Features of C++

Introduction

C++ is a general-purpose programming language and is widely used nowadays for competitive programming. It has imperative, object-oriented, and generic programming features. C++ runs on lots of platforms like Windows, Linux, Unix, Mac, etc. It can be used to develop operating systems, browsers, games, and so on. This makes C++ powerful as well as flexible.

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, various compilers like Code blocks, VS Code, Dev C++, Atom, and many more exist. You may choose one for yourself.

Features of C++

C++ provides a lot of **features** that are given below.

- Simple
- Portability
- Powerful and Fast
- Rich Library
- Platform Dependent
- Mid-level programming language
- Structured programming language
- Object-Oriented
- Case Sensitive
- Compiler Based
- Syntax based language
- Pointers
- Dynamic Memory Management

1) **Simple:** C++ is a simple language because it provides a structured approach (to break the problem into parts), a rich set of library functions, data types, etc. It allows us to follow both procedural as well as functional approach to design our flow of control.

2) **Portability:** It is the concept of carrying the instruction from one system to another system. In C++, Language **.cpp** file contains source code, and we can also edit this code. **.exe** file contains the application, which is the only file that can be executed. When we write and compile any C++ program on the Windows operating system, it efficiently runs on other window-based systems.

3) **Powerful:** C++ is a very powerful programming language, and it has a wide variety of data types, functions, control statements, decision-making statements, etc. C++ is a fast language as compilation and execution time is less. Also, it has a wide variety of data types, functions & operators.

4) **Rich Library:** C++ library is full of in-built functions that save a tremendous amount of time in the software development process. As it contains almost all kinds of functionality, a programmer can need in the development process. Hence, saving time and increasing development speed.

5) **Platform Dependent:** Platform dependent language means the language in which programs can be executed only on that operating system where it is developed & compiled. It cannot run or execute on any other operating system. E.g., compiled programs on Linux won't run on Windows.

6) **Mid-level programming language:** C++ can do both low-level & high-level programming. That is the reason why C++ is known as a mid-level programming language.

7) **Structured programming language:** C++ is a structured programming language as it allows to break the program into parts using functions. So, it is easy to understand and modify.

8) **Object-oriented:** C++ is an object-oriented programming language. OOPs make development and maintenance easier, whereas, in Procedure-oriented programming language, it is not easy to manage if code grows as project size grows. It follows the concept of oops like polymorphism, inheritance, encapsulation, abstraction.

9) **Case sensitive:** C++ is a case-sensitive programming language. In C++ programming, 'break and BREAK' both are different.

10) **Compiler Based:** C++ is a compiler-based language, unlike Python. C++ programs used to be compiled, and their executable file is used to run it due to which C++ is a relatively faster language than Java and Python.

11) **Syntax-based language:** C++ is a strongly typed syntax-based programming language. If any language follows the rules and regulations strictly, it is known as a strongly syntax-based language.Other examples of syntax-based languages are C, C++, Java, .net etc.

12) **Pointer:** C++ supports pointers that allow the user to deal directly with the memory and control the programmer. This makes it very suitable for low-level tasks and very complicated projects. It is known to increase the speed of execution by decreasing the memory access overhead.

13) **Dynamic Memory Management:** It supports the feature of dynamic memory allocation. In C++ language, we can free the allocated memory by calling the free() function. These features are missing in languages like C.

Previous
Next Uses of C++

Uses of C++

There are several benefits of using C++ because of its features and security; below are some uses of C++ Programming Language:

Operating Systems: One of the key requirements of an Operating System is that it should be very fast as it is responsible for scheduling and running the user programs. The strongly typed and fast nature of C++ makes it an ideal candidate for writing operating systems. Also, C++ has a vast collection of system-level functions that also help in writing low-level programs**.** Microsoft Windows or Mac OS X, or Linux - all operating systems have some parts programmed in C++.

Games: Again since most of the games need to be faster to support smooth game play, C++ is extensively used in game design. C++ can easily manipulate hardware resources, and it can also provide procedural programming for CPU-intensive functions.

Browsers: With the fast performance of C++, most browsers have their rendering software written in C++. Browsers are mostly used in C++ for rendering purposes. Rendering engines need to be faster in execution as most people do not like to wait for the web page to be loaded.

Libraries: Many high-level libraries use C++ as the core programming language. For example, TensorFlow uses C++ as the back end programming language. Such libraries required high-performance computations because they involve multiplications of huge matrices to train Machine Learning models. As a result, performance becomes critical. C++ comes to the rescue in such libraries.

Graphics: C++ is widely used in almost all graphics applications that require fast rendering, image processing, real-time physics, and mobile sensors.

Cloud/Distributed Systems: Cloud storage systems use scalable file-systems that work close to the hardware; also, the multi threading libraries in C++ provide high concurrency and load tolerance.

Embedded Systems: C++ is closer to the hardware level, and so it is quite useful in embedded systems as the software and hardware in these are closely coupled. Many embedded systems use C++, such as smartwatches, MP3 players, GPS systems, etc.

Compilers: Compilers of various programming languages use C++ as the back-end programming language.

Previous Next

Headers in C++

Headers

C++ code begins with the inclusion of header files. There are many header files available in the C++ programming language, which will be discussed 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 saves 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.

Now, moving forward to the code:

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

iostream stands for Input/Output stream, meaning this header file is necessary 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. i.e., different statements in a C++ program are separated by a semicolon.

Previous
Next main()
function

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 part 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 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 this function. 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 a particular function scope. Again, these things will be clear when you will study functions.** For now, just understand that this is the format in which we are going to write our basic C++ code. As you will move forward with the course, you will get a clear and better understanding.

Previous Next
Complile and Run

Compile and run a C++ code

For compiling and running a CPP program in Linux following are the command lines:

```
Compile: g++ Filename.cpp
Run or execute: ./a.out
```

For compiling and running a CPP program in Windows following are the command lines:

```
Compile: gcc Filename.cpp
Run or execute: filename
```

Previous Next

Macros in C++

Macros

Macros are a piece of code in a program that is given some name. Whenever the compiler encounters this name, the compiler replaces the name with the actual piece of code. The ‘#define’ directive is used to define a macro.

Note: There is no semicolon(‘;’) at the end of the macro definition.

Example:

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

// macro definition
#define LIMIT 5
 int main()
{
    for (int i = 0; i < LIMIT; i++) {
cout << i << " ";
    }
return 0;
}
```

```
Output:
0 1 2 3 4
```

Macros with arguments: We can also pass arguments to macros. Macros defined with arguments work similarly as functions.

Example:

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

// macro with parameter #define
Area(l, b)(l * b)
```

```
int main() {     int l =
10, b = 5, a;     a =
Area(l, b);
    cout << "The Area of the rectangle is: " << a;
return 0;
}
```

```
Output:
The area of the rectangle is: 50
```

Previous Next

Comments in C++

Comments

C++ comments are hints that a programmer can add to make their code easier to read and understand. They are completely ignored by C++ compilers.

There are two ways to add comments to code:

// **-** Single Line Comment
/* */ **-** Multi-line Comments

Example: Single line comment.

```
#include <iostream>
using namespace std;
 int main()
{    //
This is a
comment
cout <<
"Hello
World!";
return 0;
}
 Output:
Hello World!
```

Example: Multi-line Comment.

```
#include <iostream>
using namespace std;
 int main()
{
    /* The code below will print - Hello World!
to the screen*/     cout << "Hello World!";
return 0;
}

 Output:
Hello World!
```

Previous Next

Introduction

Introduction

A **variable** is a storage place that has some memory allocated to it. It is used to store some form of data. Different types of variables require different amounts of memory.

Variable Declaration

In C++, we can declare variables as follows:

- **data_type:** Type of the data that can be stored in this variable. It can be int, float, double, etc.
- **variable_name:** Name given to the variable.

```
data_type variable_name;
Example:  int x;
```

In this way, we can only create a variable in the memory location. Currently, it doesn't have any value. We can assign the value in this variable by using two ways:

- By using variable initialization.
- By taking input

Here, we can discuss only the first way, i.e., variable initialization. We will discuss the second way later.

```
data_type variable_name = value;
Example:  int x = 20;
```

Rules for defining variables in C++

- You can’t begin with a number. **Ex- 9a** can't be a variable, but **a9** can be a variable.
- Spaces and special characters except for underscore(_) are not allowed.
- C++ keywords (reserved words) must not be used as a variable name. C++ is case-sensitive, meaning a variable with the name ‘A’ is different from a variable with the name ‘a’. (Difference in the upper-case and lower-case holds true).

C++ Keywords

asm	else	new	this
auto	enum	operator	throw
bool	explicit	private	true
break	export	protected	try
case	extern	public	typedef
catch	false	register	typeid
char	float	reinterpret_cast	typename
class	for	return	union
const	friend	short	unsigned
const_cast	goto	signed	using
continue	if	sizeof	virtual
default	inline	static	void
delete	int	static_cast	volatile
do	long	struct	wchar_t
double	mutable	switch	while
dynamic_cast	namespace	template	

In addition, the following words are also reserved:

And	bitor	not_eq	xor
and_eq	compl	or	xor_eq
bitand	not	or_eq	

Examples:

```
Correct int
_a = 10; int
b4 = 10;
float c_5;
```

```
Incorrect int
1a; float b@
= 2;
String true;
```

Previous Next

Introduction

Introduction

All variables use data-type during declaration to restrict the type of data to be stored. Therefore, we can say that data types tell the variables the type of data they can store.

Pre-defined data types available in C++ are:
● **int**: Integer value
● **unsigned int:** Can store only positive integers.
● **float**, **double**: Decimal number
● **char**: Character values (including special characters)
● **unsigned char**: Character values
● **bool**: Boolean values (true or false)
● **long**: Contains integer values but with the larger size
● **unsigned long**: Contains large positive integers or 0
● **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)

Data Type	Default Size	Range
bool	1 byte or 8 bits	true or false
char	1 byte or 8 bits	-2^7 to 2^7-1 -128 to 127(by default 0 to 255)
unsigned char	1 byte or 8 bits	0 to 255
short	2 bytes or 16 bits	-2^15 to 2^15-1 -32,768 to 32,767
int	4 bytes or 32 bits	-2^31 to 2^31-1 -2,147,483,648 to 2,147,483,647
unsigned int	4 bytes or 32 bits	0 to 2^32-1 0 to 4,294,967,295
long	8 bytes or 64 bits	-2^63 to 2^63-1
unsigned long	8 bytes or 64 bits	0 to 2^32-1 0 to 4,294,967,295
float	4 bytes or 32 bits	-2^31 to 2^31-1 -2,147,483,648 to 2,147,483,647
double	8 bytes or 64 bits	-2^63 to 2^63-1

Examples:

```
int price = 5000;                    // Integer (whole number)
float interestRate = 5.99f;          // Floating point number
char myLetter = 'D';                 // Character bool
isPossible = true;                 // Boolean
```

```
String myText = "Coding Ninjas";      // String
```

auto keyword in c++

The auto keyword specifies that the type of the declared variable will automatically be deduced from its initializer. It would set the variable type to initialize that variable's value type or set the function return type as the value to be returned.

Example:

```
auto a = 11           // will set the variable a as int type
auto b = 7.65         //will set the variable b as float auto
c = "abcdefg"   // will set the variable c as string
```

Previous Next

Scope of a Variable

Scope of a Variable

Scope of a variable refers to the region of visibility or availability of a variable i.e the parts of your program in which the variable can be used or accessed.

There are mainly two types of variable scopes:

- Local Scope
- Global Scope

1. Local Scope

Variables declared within the body of a function or block are said to have local scope and are referred to as **'local variables'**. They can be used only by the statements inside the body of the function or the block they are declared within.

Example:

```
void person() {     string
gender = "Male";
    //This variable gender is local to the function person()
//and cannot be used outside this function.
}
```

2. Global Scope

The variables whose scope is not limited to any block or function are said to have global scope and are referred to as **'global variables'**. Global variables are declared outside of all the functions, generally on the top of the program, and can be accessed from any part of your program. These variables hold their values throughout the lifetime of the program.

Example:

```
#include <iostream>
using namespace std;
// Global variable declaration: and can be used anywhere in code
int g; int main() {
    g = 10; // Using global variable
cout << g;     return 0;
}
```

Previous Next

Types of Variables

Types of Variables

There are three types of variables based on the scope of the variable in C++:

- Local Variables
- Instance Variables
- Static Variables

1. Local Variables

The variables declared within the body of a function or block are known as local variables and are created(occupy memory) when the program enters the block or makes a function call. The local variables get destroyed(memory is released) after exiting from the block or return from the function call. They can be used only by the statements inside the body of the function or the block they are declared within.

Example:

```
void Function() {
    //local variable marks
    int marks = 90;
marks = marks + 2;
    cout << "Student obtained "<<marks<<" marks."
return 0;
}
Output: Student obtained 92 marks.
```

2. Instance Variables

Instance variables are non-static variables and belong to an instance of a class and are declared in a class outside any method, constructor, or block. These variables are created when an object of the class is created and destroyed when the object is destroyed and are accessible to all the constructors, methods, or blocks in the class. Each object of the class within which the instance variable is declared will have its own separate copy or instance of this variable. Unlike local variables, we may use access specifiers for instance variables.

Example:

```
class A {
    int a; // by default private instance variables
int b;     public:          int c; // public
instance variable      Void     function () {
a = 10;          cout << a;
    }
};
```

3. Static Variables

Static variables are declared using the keyword **‘static’**, within a class outside any method, constructor, or block. Space is allocated only once for static variables i.e we have a single copy of the static variable corresponding to a class, unlike instance variables. The static variables are created at the start of the program and get destroyed at the end of the program i.e the lifetime of a static variable is the lifetime of the program. Static variables are initialized only once and they hold their value throughout the lifetime of the program.

Example:

```
class A {      static int
var; //static variable
Void func() {
         ++var;
    }
};
```

Previous Next

Overflow and Underflow

Overflow and Underflow

Overflow occurs when we assign a value to more than its range, and Underflow is the opposite of the overflow. In the case of overflow and underflow, the C++ compiler doesn’t throw any errors. It simply changes the value.

For example, in case of an int variable, the maximum value of int data type is 2,147,483,647 (INT_MAX) and after incrementing 1 on this value, it will return -2,147,483,648 (INT_MIN). This is known as overflow.

The minimum value of int data type is -2,147,483,648 (INT_MIN) and after decrementing 1 on this value, it will return 2,147,483,647 (INT_MAX). This is known as underflow.

