be executed.

Example Code:

```
int main() {
        int a = 5;
        if(a < 3) {
                cout<<"one"<< endl;
        }
        else if(a < 10) {
                cout<<"two"<< endl;
        }
        else if(a < 20) {
                cout<<"three"<< endl;
        }
        else {
                cout<<"four"<< endl;
        }
}
```

Output :

```
two
```

- **Nested if statement**

  We can put another if – else statement inside an if.

  Syntax

```
if(test_expression_1) {
        // Statements to be executed when test_expression_1 is true
        if(test_expression_2) {
                // Statements to be executed when test_expression_2 is
true
        }
        else {
                // Statements to be executed when test_expression_2 is
false
        }
}
```

Example Code:

```
int main() {
        int a = 15;
        if(a > 10) {
                if(a > 20) {
                        cout<<"Hello"<<endl;
                }
                else {
                        cout<<"Hi"<<endl;
                }
        }
}

Output :
Hi
```

return keyword

return is a special keyword, when encountered ends the main. That means, no statement will be executed after return statement. We'll study about return in more detail when we'll study functions.

Example Code:

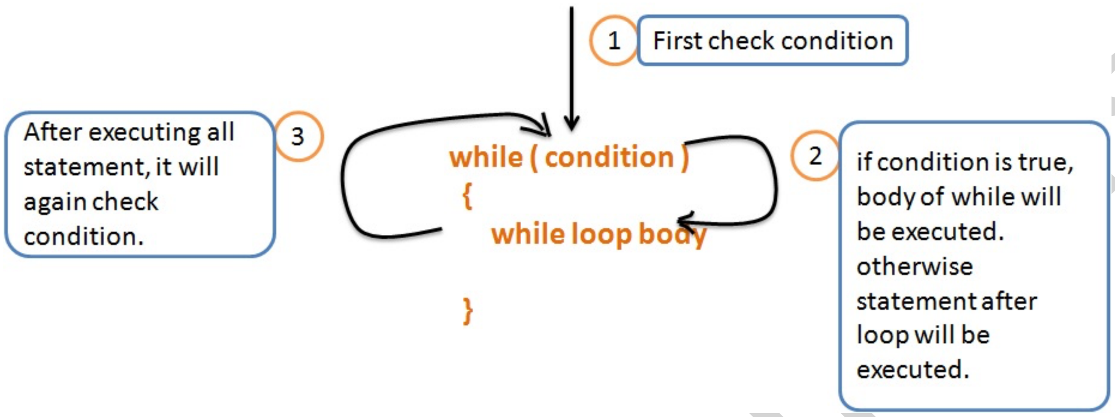
```
int main() {
                int a = 10;
                if(a > 5) {
                        cout<<Hello"<<endl;
                        return 0;
                }
                cout<<"Hi"<<endl;
}

Output:
Hello
```

while loop

Loop statements allows us to execute a block of statements several number of times depending on certain condition. **while** is one kind of loop that we can use. When executing, if the *test expression* result is true, then the actions inside the loop will be executed. This will continue as long as the expression result is true.

Syntax

```
while(test_expression) {
        // Statements to be executed till test_expression is true
}
```


1
First check condition
3
After executing all statement, it will again check condition.
while (condition)
{
while loop body
}
2
if condition is true, body of while will be executed. otherwise statement after loop will be executed.

Example Code:

```
int main() {
                int i = 1;
                while(i <= 5) {
                        cout<<i<<endl;
                        i++;
                }
}
```

Output:

```
1
2
3
4
5
```

In while loop, first given test expression will be checked. If that evaluates to be true, then the statements inside while will be executed. After that, the condition will be checked again and the process continues till the given condition becomes false.

Patterns

Introduction

Patterns are a good application of loops. These will help you get yourself a better clarity over implementing loops.

Suppose, we want to print the following pattern:

```
1
1 2
1 2 3
1 2 3 4
```

Solution:

- ➔ Here, you can see that we have 4 rows.
- ➔ Now, moving to find the generic number of columns, it's clear that the i-th row (where i = 1, 2, ...) has i columns. Hence, the number of columns for i-th row is i.
- ➔ Visually, it's also clear that we want to print numbers starting from 1 to the row-th number. Like for the first row, we print numbers starting from 1 and ending at 1, for second row numbers are starting from 1 and ending at 2 and so on...

The above approach can be generalised in the following way:

- We start to figure out the number of rows that our pattern requires.
- Now, we should know how many columns do we have to print in the generic i-th row.

- Once, we have figured out the number of rows and columns, then we should focus on what to print.

There are two popular types of patterns-related questions that are usually posed:

- Square Pattern
- Triangular Pattern

Coding a pattern:

Now, starting to code this pattern. It's clear that we have to start with a loop for running over the row. So let's first begin with writing the basic structure and the loop for the row:

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

int main() {

        int n = 4;   //Number of rows taken input

        int i = 1;   // for iterating over rows starting from the first row

        while(i < n) {

                i++;     // for iterating over each row
        }
}
```

Here, we have iterated over each row.

Now for printing the above pattern, we also need to iterate over columns and that too, on each row.

For this, we will run another loop inside the while() loop. Let’s see that also…

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

int main() {

        int n = 4;                    //Number of rows taken input

        int i = 1;                    // for iterating over rows starting from the first row

        while(i < n) {

                int j = 1;             // for iterating over columns
                while(j <= i) {        // since for each column row, we will be iterating
                                       //  over each column
                        cout << j;
                        j = j + 1;     // as we have discussed above that i-th row will
                                       // have i columns
                }
                cout << endl;
                i++;                   // for iterating over each row
        }
}
```

This will result in the above pattern that we discussed. These types of patterns are known as **triangular patterns** as the shape resembles a triangle.

Similarly, there are other patterns too like:

```
11111
11111
11111
11111
11111
```

This type of pattern is **square Pattern.** Try coding it out yourself…

Once, you have tried it, you can check your solution with the one given below:

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

int main() {

	int n = 5;

	int i = 0;
	while(i <= n) {

		int j = 1;
		while(j <= n) {
			cout << 1;
			j = j + 1;
		}
		cout << endl;
		i = i + 1;
	}
}
```

Like these integral valued patterns there are character valued patterns too. The only difference is that here we are printing the character values.

For example:

Given below are three patterns of the characters. They are very similar to what we discussed above.

****** ****** ****** ****** ******	abcde abcde abcde abcde abcde	A AB ABC ABCD ABCDE

Practice Examples:

Print the following patterns:

1)

```
55555
45555
34555
23455
12345
```

2)

```
ABCDE
ABCD
ABC
AB
A
```

3)

```
12344321
123**321
12****21
1******1
```

Try these yourselves and in case, you find yourself stuck then following is the approach provided...

Solution 1

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

int main() {

    int i = 5, j, k;
    while(i >= 1) {
            k = i;
            j = 1;
            while(j <= 5) {
                    if(k <= 5) {
                            cout << k;
                    }
                    else {
                            cout << 5;
                    }
                    k = k + 1;
                    j = j + 1;
            }
            cout << endl;
            i = i - 1;
    }
}
```

Solution 2

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

int main() {

            int i = 1, j;
            while(i <= 5) {
                    j = 5 - i + 1;
                    int k = 1;
                    while(k <= j) {
                            cout<<(char)(64 + k);
                            k = k + 1;
                    }
                    cout << endl;
                    i = i + 1;
            }
}
```

Solution 3

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

int main() {

            int i = 4, j;
            while(i >= 1) {

                        j = 1;
                        while(j <= 4) {
                                    if(j <= i) {
                                            cout << j;
                                    }
                                    else {
                                            cout << "*";
                                    }
                                    j = j + 1;
                        }

                        while(j >= 1) {
                                    if(j <= i) {
                                            cout << j;
                                    }
                                    else {
                                            cout << "*";
                                    }
                                    j = j - 1;
                        }
                        i = i - 1;
                        cout << endl;
            }
}
```

Advanced patterns

Here, we are gonna study the patterns that include a form of mirror images at some part of the program. We'll be solving these like we did in the "Patterns 1" portion by following the same three steps of identifying the number of rows, then identifying the columns and finally what to print.

So directly moving on to programming questions:

Example 1: Print the following pattern:

```
  1  2  3  4 5
 16          6
 15          7
 14          8
 13 12 11 10 9
```

Try this out, in case you are stuck, just checkout the code below for the same.

Note: The pattern may not be totally similar, meaning it may be a bit shifted due to the difference in the number of digits of a 2-digit number and a 1-digit number.

