OOP(C++)

(UNIT II)

FUNCTIONS IN C++

TOKENS

- Smallest individual unit in a program
- C++ has the following tokens:

  1. Keywords
  2. Identifiers
  3. Constants
  4. Strings
  5. Operators
- C++ program is written using these tokens, white spaces & syntax of language.

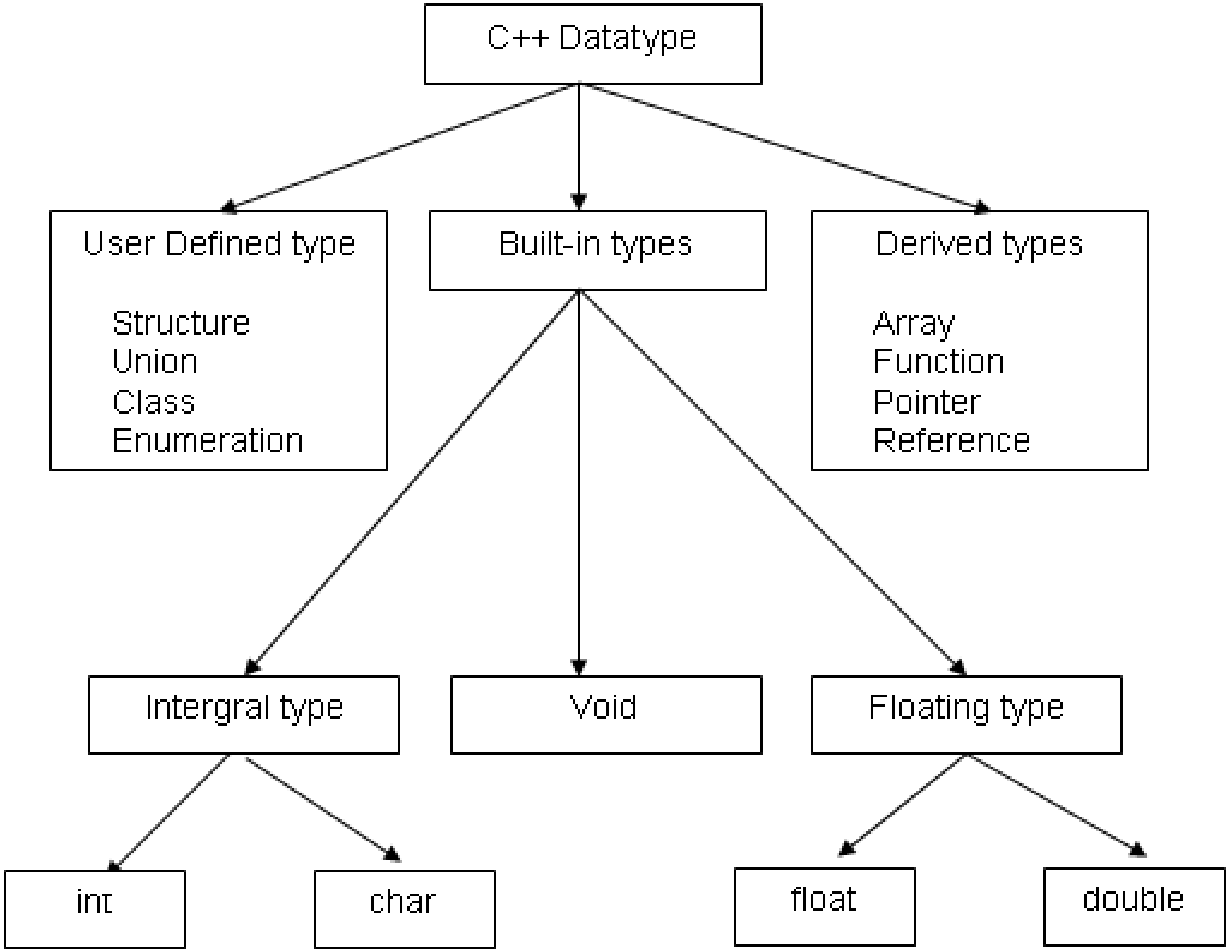
KEYWORDS IN C++

auto	**break**	**case**	**char**
const	**continue**	**default**	**do**
double	**else**	**enum**	**extern**
float	**for**	**goto**	**if**
int	**long**	**register**	**return**
short	**signed**	**sizeof**	**static**
struct	**switch**	**typedef**	**union**
unsigned	**void**	**volatile**	**while**
asm	friend	operator	public
catch	inline	template	throw
class	try	private	virtual
delete	new	this	protected

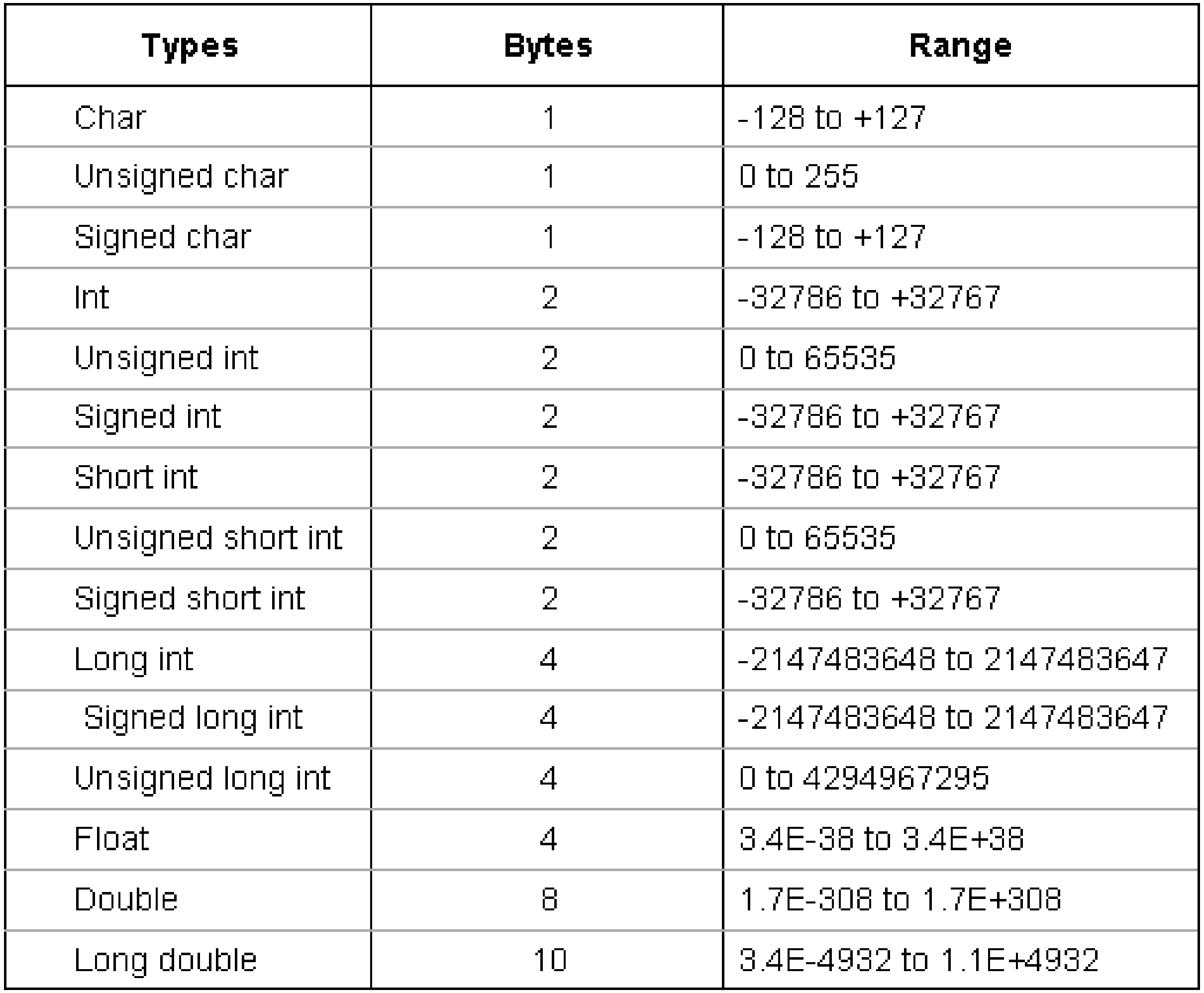
Identifiers and Constants

- **Identifiers** refers to the name of variables, functions, arrays, classes, etc. created by the user. Identifiers are the fundamental requirement of any language.
- **Identifier naming conventions**
  - Only alphabetic characters, digits and underscores are permitted.
  - First letter must be an alphabet or underscore (_).
  - Identifiers are case sensitive.
  - Reserved keywords can not be used as an identifier's name.
- **Constants**
  - **Constants** refers to fixed values that do not change during the execution of a program.
- **Declaration of a constant :**
  - const [data_type] [constant_name]=[value];

Basic Data Types



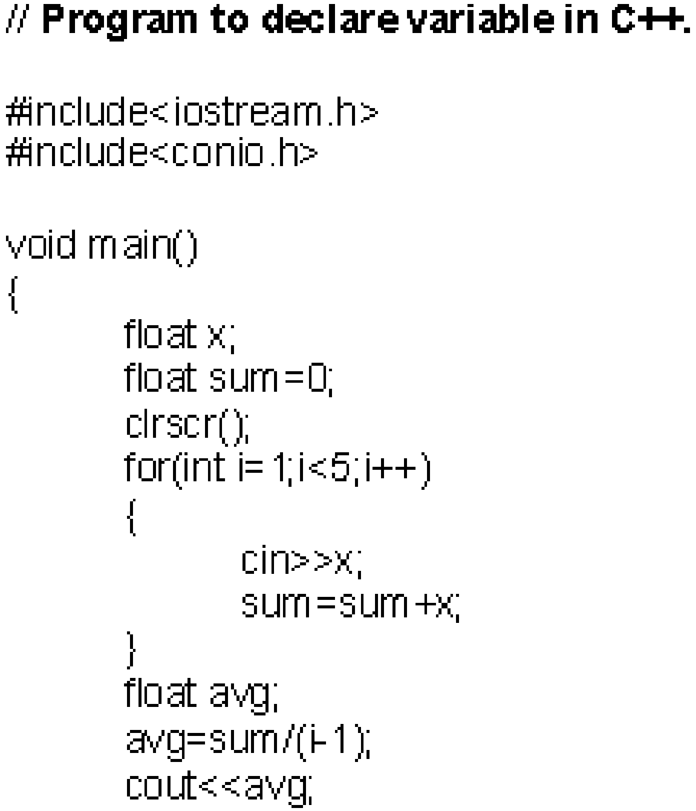
Basic Data Types



Types	Bytes	Range
Char	1	-128 to +127
Unsigned char	1	0 to 255
Signed char	1	-128 to +127
Int	2	-32786 to +32767
Unsigned int	2	0 to 65535
Signed int	2	-32786 to +32767
Short int	2	-32786 to +32767
Unsigned short int	2	0 to 65535
Signed short int	2	-32786 to +32767
Long int	4	-2147483648 to 2147483647
Signed long int	4	-2147483648 to 2147483647
Unsigned long int	4	0 to 4294967295
Float	4	3.4E-38 to 3.4E+38
Double	8	1.7E-308 to 1.7E+308
Long double	10	3.4E-4932 to 1.1E+4932

Declaration of Variables

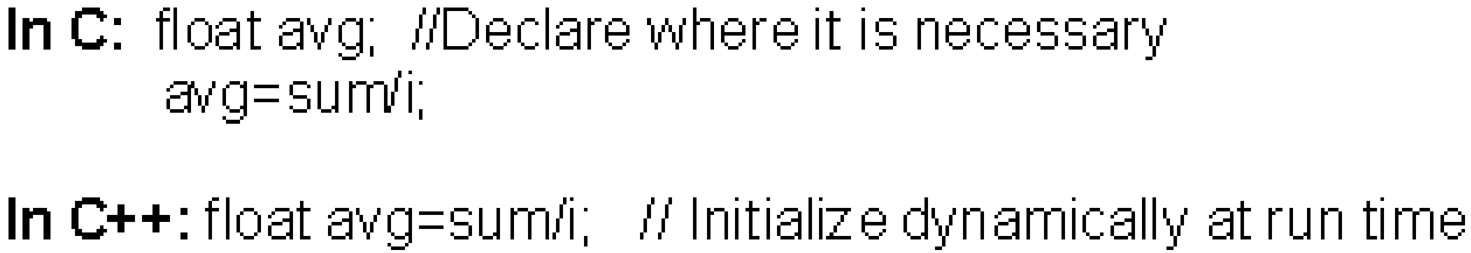
- Variables must be declared before they are used.
- Allow declarations of variables anywhere in the scope. This means that a variable can be declared right at place of its first use.
- Make program much easier to write & reduce errors that may caused.



```
// Program to declare variable in C++.

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

void main()
{
        float x;
        float sum=0;
        clrscr();
        for(int i= 1;i<5;i++)
        {
                cin>>x;
                sum=sum+x;
        }
        float avg;
        avg=sum/(i-1);
        cout<<avg;
```

Dynamic Initialization of Variables

- Dynamic Initialization: C++ permits the initialization of variable at runtime.
- In C, Compiler would fix the initialization code at the compile time.
- In C++, variable can be initialized at run time using expression at the place of declaration.
- For example:



```
In C:  float avg;  //Declare where it is necessary
           avg=sum/i;

In C++: float avg=sum/i;   // Initialize dynamically at run time
```

Reference Variables

- It provides an alias (alternative name) for a previously defined variable.
- Syntax: datatype & reference_name = variable_name;
- Example:

```
float total=100;
float & sum= total;
```

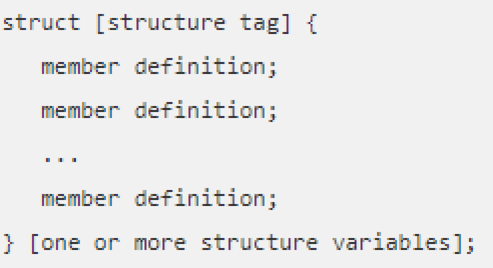
- It must be initialized at the time of declaration.
- If variable 'total' value changes then automatically 'sum' reference variable also changes.
- Application is in passing arguments to functions

User Defined Data Types

Struct

- The struct statement defines a new data type, with more than one member, for your program.

- Structure is used to group dissimilar data types.



