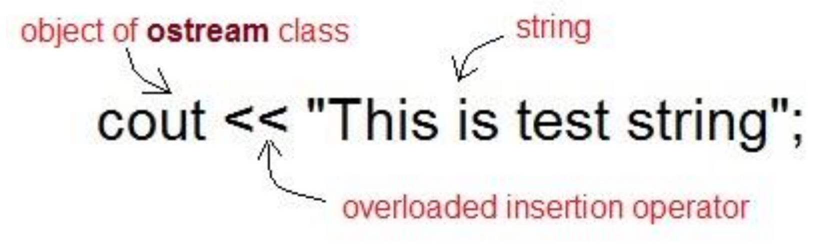
C++:

Operator Overloading in C++

Operator overloading is an important concept in C++. It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it. Overloaded operator is used to perform operation on user-defined data type. For example '+' operator can be overloaded to perform addition on various data types, like for Integer, String(concatenation) etc.

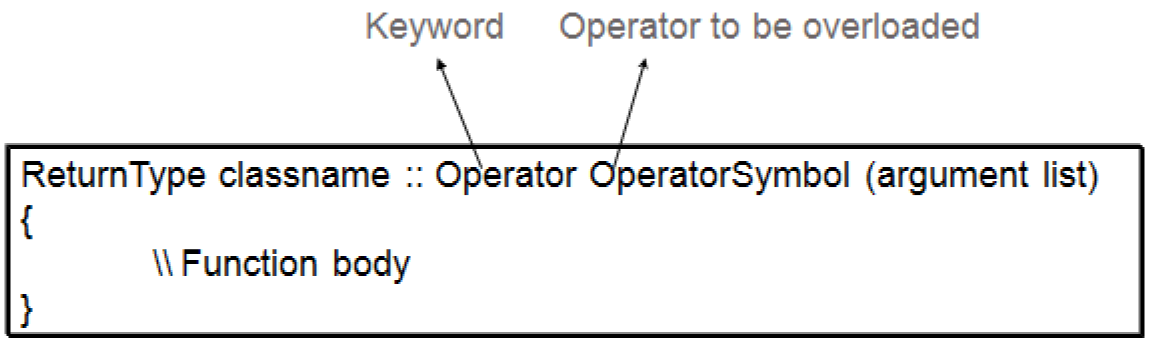


```
cout << "This is test string";
```

Almost any operator can be overloaded in C++. However there are few operator which can not be overloaded. **Operator that are not overloaded** are follows

- scope operator - ::
- sizeof
- member selector - .
- member pointer selector - *
- ternary operator - ?:

Operator Overloading Syntax



```
ReturnType classname :: Operator OperatorSymbol (argument list)
{
        \\ Function body
}
```

Implementing Operator Overloading in C++

Operator overloading can be done by implementing a function which can be :

1. Member Function
2. Non-Member Function
3. Friend Function

Operator overloading function can be a member function if the Left operand is an Object of that class, but if the Left operand is different, then Operator overloading function must be a non-member function.

Operator overloading function can be made friend function if it needs access to the private and protected members of class.

Restrictions on Operator Overloading in C++

Following are some restrictions to be kept in mind while implementing operator overloading.

1. Precedence and Associativity of an operator cannot be changed.
2. Arity (numbers of Operands) cannot be changed. Unary operator remains unary, binary remains binary etc.
3. No new operators can be created, only existing operators can be overloaded.
4. Cannot redefine the meaning of a procedure. You cannot change how integers are added.

Overloading Arithmetic Operator in C++

Arithmetic operator are most commonly used operator in C++. Almost all arithmetic operator can be overloaded to perform arithmetic operation on user-defined data type. In the below example we have overridden the **+** operator, to add to Time(hh:mm:ss) objects.

Example: overloading + Operator to add two Time class object

```
#include <iostream.h>
#include <conio.h>
class Time
{
   int h,m,s;
   public:
   Time()
   {
      h=0, m=0; s=0;
   }
   void setTime();
   void show()
   {
      cout<< h<< ":"<< m<< ":"<< s;
   }
```

```
    //overloading '+' operator
    Time operator+(time);
};

Time Time::operator+(Time t1)    //operator function
{
    Time t;
    int a,b;
    a = s+t1.s;
    t.s = a%60;
    b = (a/60)+m+t1.m;
    t.m = b%60;
    t.h = (b/60)+h+t1.h;
    t.h = t.h%12;
    return t;
}

void time::setTime()
{
    cout << "\n Enter the hour(0-11) ";
    cin >> h;
    cout << "\n Enter the minute(0-59) ";
    cin >> m;
    cout << "\n Enter the second(0-59) ";
    cin >> s;
}

void main()
{
    Time t1,t2,t3;

    cout << "\n Enter the first time ";
    t1.setTime();
    cout << "\n Enter the second time ";
    t2.setTime();
    t3 = t1 + t2;             //adding of two time object using '+' operator
    cout << "\n First time ";
    t1.show();
    cout << "\n Second time ";
    t2.show();
    cout << "\n Sum of times ";
    t3.show();
    getch();
}
```

While normal addition of two numbers return the sumation result. In the case above we have overloaded the `+` operator, to perform addition of two `Time` class objects. We add the **seconds**, **minutes** and **hour** values separately to return the new value of time.

