

C++ Language Specification

CS201- Data Structures

Difference between C and C++

C vs C++

- C is a procedural programming language and does not support classes and objects (therefore no support for polymorphism, encapsulation, and inheritance).
- While C++ is a combination of both procedural and object oriented programming language;
- C is a subset of C++.
- C++ can be called a hybrid language.

Comments in C++

- Comments are portions of the code ignored by the compiler which allow the user to make simple notes in the relevant areas of the source code.
- Comments are indicated with two slashes (//).
- Longer block comments are enclosed between /* and */.

Data Types

- While writing program in any language, you need to use various variables to store various information.
- Variables are nothing but reserved memory locations to store values.
- Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.

Primitive Built-in Types

Type	Keyword
Boolean	bool
Character	char
Integer	int
Floating point	float
Double floating point	double
Valueless	void
Wide character	wchar_t

Primitive Built-in Types

Type	Typical Bit Width	Typical Range
char	1byte	-127 to 127 or 0 to 255
unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127
int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4bytes	-2147483648 to 2147483647
short int	2bytes	-32768 to 32767
unsigned short int	Range	0 to 65,535
signed short int	Range	-32768 to 32767
long int	4bytes	-2,147,483,648 to 2,147,483,647
signed long int	4bytes	same as long int
unsigned long int	4bytes	0 to 4,294,967,295
float	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	8bytes	+/- 1.7e +/- 308 (~15 digits)
wchar_t	2 or 4 bytes	1 wide character

typedef Declarations

- You can create a new name for an existing type using typedef.
- For example, the following tells the compiler that feet is another name for int.
 - typedef int feet;
- Following creates an integer variable called distance
 - feet distance;

```
typedef type newname;
```


Enumerated Types

- An enumerated type declares an optional type name and a set of zero or more identifiers that can be used as values of the type.
- An enumeration is a user-defined data type that consists of integral constants.
- Creating an enumeration requires the use of the keyword **enum**.
 - `enum color { red, green, blue } c;`
 - `c = blue;`
- By default, the values are {0,1,2}
- `enum color { red, green = 5, blue };`

Enumerated Types - Example

```
#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

Identifiers

- A **C++ identifier** is a name used to identify a variable, function, class, module, or any other user-defined item.
- An **identifier** starts with a letter A to Z or a to z or an underscore (`_`) followed by zero or more letters, underscores, and digits (0 to 9).
- C++ does not allow punctuation characters such as `@`, `$`, and `%` within identifiers.

Rules for Identifiers

- It must start with a letter of the alphabet or an underscore—not a number.
- An identifier cannot be a keyword.
- It should not include a white space.
- Identifiers are case sensitive.
- It cannot have two consecutive underscores.
- It has to be declared before it is referred.
- Two identifiers cannot have the same name.

Expression

- An expression is "a sequence of operators and operands that specifies a computation" (that's the definition given in the C++ standard).
- A combination of variables, constants and operators that represents a computation forms an expression.

Arithmetic Operators

- Assume variable A holds 10 and variable B holds 20, then

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increment operator ↗ , increases integer value by one	A++ will give 11
--	Decrement operator ↗ , decreases integer value by one	A-- will give 9

Relational Operators

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Logical Operators

- Assume variable A holds 1 and variable B holds 0, then;

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	(A && B) is false.
	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.	(A B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	!(A && B) is true.

Bitwise Operators

- Bitwise operator works on bits and perform bit-by-bit operation.
- The truth tables for $\&$, $|$, and \wedge are as follows;

p	q	p & q	p q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Bitwise Operators

- Assume if A = 60; and B = 13; now in binary format they will be as follows;

- A = 0011 1100
- B = 0000 1101

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
	Binary OR Operator copies a bit if it exists in either operand.	(A B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

Assignment Operators

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand.	$C = A + B$ will assign value of $A + B$ into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand.	$C += A$ is equivalent to $C = C + A$
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand.	$C -= A$ is equivalent to $C = C - A$
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand.	$C *= A$ is equivalent to $C = C * A$
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand.	$C /= A$ is equivalent to $C = C / A$

Assignment Operators

<code>%=</code>	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand.	<code>C %= A</code> is equivalent to <code>C = C % A</code>
<code><<=</code>	Left shift AND assignment operator.	<code>C <<= 2</code> is same as <code>C = C << 2</code>
<code>>>=</code>	Right shift AND assignment operator.	<code>C >>= 2</code> is same as <code>C = C >> 2</code>
<code>&=</code>	Bitwise AND assignment operator.	<code>C &= 2</code> is same as <code>C = C & 2</code>
<code>^=</code>	Bitwise exclusive OR and assignment operator.	<code>C ^= 2</code> is same as <code>C = C ^ 2</code>
<code> =</code>	Bitwise inclusive OR and assignment operator.	<code>C = 2</code> is same as <code>C = C 2</code>

Operators Precedence in C++

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

Selection

- Selection statements: if and else. Here, condition is the expression that is being evaluated. If this condition is true, statement is executed. If it is false, statement is not executed (it is simply ignored), and the program continues right after the entire selection statement.