Solution:

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

int main() {

   int i = 1, k = 6, l = 13, m = 16;
         while(i <= 5)
         {
                  int j = 1;
                 while(j <= 5)
                 {
                           if(i == 1)
                                  cout << j <<" ";
                            else if(j == 5)
                                  cout << k++ << " ";
                           else if(i == 5)
```

```
                                                cout << i-- << " ";
                                        else if(j == 1)
                                                cout << m-- << " ";
                                        else
                                                cout<< "  ";
                                        j++;
                        }
                        cout << endl;
                        i++;
            }
}
```

Example 2: Print the following pattern:

```
    1
   123
  12345
 1234567
123456789
 1234567
  12345
   123
    1
```

Try this out, in case you are stuck, just checkout the code below for the same.

Hint: Here you can see that there are generally two different patterns that you have to make: one spreading in the downwards directions and other one in continuation, shrinking towards the bottom.

Approach: Let's first try to make the downwards growing pattern i.e., upto row number 5. It is clearly visible that each row has 2n-1 numbers (where n = 1, 2, 3, 4, 5). Now, as we know the number of rows, columns in each row and the general formula that our pattern follows, it can be easily coded.

Moving on to the lower triangle of the pattern, if we start counting these rows in reverse order starting from number 4 down to number 1, each row contains 2n-1 numbers over here also, and can be similarly done using the upper triangle pattern but in reverse order.

Solution:

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

int main() {
   int i = 1;
   while (i <= 5)                  // Starting our first part of the pattern.
   {
         int j = i;
         while (j < 5)
         {
                 cout<<" ";
                 j++;
         }
         int k = 1;
         while(k < 2*i)            // upto k = 2n-1
         {
                cout << k;
                k++;
         }
         i++;
         cout << endl;
   }

   i = 4;                         // Starting our second pattern in reverse order.
   while (i > 0)
   {
         int j = 5;
         while (j > i)
         {
                 cout << " ";
                 j--;
         }
         int k = 1;
         while (k < 2*i)
          {
                   cout << k;
                   k++;
          }
         cout << endl;
         i--;
   }
}
```

You have already practised most of these questions earlier. So now directly moving on to some of the practice questions.

PRACTICE: Print the following programs

```
1    1
 2  2
  3 3
   4
  3 3
 2  2
1    1
```

Solutions:

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

int main()
{
   int i = 1, k=1;
   int m[7][7]={0};       // Initialises all the elements of the array equal to 0, you will
                          //   study about arrays in your further sessions.
   while (i <= 7)
   {
         int j = 1;
         while (j <= 7)
         {
               if (j == i || 8-i == j)
                     m[i-1][j-1]=k;
               j++;
         }
         if (i < 4)
               k++;
         else
               --k;
         i++;
   }

   i = 0;
   while (i < 7)
```

```
    {
         int j = 0;

         while (j < 7)
         {
                 if(m[i][j]==0)
                         cout << " ";
                 else
                         cout << m[i][j];
                 j++;
         }
         i++;
         cout << endl;
    }
}
```

For more questions, visit the link:

http://cbasicprogram.blogspot.com/2012/04/number-patterns.html

C++ Foundation with Data Structures

Lecture 4 : Loops, Keywords, Associativity and Precedence

for loop

Loop statements allows us to execute a block of statements several number of times depending on certain condition. **for** loop is kind of loop in which we give initialization statement, test expression and update statement can be written in one line.

Inside for, three statements are written –
a. Initialization – used to initialize your loop control variables. This statement is executed first and only once.
b. Test condition – this condition is checked everytime we enter the loop. Statements inside the loop are executed till this condition evaluates to true. As soon as condition evaluates to false, loop terminates and then first statement after for loop will be executed next.
c. Updation – this statement updates the loop control variable after every execution of statements inside loop. After updation, again test conditon is checked. If that comes true, the loop executes and process repeats. And if condition is false, the loop terminates.

```
for (initializationStatement; test_expression; updateStatement) {
        // Statements to be executed till test_expression is true
}
```

Example Code :

```
int main(){
        for(int i = 0; i < 3; i++){
            cout<<"Inside for Loop : "<<i<<endl;
        }
        cout<<"Done";
}
```

Output:

```
Inside for Loop : 0
Inside for Loop : 1
Inside for Loop : 2
Done
```

In for loop its not compulsory to write all three statements i.e. initializationStatement, test_expression and updateStatement. We can skip one or more of them (even all three)

Above code can be written as:

```
int main () {
        int i=1;    // initialization is done outside the for loop
        for (; i <= 5; i++) {
                cout<<i<<end;
        }
}
```

OR

```
int main () {
        int i=1;    //// initialization is done outside the for loop
        for (; i <= 5; ) {
                cout<<i<<end;
                i++;   // updateStatement written here
        }
}
```

We can also skip the test_expression. See the example below :

Variations of for loop

- The three expressions inside for loop are optional. That means, they can be omitted as per requirement.

  Example code 1: Initialization part removed –

```
int main(){
       int i = 0;
       for(; i < 3; i++){
              cout<<i<<endl;
       }
}
```

Output:

```
0
1
2
```

Example code 2: Updation part removed

```
int main(){
        for(int i = 0; i < 3; ){
                cout<<i<<endl;
                 i++;
    }
}
```

Output:

```
0
1
2
```

Example code 3: Condition expression removed , thus making our loop infinite –

```
int main(){
        for(int i = 0 ; ;i++){
              cout<<i<<endl;
        }
}
```

Example code 4:
We can remove all the three expression, thus forming an infinite loop-

```
#include<iostream>
using namespace std;
int main(){
        for(; ;){
              cout<<"Inside for loop : "<<endl;
        }
}
```

- **Multiple statements inside for loop**

We can initialize multiple variables, have multiple conditions and multiple update statements inside a *for loop*. We can separate multiple statements using comma, but not for conditions. They need to be combined using logical operators.

Example code:

```
int main(){
        for(int i = 0,j = 4; i < 5 && j >= 0; i++, j--){
              cout<<i<<" "<<j<<endl;
        }
}
```

Output:

```
0 4
1 3
2 2
3 1
4 0
```

break and continue

1. **break statement**: The break statement terminates the loop (for, while and do. while loop) immediately when it is encountered. As soon as break is encountered inside a loop, the loop terminates immediately. Hence the statement after loop will be executed next.
2. **continue statement:** The continue statement skips some statements inside the loop. The continue statement is used with decision making statement such as if...else. (caution always update the counter in case of while loop else loop will never end)

```
while(test_expression) {
        // codes
        if (condition for break) {
                break;
        }
        //codes
}
```

```
for (initializationStatement; test_expression; updateStatement) {
        // codes
        if (condition for break) {
                break;
        }
        //codes
}
```

❖ **break**

- Example: (using break inside for loop)

```
int main () {
                for (int i=1; i <= 10; i++) {
                        cout<<i<<end;
                        if(i==5)
                        {
                                break;
                        }
                }
}
```

```
Output:
1
2
3
4
5
```

- Example: (using while loop)

```
int main () {
                int i = 1;
                while (i <= 10) {
                        cout<<i<<endl;
                        if(i==5)
                        {
                                break;
                        }
                        i++;
                }
}
```

Output:

```
1
2
3
4
5
```

- ## Inner loop break:

When there are two more loops inside one another. Break from innermost loop will just exit that loop.

Example Code 1:

```
int main () {
        for (int i=1; i <=3; i++) {
                cout<<i<<end;
                for (int j=1; j<= 5; j++)
                {
                        cout<<"in..." <<endl;
                        if(j==1)
                        {
                                break;
                        }
                }
        }
}
```

Output:

```
1
in…
2
in…
3
in…
```

Example Code 2:

```
int main () {
        int i=1;
        while (i <=3) {
                cout<<i<<end;
                int j=1;
                while (j <= 5)
                {
                        cout<<"in…" <<endl;
                        if(j==1)
                        {
                                break;
                        }
                        j++;
                 }
                i++;
        }
}
```

```
Output:
1
in…
2
in…
3
in…
```

❖ Continue

The **continue** keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loopx.

- Example: (using for loop)

```
int main () {
        for (int i=1; i <= 5; i++) {
                if(i==3)
                {
                        continue;
                }
                cout<<i<<end;
        }
}

Output:
1
2
4
5
```

- **Example: (using while loop)**

```
int main () {
        int i=1;
        while (i <= 5) {
                if(i==3)
                {
                        i++;
                   // if increment isn't done here then loop  will run
        infinite time for i=3
                        continue;
                }
                cout<<i<<end;
                i++;
        }
}
```

Output:

```
1
2
4
5
```

Scope of variables

Scope of variables is the curly brackets {} inside which they are defined. Outside which they aren't known to the compiler. Same is for all loops and conditional statement (if).

❖ Scope of variable - for loop

```
for (initializationStatement; test_expression; updateStatement) {
        // Scope of variable defined in loop
}
```

Example:

```
int main() {
        for (int i=0; i<5; i++) {
                int j=2;        // Scope of i and j are both inside the loop they
                can't be used outside
}
```

❖ Scope of variable for while loop

```
while(test_expression) {
        // Scope of variable defined in loop
}


int main() {
        int i=0;
        while(i<5)
        {
                int j=2;   // Scope of i is main and scope of j is only the loop
                i++;
        }
}
```

❖ Scope of variable for conditional statements

```
if(test_expression) {
        // Scope of variable defined in the conditional statement
}

int main () {
        int i=0;
        if (i<5)
        {
                int j=5;          // Scope of j is only in this block
        }
            // cout<<j;  → This statement if written will give an error because
                           scope of j is inside if and is not accessible outside if.
    }
```

Increment Decrement operator

Explanation

Pre-increment and *pre-decrement* operators' increments or decrements the value of the object and returns a reference to the result.

Post-increment and *post-decrement* creates a copy of the object, increments or decrements the value of the object and returns the copy from before the increment or decrement.

Post-increment(a++):

This increases value by 1, but uses old value of a in any statement.

Pre-increment(++a):

This increases value by 1, and uses increased value of a in any statement.

Post-decrement(a--):

This decreases value by 1, but uses old value of a in any statement.

Pre-decrement(++a):

This decreases value by 1, and uses decreased value of a in any statement.