In the `setTime()` funtion we are separately asking the user to enter the values for hour, minute and second, and then we are setting those values to the `Time` class object.

For inputs, `t1` as **01:20:30**(1 hour, 20 minute, 30 seconds) and `t2` as **02:15:25**(2 hour, 15 minute, 25 seconds), the output for the above program will be:

```
1:20:30

2:15:25

3:35:55
```

First two are values of t1 and t2 and the third is the result of their addition.

Example: overloading << Operator to print Class Object

We will now overload the << operator in the Time class,

```
#include<iostream.h>
#include<conio.h>
class Time
{
   int hr, min, sec;
   public:
   // default constructor
   Time()
   {
      hr=0, min=0; sec=0;
   }

   // overloaded constructor
   Time(int h, int m, int s)
   {
      hr=h, min=m; sec=s;
   }

   // overloading '<<' operator
   friend ostream& operator << (ostream &out, Time &tm);
};

// define the overloaded function
ostream& operator << (ostream &out, Time &tm)
{
   out << "Time is: " << tm.hr << " hour : " << tm.min << " min : " << tm.sec << " sec";
   return out;
}

void main()
{
   Time tm(3,15,45);
   cout << tm;
}
```

```
Time is: 3 hour : 15 min : 45 sec
```

This is simplied in languages like Core Java, where all you need to do in a class is override the toString() method of the String class, and you can define how to print the object of that class.

Overloading Relational Operator in C++

You can also overload relational operators like == , != , >= , <= etc. to compare two object of any class.

Let's take a quick example by overloading the `==` operator in the `Time` class to directly compare two objects of `Time` class.

```
class Time
{
    int hr, min, sec;
    public:
    // default constructor
    Time()
    {
        hr=0, min=0; sec=0;
    }

    // overloaded constructor
    Time(int h, int m, int s)
    {
        hr=h, min=m; sec=s;
    }

    //overloading '==' operator
    friend bool operator==(Time &t1, Time &t2);
};

/*
    Defining the overloading operator function
    Here we are simply comparing the hour, minute and
    second values of two different Time objects to compare
    their values
*/
bool operator== (Time &t1, Time &t2)
{
    return ( t1.hr == t2.hr && t1.min == t2.min && t1.sec == t2.sec );
}

void main()
{
    Time t1(3,15,45);
    Time t2(4,15,45);
    if(t1 == t2)
    {
        cout << "Both the time values are equal";
    }
    else
    {
        cout << "Both the time values are not equal";
    }
}
```

```
Both the time values are not equal
```

Operator Precedence and Associativity:

Precedence Rules: The precedence rules specify which operator is evaluated first when two operators with different precedence are adjacent in an expression. For example: x=a+++b This expression can be seen as postfix increment on a and addition with b or prefix increment on b and addtion to a. Such issues are resolved by using precedence rules.

Associativity Rules: The associativity rules specify which operator is evaluated first when two operators with the same precedence are adjacent in an expression. For example: a∗b/c

Operator Precedence: The following table describes the precedence order of the operators mentioned above. Here, the operators with the highest precedence appear at the top and those with the lowest at the bottom. In any given expression, the operators with higher precedence will be evaluated first. LR= Left to Right RL=Right to Left

Category	**Associativity**	**Operator**
Postfix	**LR**	**++ --**
Unary	**RL**	**+ - ! ~ ++ --**
Multiplicative	**LR**	*** / %**
Additive	**LR**	**+ -**
Shift	**LR**	**<< >>**
Relational	**LR**	**< <= > >=**
Equality	**LR**	**== !=**
Bitwise AND	**LR**	**&**
Bitwise XOR	**LR**	**^**
Bitwise OR	**LR**	**|**
Logical AND	**LR**	**&&**
Logical OR	**LR**	**||**
Conditional	**RL**	**?:**
Assignment	**RL**	**= += -= *= /= %= >>= <<= &= ^= |=**

C++ Functions

C++ functions are a group of statements in a single logical unit to perform some specific task.
Along with the main function, a program can have multiple functions that are designed for different operation.
The results of functions can be used throughout the program without concern about the process and the mechanism of the function.

