

---

# UNIT 2:

## Elements of C++ Language

### What is a token?

**A C++ program is composed of tokens which are the smallest individual unit.** Tokens can be one of several things, including keywords, identifiers, constants, operators, or punctuation marks.

**There are 6 types of tokens in c++:**

1. Keyword
2. Identifiers
3. Constants
4. Strings
5. Special symbols
6. Operators

### 1. Keywords:

**In C++, keywords are reserved words that have a specific meaning in the language and cannot be used as identifiers.** Keywords are an essential part of the C++ language and are used to perform specific tasks or operations.

**There are 95 keywords in c++:**

alignas (since C++11)	alignof (since C++11)	and
and	and_eq	asm

atomic_cancel (TM TS)	atomic_commit (TM TS)	atomic_noexcept (TM TS)
auto(1)	bitand	bitor
bool	break	case
catch	char	char16_t (since C++11)
char32_t (since C++11)	class(1)	compl
concept (since C++20)	const	constexpr (since C++11)
const_cast	continue	co_await (coroutines TS)
co_return (coroutines TS)	co_yield (coroutines TS)	decltype (since C++11)
default(1)	delete(1)	do
double	dynamic_cast	else
enum	explicit	export(1)
extern(1)	false	float
for	friend	goto
if	import (modules TS)	inline(1)
int	long	module (modules TS)
mutable(1)	namespace	new

---

noexcept (since C++11)	not	not_eq
nullptr (since C++11)	operator	or
or_eq	private	protected
public	register(2)	reinterpret_cast
requires (since C++20)	return	short
signed	sizeof(1)	static
static_assert (since C++11)	static_cast	struct(1)
switch	synchronized (TM TS)	template
this	thread_local (since C++11)	throw
true	try	typedef
typeid	typename	union
unsigned	using(1)	virtual
void	volatile	wchar_t
while	xor	xor_eq

**If you try to use a reserved keyword as an identifier, the compiler will produce an error because it will not know what to do with the keyword.**

Here is an example of a program that tries to use a reserved keyword as an identifier:

---

```
1. #include <iostream>
2.
3. using namespace std;
4.
5. int main()
6. {
7.     // Define a variable named "int" of type int and initialize it to 10
8.     int int = 10;
9.
10.    // Print the value of the "int" variable
11.    cout << "The value of the variable " << int << " is " << int << "." << endl;
12.
13.    return 0;
14. }
```

### Output:

```
error: expected unqualified-id before numeric constant
```

### Explanation:

When we compile the above code, the compiler will produce an error. This error occurs because the `int` keyword cannot be used as an identifier. It would help if you chose a different name for your variable to avoid this error.

***In summary, reserved keywords are an essential part of the C++ language and cannot be used as identifiers.*** You must choose a different name for your variables, functions, and other objects in your code.

## 2. Identifiers

In C++, an identifier is a name given to a variable, function, or another object in the code. Identifiers are used to refer to these entities in the program, and they can consist of letters, digits, and underscores. However, some rules must be followed when choosing an identifier:

- The first character must be a letter or an underscore.
- Identifiers cannot be the same as a keyword.
- Identifiers cannot contain any spaces or special characters except for the underscore.
- Identifiers are case-sensitive, meaning that a variable named "myVariable" is different from a variable named "myvariable".

Here are some examples of valid and invalid identifiers in C++:

- **Valid:**
  - my\_variable
  - student\_name
  - balance\_due
- **Invalid:**
  - my variable (contains a space)
  - student# (contains a special character)
  - int (same as a keyword)

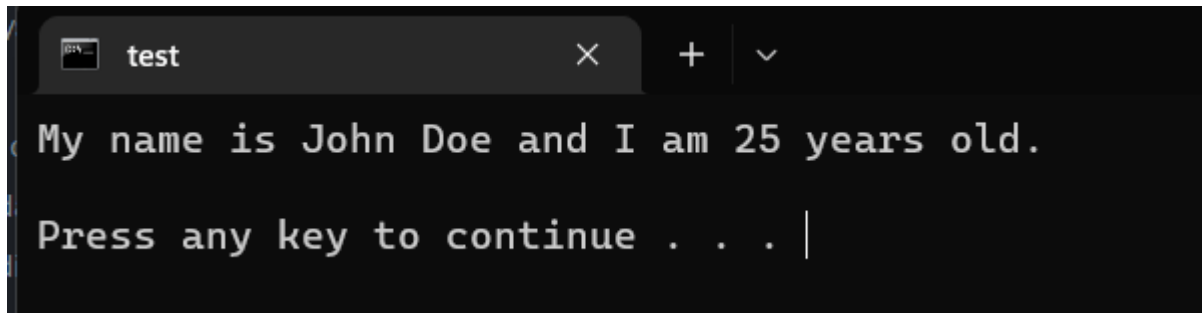
It's important to choose meaningful and descriptive names for your identifiers to make your code easier to read and understand.

### Example:

```
1. #include <iostream>
2.
3. using namespace std;
4.
5. int main()
6. {
7.     // Define a variable named "age" of type int and initialize it to 25
8.     int age = 25;
9.
10.    // Define a variable named "name" of type string and initialize it to "John Doe"
11.    string name = "John Doe";
12.
```

```
13. // Print the value of the "age" and "name" variables
14. cout << "My name is " << name << " and I am " << age << " years old." << endl;
15.
16. return 0;
17. }
```

### Output:



```
test
My name is John Doe and I am 25 years old.
Press any key to continue . . . |
```

### Explanation:

In this program, the variable's age and name are identifiers used to store and access the values of the variables. The identifier age is used to store a person's age, and the identifier name is used to store the person's name. Notice how the identifiers are chosen to be descriptive and meaningful, which makes the code easier to read and understand. You can use any valid identifier in your C++ programs if you follow the rules for naming identifiers discussed earlier.

## 3. Constants

In C++, a constant is a value that cannot be changed during the execution of the program. Constants often represent fixed values used frequently in the code, such as the value of pi or the maximum size of an array.

To define a constant in C++, you use the const keyword followed by the data type and the constant's name and then initialize it with a value. The value of a constant cannot be changed once it has been defined. Here is an example:

```
1. const double PI = 3.14159;
```

In this example, the constant PI is of type double and is initialized with the value of pi. You can then use the constant PI in your code wherever you need to use the value of pi.

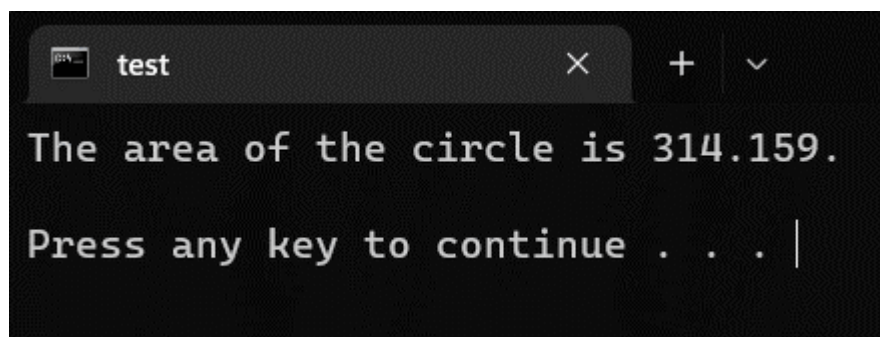
---

Using constants instead of regular variables is important when you need to represent fixed values in your code. This can make your code more readable and maintainable and prevent accidental changes to the value of the constant.

**Example:**

```
1. #include <iostream>
2.
3. using namespace std;
4.
5. int main()
6. {
7.     // Define a constant named "PI" of type double and initialize it to 3.14159
8.     const double PI = 3.14159;
9.
10.    // Define a variable named "radius" of type double and initialize it to 10.0
11.    double radius = 10.0;
12.
13.    // Calculate the area of a circle with radius 10.0
14.    double area = PI * radius * radius;
15.
16.    // Print the area of the circle
17.    cout << "The area of the circle is " << area << "." << endl;
18.
19.    return 0;
20. }
```

**Output:**

A screenshot of a terminal window with a dark background. The window has a title bar with the text 'test' and standard window controls (close, maximize, and a dropdown arrow). The terminal displays the output of the C++ program: 'The area of the circle is 314.159.' followed by a new line and 'Press any key to continue . . . |' with a vertical cursor at the end.

---

## Explanation:

In this program, the constant PI is used to represent the value of pi. The value of PI is defined using the `const` keyword and is initialized with the value of pi. The constant PI is then used in calculating the area of a circle, which is multiplied by the radius of the circle squared to determine the area.

Notice how the constant PI is used the same way as a regular variable, but it cannot be changed once it has been defined. This makes it a perfect choice for representing fixed values used frequently in the code.

You can use constants in your C++ programs to represent fixed values that cannot be changed. This can make your code more readable and maintainable and prevent accidental changes to the value of the constant.