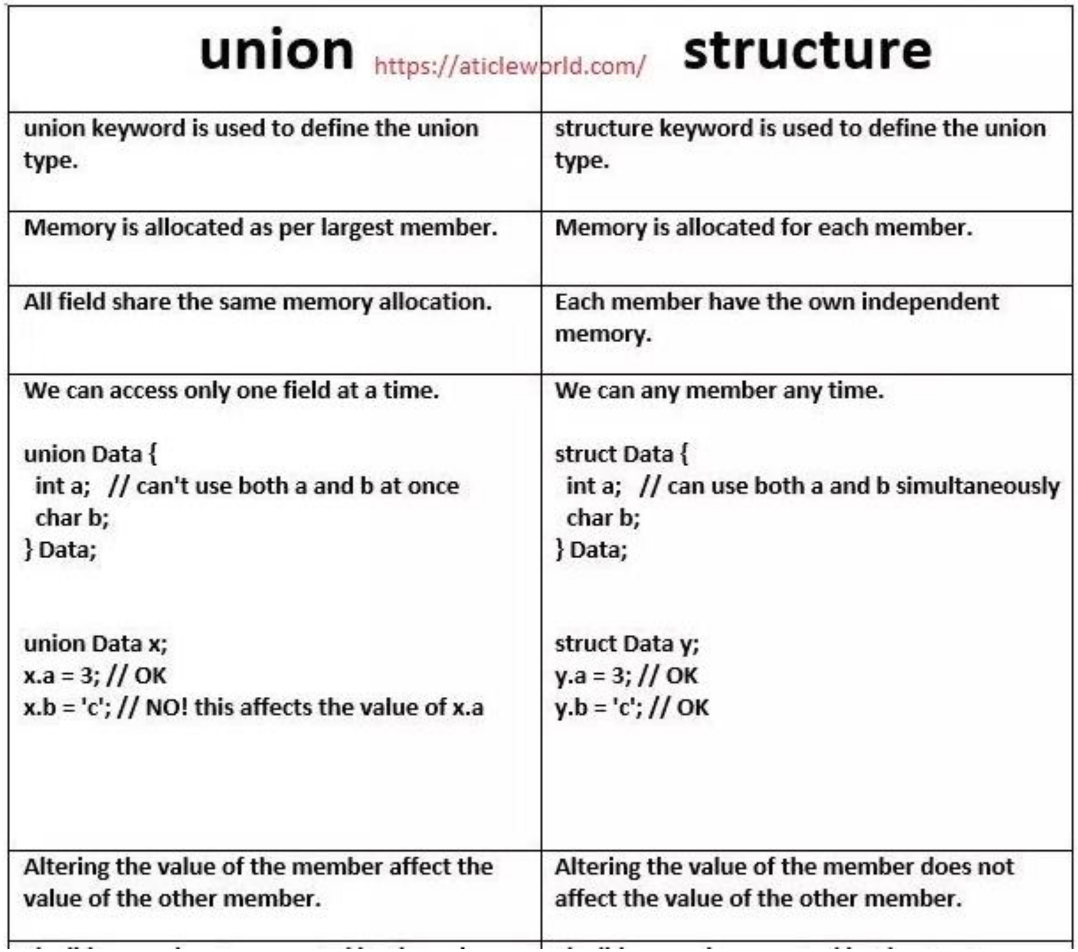
```
struct [structure tag] {
   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

Difference b/w Structure & Class

STRUCTURE	**CLASS**
Collections of variables of different data types	Collections of variables of different data types & functions
'struct' keyword are used while declaring a structure	'class' keyword are used while declaring a class
Structure variables can be created	Class variables can be created known as objects
No public & private member declaration present in structure	Data members & member function can be declared as public, private as well as protected
Data hiding is not achieved	Data hiding is achieved with private key

Syntax:
struct struct_name
{
Variable 1;
……
Variable n;
} struct_var;
	Syntax;
	class class_name
	{
	private:
	data member declaration;
	member function declaration;
	public:
	………
	};

Difference b/w union & class



union	structure
union keyword is used to define the union type.	structure keyword is used to define the union type.
Memory is allocated as per largest member.	Memory is allocated for each member.
All field share the same memory allocation.	Each member have the own independent memory.

We can access only one field at a time.
 	
union Data {
int a; // can't use both a and b at once
char b;
} Data;
 	
union Data x;
x.a = 3; // OK
x.b = 'c'; // NO! this affects the value of x.a
	We can any member any time.
	
	struct Data {
	int a; // can use both a and b simultaneously
	char b;
	} Data;
	
	struct Data y;
	y.a = 3; // OK
	y.b = 'c'; // OK

Altering the value of the member affect the value of the other member.	Altering the value of the member does not affect the value of the other member.

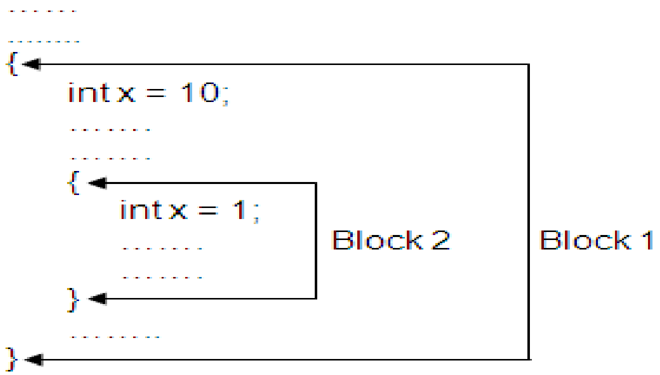
Operators in C++

- All C operators are valid in C++
- Other new operators are,

<<	Insertion operator
>>	Extraction operator
::	Scope Resolution operator
::*	Pointer to Member declarator
->*	Pointer to member operator
.*	Pointer to member operator
delete	Memory Release operator
endl	Line Feed operator
new	Memory Allocation operator
setw	Field Width operator

SCOPE RESOLUTION OPERATOR

- C is block structured language.
- Scope of variable extends from the point of its declaration till end of block containing the declaration.

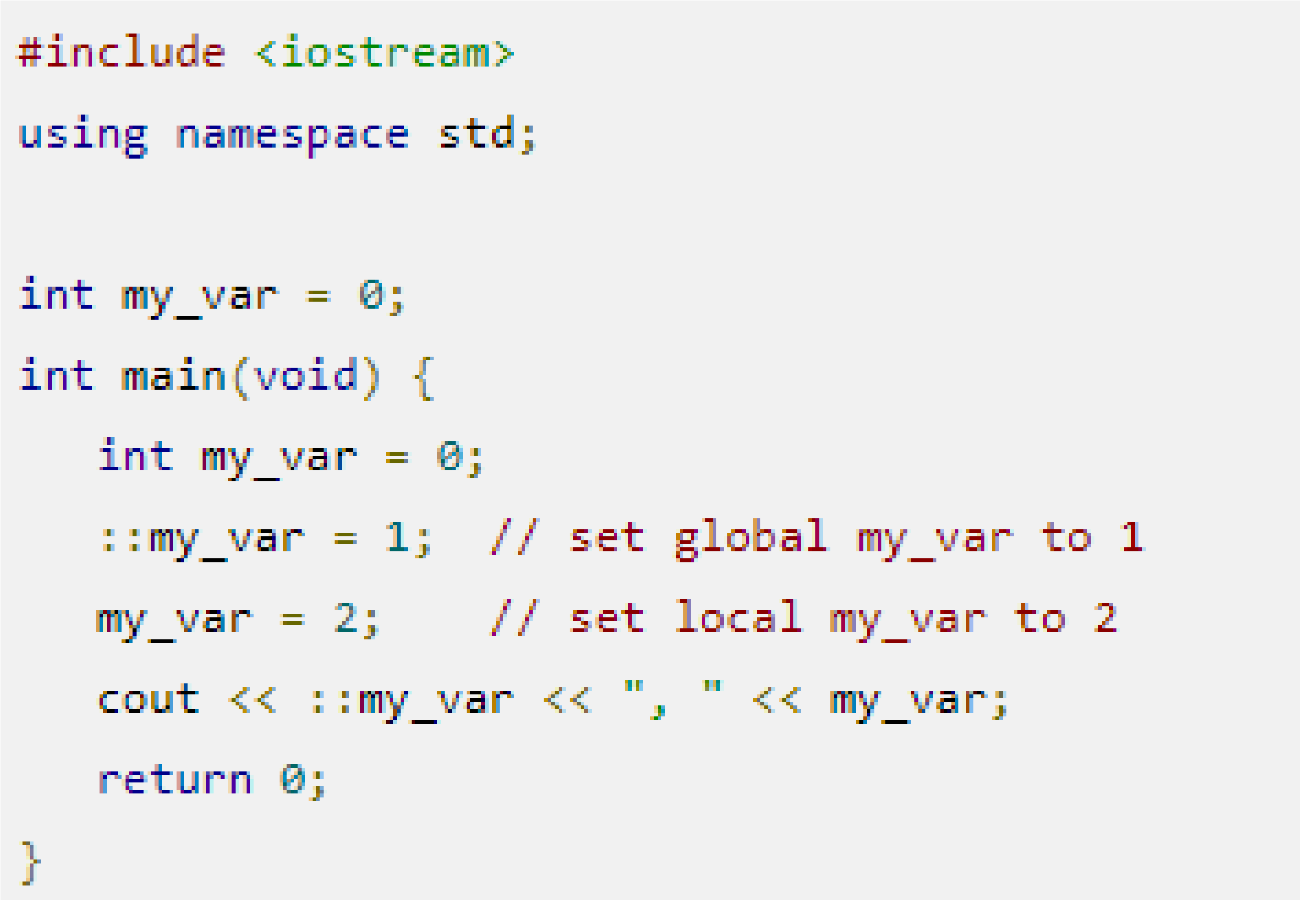


```
......
........
{
    int x = 10;
    .......
    .......
    {
        int x = 1;
        .......
        .......
    }
    ........
}
```

- In C, global version of variable cannot be accessed from within inner block, C++ will resolve this problem by using ::
- Allows access to global version of variable.

- Syntax : :: variable-name

SCOPE RESOLUTION OPERATOR



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

int my_var = 0;
int main(void) {
   int my_var = 0;
   ::my_var = 1;  // set global my_var to 1
   my_var = 2;    // set local my_var to 2
   cout << ::my_var << ", " << my_var;
   return 0;
}
```

1,2

Manipulators

- Used to format the data display.
- Manipulators: endl & setw

1. endl operator:

- Used in O/P statement, causes linefeed to be inserted same as '\n'.
- Example: cout<< "m= "<<m<<endl

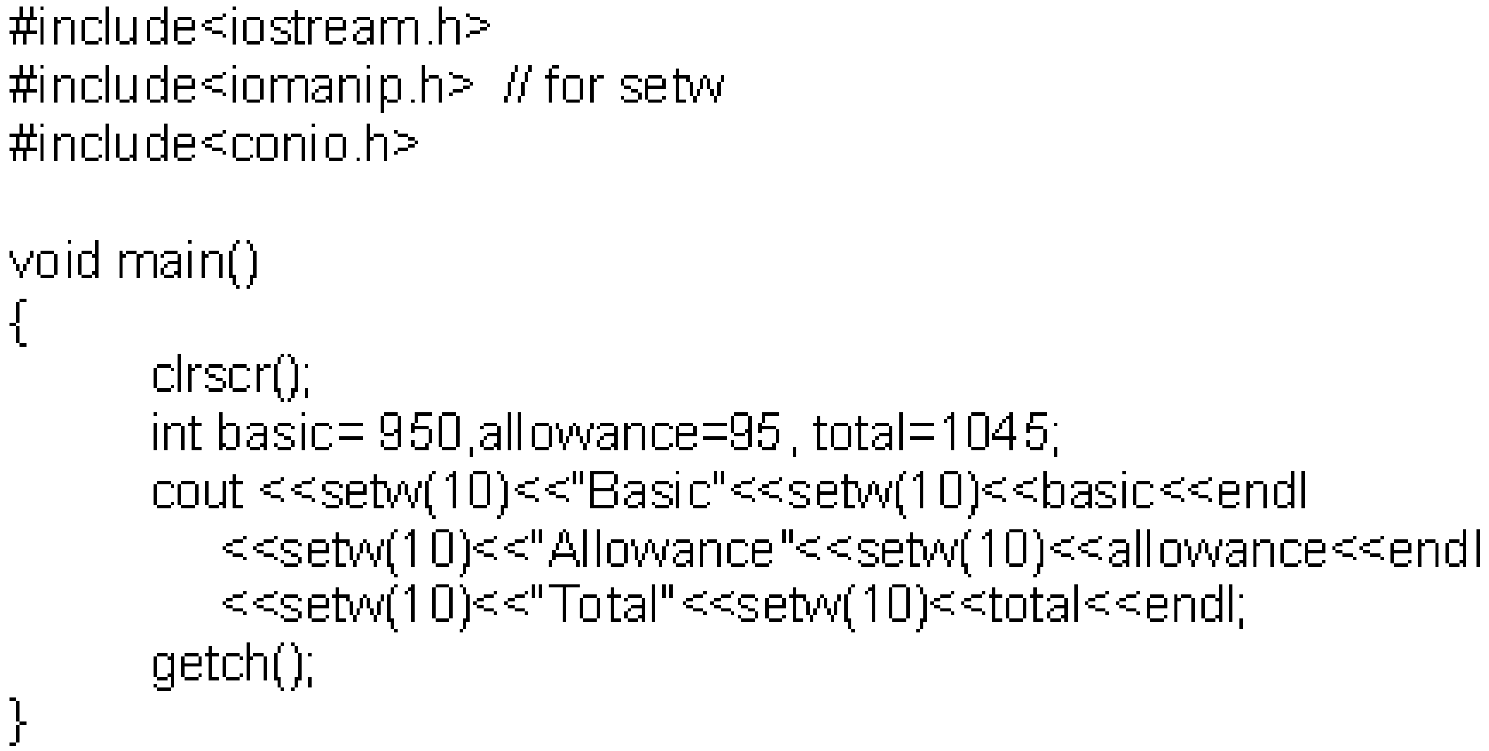
```
                    <<"n=  "<<n<<endl;
```

2. Setw operator:

- Used to all numbers & character strings force them to be printed right justified.
- Example: cout<<setw(5)<<sum;

MANIPULATORS

- Program to use manipulators.