In POP (Procedural Oriented Programming) language like C, programs are divided into different **functions** but in OOP (Object Oriented Programming) approach program is divided into objects where functions are the components of the object.
Generally, C++ function has three parts:

- **Function Prototype**
- **Function Definition**
- **Function Call**

C++ Function Prototype

While writing a program, we can't use a function without specifying its type or without telling the compiler about it.
So before calling a function, we must either declare or define a function.
Thus, declaring a function before calling a function is called **function declaration or prototype** which tells the compiler that at some point of the program we will use the function of the name specified in the prototype.
Syntax

```
return_type function_name(parameter_list);
```

Note: function prototype must end with a semicolon.
Here, return_type is the type of value that the function will return. It can be int, float or any user-defined data type.
function_name means any name that we give to the function. However, it must not resemble any standard keyword of C++.
Finally, parameter_list contains a total number of arguments that need to be passed to the function.
C++ Function Call

Function call means calling the function with a statement. When the program encounters the function call statement the specific function is invoked.
Syntax

```
function_name( argument_list );
```

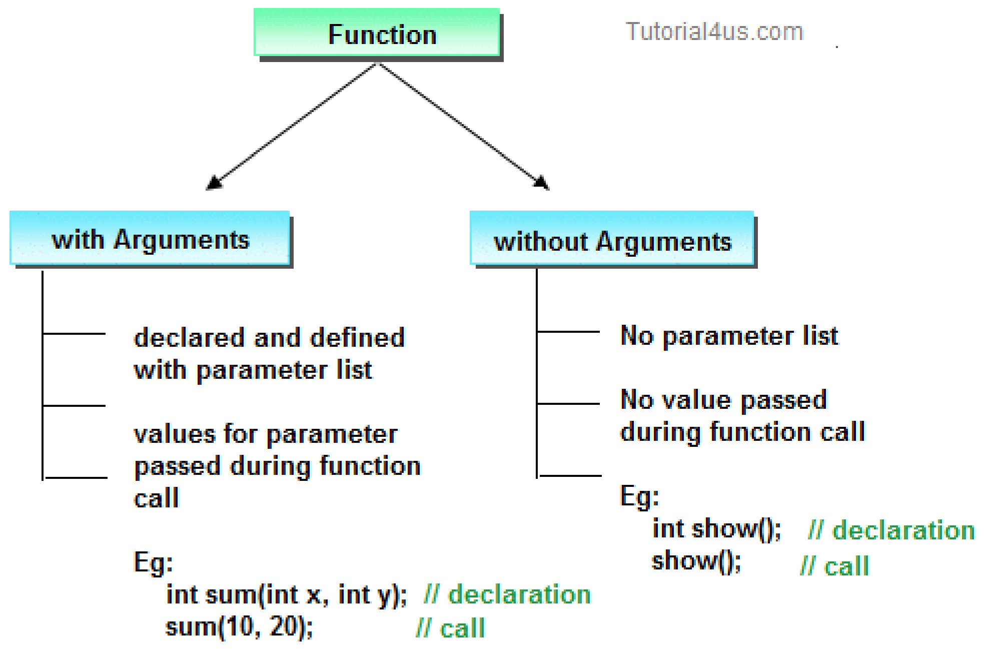
Here, function_name is the name of the called function and argument_list is the comma-separated list of expressions that constitute the arguments.
The syntax is similar to that of prototype except that return_type is not used.
C++ functions definition

Function definition is a part where we define the operation of a function. It consists of the declarator followed by the function body.
Syntax

```
return_type function_name( parameter_list )
{
   function body;
}
```

Call by Value and Call by Reference in C++

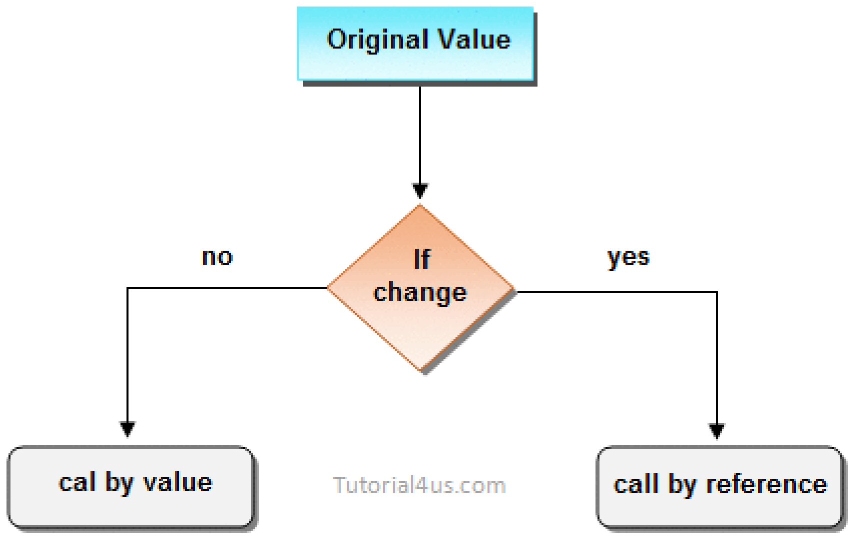
On the basis of arguments there are two types of function are available in C++ language, they are;


Function
Tutorial4us.com
with Arguments
without Arguments
declared and defined with parameter list
values for parameter passed during function call
Eg:
int sum(int x, int y); // declaration
sum(10, 20); // call
No parameter list
No value passed during function call
Eg:
int show(); // declaration
show(); // call

- With argument
- Without argument

If a function takes any arguments, it must declare variables that accept the values as a arguments. These variables are called the formal parameters of the function. There are two ways to pass value or data to function in C++ language which is given below;

- call by value
- call by reference



Call by value

In call by value, **original value can not be changed** or modified. In call by value, when you passed value to the function it is locally stored by the function parameter in stack memory location. If you change the value of function parameter, it is changed for the current function only but it not change the value of variable inside the caller function such as main().

