Learn C++ Programming

programiz.com/cpp-programming

About C++ Programming

- **Multi-paradigm Language** - C++ supports at least seven different styles of programming. Developers can choose any of the styles.
- **General Purpose Language** - You can use C++ to develop games, desktop apps, operating systems, and so on.
- **Speed** - Like C programming, the performance of optimized C++ code is exceptional.
- **Object-oriented** - C++ allows you to divide complex problems into smaller sets by using objects.

Why Learn C++?

- C++ is used to develop games, desktop apps, operating systems, browsers, and so on because of its performance.
- After learning C++, it will be much easier to learn other programming languages like Java, Python, etc.
- C++ helps you to understand the internal architecture of a computer, how computer stores and retrieves information.

How to learn C++?

- **C++ tutorial from Programiz** - We provide step by step C++ tutorials, examples, and references. Get started with C++.
- **Official C++ documentation** - Might be hard to follow and understand for beginners. Visit official C++ documentation.
- **Write a lot of C++ programming code**- The only way you can learn programming is by writing a lot of code.
- **Read C++ code**- Join Github's open-source projects and read other people's code.

C++ Variables, Literals and Constants

programiz.com/cpp-programming/variables-literals

Join our newsletter for the latest updates.

In this tutorial, we will learn about variables, literals, and constants in C++ with the help of examples.

C++ Variables

In programming, a variable is a container (storage area) to hold data.

To indicate the storage area, each variable should be given a unique name (identifier). For example,

```
int age = 14;
```

Here, *age* is a variable of the `int` data type, and we have assigned an integer value 14 to it.

Note: The `int` data type suggests that the variable can only hold integers. Similarly, we can use the `double` data type if we have to store decimals and exponentials.

We will learn about all the data types in detail in the next tutorial.

The value of a variable can be changed, hence the name **variable**.

```
int age = 14;   // age is 14
age = 17;       // age is 17
```

Rules for naming a variable

- A variable name can only have alphabets, numbers, and the underscore `_`.
- A variable name cannot begin with a number.
- Variable names should not begin with an uppercase character.
- A variable name cannot be a keyword. For example, `int` is a keyword that is used to denote integers.
- A variable name can start with an underscore. However, it's not considered a good practice.

Note: We should try to give meaningful names to variables. For example, *first_name* is a better variable name than *fn*.

C++ Literals

Literals are data used for representing fixed values. They can be used directly in the code. For example: `1`, `2.5`, `'c'` etc.

Here, `1`, `2.5` and `'c'` are literals. Why? You cannot assign different values to these terms.

Here's a list of different literals in C++ programming.

---

1. Integers

An integer is a numeric literal(associated with numbers) without any fractional or exponential part. There are three types of integer literals in C programming:

- decimal (base 10)
- octal (base 8)
- hexadecimal (base 16)

For example:

```
Decimal: 0, -9, 22 etc
Octal: 021, 077, 033 etc
Hexadecimal: 0x7f, 0x2a, 0x521 etc
```

In C++ programming, octal starts with a `0`, and hexadecimal starts with a `0x`.

---

2. Floating-point Literals

A floating-point literal is a numeric literal that has either a fractional form or an exponent form. For example:

`-2.0`

`0.0000234`

`-0.22E-5`

Note: `E-5 = 10^-5`

---

3. Characters

A character literal is created by enclosing a single character inside single quotation marks. For example: `'a'`, `'m'`, `'F'`, `'2'`, `'}'` etc.

---

4. Escape Sequences

Sometimes, it is necessary to use characters that cannot be typed or has special meaning in C++ programming. For example, newline (enter), tab, question mark, etc.

In order to use these characters, escape sequences are used.

Escape Sequences	Characters
`\b`	Backspace
`\f`	Form feed
`\n`	Newline
`\r`	Return
`\t`	Horizontal tab
`\v`	Vertical tab
`\\`	Backslash
`\'`	Single quotation mark
`\"`	Double quotation mark
`\?`	Question mark
`\0`	Null Character

5. String Literals

A string literal is a sequence of characters enclosed in double-quote marks. For example:

`"good"`	string constant
`""`	null string constant
`" "`	string constant of six white space
`"x"`	string constant having a single character
`"Earth is round\n"`	prints string with a newline

We will learn about strings in detail in the C++ string tutorial.

C++ Constants

In C++, we can create variables whose value cannot be changed. For that, we use the `const` keyword. Here's an example:

```
const int LIGHT_SPEED = 299792458;
LIGHT_SPEED = 2500 // Error! LIGHT_SPEED is a constant.
```

Here, we have used the keyword `const` to declare a constant named `LIGHT_SPEED`. If we try to change the value of `LIGHT_SPEED`, we will get an error.

A constant can also be created using the `#define` preprocessor directive. We will learn about it in detail in the C++ Macros tutorial.

C++ Data Types

programiz.com/cpp-programming/data-types

Join our newsletter for the latest updates.

In this tutorial, we will learn about basic data types such as int, float, char, etc. in C++ programming with the help of examples.

In C++, data types are declarations for variables. This determines the type and size of data associated with variables. For example,

```
int age = 13;
```

Here, *age* is a variable of type `int`. Meaning, the variable can only store integers of either 2 or 4 bytes.

C++ Fundamental Data Types

The table below shows the fundamental data types, their meaning, and their sizes (in bytes):

Data Type	Meaning	Size (in Bytes)
`int`	Integer	2 or 4
`float`	Floating-point	4
`double`	Double Floating-point	8
`char`	Character	1
`wchar_t`	Wide Character	2
`bool`	Boolean	1
`void`	Empty	0

Now, let us discuss these fundamental data types in more detail.

1. C++ int

- The `int` keyword is used to indicate integers.
- Its size is usually 4 bytes. Meaning, it can store values from **-2147483648 to 2147483647**.
- For example,

```
int salary = 85000;
```

2. C++ float and double

- `float` and `double` are used to store floating-point numbers (decimals and exponentials).
- The size of `float` is 4 bytes and the size of `double` is 8 bytes. Hence, `double` has two times the precision of `float`. To learn more, visit C++ float and double.
- For example,

```
float area = 64.74;
double volume = 134.64534;
```

As mentioned above, these two data types are also used for exponentials. For example,

```
double distance = 45E12    // 45E12 is equal to 45*10^12
```

3. C++ char

- Keyword `char` is used for characters.
- Its size is 1 byte.
- Characters in C++ are enclosed inside single quotes `' '`.
- For example,

```
char test = 'h';
```

Note: In C++, an integer value is stored in a `char` variable rather than the character itself. To learn more, visit C++ characters.

4. C++ wchar_t

- Wide character `wchar_t` is similar to the `char` data type, except its size is 2 bytes instead of 1.
- It is used to represent characters that require more memory to represent them than a single `char`.
- For example,

```
wchar_t test = L'□'  // storing Hebrew character;
```

Notice the letter L before the quotation marks.

Note: There are also two other fixed-size character types `char16_t` and `char32_t` introduced in C++11.

5. C++ bool

- The `bool` data type has one of two possible values: `true` or `false`.

- Booleans are used in conditional statements and loops (which we will learn in later chapters).
- For example,

```
bool cond = false;
```

6. C++ void

- The `void` keyword indicates an absence of data. It means "nothing" or "no value".
- We will use void when we learn about functions and pointers.

Note: We cannot declare variables of the `void` type.

C++ Type Modifiers

We can further modify some of the fundamental data types by using type modifiers. There are 4 type modifiers in C++. They are:

1. `signed`
2. `unsigned`
3. `short`
4. `long`

We can modify the following data types with the above modifiers:

- `int`
- `double`
- `char`

C++ Modified Data Types List

Data Type	Size (in Bytes)	Meaning
`signed int`	4	used for integers (equivalent to `int`)
`unsigned int`	4	can only store positive integers
`short`	2	used for small integers (range **-32768 to 32767**)
`unsigned short`	2	used for small positive integers (range **0 to 65,535**)
`long`	at least 4	used for large integers (equivalent to `long int`)
`unsigned long`	4	used for large positive integers or 0 (equivalent to `unsigned long int`)

`long long`	8	used for very large integers (equivalent to `long long int`).
`unsigned long long`	8	used for very large positive integers or 0 (equivalent to `unsigned long long int`)
`long double`	12	used for large floating-point numbers
`signed char`	1	used for characters (guaranteed range **-127 to 127**)
`unsigned char`	1	used for characters (range **0 to 255**)

Let's see a few examples.

```
long b = 4523232;
long int c = 2345342;
long double d = 233434.56343;
short d = 3434233; // Error! out of range
unsigned int a = -5;    // Error! can only store positive numbers or 0
```

Derived Data Types

Data types that are derived from fundamental data types are derived types. For example: arrays, pointers, function types, structures, etc.

We will learn about these derived data types in later tutorials.

C++ Basic Input/Output

programiz.com/cpp-programming/input-output

In this tutorial, we will learn to use the cin object to take input from the user, and the cout object to display output to the user with the help of examples.

C++ Output

In C++, `cout` sends formatted output to standard output devices, such as the screen. We use the `cout` object along with the `<<` operator for displaying output.

Example 1: String Output

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

int main() {
    // prints the string enclosed in double quotes
    cout << "This is C++ Programming";
    return 0;
}
```

Output

```
This is C++ Programming
```

How does this program work?

- We first include the `iostream` header file that allows us to display output.
- The `cout` object is defined inside the `std` namespace. To use the `std` namespace, we used the `using namespace std;` statement.
- Every C++ program starts with the `main()` function. The code execution begins from the start of the `main()` function.
- `cout` is an object that prints the string inside quotation marks `" "`. It is followed by the `<<` operator.
- `return 0;` is the "exit status" of the `main()` function. The program ends with this statement, however, this statement is not mandatory.

Note: If we don't include the `using namespace std;` statement, we need to use `std::cout` instead of `cout`.

This is the preferred method as using the `std` namespace can create potential problems.

However, we have used the `std` namespace in our tutorials in order to make the codes more readable.

```
#include <iostream>

int main() {
    // prints the string enclosed in double quotes
    std::cout << "This is C++ Programming";
    return 0;
}
```

Example 2: Numbers and Characters Output

To print the numbers and character variables, we use the same `cout` object but without using quotation marks.

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

int main() {
    int num1 = 70;
    double num2 = 256.783;
    char ch = 'A';

    cout << num1 << endl;    // print integer
    cout << num2 << endl;    // print double
    cout << "character: " << ch << endl;    // print char
    return 0;
}
```

Output

```
70
256.783
character: A
```

Notes:

- The `endl` manipulator is used to insert a new line. That's why each output is displayed in a new line.
- The `<<` operator can be used more than once if we want to print different variables, strings and so on in a single statement. For example:

```
cout << "character: " << ch << endl;
```

C++ Input

In C++, `cin` takes formatted input from standard input devices such as the keyboard. We use the `cin` object along with the `>>` operator for taking input.

Example 3: Integer Input/Output

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

int main() {
    int num;
    cout << "Enter an integer: ";
    cin >> num;   // Taking input
    cout << "The number is: " << num;
    return 0;
}
```

Output

```
Enter an integer: 70
The number is: 70
```

In the program, we used

```
cin >> num;
```

to take input from the user. The input is stored in the variable *num*. We use the `>>` operator with `cin` to take input.

Note: If we don't include the `using namespace std;` statement, we need to use `std::cin` instead of `cin`.

C++ Taking Multiple Inputs

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

int main() {
    char a;
    int num;

    cout << "Enter a character and an integer: ";
    cin >> a >> num;

    cout << "Character: " << a << endl;
    cout << "Number: " << num;

    return 0;
}
```

Output

```
Enter a character and an integer: F
23
Character: F
Number: 23
```

C++ Type Conversion

In this tutorial, we will learn about the basics of C++ type conversion with the help of examples.

C++ allows us to convert data of one type to that of another. This is known as type conversion.

There are two types of type conversion in C++.

1. Implicit Conversion
2. Explicit Conversion (also known as Type Casting)

Implicit Type Conversion

The type conversion that is done automatically done by the compiler is known as implicit type conversion. This type of conversion is also known as automatic conversion.

Let us look at two examples of implicit type conversion.

Example 1: Conversion From int to double

```
// Working of implicit type-conversion

#include <iostream>
using namespace std;

int main() {
   // assigning an int value to num_int
   int num_int = 9;

   // declaring a double type variable
   double num_double;

   // implicit conversion
   // assigning int value to a double variable
   num_double = num_int;

   cout << "num_int = " << num_int << endl;
   cout << "num_double = " << num_double << endl;

   return 0;
}
```

Output

```
num_int = 9
num_double = 9
```

In the program, we have assigned an `int` data to a `double` variable.

```
num_double = num_int;
```

Here, the `int` value is automatically converted to `double` by the compiler before it is assigned to the *num_double* variable. This is an example of implicit type conversion.

Example 2: Automatic Conversion from double to int

```
//Working of Implicit type-conversion

#include <iostream>
using namespace std;

int main() {

   int num_int;
   double num_double = 9.99;

   // implicit conversion
   // assigning a double value to an int variable
   num_int = num_double;

   cout << "num_int = " << num_int << endl;
   cout << "num_double = " << num_double << endl;

   return 0;
}
```

Output

```
num_int = 9
num_double = 9.99
```

In the program, we have assigned a `double` data to an `int` variable.

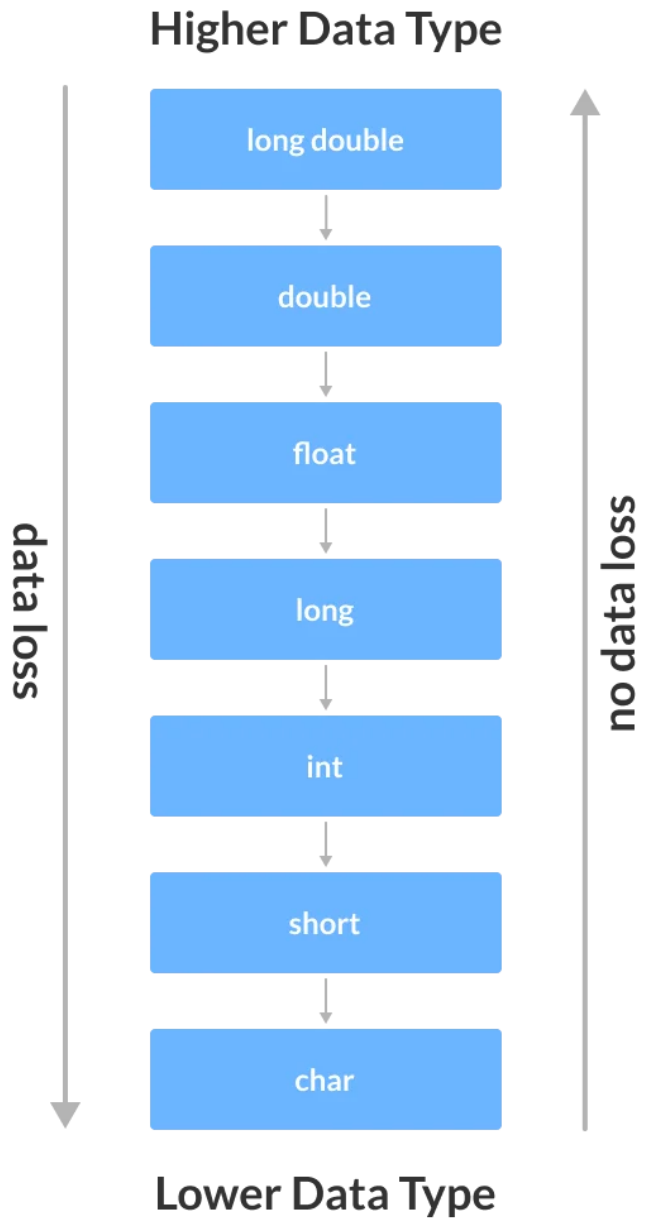
```
num_double = num_int;
```

Here, the `double` value is automatically converted to `int` by the compiler before it is assigned to the *num_int* variable. This is also an example of implicit type conversion.

Note: Since `int` cannot have a decimal part, the digits after the decimal point are truncated in the above example.

Data Loss During Conversion (Narrowing Conversion)

As we have seen from the above example, conversion from one data type to another is prone to data loss. This happens when data of a larger type is converted to data of a smaller type.



Possible Data Loss During Type Conversion

C++ Explicit Conversion

When the user manually changes data from one type to another, this is known as **explicit conversion**. This type of conversion is also known as **type casting**.

There are three major ways in which we can use explicit conversion in C++. They are:

1. C-style type casting (also known as **cast notation**)
2. Function notation (also known as **old C++ style type casting**)
3. Type conversion operators

C-style Type Casting

As the name suggests, this type of casting is favored by the **C programming language**. It is also known as **cast notation**.

The syntax for this style is:

```
(data_type)expression;
```

For example,

```
// initializing int variable
int num_int = 26;

// declaring double variable
double num_double;

// converting from int to double
num_double = (double)num_int;
```

Function-style Casting

We can also use the function like notation to cast data from one type to another.

The syntax for this style is:

```
data_type(expression);
```

For example,

```
// initializing int variable
int num_int = 26;

// declaring double variable
double num_double;

// converting from int to double
num_double = double(num_int);
```

Example 3: Type Casting

```
#include <iostream>

using namespace std;

int main() {
    // initializing a double variable
    double num_double = 3.56;
    cout << "num_double = " << num_double << endl;

    // C-style conversion from double to int
    int num_int1 = (int)num_double;
    cout << "num_int1   = " << num_int1 << endl;

    // function-style conversion from double to int
    int num_int2 = int(num_double);
    cout << "num_int2   = " << num_int2 << endl;

    return 0;
}
```

Output

```
num_double = 3.56
num_int1   = 3
num_int2   = 3
```

We used both the **C style type conversion** and the **function-style casting for type conversion** and displayed the results. Since they perform the same task, both give us the same output.

Type Conversion Operators

Besides these two type castings, C++ also has four operators for type conversion. They are known as **type conversion operators**. They are:

- `static_cast`
- `dynamic_cast`
- `const_cast`
- `reinterpret_cast`

We will learn about these casts in later tutorials.

Recommended Tutorials:

C++ Operators

programiz.com/cpp-programming/operators

Join our newsletter for the latest updates.

In this tutorial, we will learn about the different types of operators in C++ with the help of examples. In programming, an operator is a symbol that operates on a value or a variable.

Operators are symbols that perform operations on variables and values. For example, `+` is an operator used for addition, while `-` is an operator used for subtraction.

Operators in C++ can be classified into 6 types:

1. C++ Arithmetic Operators

Arithmetic operators are used to perform arithmetic operations on variables and data. For example,

```
a + b;
```

Here, the `+` operator is used to add two variables *a* and *b*. Similarly there are various other arithmetic operators in C++.

Operator	Operation
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`%`	Modulo Operation (Remainder after division)

Example 1: Arithmetic Operators

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

int main() {
    int a, b;
    a = 7;
    b = 2;

    // printing the sum of a and b
    cout << "a + b = " << (a + b) << endl;

    // printing the difference of a and b
    cout << "a - b = " << (a - b) << endl;

    // printing the product of a and b
    cout << "a * b = " << (a * b) << endl;

    // printing the division of a by b
    cout << "a / b = " << (a / b) << endl;

    // printing the modulo of a by b
    cout << "a % b = " << (a % b) << endl;

    return 0;
}
```

Output

```
a + b = 9
a - b = 5
a * b = 14
a / b = 3
a % b = 1
```

Here, the operators `+`, `-` and `*` compute addition, subtraction, and multiplication respectively as we might have expected.

/ Division Operator

Note the operation `(a / b)` in our program. The `/` operator is the division operator.

As we can see from the above example, if an integer is divided by another integer, we will get the quotient. However, if either divisor or dividend is a floating-point number, we will get the result in decimals.

```
In C++,

7/2 is 3
7.0 / 2 is 3.5
7 / 2.0 is 3.5
7.0 / 2.0 is 3.5
```

% Modulo Operator

The modulo operator `%` computes the remainder. When `a = 9` is divided by `b = 4`, the remainder is **1**.

Note: The `%` operator can only be used with integers.

Increment and Decrement Operators

C++ also provides increment and decrement operators: `++` and `--` respectively.

- `++` increases the value of the operand by **1**
- `--` decreases it by **1**

For example,

```
int num = 5;

// increment operator
++num;  // 6
```

Here, the code `++num;` increases the value of *num* by **1**.

Example 2: Increment and Decrement Operators

```
// Working of increment and decrement operators

#include <iostream>
using namespace std;

int main() {
    int a = 10, b = 100, result_a, result_b;

    // incrementing a by 1 and storing the result in result_a
    result_a = ++a;
    cout << "result_a = " << result_a << endl;


    // decrementing b by 1 and storing the result in result_b
    result_b = --b;
    cout << "result_b = " << result_b << endl;

    return 0;
}
```

Output

```
result_a = 11
result_b = 99
```

In the above program, we have used the `++` and `--` operators as **prefixes (++a and --b)**. However, we can also use these operators as **postfix (a++ and b--)**.

To learn more, visit increment and decrement operators.

2. C++ Assignment Operators

In C++, assignment operators are used to assign values to variables. For example,

```
// assign 5 to a
a = 5;
```

Here, we have assigned a value of `5` to the variable *a*.

Operator	Example	Equivalent to
`=`	`a = b;`	`a = b;`
`+=`	`a += b;`	`a = a + b;`
`-=`	`a -= b;`	`a = a - b;`
`*=`	`a *= b;`	`a = a * b;`
`/=`	`a /= b;`	`a = a / b;`
`%=`	`a %= b;`	`a = a % b;`

Example 3: Assignment Operators

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

int main() {
    int a, b;

    // 2 is assigned to a
    a = 2;

    // 7 is assigned to b
    b = 7;

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "\nAfter a += b;" << endl;

    // assigning the sum of a and b to a
    a += b;  // a = a +b
    cout << "a = " << a << endl;

    return 0;
}
```

Output

```
a = 2
b = 7

After a += b;
a = 9
```

3. C++ Relational Operators

A relational operator is used to check the relationship between two operands. For example,

```
// checks if a is greater than b
a > b;
```

Here, `>` is a relational operator. It checks if *a* is greater than *b* or not.

If the relation is **true**, it returns **1** whereas if the relation is **false**, it returns **0**.

Operator	Meaning	Example
`==`	Is Equal To	`3 == 5` gives us **false**
`!=`	Not Equal To	`3 != 5` gives us **true**
`>`	Greater Than	`3 > 5` gives us **false**
`<`	Less Than	`3 < 5` gives us **true**
`>=`	Greater Than or Equal To	`3 >= 5` give us **false**
`<=`	Less Than or Equal To	`3 <= 5` gives us **true**

Example 4: Relational Operators

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

int main() {
    int a, b;
    a = 3;
    b = 5;
    bool result;

    result = (a == b);   // false
    cout << "3 == 5 is " << result << endl;

    result = (a != b);  // true
    cout << "3 != 5 is " << result << endl;

    result = a > b;   // false
    cout << "3 > 5 is " << result << endl;

    result = a < b;   // true
    cout << "3 < 5 is " << result << endl;

    result = a >= b;  // false
    cout << "3 >= 5 is " << result << endl;

    result = a <= b;  // true
    cout << "3 <= 5 is " << result << endl;

    return 0;
}
```

Output

```
3 == 5 is 0
3 != 5 is 1
3 > 5 is 0
3 < 5 is 1
3 >= 5 is 0
3 <= 5 is 1
```

Note: Relational operators are used in decision-making and loops.

4. C++ Logical Operators

Logical operators are used to check whether an expression is **true** or **false**. If the expression is **true**, it returns **1** whereas if the expression is **false**, it returns **0**.

Operator	Example	Meaning
`&&`	expression1 **&&** expression2	Logical AND. True only if all the operands are true.
`||`	expression1 **||** expression2	Logical OR. True if at least one of the operands is true.

!	!expression	Logical NOT. True only if the operand is false.

In C++, logical operators are commonly used in decision making. To further understand the logical operators, let's see the following examples,

```
Suppose,
a = 5
b = 8

Then,

(a > 3) && (b > 5) evaluates to true
(a > 3)  && (b < 5) evaluates to false

(a > 3) || (b > 5) evaluates to true
(a > 3) || (b < 5) evaluates to true
(a < 3) || (b < 5) evaluates to false

!(a < 3) evaluates to true
!(a > 3) evaluates to false
```

Example 5: Logical Operators

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

int main() {
    bool result;

    result = (3 != 5) && (3 < 5);     // true
    cout << "(3 != 5) && (3 < 5) is " << result << endl;

    result = (3 == 5) && (3 < 5);    // false
    cout << "(3 == 5) && (3 < 5) is " << result << endl;

    result = (3 == 5) && (3 > 5);    // false
    cout << "(3 == 5) && (3 > 5) is " << result << endl;

    result = (3 != 5) || (3 < 5);    // true
    cout << "(3 != 5) || (3 < 5) is " << result << endl;

    result = (3 != 5) || (3 > 5);    // true
    cout << "(3 != 5) || (3 > 5) is " << result << endl;

    result = (3 == 5) || (3 > 5);    // false
    cout << "(3 == 5) || (3 > 5) is " << result << endl;

    result = !(5 == 2);    // true
    cout << "!(5 == 2) is " << result << endl;

    result = !(5 == 5);    // false
    cout << "!(5 == 5) is " << result << endl;

    return 0;
}
```

Output

```
(3 != 5) && (3 < 5) is 1
(3 == 5) && (3 < 5) is 0
(3 == 5) && (3 > 5) is 0
(3 != 5) || (3 < 5) is 1
(3 != 5) || (3 > 5) is 1
(3 == 5) || (3 > 5) is 0
!(5 == 2) is 1
!(5 == 5) is 0
```

Explanation of logical operator program

- `(3 != 5) && (3 < 5)` evaluates to **1** because both operands `(3 != 5)` and `(3 < 5)` are **1** (true).
- `(3 == 5) && (3 < 5)` evaluates to **0** because the operand `(3 == 5)` is **0** (false).
- `(3 == 5) && (3 > 5)` evaluates to **0** because both operands `(3 == 5)` and `(3 > 5)` are **0** (false).
- `(3 != 5) || (3 < 5)` evaluates to **1** because both operands `(3 != 5)` and `(3 < 5)` are **1** (true).
- `(3 != 5) || (3 > 5)` evaluates to **1** because the operand `(3 != 5)` is **1** (true).
- `(3 == 5) || (3 > 5)` evaluates to **0** because both operands `(3 == 5)` and `(3 > 5)` are **0** (false).
- `!(5 == 2)` evaluates to **1** because the operand `(5 == 2)` is **0** (false).
- `!(5 == 5)` evaluates to **0** because the operand `(5 == 5)` is **1** (true).

---

5. C++ Bitwise Operators

In C++, bitwise operators are used to perform operations on individual bits. They can only be used alongside `char` and `int` data types.

Operator	Description
`&`	Binary AND
`|`	Binary OR
`^`	Binary XOR
`~`	Binary One's Complement
`<<`	Binary Shift Left
`>>`	Binary Shift Right

To learn more, visit C++ bitwise operators.

---

6. Other C++ Operators

Here's a list of some other common operators available in C++. We will learn about them in later tutorials.

Operator	Description	Example
`sizeof`	returns the size of data type	`sizeof(int); // 4`
`?:`	returns value based on the condition	`string result = (5 > 0) ? "even" : "odd"; // "even"`
`&`	represents memory address of the operand	`&num; // address of num`
`.`	accesses members of struct variables or class objects	`s1.marks = 92;`
`->`	used with pointers to access the class or struct variables	`ptr->marks = 92;`
`<<`	prints the output value	`cout << 5;`
`>>`	gets the input value	`cin >> num;`

C++ Comments

programiz.com/cpp-programming/comments

In this tutorial, we will learn about C++ comments, why we use them, and how to use them with the help of examples.

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 Comments

`/* */` -Multi-line Comments

Single Line Comments

In C++, any line that starts with `//` is a comment. For example,

```
// declaring a variable
int a;

// initializing the variable 'a' with the value 2
a = 2;
```

Here, we have used two single-line comments:

- `// declaring a variable`
- `// initializing the variable 'a' with the value 2`

We can also use single line comment like this:

```
int a;    // declaring a variable
```

Multi-line comments

In C++, any line between `/*` and `*/` is also a comment. For example,

```
/* declaring a variable
to store salary to employees
*/
int salary = 2000;
```

This syntax can be used to write both single-line and multi-line comments.

Using Comments for Debugging

Comments can also be used to disable code to prevent it from being executed. For example,

```
#include <iostream>
using namespace std;
int main() {
   cout << "some code";
   cout << ''error code;
   cout << "some other code";

   return 0;
}
```

If we get an error while running the program, instead of removing the error-prone code, we can use comments to disable it from being executed; this can be a valuable debugging tool.

```
#include <iostream>
using namespace std;
int main() {
   cout << "some code";
   // cout << ''error code;
   cout << "some other code";

   return 0;
}
```

Pro Tip: Remember the shortcut for using comments; it can be really helpful. For most code editors, it's `Ctrl + /` for Windows and `Cmd + /` for Mac.

Why use Comments?

If we write comments on our code, it will be easier for us to understand the code in the future. Also, it will be easier for your fellow developers to understand the code.

Note: Comments shouldn't be the substitute for a way to explain poorly written code in English. We should always write well-structured and self-explanatory code. And, then use comments.

As a general rule of thumb, use comments to explain **Why** you did something rather than **How** you did something, and you are good.

C++ if, if...else and Nested if...else

programiz.com/cpp-programming/if-else

Join our newsletter for the latest updates.

In this tutorial, we will learn about the if...else statement to create decision making programs with the help of examples.

In computer programming, we use the `if` statement to run a block code only when a certain condition is met.

For example, assigning grades (A, B, C) based on marks obtained by a student.

- if the percentage is above **90**, assign grade **A**
- if the percentage is above **75**, assign grade **B**
- if the percentage is above **65**, assign grade **C**

There are three forms of `if...else` statements in C++.

