# LESSION 3:

*Variable Scope, Operator and Control Flow*

# Scope of variables

Nói chung, phạm vi được định nghĩa như là mức độ mà thứ gì đó có thể được làm việc. Trong lập trình, phạm vi của một biến cũng được định nghĩa là mức độ của chương trình mà biến đố có thể được truy cập, khai báo hoặc làm việc. Có 2 phạm vi biến chính:
+ biến cục bộ : biến cục bộ thông thường và biến cục bộ tĩnh // lưu trữ trong stack
+biến toàn cục: biến toàn cục thông thường và biến toàn cục tĩnh // lưu trữ trong uninitialized data (bss)- khi khai báo mà chưa được khởi tạo hoặc initialzed data khi đã được khai báo và khởi tạo

In general, the scope is defined as the extent up to which something can be worked with.
In programming also the scope of a variable is defined as the extent of the program code within which the variable can we accessed or declared or worked with. There are mainly two types of variable scopes:
1. Local Variables
2. Global Variables

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

// global variable
int num = 100;
int main()
{
    // local variable
    // name as that of global variable
    int num = 1;
    return 0;
}
```

khai báo biến toàn cục thông thường
// khi khai báo biến toàn cục thông thường thì có nghĩa nó có thể được extern để sử dụng ở 1 file khác ~ trong khi khai báo static thì chắc chắn là chỉ được dùng ở 1 file được khai báo
// extern chỉ là thông báo sử dụng 1 biến được khai báo ở vị trí khác (chỉ là thông báo chứ -> có thể được thông báo nhiều lần)

# Scope of variables – Local

- Variables defined within a function or block are said to be local to those functions. Các biến được định nghĩa trong một hàm hoặc khối được gọi là biến cục bộ của các hàm đó.
- Anything between '{' and '}' is said to inside a block.

  Bất cứ thứ gì nằm giữa dấu '{' và '}' được gọi là bên trong một khối.
- Local variables do not exist outside the block in which they are declared, i.e. they **can not** be accessed or used outside that block and it is destroyed when program go to out of the block.

  khai báo
- Declaring local variables: Local variables are declared inside a block.

  Các biến cục bộ không tồn tại bên ngoài khối mà chúng được khai báo, tức là chúng không thể được truy cập hoặc sử dụng bên ngoài khối đó và sẽ bị hủy khi chương trình thoát ra khỏi khối.

  Khai báo biến cục bộ: Các biến cục bộ được khai báo bên trong một khối.

# Variables: Scope of variables – Local – Example

```
// CPP program to illustrate
// usage of local variables
#include<iostream>
using namespace std;

void func()
{
    // this variable is local to the
    // function func() and cannot be
    // accessed outside this function
    int age=18;
}

int main()
{
    cout<<"Age is: "<<age;

    return 0;
}
```

OUTPUT

```
Error: age was not declared in this scope
```

*Program displays an error saying "age was not declared in this scope". The variable age was declared within the function func() so it is local to that function and not visible to portion of program outside this function.*

# Variables: Scope of variables – Global

- As the name suggests, global variables can be accessed from any part of the program. Như tên gọi của nó, biến toàn cục có thể được truy cập từ bất kỳ phần nào của chương trình.
- They are available through out the life time of a program.

  Chúng có sẵn trong suốt thời gian chạy của chương trình.
- They are declared at the top of the program outside all of the functions or blocks. Chúng được khai báo ở đầu chương trình, bên ngoài tất cả các hàm hoặc khối.
- Declaring global variables: global variables are usually declared outside of all of the functions and blocks, at the top of the program. They can be accessed from any portion of the program.

  Khai báo biến toàn cục: Biến toàn cục thường được khai báo bên ngoài tất cả các hàm và khối, ở đầu chương trình. Chúng có thể được truy cập từ bất kỳ phần nào của chương trình.

# Variables: Scope of variables - Global - Example

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

// global variable
int global = 5;

// global variable accessed from
// within a function
void display()
{
    cout<<global<<endl;
}
int main()
{
    display();

    // changing value of global
    // variable from main function
    global = 10;
    display();
}
```

output

```
5
10
```

In the program, the variable “global” is declared at the top of the program outside all of the functions so it is a global variable and can be accessed or updated from anywhere in the program.

# Variables: Scope of variables – Global vs. Local

```
// C++ program to show that we can access a global
// variable using scope resolution operator :: when
// there is a local variable with same name
#include<iostream>
using namespace std;

// Global x
int x = 0;

int main()
{
  // Local x
  int x = 10;
  cout << "Value of global x is " << ::x;
  cout<< "\nValue of local x is " << x;
  return 0;
}
```

**Output:**

```
Value of global x is 0
Value of local x is 10
```

***How to access a global variable when there is a local variable with same name?***

## Variables: Scope of variables – Global - Extern

- When your program contains multiple modules that mean you must split source code into multiple files (.h,.cpp) and you want use a variable defined in a file as a global variable.

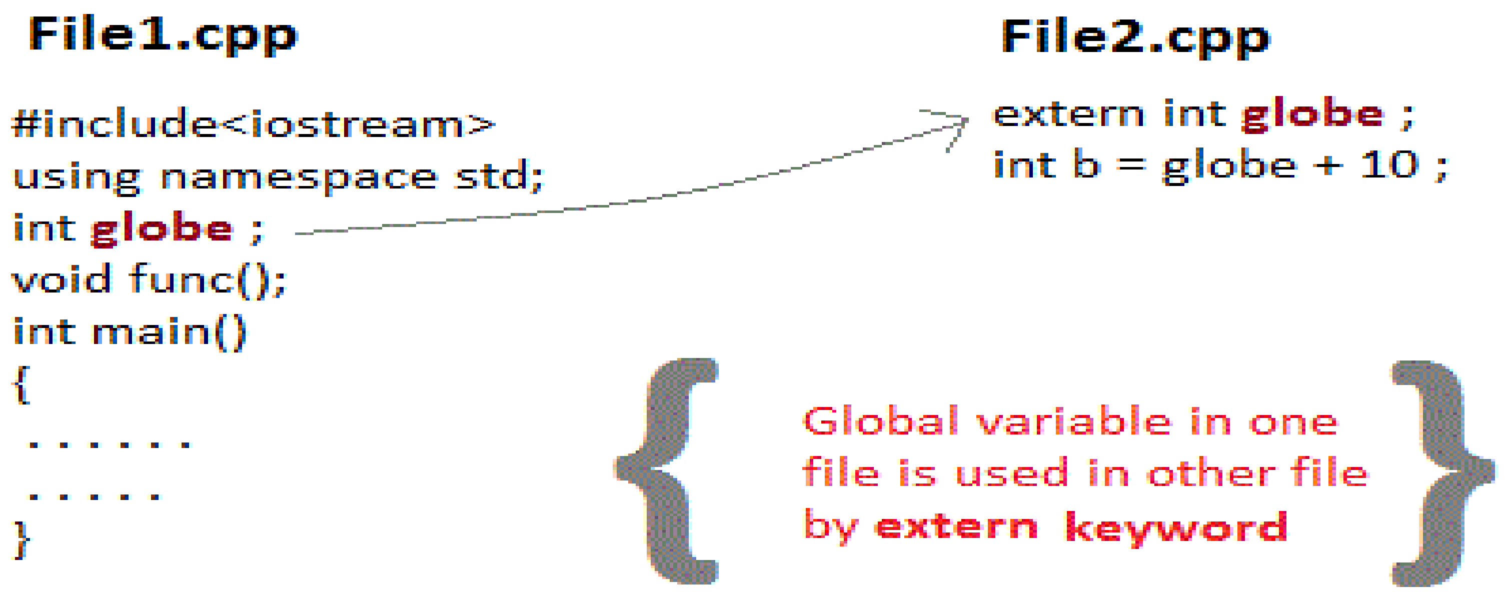
khi chương trình của bạn chứa nhiều module, có nghĩa là bạn phải chỉa mã nguồn thành nhiều tệp (.h, .cpp) và bạn muốn sử dụng một biến được định nghĩa trong một tệp như là biến toàn cục

**Extern variable** can solve this problem biến extern có thể giải quyết vấn đề này

giải quyết

# Variables: Scope of variables – Global - Extern

- Example:



File1.cpp

```
#include<iostream>
using namespace std;
int globe ;
void func();
int main()
{
 .......
 .....
}
```

File2.cpp

```
extern int globe ;
int b = globe + 10 ;
```

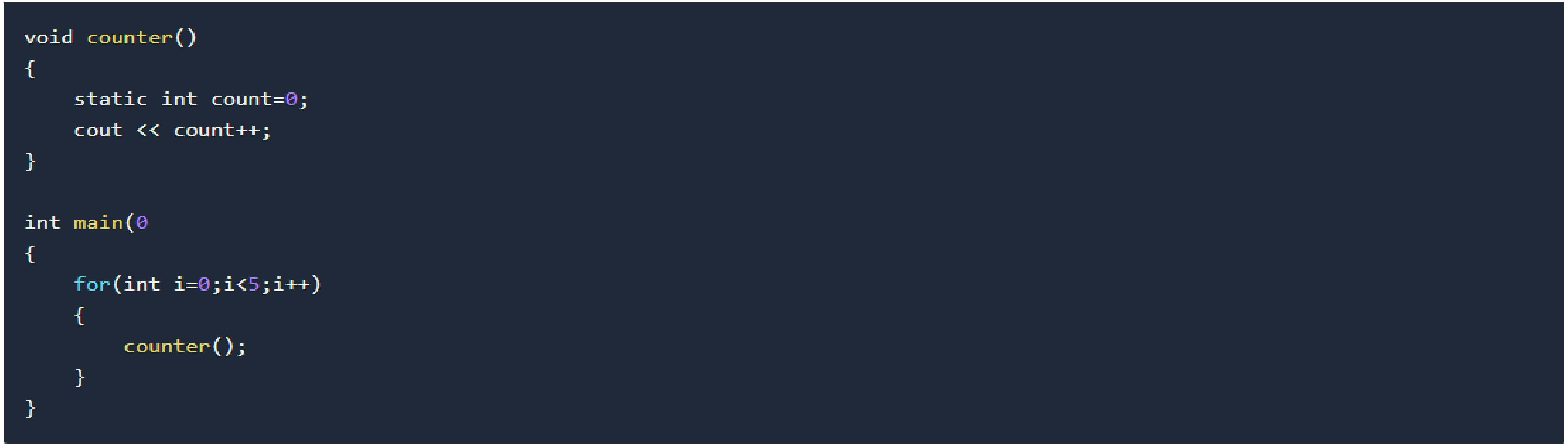
Global variable in one file is used in other file by **extern keyword**

biến global trong 1 file được sử dụng ở các file khác thông qua extern

# Variables: Static

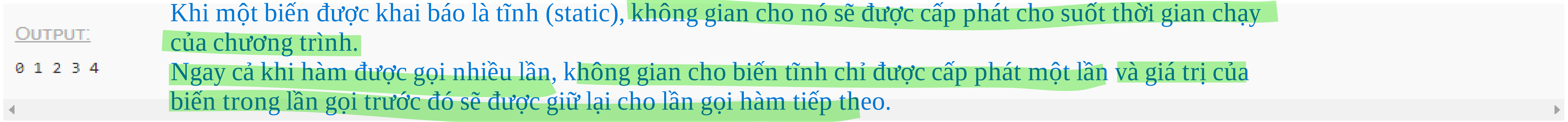
cấp phát

- 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.



```
void counter()
{
    static int count=0;
    cout << count++;
}

int main(0
{
    for(int i=0;i<5;i++)
    {
        counter();
    }
}
```



OUTPUT:

```
0 1 2 3 4
```

Khi một biến được khai báo là tĩnh (static), không gian cho nó sẽ được cấp phát cho suốt thời gian chạy của chương trình.
Ngay cả khi hàm được gọi nhiều lần, không gian cho biến tĩnh chỉ được cấp phát một lần và giá trị của biến trong lần gọi trước đó sẽ được giữ lại cho lần gọi hàm tiếp theo.