```
if (testExpression)
{
    // statements
}
```

If...else Statement

How if...else statement works?

Test expression is true

```
int test = 5;

if (test < 10)
{
    // codes
}
else
{
    // codes
}

// codes after if...else
```

Test expression is false

```
int test = 5;

if (test > 10)
{
    // codes
}
else
{
    // codes
}

// codes after if...else
```

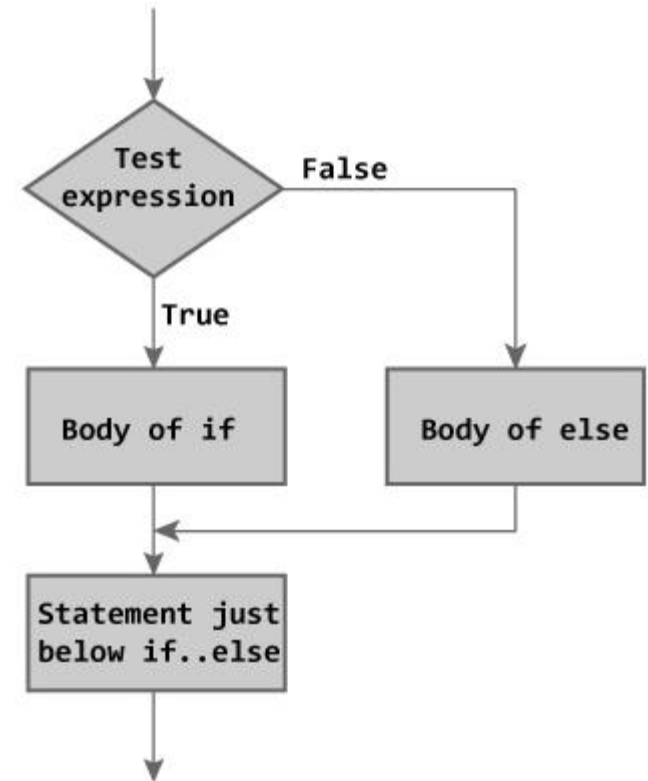


Figure: Flowchart of if...else Statement

If...else Statement - Example

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

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


Nested if...else

```
if (testExpression1)
{
    // statements to be executed if testExpression1 is true
}
else if(testExpression2)
{
    // statements to be executed if testExpression1 is false and testExpression2
}
else if (testExpression 3)
{
    // statements to be executed if testExpression1 and testExpression2 is false
}
.
.
else
{
    // statements to be executed if all test expressions are false
}
```

Nested if...else - Example

```
// 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

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

Conditional/Ternary Operator ?:

A ternary operator operates on 3 operands which can be used instead of a `if...else` statement.

Consider this code:

```
if ( a < b ) {  
    a = b;  
}  
else {  
    a = -b;  
}
```

You can replace the above code with:

```
a = (a < b) ? b : -b;
```

The ternary operator is more readable than a `if...else` statement for short conditions.

