

Computer programming

Chapter 1

Basics of C++

- C++ is one of the world's most popular programming languages.
- can be found in today's operating systems, Graphical User Interfaces, and embedded systems.
- is an object-oriented programming language which gives a clear structure to programs and allows code to be reused, lowering development costs.

Cont'd

- is portable and can be used to develop applications that can be adapted to multiple platforms.
- is a cross-platform language that can be used to create high-performance applications.
- gives programmers a high level of control over system resources and memory.

Start C++

- A text editor, like Notepad, to write C++ code &
- A compiler, like GCC, to translate the C++ code into a language that the computer will understand

Structure of C++

#include <iostream> *□ a header file library that lets us work with input and output objects, Header files add functionality to C++ programs.*

using namespace std; *□ means that we can use names for objects and variables from the standard library.*

*□ can be omitted and replaced with the **std** keyword, followed by the **::** operator for some objects,*

*i.e. **std::cout << "Hello World!";** or **cout << std::endl<<"Hello world!";** or*

int main() *□ called a function, any code inside its curly brackets {} will be executed.*

{
 cout << "Hello World!"; *□ an object used together with the insertion operator (<<) to output values/print text. e.g "Hello World!";*

return 0; *□ ends the main function.*
}

Multiple cout

- You can add as many cout objects as you want. However, it does not insert a new line at the end of the output.
- you can use **\n character** or **endl manipulator** to insert a new line.

Example... \n character

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

```
int main() {
    cout << "Hello World! \n"; \\ \n\n" create a
    blank line
```

/* and */ used to insert multi-line comments

```
cout << "I am learning C++";
return 0;
}
```

Example ... endl manipulator

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

```
int main() {  
    cout << "Hello World!" << endl;  
    cout << "I am learning C++";  
    return 0;  
}
```

Variables

- Variables are containers for storing data values.
- provides us with ***named storage*** that our programs can manipulate.
- Each variable in C++ has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

Cont'd

- A variable definition tells the compiler where and how much storage to create for the variable , specifies a data type, and contains a list of one or more variables of that type:

type variable_list;

e.g. `int a, b, c, d;`

Cont'd

- Variables can be initialized (assigned an initial value) in their declaration.

type variable_name(identifier) = value;

e.g. int d = 3, f = 5;

- A variable declaration is useful when you are using multiple files and you define your variable in one of the files which will be available at the time of linking of the program.
- use **extern** keyword to declare a variable at any place. Though you can declare a variable multiple times in your C++ program, but it can be defined only once in a file, a function or a block of code.

Cont'd

```
#include <iostream>
using namespace std; // Variable declaration: extern int a, b;
    extern int c;
extern float f;
int main ()
{
// Variable definition:
int a, b;
int c;
    float f;
// actual initialization
a = 10;
b = 20;
c = a + b;
cout << c << endl ;
    f = 70.0/3.0;
cout << f << endl ;
    return 0; }
```

Cont'd

- names for variables (unique identifiers)
 - must begin with a letter or an underscore (`_`)
 - case sensitive
 - cannot contain whitespaces or special characters like `!`, `#`, `%`, etc.
- Reserved words (like C++ keywords, such as `int`, `float`, `char`...) cannot be used as names

Data types

- Data types define the type of data a variable can hold, for example an integer variable can hold integer data, a character type variable can hold character data etc.
- Data types in C++ are categorized in three groups: **Built-in**, **user-defined** and **Derived**.

Cont'd

Built in data types

Int,char,float,double,boolean,void,wide character...

Eg. Char ,for characters with size 1 byte.

Stores a single character/letter/number, or ASCII values

int , for integers with size 2bytes.

Stores whole numbers, without

Cont'd

float: Stores fractional numbers, containing one or more decimals. Sufficient for storing 7 decimal digits with size 4bytes

double: For double precision floating point. Stores fractional numbers, containing one or more decimals. Sufficient for storing 15 decimal digits with size 8 bytes.

e.g. double num = 10098.98899;

boolean: For booleans, true or false. Stores true or false values with size of 1byte

e.g. bool b = false;

Constants

- Constants refer to fixed values that the program may not alter and they are called **literals**.
- Constants can be of any of the basic data types and can be divided into Integer Numerals, Floating-Point Numerals, Characters, Strings and Boolean Values.
- use the **const** keyword (this will declare the variable as "constant")

e.g. **const** int numStud = 30 ; // numStud will always be 30

Basic input/output

- sequences of bytes flow from a device like a keyboard, a disk drive, or a network connection etc. to main memory is called **input operation**(cin >>)
- **Output operation** (cout <<) is bytes flow from main memory to a device like a display screen, a printer, a disk drive, or a network connection, etc.