Call by value

```
#include<iostream.h>
#include<conio.h>

void swap(int a, int b)
{
 int temp;
 temp=a;
 a=b;
 b=temp;
}

void main()
{
 int a=100, b=200;
 clrscr();
 swap(a, b);  // passing value to function
 cout<<"Value of a"<<a;
 cout<<"Value of b"<<b;
 getch();
}
```

Output

```
Value of a: 200

Value of b: 100
```

Call by reference

In call by reference, **original value is changed** or modified because we pass reference (address). Here, address of the value is passed in the function, so actual and formal arguments shares the same address space. Hence, any value changed inside the function, is reflected inside as well as outside the function.

Example Call by reference

```
#include<iostream.h>
#include<conio.h>

void swap(int *a, int *b)
{
 int temp;
 temp=*a;
 *a=*b;
 *b=temp;
}

void main()
{
 int a=100, b=200;
 clrscr();
 swap(&a, &b);  // passing value to function
 cout<<"Value of a"<<a;
 cout<<"Value of b"<<b;
 getch();
}
```

Output

```
Value of a: 200

Value of b: 100
```

Difference between call by value and call by reference.

	call by value	call by reference
1	This method copy original value into function as a arguments.	This method copy address of arguments into function as a arguments.
2	Changes made to the parameter inside the function have no effect on the argument.	Changes made to the parameter affect the argument. Because address is used to access the actual argument.
3	Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in same memory location

Friend Function in C++

In C++ a **Friend Function** that is a "friend" of a given class is allowed access to private and protected data in that class.

A function can be made a friend function using keyword friend. Any friend function is preceded with **friend** keyword. The declaration of friend function should be made inside the body of class (can be anywhere inside class either in private or public section) starting with keyword friend.

Why use friend function ?

You do not access private or protected data member of any class, to access private and protected data member of any class you need a friend function.

Syntax

```
class class_name
{
 ......
 friend returntype function_name(arguments);
}
```

Friend class

Similarly like, friend function a class can be made a friend of another class using keyword friend.

Syntax

```
class A
{
  friend class B; // class B is a friend class
  ......
}
class B
{
  ......
}
```

When a class is made a friend class, all the member functions of that class becomes friend function.

If B is declared friend class of A then, all member functions of class B can access private and protected data of class A but, member functions of class A can not private and protected data of class B.

Note: Remember, friendship relation in C++ is always granted not taken.

Example Friend function

In below example you can access private function of class employee by using friend function.

Example

```
#include<iostream.h>
#include<conio.h>

class employee
{
private:
      friend void sal();
};

void sal()
{
int salary=4000;
cout<<"Salary: "<<salary;
}

void main()
{
employee e;
sal();
getch();
}
```

Output

```
Salary: 4000
```

inline function:

C++ **inline** function is powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality.

To inline a function, place the keyword **inline** before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line.

A function definition in a class definition is an inline function definition, even without the use of the **inline** specifier.
Following is an example, which makes use of inline function to return max of two numbers −

```
#include <iostream>

using namespace std;

inline int Max(int x, int y) {
   return (x > y)? x : y;
}
// Main function for the program
int main() {
   cout << "Max (20,10): " << Max(20,10) << endl;
   cout << "Max (0,200): " << Max(0,200) << endl;
   cout << "Max (100,1010): " << Max(100,1010) << endl;
      return 0;
}
OutPut:
Max (20,10): 20
Max (0,200): 200
Max (100,1010): 1010
```

inline Function Example Program

```
#include<iostream.h>
#include<conio.h>

class line {
public:
    inline float mul(float x, float y) {
        return (x * y);
    }
    inline float cube(float x) {
        return (x * x * x);
    }
};

void main() {
    line obj;
    float val1, val2;
    clrscr();
    cout << "Enter two values:";
    cin >> val1>>val2;
    cout << "\nMultiplication value is:" << obj.mul(val1, val2);
    cout << "\n\nCube value is          :" << obj.cube(val1) << "\t" <<
obj.cube(val2);
    getch();
}
```

Output

```
Enter two values: 5  7
Multiplication Value is: 35
Cube Value is: 25 and 343
```

Private Member Function

A function declared inside the class's private section is known as **"private member function"**. A **private member function** is accessible through the only public member function. (Read more: data members and member functions in C++).

Example:

In this example, there is a class named **"Student"**, which has following data members and member functions:

- **Private**
  - **Data members**
    - rNo - to store roll number
    - perc - to store percentage
  - **Member functions**
    - inputOn() - to print a message **"Input start..."** before reading the roll number and percentage using public member function.
    - inputOff() - to print a message **"Input end..."** after reading the roll number and percentage using public member function.
- **Public**
  - **Member functions**
    - read() - to read roll number and percentage of the student

- print() - to print roll number and percentage of the student

Here, inputOn() and inputOff() are the private member functions which are calling inside public member function read().