1. `if` statement
2. `if...else` statement
3. `if...else if...else` statement

C++ if Statement

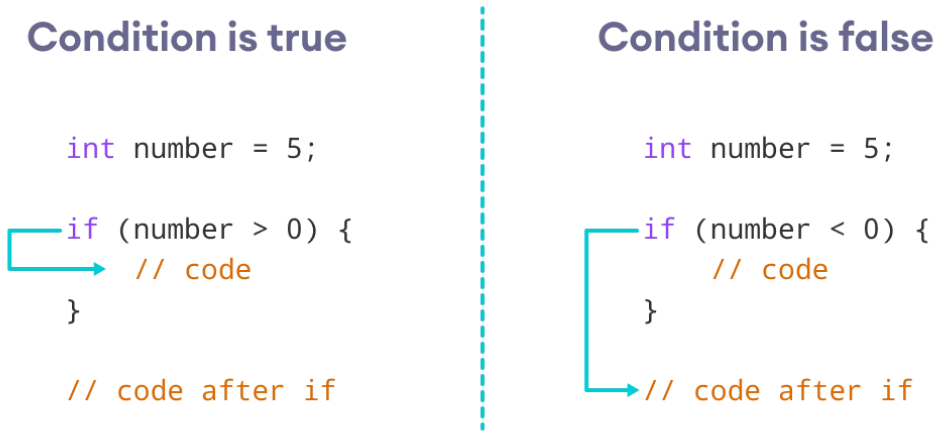
The syntax of the `if` statement is:

```
if (condition) {
   // body of if statement
}
```

The `if` statement evaluates the `condition` inside the parentheses `( )`.

- If the `condition` evaluates to `true`, the code inside the body of `if` is executed.
- If the `condition` evaluates to `false`, the code inside the body of `if` is skipped.

Note: The code inside `{ }` is the body of the `if` statement.



Working of C++ if Statement

Example 1: C++ if Statement

```
// Program to print positive number entered by the user
// If the user enters a negative number, it is skipped

#include <iostream>
using namespace std;

int main() {
    int number;

    cout << "Enter an integer: ";
    cin >> number;

    // checks if the number is positive
    if (number > 0) {
        cout << "You entered a positive integer: " << number << endl;
    }
    cout << "This statement is always executed.";
    return 0;
}
```

Output 1

```
Enter an integer: 5
You entered a positive number: 5
This statement is always executed.
```

When the user enters `5`, the condition `number > 0` is evaluated to `true` and the statement inside the body of `if` is executed.

Output 2

```
Enter a number: -5
This statement is always executed.
```

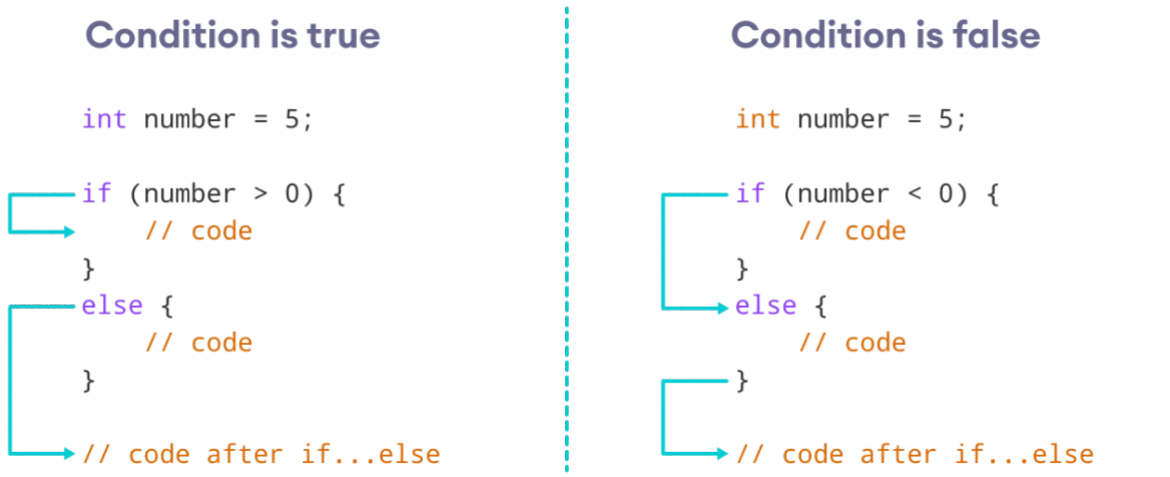
When the user enters `-5`, the condition `number > 0` is evaluated to `false` and the statement inside the body of `if` is not executed.

C++ if...else

The `if` statement can have an optional `else` clause. Its syntax is:

```
if (condition) {
    // block of code if condition is true
}
else {
    // block of code if condition is false
}
```

The `if..else` statement evaluates the `condition` inside the parenthesis.



Working of C++ if...else

If the `condition` evaluates `true`,

- the code inside the body of `if` is executed
- the code inside the body of `else` is skipped from execution

If the `condition` evaluates `false`,

- the code inside the body of `else` is executed
- the code inside the body of `if` is skipped from execution

Example 2: C++ if...else Statement

```
// Program to check whether an integer is positive or negative
// This program considers 0 as a positive number

#include <iostream>
using namespace std;

int main() {
     int number;

    cout << "Enter an integer: ";
    cin >> number;
    if (number >= 0) {
        cout << "You entered a positive integer: " << number << endl;
    }
    else {
        cout << "You entered a negative integer: " << number << endl;
    }
    cout << "This line is always printed.";
    return 0;
}
```

Output 1

```
Enter an integer: 4
You entered a positive integer: 4.
This line is always printed.
```

In the above program, we have the condition `number >= 0`. If we enter the number greater or equal to *0*, then the condition evaluates `true`.

Here, we enter *4*. So, the condition is `true`. Hence, the statement inside the body of `if` is executed.

Output 2

```
Enter an integer: -4
You entered a negative integer: -4.
This line is always printed.
```

Here, we enter *-4*. So, the condition is `false`. Hence, the statement inside the body of `else` is executed.

C++ if...else...else if statement

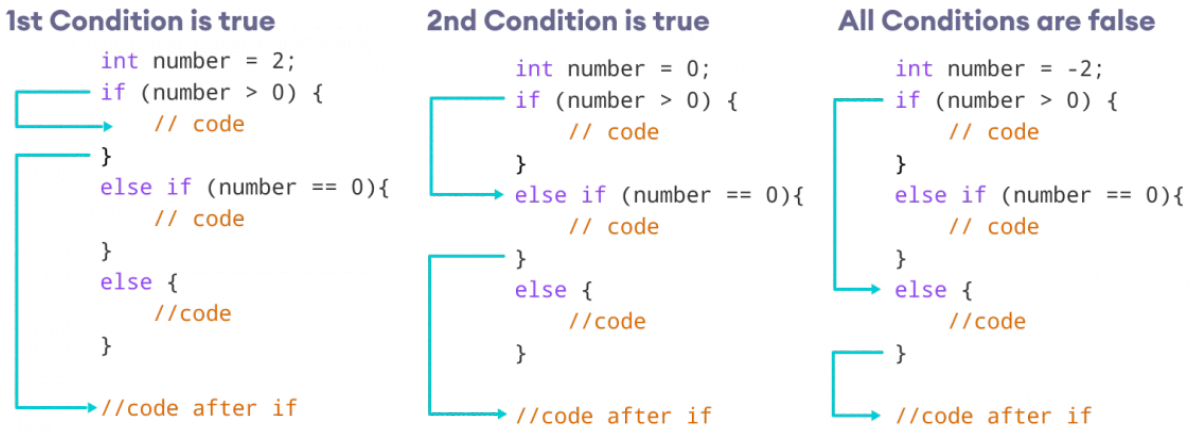
The `if...else` statement is used to execute a block of code among two alternatives. However, if we need to make a choice between more than two alternatives, we use the `if...else if...else` statement.

The syntax of the `if...else if...else` statement is:

```
if (condition1) {
    // code block 1
}
else if (condition2){
    // code block 2
}
else {
    // code block 3
}
```

Here,

- If `condition1` evaluates to `true`, the `code block 1` is executed.
- If `condition1` evaluates to `false`, then `condition2` is evaluated.
- If `condition2` is `true`, the `code block 2` is executed.
- If `condition2` is `false`, the `code block 3` is executed.



How if...else if...else Statement Works

Note: There can be more than one `else if` statement but only one `if` and `else` statements.

Example 3: C++ if...else...else if

```
// Program to check whether an integer is positive, negative or zero

#include <iostream>
using namespace std;

int main() {
     int number;

    cout << "Enter an integer: ";
    cin >> number;
    if (number > 0) {
        cout << "You entered a positive integer: " << number << endl;
    }
else if (number < 0) {
      cout << "You entered a negative integer: " << number << endl;
     }
else {
        cout << "You entered 0." << endl;
    }
     cout << "This line is always printed.";
    return 0;
}
```

Output 1

```
Enter an integer: 1
You entered a positive integer: 1.
This line is always printed.
```

Output 2

```
Enter an integer: -2
You entered a negative integer: -2.
This line is always printed.
```

Output 3

```
Enter an integer: 0
You entered 0.
This line is always printed.
```

In this program, we take a number from the user. We then use the `if...else if...else` ladder to check whether the number is positive, negative, or zero.

If the number is greater than `0`, the code inside the `if` block is executed. If the number is less than `0`, the code inside the `else if` block is executed. Otherwise, the code inside the `else` block is executed.

C++ Nested if...else

Sometimes, we need to use an `if` statement inside another `if` statement. This is known as nested `if` statement.

Think of it as multiple layers of `if` statements. There is a first, outer `if` statement, and inside it is another, inner `if` statement. Its syntax is:

```
// outer if statement
if (condition1) {
    // statements

    // inner if statement
    if (condition2) {
        // statements
    }
}
```

Notes:

- We can add `else` and `else if` statements to the inner `if` statement as required.
- The inner `if` statement can also be inserted inside the outer `else` or `else if` statements (if they exist).
- We can nest multiple layers of `if` statements.

Example 4: C++ Nested if

```
// C++ program to find if an integer is even or odd or neither (0)
// using nested if statements

#include <iostream>
using namespace std;

int main() {
    int num;

    cout << "Enter an integer: ";
     cin >> num;

    // outer if condition
    if (num != 0) {

        // inner if condition
        if ((num % 2) == 0) {
            cout << "The number is even." << endl;
        }
         // inner else condition
        else {
            cout << "The number is odd." << endl;
        }
    }
    // outer else condition
    else {
        cout << "The number is 0 and it is neither even nor odd." << endl;
    }
    cout << "This line is always printed." << endl;
}
```

Output 1

```
Enter an integer: 34
The number is even.
This line is always printed.
```

Output 2

```
Enter an integer: 35
The number is odd.
This line is always printed.
```

Output 3

```
Enter an integer: 0
The number is 0 and it is neither even nor odd.
This line is always printed.
```

In the above example,

- We take an integer as an input from the user and store it in the variable *num*.
- We then use an `if...else` statement to check whether num is not equal to `0`.
  - If `true`, then the **inner** `if...else` statement is executed.
  - If `false`, the code inside the **outer** `else` condition is executed, which prints "The number is 0 and neither even nor odd."
- The **inner** `if...else` statement checks whether the input number is divisible by `2`.
  - If `true`, then we print a statement saying that the number is even.
  - If `false`, we print that the number is odd.

Notice that `0` is also divisible by `2`, but it is actually not an even number. This is why we first make sure that the input number is not `0` in the outer `if` condition.

Note: As you can see, nested `if...else` makes your logic complicated. If possible, you should always try to avoid nested `if...else`.

Body of if...else With Only One Statement

If the body of `if...else` has only one statement, you can omit `{ }` in the program. For example, you can replace

```
int number = 5;

    if (number > 0) {
        cout << "The number is positive." << endl;
    }
    else {
        cout << "The number is negative." << endl;
    }
```

with

```
int number = 5;

    if (number > 0)
        cout << "The number is positive." << endl;
    else
        cout << "The number is negative." << endl;
```

The output of both programs will be the same.

Note: Although it's not necessary to use `{ }` if the body of `if...else` has only one statement, using `{ }` makes your code more readable.

More on Decision Making

In certain situations, a **ternary operator** can replace an `if...else` statement. To learn more, visit C++ Ternary Operator.

If we need to make a choice between more than one alternatives based on a given test condition, the `switch` statement can be used. To learn more, visit C++ switch.

Check out these examples to learn more:

C++ Program to Check Whether Number is Even or Odd

C++ Program to Check Whether a Character is Vowel or Consonant.

C++ Program to Find Largest Number Among Three Numbers

C++ for Loop

programiz.com/cpp-programming/for-loop

Join our newsletter for the latest updates.

In this tutorial, we will learn about the C++ for loop and its working with the help of some examples.

In computer programming, loops are used to repeat a block of code.

For example, let's say we want to show a message 100 times. Then instead of writing the print statement 100 times, we can use a loop.

That was just a simple example; we can achieve much more efficiency and sophistication in our programs by making effective use of loops.

There are 3 types of loops in C++.

- `for` loop
- `while` loop
- `do...while` loop

This tutorial focuses on C++ `for` loop. We will learn about the other type of loops in the upcoming tutorials.

C++ for loop

The syntax of for-loop is:

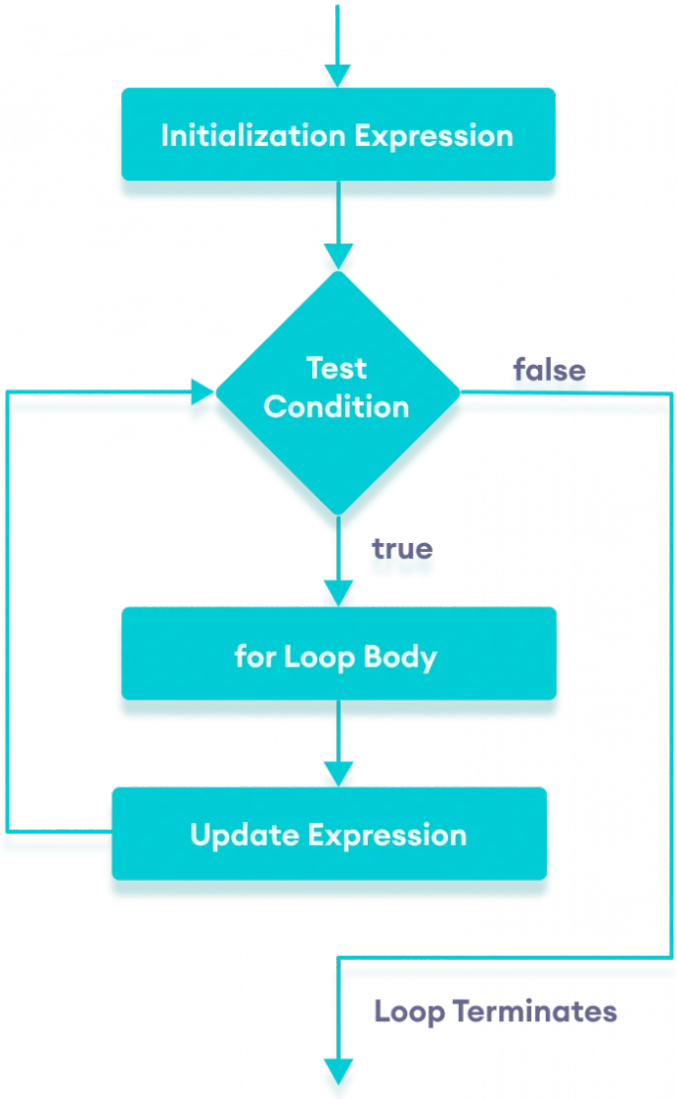
```
for (initialization; condition; update) {
    // body of-loop
}
```

Here,

- `initialization` - initializes variables and is executed only once
- `condition` - if `true`, the body of `for` loop is executed
  if `false`, the for loop is terminated
- `update` - updates the value of initialized variables and again checks the condition

To learn more about `conditions`, check out our tutorial on C++ Relational and Logical Operators.

Flowchart of for Loop in C++



Flowchart of for loop in C++

Example 1: Printing Numbers From 1 to 5

```
#include <iostream>

using namespace std;

int main() {
        for (int i = 1; i <= 5; ++i) {
        cout << i << " ";
    }
    return 0;
}
```

Output

```
1 2 3 4 5
```

Here is how this program works

Iteration	Variable	i <= 5	Action
1st	`i = 1`	`true`	1 is printed. `i` is increased to `2`.
2nd	`i = 2`	`true`	2 is printed. `i` is increased to `3`.
3rd	`i = 3`	`true`	3 is printed. `i` is increased to `4`.
4th	`i = 4`	`true`	4 is printed. `i` is increased to `5`.
5th	`i = 5`	`true`	5 is printed. `i` is increased to `6`.
6th	`i = 6`	`false`	The loop is terminated

Example 2: Display a text 5 times

```
// C++ Program to display a text 5 times

#include <iostream>

using namespace std;

int main() {
    for (int i = 1; i <= 5; ++i) {
        cout <<  "Hello World! " << endl;
    }
    return 0;
}
```

Output

```
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

Here is how this program works

Iteration	Variable	i <= 5	Action
1st	`i = 1`	`true`	`Hello World!` is printed and `i` is increased to `2`.
2nd	`i = 2`	`true`	`Hello World!` is printed and `i` is increased to `3`.
3rd	`i = 3`	`true`	`Hello World!` is printed and `i` is increased to `4`.
4th	`i = 4`	`true`	`Hello World!` is printed and `i` is increased to `5`.
5th	`i = 5`	`true`	`Hello World!` is printed and `i` is increased to `6`.
6th	`i = 6`	`false`	The loop is terminated

Example 3: Find the sum of first n Natural Numbers

```
// C++ program to find the sum of first n natural numbers
// positive integers such as 1,2,3,...n are known as natural numbers

#include <iostream>

using namespace std;

int main() {
    int num, sum;
    sum = 0;

    cout << "Enter a positive integer: ";
    cin >> num;

    for (int i = 1; i <= num; ++i) {
        sum += i;
    }

    cout << "Sum = " << sum << endl;

    return 0;
}
```

Output

```
Enter a positive integer: 10
Sum = 55
```

In the above example, we have two variables *num* and *sum*. The *sum* variable is assigned with *0* and the *num* variable is assigned with the value provided by the user.

Note that we have used a `for` loop.

```
for(int i = 1; i <= num; ++i)
```

Here,

- `int i = 1` : initializes the *i* variable
- `i <= num` : runs the loop as long as *i* is less than or equal to *num*
- `++i` : increases the *i* variable by 1 in each iteration

When *i* becomes `11` , the `condition` is `false` and *sum* will be equal to `0 + 1 + 2 + ... + 10` .

Ranged Based for Loop

In C++11, a new range-based `for` loop was introduced to work with collections such as **arrays** and **vectors**. Its syntax is:

```
for (variable : collection) {
    // body of loop
}
```

Here, for every value in the *collection*, the for loop is executed and the value is assigned to the *variable*.

Example 4: Range Based for Loop

```
#include <iostream>

using namespace std;

int main() {

    int num_array[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    for (int n : num_array) {
        cout << n << " ";
    }

    return 0;
}
```

Output

```
1 2 3 4 5 6 7 8 9 10
```

In the above program, we have declared and initialized an `int` array named *num_array*. It has 10 items.

Here, we have used a range-based `for` loop to access all the items in the array.

C++ Infinite for loop

If the `condition` in a `for` loop is always `true`, it runs forever (until memory is full). For example,

```
// infinite for loop
for(int i = 1; i > 0; i++) {
    // block of code
}
```

In the above program, the `condition` is always `true` which will then run the code for infinite times.

Check out these examples to learn more:

In the next tutorial, we will learn about `while` and `do...while` loop.

C++ while and do...while Loop

programiz.com/cpp-programming/do-while-loop

Join our newsletter for the latest updates.

In this tutorial, we will learn the use of while and do...while loops in C++ programming with the help of some examples.

In computer programming, loops are used to repeat a block of code.

For example, let's say we want to show a message 100 times. Then instead of writing the print statement 100 times, we can use a loop.

That was just a simple example; we can achieve much more efficiency and sophistication in our programs by making effective use of loops.

There are **3** types of loops in C++.

1. `for` loop
2. `while` loop
3. `do...while` loop

In the previous tutorial, we learned about the C++ for loop. Here, we are going to learn about `while` and `do...while` loops.

C++ while Loop

The syntax of the `while` loop is:

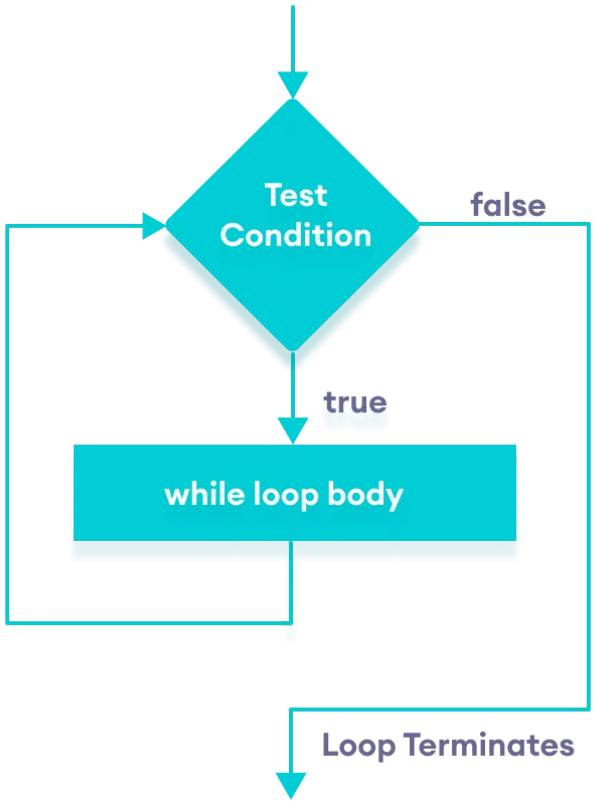
```
while (condition) {
    // body of the loop
}
```

Here,

- A `while` loop evaluates the `condition`
- If the `condition` evaluates to `true`, the code inside the `while` loop is executed.
- The `condition` is evaluated again.
- This process continues until the `condition` is `false`.
- When the `condition` evaluates to `false`, the loop terminates.

To learn more about the `conditions`, visit C++ Relational and Logical Operators.

Flowchart of while Loop



Flowchart of C++ while loop

Example 1: Display Numbers from 1 to 5

```
// C++ Program to print numbers from 1 to 5

#include <iostream>

using namespace std;

int main() {
    int i = 1;

    // while loop from 1 to 5
    while (i <= 5) {
        cout << i << " ";
        ++i;
    }

    return 0;
}
```

Output

```
1 2 3 4 5
```

Here is how the program works.

Iteration	Variable	i <= 5	Action
1st	`i = 1`	`true`	1 is printed and `i` is increased to `2`.
2nd	`i = 2`	`true`	2 is printed and `i` is increased to `3`.
3rd	`i = 3`	`true`	3 is printed and `i` is increased to `4`
4th	`i = 4`	`true`	4 is printed and `i` is increased to `5`.
5th	`i = 5`	`true`	5 is printed and `i` is increased to `6`.
6th	`i = 6`	`false`	The loop is terminated

Example 2: Sum of Positive Numbers Only

```
// program to find the sum of positive numbers
// if the user enters a negative number, the loop ends
// the negative number entered is not added to the sum

#include <iostream>
using namespace std;

int main() {
    int number;
    int sum = 0;

    // take input from the user
    cout << "Enter a number: ";
    cin >> number;

    while (number >= 0) {
        // add all positive numbers
        sum += number;

        // take input again if the number is positive
        cout << "Enter a number: ";
        cin >> number;
    }

    // display the sum
    cout << "\nThe sum is " << sum << endl;

    return 0;
}
```

Output

```
Enter a number: 6
Enter a number: 12
Enter a number: 7
Enter a number: 0
Enter a number: -2

The sum is 25
```

In this program, the user is prompted to enter a number, which is stored in the variable *number*.

In order to store the sum of the numbers, we declare a variable *sum* and initialize it to the value of `0`.

The `while` loop continues until the user enters a negative number. During each iteration, the number entered by the user is added to the *sum* variable.

When the user enters a negative number, the loop terminates. Finally, the total sum is displayed.

C++ do...while Loop

The `do...while` loop is a variant of the `while` loop with one important difference: the body of `do...while` loop is executed once before the `condition` is checked.

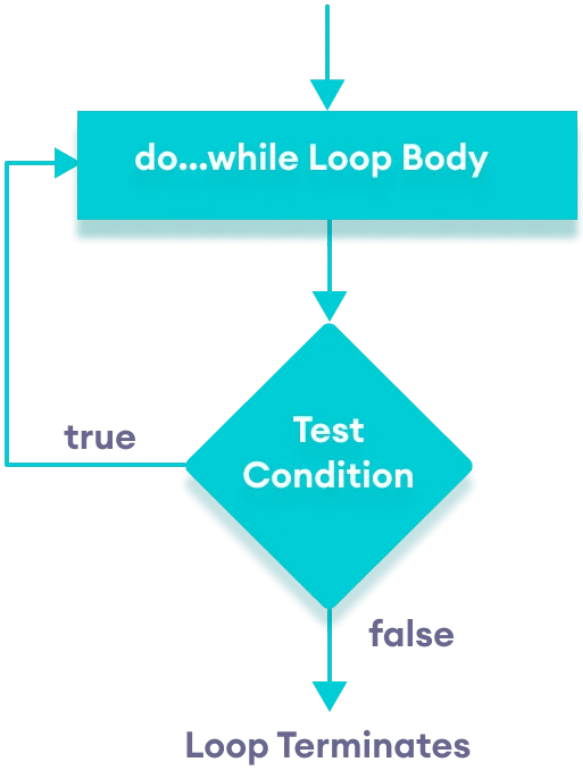
Its syntax is:

```
do {
   // body of loop;
}
while (condition);
```

Here,

- The body of the loop is executed at first. Then the `condition` is evaluated.
- If the `condition` evaluates to `true`, the body of the loop inside the `do` statement is executed again.
- The `condition` is evaluated once again.
- If the `condition` evaluates to `true`, the body of the loop inside the `do` statement is executed again.
- This process continues until the `condition` evaluates to `false`. Then the loop stops.

Flowchart of do...while Loop



Flowchart of C++ do...while loop

Example 3: Display Numbers from 1 to 5

```
// C++ Program to print numbers from 1 to 5

#include <iostream>

using namespace std;

int main() {
    int i = 1;

    // do...while loop from 1 to 5
    do {
        cout << i << " ";
        ++i;
    }
    while (i <= 5);

    return 0;
}
```

Output

```
1 2 3 4 5
```

Here is how the program works.

Iteration	Variable	i <= 5	Action
	`i = 1`	not checked	1 is printed and `i` is increased to 2
1st	`i = 2`	`true`	2 is printed and `i` is increased to 3
2nd	`i = 3`	`true`	3 is printed and `i` is increased to 4
3rd	`i = 4`	`true`	4 is printed and `i` is increased to 5
4th	`i = 5`	`true`	5 is printed and `i` is increased to **6**
5th	`i = 6`	`false`	The loop is terminated

Example 4: Sum of Positive Numbers Only

```
// program to find the sum of positive numbers
// If the user enters a negative number, the loop ends
// the negative number entered is not added to the sum

#include <iostream>
using namespace std;

int main() {
    int number = 0;
    int sum = 0;

    do {
        sum += number;

        // take input from the user
        cout << "Enter a number: ";
        cin >> number;
    }
    while (number >= 0);

    // display the sum
    cout << "\nThe sum is " << sum << endl;

    return 0;
}
```

Output 1

```
Enter a number: 6
Enter a number: 12
Enter a number: 7
Enter a number: 0
Enter a number: -2

The sum is 25
```

Here, the `do...while` loop continues until the user enters a negative number. When the number is negative, the loop terminates; the negative number is not added to the `sum` variable.

Output 2

```
Enter a number: -6
The sum is 0.
```

The body of the `do...while` loop runs only once if the user enters a negative number.

Infinite while loop

If the `condition` of a loop is always `true`, the loop runs for infinite times (until the memory is full). For example,

```
// infinite while loop
while(true) {
    // body of the loop
}
```

Here is an example of an infinite `do...while` loop.