```
int main () {

    int I=1, J=1, K=1, L=1;

    cout<<I++<<' '<<J-- <<' '<<++K<<' '<< --L<<endl;

    cout<<I<<' '<<J<<' '<<K<<' '<<L<<endl;

}
```

Output:

```
1 1 2 0
```

Bitwise Operators

Operator	Name	Example	Result	Description
a & *b*	and	4 & 6	4	1 if both bits are 1.
a | *b*	or	4 | 6	6	1 if either bit is 1.
a ^ *b*	xor	4 ^ 6	2	1 if both bits are different.
~*a*	not	~4	-5	Inverts the bits. (Unary bitwise compliment)
n << *p*	left shift	3 << 2	12	Shifts the bits of *n* left *p* positions. Zero bits are shifted into the low-order positions.
n >> *p*	right shift	5 >> 2	1	Shifts the bits of *n* right *p* positions.

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

int main() {
    int a = 19;   // 19 = 10011
    int b = 28;   // 28 = 11000
   int c = 0;

   c = a & b;           //  16 = 10000
   cout << "a & b = " << c << endl ;

   c = a | b;          // 31 = 11111
   cout << "a | b = " << c << endl ;

   c = a ^ b;          // 15 = 01111
   cout << "a ^ b = " << c << endl ;

   c = ~a;             // -20 = 01100
   cout << "~a = " << c << endl ;

   c = a << 2;          // 76 = 1001100
   cout << "a ≪ 2 = " << c << endl ;
```

```
  c = a >> 2;          // 4 = 00100
  cout << "a >> 2 = " << c << endl ;

  return 0;
}
```

Output

```
a & b = 16
a | b = 31
a ^ b = 15
~a = -20
a << 2 = 76
a >> 2 = 4
```

Precedence and Associativity

- **Operator precedence** determines which operator is performed first in an expression with more than one operators with different precedence.
  For example, 10 + 20 * 30 is calculated as 10 + (20 * 30) and not as (10 + 20) * 30.
- **Associativity** is used when two operators of same precedence appear in an expression. Associativity can be either **L**eft **t**o **R**ight or **R**ight **t**o **L**eft. For example, '*' and '/' have same precedence and their associativity is **L**eft **t**o **R**ight, so the expression "100 / 10 * 10" is treated as "(100 / 10) * 10".

Precedence and Associativity are two characteristics of operators that determine the evaluation order of subexpressions in absence of brackets.

Note : We should generally use add proper brackets in expressions to avoid confusion and bring clarity.

1) Associativity is only used when there are two or more operators are of same precedence.

The point to note is associativity doesn't define the order in which operands of a single operator are evaluated. For example, consider the following program, associativity of the + operator is left to right, but it doesn't mean f1() is always called before f2(). The output of following program is in-fact compiler dependent.

```
// Associativity is not used in the below program. Output is compiler dependent.
 x = 0;
int f1() {
  x = 5;
  return x;
}
int f2() {
  x = 10;
  return x;
}
int main () {
  int p = f1() + f2();
  cout<<x;
  return 0;
}
```

2) All operators with same precedence have same associativity
This is necessary, otherwise there won't be any way for compiler to decide evaluation order of expressions which have two operators of same precedence and different associativity. For example, + and – have same associativity.

3) There is no chaining of comparison operators in C++
In C++ expression (a>b>c) will be treated as (a>b)>c .For example, consider the following program.. For example, consider the following program. The output of following program is "FALSE".

```
int main ()
{
  int a = 10, b = 20, c = 30;

  // (c > b > a) is treated as ((c > b) > a), associativity of '>'
  // is left to right. Therefore, the value becomes ((30 > 20) > 10). Since
(30>20) is true thus its answer is 1. So the expression becomes (1 > 20).

  if (c > b > a)
   cout<<"TRUE";
  else
```

```
     cout<<"FALSE";
    return 0;

}
```

Following is the Precedence table along with associativity for different operators.

OPERATOR	**DESCRIPTION**	**ASSOCIATIVITY**
( ) [ ] . -> ++ —	Parentheses (function call) (see Note 1) Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement (see Note 2)	left-to-right
++ — + – ! ~ (*type*) * & sizeof	Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (convert value to temporary value of *type*) Dereference Address (of operand) Determine size in bytes on this implementation	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ –	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right

^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
| |	Logical OR	left-to-right
? :	Ternary conditional	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left
,	Comma (separate expressions)	left-to-right

C++ Foundation with Data Structures

Lecture 5 : Functions, Variables and Scope

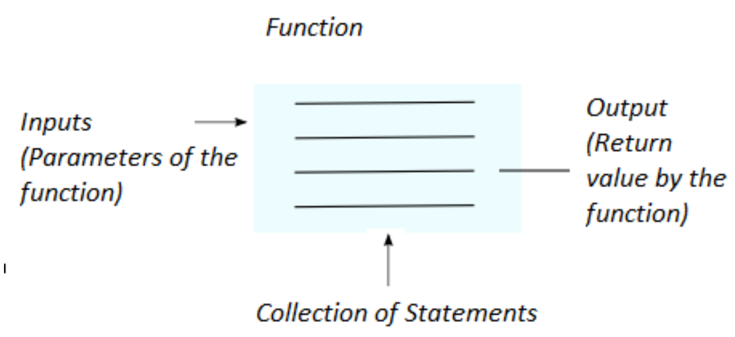
s

Functions

A Function is a collection of statements designed to perform a specific task. Function is like a black box that can take certain input(s) as its parameters and can output a value which is the return value. A function is created so that one can use it as many time as needed just by using the name of the function, you do not need to type the statements in the function every time required.

Defining Function

```
return_type  function_name(parameter 1, parameter 2, ………) {
        statements;
}
```


Function
Inputs (Parameters of the function)
Output (Return value by the function)
Collection of Statements

- ***return type:*** A function may return a value. The *return type* of the function is the data type of the value that function returns. Sometimes function is not required to return any value and still performs the desired task. The return type of such functions is ***void.***

Example:

Following is the example of a function that sum of two numbers. Here input to the function are the numbers and output is their sum.

```
int findSum( int a, int b){
  int sum = a + b;
  return  sum;
}
```

Function Calling

Now that we have read about how to create a function lets see how to call the function. To call the function you need to know the name of the function and number of parameters required and their data types and to collect the returned value by the function you need to know the return type of the function.

Example

```
#include<iostream>
using namespace std;
int findSum( int a, int b){
  int sum = a + b;
  return  sum;
}
int main () {
  int a = 10, b = 20;
  int c = findSum (a, b); // function findSum () is called using its name and
by knowing
  cout<< c;                // the number of parameters and their data type.
}                          // integer c is used to collect the returned value by
the function
```

Output:

```
30
```

IMPORTANT POINTS:

- ***Number of parameter*** and their ***data type*** while calling must match with function signature. Consider the above example, while calling function ***findSum ()*** the number of parameters are two and both the parameter are of integer type.

- It is okay not to collect the return value of function. For example, in the above code to find the sum of two numbers it is right to print the return value directly.

  “***cout<<findSum(a,b);***”

- **void return type functions:** These are the functions that do not return any value to calling function. These functions are created and used to perform specific task just like the normal function except they do not return a value after function executes.

Following are some more examples of functions and their use to give you a better idea.

Function to find area of circle

```
#include<iostream>
using namespace std;
double findArea( double radius){   //return type of the function is double.
            double area = 3.14 * radius * radius;
            return  area;
}
int main () {
            double radius = 5.8;
             double c = findArea (radius);
             cout<< c;
}
```

Function to print average

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

void printAverage(int a, int b ){  //return type of the function is void
          int avg = (a + b) / 2;
          cout<<avg << endl;
}                        // This function does not return any value