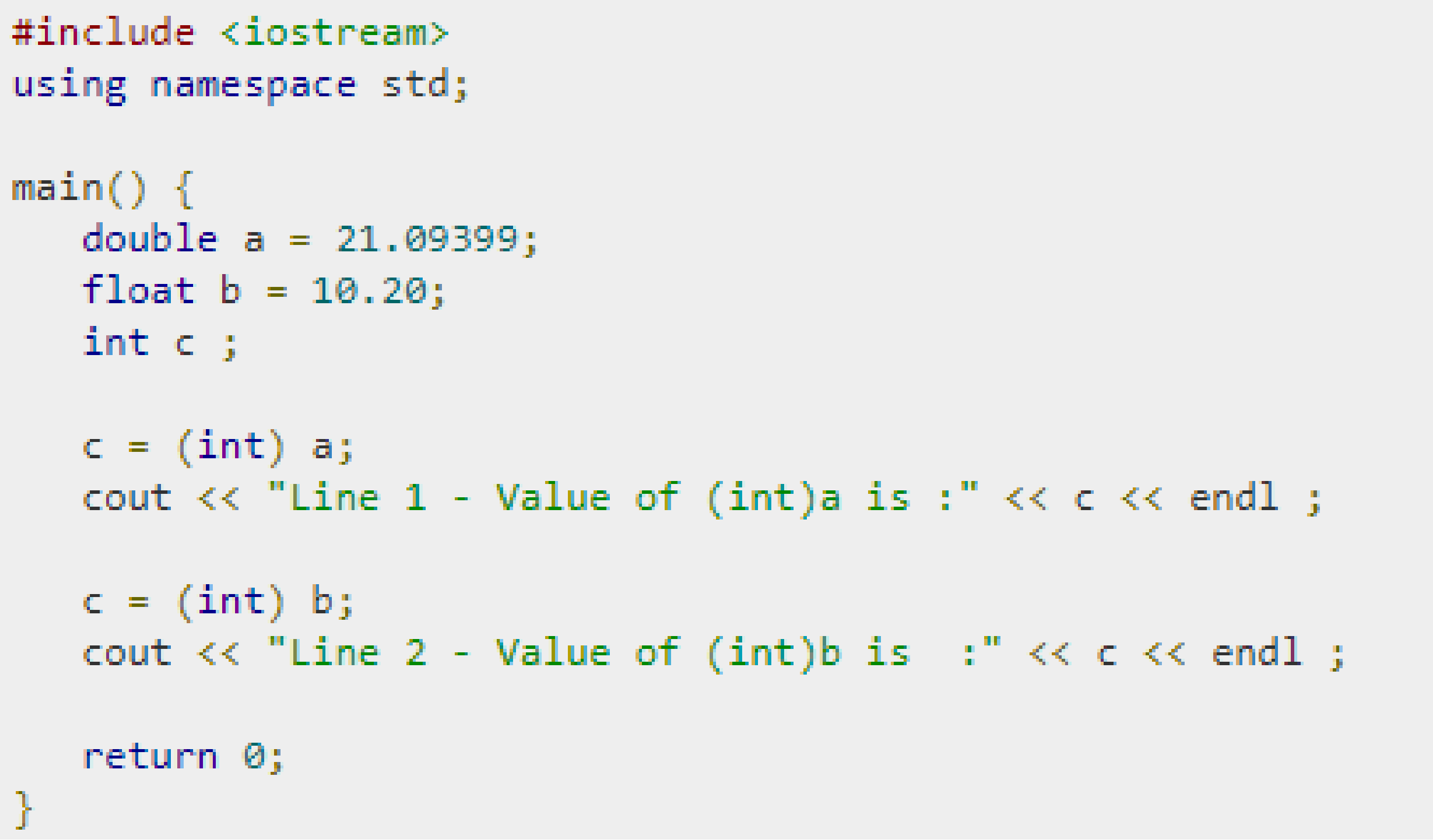


```
#include<iostream.h>
#include<iomanip.h>  // for setw
#include<conio.h>

void main()
{
        clrscr();
        int basic= 950,allowance=95 , total=1045;
        cout <<setw(10)<<"Basic"<<setw(10)<<basic<<endl
             <<setw(10)<<"Allowance"<<setw(10)<<allowance<<endl
             <<setw(10)<<"Total"<<setw(10)<<total<<endl;
        getch();
}
```

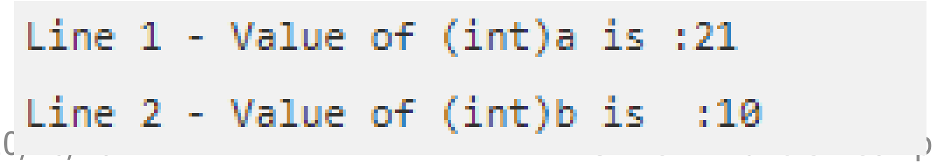
Type Cast Operators

- Permits explicit type conversion of variables using this
- Syntax: (typename) expression // C Notation
  typename (expression) // C++ Notation
- Examples: avg = sum/(float) i; // C Notation
  avg = sum/float (i); // C++ Notation

Type Cast Operators



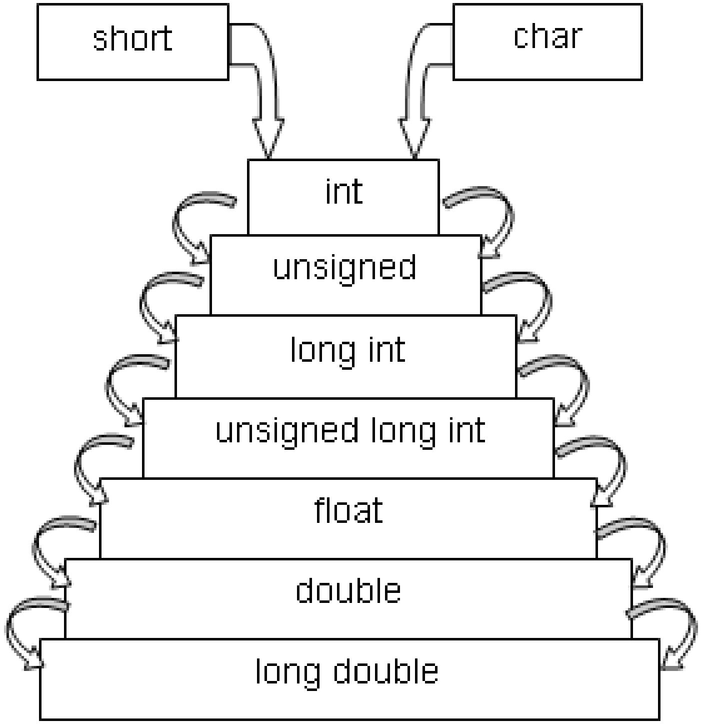
```cpp
#include <iostream>
using namespace std;

main() {
   double a = 21.09399;
   float b = 10.20;
   int c ;

   c = (int) a;
   cout << "Line 1 - Value of (int)a is :" << c << endl ;

   c = (int) b;
   cout << "Line 2 - Value of (int)b is  :" << c << endl ;

   return 0;
}
```



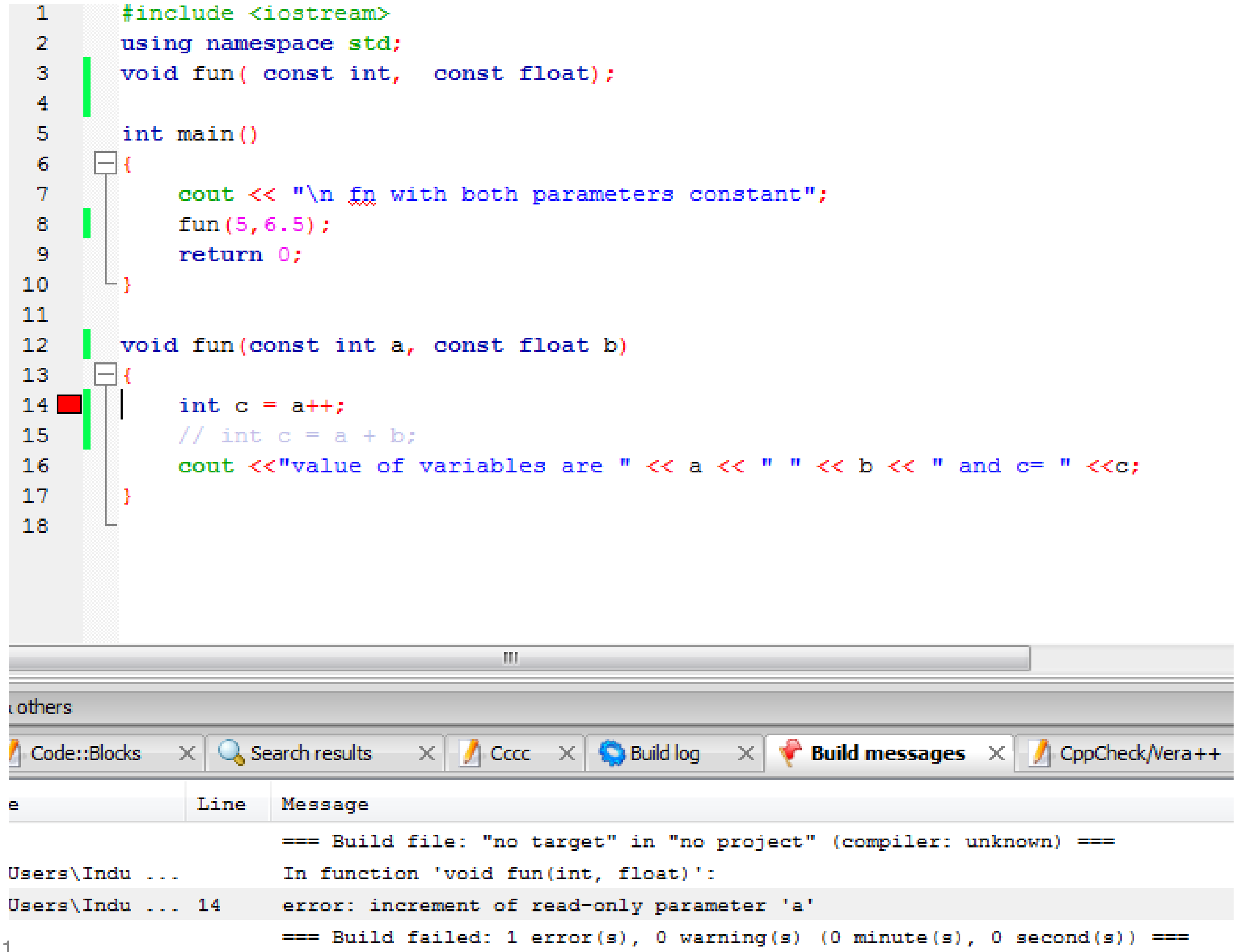
```
Line 1 - Value of (int)a is :21

Line 2 - Value of (int)b is  :10
```

Implicit Conversions

- Wherever data types are mixed in expression, C++ performs conversion automatically called as implicit conversion
- For Example, if one of the operand is **int** & other is **float**, int is converted into **float** because **float** is wider than **int**.
- **Water Fall Model:**



CONST ARGUMENTS



```
1   #include <iostream>
2   using namespace std;
3   void fun( const int,  const float);
4
5   int main()
6   {
7       cout << "\n fn with both parameters constant";
8       fun(5,6.5);
9       return 0;
10  }
11
12  void fun(const int a, const float b)
13  {
14      int c = a++;
15      // int c = a + b;
16      cout <<"value of variables are " << a << " " << b << " and c= " <<c;
17  }
18
```

```
e                 Line    Message
                          === Build file: "no target" in "no project" (compiler: unknown) ===
Users\Indu ...            In function 'void fun(int, float)':
Users\Indu ...    14      error: increment of read-only parameter 'a'
                          === Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===
```

MEMORY MANAGEMENT OPERATOR

- Dynamic allocation technique
- Operators new & delete that perform task of allocating & freeing memory is better & easier way.

1. **New operator:**

- Used to create object of any type when it required, will remain in existence until it explicitly destroyed by using delete.
- Syntax: pointer-variable= new datatype;

  pointer-variable= new datatype (value);

  pointer-variable= new datatype [size];
- Example: p= new int; **OR** int *p = new int;

  int *p = new int[10]

Memory Management Operator

2. Delete operator:

- When a data object is no longer needed, it is destroyed to release memory space for reuse.
- Syntax: delete pointer-variable;

  delete [size] pointer-variable;
- Example: delete p;

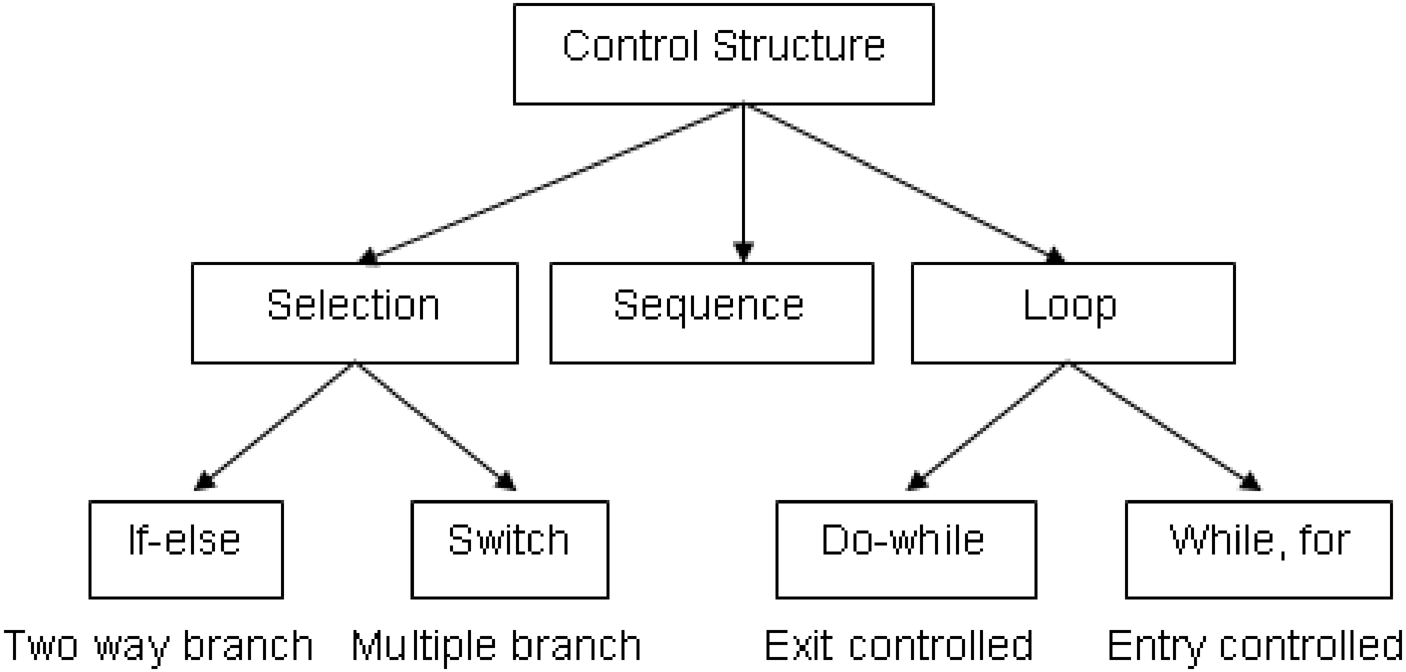
  delete [ ] p;