## 4. String:

In C++, a string is a sequence of characters that represents text. ***Strings are commonly used in C++ programs to store and manipulate text data.***

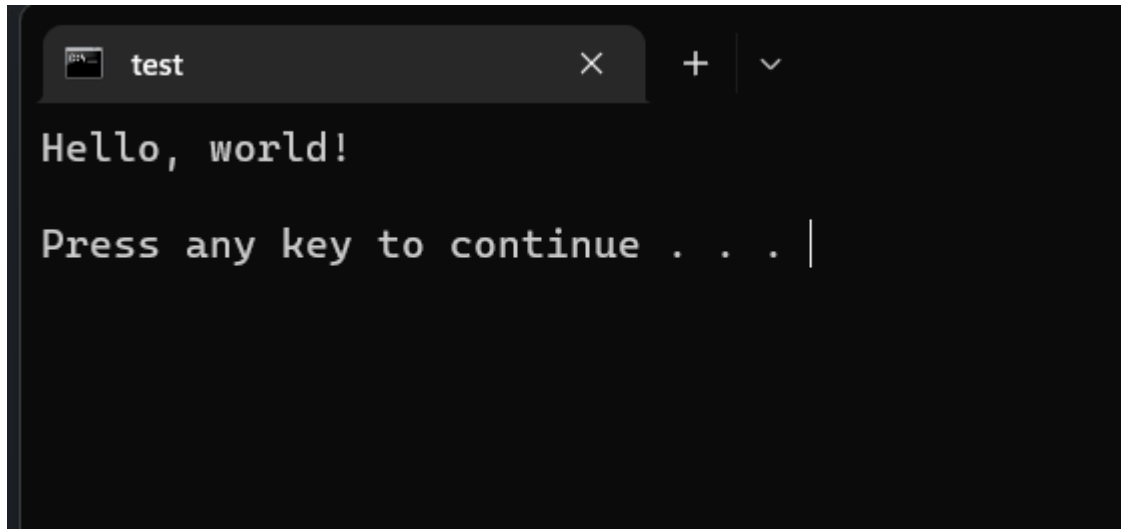
To use strings in C++, you must include the `<string>` header file at the beginning of your program. This will provide access to the `string` class, which is used to create and manipulate strings in C++. Here is an example:

```
1. #include <iostream>
2. #include <string>
3.
4. using namespace std;
5.
6. int main()
7. {
8.     // Define a string variable named "message" and initialize it with a string value
9.     string message = "Hello, world!";
10.
11.    // Print the value of the "message" variable
12.    cout << message << endl;
13.
14.    return 0;
```



15. }

**Output:**

A screenshot of a terminal window with a dark background. The window has a title bar with the text 'test' and standard window controls (close, maximize, and a dropdown arrow). The terminal displays the text 'Hello, world!' on the first line. On the second line, it shows 'Press any key to continue . . . |', where the vertical bar indicates a cursor position.

**Explanation:**

In this example, the `string` class creates a string variable named `message`. The `message` variable is initialized with the string "Hello, world!" and then printed to the screen using the `cout` object.

You can use the `string` class to perform various operations on strings, such as concatenating strings, comparing strings, searching for substrings, and more. For more information, you can consult the documentation for the `string` class.

## 5. Special symbols:

In C++, several special symbols are used for various purposes. The ampersand (&) is used to represent the address of a variable, while the tilde (~) is used to represent the bitwise NOT operator. Some of the most common special symbols include the asterisk (\*), used as the multiplication and pointer operators.

The pound sign (#) is used to represent the preprocessor directive, a special type of instruction processed by the preprocessor before the code is compiled. The percent sign (%) is used as the modulus operator, which is used to find the remainder when one number is divided by another.

---

The vertical bar (|) is used to represent the bitwise OR operator, while the caret (^) is used to represent the bitwise XOR operator. The exclamation point (!) is used to represent the logical NOT operator, which is used to negate a Boolean value.

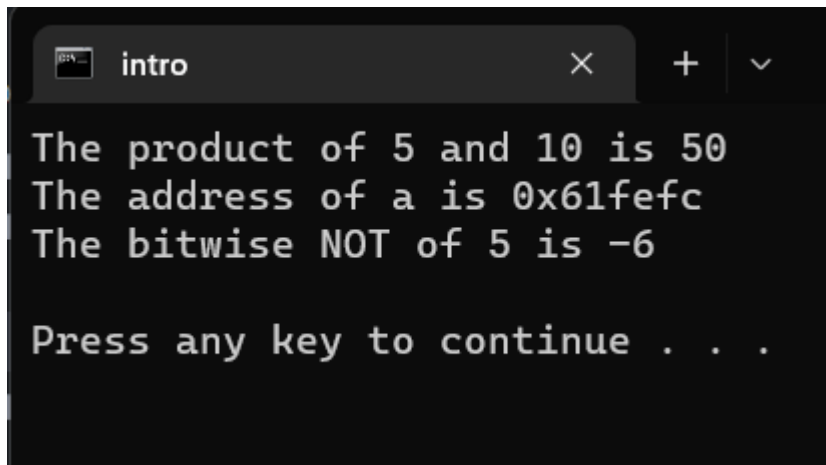
In addition to these symbols, C++ also has a number of other special characters that are used for a variety of purposes. For example, the backslash (/) is used to escape special characters, the double quotes (") are used to enclose string literals, and the single quotes (') are used to enclose character literals.

Overall, the special symbols in C++ are an important part of the language and are used in a variety of different contexts to perform a wide range of operations.

### Example:

```
1. #include <iostream>
2.
3. using namespace std;
4.
5. int main() {
6.     int a = 5;
7.     int b = 10;
8.     int c = a * b; // Use of the asterisk (*) as the multiplication operator
9.     cout << "The product of " << a << " and " << b << " is " << c << endl;
10.
11.    int *ptr = &a; // Use of the ampersand (&) to get the address of a variable
12.    cout << "The address of a is " << ptr << endl;
13.
14.    int d = ~a; // Use of the tilde (~) as the bitwise NOT operator
15.    cout << "The bitwise NOT of " << a << " is " << d << endl;
16.
17.    return 0;
18. }
```

### Output:



```
intro
The product of 5 and 10 is 50
The address of a is 0x61fefc
The bitwise NOT of 5 is -6

Press any key to continue . . .
```

### Explanation:

In this program, the asterisk (\*) is used as the multiplication operator to calculate the product of two variables, a and b. The ampersand (&) is used to get the address of the a variable, which is then printed to the console. Finally, the tilde (~) is used as the bitwise NOT operator to negate the value of a.

## 6. Operators:

In C++, operators are special symbols that are used to perform operations on one or more operands. An operand is a value on which an operator acts. C++ has a wide range of operators, **including arithmetic operators, relational operators, logical operators, and others.**

**Arithmetic operators** are used to perform mathematical operations such as addition, subtraction, multiplication, and division. Some examples of arithmetic operators include + (used for addition), - (used for subtraction), \* (used for multiplication), and / (used for division).

**Relational operators** are used to compare two values and determine their relationship. Some examples of relational operators include == (used to check for equality), != (used to check for inequality), > (used to check if a value is greater than another), and < (used to check if a value is less than another).

**Logical operators** are used to combine two or more relational expressions and determine their logical relationship. Some examples of logical operators include && (used for the logical AND operation), || (used for the logical OR operation), and ! (used for the logical NOT operation).

---

In addition to these operators, C++ also has a number of other operators that are used for different purposes. For example, the bitwise operators are used to perform bitwise operations on individual bits of a value, and the assignment operators are used to assign a value to a variable.

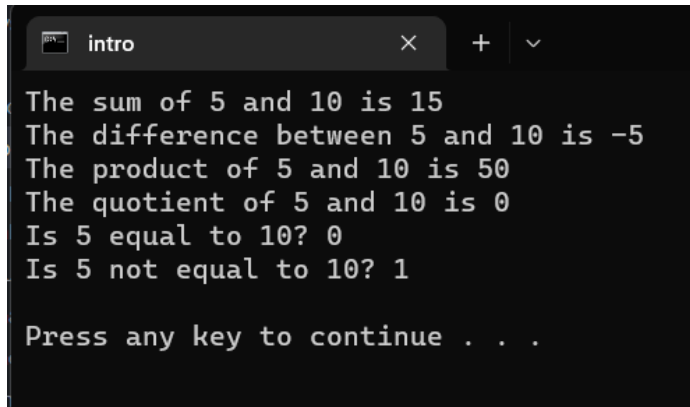
Overall, operators are an essential part of C++ and are used to perform a wide range of operations in a program.

### Example:

```
1. #include <iostream>
2.
3. using namespace std;
4.
5. int main() {
6.     int a = 5;
7.     int b = 10;
8.     int c = a + b; // Use of the plus (+) operator for addition
9.     cout << "The sum of " << a << " and " << b << " is " << c << endl;
10.
11.    c = a - b; // Use of the minus (-) operator for subtraction
12.    cout << "The difference between " << a << " and " << b << " is " << c << endl;
13.
14.    c = a * b; // Use of the asterisk (*) operator for multiplication
15.    cout << "The product of " << a << " and " << b << " is " << c << endl;
16.
17.    c = a / b; // Use of the forward slash (/) operator for division
18.    cout << "The quotient of " << a << " and " << b << " is " << c << endl;
19.
20.    bool d = (a == b); // Use of the double equals (==) operator to check for equality
21.    cout << "Is " << a << " equal to " << b << "? " << d << endl;
22.
23.    d = (a != b); // Use of the exclamation point and equals (!=) operator to check for inequality
24.    cout << "Is " << a << " not equal to " << b << "? " << d << endl;
25.
26.    return 0;
```

27. }

### Output:



```
intro x + v
The sum of 5 and 10 is 15
The difference between 5 and 10 is -5
The product of 5 and 10 is 50
The quotient of 5 and 10 is 0
Is 5 equal to 10? 0
Is 5 not equal to 10? 1

Press any key to continue . . .
```

### Explanation:

In this program, various operators are used to perform different operations. The + operator is used for addition, the - operator is used for subtraction, the \* operator is used for multiplication, and the / operator is used for division. The == and != operators are used to check for equality and inequality, respectively.