int main () {
           int a = 15, b = 25;
            printAverage (a, b);
}
```

Why do we need function?

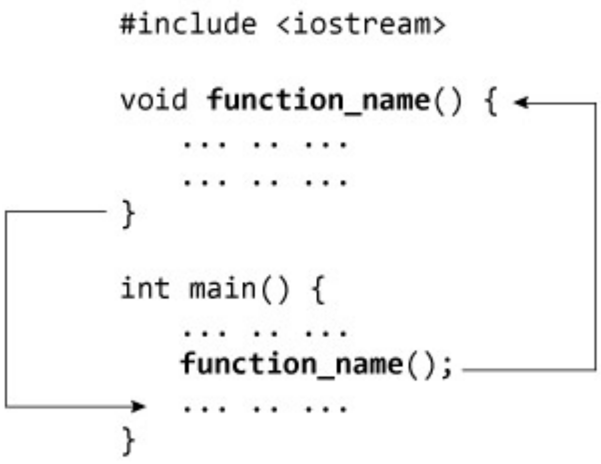
- **Reusability:** Once a function is defined, it can be used over and over again. You can call the function as many time as it is needed, which saves work. Consider that you are required to find out the area of the circle, now either you can apply the formula every time to get the area of circle or you can make a function for finding area of the circle and invoke the function whenever it is needed.
- **Neat code:** A code created with function is easy to read and dry run. You don’t need to type the same statements over and over again, instead you can invoke the function whenever needed.
- **Modularisation –** Functions help in modularising code. Modularisation means to divides the code in small modules each performing specific task. Functions helps in doing so as they are the small fragments of the programme designed to perform the specified task.
- **Easy Debugging:** It is easy to find and correct the error in function as compared to raw code without function where you must correct the error (if there is any) everywhere the specific task of the function is performed.

How does function calling works?

Consider the following code where there is a function called findsum which calculates and returns sum of two numbers.

//Find Sum of two integer numbers

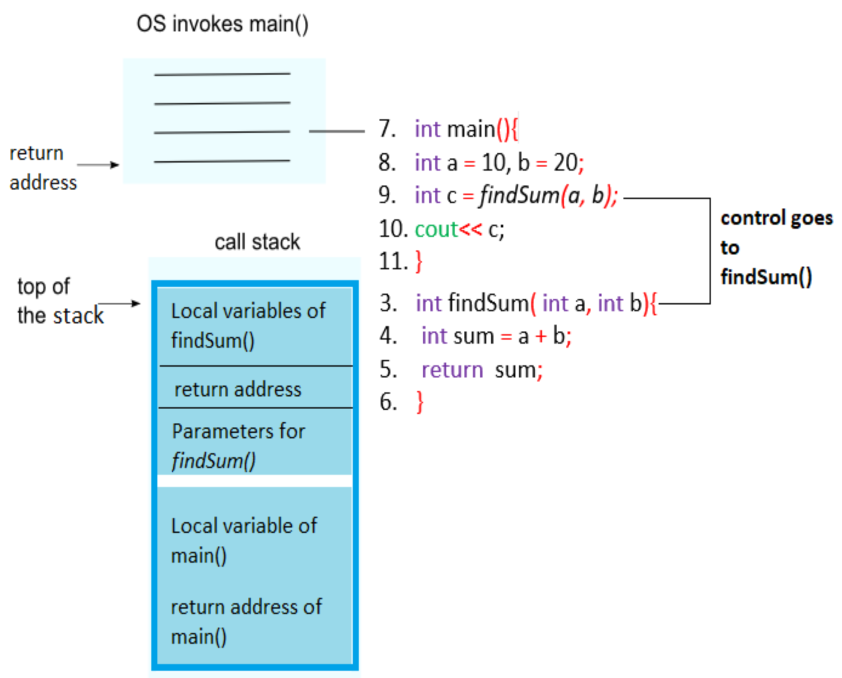
```
#include<iostream>
using namespace std;
int findSum( int a, int b){  // return type of the function is int.
        int sum = a + b;
        return  sum;
}
int main () {
        int a = 10, b = 20;
        int c = findSum (a, b);
        cout<< c;  }
```



```
#include <iostream>

void function_name() {
    ... .. ...
    ... .. ...
}

int main() {
    ... .. ...
    function_name();
    ... .. ...
}
```

The function being called is called **callee(**here it is findsum function) and the function which calls the callee is called the **caller** (here main function is the caller) .
When a function is called, programme control goes to the entry point of the function. Entry point is where the function is defined. So focus now shifts to callee and the caller function goes in paused state .

For Example: In above code entry point of the function ***findSum ()*** is at line number 3. So when at line number 9 the function call occurs the control goes to line number 3, then after the statements in the function ***findSum ()*** are executed the programme control comes back to line number 9.



```
7.  int main(){
8.  int a = 10, b = 20;
9.  int c = findSum(a, b);
10. cout<< c;
11. }
3.  int findSum( int a, int b){
4.   int sum = a + b;
5.   return  sum;
6.  }
```

Role of stack in function calling (call stack)

A call stack is a storage area that store information about the *active* function and paused functions. It stores parameters of the function, return address of the function and variables of the function that are created statically.

Once the function statements are terminated or the function has returned a value, the call stack removes all the information about that function from the stack.

Benefits of functions

- **Modularisation**
- **Easy Debugging:** It is easy to find and correct the error in function as compared to raw code without function where you must correct the error (if there is any) everywhere the specific task of the function is performed.
- **Neat code:** A code created with function is easy to read and dry run.

Variables and Scopes

Local Variables

Local variable is a variable that is given a local scope. Local variable belonging to a function or a block overrides any other variable with same name in the larger scope. Scope of a variable is part of a programme for which this variable is accessible.

Example:

```
#include<iostream>
using namespace std;
int main(){
    int a = 10;
    if (1){
        int a = 5;
        cout<< a<<endl;
    }
    cout<< a;
}
```

Output

```
5
10
```

In the above code the variable ***a*** declared inside the block after if statement is a local variable for this block and is different from the the variable ***a*** declared outside this block. Both these variable are allocated to different memory.

Lifetime of a Variable

The lifetime of a variable is the time period for which the declared variable has a valid memory. Scope and lifetime of a variable are two different concepts, scope of a variable is part of a programme for which this variable is accessible whereas lifetime is duration for which this variable has a valid memory.

Loop variable

Loop variable is a variable which defines the loop index for each iteration.

Example

```
“for (int i = 0; i < 3; i++) {     // variable i is the loop variable
                .......;
                ........;
                statements;
           } “
```
For this example, variable ***i*** is the loop variable.

Variables in the same scope

Scope is part of programme where the declared variable is accessible. In the same scope, no two variables can have name. However, it is possible for two variables to have same name if they are declared in different scope.

Example:

```
#include<iostream>
using namespace std;
int main () {
        int a = 10;
        int a = 5; // two variables with same name, the code will not compile
        cout<< a;
}
```

For the above code, there are two variables with same name ***a*** in the same scope of main () function. Hence the above code will not compile.

But the code given below will compile because the two variables with same name ***a*** are declared in different scope.

```
#include<iostream>
using namespace std;
int main () {
          int a = 10;
          if (1){
                int a = 5;
                cout<< a<<endl;
          }
          cout<< a;
}
```

Pass by value:

When the parameters are passed to a function by pass by value method, then the formal parameters are allocated to a new memory. These parameters have same value as that of actual parameters. Since the formal parameters are allocated to new memory any changes in these parameters will not reflect to actual parameters.

Example:

//Function to increase the parameters value

```
#include<iostream>
using namespace std;
void increment (int x, int y) {              // x and y are formal parameters
            x++;
            y = y + 2;
            cout<<x<<”: “<<y<<endl;
}

int main () {
            int a = 10, b = 20;
            increment (a, b);                // a and b are actual parameters
            cout<< a<<”: “<<b;
}
```

Output:

```
11: 22
10: 20
```

For the above code, changes in the values of ***x*** and ***y*** are not reflected to ***a*** and ***b*** because x and y are formal parameters and are local to function increment so any changes in their values here won’t affect variables a and b inside main.

C++ Foundation with Data Structures

Lecture 6 : Arrays

What are arrays?

In cases where there is a need to use several variables of same type, for storing, example, names or marks of 'n' students we use a data structure called arrays. Arrays are basically collection of elements having same name and same data type. Using arrays, saves us from the time and effort required to declare each of the element of array individually.

Creating an array

The syntax for declaring an array is:

Data_type array_name [array_size] ;

Example :

```
float marks[5];
```

Here, we declared an array, marks, of floating-point type and size 5. Meaning, it can hold 5 floating-point values.

Similarly, we can declare array of type int as follows:

```
int age[10];
```

How are arrays stored?

The elements of arrays are stored contiguously, i.e., at consecutive memory locations. The name of the array actually has the address of the first element of the array. Hence making it possible to access any element of the array using the starting address.

Example:

```
int age[ ] = {10, 14, 16, 18, 19};
```

Suppose the starting address of an array is 1000,then address of second and third element of array will be 1004, 1008 respectively and so on.

10	14	16	18	119
1000	1004	1008	1012	1016

Accessing elements of an array

An element of array could be accessed using indices. Index of an array starts from 0 to n-1, where n is the size of the array.

Syntax for accessing array element is :

Array_name[index]

Suppose we declare array age of size 5.

```
int age[10];
age[2] = 4       //  stores value 4 at index 2
```

		4		
age[0]	age[1]	age[2]	age[3]	age[4]

Here the first element is age[0], second element is age[1] and so on.

Default values

Arrays must always be initialized. If the array is not initialized the respective memory locations will contain garbage by default. Hence, any operation on uninitialized array will lead to unexpected results.

Initializing array at the time of declaration

Elements in an array can be explicitly initialized to specific values when it is declared, by enclosing those initial values in curly braces {}.

Example

```
int age[5] = {5, 2, 10, 4, 12};
```

Alternatively,

```
int age[ ] = {5, 2, 10, 4, 12};
```

5	2	10	4	12
age[0]	age[1]	age[2]	age[3]	age[4]

If array is initialized like this -

```
int age[10] = {5, 2, 10, 4, 12};
```

Then, in memory an integer array of size 10 will be declared. That is, a continuous memory block of 40 bytes (to hold 10 integers) will be allocated. And first 5 values of age are provided, rest will be 0.

Here,

age[0] is equal to 5

age[1] is equal to 2

age[2] is equal to 10

age[3] is equal to 4

age[4] is equal to 12

And age[5] to age[9] is equal to 0.

sizeof operator

sizeof is an operator used to determine the length of the array, i.e., the number of elements in the array .

Example:

```
int main( ){
        int myArray[ ] = {5, 4, 3, 2, 1};
        int size = sizeof(myArray);
        cout << size;
        return 0;
}
```

Output :