Operator Overloading

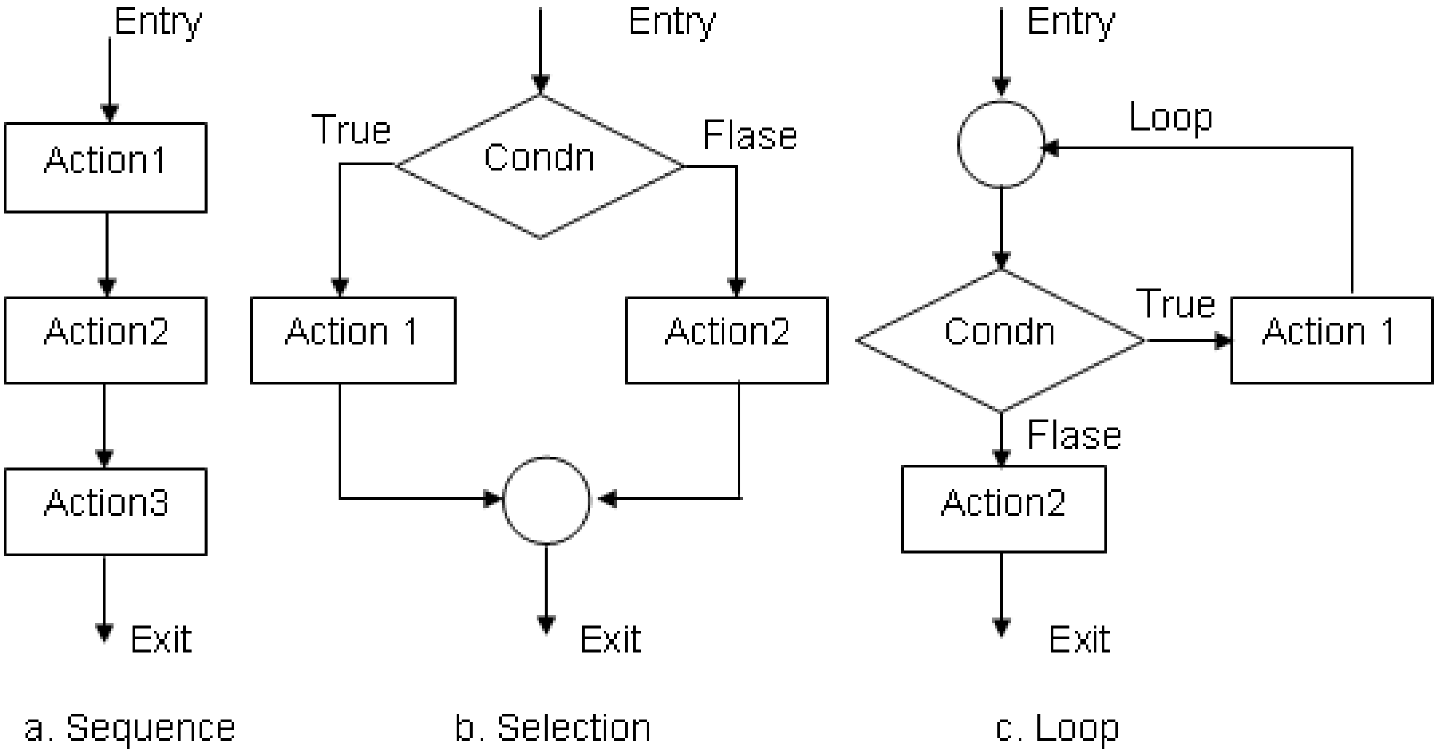
- Overloading: Assigning different meanings to an operation, depending on the context.
- Operator overloading: To assign multiple meanings to operators.
- For example, * Operator (used with pointers & also for multiplying two numbers), << & >> operator, + operator.
- Almost operators are overloaded in C++, with few exceptions such as ::, . & .*, ?: and sizeof operator.

Control Structure

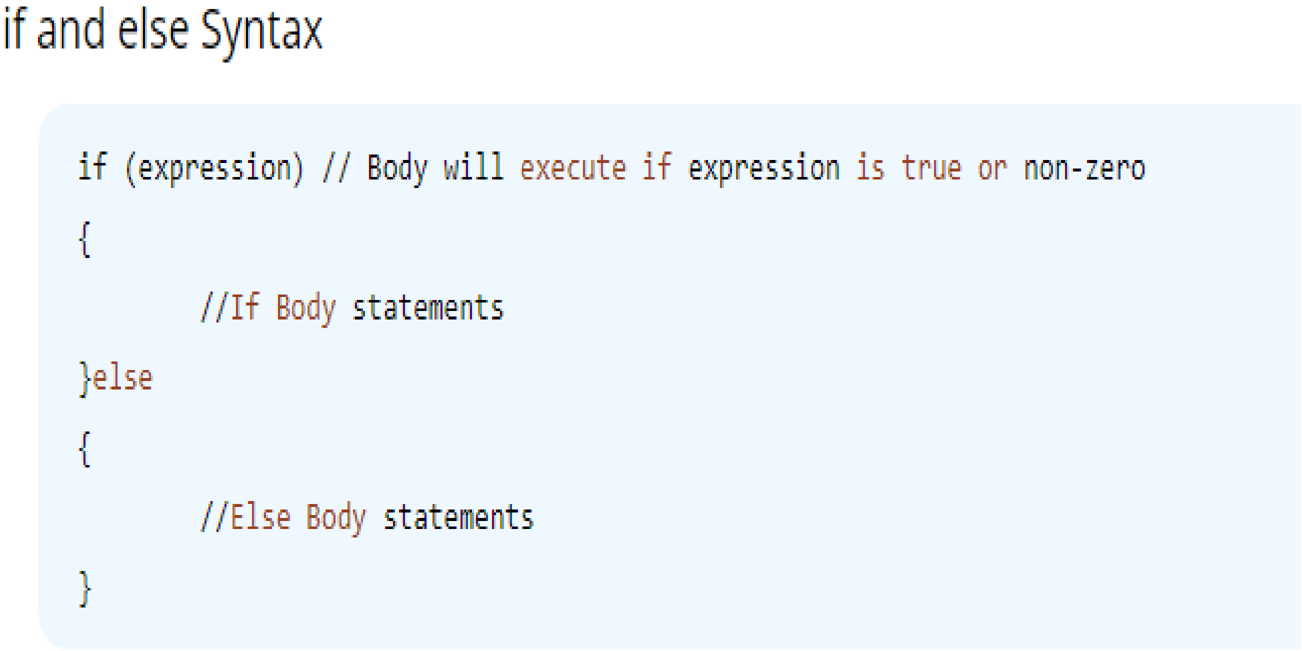
- C++ support no. of control structure are given as below,



Control Structure



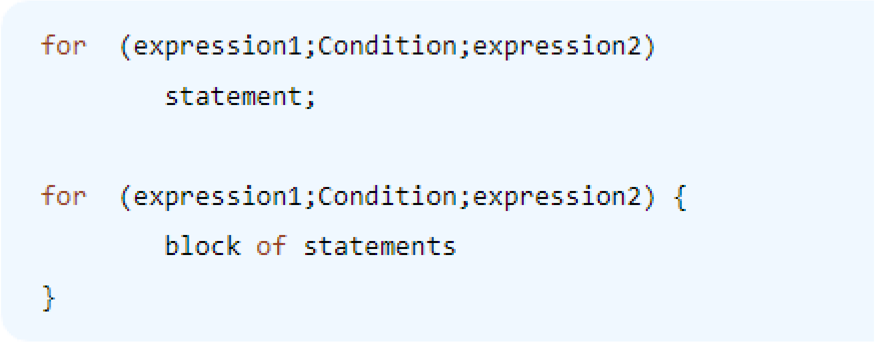
If-else



if and else Syntax

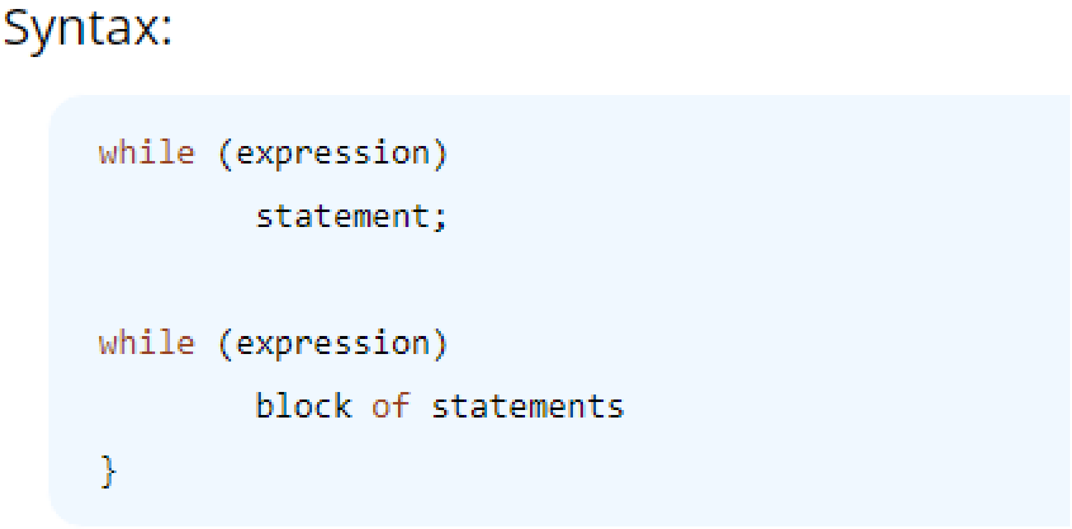
```
if (expression) // Body will execute if expression is true or non-zero
{
        //If Body statements
}else
{
        //Else Body statements
}
```

Control Structure - For



```
for  (expression1;Condition;expression2)
        statement;

for  (expression1;Condition;expression2) {
        block of statements
}
```

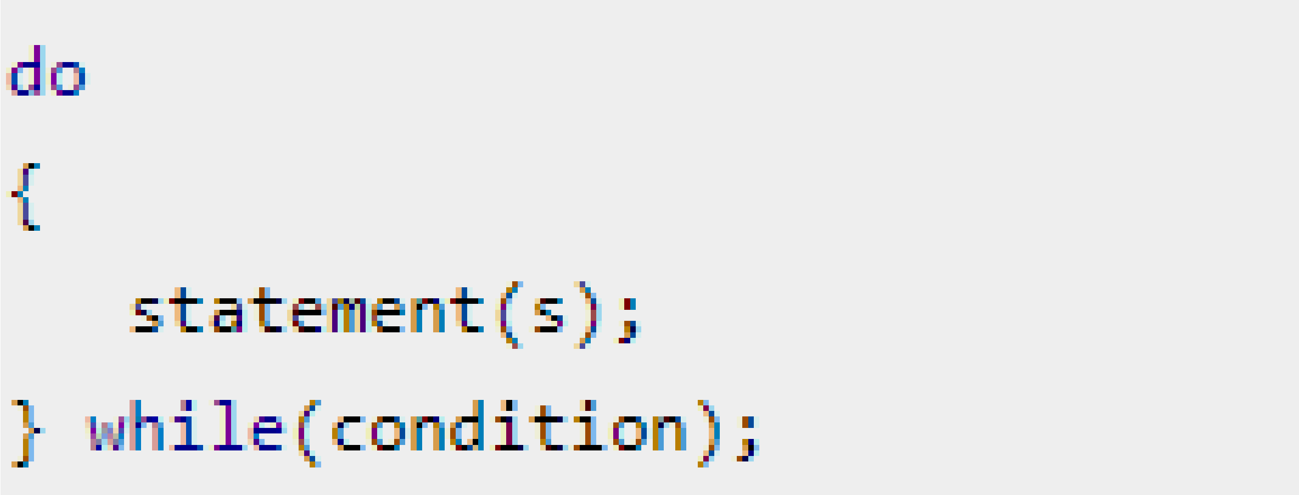
While statement



Syntax:

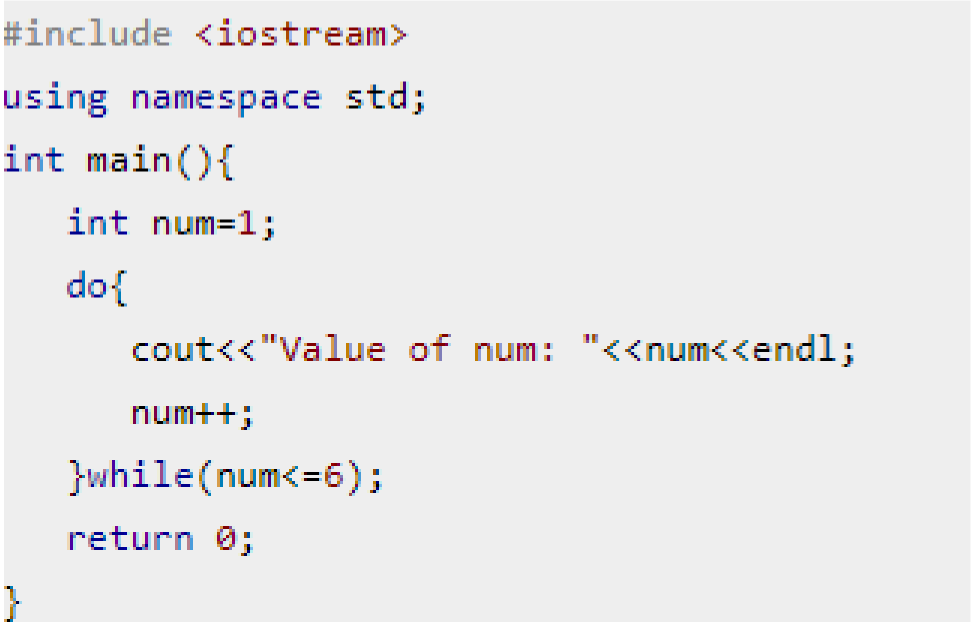
```
while (expression)
        statement;

while (expression)
        block of statements
}
```

Do while statement



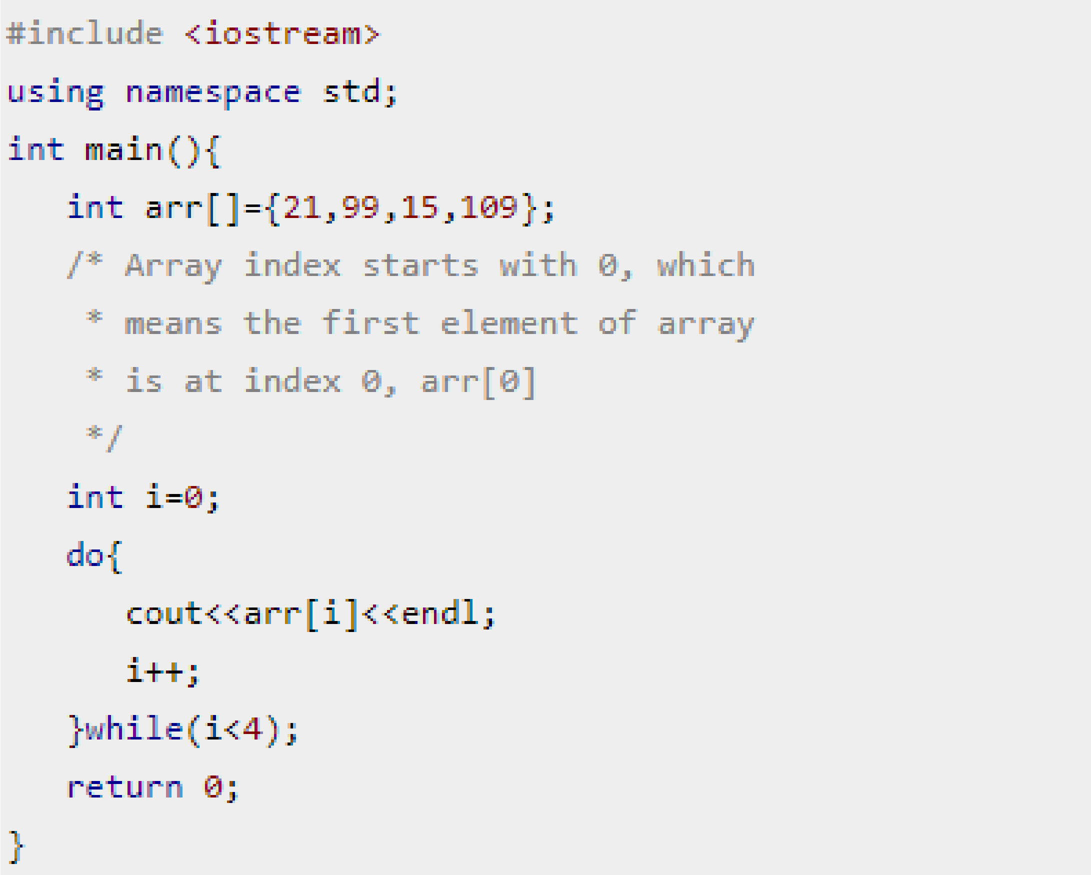
```
do
{
   statement(s);
} while(condition);
```