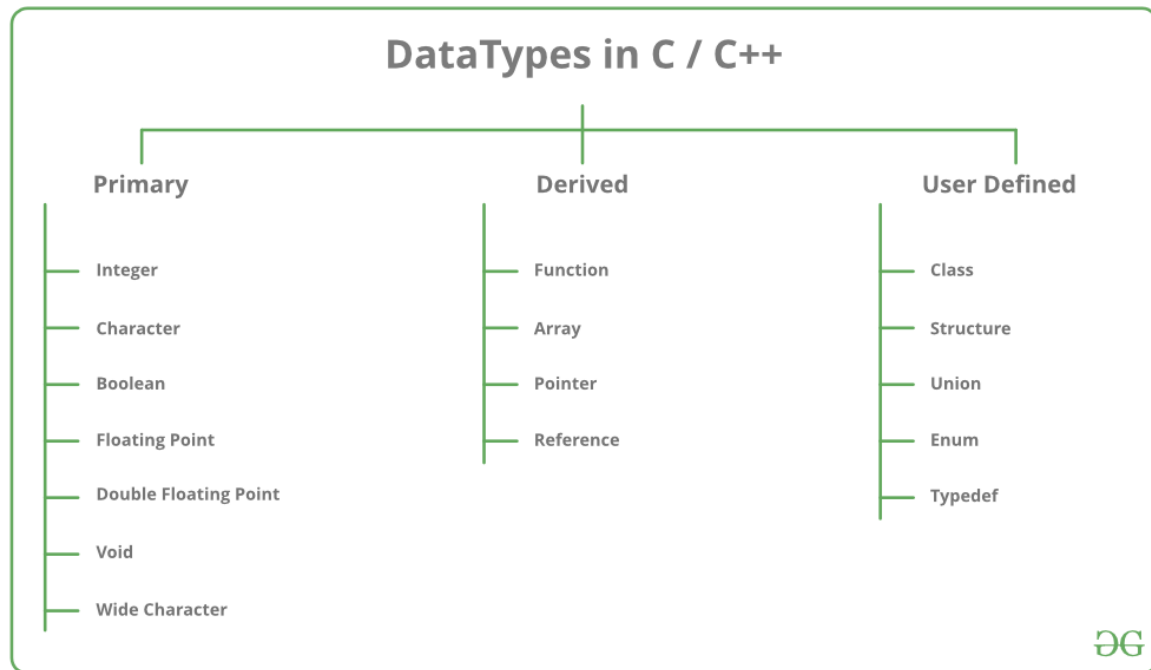
## Data Types

All [variables](#) use data type during declaration to restrict the type of data to be stored. Therefore, we can say that data types are used to tell the variables the type of data they can store. Whenever a variable is defined in C++, the compiler allocates some memory for that variable based on the data type with which it is declared. Every data type requires a different amount of memory.

C++ supports a wide variety of data types and the programmer can select the data type appropriate to the needs of the application. Data types specify the size and types of values to be stored. However, storage representation and machine instructions to manipulate each data type differ from machine to machine, although C++ instructions are identical on all machines.

### C++ supports the following data types:

1. **Primary or Built-in or Fundamental data type**
2. **Derived data types**
3. **User-defined data types**



Data Types in C++ are Mainly Divided into 3 Types:

**1. Primitive Data Types:** These data types are built-in or predefined data types and can be used directly by the user to declare variables. example: int, char, float, bool, etc. Primitive data types available in C++ are:

- Integer
- Character
- Boolean
- Floating Point
- Double Floating Point
- Valueless or Void
- Wide Character

**2. Derived Data Types:** [Derived data types](#) that are derived from the primitive or built-in datatypes are referred to as Derived Data Types. These can be of four types namely:

- Function
- Array
- Pointer
- Reference

**3. Abstract or User-Defined Data Types:** [Abstract or User-Defined data types](#) are defined by the user itself. Like, defining a class in C++ or a structure. C++ provides the following user-defined datatypes:

- Class
- Structure
- Union
- Enumeration

- Typedef defined Datatype

#### Primitive Data Types

- **Integer:** The keyword used for integer data types is **int**. Integers typically require 4 bytes of memory space and range from -2147483648 to 2147483647.
- **Character:** Character data type is used for storing characters. The keyword used for the character data type is **char**. Characters typically require 1 byte of memory space and range from -128 to 127 or 0 to 255.
- **Boolean:** Boolean data type is used for storing Boolean or logical values. A Boolean variable can store either *true* or *false*. The keyword used for the Boolean data type is **bool**.
- **Floating Point:** Floating Point data type is used for storing single-precision floating-point values or decimal values. The keyword used for the floating-point data type is **float**. Float variables typically require 4 bytes of memory space.
- **Double Floating Point:** Double Floating Point data type is used for storing double-precision floating-point values or decimal values. The keyword used for the double floating-point data type is **double**. Double variables typically require 8 bytes of memory space.
- **void:** Void means without any value. void data type represents a valueless entity. A void data type is used for those function which does not return a value.
- **Wide Character:** [Wide character](#) data type is also a character data type but this data type has a size greater than the normal 8-bit data type. Represented by **wchar\_t**. It is generally 2 or 4 bytes long.
- **sizeof() operator:** [sizeof\(\) operator](#) is used to find the number of bytes occupied by a variable/data type in computer memory.

#### Example:

```
int m , x[50];
```

```
cout<<sizeof(m); //returns 4 which is the number of bytes occupied by the integer variable "m".
```

```
cout<<sizeof(x); //returns 200 which is the number of bytes occupied by the integer array variable "x".
```

The size of variables might be different from those shown in the above table, depending on the compiler and the computer you are using.

- C++

---

```
// C++ Program to Demonstrate the correct size

// of various data types on your computer.

#include <iostream>

using namespace std;

int main()

{

    cout << "Size of char : " << sizeof(char) << endl;

    cout << "Size of int : " << sizeof(int) << endl;

    cout << "Size of long : " << sizeof(long) << endl;

    cout << "Size of float : " << sizeof(float) << endl;

    cout << "Size of double : " << sizeof(double) << endl;

    return 0;

}
```

### **Output**

Size of char : 1

Size of int : 4



Size of long : 8

Size of float : 4

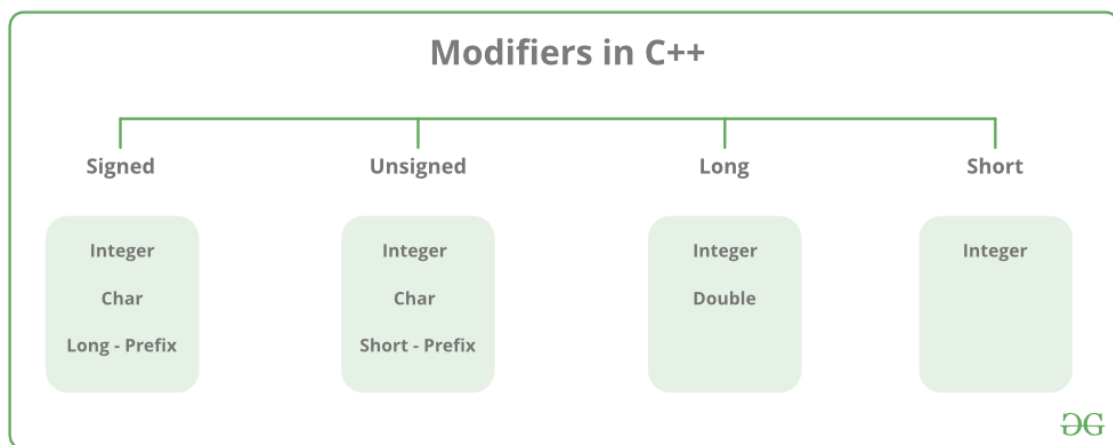
Size of double : 8

**Time Complexity:**  $O(1)$

**Space Complexity:**  $O(1)$

#### Datatype Modifiers

As the name suggests, datatype modifiers are used with built-in data types to modify the length of data that a particular data type can hold.