```
20
```

Passing arrays to a function

Arrays can be passed to a function as an argument. When an array is passed as an argument, only the starting address of the array gets passed to the function as an argument. Since, only the starting address gets passed to the function as opposed to whole array, the size of the array cannot be determined in function using sizeof operator. Hence, the arrays that are passed as an argument to a function must always be accompanied by its size as another argument.

Syntax of the function call to pass an array :

function_name(array_name, array_size);

Syntax of the function definition that takes array as an argument :

```
return_type function_name(data_type array_name, int size, <other
arguments>){
        //function body
}
```

NOTE : Square brackets '[ ]' are not used at the time of function call.

Example:

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

```
void printAge(int age[ ], int n){
        for(int i=0 ; i < n ; i++){
                cout << age[i] << endl;
        }
}
int main(){
        int age[5] = {11, 14, 15, 18, 20};
        printAge(age, 5);
        return 0;
}
```

Output:

```
11
14
15
18
20
```

Searching and Sorting

Searching

Searching means to find out whether a particular element is present in the given array/list. For instance, when you visit a simple google page and type anything that you want to know/ask about, basically you are searching that topic in the google's vast database for which google is using some technique in order to provide the desired result to you.

There are basically three types of searching techniques:

- Linear search
- Binary search
- Ternary search

We have already discussed the linear searching technique. In this module, we will be discussing binary search. We will not be discussing ternary search over here, for that You may explore the link mentioned below to know more about it.

Link for ternary search: (It is preferred to first-of-all clearly understand binary search and then move onto the ternary search)

https://cp-algorithms.com/num_methods/ternary_search.html

Linear Search

In this, we have to find a particular element in the given array and return the index of the same, in case it is not present the convention is to return -1. This can be simply done by traversing each element index one-by-one and then comparing the value at that index with the desired value.

Suppose you want to find a particular element in the sorted array, then following this technique, you have to traverse all the array elements for searching one element but guess if you only have to search at most half of the array indices for performing the same operation. This can be achieved through binary search.

Advantages of Binary search:

- This searching technique is fast and easier to implement.
- Requires no extra space.
- Reduces time complexity of the program to a greater extent. (The term **time complexity** might be new to you, you will get to understand this when you will be studying algorithmic analysis. For now, just consider it as the time taken by a particular algorithm in its execution and time complexity is determined by the number of operations that are performed by that algorithm i.e., time complexity is directly proportional to the number of operations in the program).

Binary Search

Now, let's look at what binary searching is.

Prerequisite: Binary search has one pre-requisite, the array must be sorted unlike the linear search where elements could be any order.

Let us consider the array:

0	1	2	3	4
1	**2**	**3**	**4**	**5**

You can see it is an array of size 5 with elements inside the boxes and indices above them. Our target element is 2.

Steps:

1. Find the middle index of the array. In case of even length of the array consider the index that appears first out of two middle ones.
2. Now, we will compare the middle element with the target element. In case they are equal then we will simply return the middle index.
3. In case they are not equal, then we will check if the target element is less than or greater than the middle element.
    - In case, the target element is less than the middle element, it means that the target element, if present in the given array, will be on the left side of the middle element as the array is sorted.
    - Otherwise, the target element will be on the right side of the middle element.
4. This helps us discard half of the length of the array and we have reduced the number of operations.
5. Now, since we know that the element is on the left, we will assume that the array's starting and ending indices to be:
    - **For left:** start = 0, end = n/2-1
    - **For right:** start = n/2, end = n-1

    where n are the total elements in the array.

6. We will repeat this process until we find the target element. In case start and end becomes equal or start becomes more than end, and we haven't found the target element till now, we will simply return -1 as that element is not present in the array.

In the example above, the middle element is 3 at index 2, and the target element is 2 which is greater than the middle element, so we will move towards the left part. Now marking start = 0, and end = n/2-1 = 1, now middle = (start + end)/2 = 0. Now comparing the 0-th index element with 2, we find that 2 > 1, hence we will be moving towards the right. Updating the start = 1 and end = 1, middle becomes 1, comparing 1-st index of the array with the target element, we find they are equal, meaning from here we will simply return 1 (the index of the element).

Now try implementing the approach yourself...

Implementation

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

int binarySearch(int arr[], int n, int x) {
	int start = 0, end = n - 1;
	while(start <= end) {
		int mid = (start + end) / 2;
		if(arr[mid] == x) {
			return mid;
		}
		else if(x < arr[mid]) {
			end = mid - 1;
		}
		else {
			start = mid + 1;
		}
	}

	return -1;
}

int main() {
	// Take array input from the user
	int n;
	cin >> n;

	int input[100];

	for(int i = 0; i < n; i++) {
		cin >> input[i];
	}

	int x;
	cin >> x;

	cout << binarySearch(input, n, x) << endl;
}
```

Note: There are other approaches also to perform binary searching like Recursion which you will study in advanced topics.

Sorting

Sorting means an arrangement of elements in an ordered sequence either in increasing(ascending) order or decreasing(descending) order. Sorting is very important and many softwares and programs use this. Binary search also requires sorting. There are many different sorting techniques. The major difference is the amount of space and time they consume while being performed in the program.

For now, we will be discussing the following sorting techniques only:

- Selection sort
- Bubble sort
- Insertion sort

Let's discuss them one-by-one...

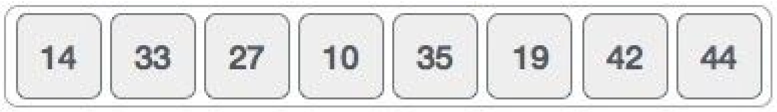
Selection sort:

Steps: (sorting in increasing order)

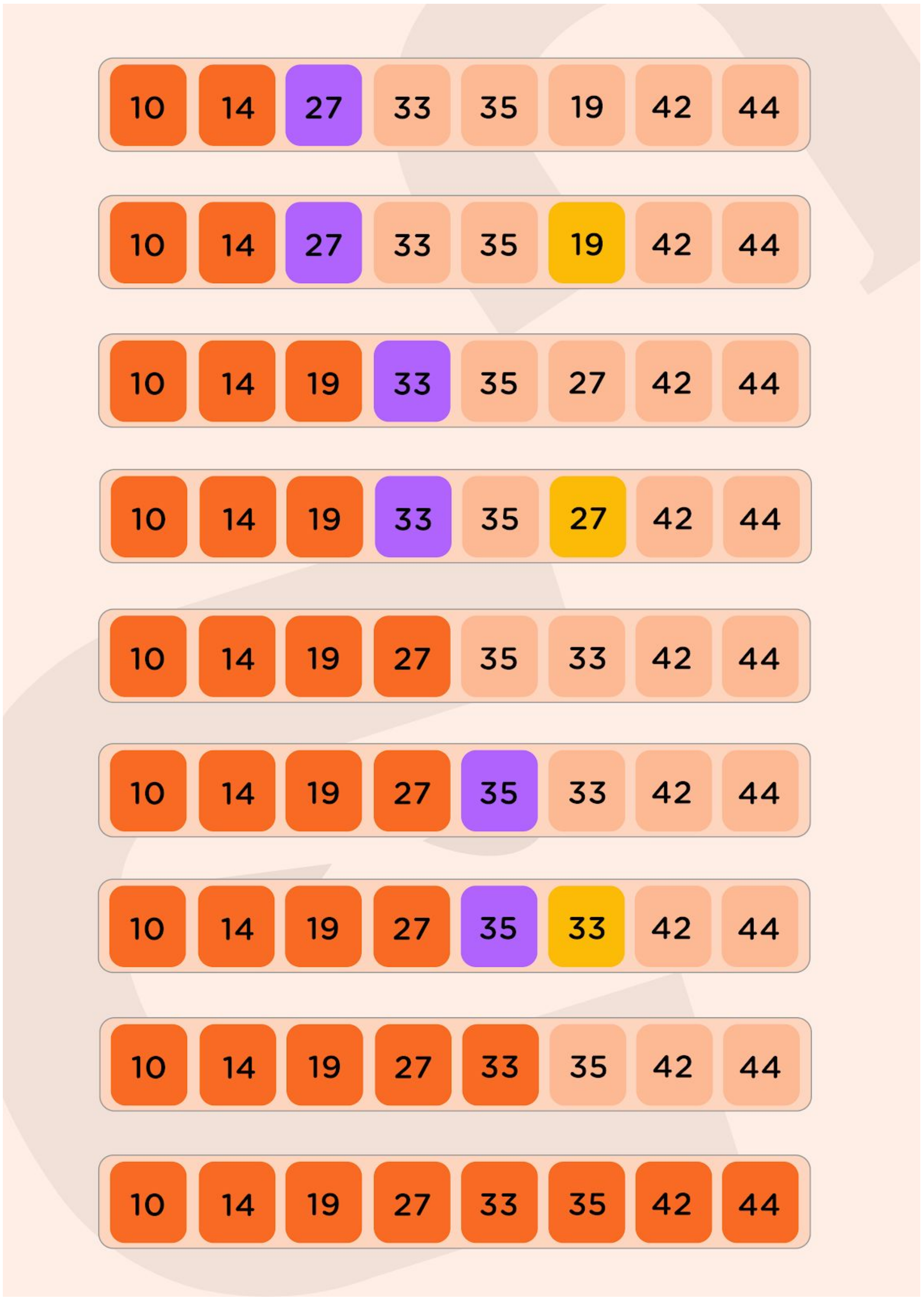
1. First-of-all, we will find the smallest element of the array and swap that with the element at index 0.
2. Similarly, we will find the second smallest and swap that with the element at index 1 and so on...
3. Ultimately, we will be getting a sorted array in increasing order only.