Do while Example



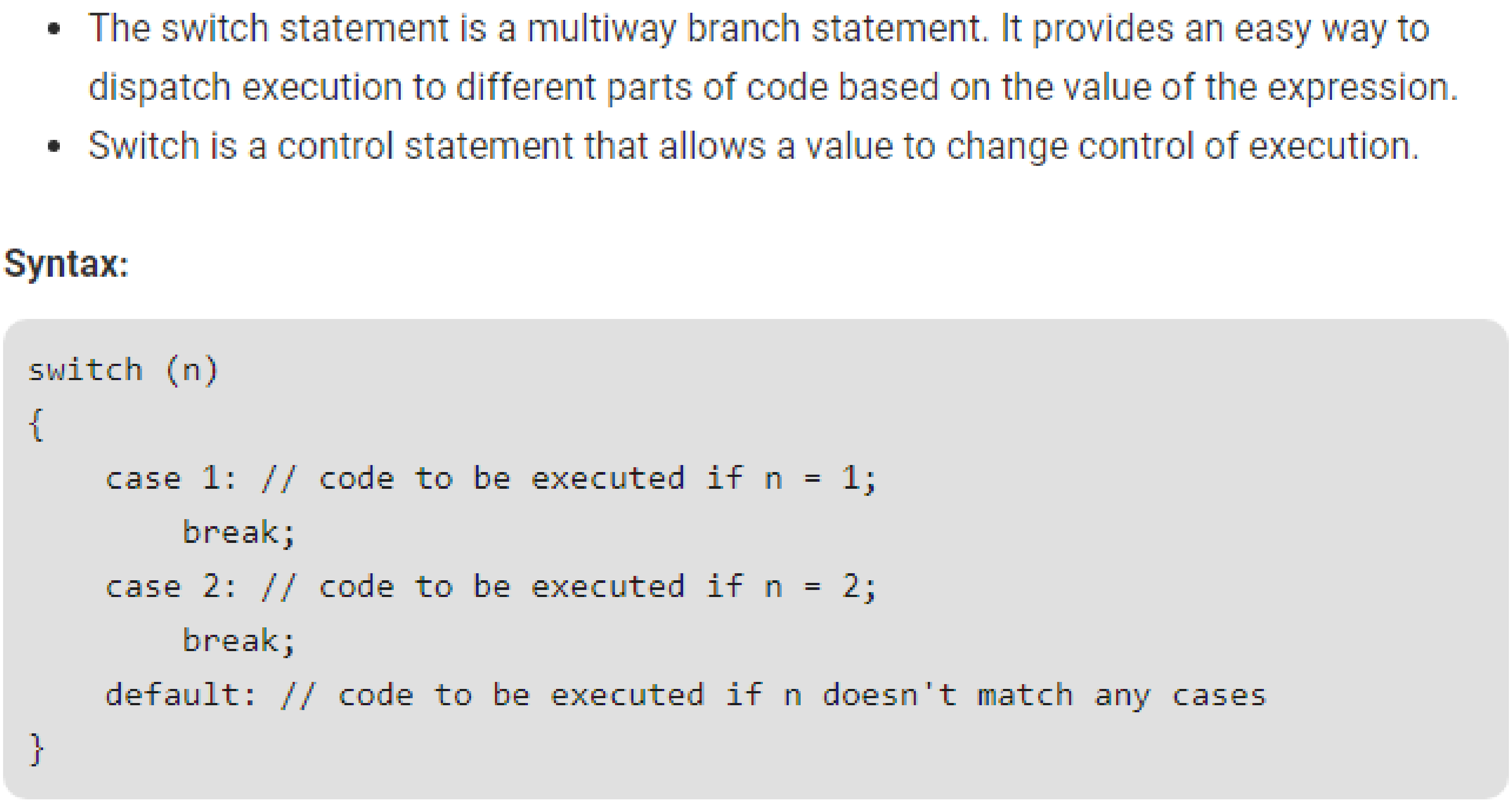
```
#include <iostream>
using namespace std;
int main(){
   int num=1;
   do{
      cout<<"Value of num: "<<num<<endl;
      num++;
   }while(num<=6);
   return 0;
}
```

Do while Example using array



```
#include <iostream>
using namespace std;
int main(){
   int arr[]={21,99,15,109};
   /* Array index starts with 0, which
    * means the first element of array
    * is at index 0, arr[0]
    */
   int i=0;
   do{
      cout<<arr[i]<<endl;
      i++;
   }while(i<4);
   return 0;
}
```

Switch Statement



- The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.
- Switch is a control statement that allows a value to change control of execution.

Syntax:

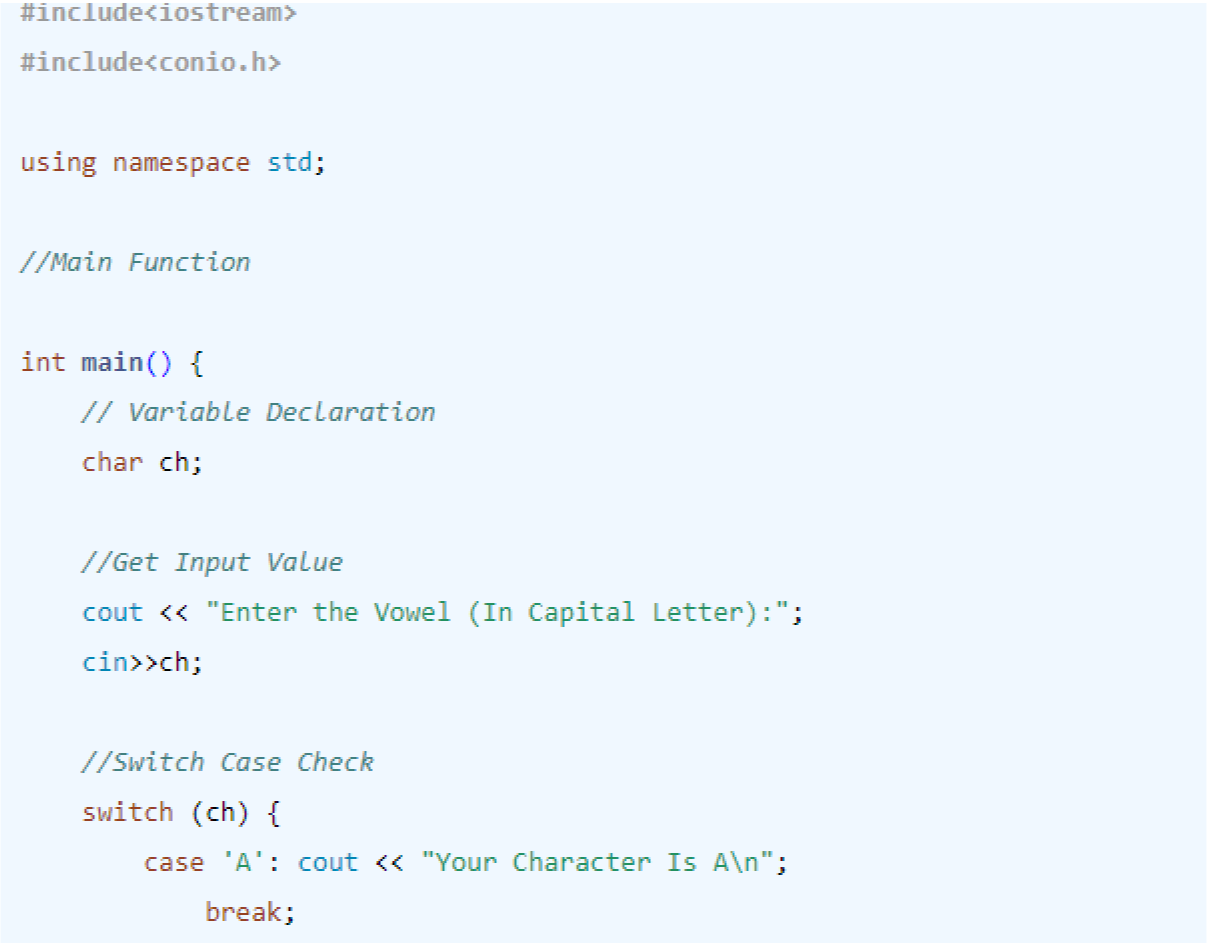
```
switch (n)
{
    case 1: // code to be executed if n = 1;
        break;
    case 2: // code to be executed if n = 2;
        break;
    default: // code to be executed if n doesn't match any cases
}
```

Switch Statement

- Execute a C++ pgm using switch case to perform the following

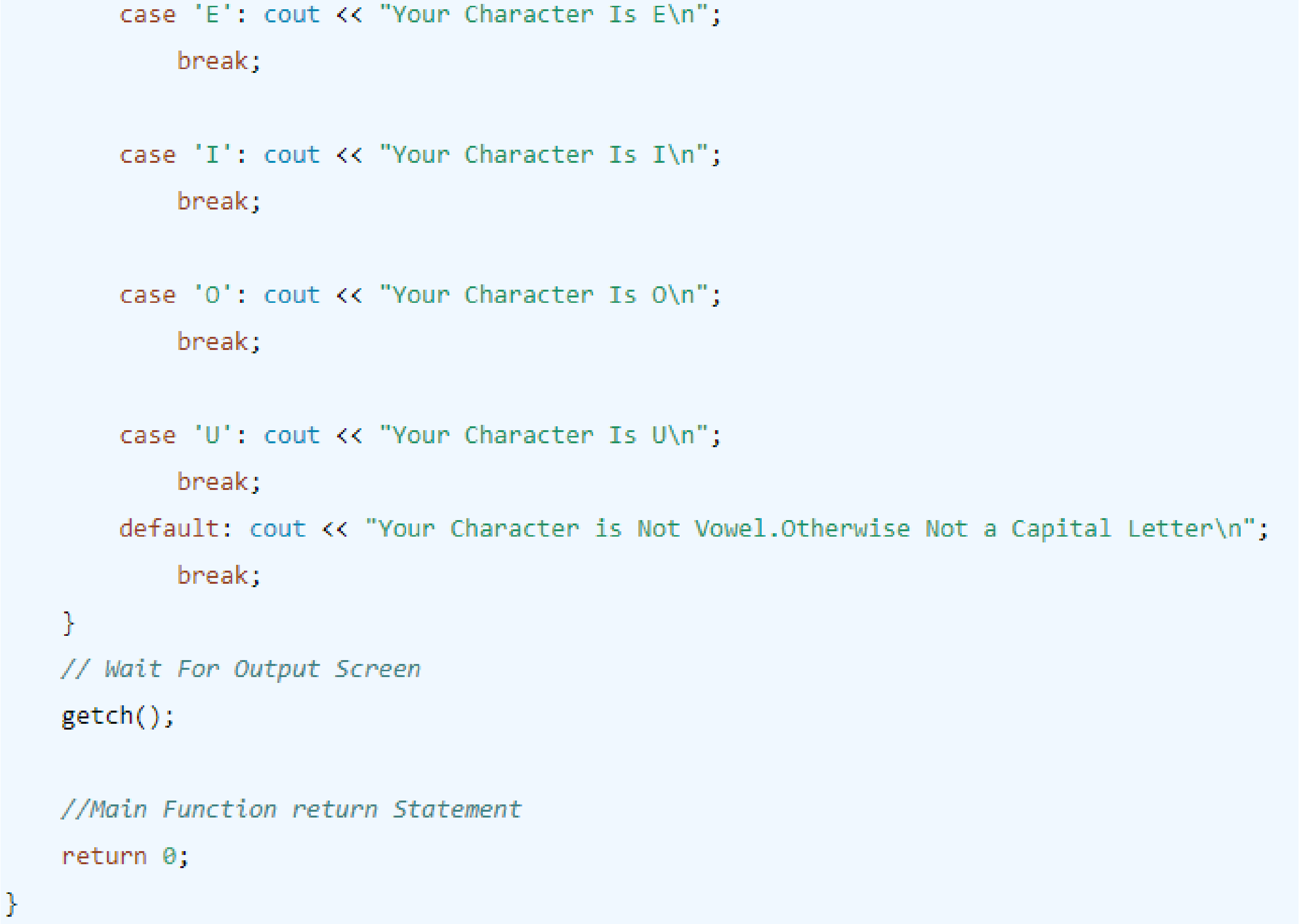
- Add
- Subtract
- Multiply
- Divide

Switch Example



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

using namespace std;

//Main Function

int main() {
    // Variable Declaration
    char ch;

    //Get Input Value
    cout << "Enter the Vowel (In Capital Letter):";
    cin>>ch;

    //Switch Case Check
    switch (ch) {
        case 'A': cout << "Your Character Is A\n";
            break;
```

Switch Example



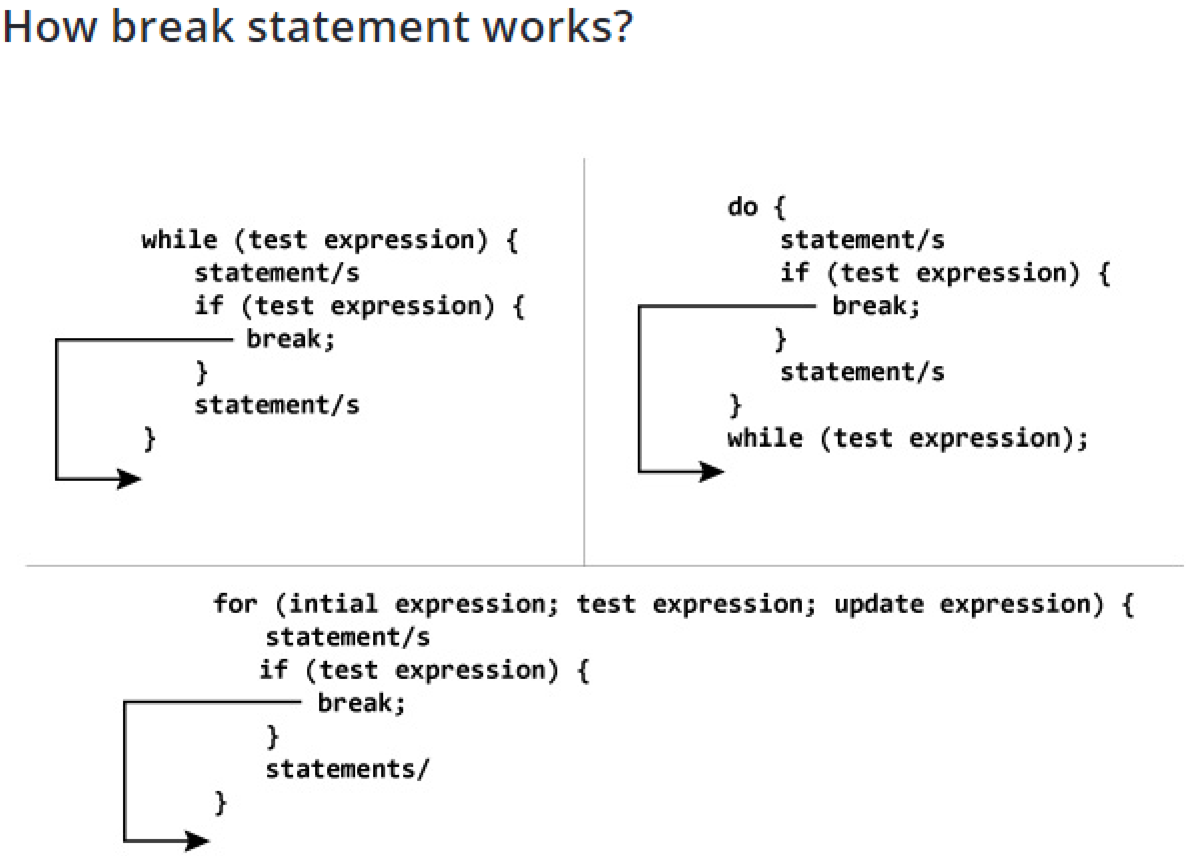
```
        case 'E': cout << "Your Character Is E\n";
            break;

        case 'I': cout << "Your Character Is I\n";
            break;

        case 'O': cout << "Your Character Is O\n";
            break;

        case 'U': cout << "Your Character Is U\n";
            break;
        default: cout << "Your Character is Not Vowel.Otherwise Not a Capital Letter\n";
            break;
    }
    // Wait For Output Screen
    getch();