Example:

```
#include <iostream>
using namespace std;
int main() {
    int x = INT_MAX; //2147483647
int y = INT_MIN; //-2147483648
x = x + 1;     y = y - 1;     cout
<< x << endl;     cout << y;
}
Output:
-2147483648
2147483647
```

Previous Next

Typecasting

Typecasting

Converting an expression of a given data type into another data type is known as **type-casting or type-conversion**. There are two types of type conversions:

- Implicit Type Conversion
- Explicit Type Conversion

1. Implicit Type Conversion

It is automatically performed by the compiler itself to ensure that the calculations between the same data types take place and avoid any loss of data. Such types of conversions take place when more than one data type is present in an expression. The rule associated with implicit type conversions involves upgrading the data type of all the variables to the data type of the variable with the **“largest data type”**.

The order of automatic type conversion or the sequence for smallest to largest data type(left to right) for this type conversion is given as:

bool -> char -> short int -> int -> unsigned int -> long -> unsigned -> long long -> float -> double -> long double

Example:

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

```
 int main() {     int
number = 200;     char
letter = 'c';
float dec = 0.7;
    int res1 = number + letter; // here letter is implicitly converted to int and its value is the
    //  ASCII value of 'c' i.e. 99     cout << res1 << " ";     float res2
= res1 + dec; // here res1 is implicitly converted to float.
    cout << res2;
}
```

```
Output: 299 299.7
```

2. Explicit Type Conversion:
This process is also called typecasting, and it is user-defined. Here the user can typecast the result to make it of a particular data type which may lead to data loss and is also known as forceful casting.

Syntax:

```
(type) expression
```

Example:

```
int main() {
double dbl = 5.6;
    int res = (int)dbl + 10; // Here dbl is explicity converted to int i.e value of
// dbl becomes 5.
    cout << "Result = " << res;
}
```

```
Output: Result = 15
```

Previous Next
Taking Input from Console

Cin

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

cin is a predefined variable that reads data from the keyboard with the extraction operator ***(>>)***.

Example1:

```
int x;
cin >> x; // Get user input from the keyboard
cout << x; Input: 5
Output: 5
```

Example2:

```
string x;
cin >> x; // Get user input from the keyboard
Input1: “Hello”
Output1: “Hello”
```

Question: Calculate the Simple Interest.

```
#include<iostream> using
namespace std; int
main() {    int
principal, time;
double rate, si;
cout << "Enter Principal
Amount: ";    cin >>
principal;    cout <<
"Enter Rate of Interest:
";    cin >> rate;
cout << "Enter Time
period ";    cin >>
time;    si =
(principal * rate *
time) / 100;    cout <<
"Simple Interest: " <<
si;
}
```

```
Output:
Enter Principal Amount: 3000
Enter Rate of Interest: 2
```

```
Enter Time period: 3
Simple Interest: 180
```

Question: Add two numbers.

```
#include <iostream> using namespace std;  int
main() {     int n1, n1, addition;     cout <<
"Enter two integers: ";     cin >> n1 >> n2;
addition = n1 + n2;     cout << "Sum of the
numbers is: " << addition;     return 0; }
```

```
Output:
Enter two integers: 5 10
Sum of the numbers is: 15
```

Getline

getline() is a standard library function used to read a string or a line from an input stream.

It is used when input strings with spaces between them or process multiple strings at once.

The *getline()* function extracts characters from the input stream and appends them to the string object until the delimiting character is encountered. While doing so, the previously stored value in the string object str will be replaced by the input string, if any.

Syntax 1: `istream& getline(istream& is, string& str, char delim);` **Parameters:**

- **is:** is an object of the istream class that tells the stream's function the input that needs to be read.
- **str:** is the string object that stores the input after the stream's reading process is finished.
- **delim:** defines the delimitation character, which commands the function to stop processing the input. The reading process will stop once the written code reads this command.

Syntax 2: `istream& getline (istream& is, string& str);`

The second declaration is almost the same as that of the first one. The only difference is that the latter have a delimitation character by defaulting a new line(\n)character.

Example:

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

```
 int main() {
string s;
getline(cin, s);
cout << s;
return 0;
}
```

```
Input: Hello world
Output: Hello world
```

Previous
Next
Printing to the Console

Cout

The C++ **cout** is used to produce output on the standard output device, usually the display screen or console. **cout** is a pre-defined variable that displays some output or text using the insertion operator(**<<**).

Example:

```
cout << “Hello World!”;
Output: Hello World!
```

endl: It is used to insert a new line character.

Example:

```
cout << “Hello” << endl;
cout << “World”; Output:
Hello
World!
```

To print the variable value, the syntax is as followed

```
cout << Variable_name;
```

Example:

```
#include <iostream> using
namespace std; int main() {
int age = 21;     string
firstName = "King";     string
lastName = "Kong";
    cout << "My name is " << firstName << " " << lastName << endl;
cout << "My age is " << age << endl;     return 0;
}
Output:
My name is King Kong
My age is 21
```

Previous Next

Introduction

Introduction

Operators in C++ are the special symbol-specific operations on one or more operands upon which it returns a result. There are different types of operators available in C++, which are given below:

- **Arithmetic Operators**
- **Unary Operators**
- **Relational Operators**
- **Logical Operators**
- **Bitwise Operators**
- **Assignment Operators**
- **Misc Operators**

Previous Next
Arithmetic Operators

1. Arithmetic Operators:

Arithmetic operators are used for performing common mathematical operations, i.e., Addition, Subtraction, Multiplication, and division. The basic arithmetic operators in C++ are given below:

- **Addition Operator (+):** It is used to add two numbers.
- **Subtraction Operator (-):** It is used to subtract two numbers.

- **Multiplication Operator (*):** It is used to multiply two numbers.
- **Division Operator (/):** It is used to divide two numbers. Mind that it gives the quotient.
- **Modulus Operator (%):** It is used to retrieve the remainder from the division operation.

Example:

```
#include<iostream>
using namespace std;
 int main() { int a =
50, b = 20;
    cin >> a, b;    int
addition = a + b;    int
subtraction = a - b;    int
mul = a * b;    int
division = a / b;    int
modulus = a % b;
    cout << "The addition of " << a << " and " << b << " is: " << addition;
cout << "The subtraction of " << a << " and " << b << " is: " << subtraction;
cout << "The multiplication of " << a << " and " << b << " is: " << mul;    cout
<< "The division of " << a << " and " << b << " is: " << division;    cout <<
"The modulus of " << a << " and " << b << " is: " << modulus;    return 0;
}
```

```
Output:
The addition of 50 and 20 is: 70
The subtraction of 50 and 20 is: 30
The multiplication of 50 and 20 is: 1000
The division of 50 and 20 is: 2
The modulus of 50 and 20 is: 10
```

Previous Next
Unary Operators

2. Unary Operators:

Unary Operators are the types of operators that require only one operand. They form various operations on single operands, such as incrementing or decrementing the value by one, negating an expression, or inverting a boolean’s value. Let’s understand the various unary operators with an example.

(i) Unary minus operator (-): This operator can be used to convert a negative value into a positive value and vice-versa.

Example:

```
#include<iostream> using namespace std; int main()
{     int num1 = 10; num1 = -num1; //Converting
positive to negative
    cout << "Negative Value: " << num1;
int num2 = -20;
    num2 = -num2; //Converting negative to positive
cout << "Positive Value: " << num2;     return 0;
}
Output:
Negative Value: -10
Positive Value: 20
```

(ii) Unary NOT Operator (!): This operator is used to convert “true” to “false” and vice versa.

Example:

```
#include<iostream>
using namespace std;
int main() {     int a
= 10, b = 1;
    cout << "!(a < b) = " << !(a < b) << endl;
cout << "(a < b) = " << (a < b);     return 0;
}
Output:
!(a < b) = 1
(a < b) = 0
```

(iii) Increment Operator (++): This operator is used to increment the value by 1. There are two types of increment operators

1. **Post-increment operator: Post increment operator is used to increment the variable’s value after it has been evaluated for use in the expression.**
2. **Pre-increment operator: Pre increment operator is used to increment the variable’s value before it’s evaluated in the expression.**

Example:

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

```
int main() {     int
num = 10;
    cout << "Post increment = " << num++;
    // first print 10, then number is increment to 11 cout
    << "Pre increment = " << ++num;
    // num was 11, incremented to 12 and print
return 0;
}
Output:
Post increment = 10
Pre increment = 12
```

(iv) Decrement Operator (--): This operator is used to decrement the value by 1. There are two types of decrement operators.

1. **Post-decrement operator: Post decrement operator is used to decrement the value of the variable after it has been evaluated for use in the expression.**
2. **Pre-decrement operator: Pre decrement operator is used to decrement the value of the variable before it's evaluated in the expression.**

Example:

```
#include<iostream>
using namespace std;
int main() {     int
num = 10;
    cout << "Post increment = " << num--;     //
first print 10, then number is decrement to 9
cout << "Pre increment = " << --num;     // num was
9, decremented to 8 and print     return 0;
}
Output:
Post decrement = 10
Pre decrement = 8
```

v) Bitwise Complement (~): This operator is used to return the one's complement representation of the input value.

Example:

```
#include<iostream>
using namespace std;
int main() {     int
```

```
num1 = 6;     int
num2 = -2;
    // Performing bitwise complement
    cout << num1 << "'s bitwise complement = " << ~num1; cout
    << num2 << "'s bitwise complement = " << ~num2;
    return 0;
}
Output:
6's bitwise complement = -7
-2's bitwise complement = 1
```

Previous Next
Relational Operators

3. Relational Operators:

The Relational operators are used to check the relationship between two operands. This operator is also called a comparison operator because it is used to make a comparison between two operands. The result of these operators is always boolean value. These operators are used in if statements and loops. There are many types of relational operators, which are given below:

i) Equal to operator (==): This operator is used to check whether the two operands are equal or not. If they are equal, it returns true(1); otherwise it returns false(0).

Example:

```
#include<iostream>
using namespace std;
 int main() {
int a = 10;
int b = 20;
int c = 20;
    cout << (a == b) << endl;
cout << (b == c);
    // Check two operands are equal or not if they equal return true,
    // else return false
return 0;
}


Output:
0
```

```
1
```

ii) Not Equal to operator (!=): This operator is used to check whether the two operands are equal or not. It returns true(1) if the left operand is not equal to the right operand; otherwise, it returns false(0).

Example:

```
#include<iostream>
using namespace std;
 int main() {
int a = 10;
int b = 20;
int c = 20;
    cout << (a != b) << endl;
cout << (b != c);
    // Returns true if the left operand is not equal to the right operand
return 0;
}


Output:
1
0
```

iii) Greater than operator (>): This operator is used to check whether the first operand is greater than the second operand or not. It returns true(1) if the first operand is greater than the second operand and false(0) if not.

Example:

```
#include<iostream>
using namespace std;
 int main() {
int a = 10;
int b = 20;
    cout << (a > b) << endl;
cout << (b > a);
    // Returns true if left operand is greater then right operand
return 0;
}


Output:
```

```
0
1
```

iv) Greater than equal to the operator (>=): This operator is used to check whether the first operand is greater than or equal to the second operand or not. It returns true(1) if the first operand is greater than or equal to the second operand; otherwise, it returns false(0).

Example:

```
#include<iostream>
using namespace std;
 int main() {     int
a = 10;     int b =
8;     cout << (a >=
b);
    // It returns true because the first operand is greater than the second
return 0;
}
```

```
Output:
1
```

v) Less than operator (<): This operator is used to check whether the first operand is less than the second operand or not. It returns true(1) if the first operand is less than the second operand else returns false(0).

Example:

```
#include<iostream>
using namespace std;
 int main() {
int a = 10;     int
b = 15;     cout <<
(a < b);
    // It returns true because the first operand is smaller than the second
return 0;
}
```

```
Output:
1
```

vi) Less than or equal to operator (<=): This operator is used to check whether the first operand is less than or equal to the second operand or not. It returns true(1) if the first operand is less than or equal to the second operand; else, return false(0).

Example:

```
#include<iostream>
using namespace std;
 int main() {     int
a = 10;     int b =
5;     cout << (a <=
b);
    // It returns false because the first operand is
//not less than or equal to the second operand.     return
0;
}
```

```
Output:
0
```

Previous Next

Logical Operators

4. Logical Operators:

These operators are used to perform logical operations such as OR, AND, and NOT operation. It operates on two boolean values, which return true or false as a result. There are three types of Logical Operators in C++:

i) Logical AND operator (&&): This operator returns true(1), if both the conditions are true else returns false(0).

Example:

```
#include<iostream>
using namespace std;
int main() {     int
a = 10;     int b =
20;
    int c = 30;
```

```
    cout << ((b > a) && (c > b)) << endl; // true
cout << ((b > a) && (c < b)); // false     return
0;
}
Output:
1
0
```

ii) Logical OR operator (||): This operator returns true(1) if any one of the conditions is true.

Example:

```
#include<iostream>
using namespace std;
int main() {     int
a = 10;     int b =
20;     int c = 30;
    cout << ((b > a) || (c < b)); // true
cout << ((b < a) || (c < b)); // false
return 0;
}
Output:
1
0
```

iii) Logical NOT operator (!): This operator is used to reverse the operand's value. If the operand's value is true, it returns false(0), and if the value of the operand is false, it returns true(1).

Example:

```
#include<iostream>
using namespace std;
int main() {     int
a = 10;     int b =
20;
    cout << (!(a == b)); // true
cout << (!(b > a)); // false
return 0;
}
Output:
```

```
1
0
```

Operator	Description	Syntax
&&	Logical AND: True if both the operands are true	x && y
||	Logical OR: True if either of the operands is true	x || y
!	Logical NOT: True if an operand is false	! x

The truth table for all combinations of values of X and Y.

X	Y	X && Y	X || Y	!(X)	!(Y)
T	T	T	T	F	F
T	F	F	T	F	T
F	T	F	T	T	F
F	F	F	F	T	T

Previous Next

Bitwise Operators

5. Bitwise Operators:

The Bitwise operators are used to perform bit manipulation on numbers. There are various types of Bit operators that are used in C++.

i) Bitwise AND operator (&): It takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1. Mind that the commutative property holds true here.

That is,

1 & 1 = 1
1 & 0 = 0

Example:

```
#include<iostream>
using namespace std;
int main() {
    int a = 6; // Binary representation of 6 is 0110
int b = 7; // Binary representation of 7 is 0111     cout
<< ”a & b = ” << (a & b); //0110 & 0111 = 0110 = 6
return 0;
} Output:
a & b = 6
```

ii) Bitwise OR operator (|): It takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1. Mind that the commutative property holds true here.

That is,

1 | 1 = 1
1 | 0 = 1
0 | 0 = 0

Example:

```
#include<iostream>
using namespace std;
int main() {
    int a = 6; // Binary representation of 6 is 0110
int b = 7; // Binary representation of 7 is 0111     cout
<< "a | b = " << (a | b); //0110 | 0111 = 0111 = 7
return 0;
} Output:
a | b = 7
```

iii) Bitwise NOT operator (~): It takes one number and inverts all bits of it.