Let us look at the example for better understanding:

Consider the following depicted array as an example. You want to sort this array in increasing order.



Following is the pictorial diagram for better explanation of how it works:



This is how we obtain the sorted array at last.

Now looking at the implementation of Selection Sort:

Implementation of selection sort

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

void selectionSort(int input[], int n) {
        for(int i = 0; i < n-1; i++ ) {
        // Find min element in the array
        int min = input[i], minIndex = i;
        for(int j = i+1; j < n; j++) {
                if(input[j] < min) {
                        min = input[j];
                        minIndex = j;
                }
        }
        // Swap
        int temp = input[i];
        input[i] = input[minIndex];
        input[minIndex] = temp;
        }
}

int main() {
        int input[] = {3, 1, 6, 9, 0, 4};
        selectionSort(input, 6);
        for(int i = 0; i < 6; i++) {
                cout << input[i] << " ";

        }
        cout << endl;
}
```

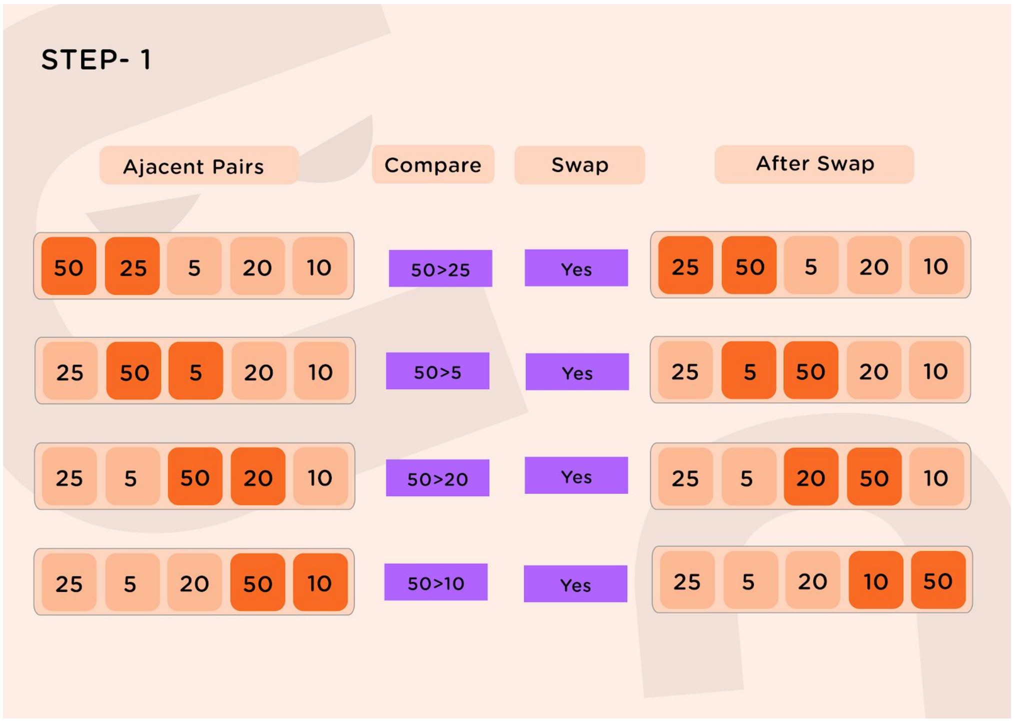
Bubble sort:

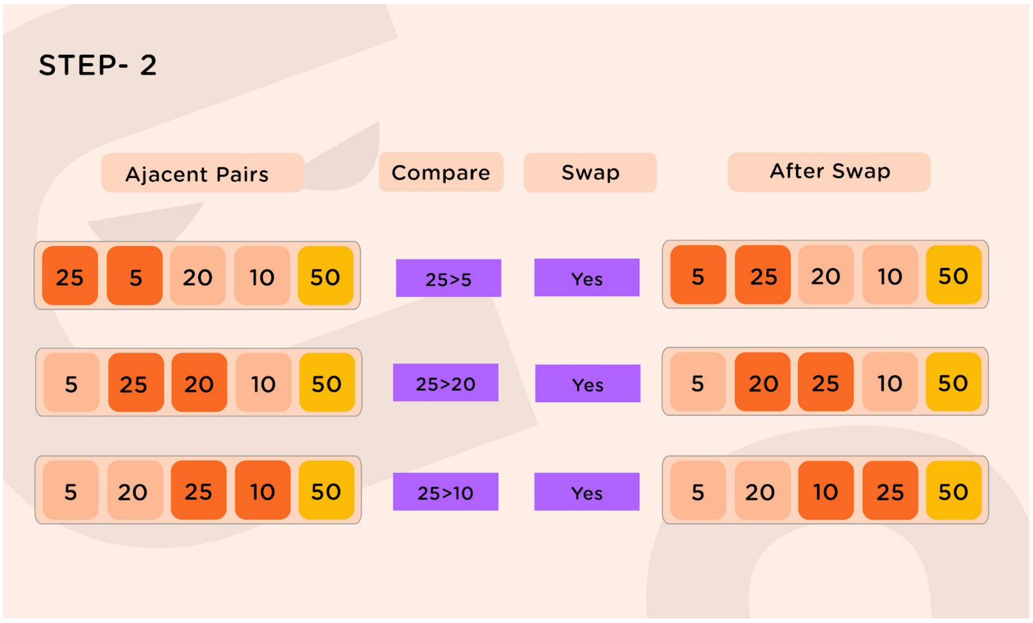
In selection sort the elements from the start get placed at the correct position first and then the further elements, but in the bubble sort, the elements start to place correctly from the end.

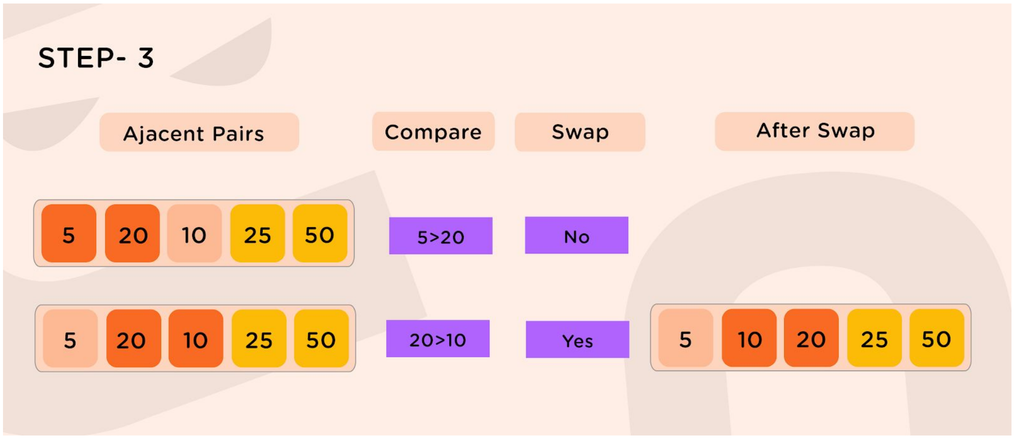
In this technique, we just compare the two adjacent elements of the array and then sort them manually by swapping if not sorted. Similarly, we will compare the next two elements (one from the previous position and the corresponding next) of the array and sort them manually. This way the elements from the last get placed in their correct position. This is the difference between selection sort and bubble sort. Consider the following depicted array as an example. You want to sort this array in increasing order.

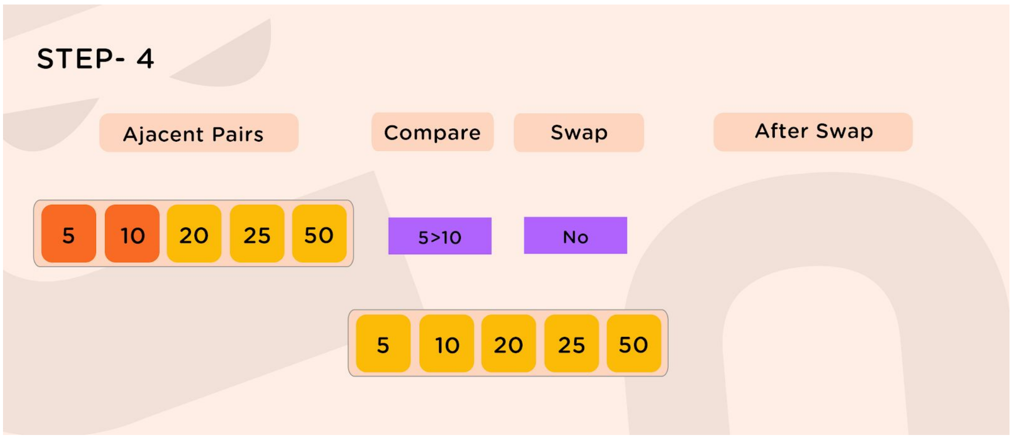
50	**25**	**5**	**20**	**10**

Following is the pictorial diagram for better explanation of how it works:




STEP- 2
Ajacent Pairs
Compare
Swap
After Swap
25 5 20 10 50
25>5
Yes
5 25 20 10 50
5 25 20 10 50
25>20
Yes
5 20 25 10 50
5 20 25 10 50
25>10
Yes
5 20 10 25 50


STEP- 3
Ajacent Pairs
Compare
Swap
After Swap
5 20 10 25 50
5>20
No
5 20 10 25 50
20>10
Yes
5 10 20 25 50


STEP- 4
Ajacent Pairs
Compare
Swap
After Swap
5 10 20 25 50
5>10
No
5 10 20 25 50

Implementation of bubble sort

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

void bubbleSort(int array[], int size) {

  // run loops two times: one for walking throught the array
  // and the other for comparison
  for (int step = 0; step < size - 1; ++step) {
    for (int i = 0; i < size - step - 1; ++i) {

      // To sort in descending order, change > to < in this line.
      if (array[i] > array[i + 1]) {

        // swap if greater is at the rear position
        int temp = array[i];
        array[i] = array[i + 1];
        array[i + 1] = temp;
      }
    }
  }
}

// function to print the array
void printArray(int array[], int size) {
  for (int i = 0; i < size; ++i) {
    cout << "  " << array[i];
  }
  cout << endl;
}

// driver code
int main() {
  int data[] = {-2, 45, 0, 11, -9};
  int size = sizeof(data) / sizeof(data[0]);
  bubbleSort(data, size);
  cout << "Sorted Array in Ascending Order:\n";
  printArray(data, size);
}
```