    //Main Function return Statement
    return 0;
}
```

Break and Continue Statements

The break; statement terminates a loop (for, while and do..while loop) and a switch statement immediately when it appears.

Syntax:

break;



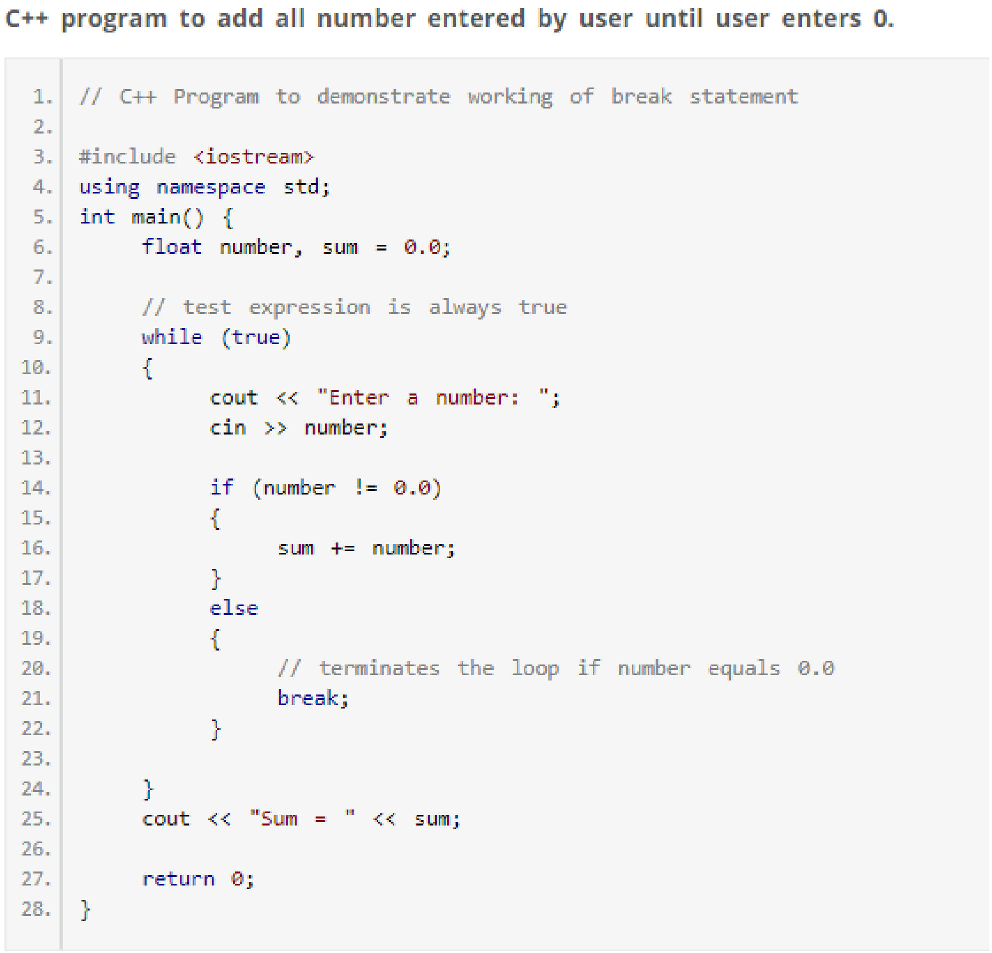
How break statement works?

```
while (test expression) {
    statement/s
    if (test expression) {
        break;
    }
    statement/s
}
```

```
do {
    statement/s
    if (test expression) {
        break;
    }
    statement/s
}
while (test expression);
```

```
for (intial expression; test expression; update expression) {
    statement/s
    if (test expression) {
        break;
    }
    statements/
}
```

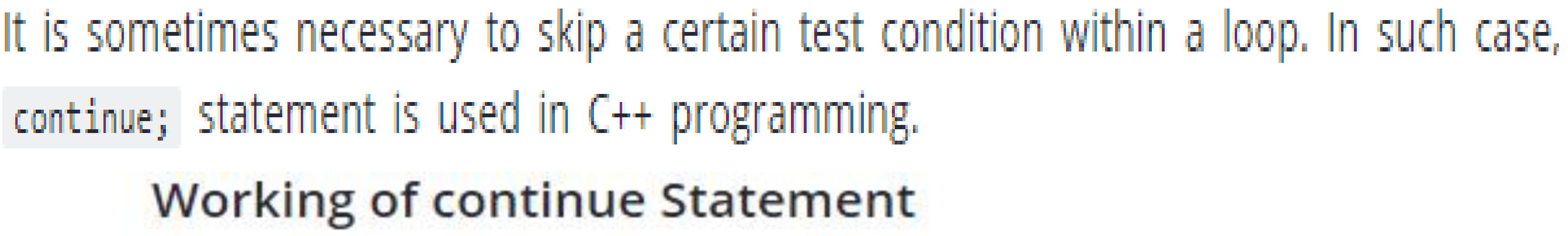
Break statement - Example



C++ program to add all number entered by user until user enters 0.

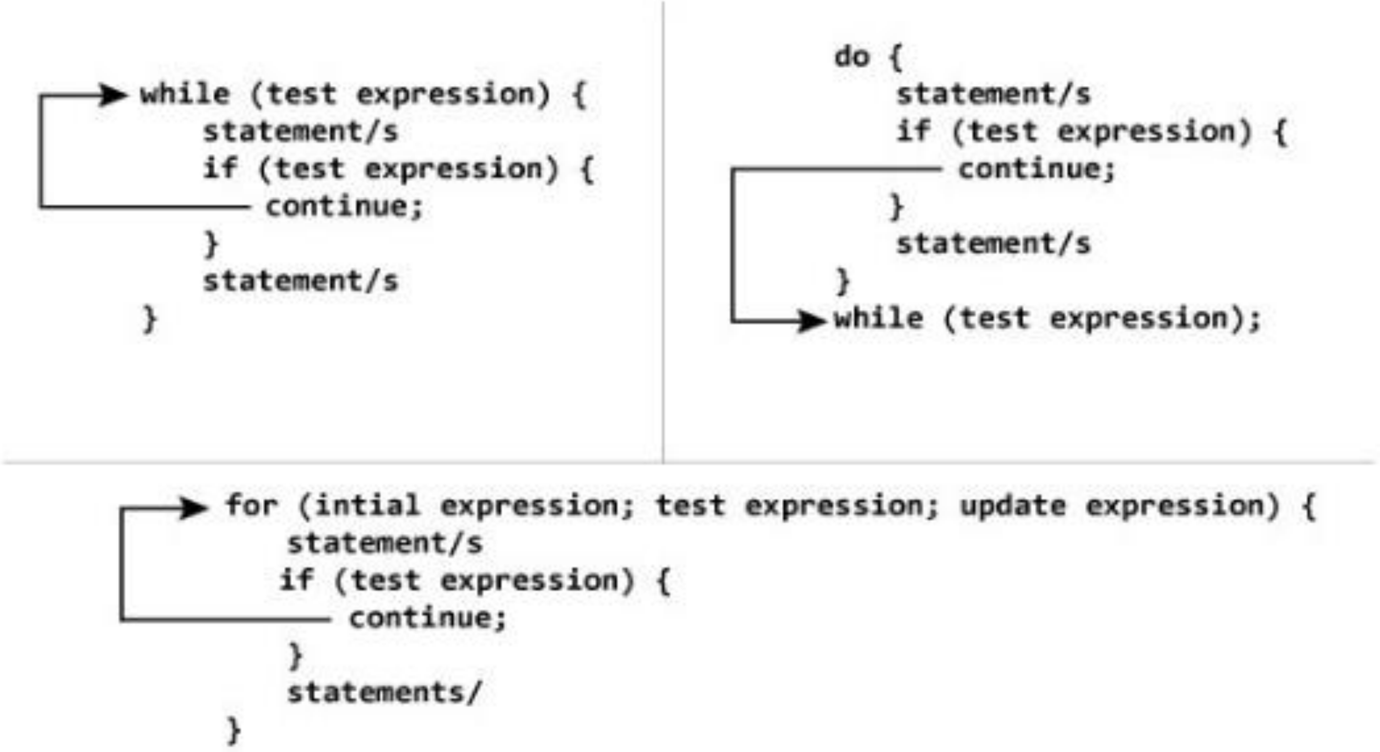
```cpp
// C++ Program to demonstrate working of break statement

#include <iostream>
using namespace std;
int main() {
     float number, sum = 0.0;

     // test expression is always true
     while (true)
     {
          cout << "Enter a number: ";
          cin >> number;

          if (number != 0.0)
          {
               sum += number;
          }
          else
          {
               // terminates the loop if number equals 0.0
               break;
          }

     }
     cout << "Sum = " << sum;

     return 0;
}
```

Continue Statement



It is sometimes necessary to skip a certain test condition within a loop. In such case, `continue;` statement is used in C++ programming.

Working of continue Statement

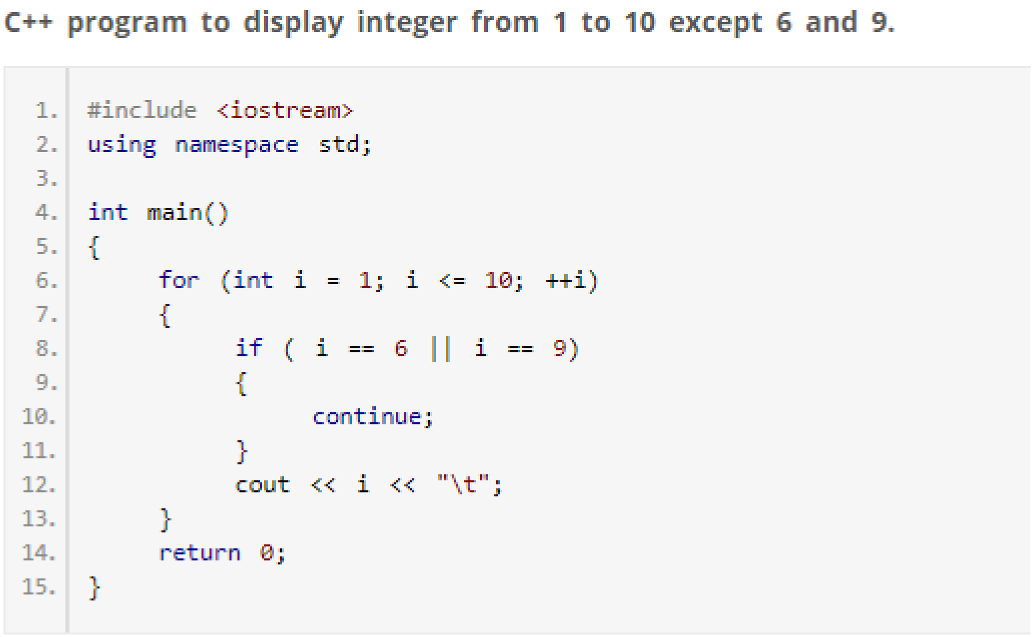


```
while (test expression) {
     statement/s
     if (test expression) {
          continue;
     }
     statement/s
}
```

```
do {
     statement/s
     if (test expression) {
          continue;
     }
     statement/s
}
while (test expression);
```

```
for (intial expression; test expression; update expression) {
     statement/s
     if (test expression) {
          continue;
     }
     statements/
}
```

Continue statement Example

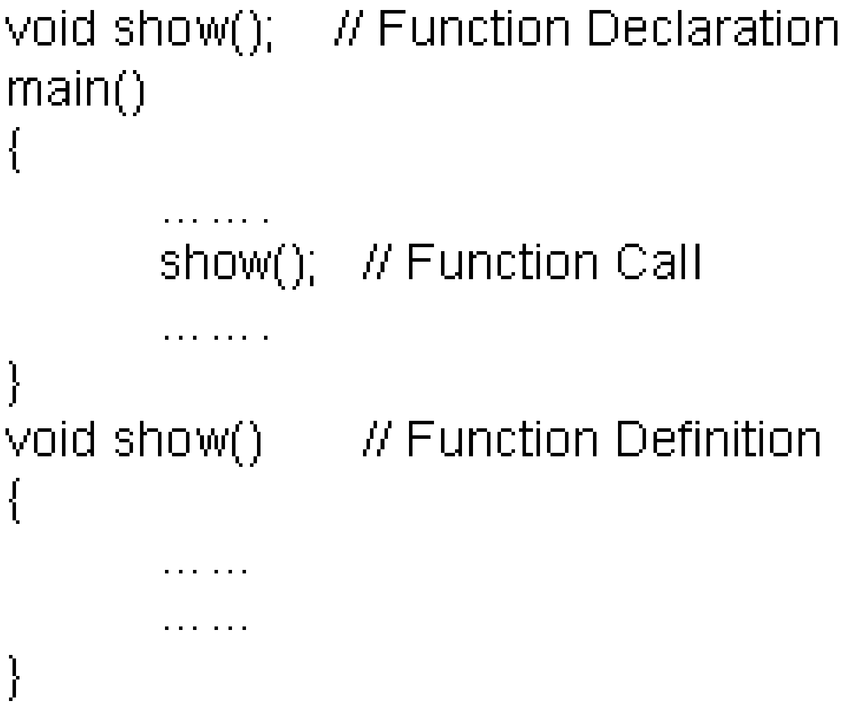


C++ program to display integer from 1 to 10 except 6 and 9.

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

int main()
{
     for (int i = 1; i <= 10; ++i)
     {
          if ( i == 6 || i == 9)
          {
               continue;
          }
          cout << i << "\t";
     }
     return 0;
}
```