# NameSpace: Definition

- **Definition:**
  - ✓ Namespaces allow us to group named entities that otherwise would have *global scope* into narrower scopes, giving them *namespace scope*. This allows organizing the elements of programs into different logical scopes referred to by names.

Namespace cho phép chúng ta nhóm các thực thể có tên(như biến, hàm, lớp,...)mà nếu không sẽ có phạm vi toàn cục, nhóm chúng vào các phạm vi hẹp hơn gọi là phạm vi namespace. Điều này cho phép tổ chức các phần tử của chương trinwhf thành các phạm vi logic khác nhau được tham chiếu bằng tên

- **Format:**

```
namespace namespace_name
{
   int x, y; // code declarations where
             // x and y are declared in
             // namespace_name's scope
}
```

namespace trong C++ không phải là một biến, mà là một cách để nhóm- tổ chức mã nguồn các tên biến, hàm, lớp, và các khai báo khác vào một phạm vi riêng biệt, nhằm tránh xung đột tên.

việc khai báo namespace trong .h và định nghĩa namespace trong .cpp là phổ biến:
+File .h: Tệp tiêu đề MyNamespace.h chứa khai báo của các thành phần trong namespace MyNamespace. Chúng được khai báo ở đây để có thể được sử dụng ở các tệp khác.
+File .cpp: Tệp nguồn MyNamespace.cpp chứa định nghĩa của các thành phần đã khai báo trong namespace MyNamespace.

- **Rules:**
  - ✓ Namespace declarations appear only at global scope. việc khai báo namespace chỉ được thực hiện ở phạm vi toàn cục
  - ✓ Namespace declarations can be nested within another namespace. khai báo namespace có thể lồng ở bên trong namespace khác
  - ✓ Namespace declarations don't have access specifiers. (Public or private) namespace không có các từ khóa truy cập như public hoặc private
  - ✓ No need to give semicolon after the closing brace of definition of namespace. không có các dấu chấm phẩy đằng sau ngoặc đóng của định nghĩa namespace
  - ✓ We can split the definition of namespace over several units.

có thể chia định nghĩa namespace thành nhiều đơn vị (có thể định nghĩa các phần của 1 namespace trong nhiều file haowjc nhiều phần khác nhau của cùng 1 file)

# NameSpace: Accessing

4 Anonymous namespace là một namespace không có tên, các thành phần bên trong namespace này chỉ có thể được truy cập trong cùng một tệp:

```
namespace {
    int myVariable;
    void myFunction() {
        // Code
    }
}

int main() {
    myVariable = 20;
    myFunction();
    return 0;
}
```

```
#include <iostream>
using namespace std;
namespace Mine
{
      int a;
}
int main()
{
      using namespace
      Mine::a = 140;
      cout << "Value of a = " << Mine::a << endl;
      return 0;
}
```

có thể lồng namespace bên trong nhau để tổ chức mã nguồn theo cấu trúc phân cấp:

1
```
namespace OuterNamespace {
    namespace InnerNamespace {
        int myVariable;
        void myFunction() {
            // Code}}}
```

2 truy cập vào namespace lồng nhau

```
int main() {
    OuterNamespace::InnerNamespace::myVariable = 10;
    OuterNamespace::InnerNamespace::myFunction();
    return 0;
}
```

```
#include <iostream>
using namespace std;
namespace Mine
{
      int a;
}
int main()
{
      using namespace Mine;
      a= 140;
      cout << "Value of a = " << a << endl;
      return 0;
}
```

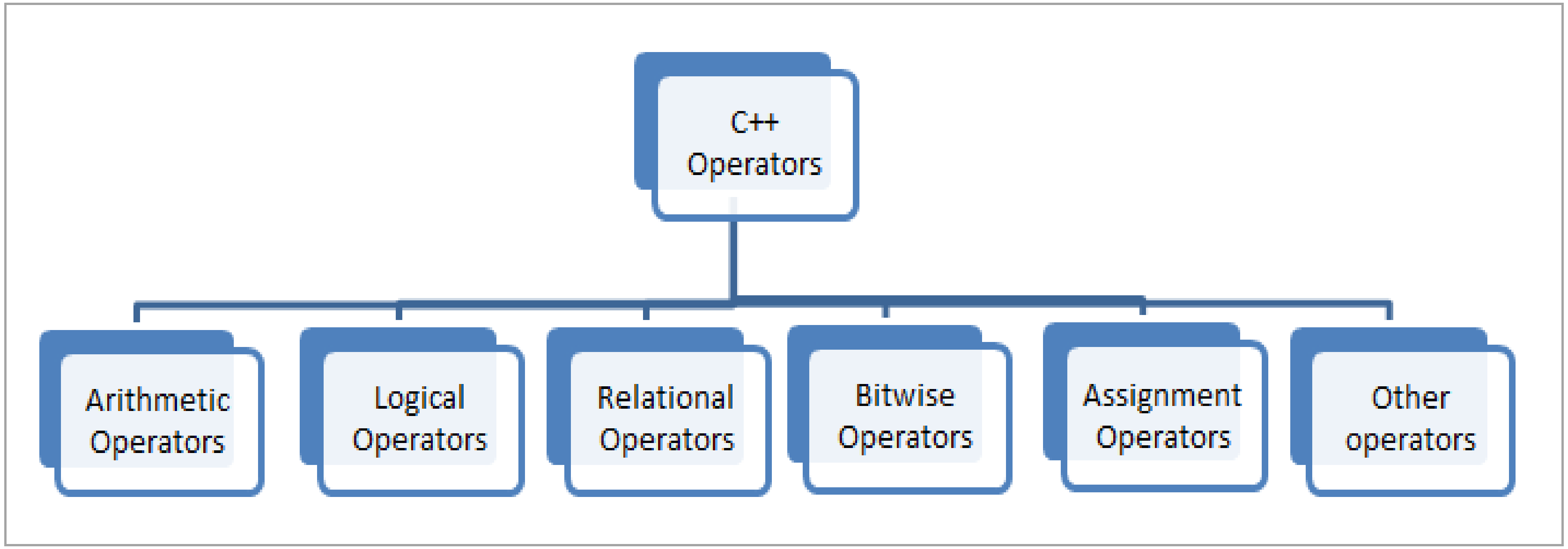
5 trong một namespace có thể khai báo và định nghĩa một function.

các biến được khai báo trong namespace thường là biến toàn cục nhưng cũng có thể là biến cục bộ nếu nó sử dụng trong 1 function được khai báo trong namespace

3 một file có thể chứa nhiều namespace khác tên, và các biến trong các namespace đó có thể có tên giống nhau mà không gây xung đột.

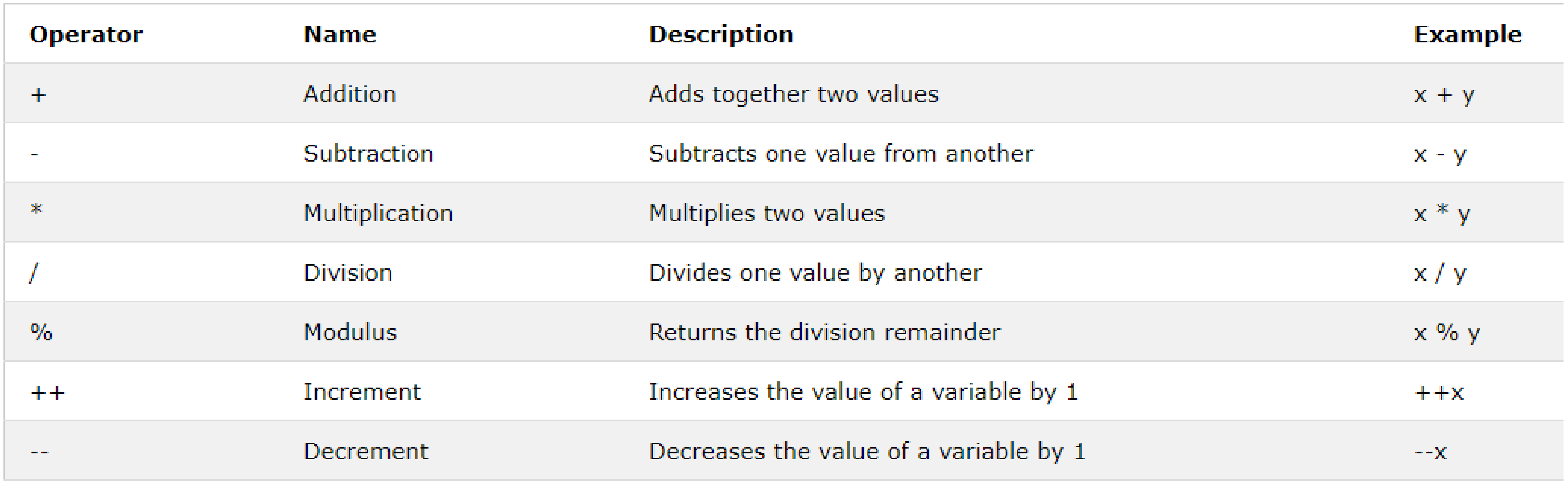
# Operator

- C++ divides the operators into the following groups:



# Operator: Arithmetic Operator

- Arithmetic operators are used to perform common mathematical operations.



| Operator | Name | Description | Example |
|---|---|---|---|
| + | Addition | Adds together two values | x + y |
| - | Subtraction | Subtracts one value from another | x - y |
| * | Multiplication | Multiplies two values | x * y |
| / | Division | Divides one value by another | x / y |
| % | Modulus | Returns the division remainder | x % y |
| ++ | Increment | Increases the value of a variable by 1 | ++x |
| -- | Decrement | Decreases the value of a variable by 1 | --x |

phân biệt ++n và n++:

1. Pre-increment (++n): Giá trị của n được tăng lên trước, sau đó giá trị mới được sử dụng trong biểu thức. Điều này có nghĩa là giá trị được trả về và sử dụng ngay lập tức là giá trị đã được tăng. Ví dụ:

int n = 5;

int a = ++n; // n trở thành 6, a cũng là 6

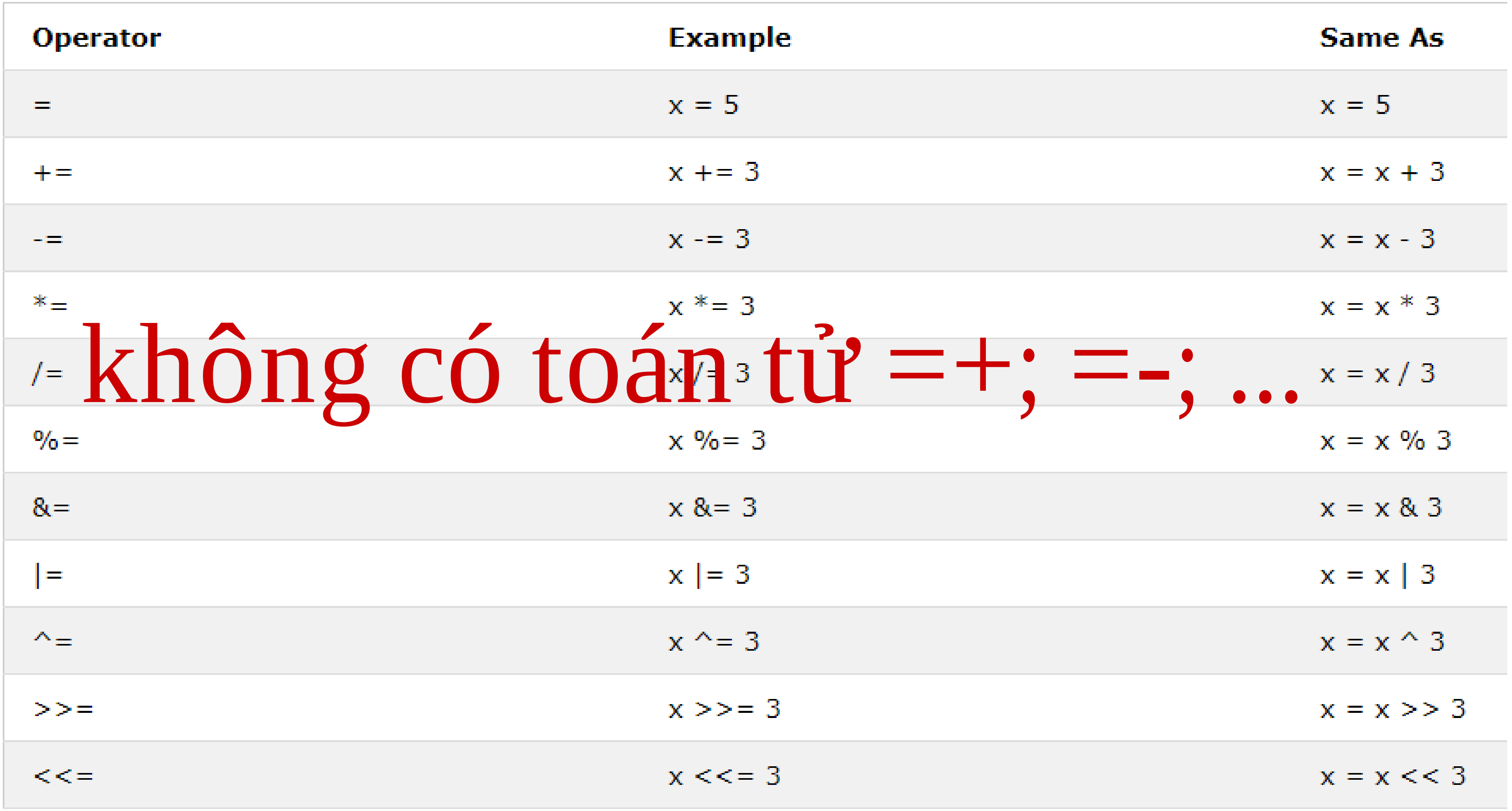
Post-increment (n++): Giá trị hiện tại của n được sử dụng trong biểu thức trước, sau đó giá trị của n mới được tăng lên. Điều này có nghĩa là giá trị được trả về và sử dụng trong biểu thức là giá trị ban đầu, trước khi tăng. Ví dụ:

int n = 5;

int a = n++; // a là 5, sau đó n trở thành 6

# Operator: Assignment Operator

- Assignment operators are used to assign values to variables.

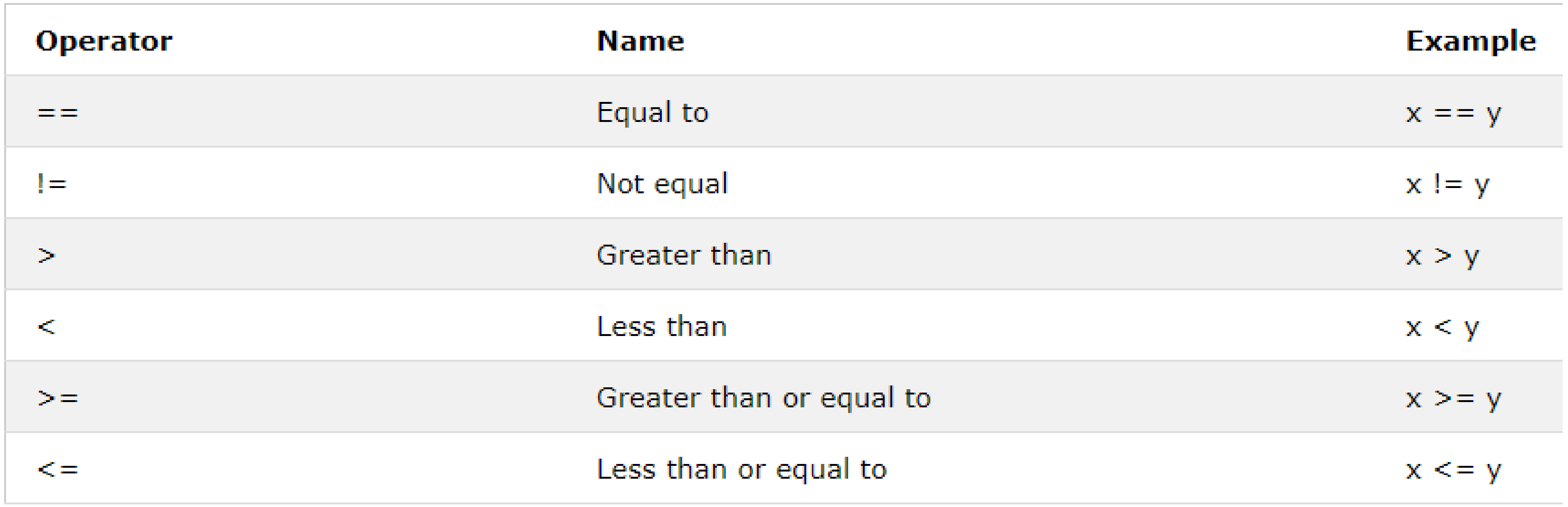


| Operator | Example | Same As |
|---|---|---|
| = | x = 5 | x = 5 |
| += | x += 3 | x = x + 3 |
| -= | x -= 3 | x = x - 3 |
| *= | x *= 3 | x = x * 3 |
| /= | x /= 3 | x = x / 3 |
| %= | x %= 3 | x = x % 3 |
| &= | x &= 3 | x = x & 3 |
| \|= | x \|= 3 | x = x \| 3 |
| ^= | x ^= 3 | x = x ^ 3 |
| >>= | x >>= 3 | x = x >> 3 |
| <<= | x <<= 3 | x = x << 3 |

không có toán tử =+; =-; ...

# Operator: Comparison Operator

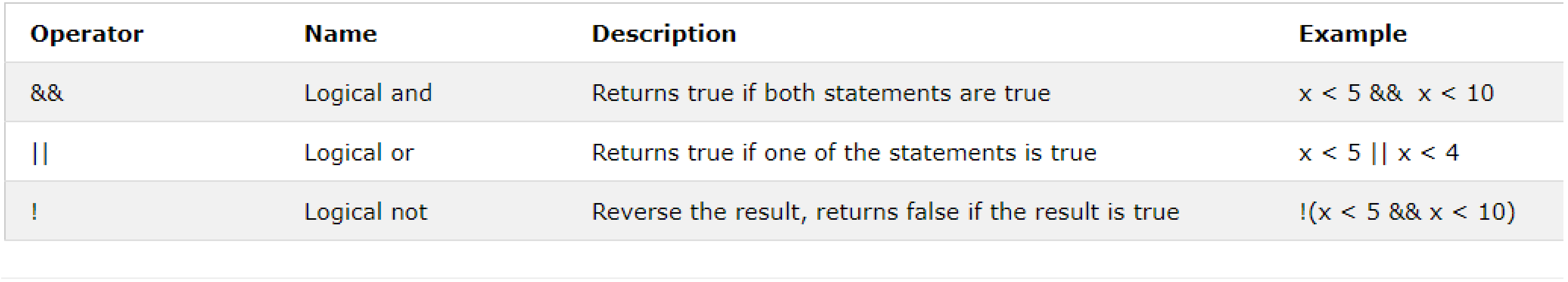
- Comparison operators are used to compare two values. The return value of a comparison is either true (1) or false (0).



| Operator | Name | Example |
|---|---|---|
| == | Equal to | x == y |
| != | Not equal | x != y |
| > | Greater than | x > y |
| < | Less than | x < y |
| >= | Greater than or equal to | x >= y |
| <= | Less than or equal to | x <= y |

# Operator: Logiccal Operator

- Logical operators are used to determine the logic between variables or values.