Insertion sort:

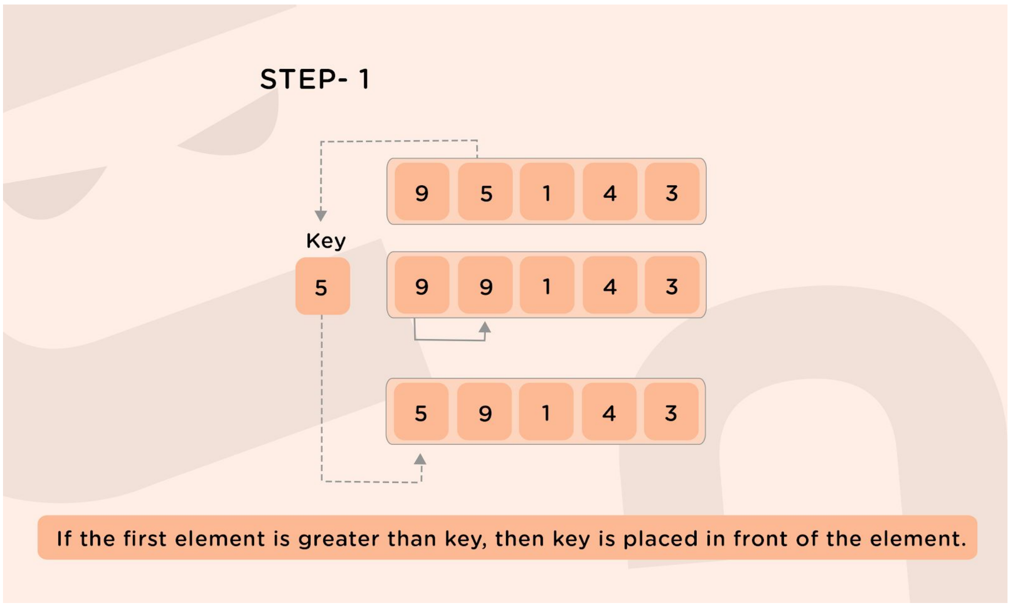
Insertion Sort works similar to how we sort a hand of playing cards.
Imagine that you are playing a card game. You're holding the cards in your hand, and these cards are sorted. The dealer hands you exactly one new card. You have to put it into the correct place so that the cards you're holding are still sorted. In selection sort, each element that you add to the sorted subarray is no smaller than the elements already in the sorted subarray. But in our card example, the new card could be smaller than some of the cards you're already holding, and so you go down the line, comparing the new card against each card in your hand, until you find the place to put it. You insert the new card in the right place, and once again, your hand holds fully sorted cards. Then the dealer gives you another card, and you repeat the same procedure. Then another card, and another card, and so on, until the dealer stops giving you cards. This is the idea behind **insertion sort**.
Loop over positions in the array, starting with index 1. Each new position is like the new card handed to you by the dealer, and you need to insert it into the correct place in the sorted subarray to the left of that position.

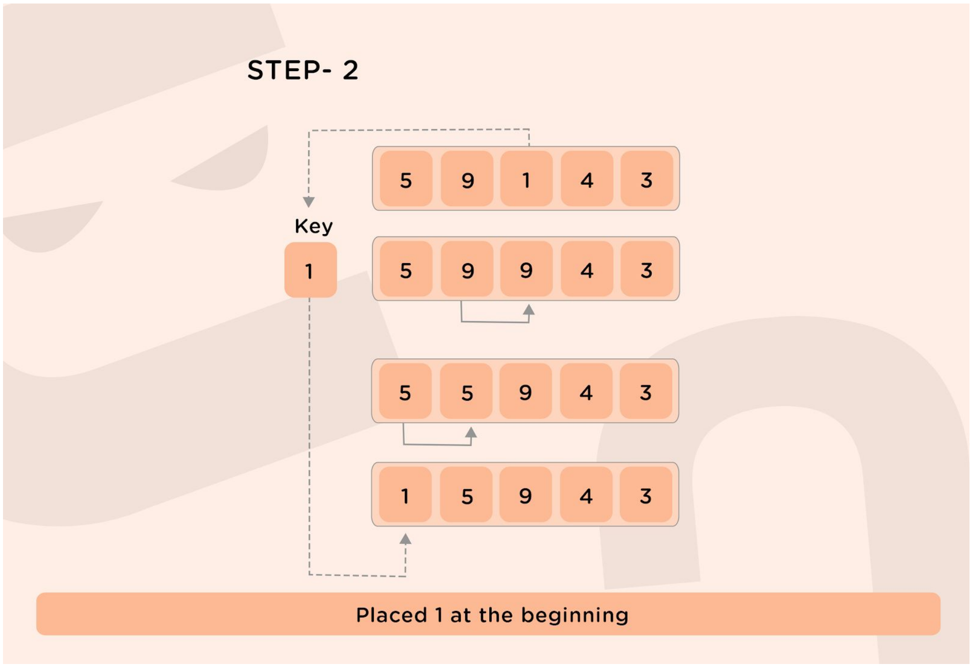
Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.

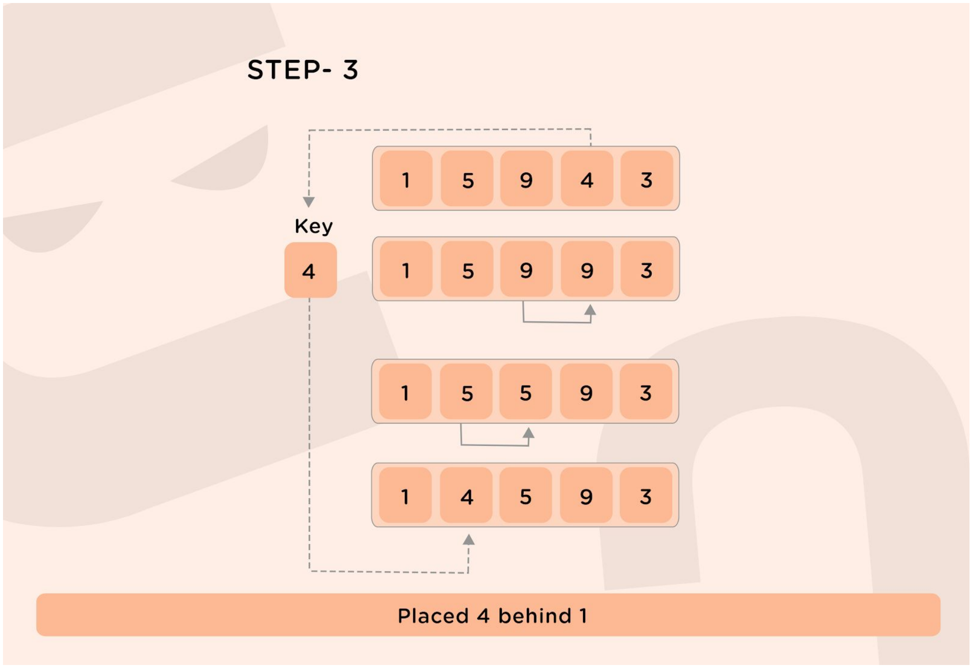
Consider the following depicted array as an example. You want to sort this array in increasing order.