FUNCTIONS

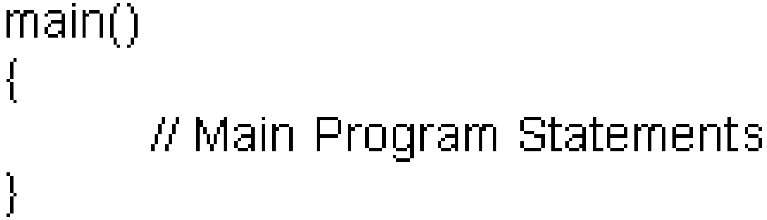
- Set of instructions that are used to performs a particular task
- Reduces the size of the program
- Help to understand the program easily
- Example:



```
void show();    // Function Declaration
main()
{
            …… … .
            show();   // Function Call
            …… … .
}
void show()       // Function Definition
{
            …… …
            …… …
}
```

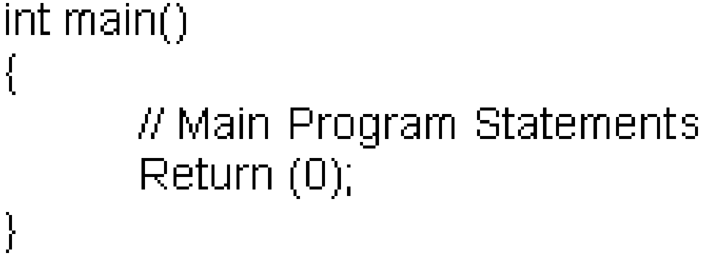
Main Function

- Starting point of the execution of program
- Definition:



```
main()
{
        // Main Program Statements
}
```

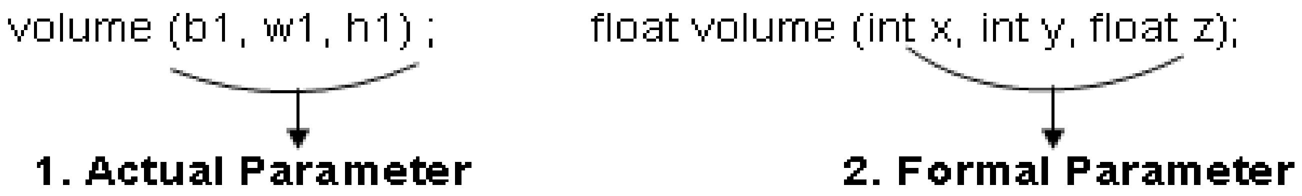
- In C++, main () returns a value of type int to operating system
- Function that have return value should use the return statement for termination
- Example:



```
int main()
{
        // Main Program Statements
        Return (0);
}
```

Function Prototyping

- Prototype describes function interface to compiler by giving details such as number, type of argument & type of return values
- Template always used when declaring & defining functions
- When function is called, compiler used template to ensure that proper arguments passed & the return value is treated correctly
- Syntax: type function-name (argument-list);
- Example: float volume (int x, int y, float z); OR
  float volume (int, int, float);
- Types of parameter used in functions as arguments



volume (b1, w1, h1) ;

1. Actual Parameter

float volume (int x, int y, float z);

2. Formal Parameter

Function Prototyping

- In a function declaration the names of the arguments are dummy variables and hence they are optional.
  - Float volume (int x, float y, float z);
  - Float volume(int, float, float);

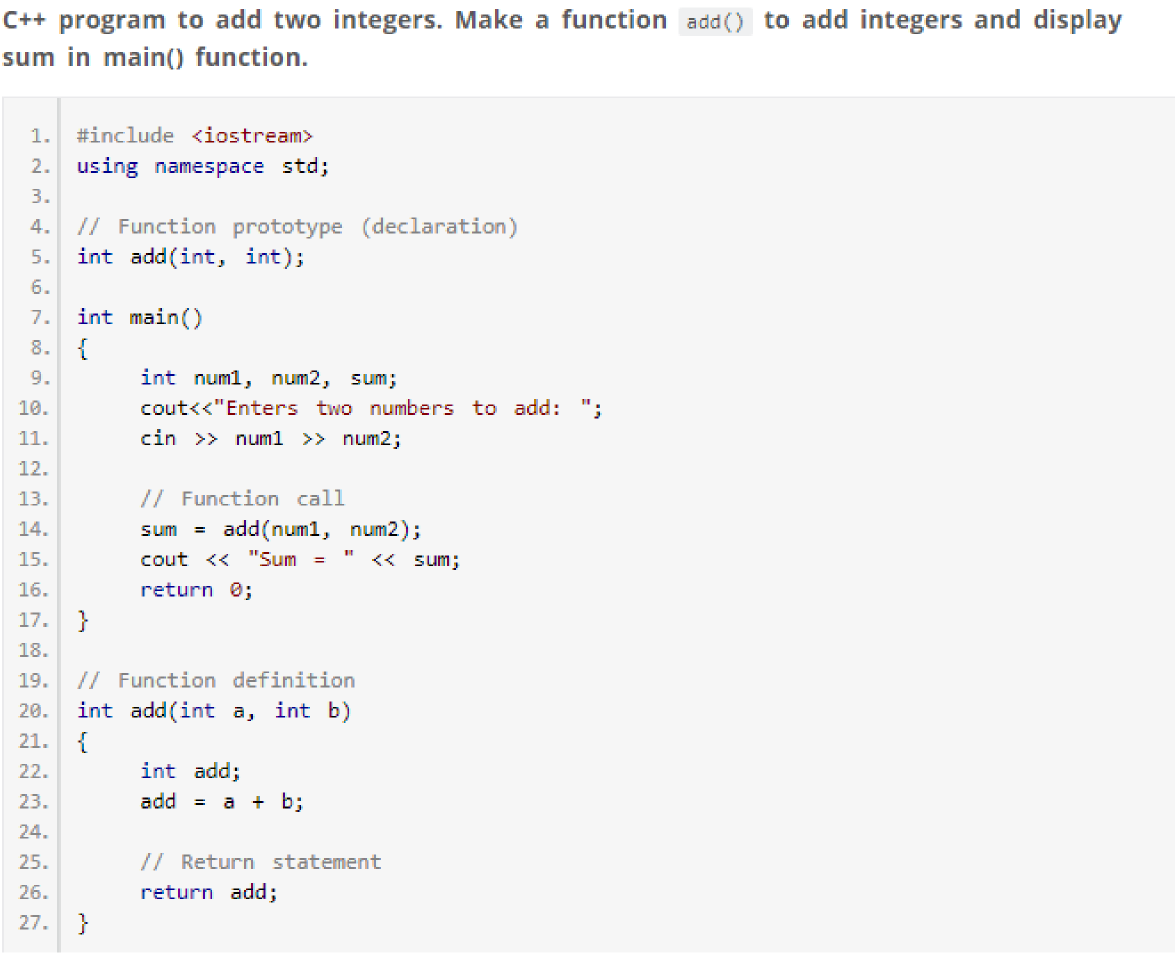
- However in the function definition, names are required because the arguments must be referenced inside the function.
  - Float volume (int a, float b, float c)

```
{
   float v = a*b*c;
}
```

Function Prototyping

- The function volume can be invoked from a program as
  - Float cube1 = voulme(b1,w1,h1);
  - The variables b1,w1,h1 are known as the actual parametered.
  - Their types, order of appearance should match with the types declared in the prototype.

Function Example

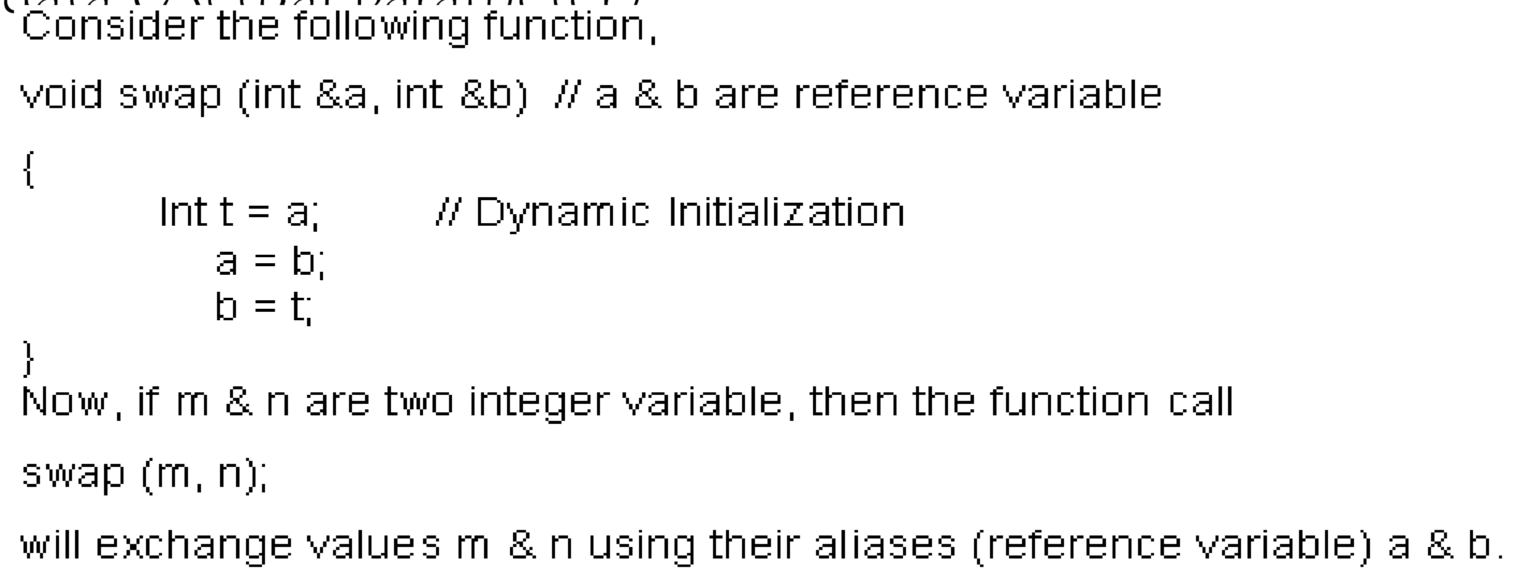


C++ program to add two integers. Make a function `add()` to add integers and display sum in main() function.

```cpp
#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;
}
```

Call by Reference

- C++ Permits us to pass parameters to functions by reference
- When the functions is working with its own arguments (formal parameter), it is actually working on the original data (Actual parameter)



Consider the following function,

```
void swap (int &a, int &b)  // a & b are reference variable

{
          Int t = a;          // Dynamic Initialization
              a = b;
              b = t;
}
```

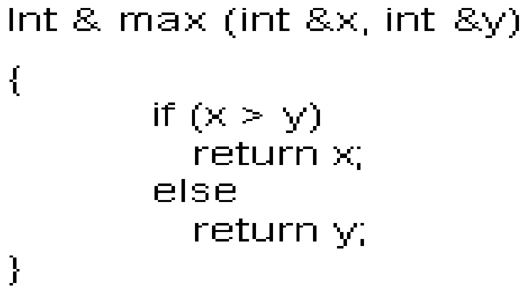
Now, if m & n are two integer variable, then the function call

```
swap (m, n);
```

will exchange values m & n using their aliases (reference variable) a & b.

Return by Reference

- Function can also return a reference
- Consider following function,



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

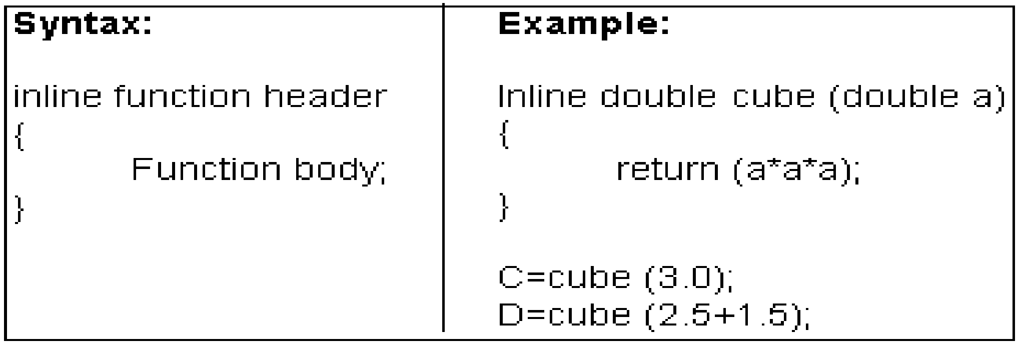
- Since return type of **max()** is **int &**, the function returns reference to x or y (and not a values). Then function call such as **max(a, b)**

  will yield reference to either a or b depending on their values.
- This means that this function call can appear on LHS of an assignment statement. That is, the statement

  max (a, b) = -1; is legal & assign -1 to a if is larger, other wise -1 to b

INLINE FUNCTION

- Used to eliminate cost of calls to small functions
- It is expanded in line when it is invoked means compiler replaces function call with the corresponding function code
- All inline functions must be defined before they are called
- Inline keyword sends request, not a command to the compiler
- Compiler may ignore request if function definition too long or complicated & compile the function as normal function



Syntax:

```
inline function header
{
        Function body;
}
```

Example:

```
Inline double cube (double a)
{
        return (a*a*a);
}

C=cube (3.0);
D=cube (2.5+1.5);
```

INLINE FUNCTION

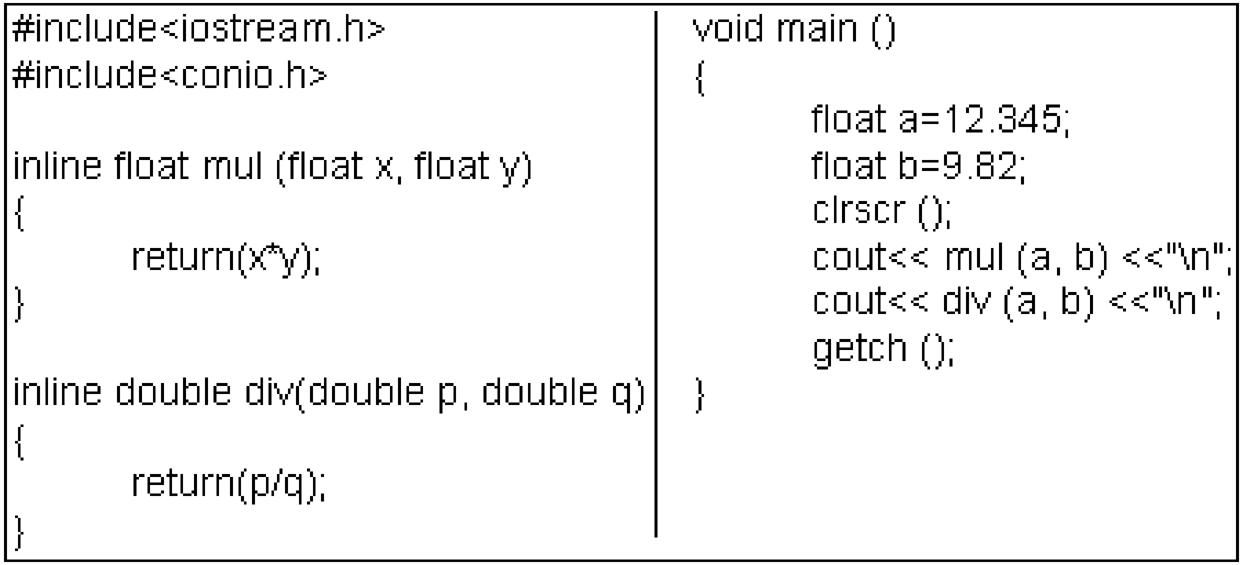
- Where inline function may not work?