That is,

~1 = 0

~0 = 1

Example:

```
#include<iostream>
using namespace std;
int main() {
    int a = 6; // Binary representation of 6 is 0110
cout << "~a = " << (~a); //0110 NOT is 1001 = 9
return 0;
}
Output:
~a = 9
```

iv) Bitwise XOR operator (^): It takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different. Mind that the commutative property holds true here.

That is,

1 ^ 1 = 0

1 ^ 0 = 1

0 ^ 0 = 0

Example:

```
#include<iostream>
using namespace std;
int main() {
    int a = 6; // Binary representation of 6 is 0110
int b = 7; // Binary representation of 7 is 0111     cout
<< "a ^ b = " << (a ^ b); //0110 ^ 0111 = 0001 = 1
return 0;
}
Output: a
^ b = 1
```

v) Left shift operator (<<): It takes two numbers, the left shift operator shifts the bits of the first operand, the second operand decides the number of places to shift.

Example:

```
#include<iostream>
using namespace std;
int main() {
    int a = 8; // Binary representation of 8 is 1 0 0 0
cout << "a << 2 = " << (a << 2);
    // Left shift means appending numbers of 0's to the right.
    //1 0 0 0 0 = 32
return 0;
} Output: a
<< 2 = 32
```

vi) Right shift operator (>>): It takes two numbers; the right shift operator shifts the bits of the first operand, the second operand decides the number of places to shift.

Example:

```
#include<iostream>
using namespace std;
int main() {
    int a = 8; // Binary representation of 8 is 1 0 0 0
cout << "a >> 2 = " << (a >> 2);
    // Right shift means remove numbers of 0's from right 1 0 = 2
return 0;
} Output:
a >> 2 = 2
```

NOTE: The left shift and right shift operators should not be used for negative numbers. If any of the operands is a negative number, it results in undefined behaviour. For example results of both -1 << 1 and 1 << -1 is undefined. Also, if the number is shifted more than the integer's size, the behaviour is undefined. For example, 1 << 33 is undefined if integers are stored using 32 bits.

Previous Next

Assignment Operators

6. Assignment Operators:

The Assignment operators are used to assign a value to the variable. In C++, we can use many assignment operators. These are explained below:

i) +=: This assignment operator is used to add the left operand with the right operand and then assigning it to a variable on the left.

Example:

```
#include<iostream> using
namespace std; int main() {
int num = 10;     num += 20; //
num = num + 20;     cout << num;
return 0;
}
Output:
30
```

ii) -=: This assignment operator is used to subtract the left operand with the right operand and then assign it to a variable on the left.

Example:

```
#include<iostream> using
namespace std; int main() {
int num = 20;     num -= 10; //
num = num - 10;     cout << num;
return 0;
}
Output:
10
```

iii) *=: This assignment operator is used to multiply the left operand with the right operand and then assign it to a variable on the left.

Example:

```
#include<iostream> using
namespace std; int main() {
int num = 10;     num *= 5; //
num = num * 5;     cout << num;
return 0;
}
Output:
50
```

iv) /=: This assignment operator is used to divide the left operand with the right operand and then assign it to a variable on the left.

Example:

```
#include<iostream> using
namespace std; int main() {
int num = 10;     num /= 2; //
num = num / 2;     cout << num;
return 0;
}
Output:
5
```

v) %=: This assignment operator is used to modulo the left operand with the right operand and then assign it to a variable on the left.

Example:

```
#include<iostream> using
namespace std; int main() {
int num = 19;     num %= 5; //
num = num % 5;     cout << num;
return 0;
} Output:
4
```

Previous Next

Misc Operators

7. Misc Operators:

Apart from the above operators, some other operators are available in C++ to perform some specific tasks. They are explained below:

i)sizeof operator: This operator determines a variable's size. sizeof operator can also be used to determine the size of a data type.

Example:

```
#include <iostream>
using namespace std;
int main() {     int
a;
```

```
    cout << "Size of int : " << sizeof(int) << "\n";
cout << "Size of char : " << sizeof(char) << "\n";
cout << "Size of float : " << sizeof(float) << "\n";
cout << "Size of double : " << sizeof(double) << "\n";
cout << "Size of a : " << sizeof(a) << "\n";     return 0;
}
Output:
Size of int : 4
Size of char : 1
Size of float : 4
Size of double : 8
Size of a : 4
```

ii)Comma operator(,): It is a binary operator that evaluates its first operand and discards the result. It then evaluates the second operand and returns this value.

Example:

```
#include <iostream>
using namespace std;
int main() {     int
x, y;     y = 100;
    x = (y + 10, 99 + y);
    cout << "With brackets value of x :" << x << endl;
x = y + 10, 99 + y;
    cout << "Without brackets value of x :" << x;
return 0;
}
Output:
With brackets value of x :199
Without brackets value of x :110
```

iii) Conditional Operator(?:) or ternary operators: It is of the form

```
Expression1 ? Expression2 : Expression3
```

Here, Expression1 is the condition to be evaluated. If the condition(Expression1) is True, then we will execute and return the result of Expression2; otherwise, if the condition(Expression1) is false, then we will execute and return the result of Expression3. Since it takes three operands to work, hence they are also called ternary operators.

Example:

```
#include <iostream>
using namespace std;
int main() {     int
a = 1, b;
    b = (a < 10) ? 2 : 5; //As a is less than 10 hence b=2
cout << "Value of b: " << b << endl;     return 0;
}
Output:
Value of b: 2
```

iv)Pointer Operator:

a)&: It refers to the address (memory location) in which the operand is stored. **b)*:** It is a pointer operator.

Example:

```
#include <iostream>
using namespace std;
int main() {
    int a = 1, * b; //Here b is a pointer operator of int type
b = & a;
    cout << "Address of variable a: " << b << endl;
cout << "Address of variable b: " << & b;     return
0;
}
//Here, answers may vary.
Output:
Address of variable a: 0x7ffe85c2db74
Address of variable b: 0x7ffe85c2db78
```

Previous

Next Introduction

Introduction

A C++ conditional statement redirects a program's flow to execute additional code. Conditional statements in C++ decide the direction of the flow of program execution. There are different conditional statements available in C++, which are given below:

- if statement
- if-else statement
- nested if statement
- if-else-if statement
- switch statements
- Jump statements

Previous Next

if statement

i) if statement: The **if** keyword is used to execute a statement or block, if, and only if, a condition is fulfilled. **Syntax:**

```
if (condition) {
    //Statements to execute if the condition is true.
}
```

Note: If we do not provide the curly braces '{' and '}' after if(condition), then by default, the if statement will consider the first immediate statement below it to be inside its block.

```
if(condition)
statement 1; statement
2;
If the condition is true, then output=Statement 1
```

Example:

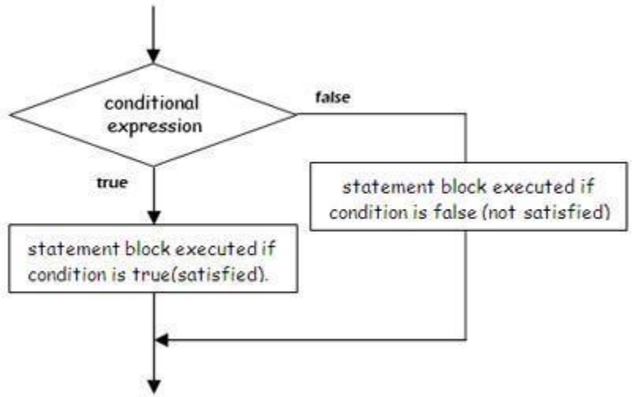
```
#include<iostream>
using namespace std;
 int main() {
int i = 20;
if (i > 10) {
        cout << "Condition is true";
    }
return 0;
}

Output:
Condition is true
```

Previous Next

if-else statement ii) if-else statement: The if statement tells us that if a condition is true, it will execute a block of statements, but it won't if the condition is false. So, using the else statement with the if statement, we execute a code block when the condition is false.

Syntax:

```
if (condition) {
    // Executes this block if the condition is true
} else {
    // Executes this block if the condition  is false
}
```



Example: Check whether the given number is even or odd.

```
#include<iostream> using namespace std;  int
main() {     int number;     cin >> number;
if (number % 2 == 0) {         cout << "The
Number Entered is Even";
    } else {         cout << "The Number
Entered is Odd";
    }
return 0;
}
```

```
Input1: 20
Output1: The Number Entered is Even
Input2: 15
Output2: The Number Entered is Odd
```

Que: Write a code a check whether a given number is divisible by 10 or not.
Solution:

```
#include<iostream> using
namespace std;  int
main() {     int number;
cin >> number;     if
(number % 10 == 0) {
cout << "The Number is
divisible by 10";
    } else {
        cout << "The Number is not divisible by 10";
    }
return 0;
}
```

```
Input1: 20
Output1: The Number is divisible by 10
Input2: 15
Output2: The Number is not divisible by 10
```

Previous Next nested if statement iii) nested if statement: Nested if statements mean an if statement inside another if statement.
Syntax:

```
if (condition1) {
    // Executes when condition1 is true
if (condition2) {
        // Executes when condition2 is true
    }
}
```

Example:

```
#include <iostream>
using namespace std;
int main() {     int
age = 15;
```

```
    // Outer If condition
if (age >= 14)
    {
        // Nested If condition
if (age >= 18)              cout <<
"You are an adult";         else
            cout << "You are a teenager";
    }
    // Outer Else condition    else
{         if (age > 0)
cout << "You are a child";
else             cout << "Invaild
age";
    }
return 0;
}
```

```
Output:
You are an adult
```

Que: Write a code a check whether a given number is even or odd or neither(0).
Solution:

```
#include <iostream>
using namespace std;
 int main() {    int
number;     cin >>
number;     // Outer If
condition     if (number
!= 0) {
        // Nested If condition, checking the divisibility of the number by 2.
if ((number % 2) == 0) {             cout << "The number is even.\n";
        }
else {
            cout << "The number is odd.\n";
        }
    }
    // Outer Else condition
else {
        cout << "The number is neither even nor odd.\n";
```

```
    }
return 0;
}

Input1: 5
Output1: The number is odd.
Input2: 0
Output2: The number is neither even nor odd.
Input3: 24
Output3: The number is even.
```

Previous Next

if-else-if statement iv) if-else-if statement: The conditional expressions are executed in a top-down manner, where as soon as a condition is found that evaluates to true, the statements associated with that condition get executed bypassing the rest of the ladder, passing the control flow to the end of this ladder. However, if none of the conditions evaluate to true, then the statements associated with the final else will get executed, this else statement acts as a default condition that gets executed when all conditions fail. If there is no final else statement and all conditions become false, then no action takes place. **Syntax:**

```
if(condition1) {
    // Executes if condition1 is true
}
else if (condition2) {
    // Executes if condition2 is true
}
else if (condition3) {
    // Executes if condition3 is true
} ...
else {
    // Executes if all conditions become false
}
```

Example: Check whether an integer is positive, negative, or zero.

```
#include<iostream>
using namespace std;
#include <iostream>
using namespace std;
 int main() {     int
number;     cin >>
number;     if
(number > 0) {
```

```
        cout << "You entered a positive integer\n";
    }
    else if (number < 0) {
        cout << "You entered a negative integer\n";
    }
else {
        cout << "You entered 0.\n";
    }
return 0;
}
```

```
Input1: 34
Output1: You entered a positive integer. Input1:
-5
Output1: You entered a negative integer.
Input2: 0
Output2: You entered 0.
```


Switch Statement

v) switch statement: Switch case statements help in testing the equality of a variable or expression against a set of values and provide a good alternative for long if statements.

Syntax:

```
switch (expression)  {
case constant1:
        // code to be executed if the expression equals constant1
        break;
case constant2:
        // code to be executed if the expression equals constant2
break;    case constant3:
        // code to be executed if the expression equals constant3
break;        ...    default:
        // code to be executed if the expression does not match any constants
}
```

The **expression** is evaluated once and must evaluate to a **"constant"** value and compared with the values of each **case** label(constant 1, constant 2, .., constant n).

- If a match is found corresponding to a case label, then the code following that label is executed until a break statement is encountered or the control flow reaches the end of the switch block.

- If there is no match, the code after default is executed.
- The default statement is optional. If there is no match then no action takes place and the control reaches the end of the switch block in absence of the default statement.
- The break statement is also optional and the code corresponding to all case labels gets executed after the matching case until a break statement is encountered.
- There cannot be duplicate case values.
- The default statement always gets executed if there is no break statement following the matching case or no match is found.

Que: Create a Calculator using the switch Statement.
Solution:

```
#include <iostream>
using namespace std;
int main() {
    float num1, num2, result;     char
oper; //to store operator choice

    cout << "Enter first number: " << endl;
cin >> num1;
    cout << "Enter second number: " << endl;
cin >> num2;

    cout << "Choose operation to perform (+,-,*,/): " << endl;
cin >> oper;

    switch (oper){
case '+':
        result = num1 + num2;
        cout << "Result: " << num1 << " " << oper << " " << num2 << " = " << result;
break;     case '-':
        result = num1 - num2;
        cout << "Result: " << num1 << " " << oper << " " << num2 << " = " << result;
break;     case '*':
        result = num1 * num2;
        cout << "Result: " << num1 << " " << oper << " " << num2 << " = " << result;
break;     case '/':
        result = num1 / num2;
        cout << "Result: " << num1 << " " << oper << " " << num2 << " = " << result;
break;     default:         cout << "Invalid operation.";
    }
return 0;
```

```
}

Output1:
Enter first number:
2.5
Enter second number:
5.7
Choose operation to perform (+,-,*,/):
+
Result: 2.5 + 5.7 = 8.2 Output2:
Enter first number:
2.5
Enter second number:
5.7
Choose operation to perform (+,-,*,/):
?
Invalid operation.
```

Previous Next

Jump Statement

vi) Jump Statements: Jump statements in C++ are used for the unconditional flow of control throughout a program's functions. There are four types of jump statements in C++ they are as follows: 1. break

2. continue
3. goto
4. return

1) break statement: In C++, the break statement terminates the loop or switch statement when encountered, and control returns from the loop or switch statement immediately to the first statement after the loop.

Syntax:

```
break;
```

Example: Check if an array contains any negative value.

```
#include <iostream>
using namespace std;
 int
main() {
```

```
    int arr[] = {5, 6, 0, -3, 3, -2, 1};
int size = 7; // No of elements in array
for (int i = 0; i < size; i++)
    {
        if (arr[i] < 0)
        {
            // Array contains a negative value, so break the loop
cout << "Array contains negative value."; break;
        }
    }
}
```

```
Output:
Array contains negative value.
```

2) continue statement: In C++, the continue statement is used to skip the current iteration of the loop, and the control of the program goes to the next iteration.