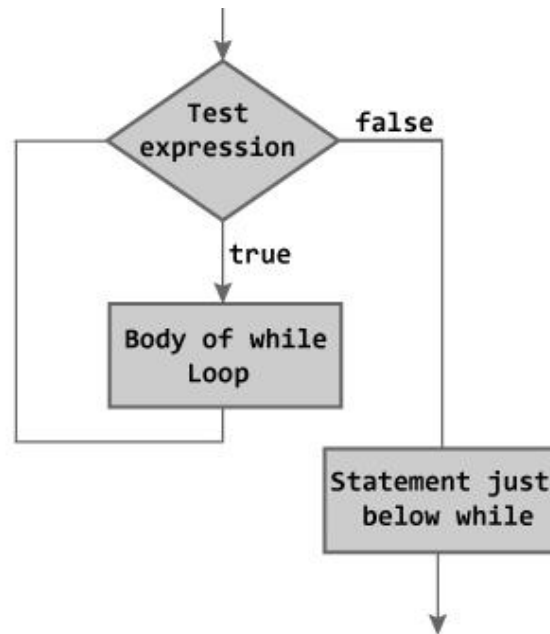
Repetition

- Two types of repetition structures: pretest and posttest loops.
- Types of pretest loop:
 - 1. while
 - 2. for
- Types of posttest loop:
 - 1. do...while

(1) While Loop

How while loop works?

- The while loop evaluates the test expression.
- If the test expression is true, codes inside the body of while loop is evaluated.
- Then, the test expression is evaluated again. This process goes on until the test expression is false.
- When the test expression is false, while loop is terminated.



```
while (testExpression)
{
    // codes
}
```

Figure: Flowchart of while Loop

While Loop - Example

```
// C++ Program to compute factorial of a number
// Factorial of n = 1*2*3...*n

#include <iostream>
using namespace std;

int main()
{
    int number, i = 1, factorial = 1;

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

    while ( i <= number) {
        factorial *= i;      //factorial = factorial * i;
        ++i;
    }

    cout<<"Factorial of "<< number <<" = "<< factorial;
    return 0;
}
```

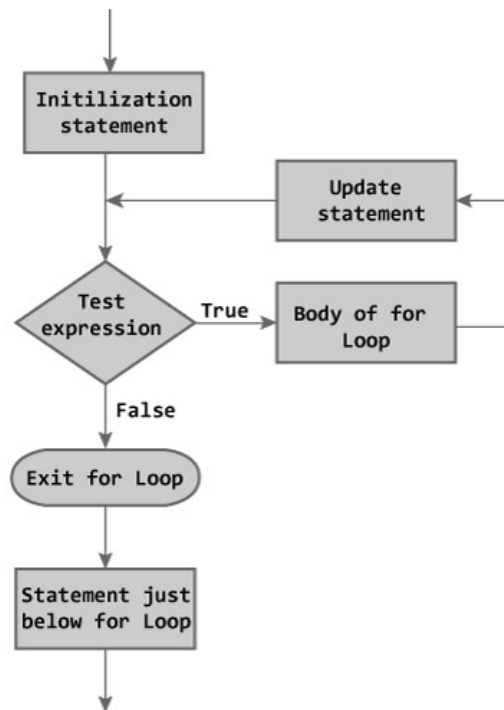
Output

```
Enter a positive integer: 4
Factorial of 4 = 24
```

(2) For Loop

How for loop works?

1. The initialization statement is executed only once at the beginning.
2. Then, the test expression is evaluated.
3. If the test expression is false, for loop is terminated. But if the test expression is true, codes inside body of `for` loop is executed and update expression is updated.
4. Again, the test expression is evaluated and this process repeats until the test expression is false.



```
for(initializationStatement; testExpression; updateStatement) {  
    // codes  
}
```

Figure: Flowchart of for Loop

For Loop - Example

```
// C++ Program to find factorial of a number
// Factorial on n = 1*2*3*...*n

#include <iostream>
using namespace std;

int main()
{
    int i, n, factorial = 1;

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

    for (i = 1; i <= n; ++i) {
        factorial *= i;    // factorial = factorial * i;
    }

    cout<< "Factorial of "<<n<<" = "<<factorial;
    return 0;
}
```

Output

```
Enter a positive integer: 5
Factorial of 5 = 120
```


(3) do...while Loop

How do...while loop works?

