Lecture I: Computer Languages

Mohammed Q. Ali

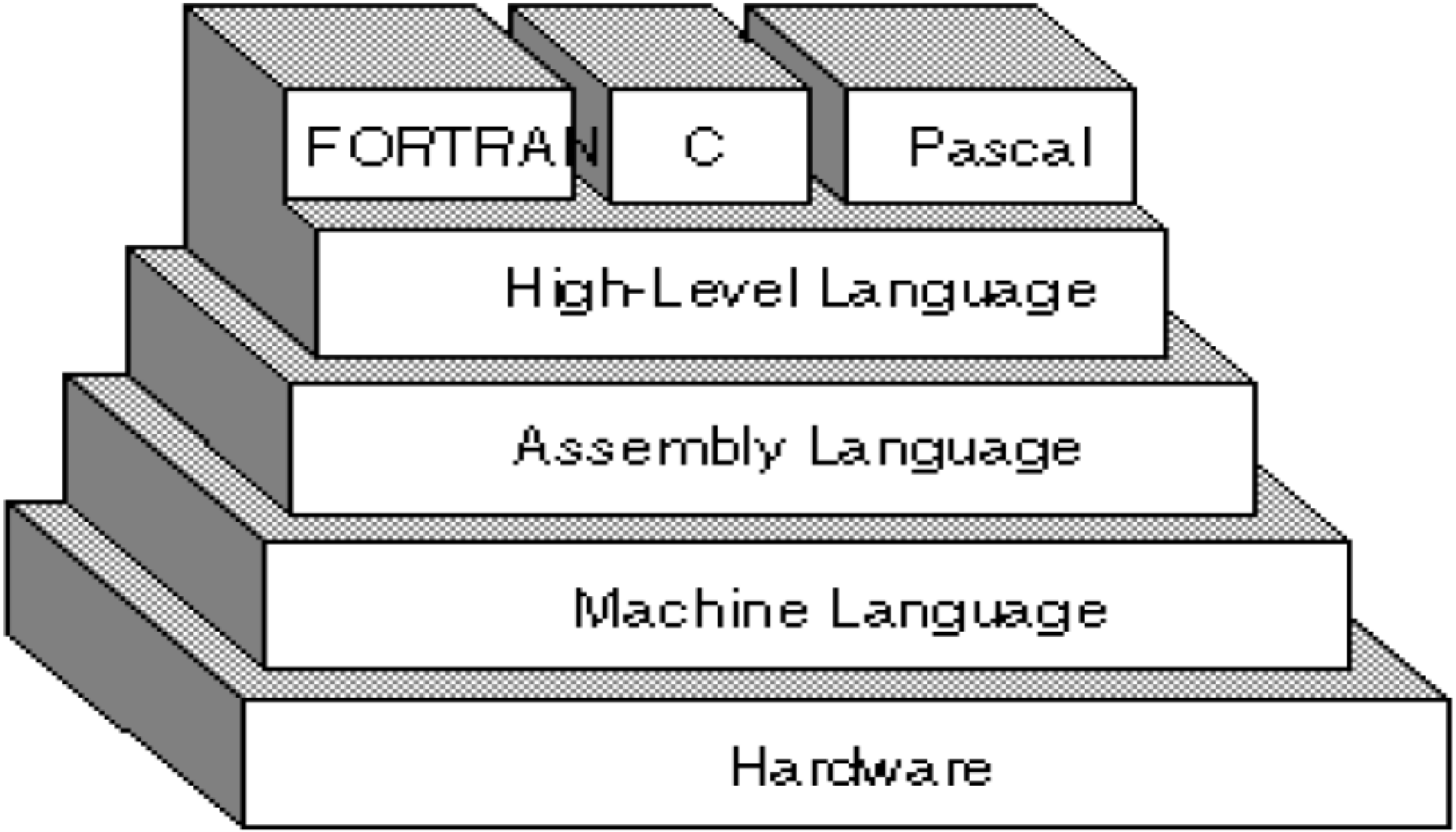
Computer Language

- ✓ A **computer language** is an **artificial language** designed to work for different applications in different environments.
- ✓ **Programming languages** can be used to create programs that control the behavior of a computer and serve any purpose.
- ✓ The term programming language usually refers to **high-level** languages such as **BASIC, C, C++, COBOL, FORTRAN, Pascal, Java, Oracle, ASP.NET, VB.NET and C#.**
- ✓ Each language has a **unique** set of keywords (words that it understands)

Computer Language Classifications

Human languages are known as **natural languages**. Unfortunately, computers can not *understand* natural languages, as a result we must *communicate* with computers using computer languages. These languages are:

- ✓ **High Level Languages**
- ✓ **Low Level Languages**
- ✓ **Assembly Language**
- ✓ **Machine Language**


FORTRAN
C
Pascal
High-Level Language
Assembly Language
Machine Language
Hardware

High Level Languages

It is a set of words and symbol which a programmer uses to write a program

- ✓ **High-level languages** are much closer to human language. A programming language such as **C, FORTRAN or Pascal** that enables to write programs which is understandable to programmer (Human) and can perform any sort of task, such languages are considered high-level because they are closer to human languages.

- ✓ High level language must use **interpreter**, **compiler** or **translator** to convert human understandable program to computer *readable* code (**machine code**).

There are **many** high level languages Some Examples:

- **COBOL** Business applications
- **FORTRAN** Engineering & Scientific Applications
- **PASCAL** General use and as a teaching tool
- **C & C++** General Purpose – currently most popular.
- **PROLOG** Artificial Intelligence
- **JAVA** General all purpose programming
- **.NET** General or web applications.

Advantages & Disadvantage of High Level Language

1) They are **near** to **English language**, that is they are easier to read, write and maintain.
2) High-level languages make **complex** programming **simpler**.
3) High level languages is **portable**, i.e., they can work on *different* **operating system**.
4) **Length** of the program is also small compared with **low level** language.
5) Many **real time** problems can be **easily** solved with **high level** language.
6) **But** They need to be **translated** for the computer to **understand**, hence **work slower than machine code.**

Low Level Languages

1) A computer low level language that **deals** with hardware registers by name is known as **assembly language**.
2) **Assembly language** is the best example of low level language, it is in between **machine** language and **high level** language.
3) A low-level language does not need a **compiler** or **interpreter** to run the program, the processor run low-level code directly.

Assembly Language

1) **Assembly languages** have the same structure and set of commands as **machine** languages, but they enable a programmer to use **names** instead of **numbers**.
2) In the early days of programming all programs were written in **assembly** language but now almost programs are written in a **high level** language.
3) Each **assembly** language is specific to a **particular** computer **architecture**, in contrast to most high level programming languages, which are generally portable across **multiple** systems.

Assembly Languages Usage

- Writing **games** or **simulation** program
- Writing **operating systems**
- Programming **robots**
- Computers **viruses**, certain **device drivers** or other items very close to the hardware or low-level operating system.

Machine Language

- The **native language** of the computer
- The **set** of **symbolic** instructions in **binary** that is used to represent operations and data in a machine called **machine code**
- **Machine code** executed directly by a computer's CPU. The lowest-level programming language that only be understood by computers.
- **Machine** language is a **collection** of **binary digits** or **bits** that the computer reads and interprets.
- Computer language that is directly executable by a computer **without** the need for translation by a **compiler** or an **assembler**.
- **Machine** language is almost impossible for humans to use because they consist **entirely of numbers**.