Cont'd

e.g.

```
int x;
```

```
cout << "Type a value: "; // Type a  
value and press enter
```

```
cin >> x; // Get user input from the  
keyboard
```

```
cout << "Your number is: " << x; //  
Display the input value
```

Exercise

create simple calculator

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

int main() {
    int x, y, z;
    int sum;
    cout << "Type a number: ";
    cin >> x;
    cout << "Type second number: ";
    cin >> y;
    cout << "Type thired number: ";
    cin >> z;
    sum = x + y+z;
    cout << "Sum is: " << sum;
    return 0;
}
```

Operators

- operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations.
- C++ has built-in operators and provide the following types of operators –
- **Arithmetic Operators(%,++,--)** □ used to perform common mathematical operations.

Example :

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

```
int main() {
    int x = 5;
    int y = 2;
    cout << x % y;    \\moduls , it returns the division remainder
    return 0;
}
```

Cont'd

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

```
int main() {
    int x = 5;
    --x;    \\
    cout << x;
    return 0;
}
```

Arithmetic Operators

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
++	Incrementoperator , increases integer value by one	A++ will give 11
--	Decrementoperator. decreases integer value by one	A-- will give 9

Assignment Operators

(= , +=,-=, >>=, &=, ^=, |=...) □ used to assign values to variables.

Example :

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

```
int main() {
    int x = 5;
    x |= 3; \\Bitwise inclusive OR and assignment operator.
    cout << x;
    return 0;
}
```

Cont'd

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

```
int main() {
    int x = 5;
    x ^= 3;    \\Bitwise exclusive OR and
assignment    operator.
    cout << x;
    return 0;
}
```

Cont'd

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

```
int main() {
    int x = 5;
    x &= 3;    /*AND assignment operator , copies a bit to the
               result if it exists in both operands*/.

    cout << x;
    return 0;
}
```

cont'd

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

```
int main() {
    int x = 5;
    x >>= 3; /*Binary Right Shift Operator. The left
operands value is moved right by the number of bits
specified by the right operand.*/
    cout << x;
    return 0;
}
```

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

Cont'd

- **Relational Operators or comparison (==,!= ...)** used to compare two values. The return value of a comparison is either true (1) or false (0).

Example :

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

```
int main() {
    int x = 5;
    int y = 3;
    cout << (x != y); // returns 1 (true) because 5 is not equal to 3
    return 0;
}
```

Relational Operators or comparison

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

Logical Operators(&& logical and , || logical or ,! Logical not)□
used to determine the logic between variables or values

Example :#include <iostream>
using namespace std;

```
int main() {  
    int x = 5;  
    int y = 3;  
    cout << (x > 3 || x < 4); // returns true (1) because one of the  
        conditions are true (5 is greater than 3, but 5 is not less  
than 4)  
    return 0;  
}
```

Logical operators

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.

Excercise

```
#include <iostream>

using namespace std;
int main ()
{

string name,sex,age,address,email,phone;

cout<<"My Name is:- \n";
cin >> name;
    cout<<"\tSex:-\n";
    cin>> sex;
    cout<<"\tAge:- \n";
    cin >> age;

    cout<<"\tAddress :- \n";
    cin >> address;

    cout<<"\tE-mail:- \n";
    cin >>email;

    cout<<"\tPhone No.:- \n";
    cin >> phone;


    return 0;
}
```

Cont'd

```
#include <iostream>

using namespace std;
int main ()
{

string name,sex,age,address,email,phone;
cout<<"My Name is:-rrrrrrrrrrr\n";

    cout<<"\tSex:-F\n";

    cout<<"\tAge:- 21\n";


    cout<<"\tAddress :- A.A\n";


    cout<<"\tE-mail:-rrrrrrrr@gmail.com \n";

    cout<<"\tPhone No.:- +2519111111n";


    return 0;
}
```

Cont'd

```
#include <iostream>

using namespace std;
int main ()
{
    int a,c,l,w;
    cout<<"this will calculate the area and circumference\n";
    cout<<"please eneter the length value\n";
    cin>>l;
    cout<<"\n";
    cout<<"please enter the width value\n";
    cin>>w;
    cout<<"\n";
    a=l+w;
    c=2*(l+w);
    cout<<"the area is:"<<a<<"\n";
    cout<<"the circumfrance is:"<<c;
    return 0;
}
```

Cont'd