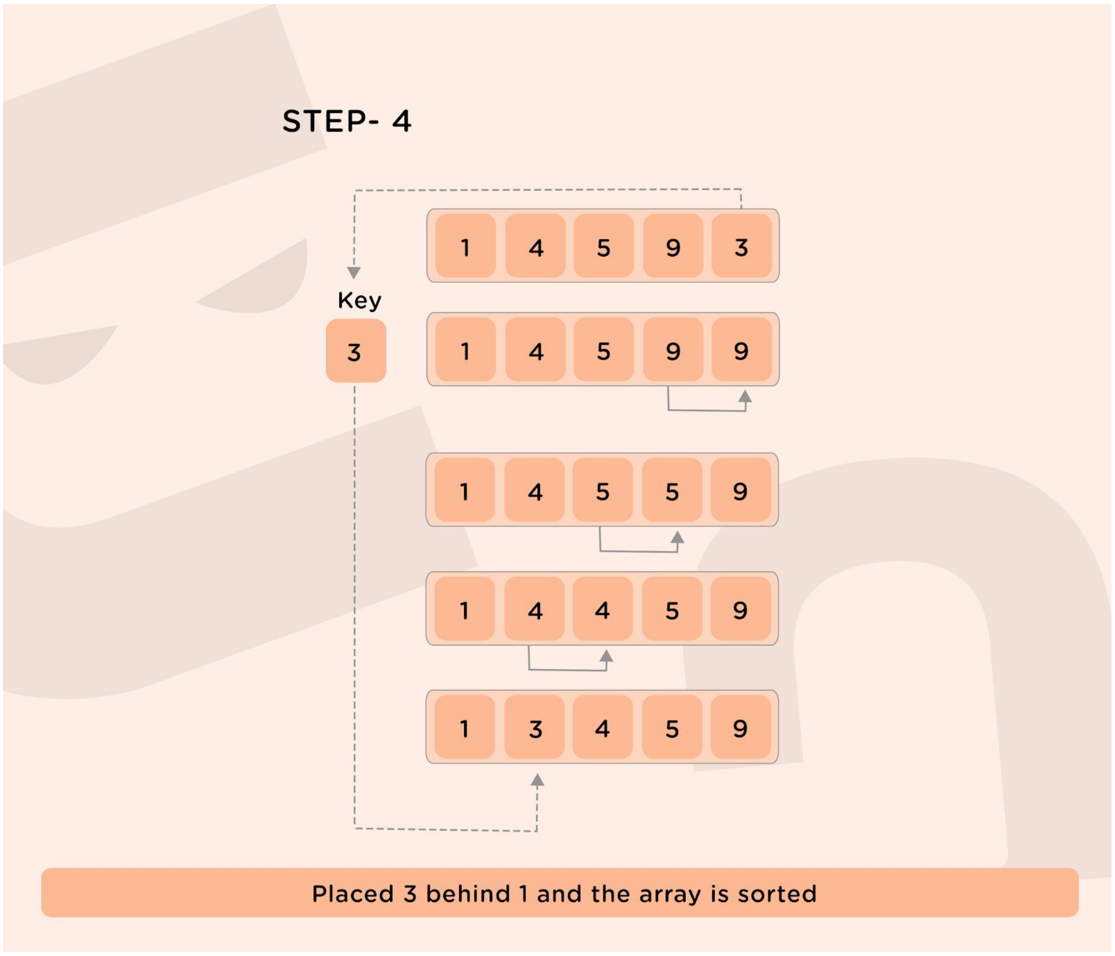
9	**5**	**4**	**1**	**3**

Following is the pictorial diagram for better explanation of how it works:


STEP- 1
9 5 1 4 3
Key
5
9 9 1 4 3
5 9 1 4 3
If the first element is greater than key, then key is placed in front of the element.


STEP- 2
Key
1
5 9 1 4 3
5 9 9 4 3
5 5 9 4 3
1 5 9 4 3
Placed 1 at the beginning


STEP- 3
Key
4
1 5 9 4 3
1 5 9 9 3
1 5 5 9 3
1 4 5 9 3
Placed 4 behind 1



Implementation of insertion sort

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

void display(int array[], int size) {
  for(int i = 0; i<size; i++)
    cout << array[i] << " ";
  cout << endl;
}
void insertionSort(int array[], int size) {
  int key, j;
  for(int i = 1; i<size; i++) {
    key = array[i];                              //take value
    j = i;
    while(j > 0 && array[j-1]>key) {
      array[j] = array[j-1];
      j--;
    }
    array[j] = key;                    //insert in right place
  }
}
int main() {
  int n;
  cout << "Enter the number of elements: ";
  cin >> n;
  int arr[n];                    //create an array with given number of elements
  cout << "Enter elements:" << endl;
  for(int i = 0; i<n; i++) {
    cin >> arr[i];
  }

  cout << "Array before Sorting: ";
  display(arr, n);

  insertionSort(arr, n);
  cout << "Array after Sorting: ";
  display(arr, n);
}
```

Now, practice different questions to get more familiar with the concepts. In the advanced course, you will study more types of sorting techniques.

Practice

- For binary search:

  https://www.hackerearth.com/practice/algorithms/searching/binary-search/practice-problems/

- For sorting:

  https://www.hackerearth.com/practice/algorithms/sorting/bubble-sort/tutorial/

  (In this link, you will find the questions for bubble sort, checkout the left tab, from there you can switch to other sorting techniques.)

Character arrays and 2D arrays

Strings

Till now, we have seen how to work with single characters but not with a sequence of characters. For storing a character sequence, we use strings. Strings are the form of 1D character arrays which are terminated by null character. Null character is a special character with a symbol '\0'.

Declaration syntax:

```
string  variableName;
```

string is the datatype in C++ to store continuous characters that terminates on encountering a space or a newline.

For example:

Suppose the input is:

```
Hello! How are you?
```

If you try to take input using string datatype over the given example, then it will only store **Hello!** in the form of an array.

To store the full sentence, we would need a total of four-string variables (though there are other methods to handle this (getline), we will soon read about them.)

Note: To use string datatype, don't forget to include the header file:

```
#include <cstring>
```

Character arrays

Declaration syntax:

```
char Name_of_array[Size_of_array];
```

All the ways to use them are the same as that of the integer array, just one difference is there. In case of an integer array, suppose we want to take 5 elements as input, an array of size 5 will be good for this.

In case of character arrays, if the length of the input is 5 then we would have to create an array of size at least 6, meaning **character arrays require one extra space in the memory from the given size.** This is because the character arrays store a NULL character at the last of the given input size as the mark of termination in the memory.

Now talking about the memory consumption, as each character requires 4 bytes in memory, the character array will require 4 multiplied by the character array size.

Also to take the character array as the input, you don't need to necessarily run the loop for each element. You can directly do the same as follows:

```
cin >> Name_of_array;
```

If we take above the example and run this statement over that and let the name of the array is arr and the size of array is at least 7, then the memory representation is as follows:

0	1	2	3	4	5	6													
'H'	**'e'**	**'l'**	**'l'**	**'o'**	**'!'**	**'\0'**													

Here, also the cin statement terminates taking input if it encounters any of the following:

- Space
- Tab
- \n

At the last you can notice some special character is placed, that is the NULL character about which we were talking earlier.

In the same way, you can directly use a cout statement to print the character array instead of running the loop.

Till now we haven't solved the "string with spaces" problem. Let's check that also...

Getline function:

In C++, we have another function named as **cin.getline()** which takes 3 arguments:

- Name of the string or character array name
- Length of the string
- Delimiter (optional argument)

Syntax:

```
cin.getline(string_name, length_of_string, delimiter);
```

Note: This function breaks at new line. This also initializes the last position to NULL.

Delimiter denotes the ending of each string. Generally, it is '\n' by default.

Let's look at some examples:

Suppose the input we want to take is: **Hello how are you?**

If the input commands are:

- **cin.getline(input, 100);**: this will input all the characters of input
- **cin.getline(input, 3);**: this will input only the first 3 characters of input and discard the rest instead of showing error.

In-built functions for character array:

- **strlen(string_name)**: To calculate the length of the string
- **strcmp(string1, string2)**: Comparison of two strings returns an integer value. If it returns 0 means equal strings. If it returns a positive value, it means that string2 is greater than string1 and if it returns a negative value, it means that string1 is greater than string2.

- **strcpy(destination_string, source_string) :** It copies the source_string to destination_string.
- **strncpy(destination_string, source_string, number_of_characters_to_copy):** it copies only a specified number of characters from source_string to destination_string.

2D arrays

Combination of many 1D arrays forms a 2D array.

Syntax to declare a 2D array:

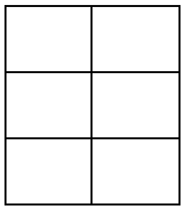
```
datatype array_name[rows][columns];
```

where rows imply the number of rows needed for the array and column implies the number of columns needed.

For example: If we want to declare an integer array of size 2x3, then:

```
int arr[2][3];
```

It looks like:



In this array you can store the values as required. Suppose in the above array you want to store 3 at every index, you can do so using the following code:

```
for (int i = 0; i < 2; i++)
{
        for (int j = 0; j < 3; j++)
        {
                arr[i][j] = 3;
        }
}
```

where, arr[i][j] invokes the element of the ith row and jth column.

How are 2D arrays stored in the memory?

They are actually stored as a 1D array of size (number_of_rows * number_of_columns).

Like if you have an array of size 5 x 5, so in the memory, a 1D array is created of size 25 and if you want to get the value of the element (2,1) in this array, it will invoke (2*5 + 1 = 11)th position in the array. We don't need to take care of this calculation, these are done by the compiler itself.

If we want to pass this array to a function, we can simply pass the name of the array as we did in the case of 1D arrays. But in the function definition, we can leave the first dimension empty, though the second dimension always needs to be specified.

Syntax for function call:

```
int fun (int arr[ ][number_of_columns], int n, int m) {
………………
}
```

Let's move to some questions now:

Practice problems:

Arrays and strings:

https://www.hackerearth.com/challenges/competitive/code-monk-array-strings/problems/

2D arrays:

https://www.hackerrank.com/challenges/2d-array/problem
https://www.techgig.com/practice/data-structure/two-dimensional-arrays

Rest there are many questions available for practice on codezen too. You can try them also...