Program Code Converters

There are **four models** of execution of program by processor in high-level languages:

- ✓ **INTERPRETER**
- ✓ **COMPILER**
- ✓ **ASSEMBLER**
- ✓ **TRANSLATOR**

Interpreter

- An **interpreter** is a program that executes programming **code directly**.
- **Interpreter** can convert a source code , usually on a **step-by-step**, **line-by-line** and **unit-by-uni**t basis into **machine** code.
- Interpreted languages are **read** and then **executed directly**, with no **compilation** stage.

Compiler

Compiler is a program that compile source code into executable instructions that a computer can understand, it compiles code and produce in **EXE** format before it run.

Assembler

- **Assembler** normally **converts** assembly language's source code into machine language.
- **Assembly** language is a family of **low-level** languages for programming computers. It **implements** a **symbolic** representation of code.
- An **assembler** is a program that takes basic computer instructions and converts them into a **pattern** of **bits (0 & 1)** that the computer runs to produce result.

Translator

- ✓ **Translator** is a computer program that **translates** one programming language instruction(s) into another programming language instruction(s) …
- ✓ A computer **source code** is to be **translated** into a **low-level** programming language where **native** code **compilers** change it in **machine** language for execution.
- ✓ The **source code** is the code written either in **assembly** or **high level** language.

Lecture II

Planning the Computer Programs



Mohammed Q. Ali

Purpose of Program Planning

- To **write** a correct **program**, a programmer must write each and every instruction in **the correct sequence**

- **Logic** (instruction sequence) of a program can be **very complex**

- Hence, **programs** must be **planned** before they are **written** to ensure program instructions are:

  - ✓ Appropriate for the problem
  - ✓ In the correct sequence

Algorithm

- Refers to the **logic** of a program and a **step-by-step description** of how to arrive at the solution of a given problem
- In order to qualify as an **algorithm**, a sequence of instructions must have following characteristics:
  - ✓ Each and every instruction should be **precise and unambiguous**
  - ✓ Each instruction should be such that it can be performed in a finite time
  - ✓ One or more instructions should not be **repeated infinitely**. This ensures that the algorithm will ultimately terminate
  - ✓ After performing the instructions, that is after the algorithm terminates, the desired results must be **obtained**

Algorithm Examples

- ✓ **Read** a number and find its **square** out.

Algorithm:

1. **Start**.
2. **Input**: Read the number n from the user.
3. **Process**: Calculate the square of the number by multiplying n with itself **(square = n * n**).
4. **Output**: Display the result (the square of the number).
5. **End**.

Algorithm Examples

✓ Read a number, **check** if it is **odd** or **even**.

Algorithm:

1. **Start**.
2. **Input**: Read the number n from the user.
3. **Check**:
   - If **(n % 2 == 0)**, then the number is **even**.
   - Otherwise, the number is **odd**.
4. **Output**: Display whether the number is **odd** or **even**.
5. **End**.

Algorithm Examples

✓ Read a number, **check** if it is **odd** or **even**.

Algorithm:

1. **Start.**
2. **Initialize:** Set **A** = 0**.**
3. **Loop:** Iterate through numbers from 1 to 5:
   - Add each number to **A.**
4. **Output:** Display the value of **A.**
5. **End.**

Presentation of Algorithm

- ❑ **As programs**
- ❑ **As flowcharts**
- ❑ **As pseudocodes**

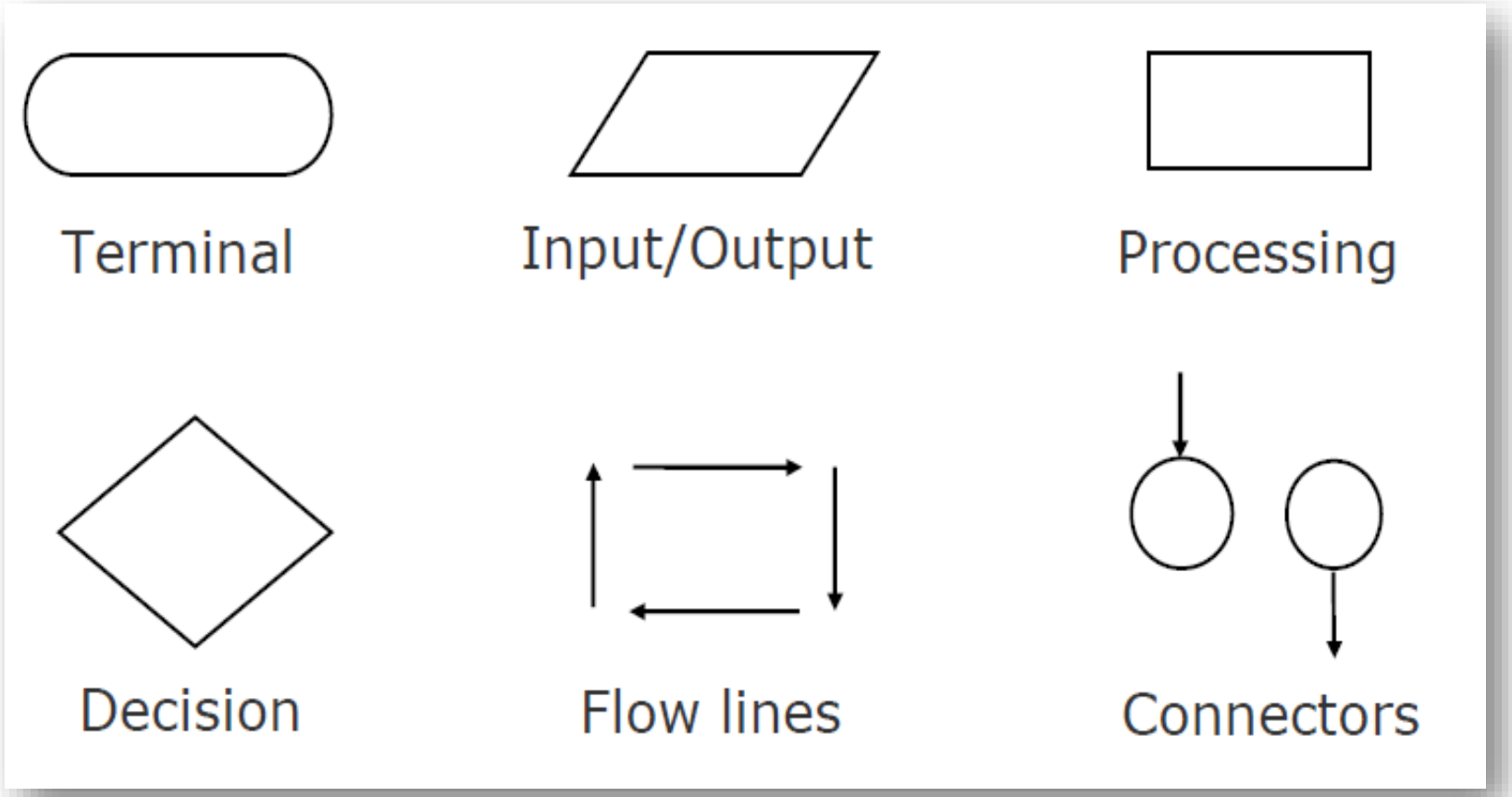
- ✓ When an **algorithm** is represented in the form of a programming language, it becomes a **program**

- ✓ Thus, any **program** is an algorithm, but any **algorithm** is **not** a program