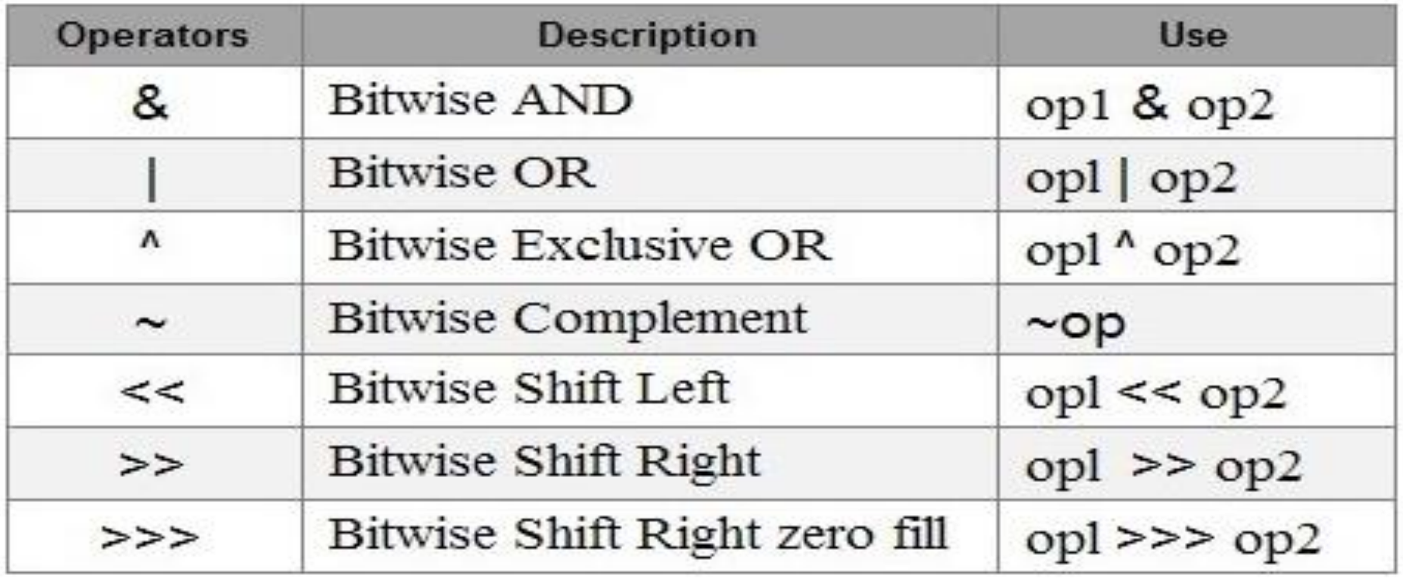


| Operator | Name | Description | Example |
|---|---|---|---|
| && | Logical and | Returns true if both statements are true | x < 5 &&  x < 10 |
| \|\| | Logical or | Returns true if one of the statements is true | x < 5 \|\| x < 4 |
| ! | Logical not | Reverse the result, returns false if the result is true | !(x < 5 && x < 10) |

- Note:
  - ✓ With AND operator if one of statement is false it will return false value immediately and don’t need check remain statements.
  - ✓ With OR operator if one of statement is true it will return true value immediately and don’t need check remain statements.

# Operator: Bitwise Operator

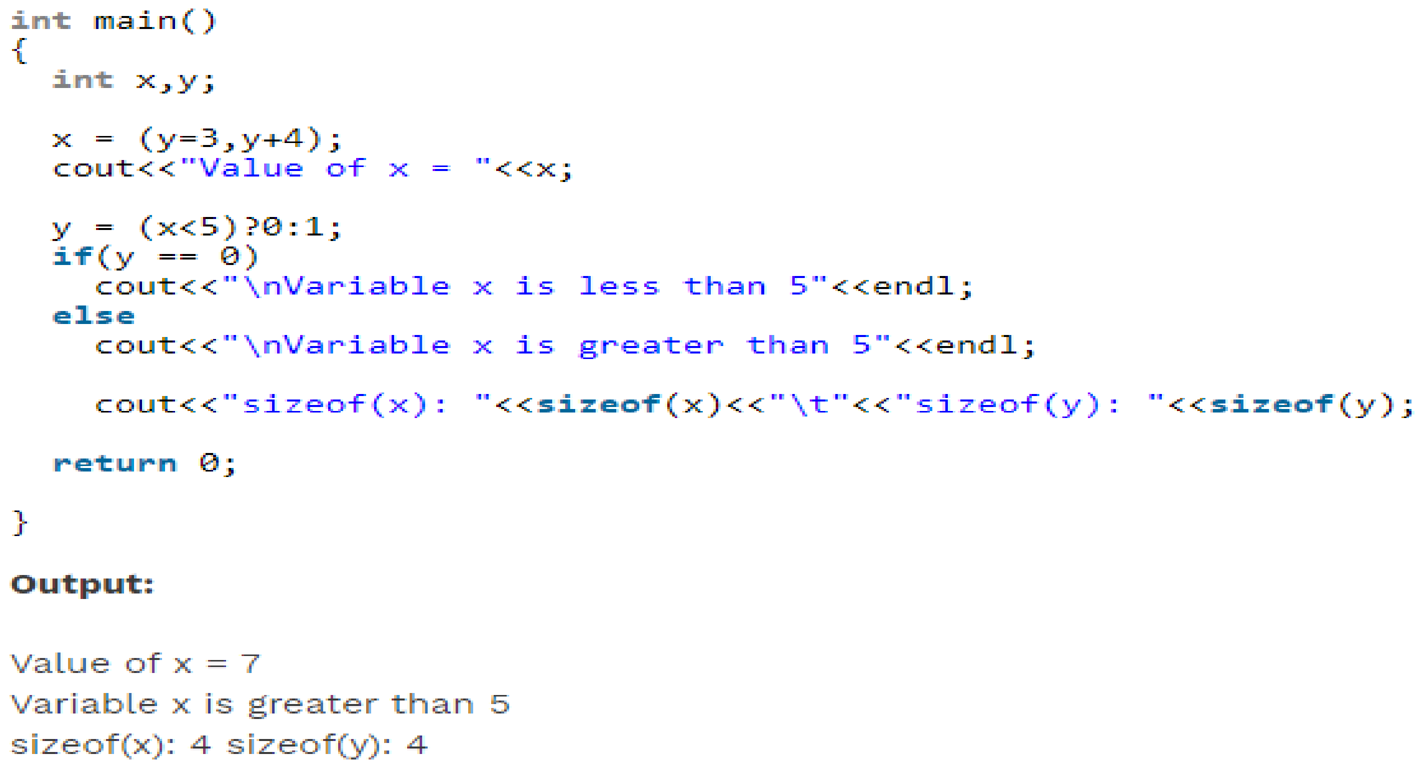
- The Bitwise operators are used to perform manipulation of individual bits of a number



| Operators | Description | Use |
|---|---|---|
| & | Bitwise AND | op1 & op2 |
| \| | Bitwise OR | opl \| op2 |
| ^ | Bitwise Exclusive OR | opl ^ op2 |
| ~ | Bitwise Complement | ~op |
| << | Bitwise Shift Left | opl << op2 |
| >> | Bitwise Shift Right | opl >> op2 |
| >>> | Bitwise Shift Right zero fill | opl >>> op2 |

# Operator: Other Operator

- Sizeof Operator
- Conditional Ternary Operator
- Comma Operator
- Member Access Operator: operator (.) and arrow (->) operator



```
int main()
{
  int x,y;

  x = (y=3,y+4);
  cout<<"Value of x = "<<x;

  y = (x<5)?0:1;
  if(y == 0)
    cout<<"\nVariable x is less than 5"<<endl;
  else
    cout<<"\nVariable x is greater than 5"<<endl;

    cout<<"sizeof(x): "<<sizeof(x)<<"\t"<<"sizeof(y): "<<sizeof(y);

  return 0;

}
```

**Output:**

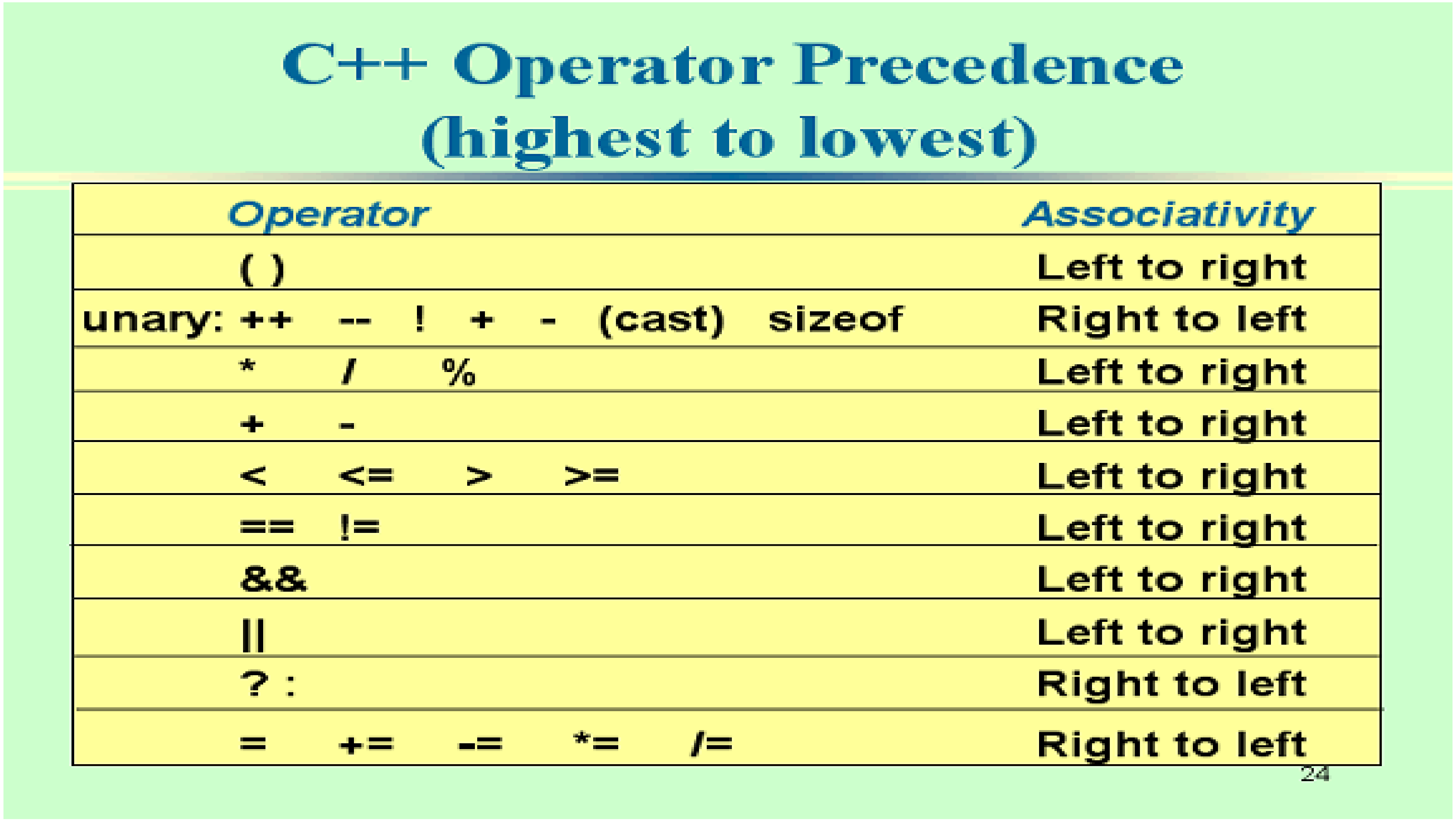
```
Value of x = 7
Variable x is greater than 5
sizeof(x): 4 sizeof(y): 4
```

## Operator: Operator Precedence

- **Problem**:
  - ✓ Such complex expressions will have more than one operator and many operands. In such a situation, we need to evaluate which operator is to be evaluated first.