- The codes inside the body of loop is executed at least once. Then, only the test expression is checked.
- If the test expression is true, the body of loop is executed. This process continues until the test expression becomes false.
- When the test expression is false, do...while loop is terminated.

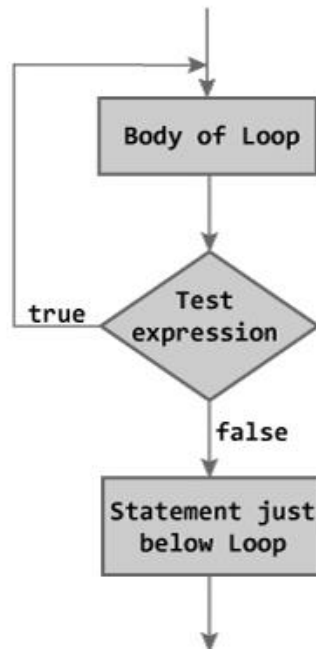


Figure: Flowchart of do...while Loop

```
do {  
    // codes;  
}  
while (testExpression);
```

do...while Loop - Example

```
// C++ program to add numbers until user enters 0

#include <iostream>
using namespace std;

int main()
{
    float number, sum = 0.0;

    do {
        cout<<"Enter a number: ";
        cin>>number;
        sum += number;
    }
    while(number != 0.0);

    cout<<"Total sum = "<<sum;

    return 0;
}
```

Output

```
Enter a number: 2
Enter a number: 3
Enter a number: 4
Enter a number: -4
Enter a number: 2
Enter a number: 4.4
Enter a number: 2
Enter a number: 0
```

Functions

- Function refers to a segment that groups code to perform a specific task.
- There are two types of function:
 - (1). Library Function: are the built-in function in C++ programming and invoked directly.
 - (2). User-defined Function: programmer to define their own function to perform a specific task and that group of code is given a name(identifier).

Library Function - Example

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
    double number, squareRoot;
    cout << "Enter a number: ";
    cin >> number;

    // sqrt() is a library function to calculate square root
    squareRoot = sqrt(number);
    cout << "Square root of " << number << " = " << squareRoot;
    return 0;
}
```

Output

```
Enter a number: 26
Square root of 26 = 5.09902
```

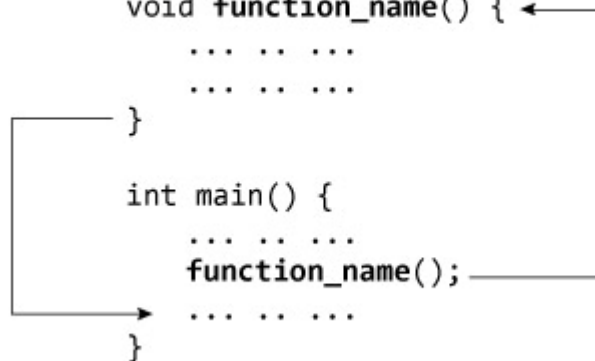
User-defined Function

- C++ allows programmer to define their own function.
- When the function is invoked from any part of program, it all executes the codes defined in the body of function.

```
#include <iostream>

void function_name() {
    ... ..
    ... ..
}

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



The diagram illustrates the execution flow of a user-defined function. It shows two code blocks: a function definition `void function_name() { }` and a `main()` function. An arrow originates from the `function_name();` line within the `main()` function and points to the opening curly brace of the `function_name()` definition. Another arrow originates from the closing curly brace of the `function_name()` definition and points back to the line immediately following `function_name();` in the `main()` function, indicating the return path.