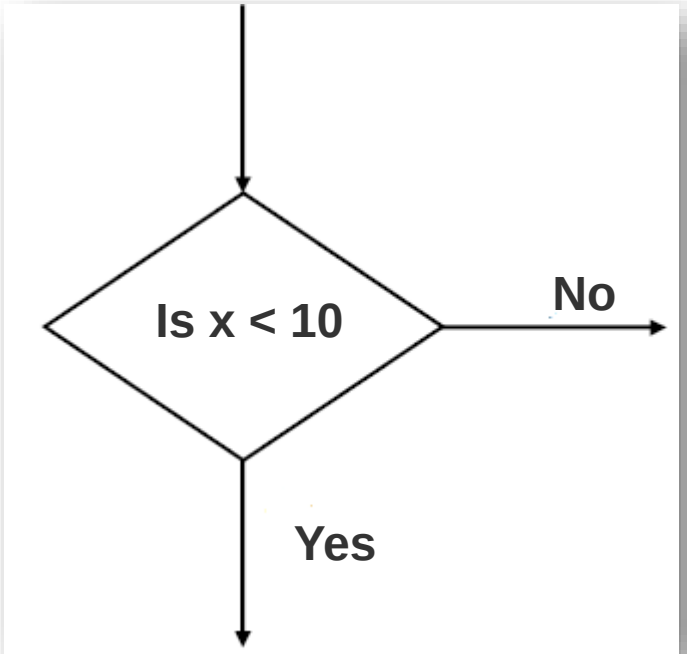
Flowchart

- **Flowchart** is a pictorial representation of an **algorithm**
- Uses **symbols** (boxes of different shapes) that have standardized meanings to denote different types of instructions
- **Actual** instructions are written within the **boxes**
- **Boxes** are connected by solid lines having arrow marks to indicate the exact sequence in which the instructions are to be executed
- Process of drawing a flowchart for an algorithm is called **flowcharting**