```
// infinite do...while loop

int count = 1;

do {
   // body of loop
}
while(count == 1);
```

In the above programs, the `condition` is always `true`. Hence, the loop body will run for infinite times.

---

for vs while loops

A `for` loop is usually used when the number of iterations is known. For example,

```
// This loop is iterated 5 times
for (int i = 1; i <=5; ++i) {
   // body of the loop
}
```

Here, we know that the for-loop will be executed 5 times.

However, `while` and `do...while` loops are usually used when the number of iterations is unknown. For example,

```
while (condition) {
    // body of the loop
}
```

Check out these examples to learn more:

C++ break Statement

programiz.com/cpp-programming/break-statement

Join our newsletter for the latest updates.

In this tutorial, we will learn about the break statement and its working in loops with the help of examples.

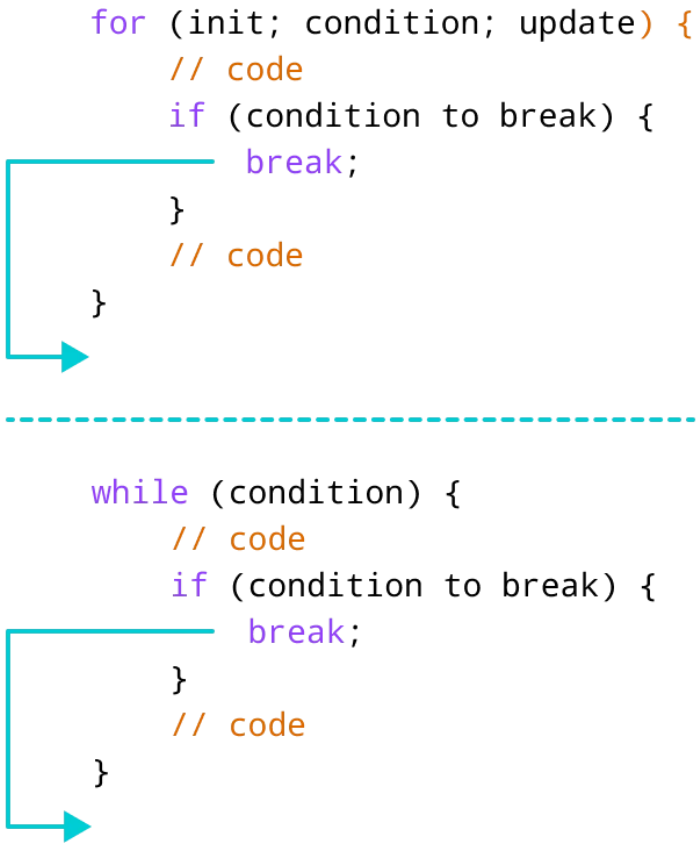
In C++, the `break` statement terminates the loop when it is encountered.

The syntax of the `break` statement is:

```
break;
```

Before you learn about the `break` statement, make sure you know about:

Working of C++ break Statement



Working of break statement in C++

Example 1: break with for loop

```
// program to print the value of i

#include <iostream>
using namespace std;

int main() {
    for (int i = 1; i <= 5; i++) {
        // break condition
        if (i == 3) {
            break;
        }
        cout << i << endl;
    }

return 0;
}
```

Output

```
1
2
```

In the above program, the `for` loop is used to print the value of *i* in each iteration. Here, notice the code:

```
if (i == 3) {
    break;
}
```

This means, when *i* is equal to **3**, the `break` statement terminates the loop. Hence, the output doesn't include values greater than or equal to 3.

Note: The `break` statement is usually used with decision-making statements.

Example 2: break with while loop

```
// program to find the sum of positive numbers
// if the user enters a negative numbers, break ends the loop
// the negative number entered is not added to sum

#include <iostream>
using namespace std;

int main() {
    int number;
    int sum = 0;

    while (true) {
        // take input from the user
        cout << "Enter a number: ";
        cin >> number;

        // break condition
        if (number < 0) {
            break;
        }

        // add all positive numbers
        sum += number;
    }

    // display the sum
    cout << "The sum is " << sum << endl;

    return 0;
}
```

Output

```
Enter a number: 1
Enter a number: 2
Enter a number: 3
Enter a number: -5
The sum is 6.
```

In the above program, the user enters a number. The `while` loop is used to print the total sum of numbers entered by the user. Here, notice the code,

```
if(number < 0) {
    break;
}
```

This means, when the user enters a negative number, the `break` statement terminates the loop and codes outside the loop are executed.

The `while` loop continues until the user enters a negative number.

---

break with Nested loop

When `break` is used with nested loops, `break` terminates the inner loop. For example,

```
// using break statement inside
// nested for loop

#include <iostream>
using namespace std;

int main() {
    int number;
    int sum = 0;

    // nested for loops

    // first loop
    for (int i = 1; i <= 3; i++) {
        // second loop
        for (int j = 1; j <= 3; j++) {
            if (i == 2) {
                break;
            }
            cout << "i = " << i << ", j = " << j << endl;
        }
    }

    return 0;
}
```

Output

```
i = 1, j = 1
i = 1, j = 2
i = 1, j = 3
i = 3, j = 1
i = 3, j = 2
i = 3, j = 3
```

In the above program, the `break` statement is executed when `i == 2`. It terminates the inner loop, and the control flow of the program moves to the outer loop.

Hence, the value of `i = 2` is never displayed in the output.

The `break` statement is also used with the `switch` statement. To learn more, visit C++ switch statement.

C++ continue Statement

programiz.com/cpp-programming/continue-statement

Join our newsletter for the latest updates.

In this tutorial, we will learn about the continue statement and its working with loops with the help of examples.

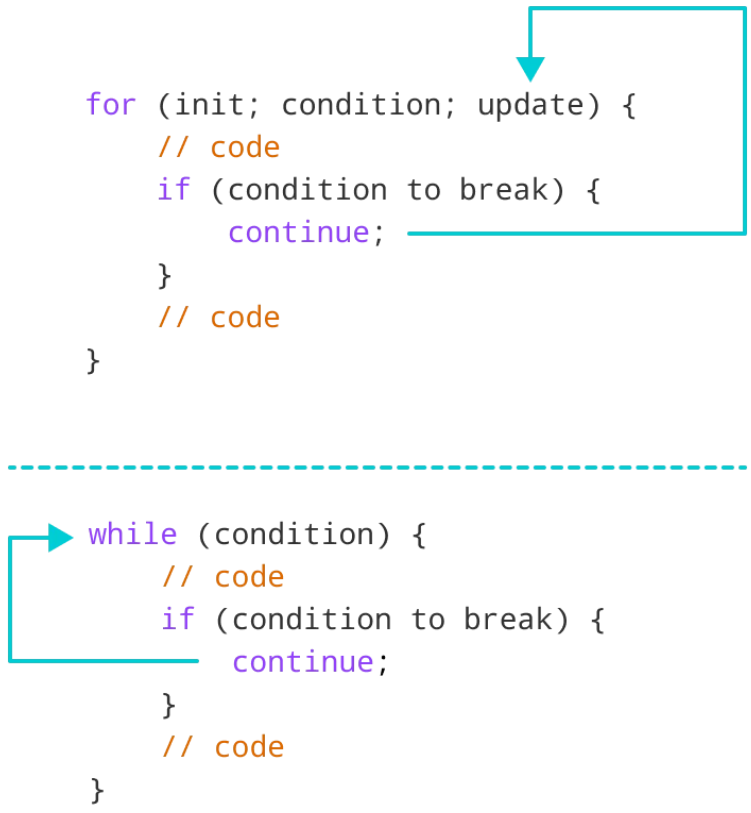
In computer programming, the `continue` statement is used to skip the current iteration of the loop and the control of the program goes to the next iteration.

The syntax of the `continue` statement is:

```
continue;
```

Before you learn about the continue statement, make sure you know about,

Working of C++ continue Statement



Working of continue statement in C++

Example 1: continue with for loop

In a `for` loop, `continue` skips the current iteration and the control flow jumps to the `update` expression.

```
// program to print the value of i

#include <iostream>
using namespace std;

int main() {
    for (int i = 1; i <= 5; i++) {
        // condition to continue
        if (i == 3) {
            continue;
        }

        cout << i << endl;
    }

    return 0;
}
```

Output

```
1
2
4
5
```

In the above program, we have used the the `for` loop to print the value of *i* in each iteration. Here, notice the code,

```
if (i == 3) {
    continue;
}
```

This means

- When *i* is equal to `3`, the `continue` statement skips the current iteration and starts the next iteration
- Then, *i* becomes `4`, and the `condition` is evaluated again.
- Hence, `4` and `5` are printed in the next two iterations.

Note: The `continue` statement is almost always used with decision-making statements.

Example 2: continue with while loop

In a `while` loop, `continue` skips the current iteration and control flow of the program jumps back to the `while` `condition`.

```
// program to calculate positive numbers till 50 only
// if the user enters a negative number,
// that number is skipped from the calculation

// negative number -> loop terminate
// numbers above 50 -> skip iteration

#include <iostream>
using namespace std;

int main() {
    int sum = 0;
    int number = 0;

    while (number >= 0) {
        // add all positive numbers
        sum += number;

        // take input from the user
        cout << "Enter a number: ";
        cin >> number;

        // continue condition
        if (number > 50) {
            cout << "The number is greater than 50 and won't be calculated." <<
endl;
            number = 0;  // the value of number is made 0 again
            continue;
        }
    }

    // display the sum
    cout << "The sum is " << sum << endl;

    return 0;
}
```

Output

```
Enter a number: 12
Enter a number: 0
Enter a number: 2
Enter a number: 30
Enter a number: 50
Enter a number: 56
The number is greater than 50 and won't be calculated.
Enter a number: 5
Enter a number: -3
The sum is 99
```

In the above program, the user enters a number. The `while` loop is used to print the total sum of positive numbers entered by the user, as long as the numbers entered are not greater than `50`.

Notice the use of the `continue` statement.

```
 if (number > 50){
    continue;
}
```

- When the user enters a number greater than `50`, the `continue` statement skips the current iteration. Then the control flow of the program goes to the `condition` of `while` loop.
- When the user enters a number less than `0`, the loop terminates.

Note: The `continue` statement works in the same way for the `do...while` loops.

---

continue with Nested loop

When `continue` is used with nested loops, it skips the current iteration of the inner loop. For example,

```
// using continue statement inside
// nested for loop

#include <iostream>
using namespace std;

int main() {
    int number;
    int sum = 0;

    // nested for loops

    // first loop
    for (int i = 1; i <= 3; i++) {
        // second loop
        for (int j = 1; j <= 3; j++) {
            if (j == 2) {
                continue;
            }
            cout << "i = " << i << ", j = " << j << endl;
        }
    }

    return 0;
}
```

Output

```
i = 1, j = 1
i = 1, j = 3
i = 2, j = 1
i = 2, j = 3
i = 3, j = 1
i = 3, j = 3
```

In the above program, when the `continue` statement executes, it skips the current iteration in the inner loop. And the control of the program moves to the **update expression** of the inner loop.

Hence, the value of `j = 2` is never displayed in the output.

Note: The break statement terminates the loop entirely. However, the `continue` statement only skips the current iteration.

C++ switch..case Statement

programiz.com/cpp-programming/switch-case

Join our newsletter for the latest updates.

In this tutorial, we will learn about switch statement and its working in C++ programming with the help of some examples.

The `switch` statement allows us to execute a block of code among many alternatives.

The syntax of the `switch` statement in C++ is:

```
switch (expression)  {
    case constant1:
        // code to be executed if
        // expression is equal to constant1;
        break;

    case constant2:
        // code to be executed if
        // expression is equal to constant2;
        break;
        .
        .
        .
    default:
        // code to be executed if
        // expression doesn't match any constant
}
```

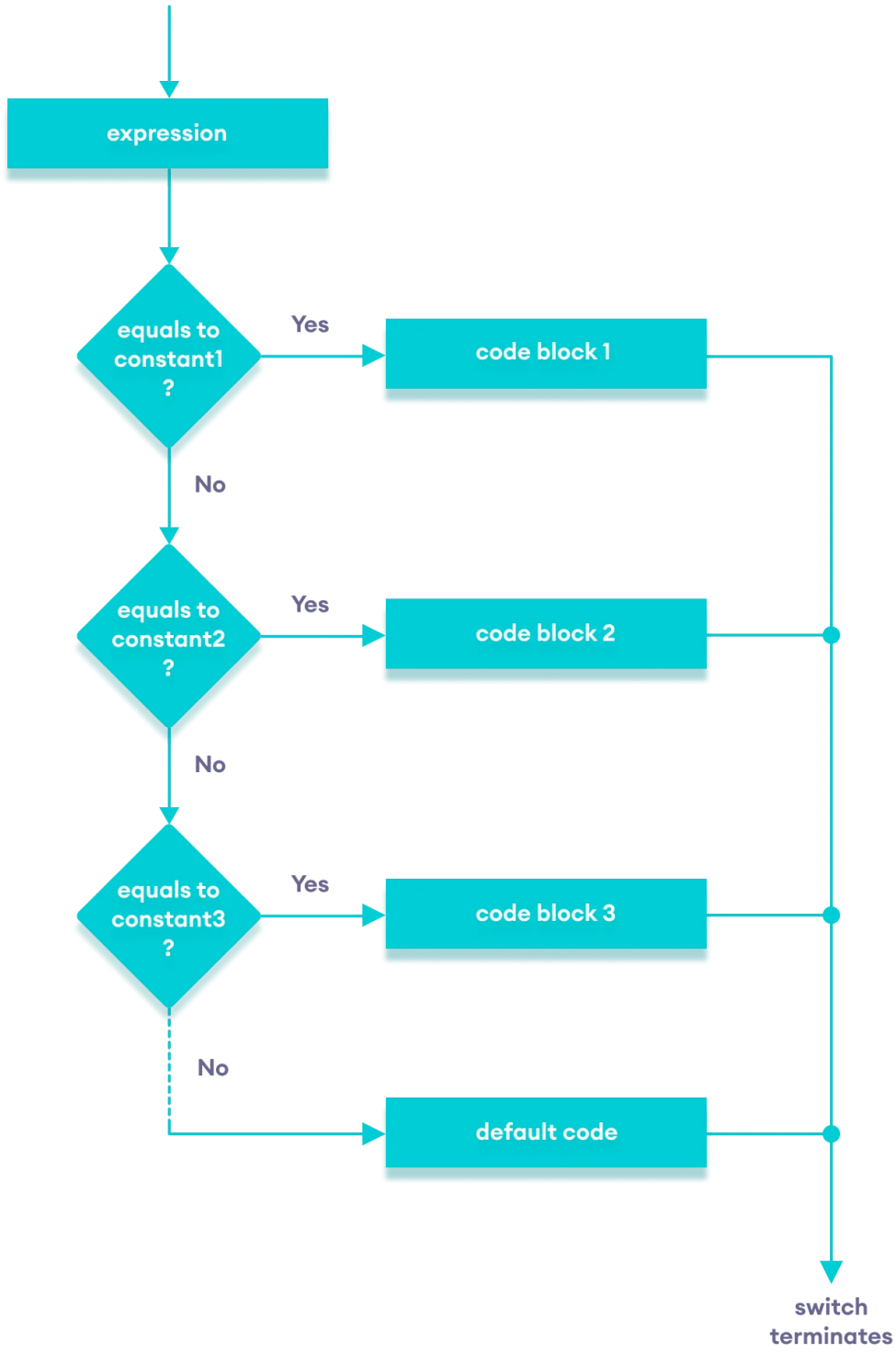
How does the switch statement work?

The `expression` is evaluated once and compared with the values of each `case` label.

- If there is a match, the corresponding code after the matching label is executed. For example, if the value of the variable is equal to `constant2`, the code after `case constant2:` is executed until the break statement is encountered.
- If there is no match, the code after `default:` is executed.

Note: We can do the same thing with the `if...else..if` ladder. However, the syntax of the `switch` statement is cleaner and much easier to read and write.

Flowchart of switch Statement



Flowchart of C++ switch...case statement

Example: Create a Calculator using the switch Statement

```
// Program to build a simple calculator using switch Statement
#include <iostream>
using namespace std;

int main() {
    char oper;
    float num1, num2;
    cout << "Enter an operator (+, -, *, /): ";
    cin >> oper;
    cout << "Enter two numbers: " << endl;
    cin >> num1 >> num2;

    switch (oper) {
        case '+':
            cout << num1 << " + " << num2 << " = " << num1 + num2;
            break;
        case '-':
            cout << num1 << " - " << num2 << " = " << num1 - num2;
            break;
        case '*':
            cout << num1 << " * " << num2 << " = " << num1 * num2;
            break;
        case '/':
            cout << num1 << " / " << num2 << " = " << num1 / num2;
            break;
        default:
            // operator is doesn't match any case constant (+, -, *, /)
            cout << "Error! The operator is not correct";
            break;
    }

    return 0;
}
```

Output 1

```
Enter an operator (+, -, *, /): +
Enter two numbers:
2.3
4.5
2.3 + 4.5 = 6.8
```

Output 2

```
Enter an operator (+, -, *, /): -
Enter two numbers:
2.3
4.5
2.3 - 4.5 = -2.2
```

Output 3

```
Enter an operator (+, -, *, /): *
Enter two numbers:
2.3
4.5
2.3 * 4.5 = 10.35
```

Output 4

```
Enter an operator (+, -, *, /): /
Enter two numbers:
2.3
4.5
2.3 / 4.5 = 0.511111
```

Output 5

```
Enter an operator (+, -, *, /): ?
Enter two numbers:
2.3
4.5
Error! The operator is not correct.
```

In the above program, we are using the `switch...case` statement to perform addition, subtraction, multiplication, and division.

How This Program Works

1. We first prompt the user to enter the desired operator. This input is then stored in the `char` variable named *oper*.
2. We then prompt the user to enter two numbers, which are stored in the float variables *num1* and *num2*.
3. The `switch` statement is then used to check the operator entered by the user:
    - If the user enters `+`, addition is performed on the numbers.
    - If the user enters `-`, subtraction is performed on the numbers.
    - If the user enters `*`, multiplication is performed on the numbers.
    - If the user enters `/`, division is performed on the numbers.
    - If the user enters any other character, the default code is printed.

Notice that the `break` statement is used inside each `case` block. This terminates the `switch` statement.

If the `break` statement is not used, all cases after the correct `case` are executed.

C++ goto Statement

programiz.com/cpp-programming/goto

Join our newsletter for the latest updates.

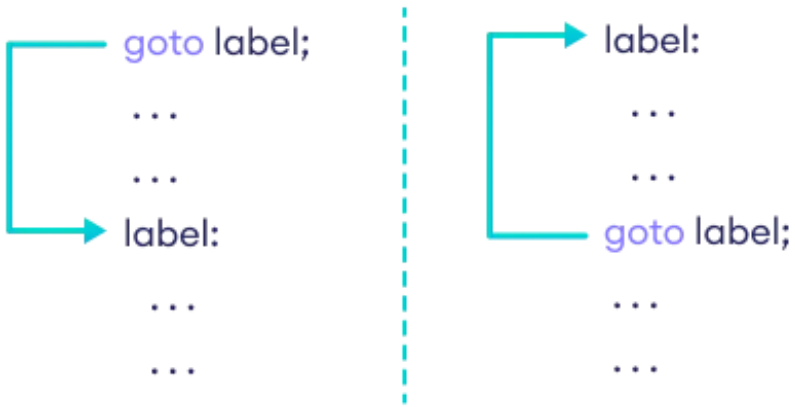
In this article, you'll learn about goto statment, how it works and why should it be avoided.

In C++ programming, the goto statement is used for altering the normal sequence of program execution by transferring control to some other part of the program.

Syntax of goto Statement

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

In the syntax above, *label* is an identifier. When `goto label;` is encountered, the control of program jumps to `label:` and executes the code below it.



Working of goto in C++

Example: goto Statement

```
// This program calculates the average of numbers entered by the user.
// If the user enters a negative number, it ignores the number and
// calculates the average number entered before it.

# include <iostream>
using namespace std;

int main()
{
    float num, average, sum = 0.0;
    int i, n;

    cout << "Maximum number of inputs: ";
    cin >> n;

    for(i = 1; i <= n; ++i)
    {
        cout << "Enter n" << i << ": ";
        cin >> num;

        if(num < 0.0)
        {
           // Control of the program move to jump:
            goto jump;
        }
        sum += num;
    }

jump:
    average = sum / (i - 1);
    cout << "\nAverage = " << average;
    return 0;
}
```

Output

```
Maximum number of inputs: 10
Enter n1: 2.3
Enter n2: 5.6
Enter n3: -5.6

Average = 3.95
```

You can write any C++ program without the use of `goto` statement and is generally considered a good idea not to use them.

Reason to Avoid goto Statement

The goto statement gives the power to jump to any part of a program but, makes the logic of the program complex and tangled.

In modern programming, the goto statement is considered a harmful construct and a bad programming practice.

The goto statement can be replaced in most of C++ program with the use of break and continue statements.

C++ Functions

programiz.com/cpp-programming/function

Join our newsletter for the latest updates.

In this tutorial, we will learn about the C++ function and function expressions with the help of examples.

A function is a block of code that performs a specific task.

Suppose we need to create a program to create a circle and color it. We can create two functions to solve this problem:

- a function to draw the circle
- a function to color the circle

Dividing a complex problem into smaller chunks makes our program easy to understand and reusable.

There are two types of function:

1. **Standard Library Functions:** Predefined in C++
2. **User-defined Function:** Created by users

In this tutorial, we will focus mostly on user-defined functions.

C++ User-defined Function

C++ allows the programmer to define their own function.

A user-defined function groups code to perform a specific task and that group of code is given a name (identifier).

When the function is invoked from any part of the program, it all executes the codes defined in the body of the function.

C++ Function Declaration

The syntax to declare a function is:

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

Here's an example of a function declaration.

```
// function declaration
void greet() {
    cout << "Hello World";
}
```

Here,

- the name of the function is `greet()`
- the return type of the function is `void`
- the empty parentheses mean it doesn't have any parameters
- the function body is written inside `{}`

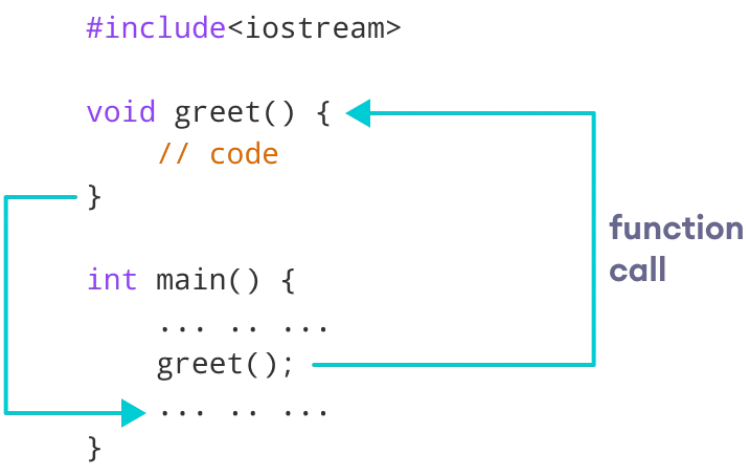
Note: We will learn about `returnType` and `parameters` later in this tutorial.

Calling a Function

In the above program, we have declared a function named `greet()`. To use the `greet()` function, we need to call it.

Here's how we can call the above `greet()` function.

```
int main() {

    // calling a function
    greet();

}
```



How Function works in C++

Example 1: Display a Text

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

// declaring a function
void greet() {
    cout << "Hello there!";
}

int main() {

    // calling the function
    greet();

    return 0;
}
```

Output

```
Hello there!
```

Function Parameters

As mentioned above, a function can be declared with parameters (arguments). A parameter is a value that is passed when declaring a function.

For example, let us consider the function below:

```
void printNum(int num) {
    cout << num;
}
```

Here, the `int` variable *num* is the function parameter.

We pass a value to the function parameter while calling the function.

```
int main() {
    int n = 7;

    // calling the function
    // n is passed to the function as argument
    printNum(n);

    return 0;
}
```

Example 2: Function with Parameters

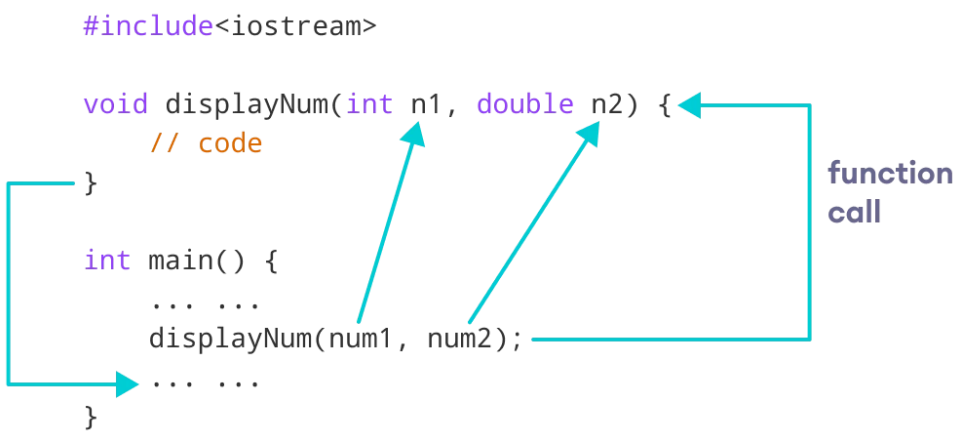
```
// program to print a text

#include <iostream>
using namespace std;

// display a number
void displayNum(int n1, float n2) {
    cout << "The int number is " << n1;
    cout << "The double number is " << n2;
}

int main() {

     int num1 = 5;
     double num2 = 5.5;

    // calling the function
    displayNum(num1, num2);

    return 0;
}
```

Output

```
The int number is 5
The double number is 5.5
```

In the above program, we have used a function that has one `int` parameter and one `double` parameter.

We then pass *num1* and *num2* as arguments. These values are stored by the function parameters *n1* and *n2* respectively.



C++ function with parameters

Note: The type of the arguments passed while calling the function must match with the corresponding parameters defined in the function declaration.

Return Statement

In the above programs, we have used void in the function declaration. For example,

```
void displayNumber() {
    // code
}
```

This means the function is not returning any value.

It's also possible to return a value from a function. For this, we need to specify the `returnType` of the function during function declaration.

Then, the `return` statement can be used to return a value from a function.

For example,

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

Here, we have the data type `int` instead of `void`. This means that the function returns an `int` value.

The code `return (a + b);` returns the sum of the two parameters as the function value.

The `return` statement denotes that the function has ended. Any code after `return` inside the function is not executed.

Example 3: Add Two Numbers

```
// program to add two numbers using a function

#include <iostream>

using namespace std;

// declaring a function
int add(int a, int b) {
    return (a + b);
}

int main() {

    int sum;

    // calling the function and storing
    // the returned value in sum
    sum = add(100, 78);

    cout << "100 + 78 = " << sum << endl;

    return 0;
}
```

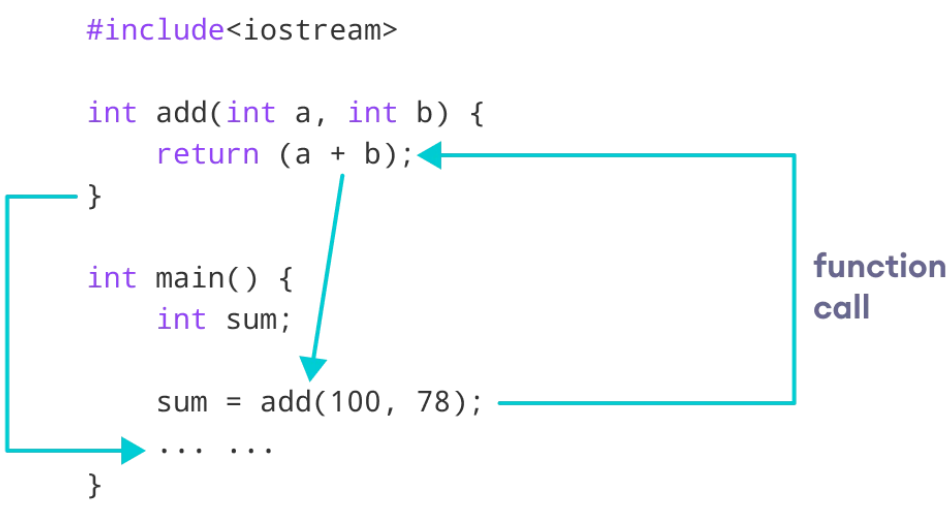
Output

```
100 + 78 = 178
```

In the above program, the `add()` function is used to find the sum of two numbers.

We pass two `int` literals `100` and `78` while calling the function.

We store the returned value of the function in the variable *sum*, and then we print it.



Working of C++ Function with return statement

Notice that *sum* is a variable of `int` type. This is because the return value of `add()` is of `int` type.

Function Prototype

In C++, the code of function declaration should be before the function call. However, if we want to define a function after the function call, we need to use the function prototype. For example,

```
// function prototype
void add(int, int);

int main() {
    // calling the function before declaration.
    add(5, 3);
    return 0;
}

// function definition
void add(int a, int b) {
    cout << (a + b);
}
```

In the above code, the function prototype is:

```
void add(int, int);
```

This provides the compiler with information about the function name and its parameters. That's why we can use the code to call a function before the function has been defined.

The syntax of a function prototype is:

```
returnType functionName(dataType1, dataType2, ...);
```

Example 4: C++ Function Prototype

```
// using function definition after main() function
// function prototype is declared before main()

#include <iostream>

using namespace std;

// function prototype
int add(int, int);

int main() {
    int sum;

    // calling the function and storing
    // the returned value in sum
    sum = add(100, 78);

    cout << "100 + 78 = " << sum << endl;

    return 0;
}

// function definition
int add(int a, int b) {
    return (a + b);
}
```

Output

```
100 + 78 = 178
```

The above program is nearly identical to **Example 3**. The only difference is that here, the function is defined **after** the function call.

That's why we have used a function prototype in this example.

Benefits of Using User-Defined Functions

- Functions make the code reusable. We can declare them once and use them multiple times.
- Functions make the program easier as each small task is divided into a function.
- Functions increase readability.

C++ Library Functions

Library functions are the built-in functions in C++ programming.

Programmers can use library functions by invoking the functions directly; they don't need to write the functions themselves.

Some common library functions in C++ are `sqrt()`, `abs()`, `isdigit()`, etc.

In order to use library functions, we usually need to include the header file in which these library functions are defined.

For instance, in order to use mathematical functions such as `sqrt()` and `abs()`, we need to include the header file `cmath`.

Example 5: C++ Program to Find the Square Root of a Number

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

int main() {
    double number, squareRoot;

    number = 25.0;

    // sqrt() is a library function to calculate the square root
    squareRoot = sqrt(number);

    cout << "Square root of " << number << " = " << squareRoot;

    return 0;
}
```

Output

```
Square root of 25 = 5
```

In this program, the `sqrt()` library function is used to calculate the square root of a number.

The function declaration of `sqrt()` is defined in the `cmath` header file. That's why we need to use the code `#include <cmath>` to use the `sqrt()` function.

To learn more, visit C++ Standard Library functions.

Types of User-defined Functions in C++

In this tutorial, you will learn about different approaches you can take to solve a single problem using functions.

For better understanding of arguments and return in functions, user-defined functions can be categorised as:

Consider a situation in which you have to check prime number. This problem is solved below by making user-defined function in 4 different ways as mentioned above.

Example 1: No arguments passed and no return value

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

void prime();

int main()
{
    // No argument is passed to prime()
    prime();
    return 0;
}


// Return type of function is void because value is not returned.
void prime()
{

    int num, i, flag = 0;

    cout << "Enter a positive integer enter to check: ";
    cin >> num;

    for(i = 2; i <= num/2; ++i)
    {
        if(num % i == 0)
        {
            flag = 1;
            break;
        }
    }

    if (flag == 1)
    {
        cout << num << " is not a prime number.";
    }
    else
    {
        cout << num << " is a prime number.";
    }
}
```

In the above program, `prime()` is called from the main() with no arguments.

`prime()` takes the positive number from the user and checks whether the number is a prime number or not.

Since, return type of `prime()` is `void`, no value is returned from the function.