Data type modifiers available in C++ are:

- **Signed**
- **Unsigned**
- **Short**
- **Long**

The below table summarizes the modified size and range of built-in datatypes when combined with the type modifiers:

Data Type	Size (in bytes)	Range
short int	2	-32,768 to 32,767
unsigned short int	2	0 to 65,535
unsigned int	4	0 to 4,294,967,295

---

Data Type	Size (in bytes)	Range
int	4	-2,147,483,648 to 2,147,483,647
long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295
long long int	8	$-(2^{63})$ to $(2^{63})-1$
unsigned long long int	8	0 to 18,446,744,073,709,551,615
signed char	1	-128 to 127
unsigned char	1	0 to 255
float	4	$-3.4 \times 10^{38}$ to $3.4 \times 10^{38}$
double	8	$-1.7 \times 10^{308}$ to $1.7 \times 10^{308}$
long double	12	$-1.1 \times 10^{4932}$ to $1.1 \times 10^{4932}$
wchar_t	2 or 4	1 wide character

## Operators in C++

An operator is simply a symbol that is used to perform operations. There can be many types of operations like arithmetic, logical, bitwise etc.

There are following types of operators to perform different types of operations in C language.

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operator
- Unary operator
- Ternary or Conditional Operator
- Misc Operator

	Operator	Type
<b>Binary Operator</b>	+, -, *, /, %	Arithmetic Operators
	<, <=, >, >=, ==, !=	Relational Operators
	&&,   , !	Logical Operators
	&,  , <<, >>, ~, ^	Bitwise Operators
	=, +=, -=, *=, /=, %=	Assignment Operators
<b>Unary Operator</b>	→ ++, --	Unary Operator
<b>Ternary Operator</b>	→ ?:	Ternary or Conditional Operator

## Precedence of Operators in C++

The precedence of operator species that which operator will be evaluated first and next. The associativity specifies the operators direction to be evaluated, it may be left to right or right to left.

Let's understand the precedence by the example given below:

1. `int data=5+10*10;`

The "data" variable will contain 105 because \* (multiplicative operator) is evaluated before + (additive operator).

The precedence and associativity of C++ operators is given below:

Category	Operator	Associativity
Postfix	() [] -> . ++ --	Left to right
Unary	+ - ! ~ ++ -- (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Right to left
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Right to left
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Right to left
Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^=  =	Right to left

---

Comma	,	Left to right
-------	---	---------------

## Type Casting in C++

This section will discuss the type casting of the variables in the C++ programming language. Type casting refers to the conversion of one data type to another in a program. Typecasting can be done in two ways: automatically by the compiler and manually by the programmer or user. Type Casting is also known as Type Conversion.

For example, suppose the given data is an integer type, and we want to convert it into float type. So, we need to manually cast int data to the float type, and this type of casting is called the Type Casting in C++.

1. **int** num = 5;
2. **float** x;
3. x = **float**(num);
4. x = 5.0

### 2<sup>nd</sup> example:

1. **float** num = 5.25;
2. **int** x;
3. x = **int**(num);
4. Output: 5

Type Casting is divided into two types: Implicit conversion or Implicit Type Casting and Explicit Type Conversion or Explicit Type Casting.

### Implicit Type Casting or Implicit Type Conversion

- It is known as the automatic type casting.
- It automatically converted from one data type to another without any external intervention such as programmer or user. It means the compiler automatically converts one data type to another.

- All data type is automatically upgraded to the largest type without losing any information.
- It can only apply in a program if both variables are compatible with each other.

1. **char** - sort **int** -> **int** -> unsigned **int** -> **long int** -> **float** -> **double** -> **long double**, etc.

**Note: Implicit Type Casting should be done from low to higher data types. Otherwise, it affects the fundamental data type, which may lose precision or data, and the compiler might flash a warning to this effect.**

### Program to use the implicit type casting in C++

Let's create an example to demonstrate the casting of one variable to another using the implicit type casting in C++.

```
1. #include <iostream>
2. using namespace std;
3. int main ()
4. {
5.     short x = 200;
6.     int y;
7.     y = x;
8.     cout << " Implicit Type Casting " << endl;
9.     cout << " The value of x: " << x << endl;
10.    cout << " The value of y: " << y << endl;
11.
12.    int num = 20;
13.    char ch = 'a';
14.    int res = 20 + 'a';
15.    cout << " Type casting char to int data type ('a' to 20): " << res << endl;
16.
17.    float val = num + 'A';
18.    cout << " Type casting from int data to float type: " << val << endl;
19.    return 0;
20. }
```

### Output:

```
Implicit Type Casting
```

```
The value of x: 200
The value of y: 200
Type casting char to int data type ('a' to 20): 117
Type casting from int data to float type: 85
```

In the above program, we declared a short data type variable x is 200 and an integer variable y. After that, we assign x value to the y, and then the compiler automatically converts short data value x to the y, which returns y is 200.

In the next expressions, we declared an int type variable num is 20, and the character type variable ch is 'a', which is equivalent to an integer value of 97. And then, we add these two variables to perform the implicit conversion, which returns the result of the expression is 117.

Similarly, in the third expression, we add the integer variable num is 20, and the character variable ch is 65, and then assign the result to the float variable val. Thus the result of the expression is automatically converted to the float type by the compiler.

## Explicit Type Casting or Explicit Type Conversion

- It is also known as the manual type casting in a program.
- It is manually cast by the programmer or user to change from one data type to another type in a program. It means a user can easily cast one data to another according to the requirement in a program.
- It does not require checking the compatibility of the variables.
- In this casting, we can upgrade or downgrade the data type of one variable to another in a program.
- It uses the cast () operator to change the type of a variable.

## Syntax of the explicit type casting

1. (type) expression;

**type:** It represents the user-defined data that converts the given expression.

**expression:** It represents the constant value, variable, or an expression whose data type is converted.

For example, we have a floating pointing number is 4.534, and to convert an integer value, the statement as:

1. `int num;`
2. `num = (int) 4.534; // cast into int data type`
3. `cout << num;`

When the above statements are executed, the floating-point value will be cast into an integer data type using the `cast ()` operator. And the float value is assigned to an integer `num` that truncates the decimal portion and displays only 4 as the integer value.

### Program to demonstrate the use of the explicit type casting in C++

Let's create a simple program to cast one type variable into another type using the explicit type casting in the C++ programming language.

1. `#include <iostream>`
2. `using namespace std;`
3. `int main ()`
4. `{`
5. `// declaration of the variables`
6. `int a, b;`
7. `float res;`
8. `a = 21;`
9. `b = 5;`
10. `cout << " Implicit Type Casting: " << endl;`
11. `cout << " Result: " << a / b << endl; // it loses some information`
12.
13. `cout << " \n Explicit Type Casting: " << endl;`
14. `// use cast () operator to convert int data to float`
15. `res = (float) 21 / 5;`
16. `cout << " The value of float variable (res): " << res << endl;`
17.
18. `return 0;`
19. `}`

### Output:

```
Implicit Type Casting:
Result: 4
```



---

```
Explicit Type Casting:  
The value of float variable (res): 4.2
```

In the above program, we take two integer variables, a and b, whose values are 21 and 2. And then, divide a by b (21/2) that returns a 4 int type value.

In the second expression, we declare a float type variable res that stores the results of a and b without losing any data using the cast operator in the explicit type cast method.

### Program to cast double data into int and float type using the cast operator

Let's consider an example to get the area of the rectangle by casting double data into float and int type in C++ programming.

```
1. #include <iostream>  
2. using namespace std;  
3. int main ()  
4. {  
5.     // declaration of the variables  
6.     double l, b;  
7.     int area;  
8.  
9.     // convert double data type to int type  
10.    cout << " The length of the rectangle is: " << endl;  
11.    cin >> l;  
12.    cout << " The breadth of the rectangle is: " << endl;  
13.    cin >> b;  
14.    area = (int) l * b; // cast into int type  
15.    cout << " The area of the rectangle is: " << area << endl;  
16.  
17.    float res;  
18.    // convert double data type to float type  
19.    cout << " \n \n The length of the rectangle is: " << l << endl;  
20.    cout << " The breadth of the rectangle is: " << b << endl;  
21.    res = (float) l * b; // cast into float type  
22.    cout << " The area of the rectangle is: " << res;  
23.    return 0;
```

---

24. }

### Output:

```
The length of the rectangle is:
57.3456
The breadth of the rectangle is:
12.9874
The area of the rectangle is: 740

The length of the rectangle is: 57.3456
The breadth of the rectangle is: 12.9874
The area of the rectangle is: 744.77
```

### Some different types of the Type Casting

In type cast, there is a cast operator that forces one data type to be converted into another data type according to the program's needs. C++ has four different types of the cast operator:

1. `Static_cast`
2. `dynamic_cast`
3. `const_cast`
4. `reinterpret_cast`

### Static Cast:

The `static_cast` is a simple compile-time cast that converts or cast one data type to another. It means it does not check the data type at runtime whether the cast performed is valid or not. Thus the programmer or user has the responsibility to ensure that the conversion was safe and valid.

The `static_cast` is capable enough that can perform all the conversions carried out by the implicit cast. And it also performs the conversions between pointers of classes related to each other (upcast - > from derived to base class or downcast - > from base to derived class).