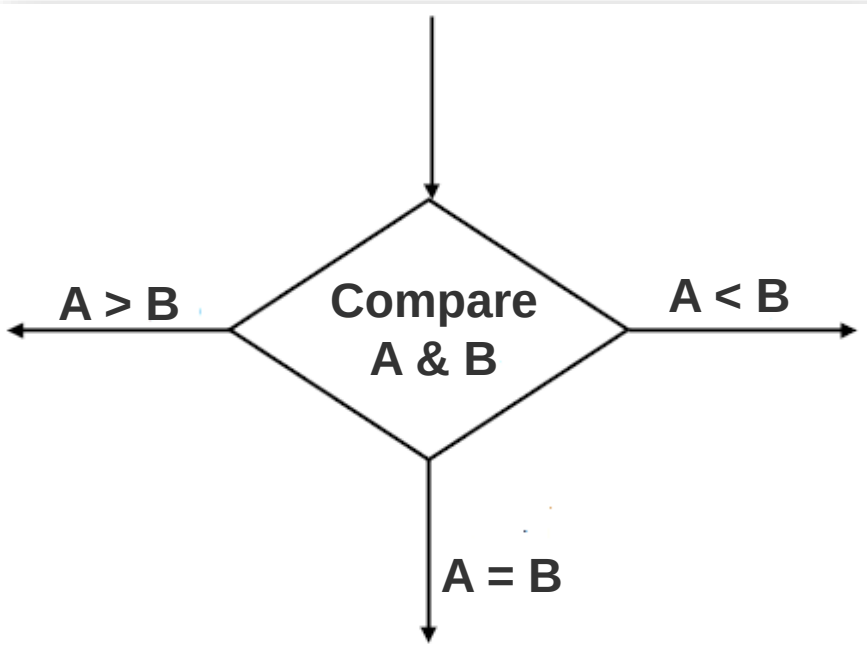
Basic Flowchart Symbols



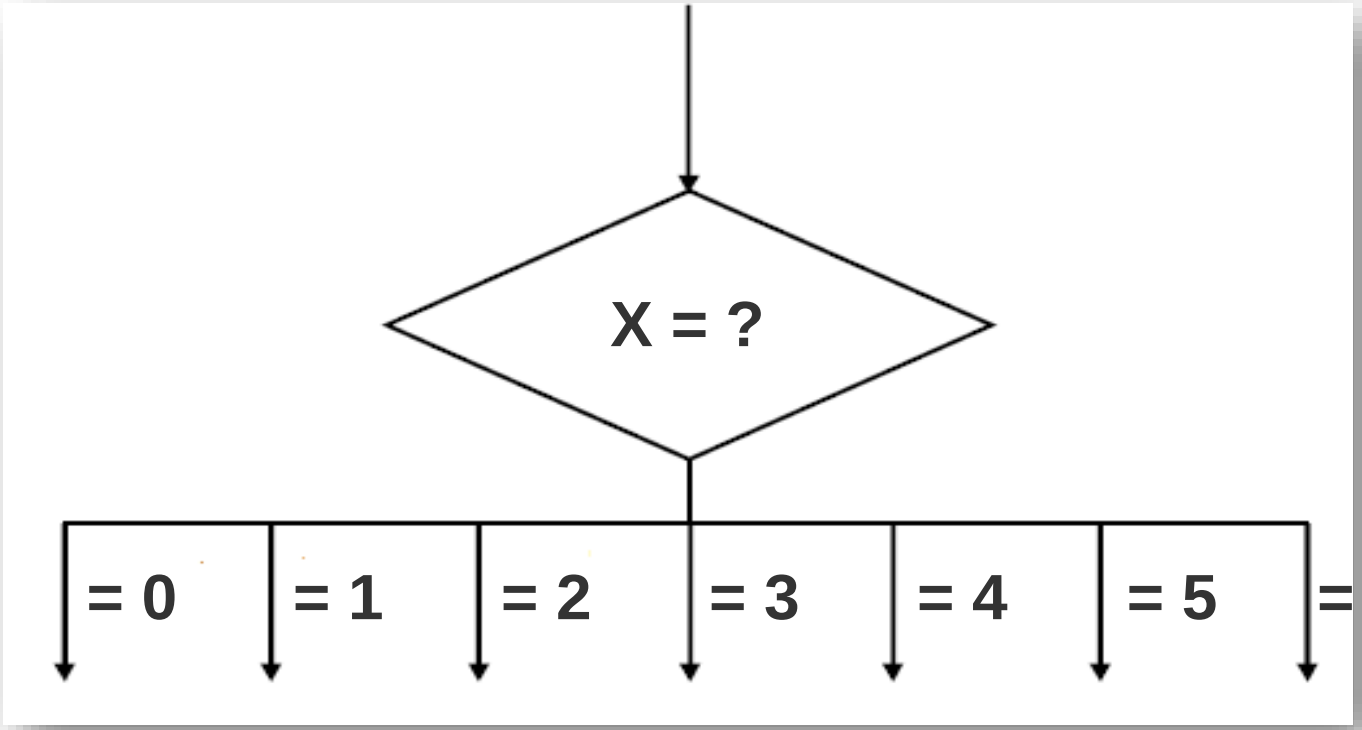
Division Symbol Examples



a) A **two-way** branch decision



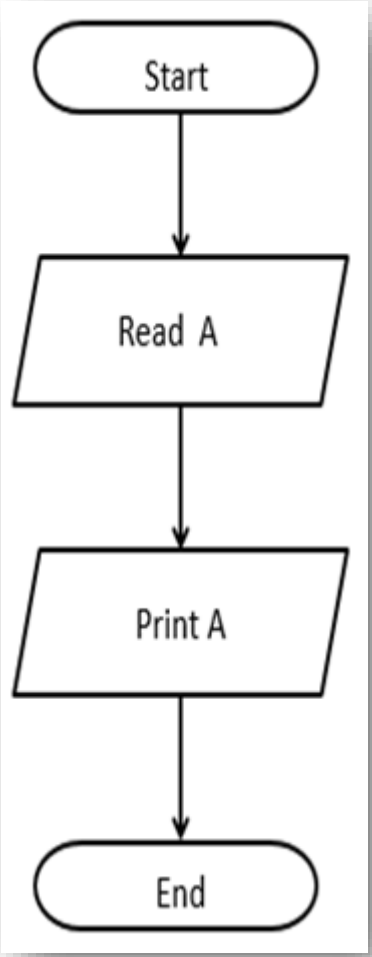
b) A **three-way** branch decision



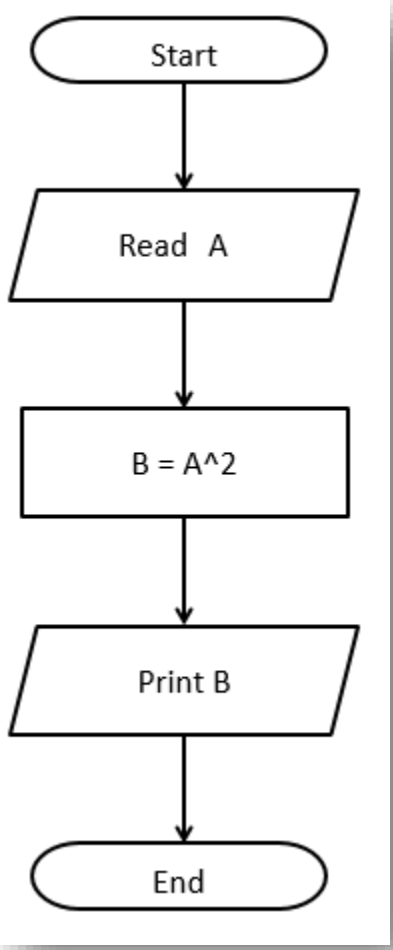
c) A **multiple-way** branch decision.

Flowchart Examples

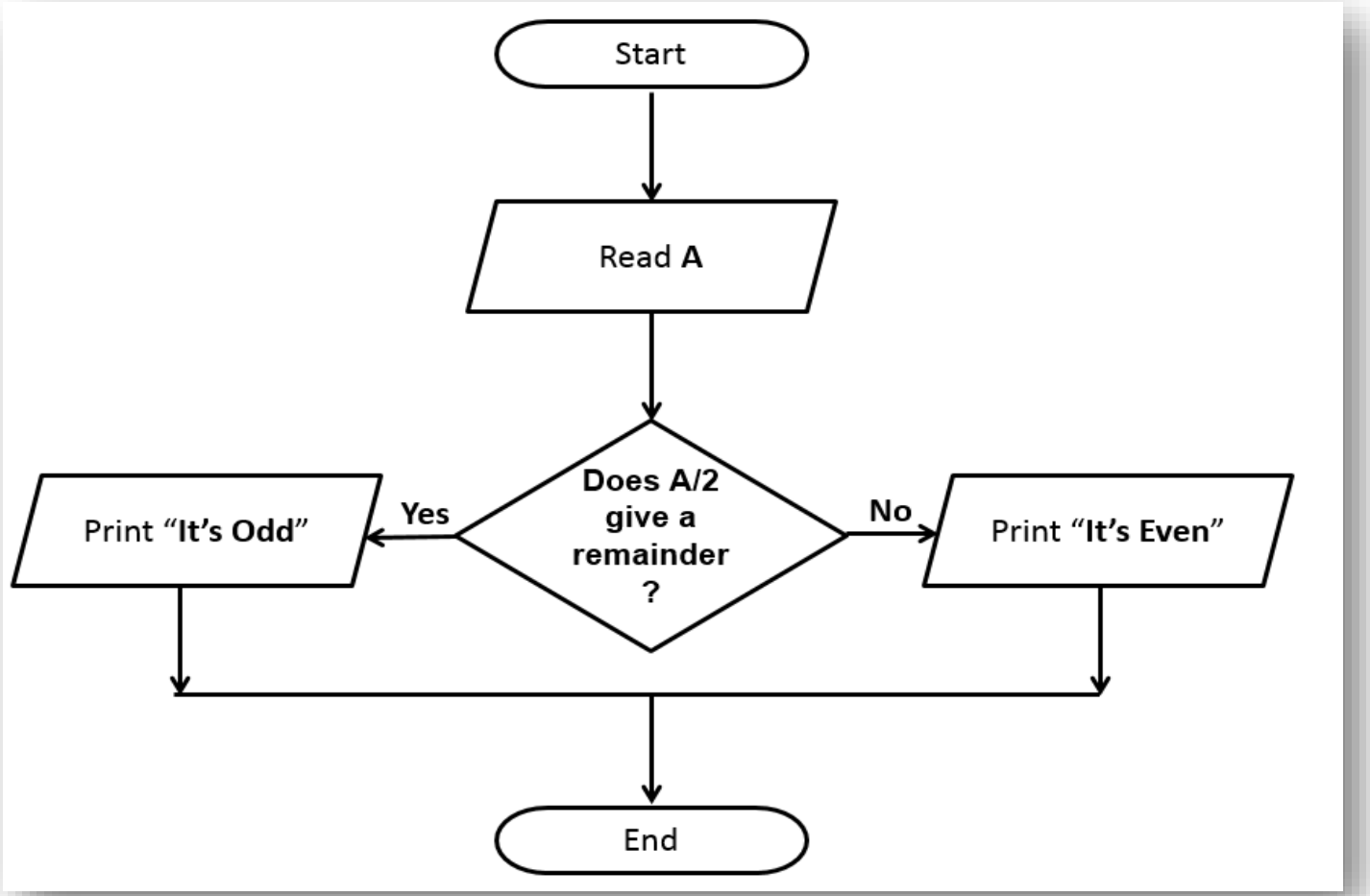
- ✓ **Read** a number and print it out.