```
#include <iostream>

using namespace std;
int main ()
{

float x,y,sum,div,subt;
cout<<"calculate the sum,div,sub\n";
cout<<"Enter numbers to add\n";
cin>>x;
cin>>y;
sum=x+y;
div=x/y;
subt=x-y;
cout<<"Sum="<<sum<<endl;
cout<<"div="<<div<<endl;
cout<<"subt="<<subt<<endl;;

    return 0;
}
```

End of chapter one

Chapter 2

Outline

Control structures

Conditional structure

Iteration structure (loops)

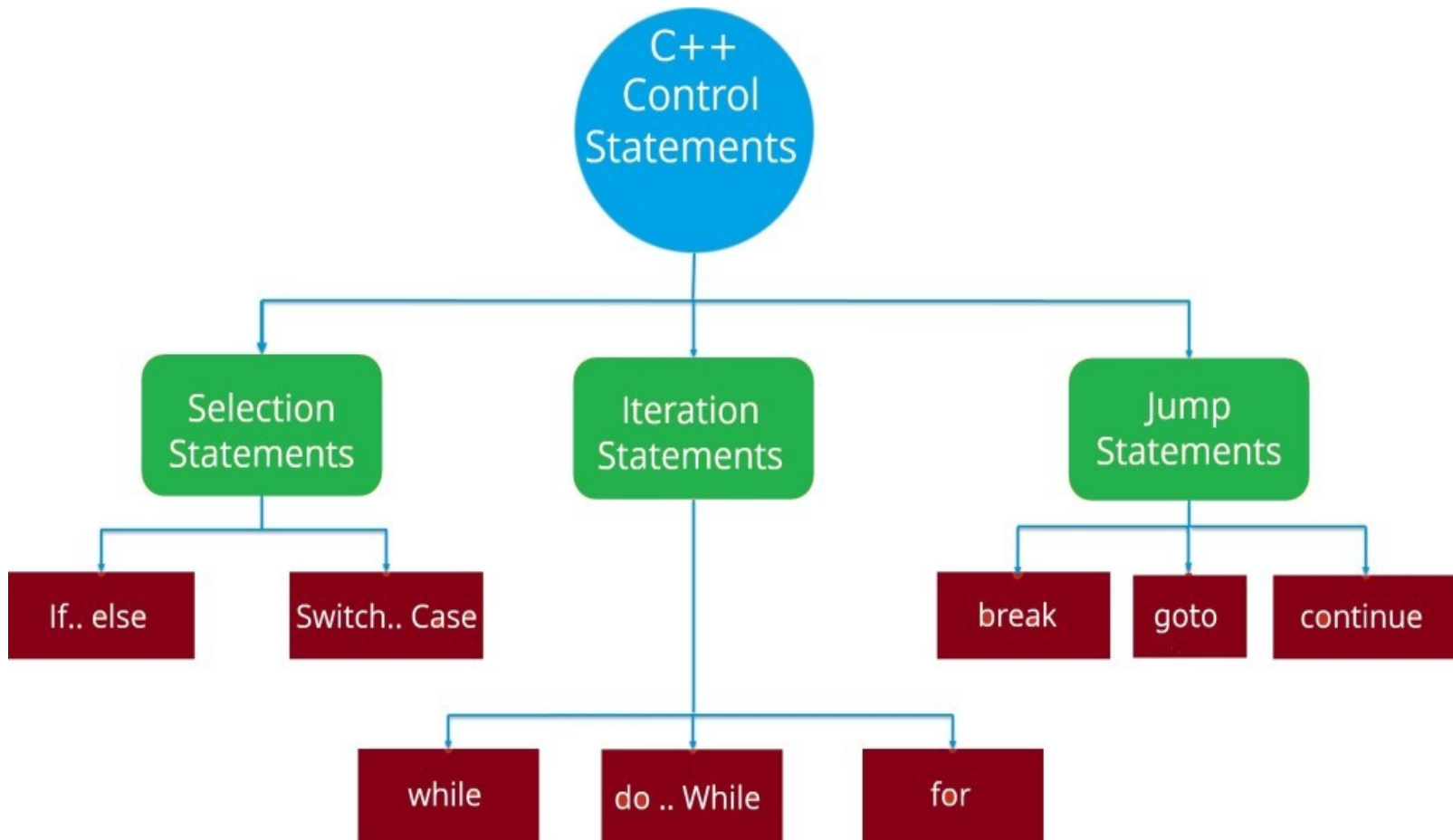
Jump statement

Selective structure

Control structures

- There may be a situation, when you need to execute a block of code several number of times. In general, statements are executed sequentially.
- Control structures allow for more complicated execution paths.

Cont'd



Iteration statements

- are used to execute the block of code repeatedly for a specified number of times or until it meets a specified condition.
- Iteration statements are commonly known as loops or looping statements.
- A loop statement allows us to execute a statement or group of statements multiple times.

While loop

- repeatedly executes a target statement or group of statements as long as a given condition is true.
- It tests the condition before executing the loop body.

Syntax