Syntax:

```
continue;
```

Example: Print all non-negative values in an array.

```
#include <iostream>
using namespace std;
 int
main()
{
    int arr[] = {5, 6, 0, -3, 3, -2, 1};
int size = 7; // no of elements in array
for (int i = 0; i < size; i++)
    {
        if (arr[i] < 0)
        {
            // If arr[i] < 0, then skip the current iteration i.e no statements following
// continue will be executed. continue;
        }
        cout<<arr[i]<<" ";
    }
}
```

```
Output:
5 6 0 3 1
```

3) goto statement: In C++ programming, the goto statement is used to alter the normal sequence of program execution by transferring control to some other part of the program. The goto statement can be used to jump from anywhere to anywhere within a function.

Syntax1:

```
goto label; .
.           .
label:
```

Example: Check if a number is even or not and print accordingly using the goto statement.

```
#include <iostream>
using namespace std;
int main() {    int
number;    cin >>
number;    if (number %
2 == 0)        goto
printeven;    else
goto printodd;
 printeven:    cout <<
"Even number";    return
0;  printodd:    cout <<
"Odd number";    return
0;
}
```

```
Input1:
5
Ouptut1:
Odd number

Input2:
4
Output2:
Even number
```

Syntax2:

```
label:
.
.
.       goto
label;
```

Example: Print numbers from 1 to 5 using goto statements.

```
#include <iostream>
using namespace std;
 int
main()
{
    int num = 1; print:
    cout<<num<<" ";
num++;
if(num<=5)
goto print;

return 0;
}

Output:
1 2 3 4 5
```

Disadvantages of using goto Statement:

- The goto statement gives the power to jump to any part of the program but makes the program's logic complex and tangled.
- In modern programming, goto statement is considered a harmful construct and a bad programming practice.
- The goto statement can be replaced in most C++ programs using break and continue statements.

4) return statement: The return statement is used to terminate a function's execution and transfer program control back to the calling function. It can also specify a value to be returned by the function. A function may contain one or more return statements.

Syntax:

```
return [expression];
```

Example:

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

// int return type function to calculate sum
int SUM(int a, int b) {     int s1 = a + b;
return s1;
}

// void returns type function to print
void Print(int s2) {     cout << "The
sum is " << s2;     return; }  int
main() {     int n1 = 10;     int n2 =
20;     int summ = SUM(n1, n2);
Print(summ);     return 0;
}
```

```
Output:
The sum is 30
```

Previous Next

Introduction

Introduction

Loops in C++ come into use when we need to execute a block of statements repeatedly.

There are mainly two types of loops:

1. **Entry Controlled Loops**: In an entry controlled loop, the test condition is tested before entering the loop body. **For Loop** and **While Loop** are entry controlled loops.

2. **Exit Controlled Loop**: In the exit controlled loop, the test condition is tested or evaluated at the end of the loop body. Therefore, the loop body will execute at least once, irrespective of whether the test condition is true or false. The **do-while loop** is an exit controlled loop.

In the later sections, we are going to cover all the loops in detail.

Previous Next

For Loop

i) For loop: A for loop is a repetition control structure that allows us to write a loop that is executed a specific number of times. The loop enables us to perform ‘n’ number of steps together in one line. **Syntax:**

```
for (initialization expression; condition expression; update expression) {
    // body of-loop
}
```

Here,

- **initialization -** Initialization is executed first done only once, where the loop variables are initialized to some value.
- **condition -** This expression involves testing the condition, if the condition evaluates to true then the body of the loop is executed however, if the condition evaluates to false then we exit from the loop, and the control jumps to the statements following the loop.
- **update - After the execution of the body of the loop, the control jumps to the update expression where the loop variables are updated by some value.**

After updating the values of the loop variables, again the condition is evaluated and if it's true then the body of the loop is executed and the process continues until the condition expression evaluated to false and we exit from the loop.

Example: Print Hello World 5 times using for loop.

```
#include <iostream>
using namespace std;
int main() {
    for (int i = 1; i <= 5; i++) {
cout << "Hello World\n";
    }
return 0;
}
```

```
Output:
Hello World
Hello World
Hello World
Hello World
Hello World
```

Example: Iterate array elements using the auto keyword.

```
#include <iostream>
using namespace std;
int main() {
```

```
    int arr[] {40, 50, 60, 70, 80, 90, 100};
for (auto element: arr)          cout <<
element << " ";     return 0;
}

Output:
40 50 60 70 80 90 100
```

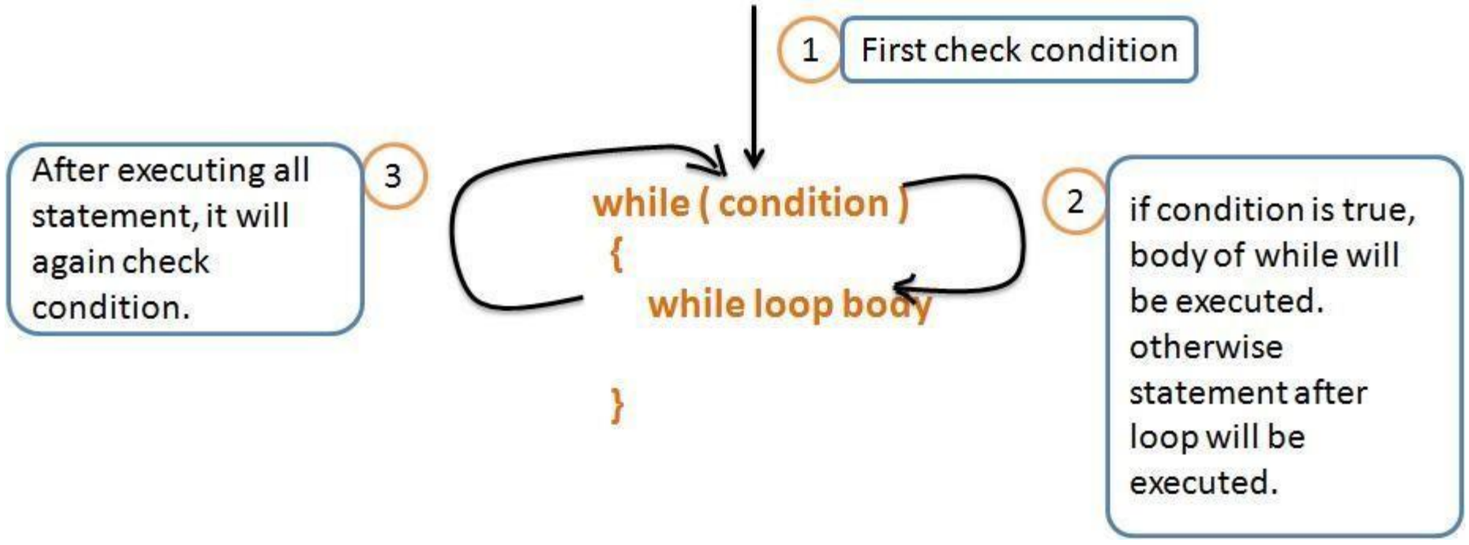
Previous Next

While Loop

ii) While loop: In a while loop, the condition is evaluated first, and if it returns true, then the statements inside the while loop execute. This repeatedly happens until the condition returns false. When the condition returns false, the control comes out of the loop and jumps to the program's next statement after the while loop. **Syntax:**

```
initialization; while
(condition) {     //
statements
update_expression;
}
```

Note: The important point to note when using while loop is that we need to use increment or decrement statement inside while loop so that the loop variable gets changed on each iteration, and at some point, the condition returns false. This way, we can end the execution of the while loop. Otherwise, the loop would execute indefinitely.



Example:

```
#include <iostream>
using namespace std;
int main() {     int
i = 1;
    // The loop would continue to print the value of i till
//the condition is false.
    while (i <= 5) {
        cout << "Value of i is: " << i << endl;
i++;
    }
}
```

```
Output:
Value of i is: 1
Value of i is: 2
Value of i is: 3
Value of i is: 4
Value of i is: 5
```

Previous
Next **do-while Loop iii)do-while loop:** The do...while loop is similar to a while loop with one difference, i.e. the condition is tested at the end of the loop-body thus the loop-body is executed at least once irrespective of the condition which makes it an exitcontrolled loop.

Syntax:

```
initialization; do
{
    // statements
update_expression;
}
while (condition);
```

Here,

- First, the body of the loop is executed, then the condition is evaluated
- If the **condition** evaluates to **true**, the loop's body inside the do statement is executed again.
- The **condition** is evaluated once again and the process continues until the condition evaluated to false.

Example:

```cpp
#include <iostream>
using namespace std;
 int
main()
{
    // Initialization expression
int i = 2;     do
    {
        // Statement of body of loop
        cout << "Inside body of do-while loop" << endl;
        // Update expression
i++;
    } while (i < 1); // Update expression
}
```

```
Output:
Inside body of do-while loop
```

Previous Next for_each Loop iv)for_each loop: Apart from the standard loops such as “for, while and do-while,” C++ also allows using another functionality that solves the same purpose termed **“for_each”** loops. This loop accepts a function that executes over each of the container elements.

This loop is defined in the header file “algorithm” and hence has to be included for this loop’s successful operation. Advantages of for_each loop:

- It is versatile, i.e., it can work with any container.
- It reduces chances of errors one can commit using generic for loop
- It makes code more readable
- for_each loops improve the overall performance of code **Syntax:**

```
for_each (InputIterator first, InputIterator last, Function fn)
```

Here,

- **first −** Input iterator to the initial position.
- **last −** Final iterator to the final position.
- **fn −** Unary function that accepts an element in the range as an argument.

Example:

```
#include <iostream>
#include <algorithm>
using namespace std; int
print_even(int n) {
if (n % 2 == 0)
cout << n << ' ';
} int main()
{
    int arr[5] = {1, 2, 3, 4, 5};
    cout << "The Array contains the following even numbers" << endl;
for_each(arr, arr + 5, print_even);     return 0;
}
Output:
The array contains the following even numbers
2 4
```

for_each loop can be executed using the keyword “for.”

Example:

```
#include <iostream>
using namespace std;
int main() {
    int arr[] = {1, 2, 3, 4};
    // Printing elements of an array using foreach loop
for (int x: arr)          cout << x << ' ';     return
0;
}
Output:
1 2 3 4
```

Previous Next
Introduction

Introduction

A Function is a set of statements designed to perform a specific task when called. The function is like a black box that can take specific input(s) as its parameters and output a value that is the return value. A function is created to use it as many times as needed just by using the function’s name. You don't need to type the function’s statements repeatedly ; you just have to call the function. A function is a block of code that only runs when called. Every C++ program has at least one function, which is main().

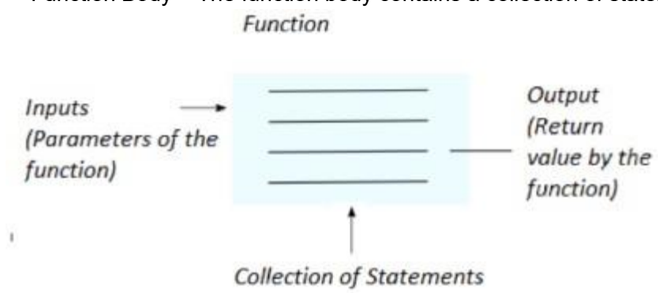
Defining a function: It provides information about the function attributes such as its name, return type, parameters, and body.

Syntax to define a function:

```
return_type functionName(parameter1, parameter2, ...) {
    // function body
}
```

Here,

- Return Type − A function may or may not return a value. The return_type of the function is the data type of the value returned . In some cases functions do not return a value, in that case the return type is the keyword void. Some other return_type can be int, string, vector, float, etc.
- Function Name − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- Parameters −When a function is invoked, you pass a value to the parameter. This value is referred to as an actual parameter or argument. The parameter list refers to the type, order, and the, number of function parameters. Parameters are optional; that is, a function may contain no parameters.
- Function Body − The function body contains a collection of statements that define what the function does.



Example: In this example, we define a function that accepts two parameters and returns the maximum of two numbers.

```
int max(int x, int y) {
if (x > y)
return x;     else
return y;
}
Here, the return type is int, so it means it will return an integer value where the function will be called.
```

C++ Function Prototype

A function prototype is a declaration of the function that tells the program about the type of the value returned by the function and the number and type of arguments.

Function prototyping is one very useful feature of the C++ function. A function prototype describes the functional interface to the compiler by giving details such as the number and type of arguments and return values.

Function Declaration: When a function is defined before the main() function in the program, then function declaration is not required, but writing the function after the main() function requires function declaration first, else there will be a compilation error.

Syntax to declare a function: `return_type function_name( parameter list );`

Example:

```
int sum(int num1,int num2);
Can be declared as int
sum(int,int)
```

Example:

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

//Function declaration int
sum(int, int);
 int main()
{
    //Calling the function
cout << sum(50, 60);     return
0;
}

/* Function is defined after main*/
int sum(int num1, int num2) {
int num3 = num1 + num2;     return
num3;
}
```

```
Output:
110
```

Function Calling in C++: Once we declare a function, we use it to perform some specific task. When we call a function, the program controls transfer to the called function.

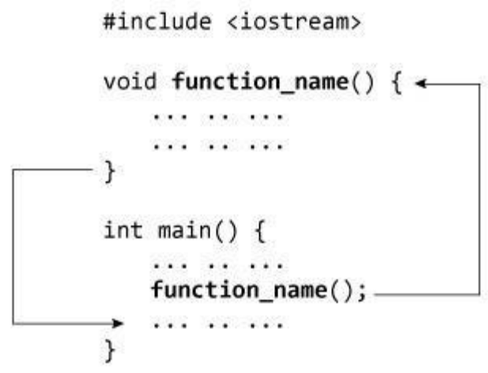
Syntax to call a function:
`function_name( parameters );`

Example:

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

int max(int x, int y) {
if (x > y)
return x;     else
return y;
}
 //Main Code
int main() {
int a = 10;
int b = 20;
// Method
Calling
int maximum
= max(a, b);
cout <<
maximum;
return 0; }
```

```
Output:
20
```

How does function calling work?



Example:

```cpp
#include <iostream> using
namespace std;  int
findSum(int a, int b) {
int sum = a + b;     return
sum; }
 //Main Code int main() {
cout << findSum(50, 60);
return 0; }
```

```
Output:
110
```

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

Why do we need function?

- **Reusability:** Once a function is defined, it can be used repeatedly. You can call the function as many times as 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 circle area or make a function to find the circle area and invoke the function whenever needed.
- **Neat code:** A code created with a function is easy to read and dry run. You don't need to repeatedly type the same statements; instead, you can invoke the function whenever needed.
- **Modularisation:** Functions help in modularizing code. Modularisation means divides the code into small modules, each performing a specific task. Functions allow in doing so as they are the program's tiny fragments designed to perform the specified task.
- **Easy Debugging:** It is easy to find and correct the error in function compared to raw code without function where you must correct the error everywhere the specific task of the function is performed.

Previous Next

Types of Functions

Types of Functions in C++

There are two types of function in C++ they are:

- **Built-in Functions**
- **User-defined Functions**

We will closely look at both of them in the next notes.

Previous Next

Built-in Function Note

1. Built-in Functions: Built-in functions are also known as library functions. These functions are not required to be declared, raised, and defineed by the user, these functions are already present in the C++ libraries such as iostream, cmath, etc. Some predefined functions are: sqrt(), max(), min(), round(), etc. these are defined inside the cmath library.

Example 1: Find the maximum of two numbers using the built-in function.