Example 2: No arguments passed but a return value

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

int prime();

int main()
{
    int num, i, flag = 0;

    // No argument is passed to prime()
    num = prime();
    for (i = 2; i <= num/2; ++i)
    {
        if (num%i == 0)
        {
            flag = 1;
            break;
        }
    }

    if (flag == 1)
    {
        cout<<num<<" is not a prime number.";
    }
    else
    {
        cout<<num<<" is a prime number.";
    }
    return 0;
}

// Return type of function is int
int prime()
{
    int n;

    printf("Enter a positive integer to check: ");
    cin >> n;

    return n;
}
```

In the above program, `prime()` function is called from the `main()` with no arguments.

`prime()` takes a positive integer from the user. Since, return type of the function is an `int`, it returns the inputted number from the user back to the calling `main()` function.

Then, whether the number is prime or not is checked in the main() itself and printed onto the screen.

Example 3: Arguments passed but no return value

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

void prime(int n);

int main()
{
    int num;
    cout << "Enter a positive integer to check: ";
    cin >> num;

    // Argument num is passed to the function prime()
    prime(num);
    return 0;
}

// There is no return value to calling function. Hence, return type of function is
void. */
void prime(int n)
{
    int i, flag = 0;
    for (i = 2; i <= n/2; ++i)
    {
        if (n%i == 0)
        {
            flag = 1;
            break;
        }
    }

    if (flag == 1)
    {
        cout << n << " is not a prime number.";
    }
    else {
        cout << n << " is a prime number.";
    }
}
```

In the above program, positive number is first asked from the user which is stored in the variable *num*.

Then, *num* is passed to the `prime()` function where, whether the number is prime or not is checked and printed.

Since, the return type of `prime()` is a `void`, no value is returned from the function.

Example 4: Arguments passed and a return value.

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

int prime(int n);

int main()
{
    int num, flag = 0;
    cout << "Enter positive integer to check: ";
    cin >> num;

    // Argument num is passed to check() function
    flag = prime(num);

    if(flag == 1)
        cout << num << " is not a prime number.";
    else
        cout<< num << " is a prime number.";
    return 0;
}

/* This function returns integer value.  */
int prime(int n)
{
    int i;
    for(i = 2; i <= n/2; ++i)
    {
        if(n % i == 0)
            return 1;
    }

    return 0;
}
```

In the above program, a positive integer is asked from the user and stored in the variable `num`.

Then, `num` is passed to the function `prime()` where, whether the number is prime or not is checked.

Since, the return type of `prime()` is an `int`, 1 or 0 is returned to the `main()` calling function. If the number is a prime number, 1 is returned. If not, 0 is returned.

Back in the `main()` function, the returned 1 or 0 is stored in the variable *flag*, and the corresponding text is printed onto the screen.

Which method is better?

All four programs above gives the same output and all are technically correct program.

There is no hard and fast rule on which method should be chosen.

The particular method is chosen depending upon the situation and how you want to solve a problem.

C++ Function Overloading

In this tutorial, we will learn about the function overloading in C++ with examples.

In C++, two functions can have the same name if the number and/or type of arguments passed is different.

These functions having the same name but different arguments are known as overloaded functions. For example:

```
// same name different arguments
int test() { }
int test(int a) { }
float test(double a) { }
int test(int a, double b) { }
```

Here, all 4 functions are overloaded functions.

Notice that the return types of all these 4 functions are not the same. Overloaded functions may or may not have different return types but they must have different arguments. For example,

```
// Error code
int test(int a) { }
double test(int b){ }
```

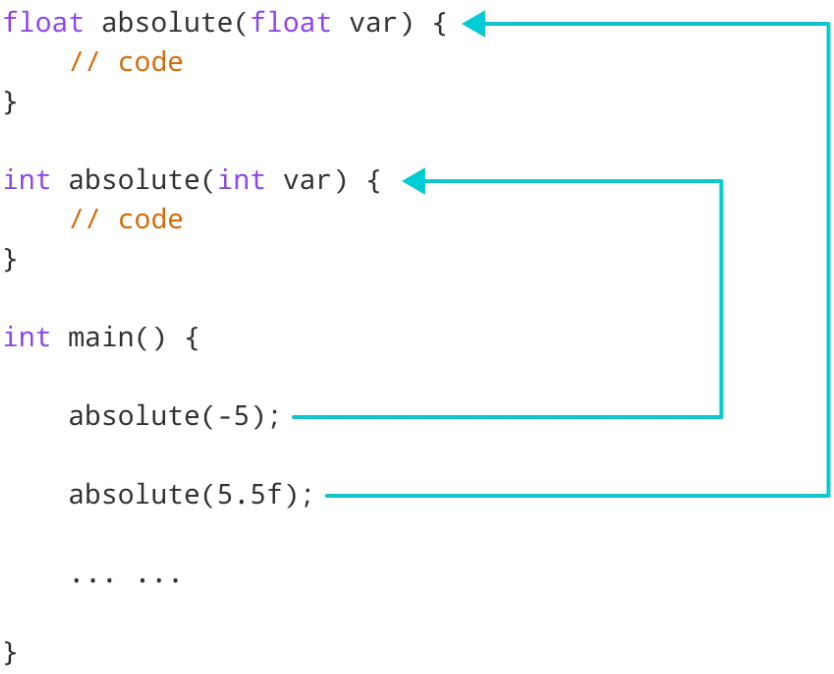
Here, both functions have the same name, the same type, and the same number of arguments. Hence, the compiler will throw an error.

Example 1: Overloading Using Different Types of Parameter

```
// Program to compute absolute value
// Works for both int and float

#include <iostream>
using namespace std;

// function with float type parameter
float absolute(float var){
    if (var < 0.0)
        var = -var;
    return var;
}

// function with int type parameter
int absolute(int var) {
     if (var < 0)
         var = -var;
    return var;
}

int main() {

    // call function with int type parameter
    cout << "Absolute value of -5 = " << absolute(-5) << endl;

    // call function with float type parameter
    cout << "Absolute value of 5.5 = " << absolute(5.5f) << endl;
    return 0;
}
```

Output

```
Absolute value of -5 = 5
Absolute value of 5.5 = 5.5
```



Working of overloading for the absolute() function

In this program, we overload the `absolute()` function. Based on the type of parameter passed during the function call, the corresponding function is called.

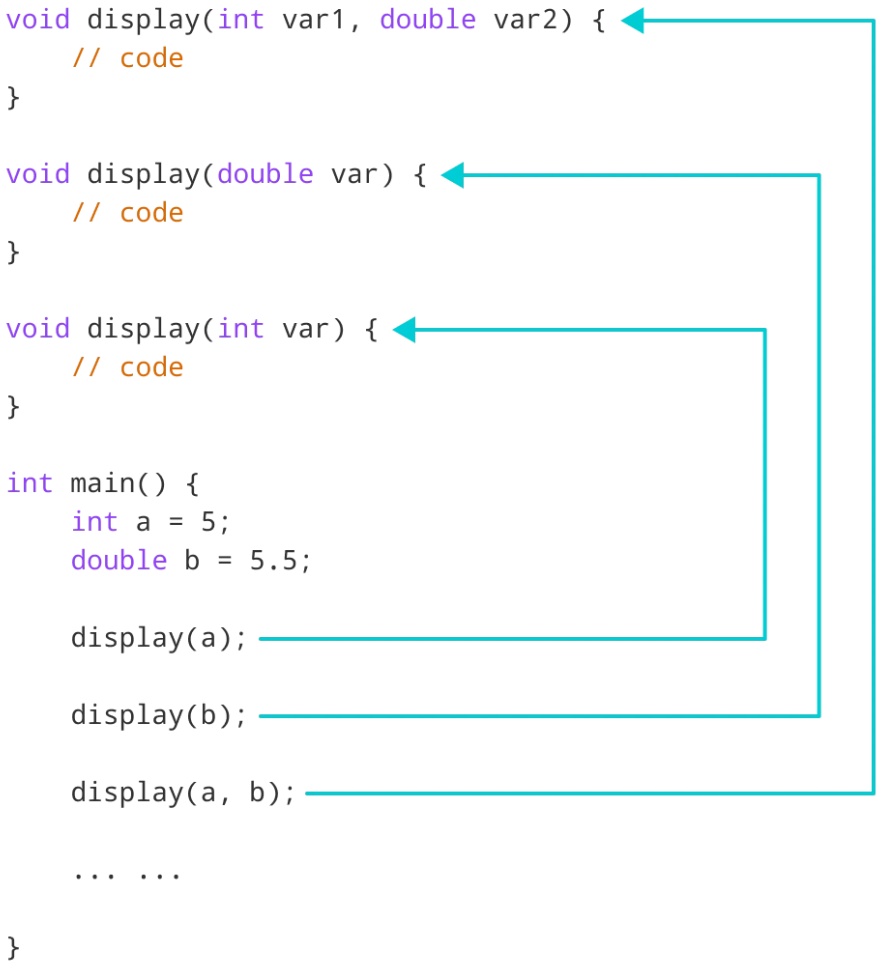
Example 2: Overloading Using Different Number of Parameters

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

// function with 2 parameters
void display(int var1, double var2) {
    cout << "Integer number: " << var1;
    cout << " and double number: " << var2 << endl;
}

// function with double type single parameter
void display(double var) {
    cout << "Double number: " << var << endl;
}

// function with int type single parameter
void display(int var) {
    cout << "Integer number: " << var << endl;
}

int main() {

    int a = 5;
    double b = 5.5;

    // call function with int type parameter
    display(a);

    // call function with double type parameter
    display(b);

    // call function with 2 parameters
    display(a, b);

    return 0;
}
```

Output

```
Integer number: 5
Float number: 5.5
Integer number: 5 and double number: 5.5
```

Here, the `display()` function is called three times with different arguments. Depending on the number and type of arguments passed, the corresponding `display()` function is called.



Working of overloading for the display() function

The return type of all these functions is the same but that need not be the case for function overloading.

Note: In C++, many standard library functions are overloaded. For example, the `sqrt()` function can take `double`, `float`, `int,` etc. as parameters. This is possible because the `sqrt()` function is overloaded in C++.

C++ Programming Default Arguments (Parameters)

Join our newsletter for the latest updates.

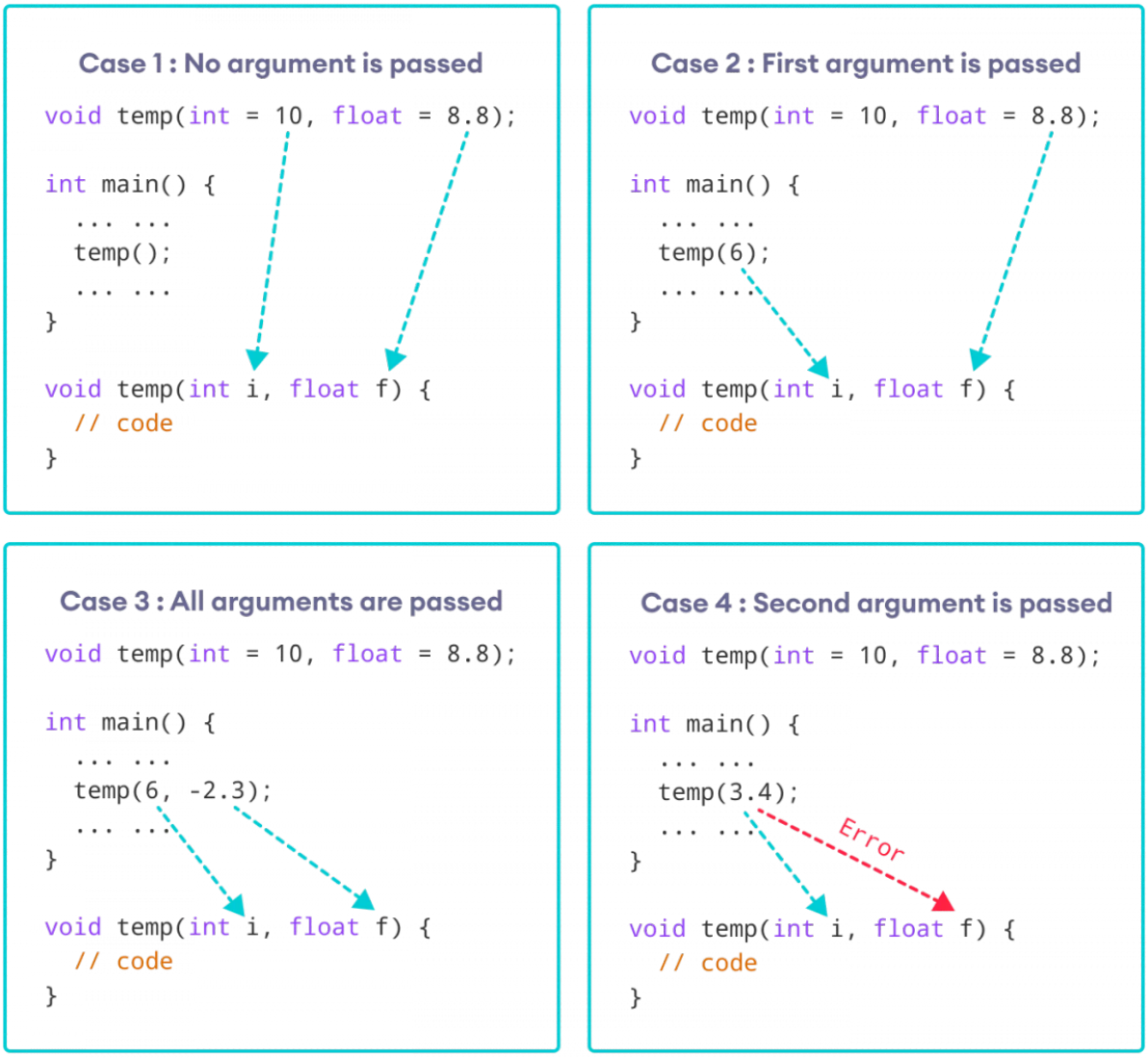
In this tutorial, we will learn C++ default arguments and their working with the help of examples.

In C++ programming, we can provide default values for function parameters.

If a function with default arguments is called without passing arguments, then the default parameters are used.

However, if arguments are passed while calling the function, the default arguments are ignored.

Working of default arguments



How default arguments work in C++

We can understand the working of default arguments from the image above:

1. When `temp()` is called, both the default parameters are used by the function.
2. When `temp(6)` is called, the first argument becomes `6` while the default value is used for the second parameter.
3. When `temp(6, -2.3)` is called, both the default parameters are overridden, resulting in `i = 6` and `f = -2.3`.
4. When `temp(3.4)` is passed, the function behaves in an undesired way because the second argument cannot be passed without passing the first argument.

   Therefore, `3.4` is passed as the first argument. Since the first argument has been defined as `int`, the value that is actually passed is `3`.

Example: Default Argument

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

// defining the default arguments
void display(char = '*', int = 3);

int main() {
    int count = 5;

    cout << "No argument passed: ";
    // *, 3 will be parameters
    display();

    cout << "First argument passed: ";
     // #, 3 will be parameters
    display('#');

    cout << "Both arguments passed: ";
    // $, 5 will be parameters
    display('$', count);

    return 0;
}

void display(char c, int count) {
    for(int i = 1; i <= count; ++i)
    {
        cout << c;
    }
    cout << endl;
}
```

Output

```
No argument passed: ***
First argument passed: ###
Both arguments passed: $$$$$
```

Here is how this program works:

1. `display()` is called without passing any arguments. In this case, `display()` uses both the default parameters `c = '*'` and `n = 1`.
2. `display('#')` is called with only one argument. In this case, the first becomes `'#'`. The second default parameter `n = 1` is retained.
3. `display('#', count)` is called with both arguments. In this case, default arguments are not used.

We can also define the default parameters in the function definition itself. The program below is equivalent to the one above.

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

// defining the default arguments
void display(char c = '*', int count = 3) {
    for(int i = 1; i <= count; ++i) {
        cout << c;
    }
    cout << endl;
}

int main() {
    int count = 5;

    cout << "No argument passed: ";
    // *, 3 will be parameters
    display();

    cout << "First argument passed: ";
     // #, 3 will be parameters
    display('#');

    cout << "Both argument passed: ";
    // $, 5 will be parameters
    display('$', count);

    return 0;
}
```

Things to Remember

1. Once we provide a default value for a parameter, all subsequent parameters must also have default values. For example,

   ```
   // Invalid
   void add(int a, int b = 3, int c, int d);

   // Invalid
   void add(int a, int b = 3, int c, int d = 4);

   // Valid
   void add(int a, int c, int b = 3, int d = 4);
   ```

2. If we are defining the default arguments in the function definition instead of the function prototype, then the function must be defined before the function call.

   ```
   // Invalid code

   int main() {
       // function call
       display();
   }

   void display(char c = '*', int count = 5) {
       // code
   }
   ```

C++ Storage Class

programiz.com/cpp-programming/storage-class

Join our newsletter for the latest updates.

In this article, you'll learn about different storage classes in C++. Namely: local, global, static local, register and thread local.

Every variable in C++ has two features: type and storage class.

Type specifies the type of data that can be stored in a variable. For example: `int`, `float`, `char` etc.

And, storage class controls two different properties of a variable: lifetime (determines how long a variable can exist) and scope (determines which part of the program can access it).

Depending upon the storage class of a variable, it can be divided into 4 major types:

Local Variable

A variable defined inside a function (defined inside function body between braces) is called a local variable or automatic variable.

Its scope is only limited to the function where it is defined. In simple terms, local variable exists and can be accessed only inside a function.

The life of a local variable ends (It is destroyed) when the function exits.

Example 1: Local variable

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

void test();

int main()
{
    // local variable to main()
    int var = 5;

    test();

    // illegal: var1 not declared inside main()
    var1 = 9;
}

void test()
{
    // local variable to test()
    int var1;
    var1 = 6;

    // illegal: var not declared inside test()
    cout << var;
}
```

The variable *var* cannot be used inside `test()` and *var1* cannot be used inside `main()` function.

Keyword `auto` was also used for defining local variables before as: `auto int var;`

But, after C++11 `auto` has a different meaning and should not be used for defining local variables.

Global Variable

If a variable is defined outside all functions, then it is called a global variable.

The scope of a global variable is the whole program. This means, It can be used and changed at any part of the program after its declaration.

Likewise, its life ends only when the program ends.

Example 2: Global variable

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

// Global variable declaration
int c = 12;

void test();

int main()
{
    ++c;

    // Outputs 13
    cout << c <<endl;
    test();

    return 0;
}

void test()
{
    ++c;

    // Outputs 14
    cout << c;
}
```

Output

```
13
14
```

In the above program, *c* is a global variable.

This variable is visible to both functions `main()` and `test()` in the above program.

Static Local variable

Keyword `static` is used for specifying a static variable. For example:

```
... .. ...
int main()
{
   static float a;
   ... .. ...
}
```

A static local variable exists only inside a function where it is declared (similar to a local variable) but its lifetime starts when the function is called and ends only when the program ends.

The main difference between local variable and static variable is that, the value of static variable persists the end of the program.

Example 3: Static local variable

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

void test()
{
    // var is a static variable
    static int var = 0;
    ++var;

    cout << var << endl;
}

int main()
{

    test();
    test();

    return 0;
}
```

Output

```
1
2
```

In the above program, `test()` function is invoked 2 times.

During the first call, variable *var* is declared as static variable and initialized to 0. Then 1 is added to *var* which is displayed in the screen.

When the function test() returns, variable *var* still exists because it is a static variable.

During second function call, no new variable *var* is created. The same *var* is increased by 1 and then displayed to the screen.

Output of above program if `var` was not specified as static variable

```
1
1
```

Register Variable (Deprecated in C++11)

Keyword `register` is used for specifying register variables.

Register variables are similar to automatic variables and exists inside a particular function only. It is supposed to be faster than the local variables.

If a program encounters a register variable, it stores the variable in processor's register rather than memory if available. This makes it faster than the local variables.

However, this keyword was deprecated in C++11 and should not be used.

Thread Local Storage

Thread-local storage is a mechanism by which variables are allocated such that there is one instance of the variable per extant thread.

Keyword `thread_local` is used for this purpose.

Learn more about thread local storage.

C++ Recursion

programiz.com/cpp-programming/recursion

Join our newsletter for the latest updates.

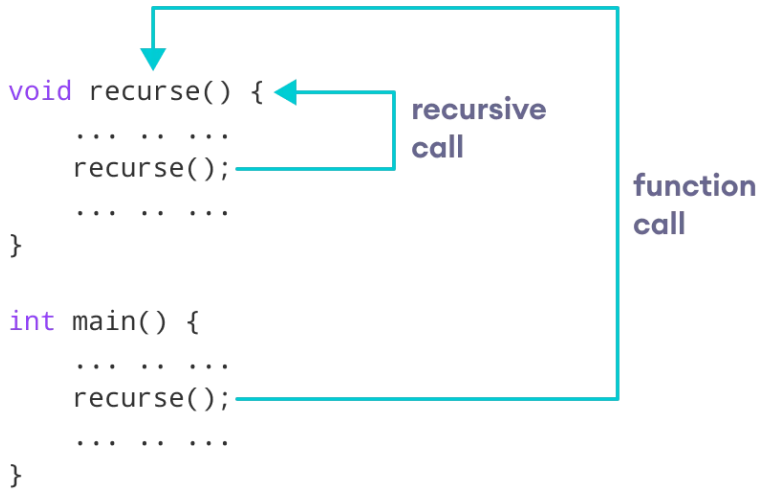
In this tutorial, we will learn about recursive function in C++ and its working with the help of examples.

A function that calls itself is known as a recursive function. And, this technique is known as recursion.

Working of Recursion in C++

```
void recurse()
{
    ... .. ...
    recurse();
    ... .. ...
}

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

The figure below shows how recursion works by calling itself over and over again.



How recursion works in C++ programming

The recursion continues until some condition is met.

To prevent infinite recursion, if...else statement (or similar approach) can be used where one branch makes the recursive call and the other doesn't.

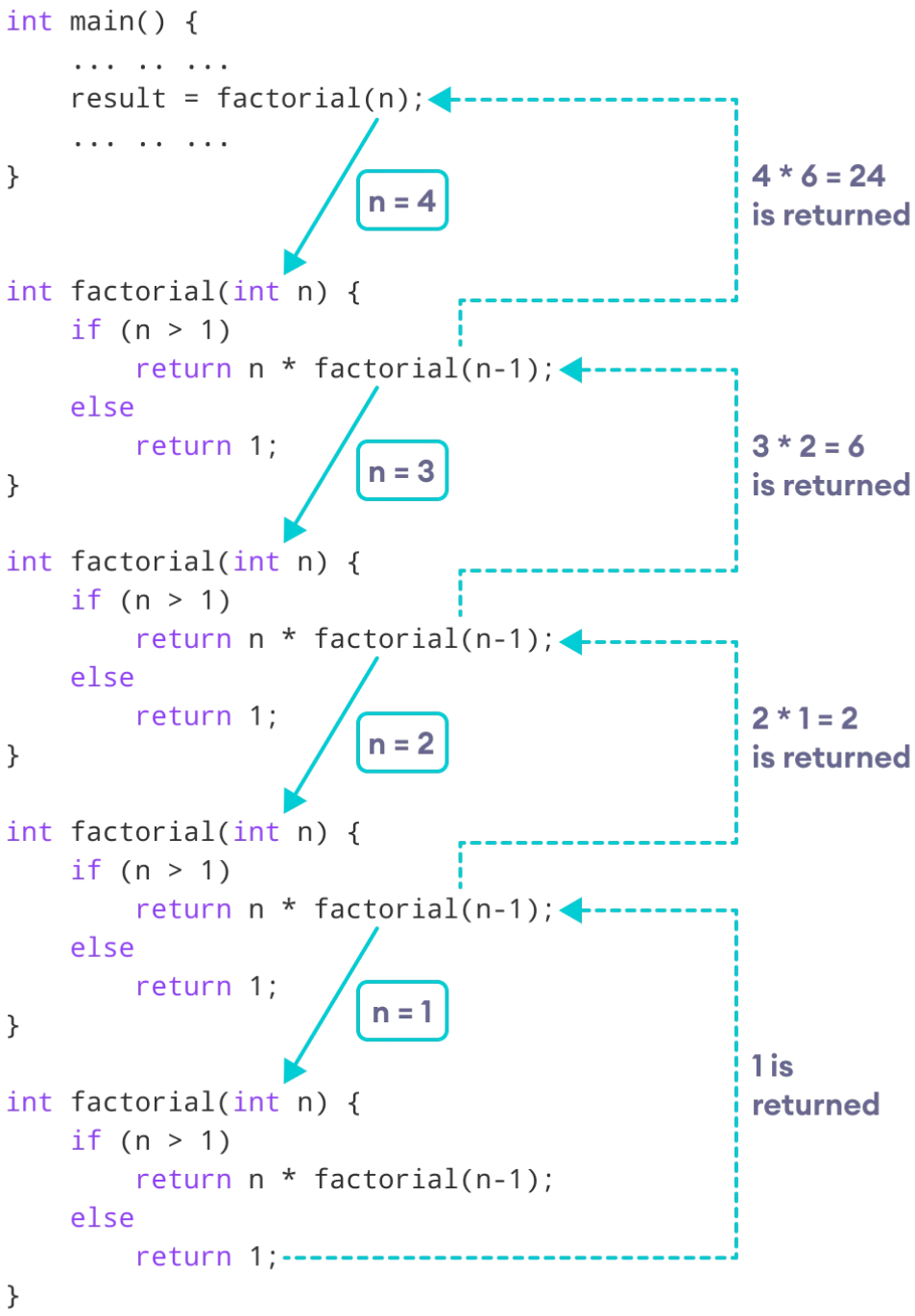
Example 1: Factorial of a Number Using Recursion

```
// Factorial of n = 1*2*3*...*n

#include <iostream>
using namespace std;

int factorial(int);

int main() {
    int n, result;

    cout << "Enter a non-negative number: ";
    cin >> n;

    result = factorial(n);
    cout << "Factorial of " << n << " = " << result;
    return 0;
}

int factorial(int n) {
    if (n > 1) {
        return n * factorial(n - 1);
    } else {
        return 1;
    }
}
```

Output

```
Enter a non-negative number: 4
Factorial of 4 = 24
```

Working of Factorial Program



How this C++ recursion program works

As we can see, the `factorial()` function is calling itself. However, during each call, we have decreased the value of *n* by `1`. When *n* is less than `1`, the `factorial()` function ultimately returns the output.

Advantages and Disadvantages of Recursion

Below are the pros and cons of using recursion in C++.

Advantages of C++ Recursion

- It makes our code shorter and cleaner.
- Recursion is required in problems concerning data structures and advanced algorithms, such as Graph and Tree Traversal.

Disadvantages of C++ Recursion

- It takes a lot of stack space compared to an iterative program.
- It uses more processor time.
- It can be more difficult to debug compared to an equivalent iterative program.

C++ Return by Reference

programiz.com/cpp-programming/return-reference

In this article, you'll learn how to return a value by reference in a function and use it efficiently in your program.

In C++ Programming, not only can you pass values by reference to a function but you can also return a value by reference.

To understand this feature, you should have the knowledge of:

- Global variables

Example: Return by Reference

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

// Global variable
int num;

// Function declaration
int& test();

int main()
{
    test() = 5;

    cout << num;

    return 0;
}

int& test()
{
    return num;
}
```

Output

```
5
```

In program above, the return type of function `test()` is `int&`. Hence, this function returns a reference of the variable *num*.

The return statement is `return num;`. Unlike return by value, this statement doesn't return value of *num*, instead it returns the variable itself (address).

So, when the **variable** is returned, it can be assigned a value as done in `test() = 5;`

This stores 5 to the variable *num*, which is displayed onto the screen.

Important Things to Remember When Returning by Reference.

- Ordinary function returns value but this function doesn't. Hence, you cannot return a constant from the function.

```
int& test() {
    return 2;
}
```

- You cannot return a local variable from this function.

```
int& test()
{
    int n = 2;
    return n;
}
```

C++ Arrays

programiz.com/cpp-programming/arrays

Join our newsletter for the latest updates.

In this tutorial, we will learn to work with arrays. We will learn to declare, initialize, and access array elements in C++ programming with the help of examples.

In C++, an array is a variable that can store multiple values of the same type. For example,

Suppose a class has 27 students, and we need to store the grades of all of them. Instead of creating 27 separate variables, we can simply create an array:

```
double grade[27];
```

Here, *grade* is an array that can hold a maximum of 27 elements of `double` type.

In C++, the size and type of arrays cannot be changed after its declaration.

C++ Array Declaration

```
dataType arrayName[arraySize];
```

For example,

```
int x[6];
```

Here,

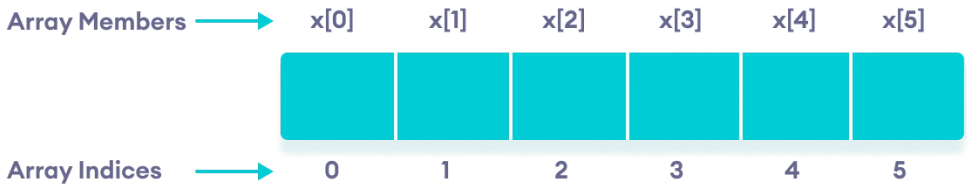
- `int` - type of element to be stored
- *x* - name of the array
- `6` - size of the array

Access Elements in C++ Array

In C++, each element in an array is associated with a number. The number is known as an array index. We can access elements of an array by using those indices.

```
// syntax to access array elements
array[index];
```

Consider the array *x* we have seen above.



Elements of an array in C++

Few Things to Remember:

- The array indices start with `0`. Meaning *x[0]* is the first element stored at index `0`.
- If the size of an array is `n`, the last element is stored at index `(n-1)`. In this example, *x[5]* is the last element.
- Elements of an array have consecutive addresses. For example, suppose the starting address of `x[0]` is 2120d. Then, the address of the next element `x[1]` will be 2124d, the address of `x[2]` will be 2128d and so on.

  Here, the size of each element is increased by 4. This is because the size of `int` is 4 bytes.

C++ Array Initialization

In C++, it's possible to initialize an array during declaration. For example,

```
// declare and initialize and array
int x[6] = {19, 10, 8, 17, 9, 15};
```

C++ Array elements and their data

Another method to initialize array during declaration:

```
// declare and initialize an array
int x[] = {19, 10, 8, 17, 9, 15};
```

Here, we have not mentioned the size of the array. In such cases, the compiler automatically computes the size.

C++ Array With Empty Members

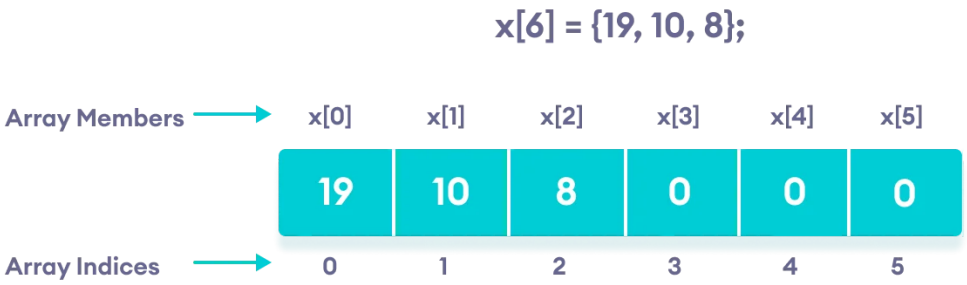
In C++, if an array has a size `n`, we can store upto *n* number of elements in the array. However, what will happen if we store less than *n* number of elements.

For example,

```
// store only 3 elements in the array
int x[6] = {19, 10, 8};
```

Here, the array *x* has a size of `6`. However, we have initialized it with only 3 elements.

In such cases, the compiler assigns random values to the remaining places. Oftentimes, this random value is simply `0`.



Empty array members are automatically assigned the value 0

How to insert and print array elements?