Program:

```
#include <iostream>
class Student
{		private:
				int rNo;
				float perc;
				//private member functions
				void inputOn(void)
				{
					cout<<"Input start..."<<endl;
				}
				void inputOff(void)
				{
					cout<<"Input end..."<<endl;
				}

		public:		//public member functions
				void read(void)
				{		//calling first member function
					inputOn();
					//read rNo and perc
					cout<<"Enter roll number: ";
					cin>>rNo;
					cout<<"Enter percentage: ";
					cin>>perc;
					//calling second member function
					inputOff();
				}
				void print(void)
				{
					cout<<endl;
					cout<<"Roll Number: "<<rNo<<endl;
					cout<<"Percentage: "<<perc<<"%"<<endl;
				}
};
//Main code
int main()
{		//declaring object of class student
		Student std;
		//reading and printing details of a student
		std.read();
		std.print();
		return 0; }
```

Output

```
Input start...
Enter roll number: 101
Enter percentage: 84.02
Input end...
Roll Number: 101
Percentage: 84.02%
```

Static Keyword in C++

Static variables in C

Static keyword has different meanings when used with different types. We can use static keyword with:

Static Variables : Variables in a function, Variables in a class
Static Members of Class : Class objects and Functions in a class
Static variables in a Function: When a variable is declared as static, space for **it gets allocated for the lifetime of the program**. Even if the function is called multiple times, space for the static variable is allocated only once and the value of variable in the previous call gets carried through the next function call. This is useful for implementing C++ or any other application where previous state of function needs to be stored.

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

void demo()
{
    // static variable
    static int count = 0;
    cout << count << " ";

    // value is updated and
    // will be carried to next
    // function calls
    count++;
}
int main()
{
    for (int i=0; i<5; i++)
        demo();
    return 0;
}
```

Output:

```
0 1 2 3 4
```

Static class members:

A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator **::** to identify which class it belongs to.

```
#include<iostream>
using namespace std;
class GfG
{
   public:
      static void printMsg()// static member function
     {
        cout<<"Welcome to GfG!";
     }
};
 int main()
{
    GfG::printMsg(); }
```

Output:

```
Welcome to GfG!
```

Static Function Members:

By declaring a function member as static, you make it independent of any particular object of the class. A static member function can be called even if no objects of the class exist and the **static** functions are accessed using only the class name and the scope resolution operator **::**.

A static member function can only access static data member, other static member functions and any other functions from outside the class.

Static member functions have a class scope and they do not have access to the **this** pointer of the class. You could use a static member function to determine whether some objects of the class have been created or not.

Passing objects to function

The objects of a class can be passed as arguments to member functions as well as nonmember functions either by value or by reference. When an object is passed by value, a copy of the actual object is created inside the function. This copy is destroyed when the function terminates. Moreover, any changes made to the copy of the object inside the function are not reflected in the actual object. On the other hand, in pass by reference, only a reference to that object (not the entire object) is passed to the function. Thus, the changes made to the object within the function are also reflected in the actual object.

Whenever an object of a class is passed to a member function of the same class, its data members can be accessed inside the function using the object name and the dot operator. However, the data members of the calling object can be directly accessed inside the function without using the object name and the dot operator.

To understand how objects are passed and accessed within a member function, consider this example.

Example: A program to demonstrate passing objects by value to a member function of the same class

```
 #include<iostream.h>
class weight
{
int kilogram;
int gram;
public:
void getdata ();
void putdata ();
void sum_weight (weight,weight) ;
} ;
void weight :: getdata()
{
cout<<"/nKilograms:";
cin>>kilogram;
cout<<"Grams:";
cin>>gram;
}
void weight :: putdata ()
{
cout<<kilogram<<" Kgs. and"<<gram<<" gros.\n";
}
void weight :: sum_weight(weight wl,weight w2)
{
```

```
gram = wl.gram + w2.gram;
kilogram=gram/1000;
gram=gram%1000;
kilogram+=wl.kilogram+w2.kilogram;
}
int main ()
{
weight wl,w2 ,w3;
cout<<"Enter weight in kilograms and grams\n";
cout<<"\n Enter weight #1" ;
wl.getdata();
cout<<" \n Enter weight #2" ;
w2.getdata();
w3.sum_weight(wl,w2);
cout<<"/n Weight #1 = ";
wl.putdata();
cout<<"Weight #2 = ";
w2.putdata();
cout<<"Total Weight = ";
w3.putdata();
return 0;
}
```

The output of the program is

```
Enter weight in kilograms and grams
Enter weight #1
Kilograms: 12
Grams: 560
Enter weight #2
Kilograms: 24
Grams: 850
Weight #1 = 12 Kgs. and 560 gms.
Weight #2 = 24 Kgs. and 850 gms.
Total Weight = 37 Kgs. and 410 gms.
```

Function Overloading in C++

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 is a feature in C++ where two or more functions can have the same name but different parameters.
Function overloading can be considered as an example of polymorphism feature in C++.
Following is a simple C++ example to demonstrate function overloading.

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

void print(int i) {
  cout << " Here is int " << i << endl;
}
void print(double  f) {
  cout << " Here is float " << f << endl;
}
void print(char const *c) {
  cout << " Here is char* " << c << endl;
}

int main() {
  print(10);
  print(10.10);
  print("ten");
  return 0;
}
```