  1. For fn returning values, if a loop, a switch or a goto exists

  2. If fn contain static variables

  3. If inline function are recursive

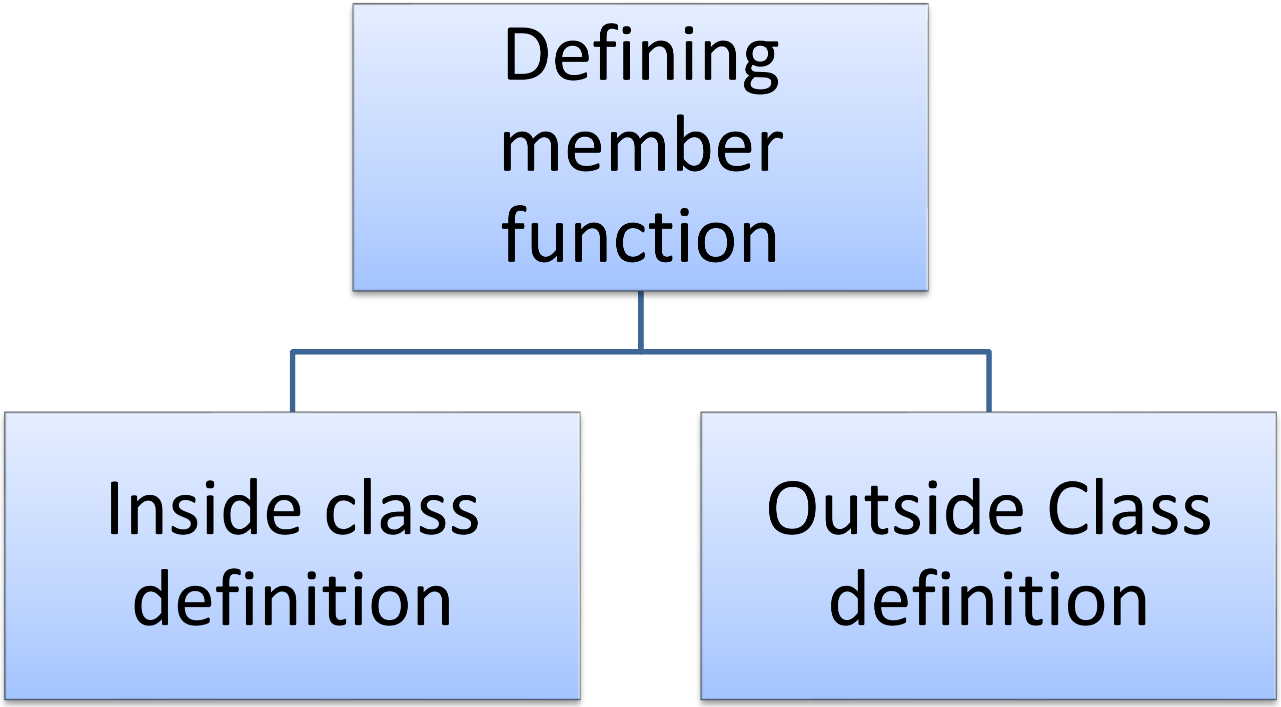
- Progra



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

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

inline double div(double p, double q)
{
        return(p/q);
}
```

```
void main ()
{
        float a=12.345;
        float b=9.82;
        clrscr ();
        cout<< mul (a, b) <<"\n";
        cout<< div (a, b) <<"\n";
        getch ();
}
```

Defining Member function



Defining member function

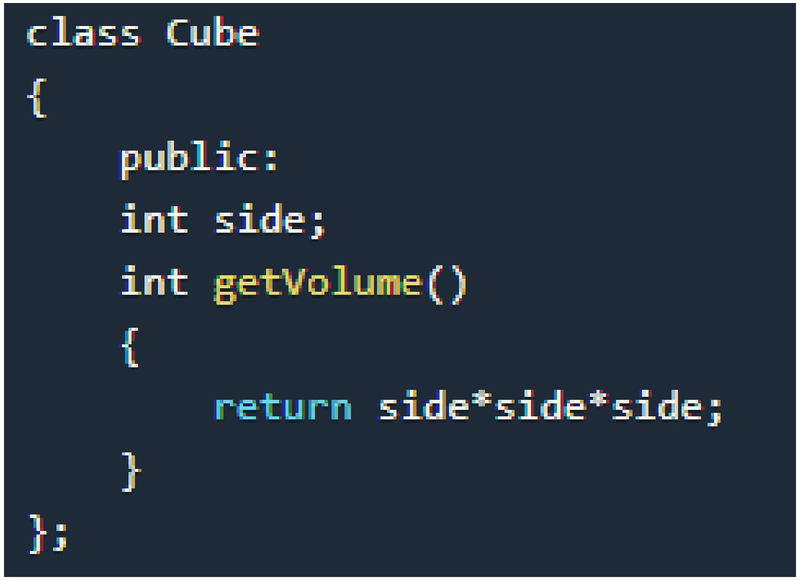
Inside class definition

Outside Class definition

Defining Member function

Inside Class Definition

If we define the function inside class then we don't not need to declare it first, we can directly define the function.

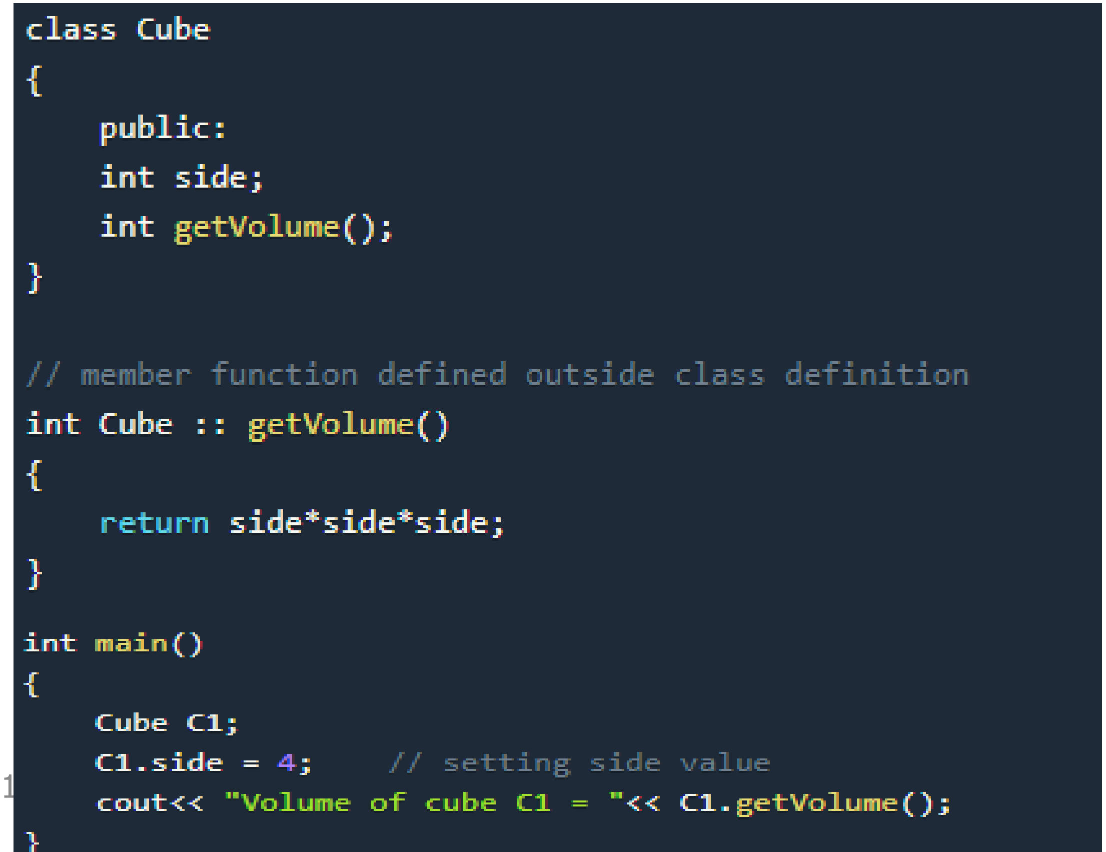


```
class Cube
{
    public:
    int side;
    int getVolume()
    {
        return side*side*side;
    }
};
```

Defining Member function

If we define member function outside of the class definition then, we must declare the function inside the class definition

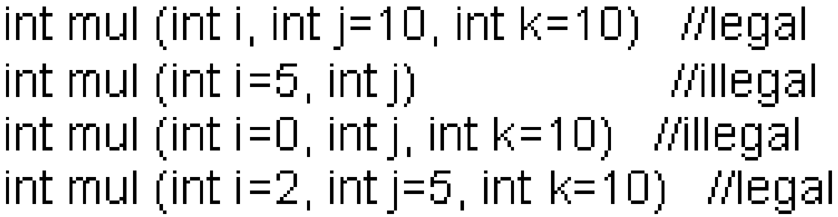
Syntax: return type Class name :: function name()



```
class Cube
{
    public:
    int side;
    int getVolume();
}

// member function defined outside class definition
int Cube :: getVolume()
{
    return side*side*side;
}

int main()
{
    Cube C1;
    C1.side = 4;    // setting side value
    cout<< "Volume of cube C1 = "<< C1.getVolume();
}
```

Default Arguments

- Allows us to call function without specifying all its arguments
- In such case fn assigns default value to parameter which does not have matching argument in fn call
- Default values are specified when function is declared
- It is checked for type at the time of declaration & evaluated at the time of call
- Only trailing arguments can have default values & therefore we must add defaults from right to left
- Useful in situations where some arguments always have same value
- Example:



```
int mul (int i, int j=10, int k=10)   //legal
int mul (int i=5, int j)               //illegal
int mul (int i=0, int j, int k=10)   //illegal
int mul (int i=2, int j=5, int k=10)   //legal
```

```
•  example: one class, two objects
•  #include <iostream>
•  Using namespace std;
•  class Crectangle
•   { int x, y;
•      public:
•        void set_values (int,int);
•        int area () {return (x*y);}
•   };
•  void CRectangle::set_values (int a, int b)
•  { x = a; y = b; }
•  int main ()
•   { CRectangle rect, rectb;
•      rect.set_values (3,4);
•      rectb.set_values (5,6);
•      cout << "rect area: " << rect.area() << endl;
•      cout << "rectb area: " << rectb.area() << endl;
•       return 0;
•  }
```

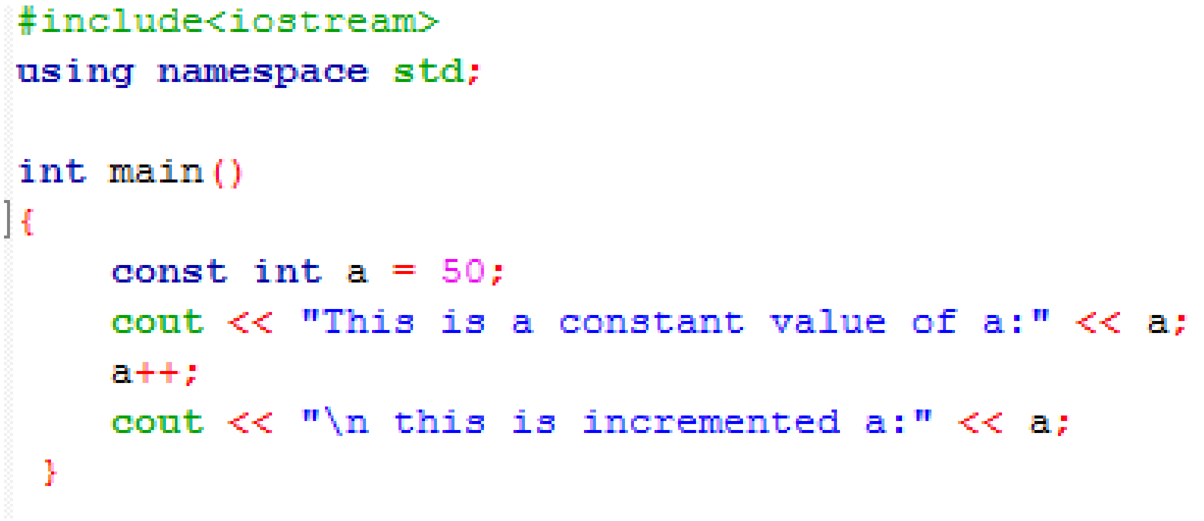
Const Arguments

- In C++, argument to function can be declared const as shown below,

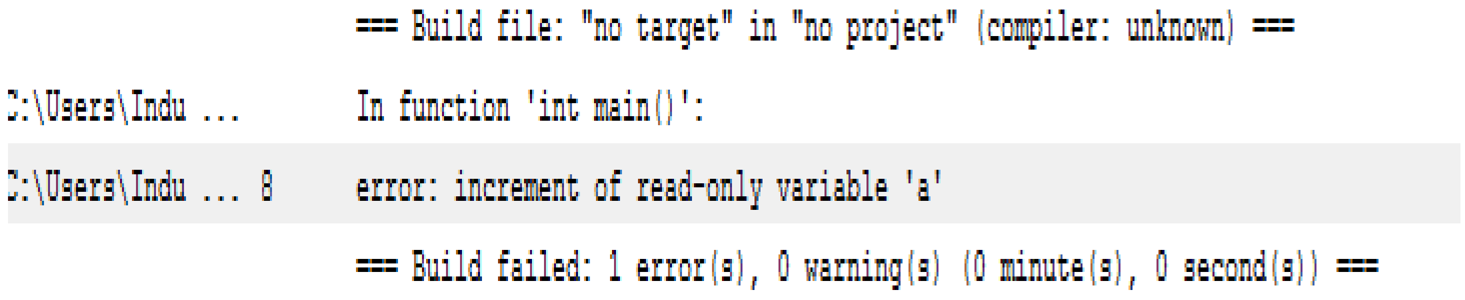
- Qualifier **const** tells the compiler that function should not modify argument.
- This type of declaration is significant only when we pass arguments by reference or pointers

- **Syntax: const int a = 50;**

Const Variables



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

int main()
{
    const int a = 50;
    cout << "This is a constant value of a:" << a;
    a++;
    cout << "\n this is incremented a:" << a;
 }
```



```
                      === Build file: "no target" in "no project" (compiler: unknown) ===
C:\Users\Indu ...     In function 'int main()':
C:\Users\Indu ... 8   error: increment of read-only variable 'a'
                      === Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===
```