```
#include <iostream>
#include <cmath>
using namespace std;
int main() {
    // Maximum of two numbers using max()
int maximum = max(100, 30);     cout <<
maximum;     return 0;
}
```

```
Output:
100
```

Example 2: Find the minimum of two numbers using the built-in function.

```
#include <iostream>
#include <cmath>
using namespace std;
int main() {
    // Maximum of two numbers using min()
int minimum = min(100, 30);     cout <<
minimum;     return 0;
}
```

```
Output:
30
```

Example 3: Find the square root of a number using the built-in function.

```
#include <iostream> #include
<cmath> using namespace std;
int main() {     // Finding sqrt
using sqrt()     double root =
sqrt(144);     cout << root;
return 0;
}
```

```
Output:
12
```

Example 4: Find string length using the built-in function.

```
#include <iostream>
using namespace std;
int main() {
    // Finding length using length()
string a = "Coding Ninjas";     cout
<< a.length();     return 0;
}
```

```
Output:
13
```

Example 5: Convert the string into the upper case using the built-in method.

```
#include <iostream>
#include<cctype>
using namespace std;
int main() {
    // Converting to uppercase using toupper()
string a = "Coding Ninjas";     for (int i =
0; i < a.length(); i++)
putchar(toupper(a[i]));     return 0;
}
```

```
Output:
CODING NINJAS
```

Previous Next

User-defined Function Note

2. User-defined Functions: The function written by the user or programmer's function is called the user-defined method. We can modify these methods based on our requirements. Let's discuss the user-defined functions with all four combinations of arguments and return type.

1. Function with no argument and no return value
2. Function with arguments but no return value
3. Function with no arguments but returns a value
4. Function with arguments and return value

i) Function with no argument and no return value: When a function has no arguments, it does not receive any calling function data. Similarly, when it does not return a value, the calling function does not receive any called function data.

Syntax:

```
Function declaration : void function();
Function call : function(); Function
definition :
void function()
                        {
                            statements;
                        }
```

Example: Check whether a number is even or odd.

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

// Function to check a number is even or odd
void checkEvenOdd() {     // Number to be
checked     int num = 24;     if (num % 2 ==
0)         cout << "Even Number";     else
        cout << "Odd Number";
}

//Main Code int
main() {
//Function
Calling
checkEvenOdd();
return 0;
}
```

Output:

```
Even Number
```

ii) Function with arguments but no return value: When a function has arguments, it receives data from the calling function, but it returns no values. **Syntax:**

```
Function declaration : void function(int x);
Function call : function(x); Function
definition :                        void
function(int x)
                    {
statements;
                    }
```

Example: Check whether a number is even or odd.

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

// Function to check a number is even or odd
void checkEvenOdd() {     if (num % 2 == 0)
cout << "Even Number";     else
        cout << "Odd Number";
}
 //Main Code
int main() {
    // Number to be checked
int num = 24;     //Function
Calling
checkEvenOdd(num);
return 0;
}
```

```
Output:
Even Number
```

iii) Function with no arguments but returns a value: There could be occasions where we may need to design functions that may not take any arguments but returns a value to the calling function. **Syntax:**

```
Function declaration : int function();
Function call : function(); Function
definition :
int function()
                    {
```

```
                        Statements;
return x;
                    }
```

Example: Add two numbers.

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

// Function to return the sum of two numbers
int sumOfTwoNumbers() {     int a = 10;
int b = 20;     int sum = a + b;     return
sum;
}

//Main Code int
main() {
    // No arguments passed in the function
int sum = sumOfTwoNumbers();     cout <<
sum;
}
```

```
Output:
30
```

iv) Function with arguments and returns a value :

Syntax:

```
Function declaration : int function(int x);
Function call : function(x); Function
definition :
int function(x)
{
Statements;
return x;
                    }
```

Example: Add two numbers.

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

```
// Function to return the sum of two numbers
int sumOfTwoNumbers(int a, int b) {     int
sum = a + b;     return sum;
}
 //Main Code int main() {     int a
= 10;     int b = 20;     int sum =
sumOfTwoNumbers(a, b);     cout <<
sum;
}
```

```
Output:
30
```

Previous Next

Passing Parameters to Functions

The parameters passed to function are called actual parameters. In the example below, a and b are actual parameters.
The parameters received by the function are called formal parameters. In the example below, x and y are formal parameters.

Example:

```
#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;
}

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

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

There are two most popular ways to pass parameters, which are:

1. **Pass by Value**
2. **Pass by Reference**

i) Pass by Value: Here, values of actual parameters are copied to the function's formal parameters, and the two types of parameters are stored in different memory locations. Any changes made inside functions are not reflected in the actual parameters of the caller.

Example:

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

void func(int x) {
x = 50;
    cout << "Value of x in func function: " << x << endl;
}

//Main Code int
main() {
int x = 10;
func(x);
    cout << "Value of x in main function: " << x;
}
```

```
Output:
Value of x in func function: 50
Value of x in main function: 10
```

ii) Pass by Reference: Here, both actual and formal parameters refer to the same location, so any changes made inside the function are reflected in the actual parameters of the caller.

Example:

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

void func(int &x) {
x = 50;
    cout << "Value of x in func function: " << x << endl;
}

//Main Code int
main() {
int x = 10;
func(x);
    cout << "Value of x in main function: " << x;
}
```

```
Output:
Value of x in func function: 50
Value of x in main function: 50
```


Function Overloading

Function overloading is a programming feature in C++ that allows having more than one function having the same name but different parameter list, it means the data type and sequence of the parameters or the different number of parameters, for example, the parameters list of function **myfuncA(int x, float y)** is **(int, float)** which is different from the function **myfuncA(float x, int y)** parameter list **(float, int)**.which is also different from **myfuncA(float a)** parameter list **(float).** Function overloading is related to compile-time polymorphism. **Example: Following are the 4 overloaded functions**

```cpp
int test() {} int test(int
a) {} float test(double a)
{} int test(int a, double b)
{}
```

Note: Functions can not be overloaded if they differ only in the return type.

Example: Here, both functions have the same name, same type, and the same number of arguments. Hence, the compiler will throw an error.

```
int test(int a) {} double
test(int b) {}
```

Ways to overload a function: There are two ways to overload a function, and they are:

- By Changing the number of arguments.
- By Changing the data type.

1. Function overloading with changing the number of arguments.

In this example, we have created two functions, the first add() performs the addition of the two numbers, and the second add() performs the addition of the three numbers. Let's look at the example:

```
#include <iostream>
using namespace std;
// Function with two parameters int
add(int num1, int num2) {
return num1 + num2;
}
// Function with three parameters int
add(int num1, int num2, int num3) {
return num1 + num2 + num3;
} int main()
{    cout
<< add(10,
20) << endl;
cout <<
add(10, 20,
30);
return 0;
}
Output:
30
60
```

2. Function overloading with changing the data type of arguments.

In this example, we have created two add() functions with different data types. The first add() takes two integer arguments and the second add() takes two double arguments.

```
#include <iostream>
using namespace std;
// Function with two integer parameters
int add(int num1, int num2) {
return num1 + num2;
```

```
}
// Function with two double parameters double
add(double num1, double num2) {     return
num1 + num2;
}
int main(void) {
    cout << add(10, 20) << endl;
cout << add(10.4, 20.5);     return
0;
}
Output:
30
30.9
```

Default Arguments

A default argument is a value provided in a function declaration automatically assigned by the compiler if the caller of the function doesn't provide a value for the argument with a default value. However, if arguments are passed while calling the function, the default arguments are ignored.

Example: A function with default arguments can be called with 2 or 3 or 4 arguments.

```
#include<iostream>
using namespace std;
int add(int x, int y, int z = 0, int w = 0) {
return (x + y + z + w);
} int main()
{
    cout << add(10, 20) << endl;
cout << add(10, 20, 30) << endl;
cout << add(10, 20, 30, 40) << endl;
return 0;
}
Output:
30
60
100
```

Previous Next

Pointers Introduction

Introduction to Pointers

Pointers are one of the most important aspects of C++. Pointers are another type of variable in CPP, and these variables store addresses of other variables.

While creating a pointer variable, we need to mention the type of data whose address is stored in the pointer. e.g., to create a pointer that stores the address of an integer, we need to write:

```
int* p;
```

This means that p will contain the address of an integer. So, if a pointer is going to store the address of datatype X, it will be declared like this:

```
int* p;
```

Address of Operator (&)

The address of a variable can be obtained by preceding the name of a variable with an ampersand sign (&), known as the *address-of operator*.

Example:

```
cout << (&var) << endl;
```

This would print the address of variable *var*; by preceding the name of the variable var with the *address-of operator* (&), we are no longer printing the content of the variable itself but its address.

Example:

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

int main() {
    int i = 10;
    int * p;
    p = & i;
    cout << "Address of the variable i is " << p << endl;
    cout << "Address of the pointer p is " << & p;
    return 0;
}
```

```
Output:
Address of the variable i is 0x7fff32eb4bb4
Address of the pointer p is 0x7fff32eb4bb8
```

Here, we have an integer i and an integer pointer p. The address of(&) operator is used to address i in p that returns the variable's address. e.g., &i will give us the address of variable i.

Dereference Operator

As just seen, a variable that stores the address of another variable is called a pointer. Pointers are said to "point to" the variable whose address they store.

An exciting property of pointers is that they can access the variable they point to directly. This is done by preceding the pointer name with the dereference operator (*). The operator itself can be read as "value pointed to by."

The reference and dereference operators are thus complementary:

- & is the address-of operator and can be read simply as "address of."
- * is the dereference operator and can be read as "value pointed to by."

Note: The asterisk (*) used when declaring a pointer only means that it is a pointer (it is part of its type compound specifier) and should not be confused with the dereference operator seen above, but which is also written with an asterisk (*). They are simply two different things represented with the same sign.

Example:

```
#include <iostream>
using namespace std;
int main() {
    int firstvalue = 5, secondvalue = 15;
    char thirdvalue = 'a';
    int * p1, * p2;
    char * p3;
    p1 = & firstvalue; // p1 = address of firstvalue
    p2 = & secondvalue; // p2 = address of secondvalue
    p3 = & thirdvalue; // p3 = address of thirdvalue
    * p1 = 10; // value pointed to by p1 = 10
    * p2 = * p1; // value pointed to by p2 = value pointed to by p1
    p1 = p2; // p1 = p2 (value of pointer is copied)
    * p1 = 20; // value pointed to by p1 = 20
    * p3 = 'b'; // value pointed to by p3 = 'b '
    cout << "firstvalue is " << firstvalue << endl;
    cout << "secondvalue is " << secondvalue << endl;
    cout << "thirdvalue is " << thirdvalue << endl;
    return 0;
}
```

```
Output:
firstvalue is 10
secondvalue is 20
thirdvalue is b
```

Note: While solving pointers questions, you should use pen and paper and draw better ideas.

Pointer Arithmetic

Arithmetic operations on pointers behave differently than they do on simple data types we studied earlier. Only addition and subtraction operations are allowed; the others aren't allowed on pointers. But both addition and subtraction have slightly different behavior with pointers, according to the size of the data type to which they point.

For example, char always has a size of 1 byte, short is generally larger than that, and int and long are even larger; the exact size of these depends on the system. For example, let's imagine that in a given system, char takes 1 byte, short takes 2 bytes, and long takes 4.

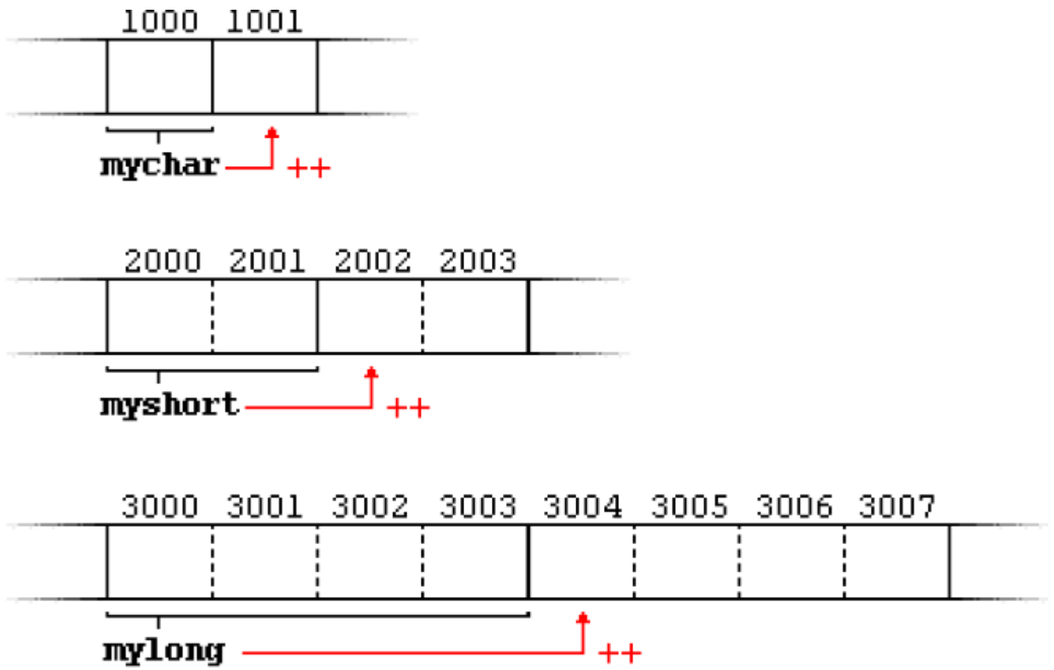
Suppose now that we define three-pointers:

```
char *mychar;
short *myshort;
long *mylong;
```

and they point to the memory locations 1000, 2000, and 3000, respectively.
Therefore, if we write:

```
++mychar;
++myshort;
++mylong;
```

mychar, as one would expect, would contain the value 1001. But not so obviously, myshort would contain the value 2002, and mylong would contain 3004, even though they have each been incremented only once. The reason is that, when adding one to a pointer, the pointer is made to point to the following element of the same type, and, therefore, the size in bytes of the type it points to is added to the pointer.