```
int mark[5] = {19, 10, 8, 17, 9}

// change 4th element to 9
mark[3] = 9;

// take input from the user
// store the value at third position
cin >> mark[2];


// take input from the user
// insert at ith position
cin >> mark[i-1];

// print first element of the array
cout << mark[0];

// print ith element of the array
cout >> mark[i-1];
```

Example 1: Displaying Array Elements

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

int main() {
    int numbers[5] = {7, 5, 6, 12, 35};

    cout << "The numbers are: ";

    //  Printing array elements
    // using range based for loop
    for (const int &n : numbers) {
        cout << n << "  ";
    }


    cout << "\nThe numbers are: ";

    //  Printing array elements
    // using traditional for loop
    for (int i = 0; i < 5; ++i) {
        cout << numbers[i] << "  ";
    }

    return 0;
}
```

Output

```
The numbers are: 7  5  6  12  35
The numbers are: 7  5  6  12  35
```

Here, we have used a `for` loop to iterate from `i = 0` to `i = 4`. In each iteration, we have printed `numbers[i]`.

We again used a range based for loop to print out the elements of the array. To learn more about this loop, check C++ Ranged for Loop.

Note: In our range based loop, we have used the code `const int &n` instead of `int n` as the range declaration. However, the `const int &n` is more preferred because:

1. Using `int n` simply copies the array elements to the variable *n* during each iteration. This is not memory-efficient.

   &n, however, uses the memory address of the array elements to access their data without copying them to a new variable. This is memory-efficient.
2. We are simply printing the array elements, not modifying them. Therefore, we use `const` so as not to accidentally change the values of the array.

Example 2: Take Inputs from User and Store Them in an Array

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

int main() {
    int numbers[5];

    cout << "Enter 5 numbers: " << endl;

    //  store input from user to array
    for (int i = 0; i < 5; ++i) {
        cin >> numbers[i];
    }

    cout << "The numbers are: ";

    //  print array elements
    for (int n = 0; n < 5; ++n) {
        cout << numbers[n] << "  ";
    }

    return 0;
}
```

Output

```
Enter 5 numbers:
11
12
13
14
15
The numbers are: 11  12  13  14  15
```

Once again, we have used a `for` loop to iterate from `i = 0` to `i = 4`. In each iteration, we took an input from the user and stored it in `numbers[i]`.

Then, we used another `for` loop to print all the array elements.

Example 3: Display Sum and Average of Array Elements Using for Loop

---

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

int main() {

    // initialize an array without specifying size
    double numbers[] = {7, 5, 6, 12, 35, 27};

    double sum = 0;
    double count = 0;
    double average;

    cout << "The numbers are: ";

    //  print array elements
    // use of range-based for loop
    for (const double &n : numbers) {
        cout << n << "  ";

        //  calculate the sum
        sum += n;

        // count the no. of array elements
        ++count;
    }

    // print the sum
    cout << "\nTheir Sum = " << sum << endl;

    // find the average
    average = sum / count;
    cout << "Their Average = " << average << endl;

    return 0;
}
```

Output

```
The numbers are: 7  5  6  12  35  27
Their Sum = 92
Their Average = 15.3333
```

In this program:

1. We have initialized a *double* array named *numbers* but without specifying its size. We also declared three double variables *sum*, *count*, and *average*.

   Here, `sum =0` and `count = 0` .
2. Then we used a range based `for` loop to print the array elements. In each iteration of the loop, we add the current array element to *sum*.
3. We also increase the value of *count* by `1` in each iteration, so that we can get the size of the array by the end of the for loop.
4. After printing all the elements, we print the sum and the average of all the numbers. The average of the numbers is given by `average = sum / count;`

Note: We used a ranged `for` loop instead of a normal `for` loop.

A normal `for` loop requires us to specify the number of iterations, which is given by the size of the array.

But a ranged `for` loop does not require such specifications.

C++ Array Out of Bounds

If we declare an array of size 10, then the array will contain elements from index 0 to 9.

However, if we try to access the element at index 10 or more than 10, it will result in Undefined Behaviour.

C++ Multidimensional Arrays

programiz.com/cpp-programming/multidimensional-arrays

Join our newsletter for the latest updates.

In this tutorial, we'll learn about multi-dimensional arrays in C++. More specifically, how to declare them, access them, and use them efficiently in our program.

In C++, we can create an array of an array, known as a multidimensional array. For example:

```
int x[3][4];
```

Here, *x* is a two-dimensional array. It can hold a maximum of 12 elements.

We can think of this array as a table with 3 rows and each row has 4 columns as shown below.

	Col 1	Col 2	Col 3	Col 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

Elements in two-dimensional array in C++ Programming

Three-dimensional arrays also work in a similar way. For example:

```
float x[2][4][3];
```

This array *x* can hold a maximum of 24 elements.

We can find out the total number of elements in the array simply by multiplying its dimensions:

```
2 x 4 x 3 = 24
```

Multidimensional Array Initialization

Like a normal array, we can initialize a multidimensional array in more than one way.

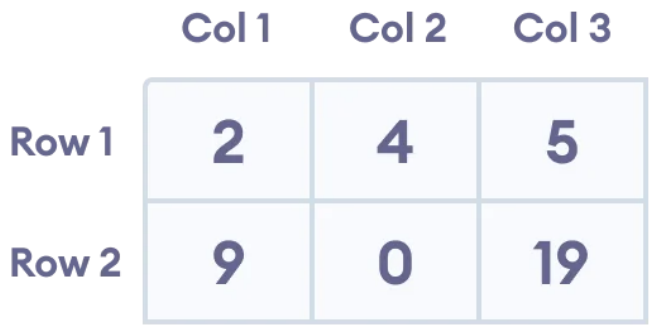
1. Initialization of two-dimensional array

```
int test[2][3] = {2, 4, 5, 9, 0, 19};
```

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

```
int  test[2][3] = { {2, 4, 5}, {9, 0, 19}};
```

This array has 2 rows and 3 columns, which is why we have two rows of elements with 3 elements each.



Initializing a two-dimensional array in C++

2. Initialization of three-dimensional array

```
int test[2][3][4] = {3, 4, 2, 3, 0, -3, 9, 11, 23, 12, 23,
                2, 13, 4, 56, 3, 5, 9, 3, 5, 5, 1, 4, 9};
```

This is not a good way of initializing a three-dimensional array. A better way to initialize this array is:

```
int test[2][3][4] = {
                    { {3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2} },
                    { {13, 4, 56, 3}, {5, 9, 3, 5}, {5, 1, 4, 9} }
                };
```

Notice the dimensions of this three-dimensional array.

The first dimension has the value `2`. So, the two elements comprising the first dimension are:

```
Element 1 = { {3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2} }
Element 2 = { {13, 4, 56, 3}, {5, 9, 3, 5}, {5, 1, 4, 9} }
```

The second dimension has the value `3`. Notice that each of the elements of the first dimension has three elements each:

```
{3, 4, 2, 3}, {0, -3, 9, 11} and {23, 12, 23, 2} for Element 1.
{13, 4, 56, 3}, {5, 9, 3, 5} and {5, 1, 4, 9} for Element 2.
```

Finally, there are four `int` numbers inside each of the elements of the second dimension:

```
{3, 4, 2, 3}
{0, -3, 9, 11}
... .. ...
... .. ...
```

Example 1: Two Dimensional Array

```
// C++ Program to display all elements
// of an initialised two dimensional array

#include <iostream>
using namespace std;

int main() {
    int test[3][2] = {{2, -5},
                      {4, 0},
                      {9, 1}};

    // use of nested for loop
    // 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 << "test[" << i << "][" << j << "] = " << test[i][j] << endl;
        }
    }

    return 0;
}
```

Output

```
test[0][0] = 2
test[0][1] = -5
test[1][0] = 4
test[1][1] = 0
test[2][0] = 9
test[2][1] = 1
```

In the above example, we have initialized a two-dimensional `int` array named *test* that has 3 "rows" and 2 "columns".

Here, we have used the nested `for` loop to display the array elements.

- the outer loop from `i == 0` to `i == 2` access the rows of the array
- the inner loop from `j == 0` to `j == 1` access the columns of the array

Finally, we print the array elements in each iteration.

Example 2: Taking Input for Two Dimensional Array

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

int main() {
    int numbers[2][3];

    cout << "Enter 6 numbers: " << endl;

    // Storing user input in the array
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 3; ++j) {
            cin >> numbers[i][j];
        }
    }

    cout << "The numbers are: " << endl;

    //  Printing array elements
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 3; ++j) {
            cout << "numbers[" << i << "][" << j << "]: " << numbers[i][j] <<
endl;
        }
    }

    return 0;
}
```

Output

```
Enter 6 numbers:
1
2
3
4
5
6
The numbers are:
numbers[0][0]: 1
numbers[0][1]: 2
numbers[0][2]: 3
numbers[1][0]: 4
numbers[1][1]: 5
numbers[1][2]: 6
```

Here, we have used a nested `for` loop to take the input of the 2d array. Once all the input has been taken, we have used another nested `for` loop to print the array members.

Example 3: Three Dimensional Array

```
// C++ Program to Store value entered by user in
// three dimensional array and display it.

#include <iostream>
using namespace std;

int main() {
    // This array can store upto 12 elements (2x3x2)
    int test[2][3][2] = {
                                {
                                    {1, 2},
                                    {3, 4},
                                    {5, 6}
                                },
                                {
                                    {7, 8},
                                    {9, 10},
                                    {11, 12}
                                }
                            };

    // Displaying the values with proper index.
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 3; ++j) {
            for (int k = 0; k < 2; ++k) {
                cout << "test[" << i << "][" << j << "][" << k << "] = " <<
test[i][j][k] << endl;
            }
        }
    }

    return 0;
}
```

Output

```
test[0][0][0] = 1
test[0][0][1] = 2
test[0][1][0] = 3
test[0][1][1] = 4
test[0][2][0] = 5
test[0][2][1] = 6
test[1][0][0] = 7
test[1][0][1] = 8
test[1][1][0] = 9
test[1][1][1] = 10
test[1][2][0] = 11
test[1][2][1] = 12
```

The basic concept of printing elements of a 3d array is similar to that of a 2d array.

However, since we are manipulating 3 dimensions, we use a nested for loop with 3 total loops instead of just 2:

- the outer loop from `i == 0` to `i == 1` accesses the first dimension of the array
- the middle loop from `j == 0` to `j == 2` accesses the second dimension of the array

- the innermost loop from `k == 0` to `k == 1` accesses the third dimension of the array

As we can see, the complexity of the array increases exponentially with the increase in dimensions.

Passing Array to a Function in C++ Programming

programiz.com/cpp-programming/passing-arrays-function

Join our newsletter for the latest updates.

In this tutorial, we will learn how to pass a single-dimensional and multidimensional array as a function parameter in C++ with the help of examples.

In C++, we can pass arrays as an argument to a function. And, also we can return arrays from a function.

Before you learn about passing arrays as a function argument, make sure you know about C++ Arrays and C++ Functions.

Syntax for Passing Arrays as Function Parameters

The syntax for passing an array to a function is:

```
returnType functionName(dataType arrayName[arraySize]) {
    // code
}
```

Let's see an example,

```
int total(int marks[5]) {
    // code
}
```

Here, we have passed an `int` type array named *marks* to the function `total()`. The size of the array is *5*.

Example 1: Passing One-dimensional Array to a Function

```
// C++ Program to display marks of 5 students

#include <iostream>
using namespace std;

// declare function to display marks
// take a 1d array as parameter
void display(int m[5]) {
    cout << "Displaying marks: " << endl;

    // display array elements
    for (int i = 0; i < 5; ++i) {
        cout << "Student " << i + 1 << ": " << m[i] << endl;
    }
}

int main() {

    // declare and initialize an array
    int marks[5] = {88, 76, 90, 61, 69};

    // call display function
    // pass array as argument
    display(marks);

    return 0;
}
```

Output

```
Displaying marks:
Student 1: 88
Student 2: 76
Student 3: 90
Student 4: 61
Student 5: 69
```

Here,

1. When we call a function by passing an array as the argument, only the name of the array is used.

   ```
   display(marks);
   ```

   Here, the argument *marks* represent the memory address of the first element of array *marks[5]*.
2. However, notice the parameter of the `display()` function.

   ```
   void display(int m[5])
   ```

   Here, we use the full declaration of the array in the function parameter, including the square braces `[]`.

3. The function parameter `int m[5]` converts to `int* m;`. This points to the same address pointed by the array *marks*. This means that when we manipulate *m[5]* in the function body, we are actually manipulating the original array *marks*.

   C++ handles passing an array to a function in this way to save memory and time.

Passing Multidimensional Array to a Function

We can also pass Multidimensional arrays as an argument to the function. For example,

Example 2: Passing Multidimensional Array to a Function

```
// C++ Program to display the elements of two
// dimensional array by passing it to a function

#include <iostream>
using namespace std;

// define a function
// pass a 2d array as a parameter
void display(int n[][2]) {
    cout << "Displaying Values: " << endl;
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 2; ++j) {
            cout << "num[" << i << "][" << j << "]: " << n[i][j] << endl;
        }
    }
}

int main() {

    // initialize 2d array
    int num[3][2] = {
        {3, 4},
        {9, 5},
        {7, 1}
    };

    // call the function
    // pass a 2d array as an argument
    display(num);

    return 0;
}
```

Output

```
Displaying Values:
num[0][0]: 3
num[0][1]: 4
num[1][0]: 9
num[1][1]: 5
num[2][0]: 7
num[2][1]: 1
```

In the above program, we have defined a function named `display()`. The function takes a two dimensional array, `int n[][2]` as its argument and prints the elements of the array.

While calling the function, we only pass the name of the two dimensional array as the function argument `display(num)`.

Note: It is not mandatory to specify the number of rows in the array. However, the number of columns should always be specified. This is why we have used `int n[][2]`.

We can also pass arrays with more than 2 dimensions as a function argument.

C++ Returning an Array From a Function

We can also return an array from the function. However, the actual array is not returned. Instead the address of the first element of the array is returned with the help of pointers.

We will learn about returning arrays from a function in the coming tutorials.

C++ Strings

programiz.com/cpp-programming/strings

In this tutorial, you'll learn to handle strings in C++. You'll learn to declare them, initialize them and use them for various input/output operations.

String is a collection of characters. There are two types of strings commonly used in C++ programming language:

- Strings that are objects of string class (The Standard C++ Library string class)
- C-strings (C-style Strings)

C-strings

In C programming, the collection of characters is stored in the form of arrays. This is also supported in C++ programming. Hence it's called C-strings.

C-strings are arrays of type `char` terminated with null character, that is, `\0` (ASCII value of null character is 0).

How to define a C-string?

```
char str[] = "C++";
```

In the above code, `str` is a string and it holds 4 characters.

Although, " `C++` " has 3 character, the null character `\0` is added to the end of the string automatically.

Alternative ways of defining a string

```
char str[4] = "C++";
```

```
char str[] = {'C','+','+','\0'};
```

```
char str[4] = {'C','+','+','\0'};
```

Like arrays, it is not necessary to use all the space allocated for the string. For example:

```
char str[100] = "C++";
```

Example 1: C++ String to read a word

C++ program to display a string entered by user.

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

int main()
{
    char str[100];

    cout << "Enter a string: ";
    cin >> str;
    cout << "You entered: " << str << endl;

    cout << "\nEnter another string: ";
    cin >> str;
    cout << "You entered: "<<str<<endl;

    return 0;
}
```

Output

```
Enter a string: C++
You entered: C++

Enter another string: Programming is fun.
You entered: Programming
```

Notice that, in the second example only "Programming" is displayed instead of "Programming is fun".

This is because the extraction operator >> works as `scanf()` in C and considers a space " " has a terminating character.

Example 2: C++ String to read a line of text

C++ program to read and display an entire line entered by user.

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

int main()
{
    char str[100];
    cout << "Enter a string: ";
    cin.get(str, 100);

    cout << "You entered: " << str << endl;
    return 0;
}
```

Output

```
Enter a string: Programming is fun.
You entered: Programming is fun.
```

To read the text containing blank space, `cin.get` function can be used. This function takes two arguments.

First argument is the name of the string (address of first element of string) and second argument is the maximum size of the array.

In the above program, *str* is the name of the string and `100` is the maximum size of the array.

---

string Object

In C++, you can also create a string object for holding strings.

Unlike using char arrays, string objects has no fixed length, and can be extended as per your requirement.

Example 3: C++ string using string data type

```
#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: Programming is fun.
You entered: Programming is fun.
```

In this program, a 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 location of the line to be stored.

Passing String to a Function

Strings are passed to a function in a similar way arrays are passed to a function.

```
#include <iostream>

using namespace std;

void display(char *);
void display(string);

int main()
{
    string str1;
    char str[100];

    cout << "Enter a string: ";
    getline(cin, str1);

    cout << "Enter another string: ";
    cin.get(str, 100, '\n');

    display(str1);
    display(str);
    return 0;
}

void display(char s[])
{
    cout << "Entered char array is: " << s << endl;
}

void display(string s)
{
    cout << "Entered string is: " << s << endl;
}
```

Output

```
Enter a string:  Programming is fun.
Enter another string:  Really?
Entered string is: Programming is fun.
Entered char array is: Really?
```

In the above program, two strings are asked to enter. These are stored in *str* and *str1* respectively, where str is a `char` array and str1 is a `string` object.

Then, we have two functions `display()` that outputs the string onto the string.

The only difference between the two functions is the parameter. The first `display()` function takes char array as a parameter, while the second takes string as a parameter.

This process is known as function overloading. Learn more about Function Overloading.

C++ Structures

programiz.com/cpp-programming/structure

Join our newsletter for the latest updates.

In this article, you'll learn about structures in C++ programming; what is it, how to define it and use it in your program.

Structure is a collection of variables of different data types under a single name. It is similar to a class in that, both holds a collecion of data of different data types.

For example: You want to store some information about a person: his/her name, citizenship number and salary. You can easily create different variables *name, citNo, salary* to store these information separately.

However, in the future, you would want to store information about multiple persons. Now, you'd need to create different variables for each information per person: *name1, citNo1, salary1, name2, citNo2, salary2*

You can easily visualize how big and messy the code would look. Also, since no relation between the variables (information) would exist, it's going to be a daunting task.

A better approach will be to have a collection of all related information under a single name `Person`, and use it for every person. Now, the code looks much cleaner, readable and efficient as well.

This collection of all related information under a single name `Person` is a structure.

How to declare a structure in C++ programming?

The `struct` keyword defines a structure type followed by an identifier (name of the structure).

Then inside the curly braces, you can declare one or more members (declare variables inside curly braces) of that structure. For example:

```
struct Person
{
    char name[50];
    int age;
    float salary;
};
```

Here a structure *person* is defined which has three members: *name*, *age* and *salary*.

When a structure is created, no memory is allocated.

The structure definition is only the blueprint for the creating of variables. You can imagine it as a datatype. When you define an integer as below:

```
int foo;
```

The `int` specifies that, variable *foo* can hold integer element only. Similarly, structure definition only specifies that, what property a structure variable holds when it is defined.

Note: Remember to end the declaration with a semicolon **(;)**

How to define a structure variable?

Once you declare a structure *person* as above. You can define a structure variable as:

```
Person bill;
```

Here, a structure variable *bill* is defined which is of type structure P`erson`.

When structure variable is defined, only then the required memory is allocated by the compiler.

Considering you have either 32-bit or 64-bit system, the memory of `float` is 4 bytes, memory of `int` is 4 bytes and memory of `char` is 1 byte.

Hence, 58 bytes of memory is allocated for structure variable *bill*.

How to access members of a structure?

The members of structure variable is accessed using a **dot (.)** operator.

Suppose, you want to access *age* of structure variable *bill* and assign it 50 to it. You can perform this task by using following code below:

```
bill.age = 50;
```

Example: C++ Structure

C++ Program to assign data to members of a structure variable and display it.

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

struct Person
{
    char name[50];
    int age;
    float salary;
};

int main()
{
    Person p1;

    cout << "Enter Full name: ";
    cin.get(p1.name, 50);
    cout << "Enter age: ";
    cin >> p1.age;
    cout << "Enter salary: ";
    cin >> p1.salary;

    cout << "\nDisplaying Information." << endl;
    cout << "Name: " << p1.name << endl;
    cout <<"Age: " << p1.age << endl;
    cout << "Salary: " << p1.salary;

    return 0;
}
```

Output

```
Enter Full name: Magdalena Dankova
Enter age: 27
Enter salary: 1024.4

Displaying Information.
Name: Magdalena Dankova
Age: 27
Salary: 1024.4
```

Here a structure P*erson* is declared which has three members *name*, *age* and *salary*.

Inside `main()` function, a structure variable *p1* is defined. Then, the user is asked to enter information and data entered by user is displayed.

You should also check out these structure related tutorials:

C++ Structure and Function

programiz.com/cpp-programming/structure-function

Join our newsletter for the latest updates.

In this article, you'll find relevant examples to pass structures as an argument to a function, and use them in your program.

Structure variables can be passed to a function and returned in a similar way as normal arguments.

Passing structure to function in C++

A structure variable can be passed to a function in similar way as normal argument. Consider this example:

Example 1: C++ Structure and Function

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

struct Person {
    char name[50];
    int age;
    float salary;
};

void displayData(Person);   // Function declaration

int main() {
    Person p;

    cout << "Enter Full name: ";
    cin.get(p.name, 50);
    cout << "Enter age: ";
    cin >> p.age;
    cout << "Enter salary: ";
    cin >> p.salary;

    // Function call with structure variable as an argument
    displayData(p);

    return 0;
}

void displayData(Person p) {
    cout << "\nDisplaying Information." << endl;
    cout << "Name: " << p.name << endl;
    cout <<"Age: " << p.age << endl;
    cout << "Salary: " << p.salary;
}
```

Output

```
Enter Full name: Bill Jobs
Enter age: 55
Enter salary: 34233.4

Displaying Information.
Name: Bill Jobs
Age: 55
Salary: 34233.4
```

In this program, user is asked to enter the *name*, *age* and *salary* of a Person inside `main()` function.

Then, the structure variable *p* is to passed to a function using.

```
displayData(p);
```

The return type of `displayData()` is `void` and a single argument of type structure *Person* is passed.

Then the members of structure `p` is displayed from this function.

Example 2: Returning structure from function in C++

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

struct Person {
    char name[50];
    int age;
    float salary;
};

Person getData(Person);
void displayData(Person);

int main() {

    Person p, temp;

    temp = getData(p);
    p = temp;
    displayData(p);

    return 0;
}

Person getData(Person p) {

    cout << "Enter Full name: ";
    cin.get(p.name, 50);

    cout << "Enter age: ";
    cin >> p.age;

    cout << "Enter salary: ";
    cin >> p.salary;

    return p;
}

void displayData(Person p) {
    cout << "\nDisplaying Information." << endl;
    cout << "Name: " << p.name << endl;
    cout <<"Age: " << p.age << endl;
    cout << "Salary: " << p.salary;
}
```

The output of this program is the same as the program above.

In this program, we have created two structure variables *p* and *temp* of type `Person` under the `main()` function.

The structure variable *p* is passed to `getData()` function which takes input from the user which is then stored in the *temp* variable.

```
temp = getData(p);
```

We then assign the value of *temp* to *p*.

```
p = temp;
```

Then the structure variable *p* is passed to `displayData()` function, which displays the information.

Note: We don't really need to use the *temp* variable for most compilers and C++ versions. Instead, we can simply use the following code:

```
p = getData(p);
```

C++ Pointers to Structure

programiz.com/cpp-programming/structure-pointer

In this article, you'll find relevant examples that will help you to work with pointers to access data within a structure.

A pointer variable can be created not only for native types like (`int` , `float` , `double` etc.) but they can also be created for user defined types like structure.

If you do not know what pointers are, visit C++ pointers.

Here is how you can create pointer for structures:

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

struct temp {
    int i;
    float f;
};

int main() {
    temp *ptr;
    return 0;
}
```

This program creates a pointer *ptr* of type structure *temp*.

Example: Pointers to Structure

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

struct Distance {
    int feet;
    float inch;
};

int main() {
    Distance *ptr, d;

    ptr = &d;

    cout << "Enter feet: ";
    cin >> (*ptr).feet;
    cout << "Enter inch: ";
    cin >> (*ptr).inch;

    cout << "Displaying information." << endl;
    cout << "Distance = " << (*ptr).feet << " feet " << (*ptr).inch << " inches";

    return 0;
}
```

Output

```
Enter feet: 4
Enter inch: 3.5
Displaying information.
Distance = 4 feet 3.5 inches
```

In this program, a pointer variable *ptr* and normal variable *d* of type structure *Distance* is defined.

The address of variable *d* is stored to pointer variable, that is, *ptr* is pointing to variable *d*. Then, the member function of variable *d* is accessed using pointer.

Notes:

- Since pointer *ptr* is pointing to variable *d* in this program, `(*ptr).inch` and `d.inch` are equivalent. Similarly, `(*ptr).feet` and `d.feet` are equivalent.
- However, if we are using pointers, it is far more preferable to access struct members using the `->` operator, since the `.` operator has a higher precedence than the `*` operator.

  Hence, we enclose `*ptr` in brackets when using `(*ptr).inch`. Because of this, it is easier to make mistakes if both operators are used together in a single code.

```
ptr->feet is same as (*ptr).feet
ptr->inch is same as (*ptr).inc
```

C++ Enumeration

programiz.com/cpp-programming/enumeration

Join our newsletter for the latest updates.

In this article, you will learn to work with enumeration (enum). Also, you will learn where enums are commonly used in C++ programming.

An enumeration is a user-defined data type that consists of integral constants. To define an enumeration, keyword **enum** is used.

```
enum season { spring, summer, autumn, winter };
```

Here, the name of the enumeration is *season*.

And, *spring*, *summer* and *winter* are values of type *season*.

By default, *spring* is 0, *summer* is 1 and so on. You can change the default value of an enum element during declaration (if necessary).

```
enum season
{   spring = 0,
    summer = 4,
    autumn = 8,
    winter = 12
};
```

Enumerated Type Declaration

When you create an enumerated type, only blueprint for the variable is created. Here's how you can create variables of enum type.

```
enum boolean { false, true };

// inside function
enum boolean check;
```

Here, a variable *check* of type `enum boolean` is created.

Here is another way to declare same *check* variable using different syntax.

```
enum boolean
{
   false, true
} check;
```

Example 1: Enumeration Type

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

enum week { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };

int main()
{
    week today;
    today = Wednesday;
    cout << "Day " << today+1;
    return 0;
}
```

Output

```
Day 4
```

Example2: Changing Default Value of Enums

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

enum seasons { spring = 34, summer = 4, autumn = 9, winter = 32};

int main() {

    seasons s;

    s = summer;
    cout << "Summer = " << s << endl;

    return 0;
}
```

Output

```
Summer = 4
```

Why enums are used in C++ programming?

An enum variable takes only one value out of many possible values. Example to demonstrate it,

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

enum suit {
    club = 0,
    diamonds = 10,
    hearts = 20,
    spades = 3
} card;

int main()
{
    card = club;
    cout << "Size of enum variable " << sizeof(card) << " bytes.";
    return 0;
}
```

Output

```
Size of enum variable 4 bytes.
```

It's because the size of an integer is 4 bytes.;

This makes enum a good choice to work with flags.

You can accomplish the same task using C++ structures. However, working with enums gives you efficiency along with flexibility.

How to use enums for flags?

Let us take an example,

```
enum designFlags {
        ITALICS = 1,
        BOLD = 2,
        UNDERLINE = 4
} button;
```

Suppose you are designing a button for Windows application. You can set flags *ITALICS*, *BOLD* and *UNDERLINE* to work with text.

There is a reason why all the integral constants are power of 2 in above pseudocode.

```
// In binary

ITALICS = 00000001
BOLD = 00000010
UNDERLINE = 00000100
```

Since, the integral constants are power of 2, you can combine two or more flags at once without overlapping using bitwise OR | operator. This allows you to choose two or more flags at once. For example,

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

enum designFlags {
    BOLD = 1,
    ITALICS = 2,
    UNDERLINE = 4
};

int main()
{
    int myDesign = BOLD | UNDERLINE;

        //    00000001
        //  | 00000100
        //  ____________
        //    00000101

    cout << myDesign;

    return 0;
}
```

Output

```
5
```

When the output is 5, you always know that bold and underline is used.

Also, you can add flag to your requirements.

```
if (myDesign & ITALICS) {
    // code for italics
}
```

Here, we have added italics to our design. Note, only code for italics is written inside the if statement.

You can accomplish almost anything in C++ programming without using enumerations. However, they can be pretty handy in certain situations. That's what differentiates good programmers from great programmers.

C++ Classes and Objects

programiz.com/cpp-programming/object-class

Join our newsletter for the latest updates.

In this tutorial, we will learn about objects and classes and how to use them in C++ with the help of examples.

In previous tutorials, we learned about functions and variables. Sometimes it's desirable to put related functions and data in one place so that it's logical and easier to work with.

Suppose, we need to store the length, breadth, and height of a rectangular room and calculate its area and volume.

To handle this task, we can create three variables, say, *length*, *breadth*, and *height* along with the functions `calculateArea()` and `calculateVolume()`.

However, in C++, rather than creating separate variables and functions, we can also wrap these related data and functions in a single place (by creating **objects**). This programming paradigm is known as object-oriented programming.

But before we can create **objects** and use them in C++, we first need to learn about **classes**.

C++ Class

A class is a blueprint for the object.

We can think of a class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions we build the house. House is the object.

Create a Class

A class is defined in C++ using keyword `class` followed by the name of the class.

