**C++ Function Overloading**

Two or more functions having same name but different argument(s) are known as overloaded functions. In this article, you will learn about function overloading with examples.

[Function](https://www.programiz.com/cpp-programming/function) refers to a segment that groups code to perform a specific task.

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

These functions having different number or type (or both) of parameters are known as overloaded functions. For example:

int test() { }

int test(int a) { }

float test(double a) { }

int test(int a, double b) { }

Here, all 4 functions are overloaded functions because argument(s) passed to these functions are different.

Notice that, the return type of all these 4 functions are not same. Overloaded functions may or may not have different return type but it should have different argument(s).

// Error code

int test(int a) { }

double test(int b){ }

The number and type of arguments passed to these two functions are same even though the return type is different. Hence, the compiler will throw error.

### Example 1: Function Overloading

#include <iostream>

using namespace std;

void display(int);

void display(float);

void display(int, float);

int main() {

int a = 5;

float b = 5.5;

display(a);

display(b);

display(a, b);

return 0;

}

void display(int var) {

cout << "Integer number: " << var << endl;

}

void display(float var) {

cout << "Float number: " << var << endl;

}

void display(int var1, float var2) {

cout << "Integer number: " << var1;

cout << " and float number:" << var2;

}

**Output**

Integer number: 5

Float number: 5.5

Integer number: 5 and float number: 5.5

Here, the display() function is called three times with different type or number of arguments.

The return type of all these functions are same but it's not necessary.

### Example 2: Function Overloading

// Program to compute absolute value

// Works both for integer and float

#include <iostream>

using namespace std;

int absolute(int);

float absolute(float);

int main() {

int a = -5;

float b = 5.5;

cout << "Absolute value of " << a << " = " << absolute(a) << endl;

cout << "Absolute value of " << b << " = " << absolute(b);

return 0;

}

int absolute(int var) {

if (var < 0)

var = -var;

return var;

}

float absolute(float var){

if (var < 0.0)

var = -var;

return var;

}

**Output**

Absolute value of -5 = 5

Absolute value of 5.5 = 5.5

In the above example, two functions absolute() are overloaded.

Both functions take single argument. However, one function takes integer as an argument and other takes float as an argument.

When absolute() function is called with integer as an argument, this function is called:

int absolute(int var) {

if (var < 0)

var = -var;

return var;

}

When absolute() function is called with float as an argument, this function is called:

float absolute(float var){

if (var < 0.0)

var = -var;

return var;

}

# C++ Overloading (Operator and Function)

C++ allows you to specify more than one definition for a **function** name or an **operator** in the same scope, which is called **function overloading** and **operator overloading** respectively.

An overloaded declaration is a declaration that is declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).

When you call an overloaded **function** or **operator**, the compiler determines the most appropriate definition to use, by comparing the argument types you have used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called **overload resolution**.

## Function Overloading in C++

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

Following is the example where same function **print()** is being used to print different data types −

#include <iostream>

using namespace std;

class printData {

public:

void print(int i) {

cout << "Printing int: " << i << endl;

}

void print(double f) {

cout << "Printing float: " << f << endl;

}

void print(char\* c) {

cout << "Printing character: " << c << endl;

}

};

int main(void) {

printData pd;

// Call print to print integer

pd.print(5);

// Call print to print float

pd.print(500.263);

// Call print to print character

pd.print("Hello C++");

return 0;

}

When the above code is compiled and executed, it produces the following result −

Printing int: 5

Printing float: 500.263

Printing character: Hello C++

## Operators Overloading in C++

You can redefine or overload most of the built-in operators available in C++. Thus, a programmer can use operators with user-defined types as well.

Overloaded operators are functions with special names the keyword operator followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

Box operator+(const Box&);

declares the addition operator that can be used to **add** two Box objects and returns final Box object. Most overloaded operators may be defined as ordinary non-member functions or as class member functions. In case we define above function as non-member function of a class then we would have to pass two arguments for each operand as follows −

Box operator+(const Box&, const Box&);

Following is the example to show the concept of operator over loading using a member function. Here an object is passed as an argument whose properties will be accessed using this object, the object which will call this operator can be accessed using **this** operator as explained below −

#include <iostream>

using namespace std;

class Box {

public:

double getVolume(void) {

return length \* breadth \* height;

}

void setLength( double len ) {

length = len;

}

void setBreadth( double bre ) {

breadth = bre;

}

void setHeight( double hei ) {

height = hei;

}

// Overload + operator to add two Box objects.

Box operator+(const Box& b) {

Box box;

box.length = this->length + b.length;

box.breadth = this->breadth + b.breadth;

box.height = this->height + b.height;

return box;

}

private:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

// Main function for the program

int main() {

Box Box1; // Declare Box1 of type Box

Box Box2; // Declare Box2 of type Box

Box Box3; // Declare Box3 of type Box

double volume = 0.0; // Store the volume of a box here

// box 1 specification

Box1.setLength(6.0);

Box1.setBreadth(7.0);

Box1.setHeight(5.0);

// box 2 specification

Box2.setLength(12.0);

Box2.setBreadth(13.0);

Box2.setHeight(10.0);

// volume of box 1

volume = Box1.getVolume();

cout << "Volume of Box1 : " << volume <<endl;

// volume of box 2

volume = Box2.getVolume();

cout << "Volume of Box2 : " << volume <<endl;

// Add two object as follows:

Box3 = Box1 + Box2;

// volume of box 3

volume = Box3.getVolume();

cout << "Volume of Box3 : " << volume <<endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Volume of Box1 : 210

Volume of Box2 : 1560

Volume of Box3 : 5400

## Overloadable/Non-overloadableOperators

Following is the list of operators which can be overloaded −

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| + | - | \* | / | % | ^ |
| & | | | ~ | ! | , | = |
| < | > | <= | >= | ++ | -- |
| << | >> | == | != | && | || |
| += | -= | /= | %= | ^= | &= |
| |= | \*= | <<= | >>= | [] | () |
| -> | ->\* | new | new [] | delete | delete [] |

Following is the list of operators, which can not be overloaded −

|  |  |  |  |
| --- | --- | --- | --- |
| :: | .\* | . | ?: |

## Operator Overloading Examples

Here are various operator overloading examples to help you in understanding the concept.

|  |  |
| --- | --- |
| **Sr.No** | **Operators & Example** |
| 1 | [Unary Operators Overloading](https://www.tutorialspoint.com/cplusplus/unary_operators_overloading.htm) |
| 2 | [Binary Operators Overloading](https://www.tutorialspoint.com/cplusplus/binary_operators_overloading.htm) |
| 3 | [Relational Operators Overloading](https://www.tutorialspoint.com/cplusplus/relational_operators_overloading.htm) |
| 4 | [Input/Output Operators Overloading](https://www.tutorialspoint.com/cplusplus/input_output_operators_overloading.htm) |
| 5 | [++ and -- Operators Overloading](https://www.tutorialspoint.com/cplusplus/increment_decrement_operators_overloading.htm) |
| 6 | [Assignment Operators Overloading](https://www.tutorialspoint.com/cplusplus/assignment_operators_overloading.htm) |
| 7 | [Function call () Operator Overloading](https://www.tutorialspoint.com/cplusplus/function_call_operator_overloading.htm) |
| 8 | [Subscripting [] Operator Overloading](https://www.tutorialspoint.com/cplusplus/subscripting_operator_overloading.htm) |
| 9 | [Class Member Access Operator -> Overloading](https://www.tutorialspoint.com/cplusplus/class_member_access_operator_overloading.htm) |