This is applicable both when adding and subtracting any number to a pointer. It would happen the same if we wrote:

```
mychar = mychar + 1;
myshort = myshort + 1;
mylong = mylong + 1;
```

Essentially, these are the four possible combinations of the dereference operator with both the prefix and suffix versions of the increment operator (the same applies to the decrement operator).

Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables related to each other, such as the same array elements, then p1 and p2 can be meaningfully compared.

Previous
Next

Constant variables

Constant Variables

We are now moving on to the constant variables identified by the const keyword in C++. As the name suggests, if we declare any keyword as constant, we can’t change its value throughout the program.

Note: The constant variable needs to be assigned during initialization only; else, it will store garbage values that can’t be changed further.

Syntax: `const datatype variable_name = value;`

Example:
```
const int a = 5;
```

Previous Next

Types of Pointers

Types of pointers in C++

There are different types of pointers in C++, and they are as follows:

- Null Pointers
- Double Pointers
- Void Pointers
- Wild Pointers
- Dangling Pointer

1. Null Pointers:

A NULL pointer is a pointer that is pointing to nothing. If we don’t have the address to be assigned to a pointer, we can use NULL.

Advantages of Null pointer are:

- We can initialize a pointer variable when that pointer variable is not assigned any actual memory address.
- We can pass a null pointer to a function argument when we are unwilling to pass any actual memory address.

Example:
```
int *p; //Contains garbage value
int *p = NULL; //NULL is constant with vaue 0 int
*q = 0; // Same as above
```

Here, we have created a pointer variable that contains garbage values. To dereference the pointer, we have initialized it to NULL to avoid unexpected behavior.

Note: An uninitialized pointer variable contains garbage; this will lead to unexpected results or segmentation faults. Hence, we should never leave a pointer uninitialized and instead.

Example:

```
#include <iostream>
using namespace std;
int main() {     //
Null Pointer      int
* ptr = NULL;
    cout << "The value of ptr is " << ptr;
return 0;
}
Output:
The value of ptr is 0
```

2. Double Pointers:

We can create a pointer to a pointer that in turn may point to data or another pointer. The first pointer is used to store the address of the variable. And the second pointer is used to store the address of the first pointer. That is why they are also known as double pointers.

Example:

```
int a = 10; int
*p = &a;  int
**q = &p;
```

Here q is a pointer to a pointer, i.e., a double-pointer, as indicated by **.

Example:

```
#include<iostream> using namespace std;
int main() {     int a = 10;     int * p
= & a; //pointer     int ** q = & p;
//pointer-to-pointer
    /* Next three statements will print same value i.e. address of a */
cout << & a << endl;     cout << p << endl;     cout << * q << endl;
    /* Next two statements will print same value i.e. address of p */
cout << & p << endl;     cout << q << endl;
    /* Next three statements will print same value i.e. value of a */
cout << a << endl;     cout << * p << endl;     cout << ** q << endl;
return 0;
}
Output:
0x7ffcab7af9ac
0x7ffcab7af9ac
0x7ffcab7af9ac
0x7ffcab7af9b0
```

```
0x7ffcab7af9b0
10
10
10
```

3. Void Pointers:

A void pointer is a generic pointer; it has no associated type with it. A void pointer can hold an address of any type and can be typecasted to any type. Thus we can use the void pointer to store the address of any variable. Void pointer is declared by:

```
void *ptr;
```

Note:

1. void pointers **cannot be dereferenced**. It can, however, be done using typecasting the void pointer.
2. Pointer arithmetic is not possible on pointers of void due to lack of concrete value and size.

Example:

```
#include <iostream>
using namespace std;
int main() {
void * ptr;     int
i = 10;
    // assign int address to void
ptr = & i;
    cout << "Address of variable i " << & i << endl;
    cout << "Address where the void pointer is pointing " << ptr <<  endl;
return 0;
}
Output:
Address of variable i 0x7ffc848c25f4
Address where the void pointer is pointing 0x7ffc848c25f4
```

4. Wild Pointers:

A pointer behaves like a wild pointer when declared but not initialized. So, they point to any random memory location.

Example:

```
int *ptr; //wild pointer
*ptr = 5;
```

Note: If a pointer p points to a known variable, it’s not a wild pointer.

Example:

```
int * p; /* wild pointer */ int
a = 10;
p = & a; /* p is not a wild pointer now*/
* p = 12; /* This is fine. Value of a is changed */
```

In the above program, p is a wild pointer till this points to a.

5. Dangling Pointers:

A dangling pointer is a pointer pointing to a memory location that has been freed (or deleted). There are **three** different ways where Pointer acts as a dangling pointer.

a)Function Call

The pointer pointing to the local variable becomes dangling when the local variable is not static. **Example:**

```
#include<iostream>
using namespace std;
int * fun() {
int x = 10;
return &x;
} int main() {
int * p = fun();
    // p points to something which is not
    // valid anymore
cout << * p;     return
0;
}
```

b) Deallocation of memory

Deallocating a memory pointed by a pointer causes a dangling pointer. **Example:**

```
#include<iostream> using
namespace std; int
main() {
    //dynamic memory allocation.
    int * p = (int * ) malloc(sizeof(int));
    //after calling free() p becomes a dangling pointer
free(p);
    //now p no more a dangling pointer.
    p = NULL;
return 0;
}
```

c) Variable goes out of scope

When a pointer goes out of scope where it is valid, then it becomes a dangling pointer.

Example:

```
#include<iostream>
using namespace std;
void main() {
int * ptr;     {
int ch;          ptr
= & ch;
    }
    // Here ptr is dangling pointer
}
```

Previous Next

Address Typecasting

Address Typecasting

While declaring a pointer, why can’t we just write like this:

```
int a = 5; pointer
p = &a;
```

Why do we have a complicated syntax like:

```
int a = 5; int
*p = &a;
```

It’s generally because we need to specify that when we invoke a particular pointer, how will the compiler know what type of value a pointer has stored, and while invoking/transferring data, how much space needs to be allotted to it.

That is why while declaring a pointer, we start with the data type and then assign the name to the pointer. That data type specifies what type of value we are storing in the pointer.

The term **typecasting** means assigning one type of data to another type, like storing an integer value to a char data type.

Example:

```
int x= 65; char
c = x;
```

When you check the value stored in variable ‘c ‘, it will print the ASCII value stored at that integer variable, i.e., ‘x’, and print ‘A’ (whose ASCII value is 65).

This type of typecasting done above is known as **implicit typecasting** as the compiler itself interprets the conversion of integer value to ASCII character value.

Example:

```
int i = 65;  int
*p = &i;
char *pc = (char*) p;
```

You can see that in the third line, we can’t directly do like this:

```
char *pc = p;
```

This will give an error as we are trying to store an integer-type pointer value into a character-type value. To remove the error, we have to type-cast ourselves by providing (char*) on the right-hand side as done in the code above. This type of typecasting is known as **explicit typecasting**.

Previous Next

Reference and Pass by Reference

Usually (&) symbol is known as a reference operator, which means ‘Address of operator.’
This operator is used to copy the values of any variable and guarantee that the reflected changes will also be visible in the copied variable.

Example:

```
#include<iostream>
using namespace std;
int main() {     int
a = 5;     int b =
a;     a++;
    cout << "b = " << b << endl;
return 0;
} Output:
b = 5
```

Here, it means that only the value is copied, and when the value of the variable is increased by one, then the changes are not reflected in variable ‘b.’

Example:

```
#include<iostream>
using namespace std;
int main() {     int
a = 5;     int & b =
a;     a++;
    cout << "b = " << b << endl;
return 0;
} Output:
b = 6
```

Here, (&b=a) means that now variables b and a are pointing to the same address and making changes in any of them reflected in both of the variables.

This concept of referencing the variables is useful when we want to update the value passed to the function. When we normally pass a value to a variable, then a copy of those is created in the system, and the original values remain unchanged. But by passing the reference, the changes are also reflected in the original variables as there is no extra copy created. The first type of argument passing is known as **pass by value,** and the later one is known as **pass by reference.**

The syntax for pass by reference:

```
#include <iostream>
using namespace std;
void fun(int & a) {
a++; } int main() {
int a = 5;
fun(a);
    cout << "a = " << a << endl;
} Output:
a = 6
```

Advantages of Pass by reference:

- Reduction in-memory storage.
- Changes can be reflected easily.

Previous Next

Dynamic Memory Allocation

Dynamic Memory Allocation

Memory allocation done at the time of execution(run time) is known as dynamic memory allocation. Functions calloc() and malloc() support allocating dynamic memory. The Dynamic allocation space is allocated using these functions when the value is returned by functions and assigned to pointer variables.

malloc() function in c++

The malloc() function which is defined under <cstdlib> header file is uded to allocate block of memory and returns a void pointer to the allocated memory block's first byte if the allocation succeeds.

Syntax: `data_type* malloc(size_t sizeof(data_type));`

Example:

```
#include <iostream>
#include <cstdlib> using
namespace std;
 int main()
{
        int n = 5;
    int *ptr = (int * ) malloc(n * sizeof(int));
for (int i = 0; i < n; i++) {         ptr[i] = i
* 2;     }
     for (int i = 0; i < n; i++) {
        /* ptr[i] and *(ptr+i) can be used interchangeably */
cout << ptr[i] << endl;     }
    free(ptr);
return 0;
}
```

```
Output:
0
2
4
6
8
```

calloc() function in c++

The calloc() function works same as that of malloc with the only difference being that it sets all the bits of the allocated emeory to zero and returns a pointer to the allocated memory block's first byte if the allocation succeeds.

Syntax: `data_type* calloc(size_t num, sizeof(data_type));`

Example:

```
#include <iostream>
#include <cstdlib> using
namespace std;
 int main()
{
      int *ptr = (int *)calloc(5, sizeof(int));
      for (int i = 0; i < 5; i++) {
      /* ptr[i] and *(ptr+i) can be used interchangeably */
cout << ptr[i] << endl;     }
      free(ptr);
return 0;
}
```

Output:

```
0
0
0
0
0
```

Runtime error can happen if the variable that is used to allocate the memory block, contains a garbage value or some type of undefined value. Generally, we have two types of memory in our systems:

- **Stack memory:** It has a fixed value of size for which an array could be declared in the contiguous form.
- **Heap memory:** It is the memory where the array declared is not stored in a contiguous way. If we want to declare an array of size 100000, we do not have the contiguous space in the memory of the same size. We will be declaring the array using heap memory as it necessarily doesn't require a contiguous allocation; it will allocate the space wherever it gets and links all the memory blocks together.

Note: Global variables are stored using heap memory.

The memory declaration using heap memory is known as **dynamic memory allocation,** while the one using stack is known as **compile-time memory allocation**.

Syntax: `int *arr = new int[size_of_array];`

For simply allocating variables dynamically, use this:

```
int *var = new int;
```

Here, we are using a **new** keyword**,** which is used to declare the dynamic memory in C++. There are other ways too, but this one is the most efficient.

Note: Stack memory releases itself as the scope of the variable gets over, but in the case of heap memory, it needs to be released manually at last otherwise, the memory gets accumulated and, in the end, leads to memory leakage. For this purpose we use delete keyword.

The syntax for releasing an array: `delete`

```
[ ] arr;
```

The syntax for releasing the memory of the variable: `delete`

```
var;
```

Previous Next

Dynamic Memory Allocation of 2D Arrays

Dynamic Memory Allocation of Two-Dimensional(2D) Arrays

To create a 2D array in the heap memory, we use the following syntax:

Suppose we want to create a 2D array of size n x m, with the name of the array as ‘arr’:

```
#include <iostream>
using namespace std;
int main() {
    int ** arr = new int * [n];
for (int i = 0; i < n; i++) {
arr[i] = new int[m];
    }
return 0; }
```

You can see that first-of-all, we have declared a pointer of pointers of size n, and then at each pointer, while traversing, we allocated the array of size m using a new keyword.

Similarly, we can release this memory using the following syntax:

```
#include <iostream>
using namespace std;
int main() {
    int ** arr = new int * [n];
for (int i = 0; i < n; i++) {
delete[] arr[i];
    }
delete[] arr;
}
```

First-of-all, we cleared the memory allocated to each of the n pointers and then finally deleted those n pointers. This way, using the delete keyword, we can release the memory of the 2D arrays.

Previous Next

Static Memory Allocation

Static Memory Allocation

When the amount of memory to be allocated is known beforehand, i.e., it is known as Static Memory Allocation at the time of compilation. Once the memory is allocated statically, it cannot be deallocated during the program run. So it leads to wastage of storage space. We generally pass constants while declaring arrays.

Syntax: `int A[100];`

Example:

```
#include <iostream>
using namespace std;
int main() {     int a[100];     const int size = 5;     int b[size];     return 0;
}
```

Previous Next

Macros and the global variable

Macros and the Global Variable

Suppose in a code the value pi(3.14) is used many times, instead of always writing 3.14 everywhere in the code, what we can do is define this value to some variable, say pi at the beginning of the program, and now every time we need the value 3.14, we just need to write pi.

What we can do is that we can create a global variable with the value 3.14 and then use it anywhere as desired. There is one issue in using global variables, and that is if someone changes the value of pi in our code at any point of time in any of the functions, we could lose that original value.

To avoid such a situation, what we do is use macros.

The syntax **for using it:**

```
#define pi 3.14
```

This is done after declaring the header files at the beginning itself so that this value can be used in the later encountered functions. Here, what we are doing is that we are declaring the value of the pi as 3.14 using the macros (#) define, which locks the value and makes it unchangeable.

Another advantage is that it prevents extra storage in the memory for declaring a new variable. Using macros, we have specified that the value we are using is a part of the compiler code, hence no need to create any extra memory.

Now, discussing the advantages of the global variable:

- When we want to use the same variable with modified values in each of them, then we can use global variables as the changes done in one function are visible in all the others.
- It saves time for passing the values by reference in the functions.

Due to accidental changes, this method is not preferred much.

Previous Next
Inline and default arguments

Inline and Default Arguments

Disadvantages of using a normal function:

- It uses time as it pauses some portion of code to complete unless we are done with it; we can't move forward.
- Creates a copy of variables each time while calling a function.

To prevent this, what we can do is create inline functions. Inline functions replace the function call with its definition of when invoked.

Syntax:

```
inline fun() {
...
}
```

Just write the keyword inline before the function call.

When it is invoked like this:

```
fun();
```

Inside any other function, then what happens is it gets replaced with the function declaration, hence preventing the pausing of the code.

Disadvantages of using inline functions:

- Code becomes bulky.
- Time taken to copy code can be huge.