- **Rule**:
  - ✓ C++ has defined precedence for all the operators and the operators with higher precedence are evaluated first.
  - ✓ Associativity tells the compiler whether to evaluate an expression in left to right sequence or right to left sequence. Thus using precedence and associativity of an operator we can effectively evaluate an expression and get the desired result.

# Operator: Operator Precedence



C++ Operator Precedence
(highest to lowest)

| Operator | Associativity |
|---|---|
| ( ) | Left to right |
| unary: ++ -- ! + - (cast) sizeof | Right to left |
| * / % | Left to right |
| + - | Left to right |
| < <= > >= | Left to right |
| == != | Left to right |
| && | Left to right |
| \|\| | Left to right |
| ? : | Right to left |
| = += -= *= /= | Right to left |

24

## Control Flow: if

- Flow chart:

Syntax:

Flowchart of if Statement

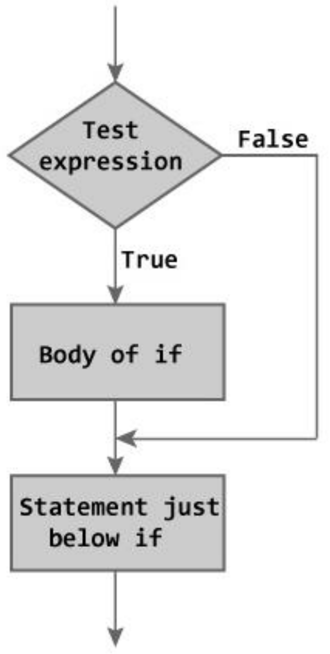
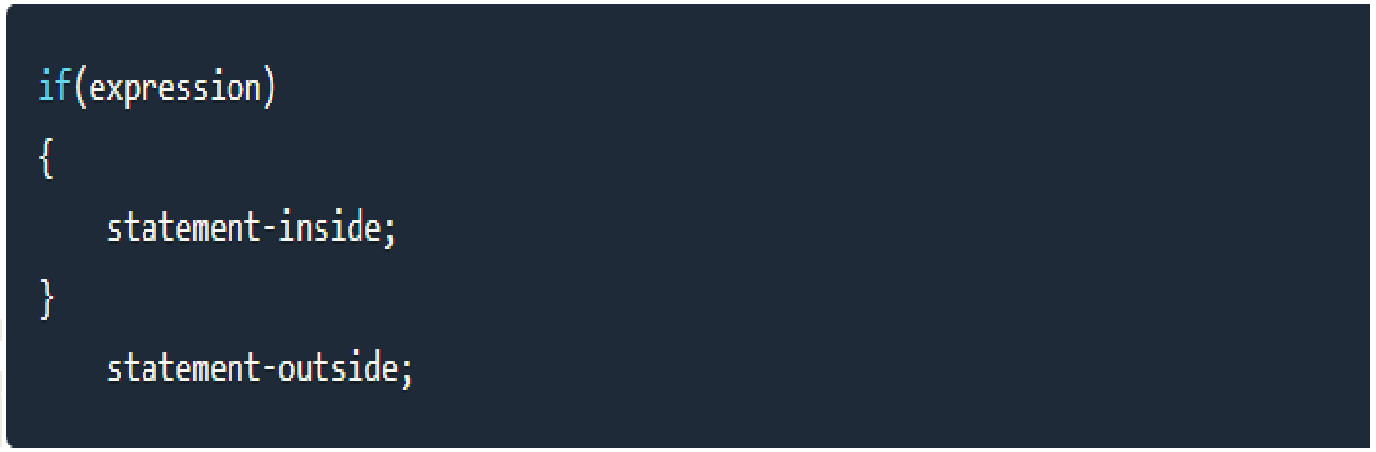


Figure: Flowchart of if Statement



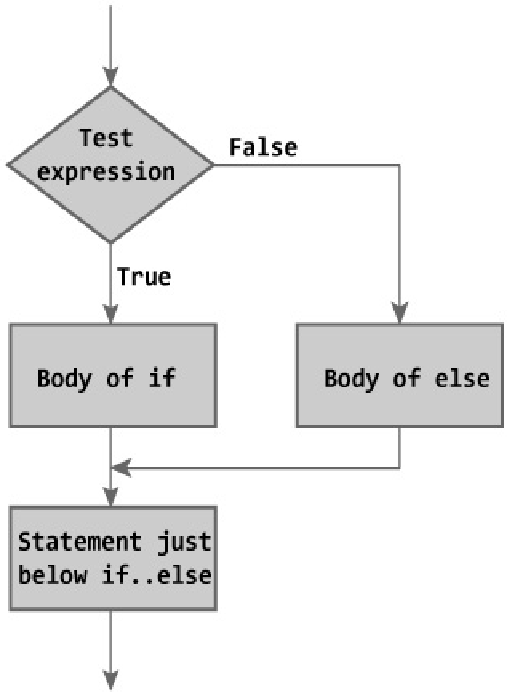
```
if(expression)
{
    statement-inside;
}
    statement-outside;
```

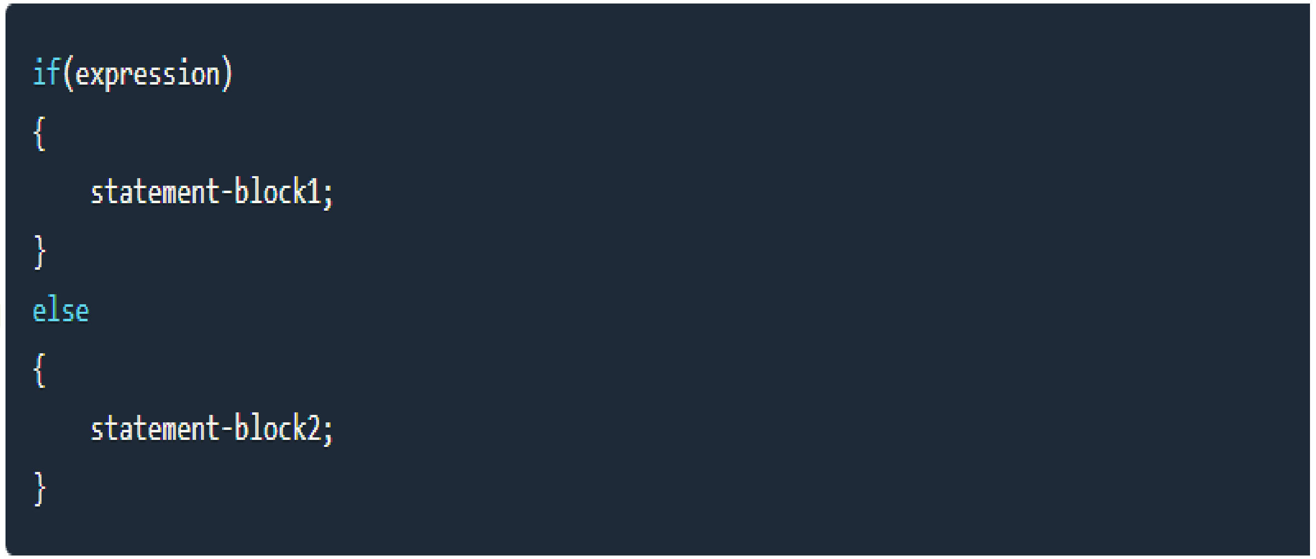
# Control Flow: if…else

- Flow chart:

Syntax:

Flowchart of if...else

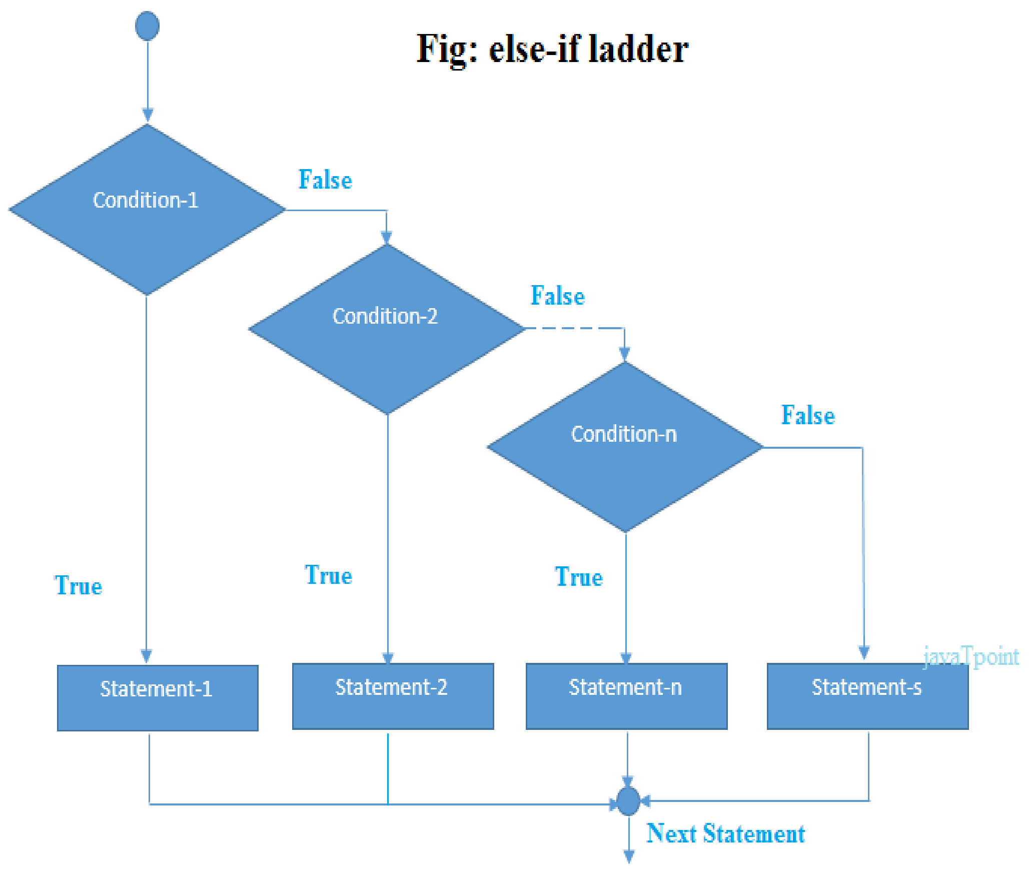




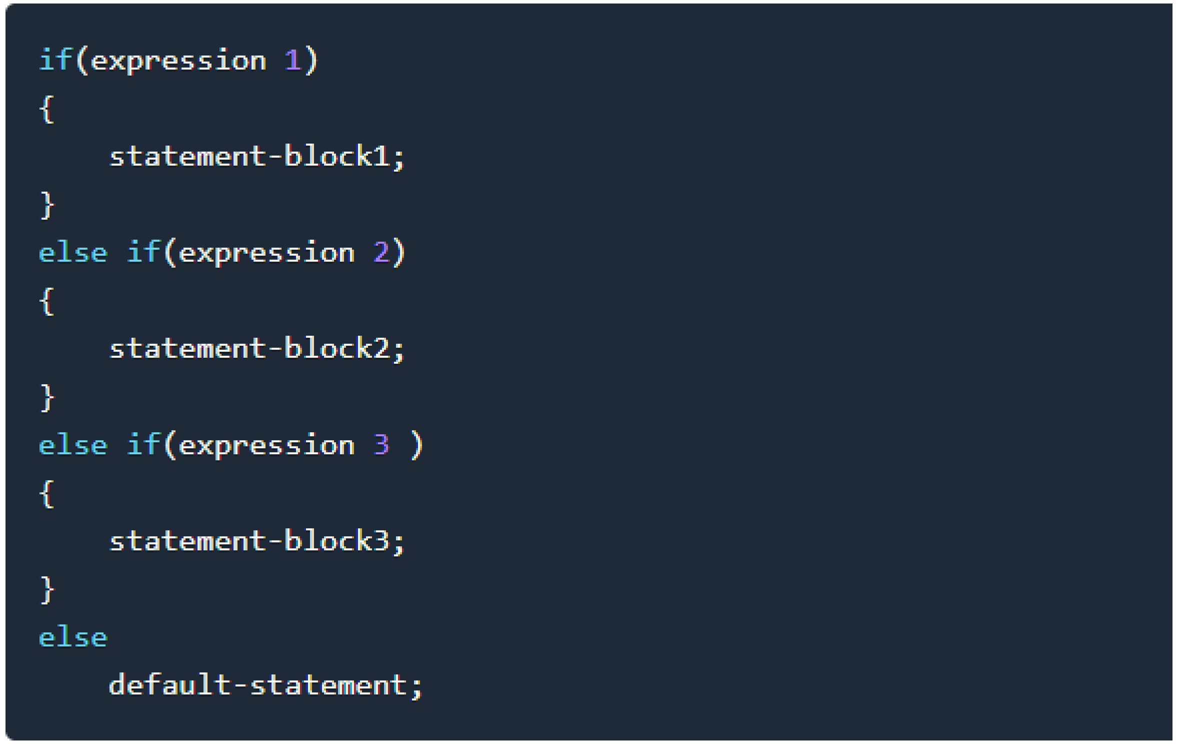
```
if(expression)
{
    statement-block1;
}
else
{
    statement-block2;
}
```

# Control Flow: if…else ladder

- Flow chart:



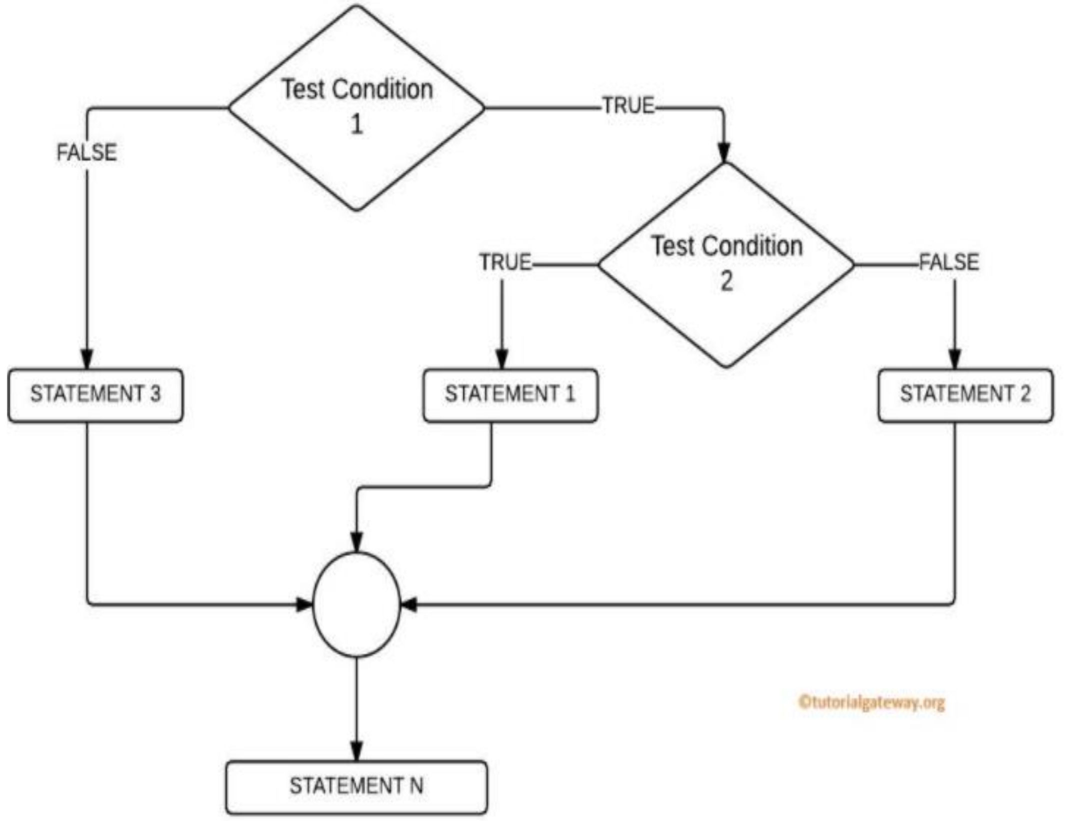
Syntax:



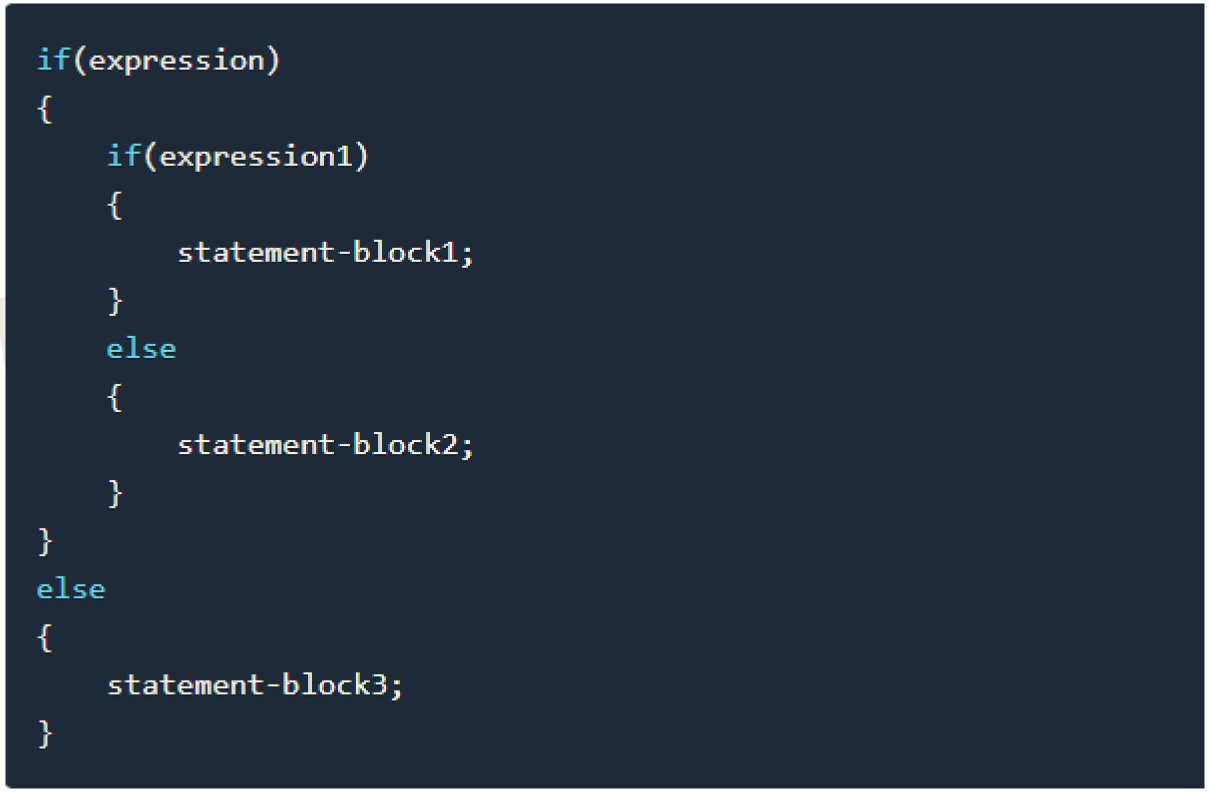
```
if(expression 1)
{
     statement-block1;
}
else if(expression 2)
{
     statement-block2;
}
else if(expression 3 )
{
     statement-block3;
}
else
     default-statement;
```

# Control Flow: Nested if…else

- Flow chart:



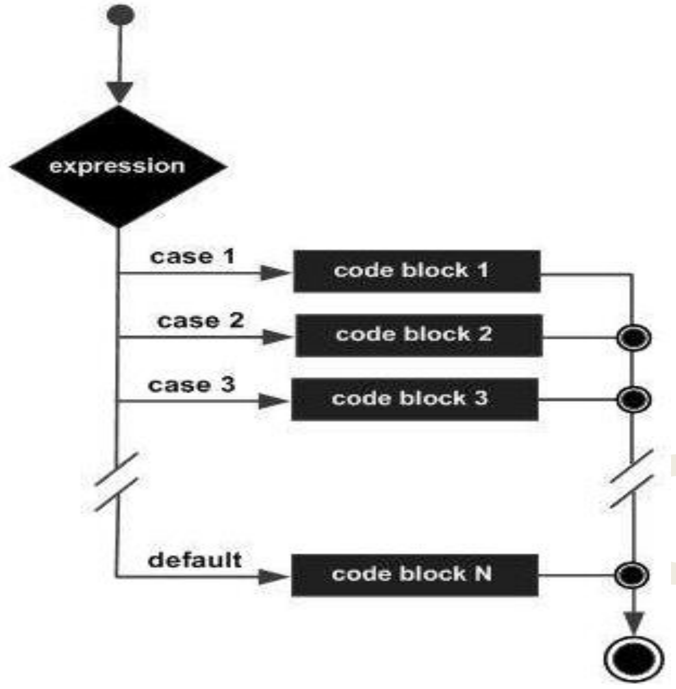
Syntax:



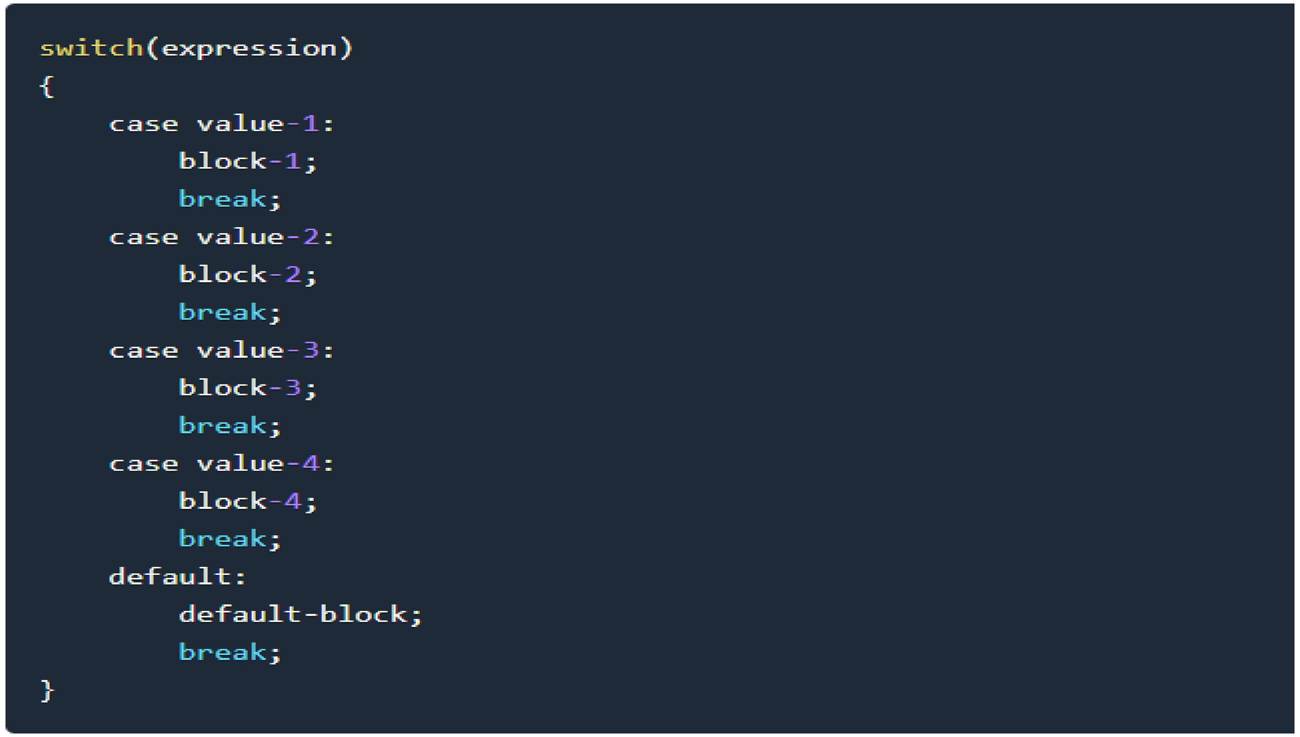
```
if(expression)
{
    if(expression1)
    {
        statement-block1;
    }
    else
    {
        statement-block2;
    }
}
else
{
    statement-block3;
}
```

# Control Flow: Switch …Case

- Flow chart:



Syntax:

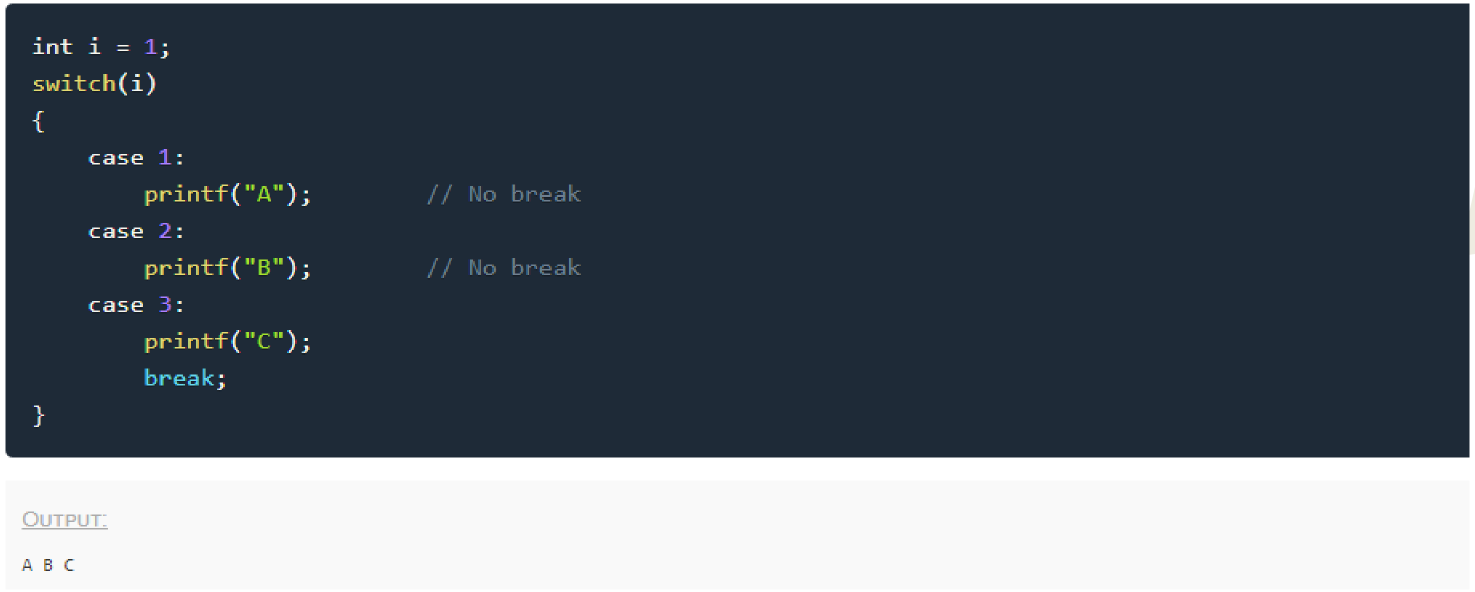


```
switch(expression)
{
     case value-1:
          block-1;
          break;
     case value-2:
          block-2;
          break;
     case value-3:
          block-3;
          break;
     case value-4:
          block-4;
          break;
     default:
          default-block;
          break;
}
```

- ✓ The expression (after switch keyword) must yield an integer value i.e the expression should be an integer or a variable or an expression that evaluates to an integer.
- ✓ The case label values must be unique.
- ✓ The case label must end with a colon(:)

## Control Flow: Switch …Case

- **Points To Remember:**
  - ✓ **break** statements are used to exit the switch block. It isn't necessary to use break after each block, but if you do not use it, then all the consecutive blocks of code will get executed after the matching block.



```
int i = 1;
switch(i)
{
    case 1:
        printf("A");        // No break
    case 2:
        printf("B");        // No break
    case 3:
        printf("C");
        break;
}
```

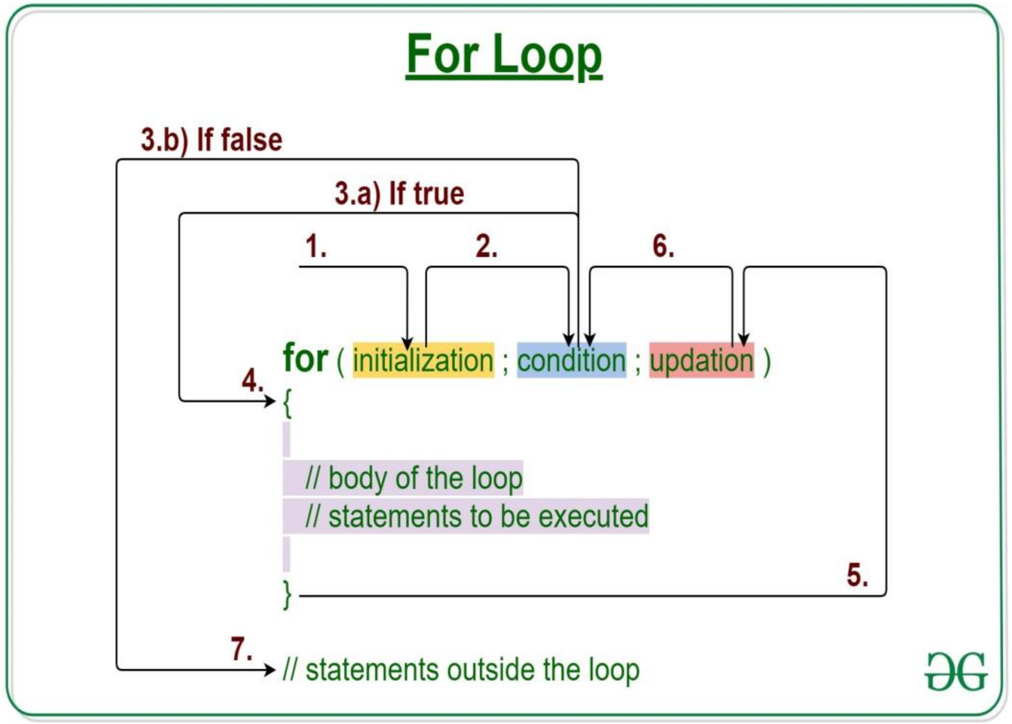
OUTPUT:

```
A B C
```

- **Default** case is executed when none of the mentioned case matches the switch expression. The default case can be placed anywhere in the switch case. Even if we don't include the default case, switch statement works.