Output:

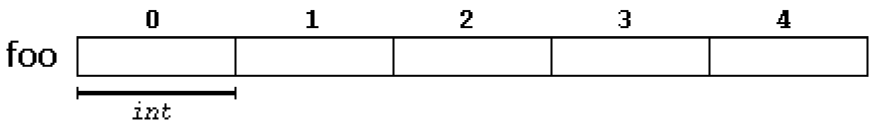
```
Here is int 10

Here is float 10.1

Here is char* ten
```

Arrays

An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.

That means that, for example, five values of type `int` can be declared as an array without having to declare 5 different variables (each with its own identifier). Instead, using an array, the five `int` values are stored in contiguous memory locations, and all five can be accessed using the same identifier, with the proper index.

For example, an array containing 5 integer values of type `int` called `foo` could be represented as:



where each blank panel represents an element of the array. In this case, these are values of type `int`. These elements are numbered from 0 to 4, being 0 the first and 4 the last; In C++, the first element in an array is always numbered with a zero (not a one), no matter its length.

Like a regular variable, an array must be declared before it is used. A typical declaration for an array in C++ is:

```
type name [elements];
```

where `type` is a valid type (such as `int`, `float`...), `name` is a valid identifier and the `elements` field (which is always enclosed in square brackets `[]`), specifies the length of the array in terms of the number of elements.

Therefore, the `foo` array, with five elements of type `int`, can be declared as:

```
int foo [5];
```

NOTE: The `elements` field within square brackets `[]`, representing the number of elements in the array, must be a *constant expression*, since arrays are blocks of static memory whose size must be determined at compile time, before the program runs.

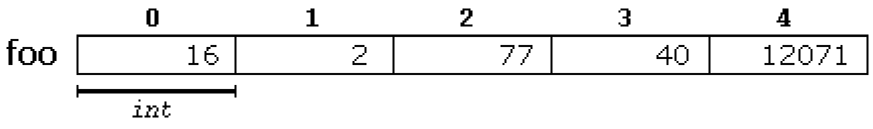
Initializing arrays

By default, regular arrays of *local scope* (for example, those declared within a function) are left uninitialized. This means that none of its elements are set to any particular value; their contents are undetermined at the point the array is declared.

But the elements in an array can be explicitly initialized to specific values when it is declared, by enclosing those initial values in braces {}. For example:

```
int foo [5] = { 16, 2, 77, 40, 12071 };
```

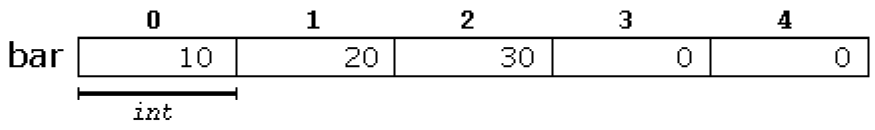
This statement declares an array that can be represented like this:



The number of values between braces {} shall not be greater than the number of elements in the array. For example, in the example above, foo was declared having 5 elements (as specified by the number enclosed in square brackets, []), and the braces {} contained exactly 5 values, one for each element. If declared with less, the remaining elements are set to their default values (which for fundamental types, means they are filled with zeroes). For example:

```
int bar [5] = { 10, 20, 30 };
```

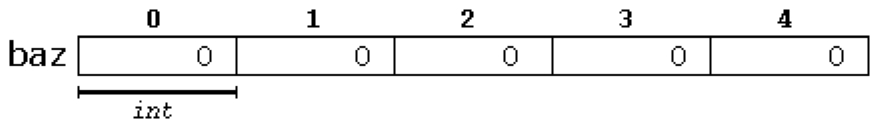
Will create an array like this:



The initializer can even have no values, just the braces:

```
int baz [5] = { };
```

This creates an array of five int values, each initialized with a value of zero:



When an initialization of values is provided for an array, C++ allows the possibility of leaving the square brackets empty []. In this case, the compiler will assume automatically a size for the array that matches the number of values included between the braces {}:

```
int foo [] = { 16, 2, 77, 40, 12071 };
```

After this declaration, array foo would be 5 int long, since we have provided 5 initialization values. Finally, the evolution of C++ has led to the adoption of *universal initialization* also for arrays. Therefore, there is no longer need for the equal sign between the declaration and the initializer. Both these statements are equivalent:

```
int foo[] = { 10, 20, 30 };
int foo[] { 10, 20, 30 };
```

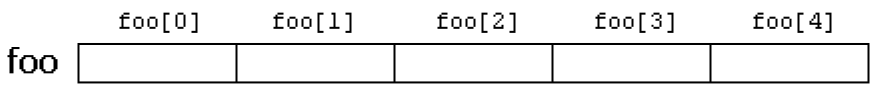
Static arrays, and those declared directly in a namespace (outside any function), are always initialized. If no explicit initializer is specified, all the elements are default-initialized (with zeroes, for fundamental types).