Now moving on to the use of default arguments:

Sometimes we are unsure about any value(s) to be passed into the function as an argument. Still, in the further calculations, we need to use them as it is necessary to use them either by defined value or through some default value. Default arguments in C++ serve this purpose. Using these, we can specify a value to any variable in the function declaration to some default value that could be used if no value is passed to it by the function call.

Note: Be careful about using these arguments in the function call; default arguments can only be declared as the rightmost set of parameters.

Example: To calculate the sum of numbers, we are unsure that if we want to find the sum of two or three numbers, then:

```
#include <iostream>
using namespace std;
int sum(int a, int b, int c = 0) { // here, c is the default argument
return a + b + c;
} int main()
{
    int a = 2, b = 3, c = 4;
    cout << sum(a, b) << endl; // here, c will automatically be taken as 0
cout << sum(a, b, c) << endl;
    // as the value of c is provided, the value of c will be 4
return 0;
} Output:
5
9
```

Previous Next

Constant variables

Constant Variables

We are now moving on to the constant variables identified by the const keyword in C++. As the name suggests, if we declare any keyword as constant, we can’t change its value throughout the program.

Note: The constant variable needs to be assigned during initialization only; else, it will store garbage values that can’t be changed further.

Syntax: `const datatype variable_name = value;`

Example:

```
const
int a = 5;
```

Introduction,Creating and Accessing element of an array

Introduction

An array is defined as a **fixed-size** collection of elements of the **same data type** stored in **contiguous memory** locations. It is the simplest data structure where each element of the array can be accessed by using its index. It is used to store multiple values in a single variable instead of declaring separate variables for each value.

Why do we need arrays?

The arrays provide a convenient way of storing variables or a collection of data of the same data type, instead of declaring separate variables for storing each value.

Creating an array

To declare an array, define the variable type, specify the array's name followed by square brackets and specify the number of elements it should store.

Syntax:

```
type arrayName [ arraySize ];
```

Example:

```
// Array declaration by specifying size  int
arr[10];
// Declare an array of user specified size
int n;  cin >> n; int arr[n];
// Taking input in the array  for
(int i = 0 ; i < n ; i++) {
cin >> arr[i]
}
```

Example:

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

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

Dynamically creating array

In C++, a dynamic array can be created using the array version of ‘new’ i.e new[] with the number of elements to be specified within the square brackets with the type name preceding them.

Syntax:

```
pointer_variable = new data_type;
```

Example:

```
// Dynamically Array declaration by specifying size  int
* a = new int[100];
// Declare an array dynamically of user specified size
int n; cin >> n;
int * arr = new int[n];
```

Dynamically Deleting Arrays

A dynamic array is deleted using the array version of ‘delete’ i.e delete[], which tells the CPU to delete multiple variables instead of a single variable. The released memory can be further used to hold other data however, even if you do not release the memory occupied by a dynamic array, it still gets deleted on the termination of the program.

Note: Using ‘delete’ instead of ‘delete[]’ will result in undefined behavior, such as data corruption, memory leaks, crashes, or other problems.

Syntax:

```
delete [] arr;
```

How are arrays stored?

The elements of arrays are stored contiguously, i.e., at consecutive memory locations. The name of the array 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, 119};
```

Suppose the starting address of an array is 1000, then the address of the second and third element of the 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 the array could be accessed using indices. The index of an array starts from 0 to n-1, where n is the array's size.

Syntax:

```
Array_name[index];
```

Example:

```
#include <iostream>
using namespace std;
int main() {
    string person[4] = {"Aman","Rahul","Riya","Arti"};
    cout << person[1];
return 0;
}
Output:
Rahul
```

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

Previous Next

Array Initialization and Passing arrays to function

Array Initialization

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 an 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 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]

Example:

```
#include <iostream>
using namespace std;
 int main() {     int
a[10] = {1, 2};
cout << a[4] << endl;
// 0     int b[10];
cout << b[4]; //
garbage value
return 0;
}

Output: 0
4196608
```

Example: Initializing a Dynamically declared Array.

```
#include<iostream>
using namespace std;
 int main() {     int i, n;
cin >> n;     int * arr = new
int[n];     for (i = 0; i <
n; i++) {         arr[i] = i
+ 1;
    }
    cout << "Array elements " << endl;
for (i = 0; i < n; i++) {         cout
<< arr[i] << " ";
    }     delete
[] arr;     return
0;
}
 Input:
5
Output:
```

```
Array elements
1 2 3 4 5
```

Passing arrays to a function

Just like variables, arrays can also be passed to a function as an argument. And, also we can return arrays from a function. We can pass an array to a function in two ways, and they are:

- Pass by Value
- Pass by reference

1. Passing array to function using pass by value method: In this type of function call, the actual parameter is copied to the formal parameters.

Syntax of the function call to pass an array:

```
function_name(array_name, array_size);
```

Syntax of the function definition that takes an **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] << " ";
    } } int
main() {
    int age[5] = {11, 14, 15, 18, 20};
printAge(age, 5);     return 0;
}
```

```
Output:
11 14 15 18 20
```

2. Passing an array to function using Pass by reference method: Passing the address of the array to the function parameter(pointer) instead of copying the actual parameter is called pass by reference. The name of the array itself represents the address of the array i.e address of the first element of the array.

Example:

```
#include<iostream>
using namespace std;
void myfuncn(int
*arr, int n) {
    /* The pointer arr is pointing to the first element of
the array, and the n is the size of the array. */     for
(int i = 0; i < n; i++) {
        cout << "Value of var_arr " << i << " is: " << * arr << endl;
        /*increment pointer for next element fetch*/
arr++;
    } } int
main() {
    int var_arr[] = {11, 22, 33, 44, 55, 66, 77};
myfuncn(var_arr, 7);     return 0;
}
```

```
Output:
Value of var_arr 0 is: 11
Value of var_arr 1 is: 22
Value of var_arr 2 is: 33
Value of var_arr 3 is: 44
Value of var_arr 4 is: 55
Value of var_arr 5 is: 66
Value of var_arr 6 is: 77
```

Previous Next

Introduction,Intialization and Accessing elements of 2D array

Two-Dimensional Array

In C/C++, we can define multidimensional arrays in simple words as an array of arrays. A multidimensional array is an array with **more than one** dimension. In a multidimensional array, **each element is another array** with a smaller number of dimensions. The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays.

Syntax to declare a 2-D array of size x,y :

```
data_type arrayName[x][y];
```

Here, x is the row number, and y is the column number.

A two–dimensional array can be seen as a table with x rows and y columns where the row number ranges from 0 to (x-1), and the column number ranges from 0 to (y-1).

Initialization of Two-Dimensional Array

```
int arr[3][4] = {0, 1 ,2 ,3 ,4 , 5 , 6 , 7 , 8 , 9 , 10 , 11};
```

The above array has 3 rows and 4 columns. The braces' elements from left to right are stored in the table also from left to right. The elements will be filled in the array in the order of the first 4 elements from the left in the first row, the next 4 elements in the second row, etc.

But, the above method is not preferred. A better way to initialize this array with the same array elements is given below:

```
int arr[3][4] = {
   {0, 1, 2, 3} ,   /* Elements of Row indexed by 0 */
   {4, 5, 6, 7} ,   /* Elements of Row indexed by 1 */
   {8, 9, 10, 11}   /* Elements of Row indexed by 2 */
};
```

The above initialization uses inner braces where each such inner brace represents a row.

Accessing Two-Dimensional Array Elements

An element in the 2-dimensional array is accessed using the subscripts, i.e. row index and column index of the array.

Example:

```
int x[2][1];
```

The above example represents the element present in the third row and second column.
Note: In arrays, if the array size is N. Its index will be from 0 to N-1. Therefore, for row index 2 row number is 2+1 = 3.

To output all the elements of a Two-Dimensional array, we can use nested for loops. We will require two for loops—one to traverse the rows and another to traverse columns.

Example:

```
#include <iostream>
using namespace std;
int main()
{
        int arr[3][2] = {{2, 5}, {4, 0}, {9, 1}};
    /* use of nested for loop to access rows of the array */
    for (int i = 0; i < 3; i++) {       // access columns of
the array  for (int j = 0; j < 2; j++) {
                cout << "arr[" << i << "][" << j << "] = " << arr[i][j] << endl;
                }
        }
        return 0;
} Output:
arr[0][0] = 2 arr[0][1]
= 5 arr[1][0] = 4
arr[1][1] = 0 arr[2][0]
= 9 arr[2][1] = 1
```

Previous Next

Allocating Memory Dynamically

Dynamic Memory Allocation of Two-Dimensional Array

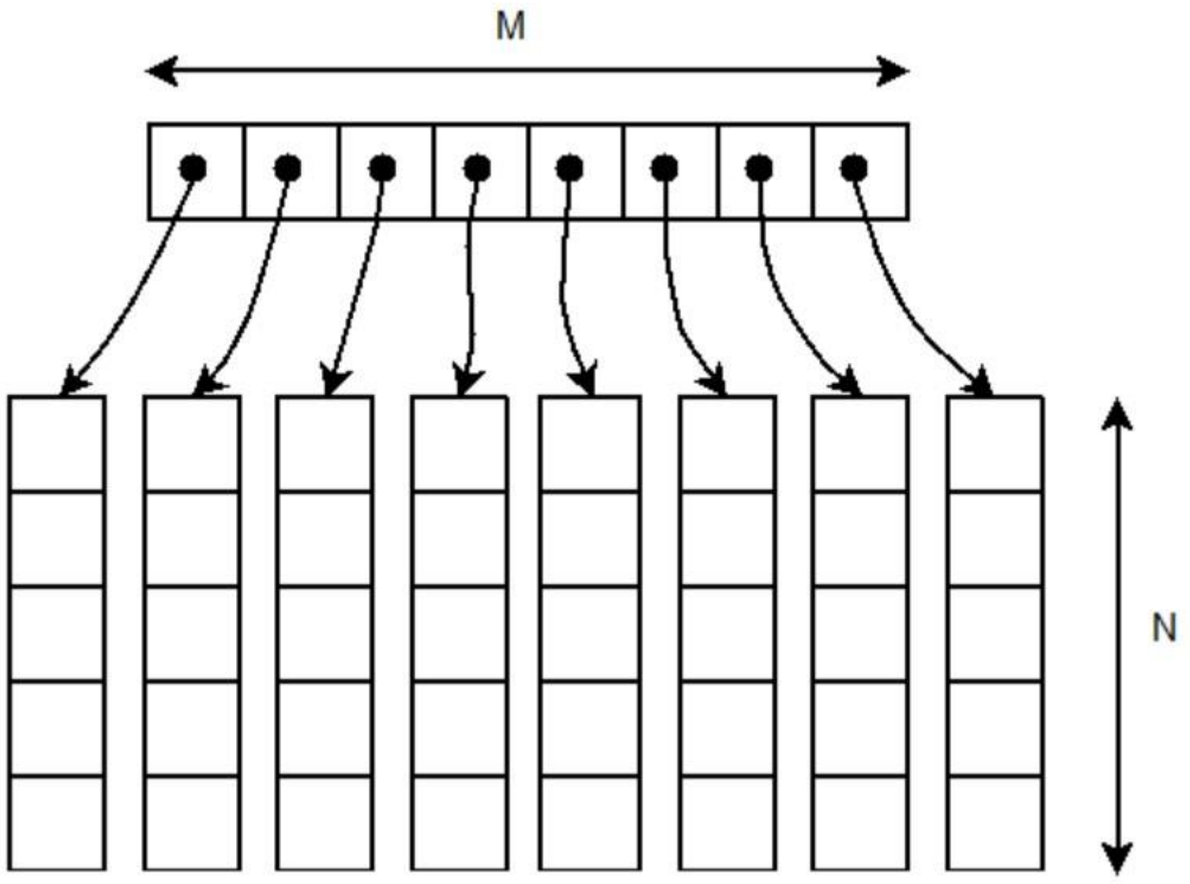
Method1: We can simply allocate one large block of memory of size row*col, where ‘row’ is the number of rows and ‘col’ is the number of columns of the 2-D array, dynamically and assign it to a pointer and then use pointer arithmetic to index the 2D array.

Example:

```
#include <iostream>
using namespace std;
 int
main()
{
    int row, col;

    cout << "Enter the number of rows" << endl;
cin >> row;
    cout << "Enter the number of columns" << endl;
cin >> col;
```

```
    // Dynamically allocate memory of size row*col
int *Arr = new int[row * col];

    // Assign values to the allocated memory
for (int i = 0; i < row; i++)         for
(int j = 0; j < col; j++)
            // Using pointer arithmetic to assign values to the 2-D array
            *(Arr + i * col + j) = rand() % 100;

    // Printing the 2D array
for (int i = 0; i < row; i++)
    {
        for (int j = 0; j < col; j++)
cout << *(Arr + i * col + j) << " ";

cout << endl;
    }

    // Deallocate memory
delete[] Arr;     return
0;
}

Output:
Enter the number of rows
4
Enter the number of columns
5
83 86 77 15 93
35 86 92 49 21
62 27 90 59 63
26 40 26 72 36
```

Method2: We can also dynamically create an array of pointers of size ‘row’ and then dynamically allocate memory of size ‘col’ for each row.



Example:

```
#include <iostream>
using namespace std;
 int main() {
int row, col;
    cout << "Enter the number of rows" << endl;
cin >> row;
    cout << "Enter the number of columns" << endl;
cin >> col;

    // Dynamically create array of pointers of size 'row'
int **Arr = new int *[row];
```

```
    // Dynamically allocate memory of col-size 'col' for each row i.e for row indexed from 0 to row-1
for (int i = 0; i < row; i++)          Arr[i] = new int[col];

    // Assigning values to the allocated memory
for (int i = 0; i < row; i++)          for (int
j = 0; j < col; j++)              Arr[i][j] =
rand() % 100;

    // Printing the 2D array
for (int i = 0; i < row; i++)
    {
        for (int j = 0; j < col; j++)
cout << Arr[i][j] << " ";
         cout <<
endl;
    }

    // Deallocate memory using delete[] operator
for (int i = 0; i < row; i++)          delete[]
Arr[i];
    delete[]
Arr;    return
0;
}

Output:
Enter the number of rows
3
Enter the number of columns
4
83 86 77 15
93 35 86 92
49 21 62 27
```

Previous

Next

Advantages and Disadvantages of Array

Advantages and Disadvantages of Array in C++

Advantages:

- In arrays, the elements can be accessed randomly by using the index number.
- Use of less line of code as it creates a single array of multiple elements.
- Easy access to all the elements of the array.
- Traversal through the array becomes easy using a single loop.
- Sorting becomes easy as it can be accomplished by writing fewer lines of code. *Disadvantages:*
- Since **arrays** are **fixed-size** data structures you cannot dynamically alter their sizes. It creates a problem when the number of elements the array is going to store is not known beforehand.
- **Insertion** and **Deletion** in arrays are difficult and costly since the elements are stored in contiguous memory locations, hence, we need to shift the elements to create/delete space for elements.
- The array is homogeneous, i.e., only one type of value can be stored in the array.
- If more memory is allocated than required, it leads to the **wastage of memory** space and **less allocation of memory** also leads to a problem.
- It does not verify the indexes while compiling the array. If there are any indexes pointed that are more than the dimension specified, we will get run time errors rather than identifying them at compile time.