The body of the class is defined inside the curly brackets and terminated by a semicolon at the end.

```
class className {
   // some data
   // some functions
};
```

For example,

```
class Room {
    public:
        double length;
        double breadth;
        double height;

        double calculateArea(){
            return length * breadth;
        }

        double calculateVolume(){
            return length * breadth * height;
        }

};
```

Here, we defined a class named `Room`.

The variables *length*, *breadth*, and *height* declared inside the class are known as **data members**. And, the functions `calculateArea()` and `calculateVolume()` are known as **member functions** of a class.

C++ Objects

When a class is defined, only the specification for the object is defined; no memory or storage is allocated.

To use the data and access functions defined in the class, we need to create objects.

Syntax to Define Object in C++

```
className objectVariableName;
```

We can create objects of `Room` class (defined in the above example) as follows:

```
// sample function
void sampleFunction() {
    // create objects
    Room room1, room2;
}

int main(){
    // create objects
    Room room3, room4;
}
```

Here, two objects *room1* and *room2* of the `Room` class are created in `sampleFunction()`. Similarly, the objects *room3* and *room4* are created in `main()`.

As we can see, we can create objects of a class in any function of the program. We can also create objects of a class within the class itself, or in other classes.

Also, we can create as many objects as we want from a single class.

C++ Access Data Members and Member Functions

We can access the data members and member functions of a class by using a `.` (dot) operator. For example,

```
room2.calculateArea();
```

This will call the `calculateArea()` function inside the `Room` class for object *room2*.

Similarly, the data members can be accessed as:

```
room1.length = 5.5;
```

In this case, it initializes the *length* variable of *room1* to `5.5`.

Example 1: Object and Class in C++ Programming

```
// Program to illustrate the working of
// objects and class in C++ Programming

#include <iostream>
using namespace std;

// create a class
class Room {

   public:
    double length;
    double breadth;
    double height;

    double calculateArea() {
        return length * breadth;
    }

    double calculateVolume() {
        return length * breadth * height;
    }
};

int main() {

    // create object of Room class
    Room room1;

    // assign values to data members
    room1.length = 42.5;
    room1.breadth = 30.8;
    room1.height = 19.2;

    // calculate and display the area and volume of the room
    cout << "Area of Room =  " << room1.calculateArea() << endl;
    cout << "Volume of Room =  " << room1.calculateVolume() << endl;

    return 0;
}
```

Output

```
Area of Room =  1309
Volume of Room =  25132.8
```

In this program, we have used the `Room` class and its object *room1* to calculate the area and volume of a room.

In `main()`, we assigned the values of *length*, *breadth*, and *height* with the code:

```
room1.length = 42.5;
room1.breadth = 30.8;
room1.height = 19.2;
```

We then called the functions `calculateArea()` and `calculateVolume()` to perform the necessary calculations.

Note the use of the keyword `public` in the program. This means the members are public and can be accessed anywhere from the program.

As per our needs, we can also create private members using the `private` keyword. The private members of a class can only be accessed from within the class. For example,

```
class Test {

private:

    int a;
    void function1() { }

public:
    int b;
    void function2() { }
}
```

Here, *a* and `function1()` are private. Thus they cannot be accessed from outside the class.

On the other hand, *b* and `function2()` are accessible from everywhere in the program.

To learn more about public and private keywords, please visit our C++ Class Access Modifiers tutorial.

Example 2: Using public and private in C++ Class

```
// Program to illustrate the working of
// public and private in C++ Class

#include <iostream>
using namespace std;

class Room {

   private:
    double length;
    double breadth;
    double height;

   public:

    // function to initialize private variables
    void initData(double len, double brth, double hgt) {
        length = len;
        breadth = brth;
        height = hgt;
    }

    double calculateArea() {
        return length * breadth;
    }

    double calculateVolume() {
        return length * breadth * height;
    }
};

int main() {

    // create object of Room class
    Room room1;

    // pass the values of private variables as arguments
    room1.initData(42.5, 30.8, 19.2);

    cout << "Area of Room =  " << room1.calculateArea() << endl;
    cout << "Volume of Room =  " << room1.calculateVolume() << endl;

    return 0;
}
```

Output

```
Area of Room =  1309
Volume of Room =  25132.8
```

The above example is nearly identical to the first example, except that the class variables are now private.

Since the variables are now private, we cannot access them directly from `main()`. Hence, using the following code would be invalid:

```
// invalid code
obj.length = 42.5;
obj.breadth = 30.8;
obj.height = 19.2;
```

Instead, we use the public function `initData()` to initialize the private variables via the function parameters `double len`, `double brth`, and `double hgt`.

To learn more on objects and classes, visit these topics:

C++ Constructors

programiz.com/cpp-programming/constructors

Join our newsletter for the latest updates.

In this tutorial, we will learn about the C++ constructor and its type with the help examples.

A constructor is a special type of member function that is called automatically when an object is created.

In C++, a constructor has the same name as that of the class and it does not have a return type. For example,

```
class  Wall {
  public:
    // create a constructor
    Wall() {
      // code
    }
};
```

Here, the function `Wall()` is a constructor of the class `Wall`. Notice that the constructor

- has the same name as the class,
- does not have a return type, and
- is `public`

C++ Default Constructor

A constructor with no parameters is known as a **default constructor**. In the example above, `Wall()` is a default constructor.

Example 1: C++ Default Constructor

```
// C++ program to demonstrate the use of default constructor

#include <iostream>
using namespace std;

// declare a class
class  Wall {
  private:
    double length;

  public:
    // default constructor to initialize variable
    Wall() {
      length = 5.5;
      cout << "Creating a wall." << endl;
      cout << "Length = " << length << endl;
    }
};

int main() {
  Wall wall1;
  return 0;
}
```

Output

```
Creating a Wall
Length = 5.5
```

Here, when the *wall1* object is created, the `Wall()` constructor is called. This sets the *length* variable of the object to `5.5`.

Note: If we have not defined a constructor in our class, then the C++ compiler will automatically create a default constructor with an empty code and no parameters.

C++ Parameterized Constructor

In C++, a constructor with parameters is known as a parameterized constructor. This is the preferred method to initialize member data.

Example 2: C++ Parameterized Constructor

```
// C++ program to calculate the area of a wall

#include <iostream>
using namespace std;

// declare a class
class Wall {
  private:
    double length;
    double height;

  public:
    // parameterized constructor to initialize variables
    Wall(double len, double hgt) {
      length = len;
      height = hgt;
    }

    double calculateArea() {
      return length * height;
    }
};

int main() {
  // create object and initialize data members
  Wall wall1(10.5, 8.6);
  Wall wall2(8.5, 6.3);

  cout << "Area of Wall 1: " << wall1.calculateArea() << endl;
  cout << "Area of Wall 2: " << wall2.calculateArea();

  return 0;
}
```

Output

```
Area of Wall 1: 90.3
Area of Wall 2: 53.55
```

Here, we have created a parameterized constructor `Wall()` that has 2 parameters: `double len` and `double hgt`. The values contained in these parameters are used to initialize the member variables *length* and *height*.

When we create an object of the `Wall` class, we pass the values for the member variables as arguments. The code for this is:

```
Wall wall1(10.5, 8.6);
Wall wall2(8.5, 6.3);
```

With the member variables thus initialized, we can now calculate the area of the wall with the `calculateArea()` function.

C++ Copy Constructor

The copy constructor in C++ is used to copy data of one object to another.

Example 3: C++ Copy Constructor

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

// declare a class
class Wall {
  private:
    double length;
    double height;

  public:

    // initialize variables with parameterized constructor
    Wall(double len, double hgt) {
      length = len;
      height = hgt;
    }

    // copy constructor with a Wall object as parameter
    // copies data of the obj parameter
    Wall(Wall &obj) {
      length = obj.length;
      height = obj.height;
    }

    double calculateArea() {
      return length * height;
    }
};

int main() {
  // create an object of Wall class
  Wall wall1(10.5, 8.6);

  // copy contents of wall1 to wall2
  Wall wall2 = wall1;

  // print areas of wall1 and wall2
  cout << "Area of Wall 1: " << wall1.calculateArea() << endl;
  cout << "Area of Wall 2: " << wall2.calculateArea();

  return 0;
}
```

Output

```
Area of Wall 1: 90.3
Area of Wall 2: 90.3
```

In this program, we have used a copy constructor to copy the contents of one object of the `Wall` class to another. The code of the copy constructor is:

```
Wall(Wall &obj) {
  length = obj.length;
  height = obj.height;
}
```

Notice that the parameter of this constructor has the address of an object of the `Wall` class.

We then assign the values of the variables of the *obj* object to the corresponding variables of the object calling the copy constructor. This is how the contents of the object are copied.

In `main()`, we then create two objects *wall1* and *wall2* and then copy the contents of *wall1* to *wall2*:

```
// copy contents of wall1 to wall2
Wall wall2 = wall1;
```

Here, the *wall2* object calls its copy constructor by passing the address of the *wall1* object as its argument i.e. `&obj = &wall1`.

Note: A constructor is primarily used to initialize objects. They are also used to run a default code when an object is created.

How to pass and return object from C++ Functions?

programiz.com/cpp-programming/pass-return-object-function

Join our newsletter for the latest updates.

In this tutorial, we will learn to pass objects to a function and return an object from a function in C++ programming.

In C++ programming, we can pass objects to a function in a similar manner as passing regular arguments.

Example 1: C++ Pass Objects to Function

```
// C++ program to calculate the average marks of two students

#include <iostream>
using namespace std;

class Student {

   public:
    double marks;

    // constructor to initialize marks
    Student(double m) {
        marks = m;
    }
};

// function that has objects as parameters
void calculateAverage(Student s1, Student s2) {

    // calculate the average of marks of s1 and s2
    double average = (s1.marks + s2.marks) / 2;

   cout << "Average Marks = " << average << endl;

}

int main() {
    Student student1(88.0), student2(56.0);

  // pass the objects as arguments
   calculateAverage(student1, student2);

    return 0;
}
```

Output

```
Average Marks = 72
```

Here, we have passed two `Student` objects *student1* and *student2* as arguments to the `calculateAverage()` function.

```
#include<iostream>

class Student {...};

void calculateAverage(Student s1, Student s2) {
    // code
}

int main() {
    ... ...
    calculateAverage(student1, student2);
    ... ...
}
```

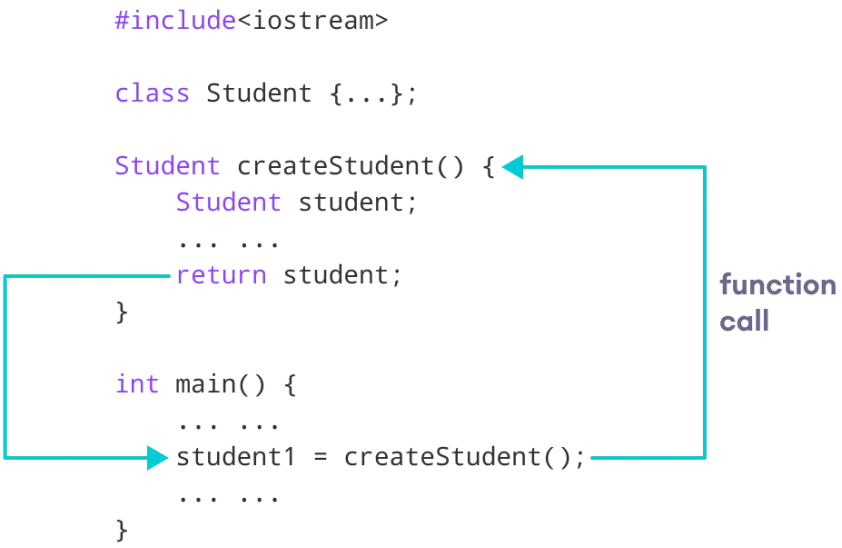
Pass objects to function in C++

Example 2: C++ Return Object from a Function

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

class Student {
   public:
    double marks1, marks2;
};

// function that returns object of Student
Student createStudent() {
    Student student;

    // Initialize member variables of Student
    student.marks1 = 96.5;
    student.marks2 = 75.0;

    // print member variables of Student
    cout << "Marks 1 = " << student.marks1 << endl;
    cout << "Marks 2 = " << student.marks2 << endl;

    return student;
}

int main() {
    Student student1;

    // Call function
    student1 = createStudent();

    return 0;
}
```

Output

```
Marks1 = 96.5
Marks2 = 75
```



Return object from function in C++

In this program, we have created a function `createStudent()` that returns an object of `Student` class.

We have called `createStudent()` from the `main()` method.

```
// Call function
student1 = createStudent();
```

Here, we are storing the object returned by the `createStudent()` method in the *student1*.

C++ Operator Overloading

programiz.com/cpp-programming/operator-overloading

Join our newsletter for the latest updates.

In this tutorial, we will learn about operator overloading with the help of examples.

In C++, we can change the way operators work for user-defined types like objects and structures. This is known as **operator overloading**. For example,

Suppose we have created three objects *c1*, *c2* and *result* from a class named `Complex` that represents complex numbers.

Since operator overloading allows us to change how operators work, we can redefine how the `+` operator works and use it to add the complex numbers of *c1* and *c2* by writing the following code:

```
result = c1 + c2;
```

instead of something like

```
result = c1.addNumbers(c2);
```

This makes our code intuitive and easy to understand.

Note: We cannot use operator overloading for fundamental data types like `int`, `float`, `char` and so on.

Syntax for C++ Operator Overloading

To overload an operator, we use a special `operator` function. We define the function inside the class or structure whose objects/variables we want the overloaded operator to work with.

```
class className {
    ... .. ...
    public
       returnType operator symbol (arguments) {
           ... .. ...
       }
    ... .. ...
};
```

Here,

- `returnType` is the return type of the function.
- `operator` is a keyword.
- `symbol` is the operator we want to overload. Like: `+`, `<`, `-`, `++`, etc.
- `arguments` is the arguments passed to the function.

Operator Overloading in Unary Operators

Unary operators operate on only one operand. The increment operator `++` and decrement operator `--` are examples of unary operators.

Example1: ++ Operator (Unary Operator) Overloading

```
// Overload ++ when used as prefix

#include <iostream>
using namespace std;

class Count {
   private:
    int value;

   public:

    // Constructor to initialize count to 5
    Count() : value(5) {}

    // Overload ++ when used as prefix
    void operator ++ () {
        ++value;
    }

    void display() {
        cout << "Count: " << value << endl;
    }
};

int main() {
    Count count1;

    // Call the "void operator ++ ()" function
    ++count1;

    count1.display();
    return 0;
}
```

Output

```
Count: 6
```

Here, when we use `++count1;` , the `void operator ++ ()` is called. This increases the *value* attribute for the object *count1* by 1.

Note: When we overload operators, we can use it to work in any way we like. For example, we could have used `++` to increase *value* by 100.

However, this makes our code confusing and difficult to understand. It's our job as a programmer to use operator overloading properly and in a consistent and intuitive way.

The above example works only when `++` is used as a prefix. To make `++` work as a postfix we use this syntax.

```
void operator ++ (int) {
    // code
}
```

Notice the `int` inside the parentheses. It's the syntax used for using unary operators as postfix; it's not a function parameter.

Example 2: ++ Operator (Unary Operator) Overloading

```
// Overload ++ when used as prefix and postfix

#include <iostream>
using namespace std;

class Count {
   private:
    int value;

   public:

    // Constructor to initialize count to 5
    Count() : value(5) {}


    // Overload ++ when used as prefix
    void operator ++ () {
        ++value;
    }


    // Overload ++ when used as postfix
    void operator ++ (int) {
        value++;
    }

    void display() {
        cout << "Count: " << value << endl;
    }
};

int main() {
    Count count1;

    // Call the "void operator ++ (int)" function
    count1++;
    count1.display();

    // Call the "void operator ++ ()" function
    ++count1;

    count1.display();
    return 0;
}
```

Output

```
Count: 6
Count: 7
```

The **Example 2** works when `++` is used as both prefix and postfix. However, it doesn't work if we try to do something like this:

```
Count count1, result;

// Error
result = ++count1;
```

This is because the return type of our operator function is `void`. We can solve this problem by making `Count` as the return type of the operator function.

```
// return Count when ++ used as prefix

Count operator ++ () {
    // code
}

// return Count when ++ used as postfix

Count operator ++ (int) {
   // code
}
```

Example 3: Return Value from Operator Function (++ Operator)

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

class Count {
   private:
    int value;

   public
       :
    // Constructor to initialize count to 5
    Count() : value(5) {}

    // Overload ++ when used as prefix
    Count operator ++ () {
        Count temp;

        // Here, value is the value attribute of the calling object
        temp.value = ++value;

        return temp;
    }

    // Overload ++ when used as postfix
    Count operator ++ (int) {
        Count temp;

        // Here, value is the value attribute of the calling object
        temp.value = value++;

        return temp;
    }

    void display() {
        cout << "Count: " << value << endl;
    }
};

int main() {
    Count count1, result;

    // Call the "Count operator ++ ()" function
    result = ++count1;
    result.display();

    // Call the "Count operator ++ (int)" function
    result = count1++;
    result.display();

    return 0;
}
```

Output

```
Count: 6
Count: 6
```

Here, we have used the following code for prefix operator overloading:

```
// Overload ++ when used as prefix
Count operator ++ () {
    Count temp;

    // Here, value is the value attribute of the calling object
    temp.value = ++value;

    return temp;
}
```

The code for the postfix operator overloading is also similar. Notice that we have created an object *temp* and returned its value to the operator function.

Also, notice the code

```
temp.value = ++value;
```

The variable *value* belongs to the *count1* object in `main()` because *count1* is calling the function, while *temp.value* belongs to the *temp* object.

Operator Overloading in Binary Operators

Binary operators work on two operands. For example,

```
result = num + 9;
```

Here, `+` is a binary operator that works on the operands *num* and `9`.

When we overload the binary operator for user-defined types by using the code:

```
obj3 = obj1 + obj2;
```

The operator function is called using the *obj1* object and *obj2* is passed as an argument to the function.

Example 4: C++ Binary Operator Overloading

```
// C++ program to overload the binary operator +
// This program adds two complex numbers

#include <iostream>
using namespace std;

class Complex {
   private:
    float real;
    float imag;

   public:
    // Constructor to initialize real and imag to 0
    Complex() : real(0), imag(0) {}

    void input() {
        cout << "Enter real and imaginary parts respectively: ";
        cin >> real;
        cin >> imag;
    }

    // Overload the + operator
    Complex operator + (const Complex& obj) {
        Complex temp;
        temp.real = real + obj.real;
        temp.imag = imag + obj.imag;
        return temp;
    }

    void output() {
        if (imag < 0)
            cout << "Output Complex number: " << real << imag << "i";
        else
            cout << "Output Complex number: " << real << "+" << imag << "i";
    }
};

int main() {
    Complex complex1, complex2, result;

    cout << "Enter first complex number:\n";
    complex1.input();

    cout << "Enter second complex number:\n";
    complex2.input();

   // complex1 calls the operator function
   // complex2 is passed as an argument to the function
    result = complex1 + complex2;
    result.output();

    return 0;
}
```

Output

```
Enter first complex number:
Enter real and imaginary parts respectively: 9 5
Enter second complex number:
Enter real and imaginary parts respectively: 7 6
Output Complex number: 16+11i
```

In this program, the operator function is:

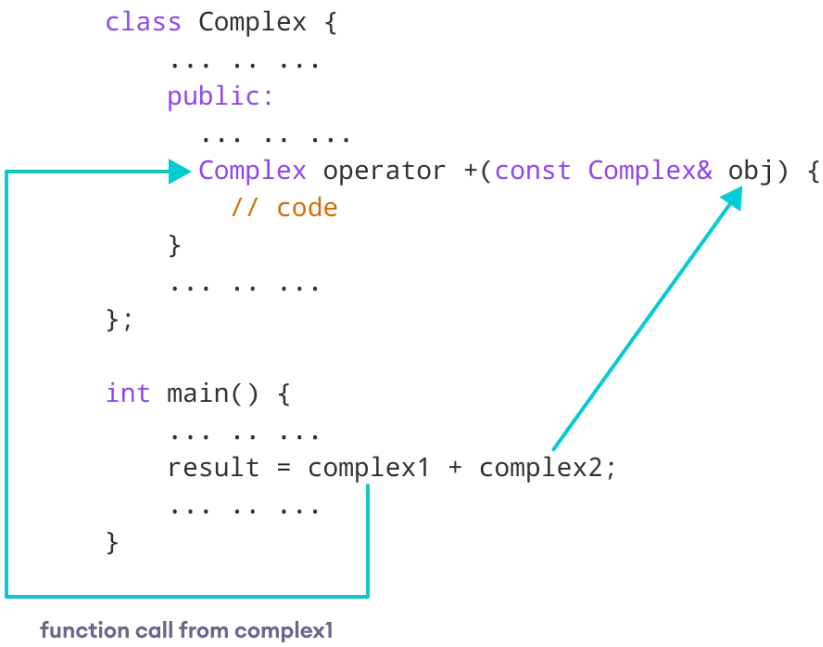
```
Complex operator + (const Complex& obj) {
    // code
}
```

Instead of this, we also could have written this function like:

```
Complex operator + (Complex obj) {
    // code
}
```

However,

- using `&` makes our code efficient by referencing the *complex2* object instead of making a duplicate object inside the operator function.
- using `const` is considered a good practice because it prevents the operator function from modifying *complex2*.



Overloading binary operators in C++

Things to Remember in C++ Operator Overloading

1. Two operators `=` and `&` are already overloaded by default in C++. For example, to copy objects of the same class, we can directly use the `=` operator. We do not need to create an operator function.

2. Operator overloading cannot change the precedence and associativity of operators. However, if we want to change the order of evaluation, parentheses should be used.
3. There are 4 operators that cannot be overloaded in C++. They are:
    a. `::` (scope resolution)
    b. `.` (member selection)
    c. `.*` (member selection through pointer to function)
    d. `?:` (ternary operator)

Visit these pages to learn more on:

C++ Pointers

programiz.com/cpp-programming/pointers

Join our newsletter for the latest updates.

In this tutorial, we will learn about pointers in C++ and their working with the help of examples.

In C++, pointers are variables that store the memory addresses of other variables.

Address in C++

If we have a variable *var* in our program, *&var* will give us its address in the memory. For example,

Example 1: Printing Variable Addresses in C++

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

int main()
{
    // declare variables
    int var1 = 3;
    int var2 = 24;
    int var3 = 17;

    // print address of var1
    cout << "Address of var1: "<< &var1 << endl;

    // print address of var2
    cout << "Address of var2: " << &var2 << endl;

    // print address of var3
    cout << "Address of var3: " << &var3 << endl;
}
```

Output

```
Address of var1: 0x7fff5fbff8ac
Address of var2: 0x7fff5fbff8a8
Address of var3: 0x7fff5fbff8a4
```

Here, `0x` at the beginning represents the address is in the hexadecimal form.

Notice that the first address differs from the second by 4 bytes and the second address differs from the third by 4 bytes.

This is because the size of an `int` variable is 4 bytes in a 64-bit system.

Note: You may not get the same results when you run the program.

As mentioned above, pointers are used to store addresses rather than values.

Here is how we can declare pointers.

```
int *pointVar;
```

Here, we have declared a pointer *pointVar* of the `int` type.

We can also declare pointers in the following way.

```
int* pointVar; // preferred syntax
```

Let's take another example of declaring pointers.

```
int* pointVar, p;
```

Here, we have declared a pointer *pointVar* and a normal variable *p*.

Note: The `*` operator is used after the data type to declare pointers.

Assigning Addresses to Pointers

Here is how we can assign addresses to pointers:

```
int* pointVar, var;
var = 5;

// assign address of var to pointVar pointer
pointVar = &var;
```

Here, `5` is assigned to the variable *var*. And, the address of *var* is assigned to the *pointVar* pointer with the code `pointVar = &var`.

Get the Value from the Address Using Pointers

To get the value pointed by a pointer, we use the `*` operator. For example:

```
int* pointVar, var;
var = 5;

// assign address of var to pointVar
pointVar = &var;

// access value pointed by pointVar
cout << *pointVar << endl;   // Output: 5
```

In the above code, the address of var is assigned to *pointVar*. We have used the `*pointVar` to get the value stored in that address.

When `*` is used with pointers, it's called the **dereference operator**. It operates on a pointer and gives the value pointed by the address stored in the pointer. That is, `*pointVar = var`.

Note: In C++, *pointVar* and **pointVar* is completely different. We cannot do something like `*pointVar = &var;`

Example 2: Working of C++ Pointers

```
#include <iostream>
using namespace std;
int main() {
    int var = 5;

    // declare pointer variable
    int* pointVar;

    // store address of var
    pointVar = &var;

    // print value of var
    cout << "var = " << var << endl;

    // print address of var
    cout << "Address of var (&var) = " << &var << endl
         << endl;

    // print pointer pointVar
    cout << "pointVar = " << pointVar << endl;

    // print the content of the address pointVar points to
    cout << "Content of the address pointed to by pointVar (*pointVar) = " <<
*pointVar << endl;

    return 0;
}
```

Output

```
var = 5
Address of var (&var) = 0x61ff08

pointVar = 0x61ff08
Content of the address pointed to by pointVar (*pointVar) = 5
```

Working of C++ pointers

Changing Value Pointed by Pointers

If *pointVar* points to the address of *var*, we can change the value of *var* by using **pointVar*.

For example,

```
int var = 5;
int* pointVar;

// assign address of var
pointVar = &var;

// change value at address pointVar
*pointVar = 1;

cout << var << endl; // Output: 1
```

Here, *pointVar* and `&var` have the same address, the value of *var* will also be changed when **pointVar* is changed.

Example 3: Changing Value Pointed by Pointers

```
#include <iostream>
using namespace std;
int main() {
    int var = 5;
    int* pointVar;

    // store address of var
    pointVar = &var;

    // print var
    cout << "var = " << var << endl;

    // print *pointVar
    cout << "*pointVar = " << *pointVar << endl
         << endl;

    cout << "Changing value of var to 7:" << endl;

    // change value of var to 7
    var = 7;

    // print var
    cout << "var = " << var << endl;

    // print *pointVar
    cout << "*pointVar = " << *pointVar << endl
         << endl;

    cout << "Changing value of *pointVar to 16:" << endl;

    // change value of var to 16
    *pointVar = 16;

    // print var
    cout << "var = " << var << endl;

    // print *pointVar
    cout << "*pointVar = " << *pointVar << endl;
    return 0;
}
```

Output

```
var = 5
*pointVar = 5

Changing value of var to 7:
var = 7
*pointVar = 7

Changing value of *pointVar to 16:
var = 16
*pointVar = 16
```

Common mistakes when working with pointers

Suppose, we want a pointer *varPoint* to point to the address of *var*. Then,

```
int var, *varPoint;

// Wrong!
// varPoint is an address but var is not
varPoint = var;

// Wrong!
// &var is an address
// *varPoint is the value stored in &var
*varPoint = &var;

// Correct!
// varPoint is an address and so is &var
varPoint = &var;

 // Correct!
// both *varPoint and var are values
*varPoint = var;
```

Recommended Readings:

C++ Pointers and Arrays

programiz.com/cpp-programming/pointers-arrays

Join our newsletter for the latest updates.

In this tutorial, we will learn about the relation between arrays and pointers with the help of examples.

In C++, Pointers are variables that hold addresses of other variables. Not only can a pointer store the address of a single variable, it can also store the address of cells of an array.

Consider this example:

```
int *ptr;
int arr[5];

// store the address of the first
// element of arr in ptr
ptr = arr;
```

Here, *ptr* is a pointer variable while *arr* is an `int` array. The code `ptr = arr;` stores the address of the first element of the array in variable *ptr*.

Notice that we have used `arr` instead of `&arr[0]`. This is because both are the same. So, the code below is the same as the code above.

```
int *ptr;
int arr[5];
ptr = &arr[0];
```

The addresses for the rest of the array elements are given by `&arr[1]`, `&arr[2]`, `&arr[3]`, and `&arr[4]`.

Point to Every Array Elements

Suppose we need to point to the fourth element of the array using the same pointer *ptr*.

Here, if *ptr* points to the first element in the above example then `ptr + 3` will point to the fourth element. For example,

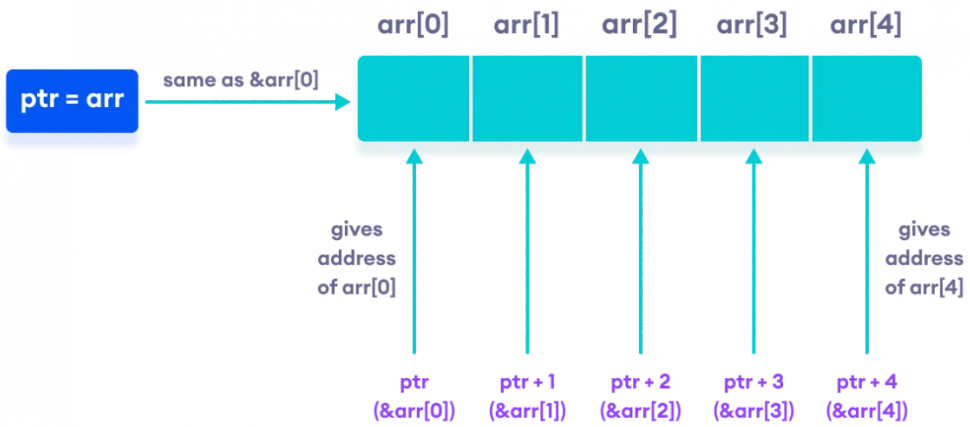
```
int *ptr;
int arr[5];
ptr = arr;

ptr + 1 is equivalent to &arr[1];
ptr + 2 is equivalent to &arr[2];
ptr + 3 is equivalent to &arr[3];
ptr + 4 is equivalent to &arr[4];
```

Similarly, we can access the elements using the single pointer. For example,

```
// use dereference operator
*ptr == arr[0];
*(ptr + 1) is equivalent to arr[1];
*(ptr + 2) is equivalent to arr[2];
*(ptr + 3) is equivalent to arr[3];
*(ptr + 4) is equivalent to arr[4];
```

Suppose if we have initialized `ptr = &arr[2];` then

```
ptr - 2 is equivalent to &arr[0];
ptr - 1 is equivalent to &arr[1];
ptr + 1 is equivalent to &arr[3];
ptr + 2 is equivalent to &arr[4];
```



Working of C++ Pointers with Arrays

Note: The address between *ptr* and *ptr + 1* differs by 4 bytes. It is because *ptr* is a pointer to an `int` data. And, the size of int is 4 bytes in a 64-bit operating system.