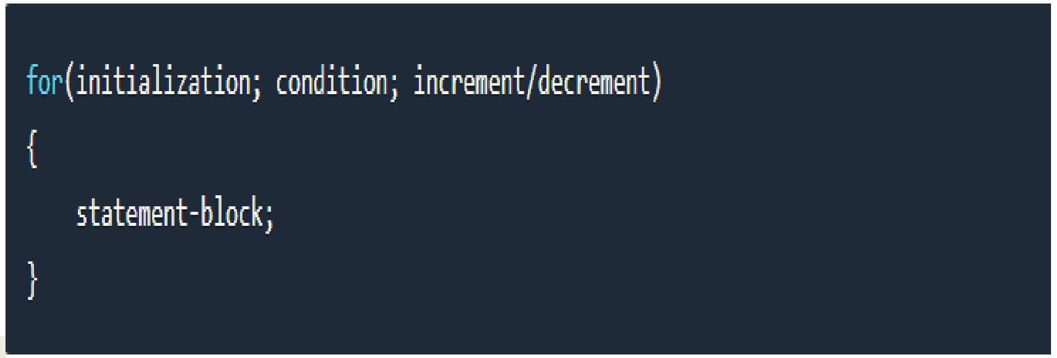
# Control Flow: For Loop

- Flow chart:



For Loop
3.b) If false
3.a) If true
1.
2.
6.
for ( initialization ; condition ; updation )
4.
{
// body of the loop
// statements to be executed
5.
}
7.
// statements outside the loop

Syntax:

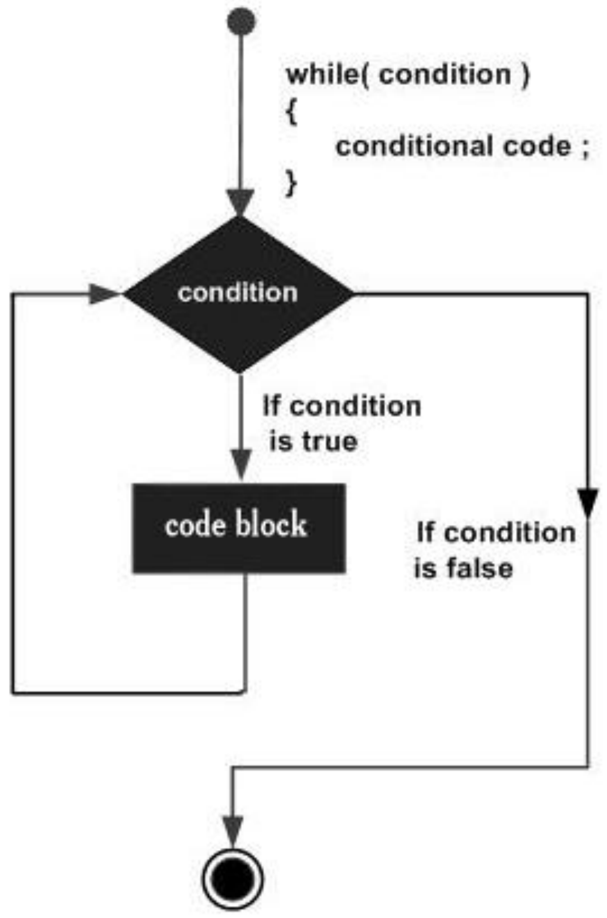


```
for(initialization; condition; increment/decrement)
{
    statement-block;
}
```

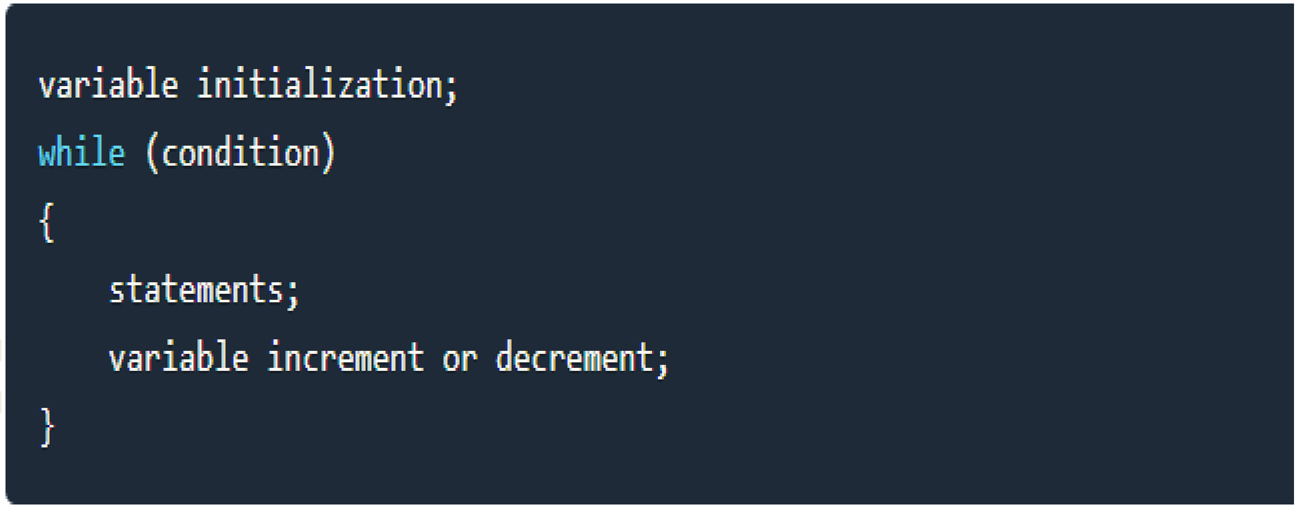
In for loop we have exactly two semicolons, one after initialization and second after condition. In this loop we can have more than one initialization or increment/decrement, separated using comma operator. for loop can have only one condition.

## Control Flow: While Loop

- Flow chart:



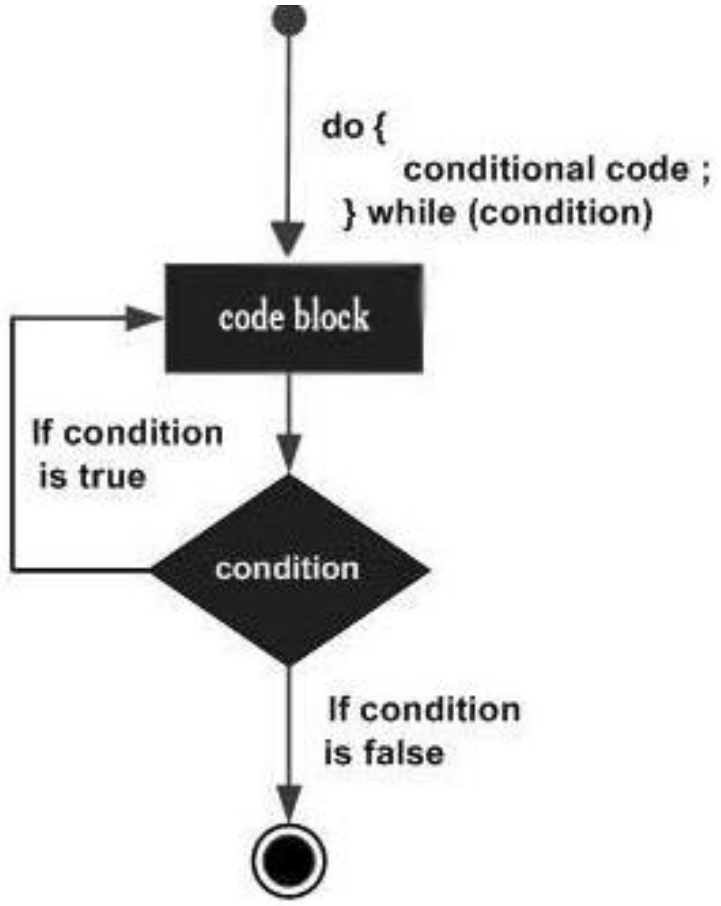
Syntax:



```
variable initialization;
while (condition)
{
    statements;
    variable increment or decrement;
}
```

## Control Flow: Do While Loop

- Flow chart:



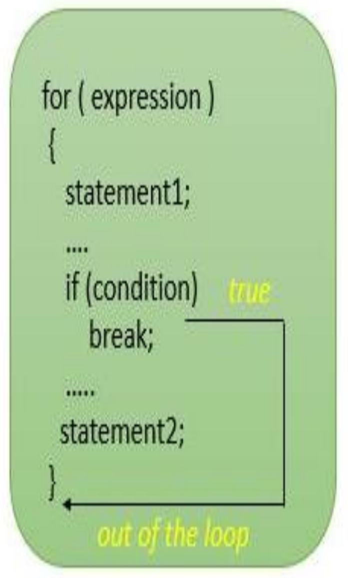
Syntax:

```
do
{
    // a couple of statements
}
while(condition);
```

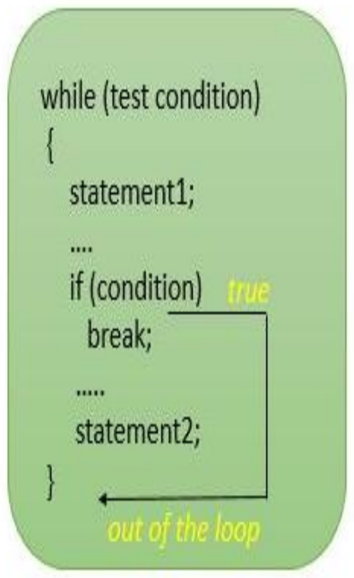
# Control Flow: Break, Continue

- Break:

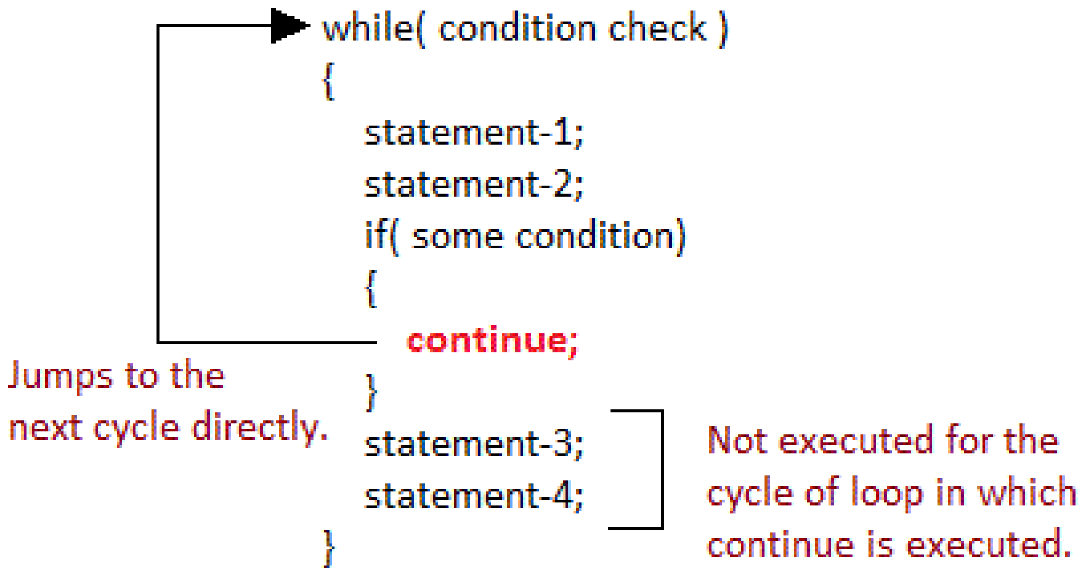
Continue:



```
for ( expression )
{
  statement1;
  ....
  if (condition)
    break;
  .....
  statement2;
}
```



```
while (test condition)
{
  statement1;
  ....
  if (condition)
    break;
  .....
  statement2;
}
```



```
while( condition check )
{
    statement-1;
    statement-2;
    if( some condition)
    {
      continue;
    }
    statement-3;
    statement-4;
}
```

Jumps to the next cycle directly.
Not executed for the cycle of loop in which continue is executed.