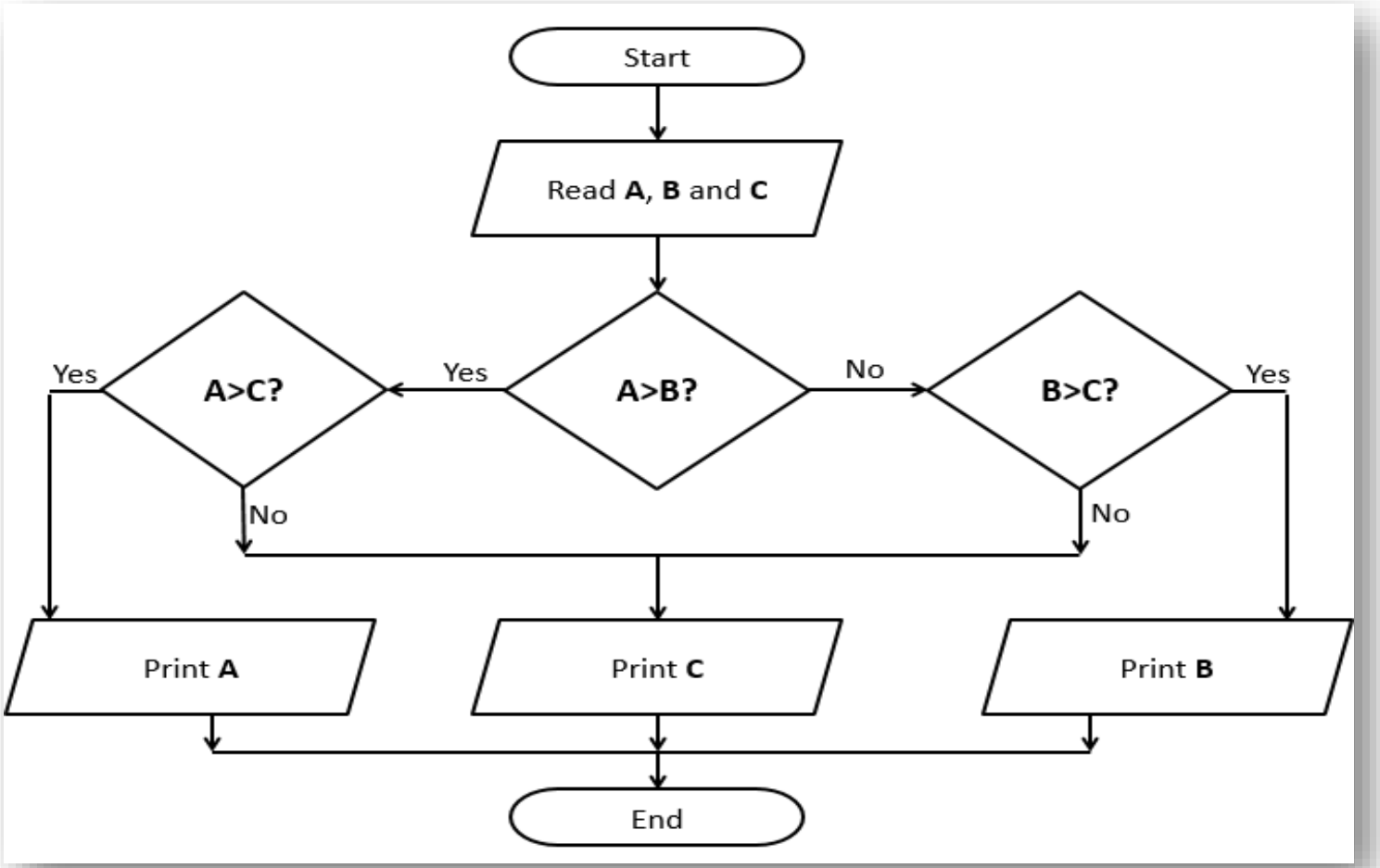
- ✓ **Read** a number and find its **square** out.



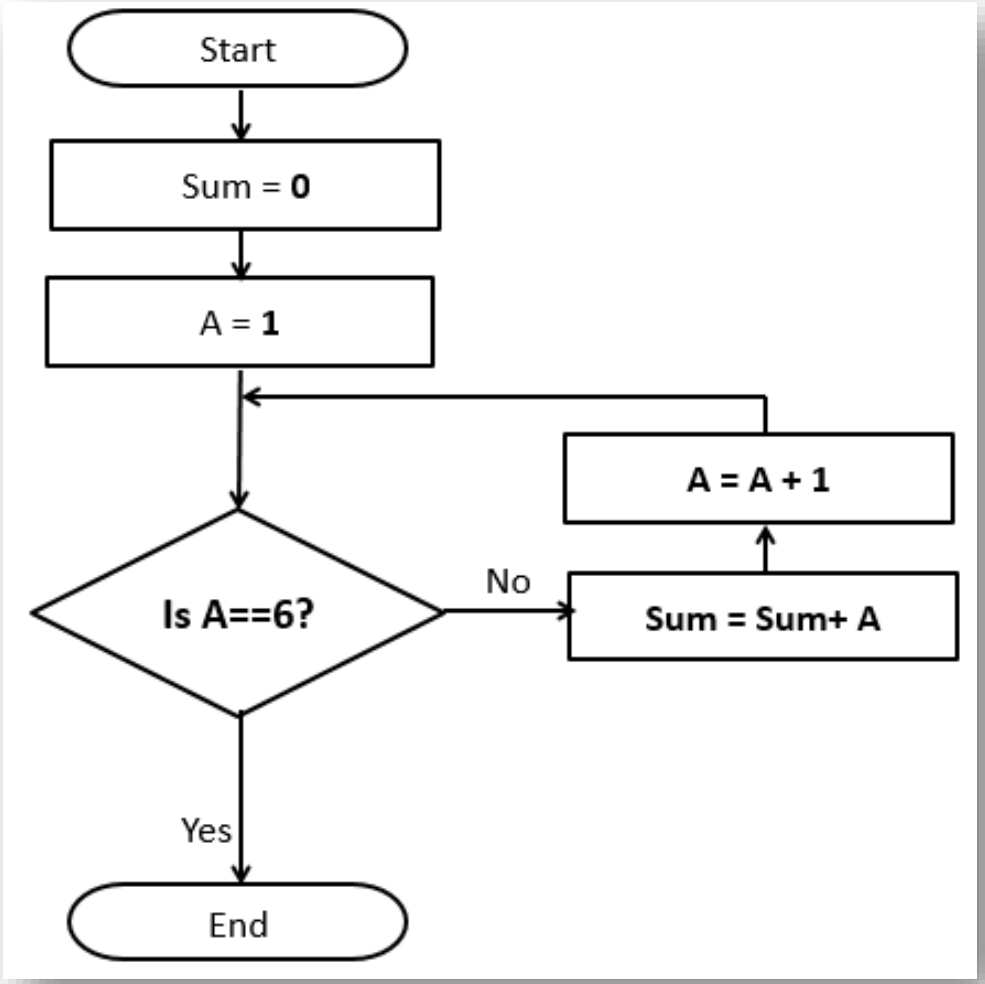
✓ Read a number, **check** if it is **odd** or **even**.



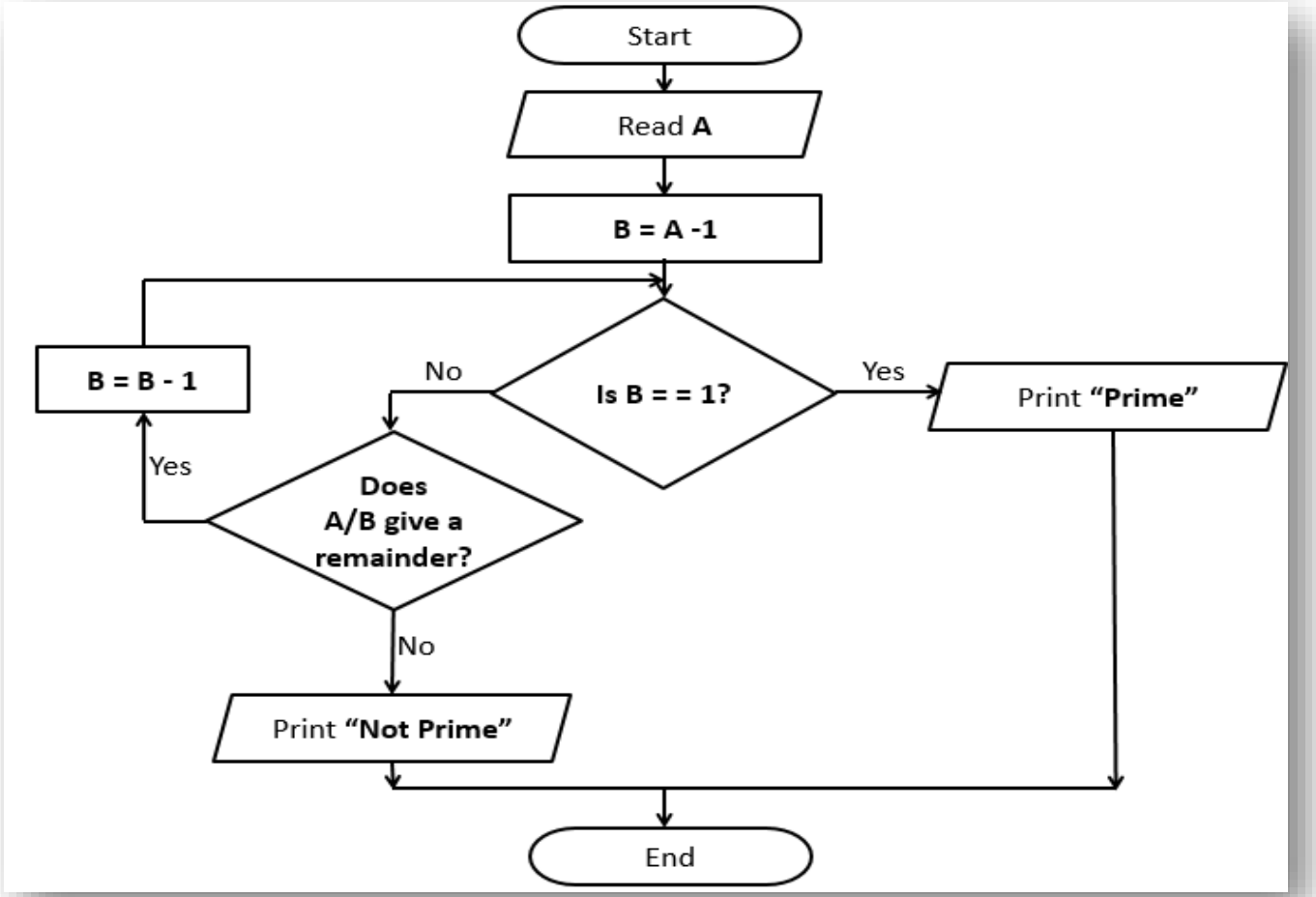
✓ Read **three numbers**, and find the **largest** Number