Similarly, if pointer *ptr* is pointing to `char` type data, then the address between *ptr* and *ptr + 1* is 1 byte. It is because the size of a character is 1 byte.

Example 1: C++ Pointers and Arrays

```
// C++ Program to display address of each element of an array

#include <iostream>
using namespace std;

int main()
{
    float arr[3];

    // declare pointer variable
    float *ptr;

    cout << "Displaying address using arrays: " << endl;

    // use for loop to print addresses of all array elements
    for (int i = 0; i < 3; ++i)
    {
        cout << "&arr[" << i << "] = " << &arr[i] << endl;
    }

    // ptr = &arr[0]
    ptr = arr;

    cout<<"\nDisplaying address using pointers: "<< endl;

    // use for loop to print addresses of all array elements
    // using pointer notation
    for (int i = 0; i < 3; ++i)
    {
        cout << "ptr + " << i << " = "<< ptr + i << endl;
    }

    return 0;
}
```

Output

```
Displaying address using arrays:
&arr[0] = 0x61fef0
&arr[1] = 0x61fef4
&arr[2] = 0x61fef8

Displaying address using pointers:
ptr + 0 = 0x61fef0
ptr + 1 = 0x61fef4
ptr + 2 = 0x61fef8
```

In the above program, we first simply printed the addresses of the array elements without using the pointer variable *ptr*.

Then, we used the pointer *ptr* to point to the address of *a[0]*, `ptr + 1` to point to the address of *a[1]*, and so on.

In most contexts, array names decay to pointers. In simple words, array names are converted to pointers. That's the reason why we can use pointers to access elements of arrays.

However, we should remember that pointers and arrays are not the same.

There are a few cases where array names don't decay to pointers. To learn more, visit: When does array name doesn't decay into a pointer?

Example 2: Array name used as pointer

```
// C++ Program to insert and display data entered by using pointer notation.

#include <iostream>
using namespace std;

int main() {
    float arr[5];

   // Insert data using pointer notation
    cout << "Enter 5 numbers: ";
    for (int i = 0; i < 5; ++i) {

        // store input number in arr[i]
        cin >> *(arr + i) ;

    }

    // Display data using pointer notation
    cout << "Displaying data: " << endl;
    for (int i = 0; i < 5; ++i) {

        // display value of arr[i]
        cout << *(arr + i) << endl ;

    }

    return 0;
}
```

Output

```
Enter 5 numbers: 2.5
3.5
4.5
5
2
Displaying data:
2.5
3.5
4.5
5
2
```

Here,

1. We first used the pointer notation to store the numbers entered by the user into the array *arr*.

   ```
   cin >> *(arr + i) ;
   ```

   This code is equivalent to the code below:

   ```
   cin >> arr[i];
   ```

   Notice that we haven't declared a separate pointer variable, but rather we are using the array name *arr* for the pointer notation.

   As we already know, the array name *arr* points to the first element of the array. So, we can think of *arr* as acting like a pointer.

2. Similarly, we then used `for` loop to display the values of *arr* using pointer notation.

   ```
   cout << *(arr + i) << endl ;
   ```

   This code is equivalent to

   ```
   cout << arr[i] << endl ;
   ```

C++ Call by Reference: Using pointers [With Examples]

programiz.com/cpp-programming/pointers-function

Join our newsletter for the latest updates.

In this tutorial, we will learn about C++ call by reference to pass pointers as an argument to the function with the help of examples.

In the C++ Functions tutorial, we learned about passing arguments to a function. This method used is called passing by value because the actual value is passed.

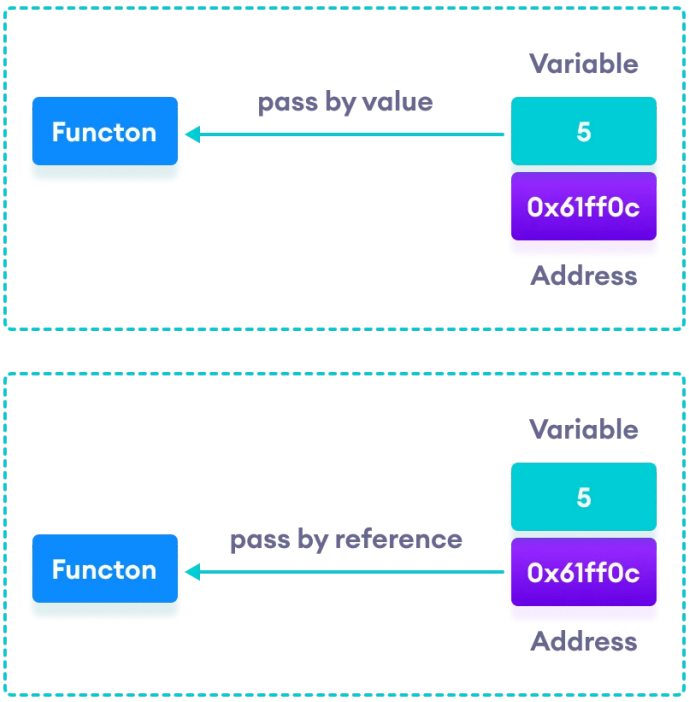
However, there is another way of passing arguments to a function where the actual values of arguments are not passed. Instead, the reference to values is passed.

For example,

```
// function that takes value as parameter

void func1(int numVal) {
    // code
}

// function that takes reference as parameter
// notice the & before the parameter
void func2(int &numRef) {
    // code
}

int main() {
    int num = 5;

    // pass by value
    func1(num);

    // pass by reference
    func2(num);

    return 0;
}
```

Notice the `&` in `void func2(int &numRef)`. This denotes that we are using the address of the variable as our parameter.

So, when we call the `func2()` function in `main()` by passing the variable *num* as an argument, we are actually passing the address of *num* variable instead of the value **5**.



C++ Pass by Value vs. Pass by Reference

Example 1: Passing by reference without pointers

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

// function definition to swap values
void swap(int &n1, int &n2) {
    int temp;
    temp = n1;
    n1 = n2;
    n2 = temp;
}

int main()
{

    // initialize variables
    int a = 1, b = 2;

    cout << "Before swapping" << endl;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;

    // call function to swap numbers
    swap(a, b);

    cout << "\nAfter swapping" << endl;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;

    return 0;
}
```

Output

```
Before swapping
a = 1
b = 2

After swapping
a = 2
b = 1
```

In this program, we passed the variables *a* and *b* to the `swap()` function. Notice the function definition,

```
void swap(int &n1, int &n2)
```

Here, we are using `&` to denote that the function will accept addresses as its parameters.

Hence, the compiler can identify that instead of actual values, the reference of the variables is passed to function parameters.

In the `swap()` function, the function parameters *n1* and *n2* are pointing to the same value as the variables *a* and *b* respectively. Hence the swapping takes place on actual value.

The same task can be done using the pointers. To learn about pointers, visit C++ Pointers.

Example 2: Passing by reference using pointers

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

// function prototype with pointer as parameters
void swap(int*, int*);

int main()
{

    // initialize variables
    int a = 1, b = 2;

    cout << "Before swapping" << endl;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;

    // call function by passing variable addresses
    swap(&a, &b);

    cout << "\nAfter swapping" << endl;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    return 0;
}

// function definition to swap numbers
void swap(int* n1, int* n2) {
    int temp;
    temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}
```

Output

```
Before swapping
a = 1
b = 2

After swapping
a = 2
b = 1
```

Here, we can see the output is the same as the previous example. Notice the line,

```
// &a is address of a
// &b is address of b
swap(&a, &b);
```

Here, the address of the variable is passed during the function call rather than the variable.

Since the address is passed instead of value, a dereference operator `*` must be used to access the value stored in that address.

```
temp = *n1;
*n1 = *n2;
*n2 = temp;
```

`*n1` and `*n2` gives the value stored at address *n1* and *n2* respectively.

Since *n1* and *n2* contain the addresses of *a* and *b*, anything is done to `*n1` and **n2* will change the actual values of *a* and *b*.

Hence, when we print the values of *a* and *b* in the `main()` function, the values are changed.

C++ Memory Management: new and delete

programiz.com/cpp-programming/memory-management

Join our newsletter for the latest updates.

In this tutorial, we will learn to manage memory effectively in C++ using new and delete operations with the help of examples.

C++ allows us to allocate the memory of a variable or an array in run time. This is known as dynamic memory allocation.

In other programming languages such as Java and Python, the compiler automatically manages the memories allocated to variables. But this is not the case in C++.

In C++, we need to deallocate the dynamically allocated memory manually after we have no use for the variable.

We can allocate and then deallocate memory dynamically using the `new` and `delete` operators respectively.

C++ new Operator

The `new` operator allocates memory to a variable. For example,

```
// declare an int pointer
int* pointVar;

// dynamically allocate memory
// using the new keyword
pointVar = new int;

// assign value to allocated memory
*pointVar = 45;
```

Here, we have dynamically allocated memory for an `int` variable using the `new` operator.

Notice that we have used the pointer *pointVar* to allocate the memory dynamically. This is because the `new` operator returns the address of the memory location.

In the case of an array, the `new` operator returns the address of the first element of the array.

From the example above, we can see that the syntax for using the `new` operator is

```
pointerVariable = new dataType;
```

delete Operator

Once we no longer need to use a variable that we have declared dynamically, we can deallocate the memory occupied by the variable.

For this, the `delete` operator is used. It returns the memory to the operating system. This is known as **memory deallocation**.

The syntax for this operator is

```
delete pointerVariable;
```

Consider the code:

```
// declare an int pointer
int* pointVar;

// dynamically allocate memory
// for an int variable
pointVar = new int;

// assign value to the variable memory
*pointVar = 45;

// print the value stored in memory
cout << *pointVar; // Output: 45

// deallocate the memory
delete pointVar;
```

Here, we have dynamically allocated memory for an `int` variable using the pointer *pointVar*.

After printing the contents of *pointVar*, we deallocated the memory using `delete`.

Note: If the program uses a large amount of unwanted memory using `new`, the system may crash because there will be no memory available for the operating system. In this case, the `delete` operator can help the system from crash.

Example 1: C++ Dynamic Memory Allocation

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

int main() {
    // declare an int pointer
    int* pointInt;

    // declare a float pointer
    float* pointFloat;

    // dynamically allocate memory
    pointInt = new int;
    pointFloat = new float;

    // assigning value to the memory
    *pointInt = 45;
    *pointFloat = 45.45f;

    cout << *pointInt << endl;
    cout << *pointFloat << endl;

    // deallocate the memory
    delete pointInt;
    delete pointFloat;

    return 0;
}
```

Output

```
45
45.45
```

In this program, we dynamically allocated memory to two variables of `int` and `float` types. After assigning values to them and printing them, we finally deallocate the memories using the code

```
delete pointInt;
delete pointFloat;
```

Note: Dynamic memory allocation can make memory management more efficient.

Especially for arrays, where a lot of the times we don't know the size of the array until the run time.

Example 2: C++ new and delete Operator for Arrays

```
// C++ Program to store GPA of n number of students and display it
// where n is the number of students entered by the user

#include <iostream>
using namespace std;

int main() {
    int num;
    cout << "Enter total number of students: ";
    cin >> num;
    float* ptr;

    // memory allocation of num number of floats
    ptr = new float[num];

    cout << "Enter GPA of students." << endl;
    for (int i = 0; i < num; ++i) {
        cout << "Student" << i + 1 << ": ";
        cin >> *(ptr + i);
    }

    cout << "\nDisplaying GPA of students." << endl;
    for (int i = 0; i < num; ++i) {
        cout << "Student" << i + 1 << " :" << *(ptr + i) << endl;
    }

    // ptr memory is released
    delete[] ptr;

    return 0;
}
```

Output

```
Enter total number of students: 4
Enter GPA of students.
Student1: 3.6
Student2: 3.1
Student3: 3.9
Student4: 2.9

Displaying GPA of students.
Student1 :3.6
Student2 :3.1
Student3 :3.9
Student4 :2.9
```

In this program, we have asked the user to enter the number of students and store it in the *num* variable.

Then, we have allocated the memory dynamically for the `float` array using *new*.

We enter data into the array (and later print them) using pointer notation.

After we no longer need the array, we deallocate the array memory using the code `delete[] ptr;`.

Notice the use of `[]` after `delete`. We use the square brackets `[]` in order to denote that the memory deallocation is that of an array.

Example 3: C++ new and delete Operator for Objects

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

class Student {
    int age;

   public:

    // constructor initializes age to 12
    Student() : age(12) {}

    void getAge() {
        cout << "Age = " << age << endl;
    }
};

int main() {

    // dynamically declare Student object
    Student* ptr = new Student();

    // call getAge() function
    ptr->getAge();

    // ptr memory is released
    delete ptr;

    return 0;
}
```

Output

```
Age = 12
```

In this program, we have created a `Student` class that has a private variable *age*.

We have initialized *age* to `12` in the default constructor `Student()` and print its value with the function `getAge()`.

In `main()`, we have created a `Student` object using the `new` operator and use the pointer *ptr* to point to its address.

The moment the object is created, the `Student()` constructor initializes *age* to `12`.

We then call the `getAge()` function using the code:

```
ptr->getAge();
```

Notice the arrow operator `->`. This operator is used to access class members using pointers.

C++ Inheritance

programiz.com/cpp-programming/inheritance

Join our newsletter for the latest updates.

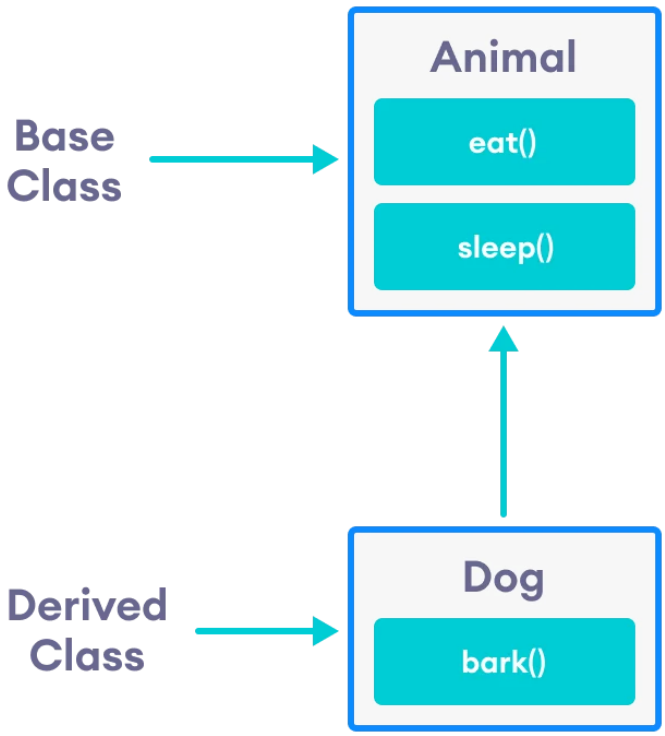
In this tutorial, we will learn about inheritance in C++ with the help of examples.

Inheritance is one of the key features of Object-oriented programming in C++. It allows us to create a new class (derived class) from an existing class (base class).

The derived class inherits the features from the base class and can have additional features of its own. For example,

```
class Animal {
    // eat() function
    // sleep() function
};

class Dog : public Animal {
    // bark() function
};
```

Here, the `Dog` class is derived from the `Animal` class. Since `Dog` is derived from `Animal`, members of `Animal` are accessible to `Dog`.



Inheritance in C++

Notice the use of the keyword `public` while inheriting Dog from Animal.

```
class Dog : public Animal {...};
```

We can also use the keywords `private` and `protected` instead of `public`. We will learn about the differences between using `private`, `public` and `protected` later in this tutorial.

---

is-a relationship

Inheritance is an **is-a relationship**. We use inheritance only if an **is-a relationship** is present between the two classes.

Here are some examples:

- A car is a vehicle.
- Orange is a fruit.
- A surgeon is a doctor.
- A dog is an animal.

Example 1: Simple Example of C++ Inheritance

```
// C++ program to demonstrate inheritance

#include <iostream>
using namespace std;

// base class
class Animal {

   public:
    void eat() {
        cout << "I can eat!" << endl;
    }

    void sleep() {
        cout << "I can sleep!" << endl;
    }
};

// derived class
class Dog : public Animal {

   public:
    void bark() {
        cout << "I can bark! Woof woof!!" << endl;
    }
};

int main() {
    // Create object of the Dog class
    Dog dog1;

    // Calling members of the base class
    dog1.eat();
    dog1.sleep();

    // Calling member of the derived class
    dog1.bark();

    return 0;
}
```

Output

```
I can eat!
I can sleep!
I can bark! Woof woof!!
```

Here, *dog1* (the object of derived class `Dog`) can access members of the base class `Animal`. It's because `Dog` is inherited from `Animal`.

```
// Calling members of the Animal class
dog1.eat();
dog1.sleep();
```

C++ protected Members

The access modifier `protected` is especially relevant when it comes to C++ inheritance.

Like `private` members, `protected` members are inaccessible outside of the class. However, they can be accessed by **derived classes** and **friend classes/functions**.

We need `protected` members if we want to hide the data of a class, but still want that data to be inherited by its derived classes.

To learn more about protected, refer to our C++ Access Modifiers tutorial.

Example 2 : C++ protected Members

```
// C++ program to demonstrate protected members

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

// base class
class Animal {

   private:
    string color;

   protected:
    string type;

   public:
    void eat() {
        cout << "I can eat!" << endl;
    }

    void sleep() {
        cout << "I can sleep!" << endl;
    }

    void setColor(string clr) {
        color = clr;
    }

    string getColor() {
        return color;
    }
};

// derived class
class Dog : public Animal {

   public:
    void setType(string tp) {
        type = tp;
    }

    void displayInfo(string c) {
        cout << "I am a " << type << endl;
        cout << "My color is " << c << endl;
    }

    void bark() {
        cout << "I can bark! Woof woof!!" << endl;
    }
};

int main() {
    // Create object of the Dog class
    Dog dog1;

    // Calling members of the base class
    dog1.eat();
    dog1.sleep();
    dog1.setColor("black");
```

```
    // Calling member of the derived class
    dog1.bark();
    dog1.setType("mammal");

    // Using getColor() of dog1 as argument
    // getColor() returns string data
    dog1.displayInfo(dog1.getColor());

    return 0;
}
```

Output

```
I can eat!
I can sleep!
I can bark! Woof woof!!
I am a mammal
My color is black
```

Here, the variable *type* is `protected` and is thus accessible from the derived class `Dog`. We can see this as we have initialized `type` in the `Dog` class using the function `setType()`.

On the other hand, the `private` variable *color* cannot be initialized in `Dog`.

```
class Dog : public Animal {

    public:
      void setColor(string clr) {
          // Error: member "Animal::color" is inaccessible
          color = clr;
      }
};
```

Also, since the `protected` keyword hides data, we cannot access *type* directly from an object of `Dog` or `Animal` class.

```
// Error: member "Animal::type" is inaccessible
dog1.type = "mammal";
```

Access Modes in C++ Inheritance

In our previous tutorials, we have learned about C++ access specifiers such as public, private, and protected.

So far, we have used the `public` keyword in order to inherit a class from a previously-existing base class. However, we can also use the `private` and `protected` keywords to inherit classes. For example,

```
class Animal {
    // code
};

class Dog : private Animal {
    // code
};

class Cat : protected Animal {
    // code
};
```

The various ways we can derive classes are known as **access modes**. These access modes have the following effect:

1. **public:** If a derived class is declared in `public` mode, then the members of the base class are inherited by the derived class just as they are.
2. **private:** In this case, all the members of the base class become `private` members in the derived class.
3. **protected:** The `public` members of the base class become `protected` members in the derived class.

The `private` members of the base class are always `private` in the derived class.

To learn more, visit our C++ public, private, protected inheritance tutorial.

Member Function Overriding in Inheritance

Suppose, base class and derived class have member functions with the same name and arguments.

If we create an object of the derived class and try to access that member function, the member function in the derived class is invoked instead of the one in the base class.

The member function of derived class overrides the member function of base class.

Learn more about Function overriding in C++.

Recommended Reading: *C++ Multiple Inheritance*

Public, Protected and Private Inheritance in C++ Programming

In this tutorial, we will learn to use public, protected and private inheritance in C++ with the help of examples.

In C++ inheritance, we can derive a child class from the base class in different access modes. For example,

```
class Base {
.... ... ....
};

class Derived : public Base {
.... ... ....
};
```

Notice the keyword `public` in the code

```
class Derived : public Base
```

This means that we have created a derived class from the base class in **public mode**. Alternatively, we can also derive classes in **protected** or **private** modes.

These 3 keywords (`public` , `protected` , and `private`) are known as **access specifiers** in C++ inheritance.

---

public, protected and private inheritance in C++

public, **protected,** and **private** inheritance have the following features:

- **public inheritance** makes `public` members of the base class `public` in the derived class, and the `protected` members of the base class remain `protected` in the derived class.
- **protected inheritance** makes the `public` and `protected` members of the base class `protected` in the derived class.
- **private inheritance** makes the `public` and `protected` members of the base class `private` in the derived class.

Note: `private` members of the base class are inaccessible to the derived class.

```
class Base {
    public:
        int x;
    protected:
        int y;
    private:
        int z;
};

class PublicDerived: public Base {
    // x is public
    // y is protected
    // z is not accessible from PublicDerived
};

class ProtectedDerived: protected Base {
    // x is protected
    // y is protected
    // z is not accessible from ProtectedDerived
};

class PrivateDerived: private Base {
    // x is private
    // y is private
    // z is not accessible from PrivateDerived
}
```

Example 1: C++ public Inheritance

```
// C++ program to demonstrate the working of public inheritance

#include <iostream>
using namespace std;

class Base {
   private:
    int pvt = 1;

   protected:
    int prot = 2;

   public:
    int pub = 3;

    // function to access private member
    int getPVT() {
        return pvt;
    }
};

class PublicDerived : public Base {
   public:
    // function to access protected member from Base
    int getProt() {
        return prot;
    }
};

int main() {
    PublicDerived object1;
    cout << "Private = " << object1.getPVT() << endl;
    cout << "Protected = " << object1.getProt() << endl;
    cout << "Public = " << object1.pub << endl;
    return 0;
}
```

Output

```
Private = 1
Protected = 2
Public = 3
```

Here, we have derived `PublicDerived` from `Base` in **public mode**.

As a result, in `PublicDerived` :

- *prot* is inherited as **protected**.
- *pub* and `getPVT()` are inherited as **public**.
- *pvt* is inaccessible since it is **private** in `Base` .

Since **private** and **protected** members are not accessible from `main()` , we need to create public functions `getPVT()` and `getProt()` to access them:

```
// Error: member "Base::pvt" is inaccessible
cout << "Private = " << object1.pvt;

// Error: member "Base::prot" is inaccessible
cout << "Protected = " << object1.prot;
```

Notice that the `getPVT()` function has been defined inside `Base`. But the `getProt()` function has been defined inside `PublicDerived`.

This is because *pvt*, which is **private** in `Base`, is inaccessible to `PublicDerived`.

However, *prot* is accessible to `PublicDerived` due to public inheritance. So, `getProt()` can access the protected variable from within `PublicDerived`.

Accessibility in public Inheritance

Accessibility	private members	protected members	public members
Base Class	Yes	Yes	Yes
Derived Class	No	Yes	Yes

Example 2: C++ protected Inheritance

```
// C++ program to demonstrate the working of protected inheritance

#include <iostream>
using namespace std;

class Base {
   private:
    int pvt = 1;

   protected:
    int prot = 2;

   public:
    int pub = 3;

    // function to access private member
    int getPVT() {
        return pvt;
    }
};

class ProtectedDerived : protected Base {
   public:
    // function to access protected member from Base
    int getProt() {
        return prot;
    }

    // function to access public member from Base
    int getPub() {
        return pub;
    }
};

int main() {
    ProtectedDerived object1;
    cout << "Private cannot be accessed." << endl;
    cout << "Protected = " << object1.getProt() << endl;
    cout << "Public = " << object1.getPub() << endl;
    return 0;
}
```

Output

```
Private cannot be accessed.
Protected = 2
Public = 3
```

Here, we have derived `ProtectedDerived` from `Base` in **protected mode**.

As a result, in `ProtectedDerived` :

- *prot*, *pub* and `getPVT()` are inherited as **protected**.
- `pvt` is inaccessible since it is **private** in `Base` .

As we know, **protected** members cannot be directly accessed from outside the class. As a result, we cannot use `getPVT()` from `ProtectedDerived` .

That is also why we need to create the `getPub()` function in `ProtectedDerived` in order to access the *pub* variable.

```
// Error: member "Base::getPVT()" is inaccessible
cout << "Private = " << object1.getPVT();

// Error: member "Base::pub" is inaccessible
cout << "Public = " << object1.pub;
```

Accessibility in protected Inheritance

Accessibility	private members	protected members	public members
Base Class	Yes	Yes	Yes
Derived Class	No	Yes	Yes (inherited as protected variables)

Example 3: C++ private Inheritance

```
// C++ program to demonstrate the working of private inheritance

#include <iostream>
using namespace std;

class Base {
   private:
    int pvt = 1;

   protected:
    int prot = 2;

   public:
    int pub = 3;

    // function to access private member
    int getPVT() {
        return pvt;
    }
};

class PrivateDerived : private Base {
   public:
    // function to access protected member from Base
    int getProt() {
        return prot;
    }

    // function to access private member
    int getPub() {
        return pub;
    }
};

int main() {
    PrivateDerived object1;
    cout << "Private cannot be accessed." << endl;
    cout << "Protected = " << object1.getProt() << endl;
    cout << "Public = " << object1.getPub() << endl;
    return 0;
}
```

Output

```
Private cannot be accessed.
Protected = 2
Public = 3
```

Here, we have derived `PrivateDerived` from `Base` in **private mode**.

As a result, in `PrivateDerived` :

- *prot*, `pub` and `getPVT()` are inherited as **private**.
- *pvt* is inaccessible since it is **private** in `Base` .

As we know, private members cannot be directly accessed from outside the class. As a result, we cannot use `getPVT()` from `PrivateDerived` .

That is also why we need to create the `getPub()` function in `PrivateDerived` in order to access the *pub* variable.

```
// Error: member "Base::getPVT()" is inaccessible
cout << "Private = " << object1.getPVT();

// Error: member "Base::pub" is inaccessible
cout << "Public = " << object1.pub;
```

Accessibility in private Inheritance

Accessibility	private members	protected members	public members
Base Class	Yes	Yes	Yes
Derived Class	No	Yes (inherited as private variables)	Yes (inherited as private variables)

C++ Function Overriding

programiz.com/cpp-programming/function-overriding

Join our newsletter for the latest updates.

In this tutorial, we will learn about function overriding in C++ with the help of examples.

As we know, inheritance is a feature of OOP that allows us to create derived classes from a base class. The derived classes inherit features of the base class.

Suppose, the same function is defined in both the derived class and the based class. Now if we call this function using the object of the derived class, the function of the derived class is executed.

This is known as **function overriding** in C++. The function in derived class overrides the function in base class.

Example 1: C++ Function Overriding

```
// C++ program to demonstrate function overriding

#include <iostream>
using namespace std;

class Base {
   public:
    void print() {
        cout << "Base Function" << endl;
    }
};

class Derived : public Base {
   public:
    void print() {
        cout << "Derived Function" << endl;
    }
};

int main() {
    Derived derived1;
    derived1.print();
    return 0;
}
```

Output

```
Derived Function
```

Here, the same function `print()` is defined in both `Base` and `Derived` classes.

So, when we call `print()` from the `Derived` object *derived1*, the `print()` from `Derived` is executed by overriding the function in `Base`.

```
class Base {
    public:
      void print() {
          // code
      }
};

class Derived : public Base {
    public:
      void print() {
          // code
      }
};

int main() {
    Derived derived1;
    derived1.print();
    return 0;
}
```

Working of function overriding in C++

As we can see, the function was overridden because we called the function from an object of the `Derived` class.

Had we called the `print()` function from an object of the `Base` class, the function would not have been overridden.

```
// Call function of Base class
Base base1;
base1.print(); // Output: Base Function
```

Access Overridden Function in C++

To access the overridden function of the base class, we use the scope resolution operator `::`.

We can also access the overridden function by using a pointer of the base class to point to an object of the derived class and then calling the function from that pointer.

Example 2: C++ Access Overridden Function to the Base Class

```
// C++ program to access overridden function
// in main() using the scope resolution operator ::

#include <iostream>
using namespace std;

class Base {
   public:
    void print() {
        cout << "Base Function" << endl;
    }
};

class Derived : public Base {
   public:
    void print() {
        cout << "Derived Function" << endl;
    }
};

int main() {
    Derived derived1, derived2;
    derived1.print();

    // access print() function of the Base class
    derived2.Base::print();

    return 0;
}
```

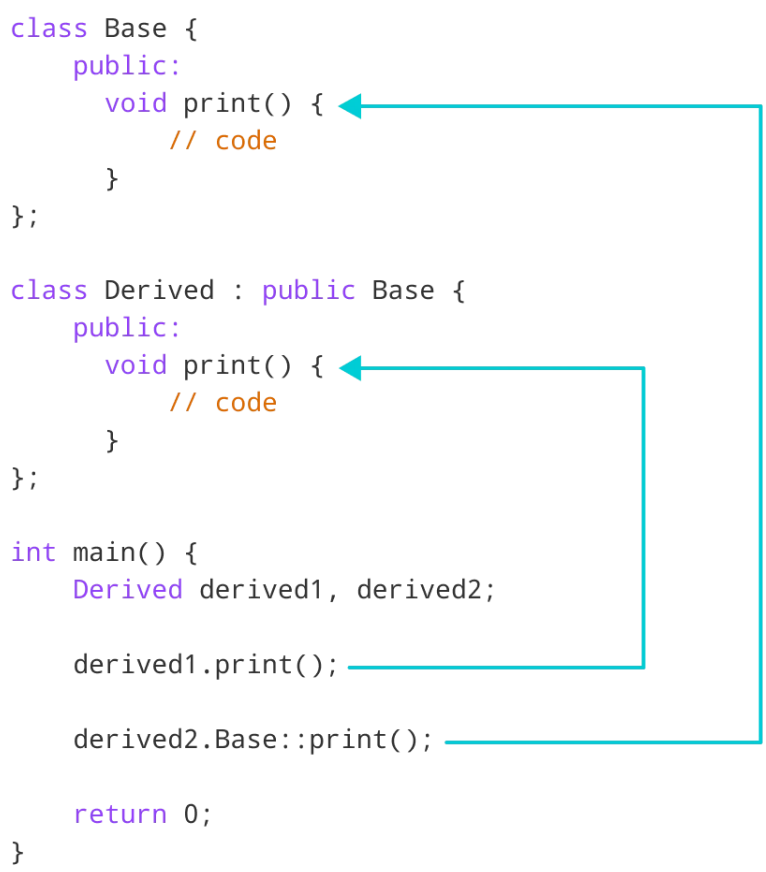
Output

```
Derived Function
Base Function
```

Here, this statement

```
derived2.Base::print();
```

accesses the `print()` function of the Base class.