User Defined Function - Example

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

// Function prototype (declaration)
int add(int, int);

int main()
{
    int num1, num2, sum;
    cout<<"Enters two numbers to add: ";
    cin >> num1 >> num2;

    // Function call
    sum = add(num1, num2);
    cout << "Sum = " << sum;
    return 0;
}

// Function definition
int add(int a, int b)
{
    int add;
    add = a + b;

    // Return statement
    return add;
}
```

Output

```
Enters two integers: 8
-4
Sum = 4
```

User-defined Function

- 1. Function prototype (declaration): write it before the main function, otherwise compiler will throw error.
- 2. Function Call: user-defined function needs to be invoked(called) in main function.
- 3. Function Definition: The function itself is referred as function definition.

Passing Arguments to Function

Notes on passing arguments

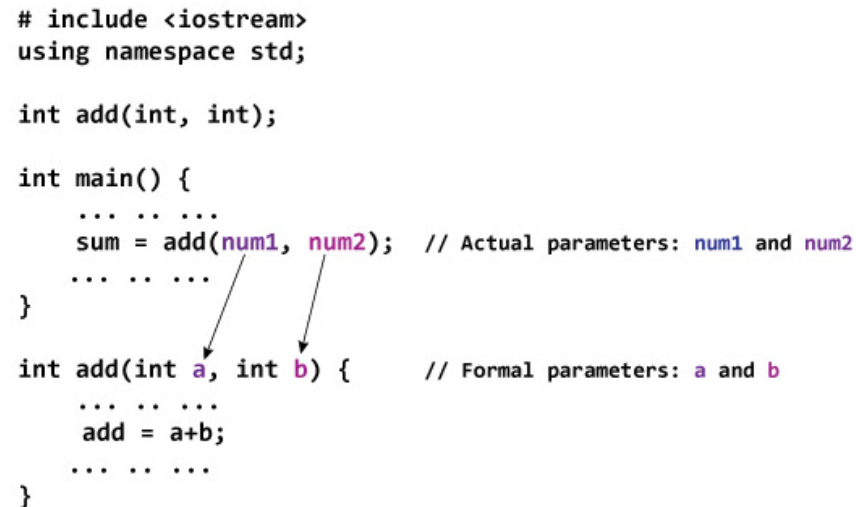
- The numbers of actual arguments and formal argument should be the same. (Exception: [Function Overloading](#))
- The type of first actual argument should match the type of first formal argument. Similarly, type of second actual argument should match the type of second formal argument and so on.
- You may call function a without passing any argument. The number(s) of argument passed to a function depends on how programmer want to solve the problem.
- You may assign default values to the argument. These arguments are known as [default arguments](#).
- In the above program, both arguments are of `int` type. But it's not necessary to have both arguments of same type.

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




int add(int, int);

int main() {
    ... ..
    sum = add(num1, num2); // Actual parameters: num1 and num2
    ... ..
}

int add(int a, int b) { // Formal parameters: a and b
    ... ..
    add = a+b;
    ... ..
}
```

A diagram illustrating the passing of arguments from the `main` function to the `add` function. Two arrows originate from the `num1` and `num2` arguments in the `sum = add(num1, num2);` line within the `main` function. These arrows point to the `a` and `b` parameters in the `int add(int a, int b) {` line of the `add` function, respectively. This visualizes the mapping of actual parameters to formal parameters.

Function Arguments

Sr.No	Call Type & Description
1	Call by Value  This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
2	Call by Pointer  This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.
3	Call by Reference  This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

Example

```
main() {
    int i = 10, j = 20;
    swapThemByVal(i, j);
    cout << i << " " << j << endl;    // displays 10  20
    swapThemByRef(i, j);
    cout << i << " " << j << endl;    // displays 20  10
    ...
}

void swapThemByVal(int num1, int num2) {
    int temp = num1;
    num1 = num2;
    num2 = temp;
}

void swapThemByRef(int& num1, int& num2) {
    int temp = num1;
    num1 = num2;
    num2 = temp;
}
```

Function Overloading

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.

Function Overloading – Example 1

```
#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
```

Function Overloading – Example 2

```
// 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
```

Both Examples - Explained

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;  
}
```

Inline Function

- Inline function is a function that is expanded in line when it is called.
- When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call.
- This substitution is performed by the C++ compiler at compile time.
- Inline function may increase efficiency if it is small.

Inline Function

```
#include <iostream>
using namespace std;
inline int cube(int s)
{
    return s*s*s;
}
int main()
{
    cout << "The cube of 3 is: " << cube(3) << "\n";
    return 0;
} //Output: The cube of 3 is: 27
```