```
while(condition) // code block to be executed  
{  
statement(s); }
```

Cont'd

- **statement(s)** may be a single statement or a block of statements.
- The **condition** may be any expression, and is any non-zero value. The loop iterates while the condition is true.
- When the condition becomes false, program control passes to the line immediately following the loop.
- The loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Cont'd

Example

```
#include <iostream>
using namespace std;
int main ()
{
// Local variable declaration:
int b ;
b = 20;
// while loop execution
while( b < 50 )
{
cout << "value of b: " << b << endl;
b++; // the code in the loop will run, over and over again, as long as a variable (b) is
less than 50:
}
return 0;
}
```

For loop

- A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

```
for ( init; condition; increment )  
{ statement(s); }
```

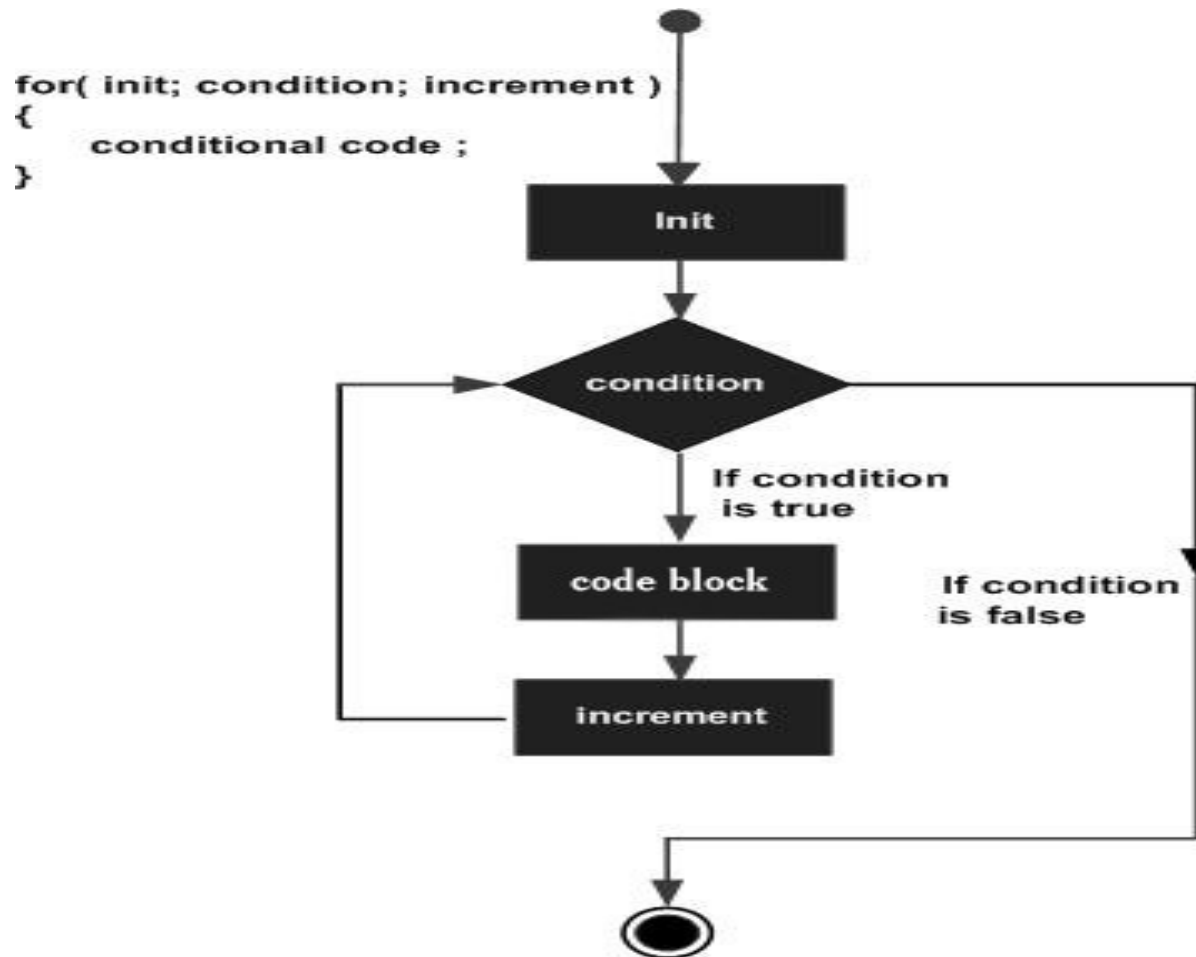
Cont'd

- ✓ The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables.
- ✓ **condition** is evaluated & If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.

Cont'd

- ✓ After the body of for loop executes, the flow of control jumps back up to **increment** statement.
- ✓ This statement can be left blank, as long as a semicolon appears after the condition.
- ✓ The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.

Flow diagram



Cont'd

Example:

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