### Syntax of the Static Cast

1. `static_cast < new_data_type> (expression);`

### Program to demonstrate the use of the Static Cast

---

Let's create a simple example to use the static cast of the type casting in C++ programming.

```
1. #include <iostream>
2. using namespace std;
3. int main ()
4. {
5.     // declare a variable
6.     double l;
7.     l = 2.5 * 3.5 * 4.5;
8.     int tot;
9.
10.    cout << " Before using the static cast:" << endl;
11.    cout << " The value of l = " << l << endl;
12.
13.    // use the static_cast to convert the data type
14.    tot = static_cast < int > (l);
15.    cout << " After using the static cast: " << endl;
16.    cout << " The value of tot = " << tot << endl;
17.
18.    return 0;
19. }
```

### Output:

```
Before using the static cast:
The value of l = 39.375
After using the static cast:
The value of tot = 39
```

### Dynamic Cast

The `dynamic_cast` is a runtime cast operator used to perform conversion of one type variable to another only on class pointers and references. It means it checks the valid casting of the variables at the run time, and if the casting fails, it returns a NULL value. Dynamic casting is based on RTTI (Runtime Type Identification) mechanism.

### Program to demonstrate the use of the Dynamic Cast in C++

---

Let's create a simple program to perform the `dynamic_cast` in the C++ programming language.

```
1. #include <iostream>
2. using namespace std;
3.
4. class parent
5. {
6.     public: virtual void print()
7.     {
8.
9.     }
10. };
11. class derived: public parent
12. {
13.
14. };
15.
16. int main ()
17. {
18.     // create an object ptr
19.     parent *ptr = new derived;
20.     // use the dynamic cast to convert class data
21.     derived* d = dynamic_cast <derived*> (ptr);
22.
23.     // check whether the dynamic cast is performed or not
24.     if ( d != NULL)
25.     {
26.         cout << " Dynamic casting is done successfully";
27.     }
28.     else
29.     {
30.         cout << " Dynamic casting is not done successfully";
31.     }
32. }
```

## Output:

```
Dynamic casting is done successfully.
```

## Reinterpret Cast Type

The `reinterpret_cast` type casting is used to cast a pointer to any other type of pointer whether the given pointer belongs to each other or not. It means it does not check whether the pointer or the data pointed to by the pointer is the same or not. And it also cast a pointer to an integer type or vice versa.

## Syntax of the `reinterpret_cast` type

1. `reinterpret_cast` <type> expression;

## Program to use the Reinterpret Cast in C++

Let's write a program to demonstrate the conversion of a pointer using the `reinterpret` in C++ language.

1. `#include <iostream>`
2. `using namespace std;`
- 3.
4. `int main ()`
5. `{`
6. `// declaration of the pointer variables`
7. `int *pt = new int (65);`
- 8.
9. `// use reinterpret_cast operator to type cast the pointer variables`
10. `char *ch = reinterpret_cast <char *> (pt);`
- 11.
12. `cout << " The value of pt: " << pt << endl;`
13. `cout << " The value of ch: " << ch << endl;`
- 14.
15. `// get value of the defined variable using pointer`
16. `cout << " The value of *ptr: " << *pt << endl;`
17. `cout << " The value of *ch: " << *ch << endl;`
18. `return 0;`

- 19.
20. }

### Output:

```
The value of pt: 0x5cfed0
The value of ch: A
The value of *ptr: 65
The value of *ch: A
```

### Const Cast

The `const_cast` is used to change or manipulate the const behavior of the source pointer. It means we can perform the const in two ways: setting a const pointer to a non-const pointer or deleting or removing the const from a const pointer.

### Syntax of the Const Cast type

1. `const_cast` <type> exp;

### Program to use the Const Cast in C++

Let's write a program to cast a source pointer to a non-cast pointer using the `const_cast` in C++.

1. `#include <iostream>`
2. `using namespace std;`
- 3.
4. `// define a function`
5. `int disp(int *pt)`
6. `{`
7.  `return (*pt * 10);`
8. `}`
- 9.
10. `int main ()`
11. `{`
12.  `// declare a const variable`
13.  `const int num = 50;`
14.  `const int *pt = # // get the address of num`

```
15.  
16. // use const_cast to change the constness of the source pointer  
17. int *ptr = const_cast<int*>(pt);  
18. cout << " The value of ptr cast: " << disp(ptr);  
19. return 0;  
20.  
21. }
```

### Output:

```
The value of ptr cast: 500
```

## Console I/O operations in C++

Every program takes some data as input and generates processed data as an output following the familiar input process output cycle. It is essential to know how to provide the input data and present the results in the desired form. The [use of the cin and cout](#) is already known with the [operator >> and <<](#) for the [input and output operations](#). In this article, we will discuss how to control the way the output is printed.

[C++](#) supports a rich set of [I/O functions](#) and operations. These functions use the advanced [features of C++](#) such as [classes](#), [derived classes](#), and [virtual functions](#). It also supports all of C's set of I/O functions and therefore can be used in C++ programs, but their use should be restrained due to two reasons.

1. I/O methods in C++ support the [concept of OOPs](#).
2. I/O methods in [C](#) cannot handle the user-defined data type such as [class and object](#).

It uses the concept of [stream and stream classes](#) to implement its I/O operations with the console and disk files.

### C++ Stream

The [I/O system in C++](#) is designed to work with a wide variety of devices including terminals, disks, and tape drives. Although each device is very different, the I/O system supplies an interface to the programmer that is independent of the actual device being accessed. This interface is known as the [stream](#).

- A stream is a sequence of bytes.
- The source stream that provides the data to the program is called the input stream.
- The destination stream that receives output from the program is called the output stream.

- The data in the input stream can come from the keyboard or any other input device.
- The data in the output stream can go to the screen or any other output device.

C++ contains several pre-defined streams that are automatically opened when a program begins its execution. These include **cin** and **cout**. It is known that **cin** represents the input stream connected to the standard input device (usually the keyboard) and **cout** represents the output stream connected to the standard output device (usually the screen).

### C++ Stream Classes

The [C++ I/O system](#) contains a hierarchy of classes that are used to define various streams to deal with both the console and disk files. These classes are called stream classes. The hierarchy of stream classes used for input and output operations is with the console unit. These classes are declared in the **header file iostream**. This file should be included in all the programs that communicate with the console unit.

- C++

```
#include <iostream>

using namespace std;

int main()

{

    cout << " This is my first"

        " Blog on the gfg";

    return 0;

}
```

**Output:**



---

This is my first Blog on the gfg

The **ios** are the base class for **istream** (input stream) and **ostream** (output stream) which are in turn base classes for **iostream** (input/output stream). The class **ios** are declared as the [virtual base class](#) so that only one [copy of its members](#) is inherited by the iostream.

The class **ios** provide the basics support for [formatted and unformatted I/O operations](#). The class **istream** provides the facilities formatted and unformatted input while the class **ostream** (through inheritance) provides the facilities for formatted output.

The class **iostream** provides the facilities for handling both input and output streams. Three classes add an assignment to these classes:

- **istream\_withassign**
- **ostream\_withassign**
- **iostream\_withassign**

---

### **Tabular Representation of stream classes for Console Operation:**

Class name	Content
<i>ios</i> (General input/output stream class)	<ul style="list-style-type: none"><li>• Contains basic facilities that are used by all other input and output classes</li><li>• Also contains a pointer to a buffer object (<b>streambuf</b> object)</li><li>• Declares Constants and functions that are necessary for handling formatted input and output operations</li></ul>
<i>istream</i> (Input stream)	<ul style="list-style-type: none"><li>• It inherits the properties of <b>ios</b></li><li>• It declares input functions such as <a href="#"><u>get()</u></a>, <a href="#"><u>getline()</u></a>, and <a href="#"><u>read()</u></a></li><li>• It contains overload extraction operator &gt;&gt;</li></ul>
<i>ostream</i> (Output Stream)	<ul style="list-style-type: none"><li>• It inherits the properties of <b>ios</b>.</li><li>• It declares input functions such as <a href="#"><u>put()</u></a> and <a href="#"><u>write()</u></a>.</li><li>• It contains an <a href="#"><u>overload extraction operator</u></a> &lt;&lt;.</li></ul>
<i>iostream</i> (Input/output stream)	<ul style="list-style-type: none"><li>• Inherits the properties of <b>ios stream</b> and <b>ostream</b> through multiple inheritances and thus contains all the input and output functions.</li></ul>
<i>Streambuf</i>	<ul style="list-style-type: none"><li>• It provides an interface to physical devices through buffers.</li><li>• It acts as a base for filebuf class used ios file</li></ul>

## Control statements

### Introduction:

C++ is a programming language from a high level that is widely used for creating applications and software. One of the most important concepts in C++ programming is **Flow Control**, which refers to the ability to direct the flow of a program based on specific conditions. This allows developers to control how their programs execute and can help to make them more efficient and effective. In this article, we will see the different

---

types of flow control available in C++, how they work, and when they are most appropriate.

## Conditional Statements:

Conditional Statements are used in C++ to run a certain piece of program only if a specific condition is met. There are generally three types of conditional statements in C++: **if**, **if-else**, and **switch**.

### if Statement:

The **if** statement is the simplest of the three and is used to run a certain piece of code only if a certain condition is true. For example:

#### C++ Code:

```
1. int x = 5;
2. if (x == 5) {
3.     std::cout << "x is 5" << std::endl;
4. }
```

In this example, the block of code inside the curly braces will only be executed if the condition inside the parentheses is true.