✓ Find $\sum_{A=1}^{5} A$



- ✓ Read a number (Number >1) and **check** if it's a **Prime** number.



Advantages of Flowchart

- Better Communication
- Proper program documentation
- Efficient coding
- Systematic debugging
- Systematic testing

Pseudocode

- A **program planning tool** where program logic is written in an ordinary natural language using a structure that resembles computer instructions

- **" Pseudo"** means imitation or false and **" Code"** refers to the instructions written in a programming language. Hence, **pseudocode** is an imitation of actual computer instructions

- Because it emphasizes the design of the program, **pseudocode** is also called ***Program Design Language (PDL)***

Pseudocode Logic Structure

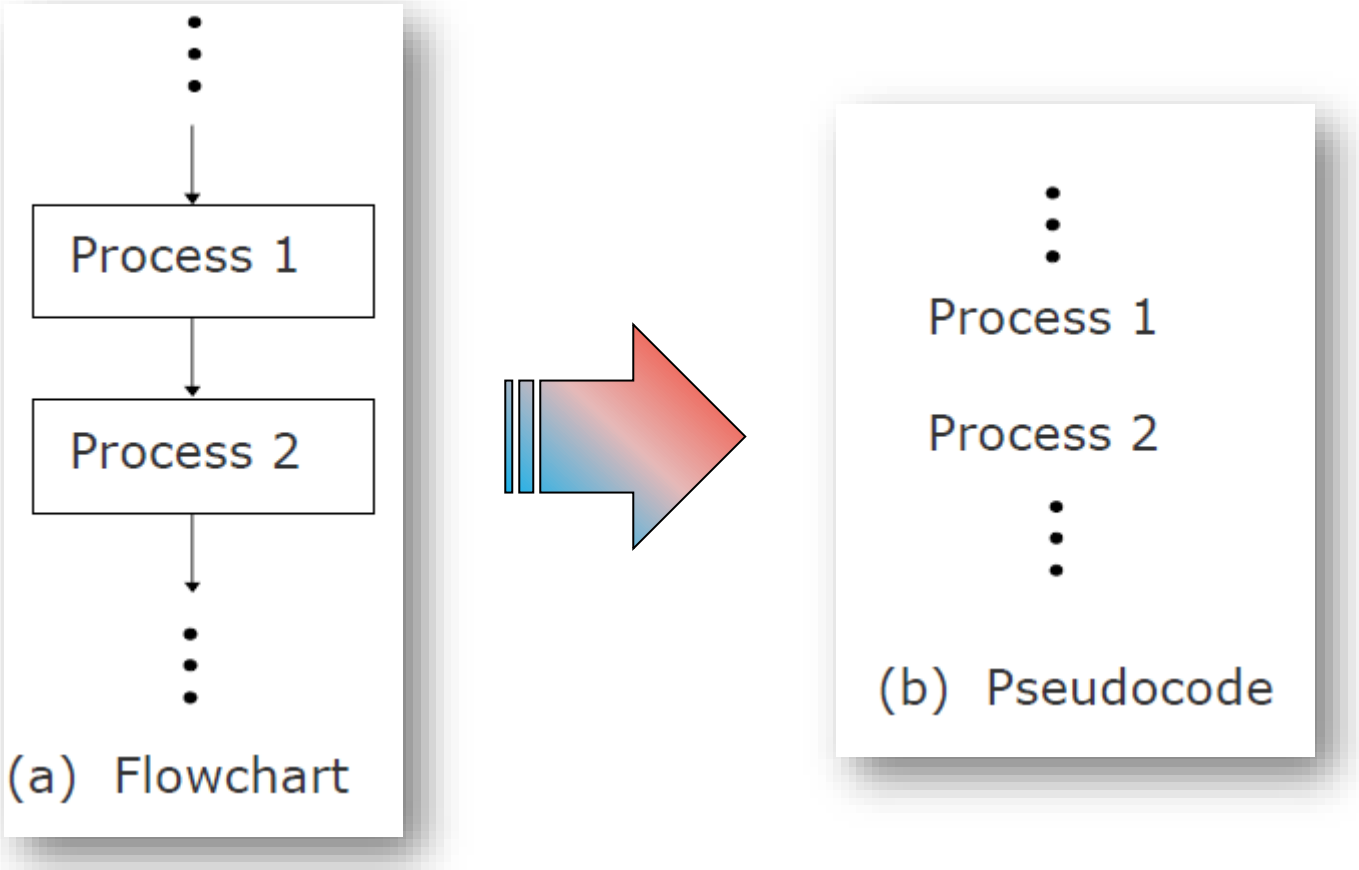
- Any program logic can be expressed by using only following **three** simple **logic** structures:

1) **Sequence** logic,
2) **Selection** logic, and
3) **Iteration** (or looping) logic

- Programs structured by using only these three logic structures are called **structured programs**, and the technique of writing such programs is known as **structured programming.**

Sequence Logic

- ✓ It is used for performing instructions one after another in sequence.