```
int main() {
    for (int i = 0; i < 5; i++) {
        cout << i << "\n";
    }
    return 0;
}
```

do...while loop

- This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.
- Like a 'while' statement, except that it tests the condition at the end of the loop body.

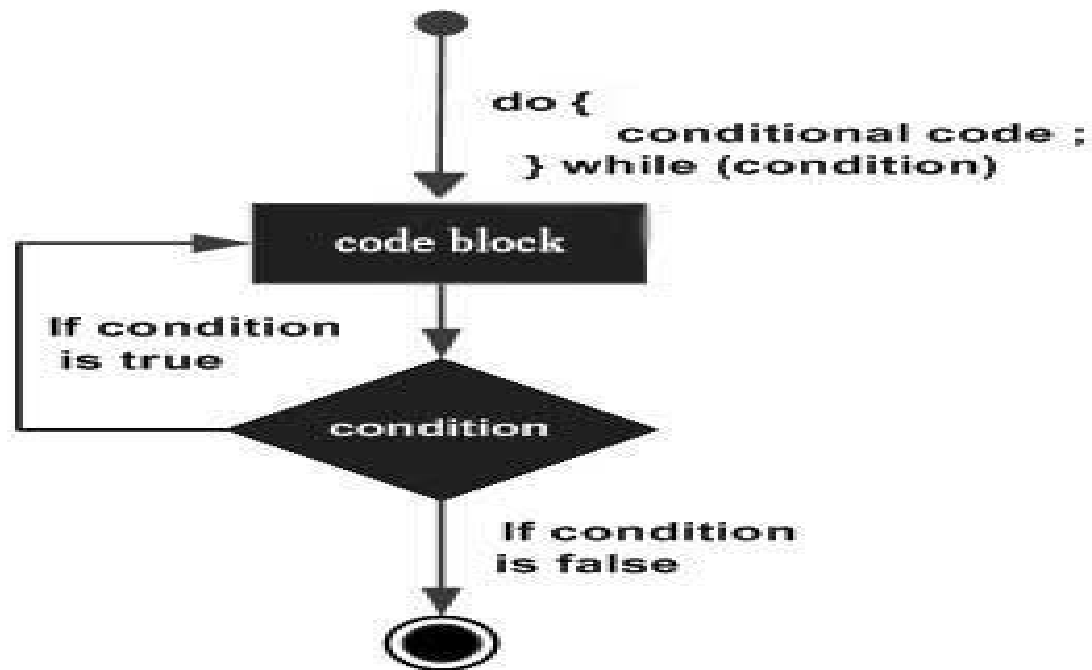
Syntax

```
do
{ statement(s); // code block to be executed
}
while( condition ); /* the condition appears at the
end of the loop, so the statement(s) in the loop execute once
before the condition is tested.*/
```

Cont'd

- If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop execute again.
- This process repeats until the given condition becomes false.

Cont'd



Cont'd

Example:

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

```
int main() {
    int i = 0;
    do {
        cout << i << "\n";
        i++;
    }
    while (i < 5);
    return 0;
}
```

Nested loops

- You can use one or more loop inside any another 'while', 'for' or 'do..while' loop.

Syntax

- The syntax for a **nested for loop** statement
for (init; condition; increment)
{ for (init; condition; increment)
{ statement(s); }
statement(s); // you can put more
statements. }

Cont'd

Example

```
#include <iostream>
using namespace std;
int main () {
    int i, j, l;
    for(i = 2; i<100; i++)
    { for(j = 2; j <= (i/j); j++) if(!(i%j))
    for(l=6;l>(i*j);l++) if(!(l%i))
    break; // if factor found, not prime
    if(j > (i/j)) cout << i << " is prime\n"; }
    return 0; }
```

Cont'd

Syntax

- The syntax for a **nested while loop** statement :

```
while(condition)
```

```
{
```

```
while(condition)
```

```
{
```

```
statement(s); }
```

```
statement(s); // you can put more  
statements. }
```

Cont'd

Example

```
#include <iostream>
using namespace std;
int main ()
{
// Local variable declaration:
int b ;
b = 20;
// while loop execution
while( b < 50 )
{
while (b>50)
cout << "value of b: " << b << endl;
cout <<"value of b:"<<b<<"\n";
b++; // the code in the loop will run, over and over again, as long as a variable (b) is less than
50:
}
return 0;
}
```

Cont'd

Syntax

The syntax for a **nested do...while loop** statement :