### if-else Statement:

The **if-else** statement is used when we want to execute some code only if some condition exists. If the given condition is true then code will be executed otherwise else statement will be used to run other part of the code. For example:

#### C++ Code:

```
1. int x = 5;
2. if (x == 5) {
3.     std::cout << "x is 5" << std::endl;
4. } else {
5.     std::cout << "x is not 5" << std::endl;
6. }
```

---

In this example, if the condition inside the parentheses is true, the first block of code will be executed. Otherwise, the second block of code will be executed.

## switch Statement:

The **switch** statement is used to execute different blocks of code based on the value of a variable. For example:

### Pseudo Code:

```
1. switch (variable) {
2.     case value1:
3.         // code to execute if the variable is equal to value1
4.         break;
5.     case value2:
6.         // code to execute if the variable is equal to value2
7.         break;
8.     // add more cases as needed
9.     default:
10.        // code to execute if the variable does not match any case
11. }
```

### C++ code:

```
1. int x = 2;
2. switch (x) {
3.     case 1:
4.         std::cout << "x is 1" << std::endl;
5.         break;
6.     case 2:
7.         std::cout << "x is 2" << std::endl;
8.         break;
9.     default:
10.        std::cout << "x is not 1 or 2" << std::endl;
11.        break;
12. }
```

---

In this example, the **switch** statement will execute the block of code associated with the value of x. If x is 1, the first block of code will be executed. If x is 2, the second block of code will be executed. If x is any other value, the default block of code will be executed.

## Loops:

Loops are used in C++ to execute a block of code multiple times, either until a certain condition is met or for a specific number of times. There are generally three types of loops in C++: **while**, **do-while**, and **for**.

### While Loop:

The **while** loop is used to execute when we want to run some code for until some specific condition matches. For example:

#### C++ Code:

1. **int** x = 0;
2. **while** (x < 5) {
3.     std::cout << x << std::endl;
4.     x++;
5. }

In this example, the **while** loop will continue to execute the block of code inside the curly braces as long as x is less than 5. Each time the loop executes, the value of x will be incremented by 1.

### do-while Loop:

The **do-while** loop is the same as the **while** loop, but the condition is checked after the first iteration of the loop. For example:

#### C++ Code:

1. **int** x = 0;
2. **do** {
3.     std::cout << x << std::endl;
4.     x++;
5. } **while** (x < 5);

---

In this example, the **do-while** loop will execute the block of code inside the curly brackets, and then it will check the condition. So it will be executed a minimum of one time.

## for Loop:

The **for** loop allows a program to execute a piece of program a fixed number of times. The syntax for the for loop is:

### Pseudo Code:

1. **for** (initialization; condition; increment/decrement) {
2.    // code to execute repeatedly
3. }

Here is an example that uses the for loop to print the numbers from 1 to 10:

### C++ Code:

1. **for** (**int** i = 1; i <= 10; i++) {
2.    cout << i << " ";
3. }

## Conclusion:

In conclusion, flow control structures are essential to any programming language. C++ provides a range of Flow Control structures that allow programmers to control the flow of their code. The **if-else** statement, **switch** statement, **for** loops, **while** loops and **do-while** loops are used for the flow control.

## Jump Statements:

- **break**
- **continue**
- **goto**
- **return**

These statements are used in C or C++ for the unconditional flow of control throughout the functions in a program. They support four types of jump statements:

### A) **break**

This loop control statement is used to terminate the loop. As soon as the [break](#) statement is encountered from within a loop, the loop iterations stop

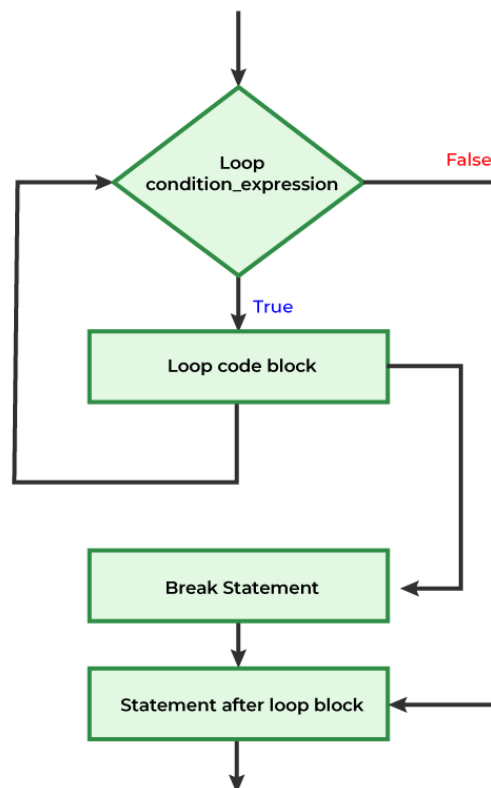
there, and control returns from the loop immediately to the first statement after the loop.

*Syntax of break*

**break;**

Basically, break statements are used in situations when we are not sure about the actual number of iterations for the loop or we want to terminate the loop based on some condition.

### Break Statement Flow Diagram



### Example of break

- C
- C++

```
// C program to illustrate
```

---

```
// to show usage of break

// statement

#include <stdio.h>


void findElement(int arr[], int size, int key)

{

    // loop to traverse array and search for key

    for (int i = 0; i < size; i++) {

        if (arr[i] == key) {

            printf("Element found at position: %d",

                (i + 1));

            break;

        }

    }

}


int main()

{

    int arr[] = { 1, 2, 3, 4, 5, 6 };
```



```
// no of elements

int n = 6;

// key to be searched

int key = 3;

// Calling function to find the key

findElement(arr, n, key);

return 0;

}
```

## Output

Element found at position: 3

## B) continue

This loop control statement is just like the break statement. The [continue statement](#) is opposite to that of the break *statement*, instead of terminating the loop, it forces to execute the next iteration of the loop. As the name suggests the continue statement forces the loop to continue or execute the next iteration. When the continue statement is executed in the loop, the code inside the loop following the continue statement will be skipped and the next iteration of the loop will begin.

*Syntax of continue*

**continue;**

*Flowchart of Continue*

*Example of continue*

- C
- C++

```
// C program to explain the use
// of continue statement

#include <stdio.h>

int main()
{
    // loop from 1 to 10

    for (int i = 1; i <= 10; i++) {

        // If i is equals to 6,

        // continue to next iteration

        // without printing

        if (i == 6)

            continue;

        else
```

```
        // otherwise print the value of i

        printf("%d ", i);

    }

    return 0;

}
```

### Output

1 2 3 4 5 7 8 9 10

If you create a variable in if-else in C/C++, it will be local to that if/else block only. You can use global variables inside the if/else block. If the name of the variable you created in if/else is as same as any global variable then priority will be given to the `local variable`.

- C
- C++

```
#include <stdio.h>

int main()

{

    int gfg = 0; // local variable for main

    printf("Before if-else block %d\n", gfg);
```

```

if (1) {

    int gfg = 100; // new local variable of if block

    printf("if block %d\n", gfg);

}

printf("After if block %d", gfg);

return 0;

}

```

## Output

Before if-else block 0

if block 100

After if block 0

## C) goto

The [goto statement](#) in C/C++ also referred to as the unconditional jump statement can be used to jump from one point to another within a function.

*Syntax of goto*

Syntax1		Syntax2
-----		
<b>goto</b> <i>label</i> ;		<i>label</i> :
.		.
.		.
.		.
<i>label</i> :		<b>goto</b> <i>label</i> ;

In the above syntax, the first line tells the compiler to go to or jump to the statement marked as a label. Here, a label is a user-defined identifier that indicates the target statement. The statement immediately followed after 'label:' is the destination statement. The 'label:' can also appear before the 'goto label;' statement in the above syntax.

*Flowchart of goto Statement*

---

### *Examples of goto*

- C
- C++

```
// C program to print numbers

// from 1 to 10 using goto

// statement

#include <stdio.h>

// function to print numbers from 1 to 10

void printNumbers()

{

    int n = 1;

label:

    printf("%d ", n);

    n++;

    if (n <= 10)

        goto label;

}

// Driver program to test above function
```

```
int main()

{

    printNumbers();

    return 0;

}
```

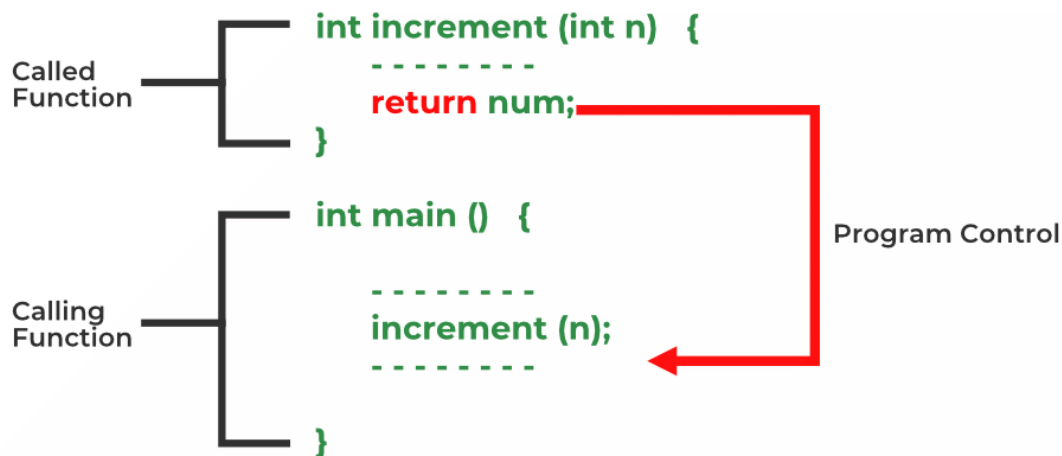
## Output

1 2 3 4 5 6 7 8 9 10

## D) return

The [return](#) in C or C++ returns the flow of the execution to the function from where it is called. This statement does not mandatorily need any conditional statements. As soon as the statement is executed, the flow of the program stops immediately and returns the control from where it was called. The return statement may or may not return anything for a void function, but for a non-void function, a return value must be returned.