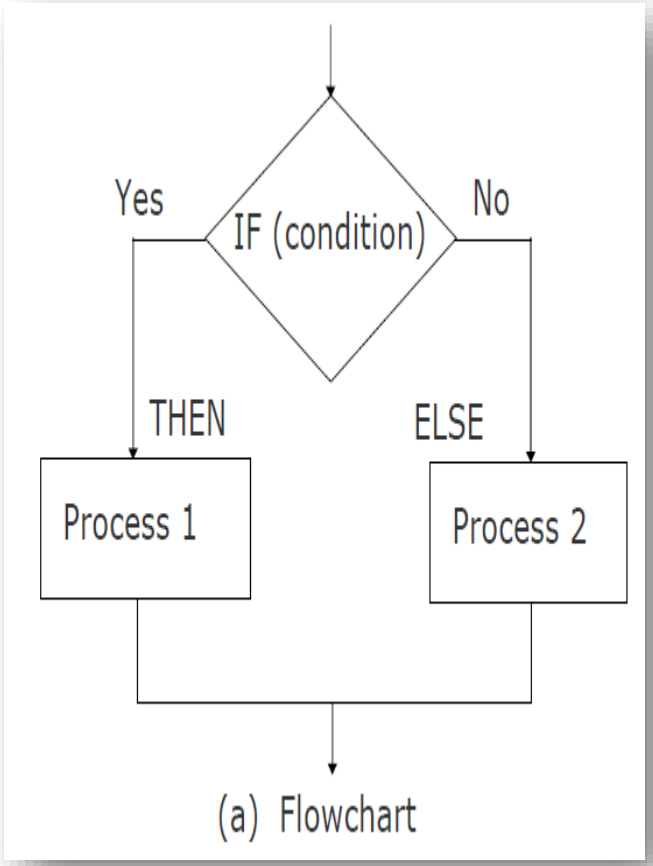
Selection Logic

- Also known as **decision logic**, it is used for making decisions

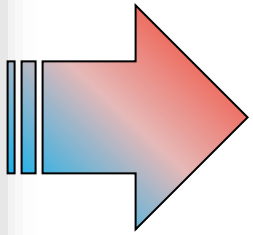
- **Three** popularly used selection logic structures are

    1) **IF…THEN…ELSE…ENDIF**
    2) **IF…THEN…ENDIF**
    3) **CASE…ENDCASE**

Selection Logic (IF…THEN… ELSE… ENDIF) Structure



(a) Flowchart



⋮

IF Condition

THEN Process 1

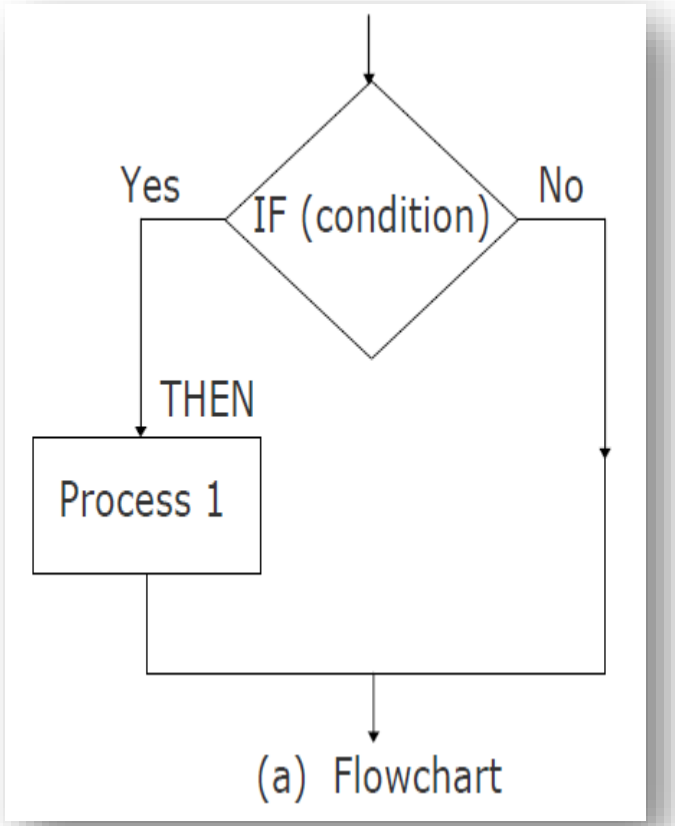
ELSE Process 2

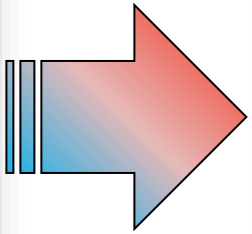
ENDIF

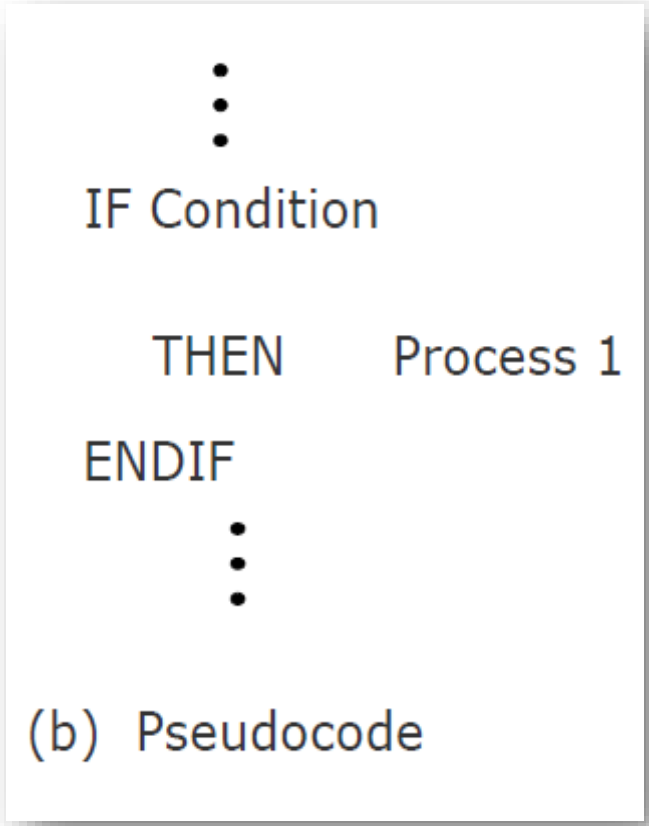
⋮

(b) Pseudocode

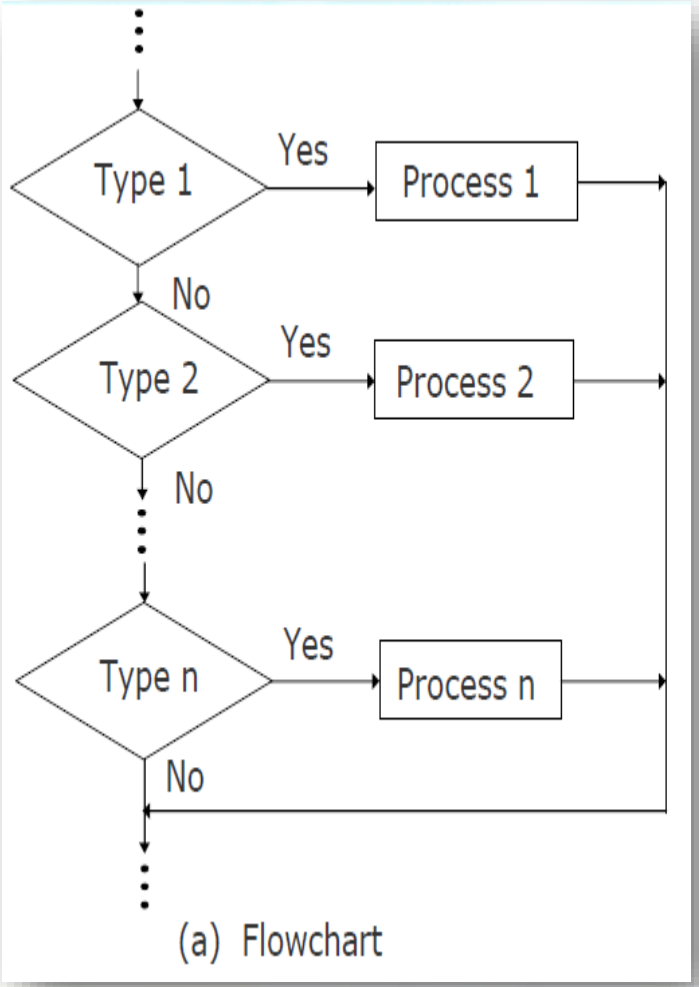
Selection Logic (IF…THEN…ENDIF) Structure