Access overridden function using object of derived class in C++

Example 3: C++ Call Overridden Function From Derived Class

```
// C++ program to call the overridden function
// from a member function of the derived class

#include <iostream>
using namespace std;

class Base {
   public:
    void print() {
        cout << "Base Function" << endl;
    }
};

class Derived : public Base {
   public:
    void print() {
        cout << "Derived Function" << endl;

        // call overridden function
        Base::print();
    }
};

int main() {
    Derived derived1;
    derived1.print();
    return 0;
}
```

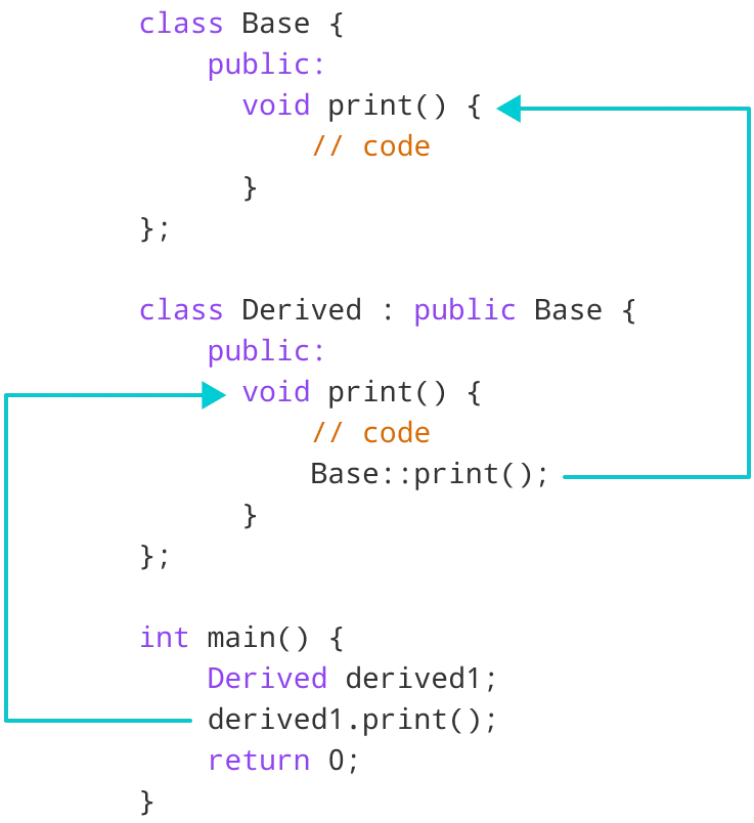
Output

```
Derived Function
Base Function
```

In this program, we have called the overridden function inside the `Derived` class itself.

```
class Derived : public Base {
   public:
    void print() {
        cout << "Derived Function" << endl;
        Base::print();
    }
};
```

Notice the code `Base::print();`, which calls the overridden function inside the `Derived` class.



Access overridden function inside derived class in C++

Example 4: C++ Call Overridden Function Using Pointer

```
// C++ program to access overridden function using pointer
// of Base type that points to an object of Derived class

#include <iostream>
using namespace std;

class Base {
   public:
    void print() {
        cout << "Base Function" << endl;
    }
};

class Derived : public Base {
   public:
    void print() {
        cout << "Derived Function" << endl;
    }
};

int main() {
    Derived derived1;

    // pointer of Base type that points to derived1
    Base* ptr = &derived1;

    // call function of Base class using ptr
    ptr->print();

    return 0;
}
```

Output

```
Base Function
```

In this program, we have created a pointer of `Base` type named *ptr*. This pointer points to the `Derived` object *derived1*.

```
// pointer of Base type that points to derived1
Base* ptr = &derived1;
```

When we call the `print()` function using *ptr*, it calls the overridden function from `Base`.

```
// call function of Base class using ptr
ptr->print();
```

This is because even though *ptr* points to a `Derived` object, it is actually of `Base` type. So, it calls the member function of `Base`.

In order to override the `Base` function instead of accessing it, we need to use virtual functions in the `Base` class.

C++ Multiple, Multilevel and Hierarchical Inheritance

In this tutorial, we will learn about different models of inheritance in C++ programming: Multiple, Multilevel and Hierarchical inheritance with examples.

Inheritance is one of the core feature of an object-oriented programming language. It allows software developers to derive a new class from the existing class. The derived class inherits the features of the base class (existing class).

There are various models of inheritance in C++ programming.

C++ Multilevel Inheritance

In C++ programming, not only you can derive a class from the base class but you can also derive a class from the derived class. This form of inheritance is known as multilevel inheritance.

```
class A {
... .. ...
};
class B: public A {
... .. ...
};
class C: public B {
... ... ...
};
```

Here, class *B* is derived from the base class *A* and the class *C* is derived from the derived class *B*.

Example 1: C++ Multilevel Inheritance

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

class A {
    public:
      void display() {
          cout<<"Base class content.";
      }
};

class B : public A {};

class C : public B {};

int main() {
    C obj;
    obj.display();
    return 0;
}
```

Output

```
Base class content.
```

In this program, class *C* is derived from class *B* (which is derived from base class *A*).

The *obj* object of class *C* is defined in the `main()` function.

When the `display()` function is called, `display()` in class *A* is executed. It's because there is no `display()` function in class *C* and class *B*.

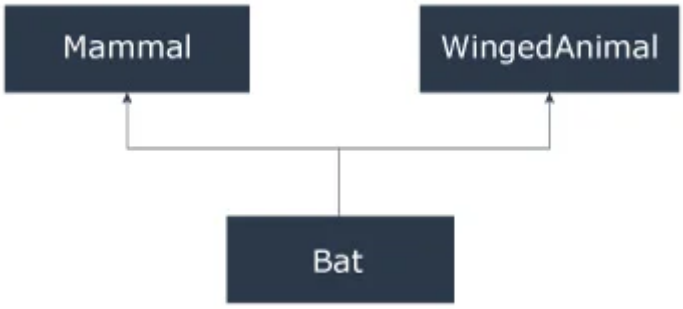
The compiler first looks for the `display()` function in class *C*. Since the function doesn't exist there, it looks for the function in class *B* (as *C* is derived from *B*).

The function also doesn't exist in class *B*, so the compiler looks for it in class *A* (as *B* is derived from *A*).

If `display()` function exists in *C*, the compiler overrides `display()` of class *A* (because of member function overriding).

C++ Multiple Inheritance

In C++ programming, a class can be derived from more than one parent. For example, A class *Bat* is derived from base classes *Mammal* and *WingedAnimal*. It makes sense because bat is a mammal as well as a winged animal.



Multiple Inheritance

Example 2: Multiple Inheritance in C++ Programming

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

class Mammal {
  public:
    Mammal() {
      cout << "Mammals can give direct birth." << endl;
    }
};

class WingedAnimal {
  public:
    WingedAnimal() {
      cout << "Winged animal can flap." << endl;
    }
};

class Bat: public Mammal, public WingedAnimal {};

int main() {
    Bat b1;
    return 0;
}
```

Output

```
Mammals can give direct birth.
Winged animal can flap.
```

Ambiguity in Multiple Inheritance

The most obvious problem with multiple inheritance occurs during function overriding.

Suppose, two base classes have a same function which is not overridden in derived class.

If you try to call the function using the object of the derived class, compiler shows error. It's because compiler doesn't know which function to call. For example,

```
class base1 {
  public:
      void someFunction( ) {....}
};
class base2 {
    void someFunction( ) {....}
};
class derived : public base1, public base2 {};

int main() {
    derived obj;
    obj.someFunction() // Error!
}
```

This problem can be solved using the scope resolution function to specify which function to class either *base1 or base2*

```
int main() {
    obj.base1::someFunction( );  // Function of base1 class is called
    obj.base2::someFunction();   // Function of base2 class is called.
}
```

C++ Hierarchical Inheritance

If more than one class is inherited from the base class, it's known as hierarchical inheritance. In hierarchical inheritance, all features that are common in child classes are included in the base class.

For example, Physics, Chemistry, Biology are derived from Science class. Similarly, Dog, Cat, Horse are derived from Animal class.

Syntax of Hierarchical Inheritance

```
class base_class {
     ... .. ...
}

class first_derived_class: public base_class {
     ... .. ...
}

class second_derived_class: public base_class {
     ... .. ...
}

class third_derived_class: public base_class {
     ... .. ...
}
```

Example 3: Hierarchical Inheritance in C++ Programming

```
// C++ program to demonstrate hierarchical inheritance

#include <iostream>
using namespace std;

// base class
class Animal {
   public:
    void info() {
        cout << "I am an animal." << endl;
    }
};

// derived class 1
class Dog : public Animal {
   public:
    void bark() {
        cout << "I am a Dog. Woof woof." << endl;
    }
};

// derived class 2
class Cat : public Animal {
   public:
    void meow() {
        cout << "I am a Cat. Meow." << endl;
    }
};

int main() {
    // Create object of Dog class
    Dog dog1;
    cout << "Dog Class:" << endl;
    dog1.info();  // Parent Class function
    dog1.bark();

    // Create object of Cat class
    Cat cat1;
    cout << "\nCat Class:" << endl;
    cat1.info();  // Parent Class function
    cat1.meow();

    return 0;
}
```

Output

```
Dog Class:
I am an animal.
I am a Dog. Woof woof.

Cat Class:
I am an animal.
I am a Cat. Meow.
```

Here, both the `Dog` and `Cat` classes are derived from the `Animal` class. As such, both the derived classes can access the `info()` function belonging to the `Animal` class.

C++ friend Function and friend Classes

programiz.com/cpp-programming/friend-function-class

Join our newsletter for the latest updates.

In this tutorial, we will learn to create friend functions and friend classes in C++ with the help of examples.

Data hiding is a fundamental concept of object-oriented programming. It restricts the access of private members from outside of the class.

Similarly, protected members can only be accessed by derived classes and are inaccessible from outside. For example,

```
class MyClass {
    private:
        int member1;
}

int main() {
    MyClass obj;

    // Error! Cannot access private members from here.
    obj.member1 = 5;
}
```

However, there is a feature in C++ called **friend functions** that break this rule and allow us to access member functions from outside the class.

Similarly, there is a **friend class** as well, which we will learn later in this tutorial.

---

friend Function in C++

A **friend function** can access the **private** and **protected** data of a class. We declare a friend function using the `friend` keyword inside the body of the class.

```
class className {
    ... .. ...
    friend returnType functionName(arguments);
    ... .. ...
}
```

Example 1: Working of friend Function

```
// C++ program to demonstrate the working of friend function

#include <iostream>
using namespace std;

class Distance {
    private:
        int meter;

        // friend function
        friend int addFive(Distance);

    public:
        Distance() : meter(0) {}

};

// friend function definition
int addFive(Distance d) {

    //accessing private members from the friend function
    d.meter += 5;
    return d.meter;
}

int main() {
    Distance D;
    cout << "Distance: " << addFive(D);
    return 0;
}
```

Output

```
Distance: 5
```

Here, `addFive()` is a friend function that can access both **private** and **public** data members.

Though this example gives us an idea about the concept of a friend function, it doesn't show any meaningful use.

A more meaningful use would be operating on objects of two different classes. That's when the friend function can be very helpful.

Example 2: Add Members of Two Different Classes

```
// Add members of two different classes using friend functions

#include <iostream>
using namespace std;

// forward declaration
class ClassB;

class ClassA {

    public:
        // constructor to initialize numA to 12
        ClassA() : numA(12) {}

    private:
        int numA;

         // friend function declaration
         friend int add(ClassA, ClassB);
};

class ClassB {

    public:
        // constructor to initialize numB to 1
        ClassB() : numB(1) {}

    private:
        int numB;

        // friend function declaration
        friend int add(ClassA, ClassB);
};

// access members of both classes
int add(ClassA objectA, ClassB objectB) {
    return (objectA.numA + objectB.numB);
}

int main() {
    ClassA objectA;
    ClassB objectB;
    cout << "Sum: " << add(objectA, objectB);
    return 0;
}
```

Output

```
Sum: 13
```

In this program, `ClassA` and `ClassB` have declared `add()` as a friend function. Thus, this function can access **private** data of both classes.

One thing to notice here is the friend function inside `ClassA` is using the `ClassB`. However, we haven't defined `ClassB` at this point.

```
// inside classA
friend int add(ClassA, ClassB);
```

For this to work, we need a forward declaration of `ClassB` in our program.

```
// forward declaration
class ClassB;
```

friend Class in C++

We can also use a friend Class in C++ using the `friend` keyword. For example,

```
class ClassB;

class ClassA {
   // ClassB is a friend class of ClassA
   friend class ClassB;
   ... .. ...
}

class ClassB {
   ... .. ...
}
```

When a class is declared a friend class, all the member functions of the friend class become friend functions.

Since `ClassB` is a friend class, we can access all members of `ClassA` from inside `ClassB`.

However, we cannot access members of `ClassB` from inside `ClassA`. It is because friend relation in C++ is only granted, not taken.

Example 3: C++ friend Class

```
// C++ program to demonstrate the working of friend class

#include <iostream>
using namespace std;

// forward declaration
class ClassB;

class ClassA {
    private:
        int numA;

        // friend class declaration
        friend class ClassB;

    public:
        // constructor to initialize numA to 12
        ClassA() : numA(12) {}
};

class ClassB {
    private:
        int numB;

    public:
        // constructor to initialize numB to 1
        ClassB() : numB(1) {}

    // member function to add numA
    // from ClassA and numB from ClassB
    int add() {
        ClassA objectA;
        return objectA.numA + numB;
    }
};

int main() {
    ClassB objectB;
    cout << "Sum: " << objectB.add();
    return 0;
}
```

Output

```
Sum: 13
```

Here, `ClassB` is a friend class of `ClassA`. So, `ClassB` has access to the members of `classA`.

In `ClassB`, we have created a function `add()` that returns the sum of *numA* and *numB*.

Since `ClassB` is a friend class, we can create objects of `ClassA` inside of `ClassB`.

C++ Virtual Functions

programiz.com/cpp-programming/virtual-functions

Join our newsletter for the latest updates.

In this tutorial, we will learn about C++ virtual function and its use with the help of examples.

A virtual function is a member function in the base class that we expect to redefine in derived classes.

Basically, a virtual function is used in the base class in order to ensure that the function is **overridden**. This especially applies to cases where a pointer of base class points to an object of a derived class.

For example, consider the code below:

```
class Base {
   public:
    void print() {
        // code
    }
};

class Derived : public Base {
   public:
    void print() {
        // code
    }
};
```

Later, if we create a pointer of `Base` type to point to an object of `Derived` class and call the `print()` function, it calls the `print()` function of the `Base` class.

In other words, the member function of `Base` is not overridden.

```
int main() {
    Derived derived1;
    Base* base1 = &derived1;

    // calls function of Base class
    base1->print();

    return 0;
}
```

In order to avoid this, we declare the `print()` function of the `Base` class as virtual by using the `virtual` keyword.

```
class Base {
   public:
    virtual void print() {
        // code
    }
};
```

Virtual functions are an integral part of polymorphism in C++. To learn more, check our tutorial on C++ Polymorphism.

Example 1: C++ virtual Function

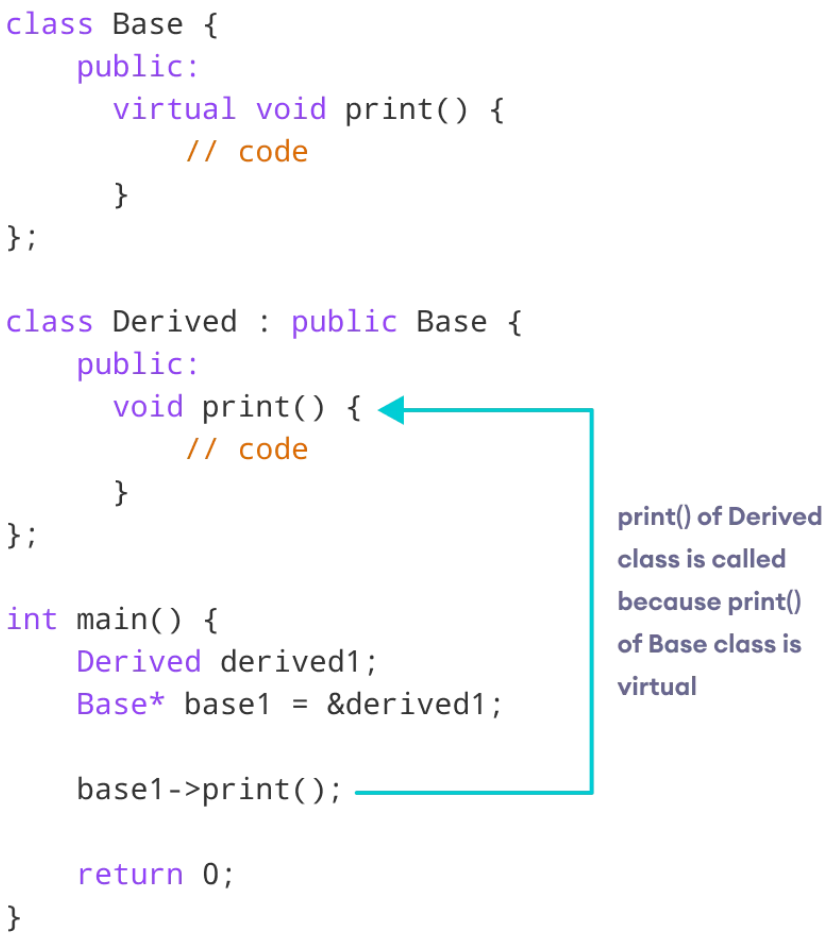
```
#include <iostream>
using namespace std;

class Base {
   public:
    virtual void print() {
        cout << "Base Function" << endl;
    }
};

class Derived : public Base {
   public:
    void print() {
        cout << "Derived Function" << endl;
    }
};

int main() {
    Derived derived1;

    // pointer of Base type that points to derived1
    Base* base1 = &derived1;

    // calls member function of Derived class
    base1->print();

    return 0;
}
```

Output

```
Derived Function
```

Here, we have declared the `print()` function of `Base` as `virtual`.

So, this function is overridden even when we use a pointer of `Base` type that points to the `Derived` object *derived1*.



Working of virtual functions in C++

C++ override Identifier

C++ 11 has given us a new identifier `override` that is very useful to avoid bugs while using virtual functions.

This identifier specifies the member functions of the derived classes that override the member function of the base class.

For example,

```
class Base {
   public:
    virtual void print() {
        // code
    }
};

class Derived : public Base {
   public:
    void print() override {
        // code
    }
};
```

If we use a function prototype in `Derived` class and define that function outside of the class, then we use the following code:

```
class Derived : public Base {
   public:
    // function prototype
    void print() override;
};

// function definition
void Derived::print() {
    // code
}
```

Use of C++ override

When using virtual functions, it is possible to make mistakes while declaring the member functions of the derived classes.

Using the `override` identifier prompts the compiler to display error messages when these mistakes are made.

Otherwise, the program will simply compile but the virtual function will not be overridden.

Some of these possible mistakes are:

- **Functions with incorrect names:** For example, if the virtual function in the base class is named `print()`, but we accidentally name the overriding function in the derived class as `pint()`.
- **Functions with different return types:** If the virtual function is, say, of `void` type but the function in the derived class is of `int` type.
- **Functions with different parameters:** If the parameters of the virtual function and the functions in the derived classes don't match.
- No virtual function is declared in the base class.

Use of C++ Virtual Functions

Suppose we have a base class `Animal` and derived classes `Dog` and `Cat`.

Suppose each class has a data member named *type*. Suppose these variables are initialized through their respective constructors.

```
class Animal {
   private:
    string type;
    ... .. ...
    public:
      Animal(): type("Animal") {}
    ... .. ...
};

class Dog : public Animal {
   private:
    string type;
    ... .. ...
    public:
      Animal(): type("Dog") {}
    ... .. ...
};

class Cat : public Animal {
   private:
    string type;
      ... .. ...
    public:
      Animal(): type("Cat") {}
    ... .. ...
};
```

Now, let us suppose that our program requires us to create two `public` functions for each class:

1. `getType()` to return the value of *type*
2. `print()` to print the value of *type*

We could create both these functions in each class separately and override them, which will be long and tedious.

Or we could make `getType()` **virtual** in the `Animal` class, then create a single, separate `print()` function that accepts a pointer of `Animal` type as its argument. We can then use this single function to override the virtual function.

```
class Animal {
    ... .. ...
   public:
    ... .. ...
    virtual string getType {...}
};

... .. ...
... .. ...

void print(Animal* ani) {
    cout << "Animal: " << ani->getType() << endl;
}
```

This will make the code **shorter**, **cleaner**, and **less repetitive**.

Example 2: C++ virtual Function Demonstration

```
// C++ program to demonstrate the use of virtual function

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

class Animal {
   private:
    string type;

   public:
    // constructor to initialize type
    Animal() : type("Animal") {}

    // declare virtual function
    virtual string getType() {
        return type;
    }
};

class Dog : public Animal {
   private:
    string type;

   public:
    // constructor to initialize type
    Dog() : type("Dog") {}

    string getType() override {
        return type;
    }
};

class Cat : public Animal {
   private:
    string type;

   public:
    // constructor to initialize type
    Cat() : type("Cat") {}

    string getType() override {
        return type;
    }
};

void print(Animal* ani) {
    cout << "Animal: " << ani->getType() << endl;
}

int main() {
    Animal* animal1 = new Animal();
    Animal* dog1 = new Dog();
    Animal* cat1 = new Cat();

    print(animal1);
    print(dog1);
    print(cat1);
```

```
    return 0;
}
```

Output

```
Animal: Animal
Animal: Dog
Animal: Cat
```

Here, we have used the virtual function `getType()` and an `Animal` pointer *ani* in order to avoid repeating the `print()` function in every class.

```
void print(Animal* ani) {
    cout << "Animal: " << ani->getType() << endl;
}
```

In `main()`, we have created 3 `Animal` pointers to dynamically create objects of `Animal`, `Dog` and `Cat` classes.

```
// dynamically create objects using Animal pointers
Animal* animal1 = new Animal();
Animal* dog1 = new Dog();
Animal* cat1 = new Cat();
```

We then call the `print()` function using these pointers:

1. When `print(animal1)` is called, the pointer points to an `Animal` object. So, the virtual function in `Animal` class is executed inside of `print()`.
2. When `print(dog1)` is called, the pointer points to a `Dog` object. So, the virtual function is overridden and the function of `Dog` is executed inside of `print()`.
3. When `print(cat1)` is called, the pointer points to a `Cat` object. So, the virtual function is overridden and the function of `Cat` is executed inside of `print()`.

C++ Templates

programiz.com/cpp-programming/templates

Join our newsletter for the latest updates.

In this article, you'll learn about templates in C++. You'll learn to use the power of templates for generic programming.

Templates are powerful features of C++ which allows you to write generic programs. In simple terms, you can create a single function or a class to work with different data types using templates.

Templates are often used in larger codebase for the purpose of code reusability and flexibility of the programs.

The concept of templates can be used in two different ways:

- Function Templates
- Class Templates

Function Templates

A function template works in a similar to a normal function, with one key difference.

A single function template can work with different data types at once but, a single normal function can only work with one set of data types.

Normally, if you need to perform identical operations on two or more types of data, you use function overloading to create two functions with the required function declaration.

However, a better approach would be to use function templates because you can perform the same task writing less and maintainable code.

How to declare a function template?

A function template starts with the keyword **template** followed by template parameter/s inside `< >` which is followed by function declaration.

```
template <class T>
T someFunction(T arg)
{
   ... .. ...
}
```

In the above code, *T* is a template argument that accepts different data types (int, float), and **class** is a keyword.

You can also use keyword `typename` instead of class in the above example.

When, an argument of a data type is passed to `someFunction( )`, compiler generates a new version of `someFunction()` for the given data type.

Example 1: Function Template to find the largest number

Program to display largest among two numbers using function templates.

```
// If two characters are passed to function template, character with larger ASCII
value is displayed.

#include <iostream>
using namespace std;

// template function
template <class T>
T Large(T n1, T n2)
{
        return (n1 > n2) ? n1 : n2;
}

int main()
{
        int i1, i2;
        float f1, f2;
        char c1, c2;

        cout << "Enter two integers:\n";
        cin >> i1 >> i2;
        cout << Large(i1, i2) <<" is larger." << endl;

        cout << "\nEnter two floating-point numbers:\n";
        cin >> f1 >> f2;
        cout << Large(f1, f2) <<" is larger." << endl;

        cout << "\nEnter two characters:\n";
        cin >> c1 >> c2;
        cout << Large(c1, c2) << " has larger ASCII value.";

        return 0;
}
```

Output

```
Enter two integers:
5
10
10 is larger.

Enter two floating-point numbers:
12.4
10.2
12.4 is larger.

Enter two characters:
z
Z
z has larger ASCII value.
```

In the above program, a function template `Large()` is defined that accepts two arguments *n1* and *n2* of data type `T`. `T` signifies that argument can be of any data type.

`Large()` function returns the largest among the two arguments using a simple conditional operation.

Inside the `main()` function, variables of three different data types: `int`, `float` and `char` are declared. The variables are then passed to the `Large()` function template as normal functions.

During run-time, when an integer is passed to the template function, compiler knows it has to generate a `Large()` function to accept the int arguments and does so.

Similarly, when floating-point data and char data are passed, it knows the argument data types and generates the `Large()` function accordingly.

This way, using only a single function template replaced three identical normal functions and made your code maintainable.

Example 2: Swap Data Using Function Templates

Program to swap data using function templates.

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

template <typename T>
void Swap(T &n1, T &n2)
{
        T temp;
        temp = n1;
        n1 = n2;
        n2 = temp;
}

int main()
{
        int i1 = 1, i2 = 2;
        float f1 = 1.1, f2 = 2.2;
        char c1 = 'a', c2 = 'b';

        cout << "Before passing data to function template.\n";
        cout << "i1 = " << i1 << "\ni2 = " << i2;
        cout << "\nf1 = " << f1 << "\nf2 = " << f2;
        cout << "\nc1 = " << c1 << "\nc2 = " << c2;

        Swap(i1, i2);
        Swap(f1, f2);
        Swap(c1, c2);

        cout << "\n\nAfter passing data to function template.\n";
        cout << "i1 = " << i1 << "\ni2 = " << i2;
        cout << "\nf1 = " << f1 << "\nf2 = " << f2;
        cout << "\nc1 = " << c1 << "\nc2 = " << c2;

        return 0;
}
```

Output

```
Before passing data to function template.
i1 = 1
i2 = 2
f1 = 1.1
f2 = 2.2
c1 = a
c2 = b

After passing data to function template.
i1 = 2
i2 = 1
f1 = 2.2
f2 = 1.1
c1 = b
c2 = a
```

In this program, instead of calling a function by passing a value, a call by reference is issued.

The `Swap()` function template takes two arguments and swaps them by reference.

Class Templates

Like function templates, you can also create class templates for generic class operations.

Sometimes, you need a class implementation that is same for all classes, only the data types used are different.

Normally, you would need to create a different class for each data type OR create different member variables and functions within a single class.

This will unnecessarily bloat your code base and will be hard to maintain, as a change is one class/function should be performed on all classes/functions.

However, class templates make it easy to reuse the same code for all data types.

How to declare a class template?

```
template <class T>
class className
{
   ... .. ...
public:
   T var;
   T someOperation(T arg);
   ... .. ...
};
```

In the above declaration, `T` is the template argument which is a placeholder for the data type used.

Inside the class body, a member variable *var* and a member function `someOperation()` are both of type `T`.

How to create a class template object?

To create a class template object, you need to define the data type inside a `< >` when creation.

```
className<dataType> classObject;
```

For example:

```
className<int> classObject;
className<float> classObject;
className<string> classObject;
```

Example 3: Simple calculator using Class template

Program to add, subtract, multiply and divide two numbers using class template

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

template <class T>
class Calculator
{
private:
        T num1, num2;

public:
        Calculator(T n1, T n2)
        {
                num1 = n1;
                num2 = n2;
        }

        void displayResult()
        {
                cout << "Numbers are: " << num1 << " and " << num2 << "." << endl;
                cout << "Addition is: " << add() << endl;
                cout << "Subtraction is: " << subtract() << endl;
                cout << "Product is: " << multiply() << endl;
                cout << "Division is: " << divide() << endl;
        }

        T add() { return num1 + num2; }

        T subtract() { return num1 - num2; }

        T multiply() { return num1 * num2; }

        T divide() { return num1 / num2; }
};

int main()
{
        Calculator<int> intCalc(2, 1);
        Calculator<float> floatCalc(2.4, 1.2);

        cout << "Int results:" << endl;
        intCalc.displayResult();

        cout << endl << "Float results:" << endl;
        floatCalc.displayResult();

        return 0;
}
```

Output

```
Int results:
Numbers are: 2 and 1.
Addition is: 3
Subtraction is: 1
Product is: 2
Division is: 2

Float results:
Numbers are: 2.4 and 1.2.
Addition is: 3.6
Subtraction is: 1.2
Product is: 2.88
Division is: 2
```

In the above program, a class template `Calculator` is declared.

The class contains two private members of type `T` : *num1* & *num2*, and a constructor to initalize the members.

It also contains public member functions to calculate the addition, subtraction, multiplication and division of the numbers which return the value of data type defined by the user. Likewise, a function `displayResult()` to display the final output to the screen.

In the `main()` function, two different `Calculator` objects `intCalc` and `floatCalc` are created for data types: `int` and `float` respectively. The values are initialized using the constructor.

Notice we use `<int>` and `<float>` while creating the objects. These tell the compiler the data type used for the class creation.

This creates a class definition each for `int` and `float` , which are then used accordingly.

Then, `displayResult()` of both objects is called which performs the Calculator operations and displays the output.