*Flowchart of return*



*Flow Diagram of return*

*Syntax of return*

**return** [expression];

---

### *Example of return*

- C
- C++

```
// C code to illustrate return

// statement

#include <stdio.h>


// non-void return type

// function to calculate sum

int SUM(int a, int b)

{

    int s1 = a + b;

    return s1;

}


// returns void

// function to print

void Print(int s2)

{

    printf("The sum is %d", s2);
```

```
        return;

    }

    int main()

    {

        int num1 = 10;

        int num2 = 10;

        int sum_of = SUM(num1, num2);

        Print(sum_of);

        return 0;

    }
```

### Output

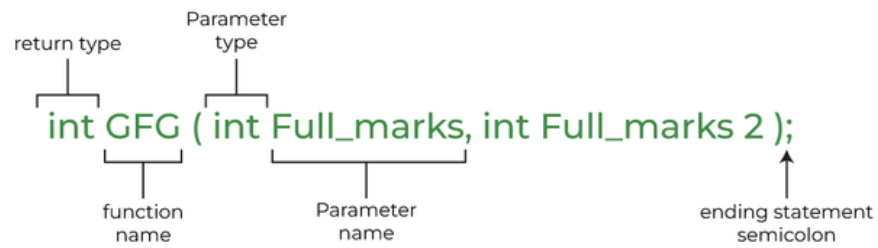
The sum is 20

## Functions in C++

A function is a set of statements that take inputs, do some specific computation, and produce output. The idea is to put some commonly or repeatedly done tasks together and make a **function** so that instead of writing the same code again and again for different inputs, we can call the function. In simple terms, a function is a block of code that only runs when it is called.

### Syntax:





*Syntax of Function*

### Example:

- C++

```
// C++ Program to demonstrate working of a function

#include <iostream>

using namespace std;

// Following function that takes two parameters 'x' and 'y'
// as input and returns max of two input numbers

int max(int x, int y)
{
    if (x > y)
        return x;
    else
        return y;
}
```

```
// main function that doesn't receive any parameter and

// returns integer

int main()

{

    int a = 10, b = 20;


    // Calling above function to find max of 'a' and 'b'

    int m = max(a, b);


    cout << "m is " << m;

    return 0;

}
```

## Output

m is 20

**Time complexity:**  $O(1)$

**Space complexity:**  $O(1)$

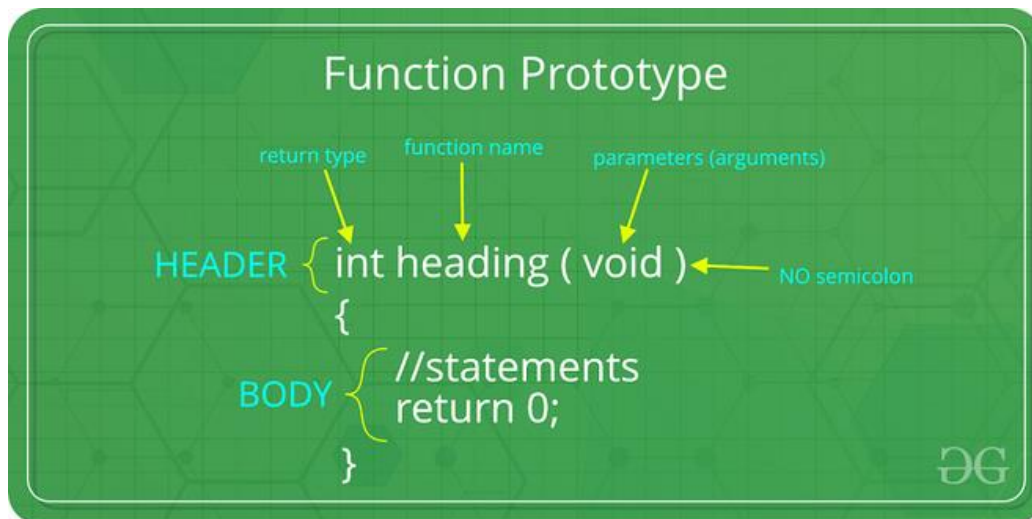
Why Do We Need Functions?

- Functions help us in **reducing code redundancy**. If functionality is performed at multiple places in software, then rather than writing the same code, again and again, we create a function and call it everywhere. This also helps in maintenance as we have to change at one place if we make future changes to the functionality.
- Functions make code **modular**. Consider a big file having many lines of code. It becomes really simple to read and use the code if the code is divided into functions.

- Functions provide **abstraction**. For example, we can use library functions without worrying about their internal work.

#### Function Declaration

A function declaration tells the compiler about the number of parameters function takes data-types of parameters, and returns the type of function. Putting parameter names in the function declaration is optional in the function declaration, but it is necessary to put them in the definition. Below are an example of function declarations. (parameter names are not there in the below declarations)



*Function Declaration*

#### Example:

- C++

```
// C++ Program to show function that takes  
  
// two integers as parameters and returns  
  
// an integer  
  
int max(int, int);
```

---

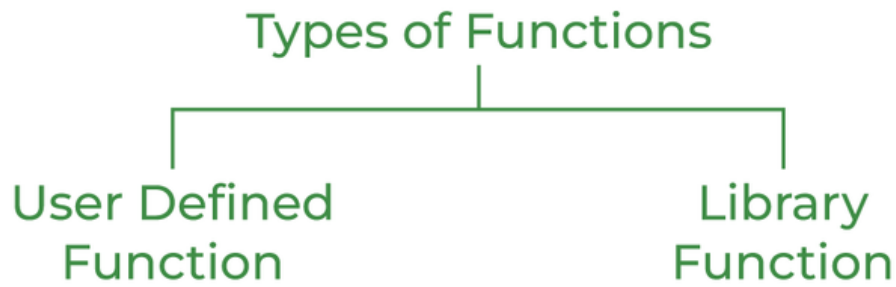
```
// A function that takes an int  
  
// pointer and an int variable  
  
// as parameters and returns  
  
// a pointer of type int  
  
int* swap(int*, int);
```

```
// A function that takes  
  
// a char as parameter and  
  
// returns a reference variable  
  
char* call(char b);
```

```
// A function that takes a  
  
// char and an int as parameters  
  
// and returns an integer  
  
int fun(char, int);
```

---

## Types of Functions



*Types of Function in C++*

### User Defined Function

User Defined functions are user/customer-defined blocks of code specially customized to reduce the complexity of big programs. They are also commonly known as “**tailor-made functions**” which are built only to satisfy the condition in which the user is facing issues meanwhile reducing the complexity of the whole program.

### Library Function

Library functions are also called “**builtin Functions**“. These functions are a part of a compiler package that is already defined and consists of a special function with special and different meanings. Builtin Function gives us an edge as we can directly use them without defining them whereas in the user-defined function we have to declare and define a function before using them.

**For Example:** `sqrt()`, `setw()`, `strcat()`, etc.

#### Parameter Passing to Functions

The parameters passed to function are called **actual parameters**. For example, in the program below, 5 and 10 are actual parameters. The parameters received by the function are called **formal parameters**. For example, in the above program x and y are formal parameters.

```

class Multiplication {
    int multiply(int x, int y) { return x * y; }
public
    static void main()
    {
        Multiplication M = new Multiplication();
        int gfg = 5, gfg2 = 10;
        int gfg3 = multiply(gfg, gfg2);
        cout << "Result is " << gfg3;
    }
}

```

Formal Parameter

Actual Parameter

*Formal Parameter and Actual Parameter*

**There are two most popular ways to pass parameters:**

1. **Pass by Value:** In this parameter passing method, values of actual parameters are copied to the function's formal parameters and the two types of parameters are stored in different memory locations. So any changes made inside functions are not reflected in the actual parameters of the caller.
2. **Pass by Reference:** Both actual and formal parameters refer to the same locations, so any changes made inside the function are actually reflected in the actual parameters of the caller.

Function Definition

**Pass by value** is used where the value of x is not modified using the function fun().

- C++

```

// C++ Program to demonstrate function definition

#include <iostream>

using namespace std;

void fun(int x)

```

```

{

    // definition of

    // function

    x = 30;

}


int main()

{

    int x = 20;

    fun(x);

    cout << "x = " << x;

    return 0;

}

```

## Output

x = 20

**Time complexity:**  $O(1)$

**Space complexity:**  $O(1)$

## Functions Using Pointers

The function `fun()` expects a pointer `ptr` to an integer (or an address of an integer). It modifies the value at the address `ptr`. The dereference operator `*` is used to access the value at an address. In the statement `*ptr = 30`, the value at address `ptr` is changed to 30. The address operator `&` is used to get the address of a variable of any data type. In the function call statement `fun(&x)`, the address of `x` is passed so that `x` can be modified using its address.