```
do { statement(s); // you can put more
statements.
```

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

Cont'd

Example

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

```
int main() {
    int i = 0,j=6;
    do {
        cout << i << "\n";
        i++;
    } while (i < 5);
    cout << j << endl;
    j--;
    while(j<12);
    return 0;
}
```

Decision making statements. (Selection statements)

- allow us to control the flow of the program during run time on the basis of the outcome of an expression or state of a variable.
- evaluates single or multiple test expressions which results in **“TRUE”** or **“FALSE”**.
- The outcome of the test expression/condition helps to determine which block of statement(s) to executed if the condition is **“TRUE”** or **“FALSE”** otherwise.

If statements

- If statements in C++ is used to control the program flow based on some condition, it's used to execute some statement code block if the expression is evaluated to true. Otherwise, it will get skipped.
- it can be used in various forms depending on the situation and complexity.

Cont'd

- consists of a boolean expression followed by one or more statements.

Syntax

```
if(boolean_expression)
{
    statement(s) //it will execute if the
boolean expression is true
}
```

Cont'd

- If the boolean expression evaluates to **true**, then the block of code inside the if statement will be executed.
- If boolean expression evaluates to **false**, then the first set of code after the end of the if statement (after the closing curly brace) will be executed.

Cont'd

Example :

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

```
int main() {
    if (20 > 18) {
        cout << "20 is greater than 18";
    }

    return 0;
}
```

If...else statement

- An **if** statement can be followed by an optional **else** statement, which executes when the boolean expression is false.

Syntax

```
if(boolean_expression) {  
    // statement(s) will execute if the boolean expression is true  
}  
else  
{  
    // statement(s) will execute if the boolean expression is false  
}
```

Cont'd

- If the boolean expression evaluates to **true**, then the **if block** of code will be executed, otherwise **else block** of code will be executed.

Example :

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

```
int main () {
    // local variable declaration:
    int c = 60;

    // check the boolean condition
    if( c > 10 ) {
        // if condition is true then print the following
        cout << "c is greter than 10;" << endl;
    } else {
        // if condition is false then print the following
        cout << "c is not greter than 10;" << endl;
    }
    cout << "value of c is : " << c << endl;

    return 0;
}
```

if...else if...else Statement

- if-else-if statement is used when we need to check multiple conditions.
- In this selection statement we have only one “if” and one “else”, however we can have multiple “else if” blocks.
- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.

Cont'd

Syntax

```
if(boolean_expression 1) { // Executes  
when the boolean expression 1 is true }  
else if( boolean_expression 2) { // Executes  
when the boolean expression 2 is true }  
else if( boolean_expression 3) { // Executes  
when the boolean expression 3 is true }  
else { // executes when the none of the  
above condition is true. }
```

Cont'd

```
#include <iostream>
using namespace std;
int main () {
int a = 100; // local variable declaration:
if( a == 10 ) // check the boolean condition I
{
cout << "Value of a is 10" << endl; } // if condition is true then print the following
else if( a == 20 ) { // if else if condition is true
cout << "Value of a is 20" << endl; }
else if( a == 30 ) { // if else if condition is true
cout << "Value of a is 30" << endl; }
else { // if none of the conditions is true
cout << "Value of a is not matching" << endl;
}
cout << "Exact value of a is : " << a << endl;
return 0;
}
```

Switch statement

- **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax

```
switch(expression)
{
case constant-expression :
statement(s);
break; //optional case
constant-expression :
statement(s);
break; //optional
// you can have any number of case statements.
default : //Optional
statement(s); }
```

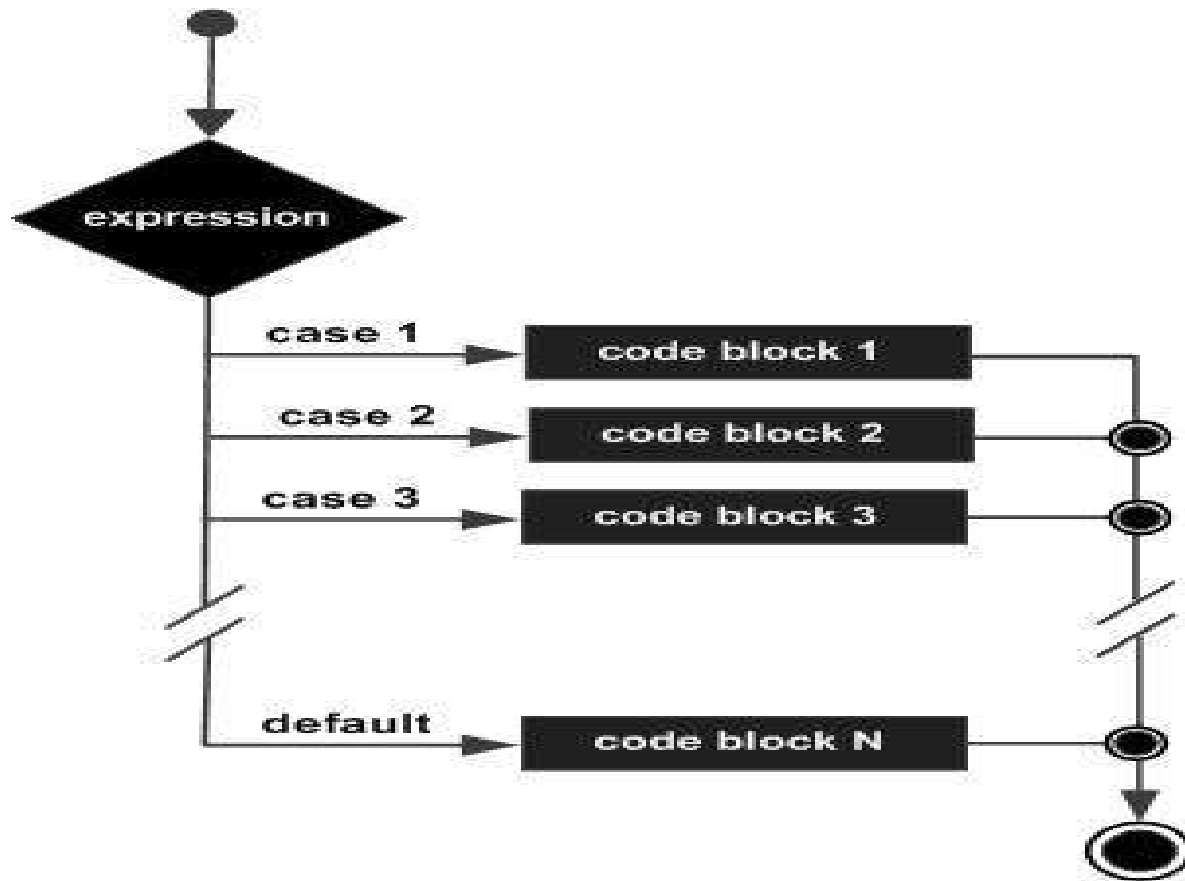
switch statement rules

- The **expression** used in a **switch** statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement is reached.

Cont'd

- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will *fall through* to subsequent cases until a break is reached.
- A **switch** statement can have an optional **default** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Cont'd



Example

```
#include <iostream>
using namespace std;
int main () { // local variable declaration:
char grade = 'D';
switch(grade) {
case 'A' :
    cout << "Excellent!" << endl;
    break;
case 'B' :
    cout << "very good!" << endl;
    break;
case 'C' :
    cout << "Well done" << endl;
    break;
case 'D':
    cout << "You passed" << endl;
    break;
case 'F' :
    cout << "Better try again" << endl;
    break;
default :
    cout << "Invalid grade" << endl; }
cout << "Your grade is " << grade << endl;
return 0;
}
```

nested switch statements

- It is possible to have a switch as part of the statement sequence of an outer switch. Even if the case constants of the inner and outer switch contain common values, no conflicts will arise.

Cont'd

- Syntax

```
switch(ch1) {  
  case 'A':  
    cout << "This A is part of outer switch"; switch(ch2) {  
  case 'A':  
    cout << "This A is part of inner switch";  
    break;  
  case 'B': // ...  
  }  
  break;  
  case 'B': // ...  
  }
```

Cont'd

Example

```
#include <iostream>
using namespace std;
int main ()
{ // local variable declaration:
  int a = 100 , b = 200, c = 12;
  switch(a)
  {
  case 100:
    cout << "This is part of outer switch" << endl;
    switch(b)
    {
      case 200:
        cout << "This is part of inner switch" << endl;
        switch (c)
        {
          case 12:
            cout<<"this is either part of outer switch or inner switch"<<"\n";
          }
        }
      }
    }
  cout << "Exact value of a is : " << a << endl;
  cout << "Exact value of b is : " << b << endl;
  cout <<"Exact values of b is : "<< c << "\n";
  return 0; }
```

Functions

- Function is block of code that together perform a task.
- Functions "Encapsulate" a task (they combine many instructions into a single line of code). Most programming languages provide many built in functions that would otherwise require many steps to accomplish.

Cont'd

- When a function is "called" the program "leaves" the current section of code and begins to execute the first line inside the function. Thus the function "flow of control" is:
- The program comes to a line of code containing a "function call".
- The program enters the function (starts at the first line in the function code).
- **All instructions** inside of the function are executed from top to bottom.
- The program leaves the function **and goes back to where it started from.**
- Any data computed and **RETURNED** by the function is used in place of the function in the original line of code.

Steps to Writing a Function

- Understand the purpose of the function.
- Define the data that comes into the function from the caller (in the form of parameters)!
- Define what data variables are needed inside the function to accomplish its goal.
- Decide on the set of steps that the program will use to accomplish this goal. (The Algorithm)

Defining a Function

Syntax