Accessing the values of an array

The values of any of the elements in an array can be accessed just like the value of a regular variable of the same type. The syntax is:

`name[index]`

Following the previous examples in which foo had 5 elements and each of those elements was of type int, the name which can be used to refer to each element is the following:



For example, the following statement stores the value 75 in the third element of `foo`:

```
foo [2] = 75;
```

and, for example, the following copies the value of the third element of `foo` to a variable called `x`:

```
x = foo[2];
```

Therefore, the expression `foo[2]` is itself a variable of type `int`.
Notice that the third element of `foo` is specified `foo[2]`, since the first one is `foo[0]`, the second one is `foo[1]`, and therefore, the third one is `foo[2]`. By this same reason, its last element is `foo[4]`. Therefore, if we write `foo[5]`, we would be accessing the sixth element of `foo`, and therefore actually exceeding the size of the array.
In C++, it is syntactically correct to exceed the valid range of indices for an array. This can create problems, since accessing out-of-range elements do not cause errors on compilation, but can cause errors on runtime. The reason for this being allowed will be seen in a later chapter when pointers are introduced.
At this point, it is important to be able to clearly distinguish between the two uses that brackets `[]` have related to arrays. They perform two different tasks: one is to specify the size of arrays when they are declared; and the second one is to specify indices for concrete array elements when they are accessed. Do not confuse these two possible uses of brackets `[]` with arrays.

```
int foo[5];         // declaration of a new array
foo[2] = 75;        // access to an element of the array.
```

The main difference is that the declaration is preceded by the type of the elements, while the access is not.
Some other valid operations with arrays:

```
foo[0] = a;
foo[a] = 75;
b = foo [a+2];
foo[foo[a]] = foo[2] + 5;
```

For example:

```
// arrays example
#include <iostream>
using namespace std;

int foo [] = {16, 2, 77, 40, 12071};
int n, result=0;

int main ()
{
  for ( n=0 ; n<5 ; ++n )
  {
    result += foo[n];
```

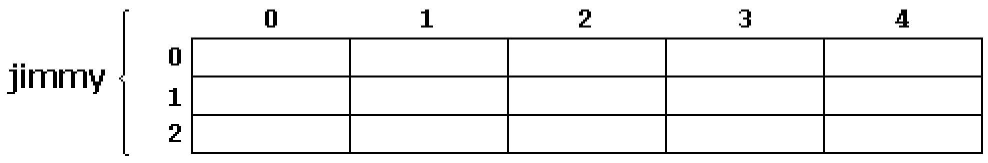
```
12206
```

Edit & Run

```
  }
  cout << result;
  return 0;
}
```

Multidimensional arrays

Multidimensional arrays can be described as "arrays of arrays". For example, a bidimensional array can be imagined as a two-dimensional table made of elements, all of them of a same uniform data type.

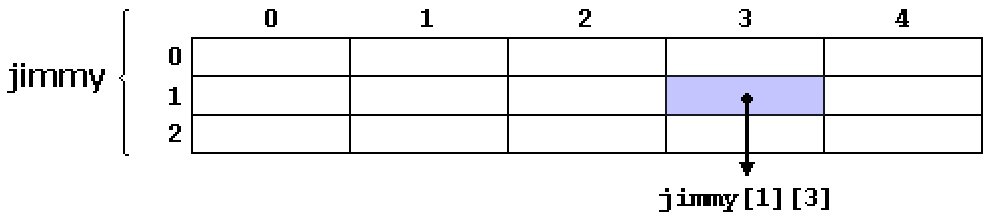


`jimmy` represents a bidimensional array of 3 per 5 elements of type `int`. The C++ syntax for this is:

```
int jimmy [3][5];
```

and, for example, the way to reference the second element vertically and fourth horizontally in an expression would be:

```
jimmy[1][3]
```



(remember that array indices always begin with zero).

Multidimensional arrays are not limited to two indices (i.e., two dimensions). They can contain as many indices as needed. Although be careful: the amount of memory needed for an array increases exponentially with each dimension. For example:

```
char century [100][365][24][60][60];
```

declares an array with an element of type `char` for each second in a century. This amounts to more than 3 billion `char`! So this declaration would consume more than 3 gigabytes of memory!

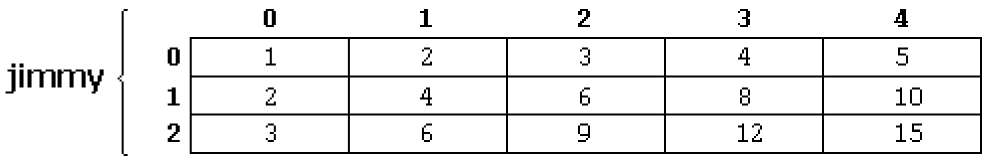
At the end, multidimensional arrays are just an abstraction for programmers, since the same results can be achieved with a simple array, by multiplying its indices:

```
int jimmy [3][5];   // is equivalent to
int jimmy [15];     // (3 * 5 = 15)
```