- C++

---

```
// C++ Program to demonstrate working of  
  
// function using pointers  
  
#include <iostream>  
  
using namespace std;  
  
void fun(int* ptr) { *ptr = 30; }  
  
int main()  
{  
  
    int x = 20;  
  
    fun(&x);  
  
    cout << "x = " << x;  
  
    return 0;  
}
```

### Output

x = 30

**Time complexity:** O(1)

**Space complexity:** O(1)



### Difference between call by value and call by reference in C++

Call by value	Call by reference
A copy of value is passed to the function	An address of value is passed to the function
Changes made inside the function is not reflected on other functions	Changes made inside the function is reflected outside the function also
Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in same memory location.

#### Points to Remember About Functions in C++

1. Most C++ program has a function called main() that is called by the operating system when a user runs the program.
2. Every function has a return type. If a function doesn't return any value, then void is used as a return type. Moreover, if the return type of the function is void, we still can use the return statement in the body of the function definition by not specifying any constant, variable, etc. with it, by only mentioning the 'return;' statement which would symbolize the termination of the function as shown below:

- C++

```
void function name(int a)

{

    ..... // Function Body

    return; // Function execution would get terminated

}
```

---

3. To declare a function that can only be called without any parameter, we should use “**void fun(void)**”. As a side note, in C++, an empty list means a function can only be called without any parameter. In C++, both `void fun()` and `void fun(void)` are same.

### Main Function

The main function is a special function. Every C++ program must contain a function named `main`. It serves as the entry point for the program. The computer will start running the code from the beginning of the main function.

## Types of Main Functions

### 1. Without parameters:

- CPP

```
// Without Parameters

int main() { ... return 0; }
```

### 2. With parameters:

- CPP

```
// With Parameters

int main(int argc, char* const argv[]) { ... return 0; }
```

The reason for having the parameter option for the main function is to allow input from the command line. When you use the main function with parameters, it saves every group of characters (separated by a space) after the program name as elements in an array named **argv**. Since the main function has the return type of **int**, the programmer must always have a return statement in the code. The number that is returned is used to inform the calling program what the result of the program’s execution was. Returning 0 signals that there were no problems.

### C++ Recursion

When function is called within the same function, it is known as recursion in C++. The function which calls the same function, is known as recursive function. A function that calls itself, and doesn’t perform any task after function call, is

---

known as tail recursion. In tail recursion, we generally call the same function with return statement.

**Syntax:**

- C++

```
recursionfunction()

{

    recursionfunction(); // calling self function

}
```

To know more see [this article](#).

#### C++ Passing Array to Function

In C++, to reuse the array logic, we can create function. To pass array to function in C++, we need to provide only array name.

```
functionname(arrayname); //passing array to function
```

**Example: Print minimum number**

- C++

```
#include <iostream>

using namespace std;

void printMin(int arr[5]);

int main()

{

    int ar[5] = { 30, 10, 20, 40, 50 };

    printMin(ar); // passing array to function
```

```
}

void printMin(int arr[5])

{

    int min = arr[0];

    for (int i = 0; i < 5; i++) {

        if (min > arr[i]) {

            min = arr[i];

        }

    }

    cout << "Minimum element is: " << min << "\n";

}

// Code submitted by Susobhan Akhuli
```

## Output

Minimum element is: 10

Time complexity:  $O(n)$  where  $n$  is the size of array

Space complexity:  $O(1)$  where  $n$  is the size of array.

## C++ Overloading (Function)

If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:

- *methods,*
- *constructors and*
- *indexed properties*

It is because these members have parameters only.

---

## Types of overloading in C++ are:

- *Function overloading*
- *Operator overloading*

## C++ Function Overloading

Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++. In function overloading, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences compiler can differentiate between the functions. The advantage of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.

**Example: changing number of arguments of add() method**

- C++

```
// program of function overloading when number of arguments
// vary

#include <iostream>

using namespace std;

class Cal {

public:

    static int add(int a, int b) { return a + b; }

    static int add(int a, int b, int c)

    {

        return a + b + c;

    }

}
```

```
};

int main(void)

{

    Cal C; // class object declaration.

    cout << C.add(10, 20) << endl;

    cout << C.add(12, 20, 23);

    return 0;

}

// Code Submitted By Susobhan Akhuli
```

### Output

30

55

**Time complexity:** O(1)

**Space complexity:** O(1)

Example: when the type of the arguments vary.

- C++

```
// Program of function overloading with different types of

// arguments.

#include <iostream>

using namespace std;
```

```
int mul(int, int);

float mul(float, int);

int mul(int a, int b) { return a * b; }

float mul(double x, int y) { return x * y; }

int main()

{

    int r1 = mul(6, 7);

    float r2 = mul(0.2, 3);

    cout << "r1 is : " << r1 << endl;

    cout << "r2 is : " << r2 << endl;

    return 0;

}

// Code Submitted By Susobhan Akhuli
```

### Output

r1 is : 42

r2 is : 0.6

**Time Complexity:**  $O(1)$

**Space Complexity:**  $O(1)$

### Function Overloading and Ambiguity

When the compiler is unable to decide which function is to be invoked among the overloaded function, this situation is known as *function overloading*.

---

When the compiler shows the ambiguity error, the compiler does not run the program.

**Causes of Function Overloading:**

- *Type Conversion.*
- *Function with default arguments.*
- *Function with pass by reference.*

**Type Conversion:-**

- **C++**

```
#include <iostream>

using namespace std;

void fun(int);

void fun(float);

void fun(int i) { cout << "Value of i is : " << i << endl; }

void fun(float j)

{

    cout << "Value of j is : " << j << endl;

}

int main()

{

    fun(12);

    fun(1.2);

    return 0;

}
```



```
// Code Submitted By Susobhan Akhuli
```

The above example shows an error “*call of overloaded ‘fun(double)’ is ambiguous*”. The `fun(10)` will call the first function. The `fun(1.2)` calls the second function according to our prediction. But, this does not refer to any function as in C++, all the floating point constants are treated as double not as a float. If we replace float to double, the program works. Therefore, this is a type conversion from float to double.

### Function with Default Arguments:-

- C++

```
#include <iostream>

using namespace std;

void fun(int);

void fun(int, int);

void fun(int i) { cout << "Value of i is : " << i << endl; }

void fun(int a, int b = 9)

{

    cout << "Value of a is : " << a << endl;

    cout << "Value of b is : " << b << endl;

}

int main()

{
```

```
    fun(12);

    return 0;

}

// Code Submitted By Susobhan Akhuli
```

The above example shows an error “*call of overloaded ‘fun(int)’ is ambiguous*”. The fun(int a, int b=9) can be called in two ways: first is by calling the function with one argument, i.e., fun(12) and another way is calling the function with two arguments, i.e., fun(4,5). The fun(int i) function is invoked with one argument. Therefore, the compiler could not be able to select among fun(int i) and fun(int a,int b=9).

### Function with Pass By Reference:-

- C++

```
#include <iostream>

using namespace std;

void fun(int);

void fun(int&);

int main()

{

    int a = 10;

    fun(a); // error, which fun()?
```

```

    return 0;

}

void fun(int x) { cout << "Value of x is : " << x << endl; }

void fun(int& b)

{

    cout << "Value of b is : " << b << endl;

}

// Code Submitted By Susobhan Akhuli

```

The above example shows an error “*call of overloaded ‘fun(int&)’ is ambiguous*”. The first function takes one integer argument and the second function takes a reference parameter as an argument. In this case, the compiler does not know which function is needed by the user as there is no syntactical difference between the fun(int) and fun(int &).

### Friend Function

- A friend function is a special function in C++ which in spite of not being member function of a class has privilege to access private and protected data of a class.
- A friend function is a non member function or ordinary function of a class, which is declared as a friend using the keyword “friend” inside the class. By declaring a function as a friend, all the access permissions are given to the function.
- The keyword “friend” is placed only in the function declaration of the friend function and not in the function definition.
- When friend function is called neither name of object nor dot operator is used. However it may accept the object as argument whose value it want to access.
- Friend function can be declared in any section of the class i.e. public or private or protected.

Declaration of friend function in C++  
 Syntax :

```
class <class_name>
{
    friend <return_type> <function_name>(argument/s);
};
```

Example\_1: Find the largest of two numbers using Friend Function

- C++

```
#include<iostream>

using namespace std;

class Largest
{
    int a,b,m;

    public:

        void set_data();

        friend void find_max(Largest);

};

void Largest::set_data()
{
    cout<<"Enter the First No:";

    cin>>a;

    cout<<"Enter the Second No:";
```

---

```
        cin>>b;

    }

void find_max(Largest t)

{

    if(t.a>t.b)

        t.m=t.a;

    else

        t.m=t.b;

    cout<<"Maximum Number is\t"<<t.m;

}

main()

{

    Largest l;

    l.set_data();

    find_max(l);

    return 0;
```

---

```
}
```

### Output

```
Enter the First No:Enter the Second No:Maximum Number is  
2117529904
```