Previous Next

Example Problems

Example Problems:

Que: Add elements of two arrays.

```
#include <iostream>
using namespace std;
int main() {
    int n; //size of first array
int m; //size of second array
cin >> n >> m;     int array1[n];
int array2[m];     int result =
0;
    // taking input in array1
for (int i = 0; i < n; i++) {
cin >> array1[i];     }     //
taking input in array2     for
(int i = 0; i < m; i++) {
cin >> array2[i];     }     for
(int i = 0; i < n; i++) {
result += array1[i];     }
for (int i = 0; i < m; i++) {
result += array2[i];     }
cout << result;     return 0;
```

```
}
Input:
4                                                                                      34
2 3 4                                                                                  53
10 20 30 40Output:                                         Error! Bookmark not defined.

164
```

Que: Find if an element is present in the array or not.

```
#include <iostream> using namespace
std; int search(int arr[], int n, int
x) {     int i;     for (i = 0; i <
n; i++)         if (arr[i] == x)
return i;     return -1; } int main()
{     int n; //size of array     cin
>> n;     int arr[n];     // taking
input in arr     for (int i = 0; i <
n; i++) {         cin >> arr[i];
}     int x; //Element to be searched
cin >> x;     int result =
search(arr, 10, x);     if (result ==
-1)         cout << "Element is not
present in array";     else
        cout << "Element is present at index " << result;
return 0;
}
Input1:
3
1 2 10
10
Output1:
Element is present at index 2 Input2:
6
34 65 22 14 25 89
10
Output2:
Element is not present in array
```

Que: Find maximum and minimum in the array.

```
#include <iostream> using namespace
std; int main() {     int n; //size
```

```
of array     cin >> n;     int
arr[n];     // taking input in arr
for (int i = 0; i < n; i++) {
cin >> arr[i];     }     int max =
arr[0], min = arr[0];     for (int
i = 1; i < n; i++) {         if
(max < arr[i])             max =
arr[i];         if (min > arr[i])
min = arr[i];     }
    cout << "Maximum Value = " << max << "\n";
cout << "Minimum Value = " << min;
    return 0;
}
Input:
4
1 13 10 21 Output:
Maximum Value = 21
Minimum Value = 1
```

Que: Count pairs with given sum in the array.

```
#include <iostream> using namespace std; int main() {
int n; //size of array     cin >> n;     int arr[n];
// taking input in arr     for (int i = 0; i < n; i++)
{         cin >> arr[i];     }     int sum;     cin >>
sum;     int count = 0; // Initialize result     //
Consider all possible pairs and check their sums
for (int i = 0; i < n; i++)         for (int j = i + 1;
j < n; j++)             if (arr[i] + arr[j] == sum)
count++;     cout << "Count of pairs is " << count;
return 0;
}
Input:
5
1 1 2 2 3
3
Output:
Count of pairs is 3
```

Previous

Next

Introduction, character array, string object

Introduction

Strings, which are widely used in C++ programming, are a sequence of characters. In C++, we have two ways to create and use strings:
1) By creating character arrays and treating them as a string 2)
By creating a string object

By Creating Character Arrays

This type of string is a one-dimensional array of characters terminated by a null character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a null.

Example: `char str[6] = {'H', 'e', 'l', 'l', 'o', '\0'};`

Here, the declaration and initialization create a string consisting of the word "Hello." To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello".
The above statement can also be written as:

```
char str[] = "Hello";
```

Example:

```
#include <iostream>
using namespace std;
int main() {
    char str[50] = "Welcome to Coding Ninjas";
cout << str;     return 0;
}
Output:
Welcome to Coding Ninjas
```

Example: Taking input from the user using cin.

```
#include <iostream>
using namespace std;
int main() {
    // Declaring a string object
char str[50];
    cout << "Enter a string: ";
cin >> str;
    cout << "You entered: " << str << endl;
return 0;
}
```

```
Output:
Enter a string: Welcome to coding ninjas
You entered: Welcome
```

This is an inefficient method of reading user input because when we read the user input string using cin, only the first word of the string is stored in the char array, and the rest get ignored. The cin function considers the space in the string as a delimiter and ignores the part after it.

You can see that only the “Welcome” got captured and the remaining part after space got ignored.

Example: Taking input from the user using cin.get.

```
#include <iostream>
using namespace std;
int main() {
    // Declaring a string object
char str[50];
    cout << "Enter a string: ";
cin.get(str, 50);
    cout << "You entered: " << str << endl;
return 0;
}
Output:
Enter a string: Welcome to Coding Ninjas
You entered: Welcome to Coding Ninjas
```

Disadvantages of this method

1) Size of the string created using char array is fixed; more memory cannot be allocated to it during runtime. Let's say you have created an array of characters with the size 10, and the user enters the string of size 15, then the last five characters would be truncated from the string.

On the other hand, if you create a larger array to accommodate user input, then the memory is wasted if the user input is small and the array is much larger than needed.

2) In this method, you can only use the in-built functions created for arrays which don’t help much in string manipulation.

By Creating a String Object

We can also create a string object for holding strings. Unlike using char arrays, string objects have no fixed length and can be extended as per your requirement. A string variable contains a collection of characters surrounded by double-quotes. The most direct and easiest way to create a string is to write:

```
string str = "Hello world";
```

Here, "Hello world" is a string literal—a series of characters in code that is enclosed in double-quotes. Whenever it encounters a string literal in code, the compiler creates a String object with its value—in this case, Hello world.

Example: Taking string as input.

```
#include <iostream>
using namespace std;
int main() {
    // Declaring a string object
string str;
    cout << "Enter a string: ";
getline(cin, str);
    cout << "You entered: " << str << endl;
return 0;
}
Output:
Enter a string: Welcome to coding ninjas
You entered: Welcome to coding ninjas
```

Here, string str is declared. Then the string is asked from the user. Instead of using cin>> or cin.get() function, you can get the entered line of text using getline().

getline() function takes the input stream as the first parameter, which is cin and str as the line's location to be stored.

Advantage of this method

The advantage of using this method is that you need not declare the size of the string, the size is determined at run time, so this is a better memory management method. The memory is allocated dynamically at runtime, so no memory is wasted.

Previous Next

Accessing the String elements

Accessing the string elements

We can access the characters in a string by referring to their index number inside square brackets [].

Syntax: `string_name[index_number];` **Example:**

```
#include <iostream>
using namespace std;
int main() {
    string str = "Hello World";
cout << str[6];     return 0;
}
Output: W
```

Previous Next

Mutability of strings

Mutability of strings

Mutable data members whose values can be changed in runtime even if they are of a constant type.
To change the value of a specific character in a string, refer to the index number, and use single quotes.

Syntax: `string_name[index_number] = 'new_value';` **Example:**

```
#include <iostream> using namespace std; int main() {
string str = "Hello World";
str[6] = 'J';     cout << str;
return 0;
}
Output: Hello
Jorld
```

Previous Next

String Concatenation

String Concatenation

The end-to-end joining of two strings is called string concatenation. For example, the concatenation of “Coding” and “Ninjas" is “Coding Ninjas"

There are various ways to concatenate strings, and they are as follows:

- ‘+’ Operator
- strcat() method
- append() method

i) + Operator: The ‘+’ operator adds the two input strings and returns a new string that contains the concatenated string.

Syntax: `string new_string = string1 + string2;` **Example:**

```
#include <iostream>
using namespace std;
int main() {
    string str1 = "Coding";
string str2 = " Ninjas!";
    cout << "Concatenated String:" << endl;
str1 = str1 + str2;     cout << str1;
return 0;
}
Output:
Concatenated String:
```

`Coding Ninjas!` **ii) strcat() method:** C++ has a built-in method to concatenate strings. The strcat() method is used to concatenate strings in C++. The strcat() function takes the char array as input and then concatenates the input values passed to the function. strcat() function is included in the cstring library.

Syntax: `strcat(char array1, char array2);` **Example:**

```
#include <iostream>
#include<cstring>
using namespace std;
int main() {
    char str1[50] = "Coding";     char str2[50] = " Ninjas!";     cout << "Concatenated String:" << endl;
strcat(str1, str2);     cout << str1;
return 0;
}
```

```
Output:
Concatenated String:
Coding Ninjas!
```

`iii) append() method:` C++ has another built-in method: append() to concatenate strings. The append() method can be used to add strings together. It takes a string as a parameter and adds it to the end of the other string object.

Syntax:

`string1.append(string2);` **Example:**

```
#include <iostream>
using namespace std;
int main() {
    string str1 = "Coding";
string str2 = " Ninjas!";
    cout << "Concatenated String:" << endl;
str1.append(str2);     cout << str1;
return 0;
}
```

```
Output:
Concatenated String:
Coding Ninjas!
```

Previous Next
String Comparison

String Comparison

To check if the two strings are equal or not, a string comparison is done. Strings in C++ can be compared using either of the following techniques:

- strcmp() method
- compare() method
- Relational Operators

i) strcmp() method: C++ String has got in-built functions for data of string type. To compare two strings, we can use the cstring strcmp() function. The strcmp() function compares two strings in lexicographical manner.

How does this work?

strcmp() starts comparison character by character starting from the first character until the characters in both strings are equal or a NULL character is encountered. If the first character in both strings is equal, this function will check the second character; if this is also equal, it will check the third and so on. This process will be continued until a character in either string is NULL, or the characters are unequal. **Syntax:** `int strcmp(const char * leftStr, const char * rightStr);` *What does strcmp() return?*

This function can return three different integer values based on the comparison, and they are as follows:

- Zero- when both strings are found to be identical
- Greater(+ve) than 0- when the first leftStr is lexicographically greater than second rightStr.
- Smaller(-ve) than 0- when the first leftStr is lexicographically greater than second rightStr.

Example:

```
#include <iostream>
#include<cstring> using namespace
std; int main() {     char
str1[50] = "Coding";     char
str2[50] = " Ninjas!";     int
res1 = strcmp(str1, str2);     int
res2 = strcmp(str2, str1);
cout << res1 << endl;     cout <<
res2 << endl;     return 0;
}
Output:
35
-35
```

ii) compare() method: C++ has in-built compare() function in order to compare two strings efficiently.

The compare() function compares two strings and returns the following values according to the matching cases:

- Returns 0 if both the strings are the same.
- Returns <0 if the first string’s character’s value is smaller than the second string input.
- Returns >0 when the second string is greater in comparison.

Syntax: `str1.compare(str2);`

Example:

```
#include <iostream>
#include<cstring> using namespace
std; int main() {     string str1
= "Coding";     string str2 = "
Ninjas!";     string str3 =
"Coding";     int res1 =
str1.compare(str2);     int res2 =
str2.compare(str1);     int res3 =
str3.compare(str1);     cout <<
res1 << endl;     cout << res2 <<
endl;     cout << res3 << endl;
return 0;
}
```

```
Output:
35
-35
0
```

iii) Relational operator: If strings are compared using relational operators, their characters are compared lexicographically according to the current character traits, which means it starts comparison character by character starting from the first character until the characters in both strings are equal or a NULL character is encountered.

Relational operators return either true or false, true if the corresponding comparison holds, false otherwise.

Following are the Relational Operators that can be used to compare two strings:

- >: Greater than
- < : Less than
- == : Equal to
- != : Not equal to
- >=: Greater than and equal to
- <=: Less than and equal to

Rules:

1. s1 < s2: A string s1 is smaller than s2 string, if either, length of s1 is shorter than s2 or first mismatched character is smaller.
2. s1 > s2: A string s1 is greater than s2 string if either, length of s1 is longer than s2, or the first mismatched character is larger.
3. <= and >= have almost the same implementation with additional features of being equal as well.
4. ==: If, after comparing lexicographically, both strings are found to be the same, then they are said to be equal.

5. !=: If any of the points from 1 to 3 follow up then, strings are said to be unequal.

Example:

```
#include <iostream>
using namespace std;
int main() {
    string str1 = "Coding";
string str2 = " Ninjas!";
string str3 = "Coding";    bool
res1 = str1 == str2;    bool
res2 = str2 < str1;    bool
res3 = str3 != str1;
   cout << res1 << endl;
    cout << res2 << endl;
cout << res3 << endl;
return 0;
}
Output:
0
1
0
```

Previous Next

String Methods

String Methods

i) Using inbuilt size method: The method size returns the length of the string.

Example:

```
#include <iostream>
#include<string> using namespace
std; int main() {     string str
= "Coding Ninjas";     cout <<
str.size() << endl;     return 0;
}
Output:
13
```

ii) Using inbuilt reverse function: The method reverses the string. It is present in the algorithm.h header file.

Example:

```
#include <iostream>
#include<algorithm> using namespace
std; int main() {     string str =
"Coding Ninjas";
reverse(str.begin(), str.end());
cout << str << endl;
    return 0;
} Output: sajniN gnidoC
```

iii) Using inbuilt sort function: The method sorts the string. It is present in the algorithm.h header file.

Example:

```
#include <iostream>
#include<algorithm> using
namespace std; int main() {
string str = "Coding Ninjas";
sort(str.begin(), str.end());
cout << str << endl;     return
0;
}
Output:
CNadgiijnnos
```

Previous Next

Example Problems

Example Problems:

Que: Given a string check whether it is Palindrome or not.

```
#include <iostream> #include
<string.h> using namespace std;
int main() {     string s;
getline(cin, s);     int low =
0;     int high = s.size() - 1;
int flag = 0;     while (low <
high) {         if (s[low] !=
s[high]) {             flag = 1;
break;
        }
```

```
        low++;
high--;
    }       if (flag == 0)
cout << "String is palindrome";
else
        cout << "String is not palindrome";
return 0;
}
Input1:
Hello
Output1:
String is not palindrome
Input2: madam
Output2:
String is palindrome
```

Que: Given a string, find the character which occurred maximum in the string.

```
#include <iostream> #include <string.h>
using namespace std; int main() {
string s;     getline(cin, s);     int max
= 0;     char result;     int count[26] =
{0};     for (int i = 0; i < s.length();
i++) {         count[s[i] - 'a']++;     }
for (int i = 0; i < 26; i++) {         if
(count[i] > max) {             max =
count[i];             result = ('a' + i);
        }     }
cout << result;
return 0;
}
Input1:
hello
Output1:
l Input2:
welcome
Output2:
e
```