With the only difference that with multidimensional arrays, the compiler automatically remembers the depth of each imaginary dimension. The following two pieces of code produce the exact same result, but one uses a bidimensional array while the other uses a simple array:

multidimensional array	**pseudo-multidimensional array**
`#define WIDTH 5`	`#define WIDTH 5`
`#define HEIGHT 3`	`#define HEIGHT 3`
`int jimmy [HEIGHT][WIDTH];`	`int jimmy [HEIGHT * WIDTH];`
`int n,m;`	`int n,m;`
`int main ()`	`int main ()`
`{`	`{`
`  for (n=0; n<HEIGHT; n++)`	`  for (n=0; n<HEIGHT; n++)`
`    for (m=0; m<WIDTH; m++)`	`    for (m=0; m<WIDTH; m++)`
`    {`	`    {`
`      jimmy[n][m]=(n+1)*(m+1);`	`      jimmy[n*WIDTH+m]=(n+1)*(m+1);`
`    }`	`    }`
`}`	`}`

None of the two code snippets above produce any output on the screen, but both assign values to the memory block called jimmy in the following way:



jimmy

	0	1	2	3	4
0	1	2	3	4	5
1	2	4	6	8	10
2	3	6	9	12	15

Note that the code uses defined constants for the width and height, instead of using directly their numerical values. This gives the code a better readability, and allows changes in the code to be made easily in one place.

Arrays as parameters

At some point, we may need to pass an array to a function as a parameter. In C++, it is not possible to pass the entire block of memory represented by an array to a function directly as an argument. But what can be passed instead is its address. In practice, this has almost the same effect, and it is a much faster and more efficient operation.

To accept an array as parameter for a function, the parameters can be declared as the array type, but with empty brackets, omitting the actual size of the array. For example:

```
void procedure (int arg[])
```

This function accepts a parameter of type "array of `int`" called `arg`. In order to pass to this function an array declared as:

```
int myarray [40];
```

it would be enough to write a call like this:

```
procedure (myarray);
```

Here you have a complete example:

```
// arrays as parameters
#include <iostream>
using namespace std;

void printarray (int arg[], int
length) {
  for (int n=0; n<length; ++n)
    cout << arg[n] << ' ';
  cout << '\n';
}

int main ()
{
  int firstarray[] = {5, 10, 15};
  int secondarray[] = {2, 4, 6, 8,
10};
  printarray (firstarray,3);
  printarray (secondarray,5);
}
```

```
5 10 15
2 4 6 8 10
```

Edit & Run

In the code above, the first parameter (`int arg[]`) accepts any array whose elements are of type `int`, whatever its length. For that reason, we have included a second parameter that tells the function the length of each array that we pass to it as its first parameter. This allows the for loop that prints out the array to know the range to iterate in the array passed, without going out of range.
In a function declaration, it is also possible to include multidimensional arrays. The format for a tridimensional array parameter is:

```
base_type[][depth][depth]
```

For example, a function with a multidimensional array as argument could be:

```
void procedure (int myarray[][3][4])
```

Notice that the first brackets `[]` are left empty, while the following ones specify sizes for their respective dimensions. This is necessary in order for the compiler to be able to determine the depth of each additional dimension.

In a way, passing an array as argument always loses a dimension. The reason behind is that, for historical reasons, arrays cannot be directly copied, and thus what is really passed is a pointer. This is a common source of errors for novice programmers. Although a clear understanding of pointers, explained in a coming chapter, helps a lot.

Library arrays

The arrays explained above are directly implemented as a language feature, inherited from the C language. They are a great feature, but by restricting its copy and easily decay into pointers, they probably suffer from an excess of optimization.

To overcome some of these issues with language built-in arrays, C++ provides an alternative array type as a standard container. It is a type template (a class template, in fact) defined in header `<array>`.

Containers are a library feature that falls out of the scope of this tutorial, and thus the class will not be explained in detail here. Suffice it to say that they operate in a similar way to built-in arrays, except that they allow being copied (an actually expensive operation that copies the entire block of memory, and thus to use with care) and decay into pointers only when explicitly told to do so (by means of its member `data`).

Just as an example, these are two versions of the same example using the language built-in array described in this chapter, and the container in the library:

language built-in array	**container library array**
`#include <iostream>`	`#include <iostream>`
	`#include <array>`
`using namespace std;`	`using namespace std;`
`int main()`	`int main()`
`{`	`{`
`  int myarray[3] = {10,20,30};`	`  array<int,3> myarray {10,20,30};`
`  for (int i=0; i<3; ++i)`	`  for (int i=0; i<myarray.size(); ++i)`
`    ++myarray[i];`	`    ++myarray[i];`
`  for (int elem : myarray)`	`  for (int elem : myarray)`
`    cout << elem << '\n';`	`    cout << elem << '\n';`
`}`	`}`

As you can see, both kinds of arrays use the same syntax to access its elements: `myarray[i]`. Other than that, the main differences lay on the declaration of the array, and the inclusion of an additional header for the *library array*. Notice also how it is easy to access the size of the *library array*.