```
return_type function_name( parameter list )  
{  
body of the function  
}
```

- The **return_type** is the data type of the value the function returns(function may return a value)
- Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.

Cont'd

- **Function Name** – This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters** – A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body** – The function body contains a collection of statements that define what the function does.

Function Declarations

- tells the compiler about a function name and how to call the function.

Syntax

```
void myFunction()  
{  
    // code to be executed  
}
```

Cont'd

- void means that the function does not have a return value.
- myFunction() is the name of the function
- inside the function (the body), add code that defines what the function should do

Cont'd

- To use a function, you will have to call or invoke that function.
- When a program calls a function, program control is transferred to the called function.
- A called function performs defined task and when it's return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

Cont'd

Example

```
void myFunction()          // Create a function
{
    cout << "create function!";
}
```

```
int main() {
    myFunction(); // call the function
    return 0;
}
```

Cont'd

```
void myFunction() {  
    cout << "A function can be called multiple  
times:"!\n";  
}
```

```
int main() {  
    myFunction();  
    myFunction();  
    myFunction();  
    return 0;  
}
```

Parameters and Arguments

- Information can be passed to functions as a parameter. Parameters act as variables inside the function.
- Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma:

Cont'd

Syntax

```
void functionName(parameter1, parameter2, parameter3)  
{  
    // code to be executed  
}
```

Cont'd

Example

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

void myFunction(string fname) {    // fname is parametre
    cout << fname <<"Redet \n";
}

int main() {
    myFunction("Firafis");    // firafis , Hana &Fenet are arguments
    myFunction("Hana");
    myFunction("Fenet");
    return 0;
}
```

Function arguments

- An argument is a way for you to provide more information to a function.
- The function can then use that information as it runs, like a variable. Said differently, when you create a function, you can pass in data in the form of an argument, also called a parameter.

Cont'd

- Arguments are variables used only in that specific function.
- You specify the value of an argument when you call the function.
- Function arguments allow your programs to utilize more information.
- While calling a function, there are two ways that arguments can be passed to a function –

Default parameter

- If we call the function without an argument, it uses the default value
- by using the equals sign (=), you can use a default parameter value.
- A parameter with a default value, is often known as an "**optional parameter**".

Cont'd

Example

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

void myFunction(string country ="Ethiopia") {
    cout << country << "\n";
}

int main() {
    myFunction("Ziway");
    myFunction("Robe");
    myFunction();
    myFunction("Sheger");
    return 0;
}
```

Multiple Parameters

- you can add many parameters inside the function

Example :

```
#include <iostream>
```

```
#include <string>
```

```
using namespace std;
```

```
void myFunction(string fname, int age) {  
    cout << fname << " Bereket. " << age << " years old. \n";  
}
```

```
int main() {  
    myFunction("hana", 3);  
    myFunction("Beti", 14);  
    myFunction("Sara", 30);  
    return 0;  
}
```

Call by value

- The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function.
- changes made to the parameter inside the function have no effect on the argument.
- By default, C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function.

Cont'd

Example

```
#include <iostream>
using namespace std;
void swap(int a, int b);
int main () { // local variable declaration:
int a = 100;
int b = 200;
cout << "Before swap, value of a :" << a << endl;
cout << "Before swap, value of b :" << b << endl; // calling a
function to swap the values.
swap(a, b);
cout << "After swap, value of a :" << a << endl;
cout << "After swap, value of b :" << b << endl;
return 0; }
```

call by reference

- The **call by reference** method of passing arguments to a function 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 passed argument.

Cont'd

- To pass the value by reference, argument reference is passed to the functions just like any other value.

Cont'd

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

void swapNums(int &x, int &y) {
    int z = x;
    x = y;
    y = z;
}

int main() {
    int firstNum = 10;
    int secondNum = 20;

    cout << "Before swap: " << "\n";
    cout << firstNum << secondNum << "\n";

    swapNums(firstNum, secondNum);

    cout << "After swap: " << "\n";
    cout << firstNum << secondNum << "\n";

    return 0;
}
```