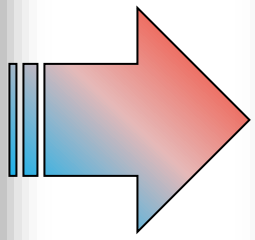






Selection Logic (CASE…ENDCASE) Structure





```
CASE Type
    Case Type 1:    Process 1
    Case Type 2:    Process 2
    Case Type n:    Process n
ENDCASE
```

(b) Pseudocode

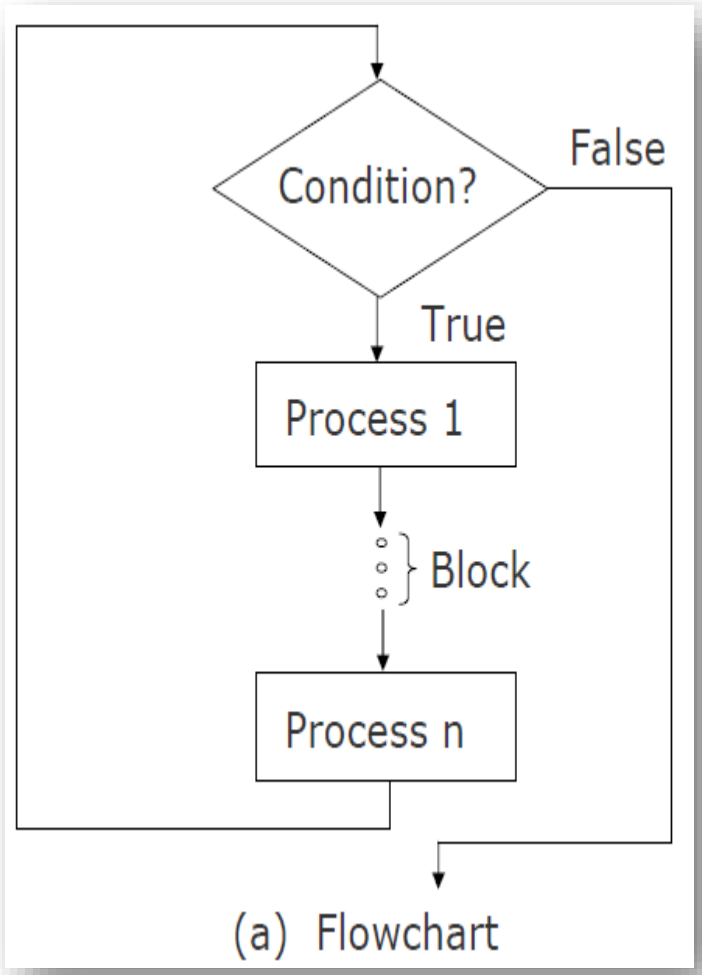
Iteration (Loop) Logic

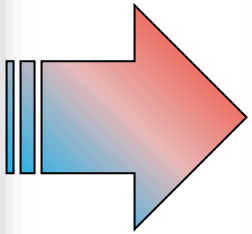
- Used to produce **loops** in program logic when one or more instructions may be executed several times depending on some conditions

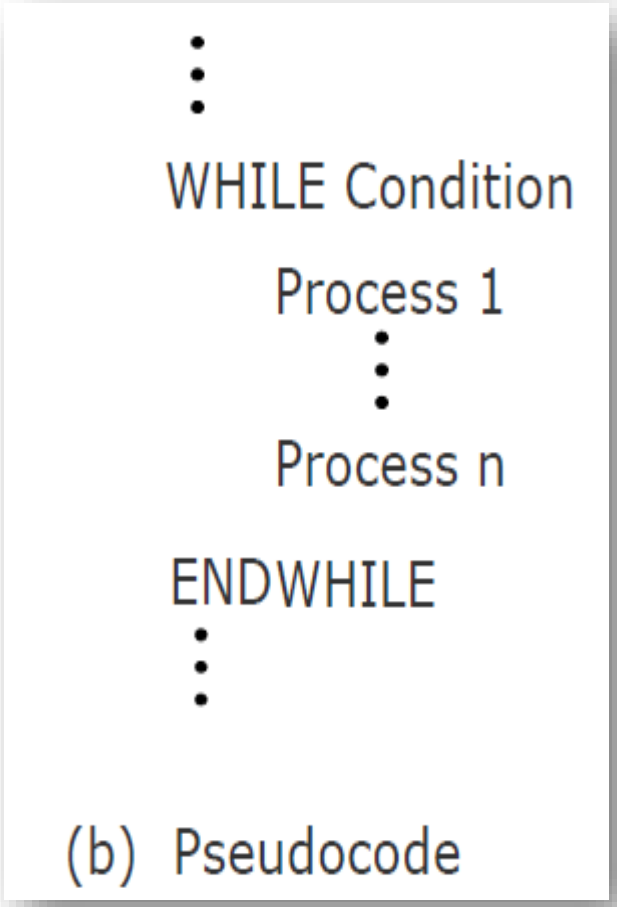
- **Two** popularly used iteration logic structures are:

  1) **DO…WHILE (WHILE…ENDWHILE)**
  2) **REPEAT…UNTIL**

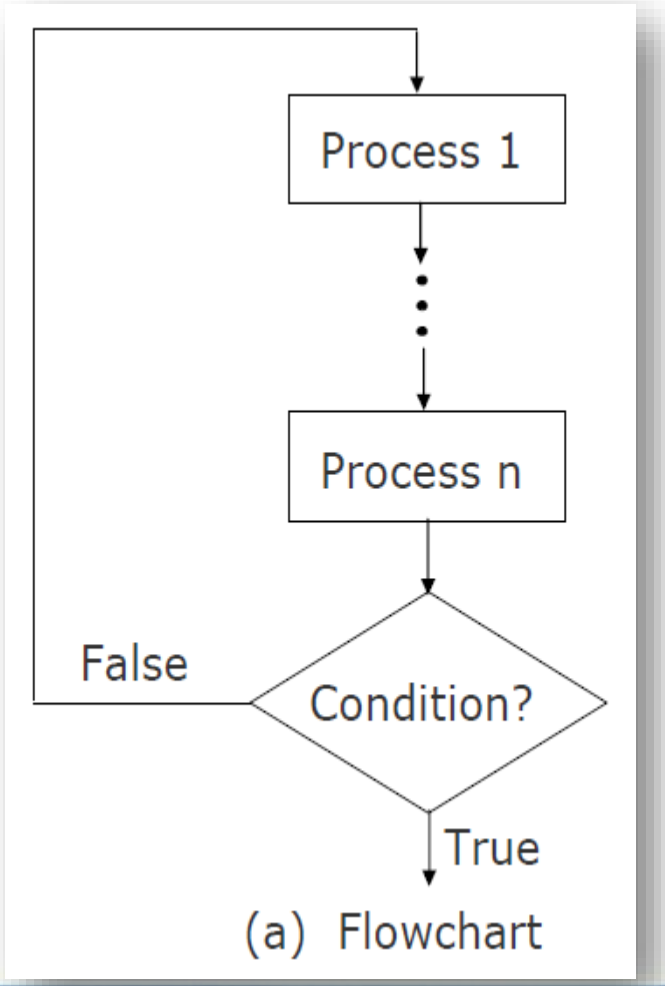
Iteration Logic (WHILE…ENDWHILE) Structure

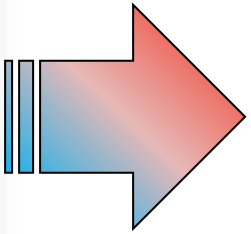


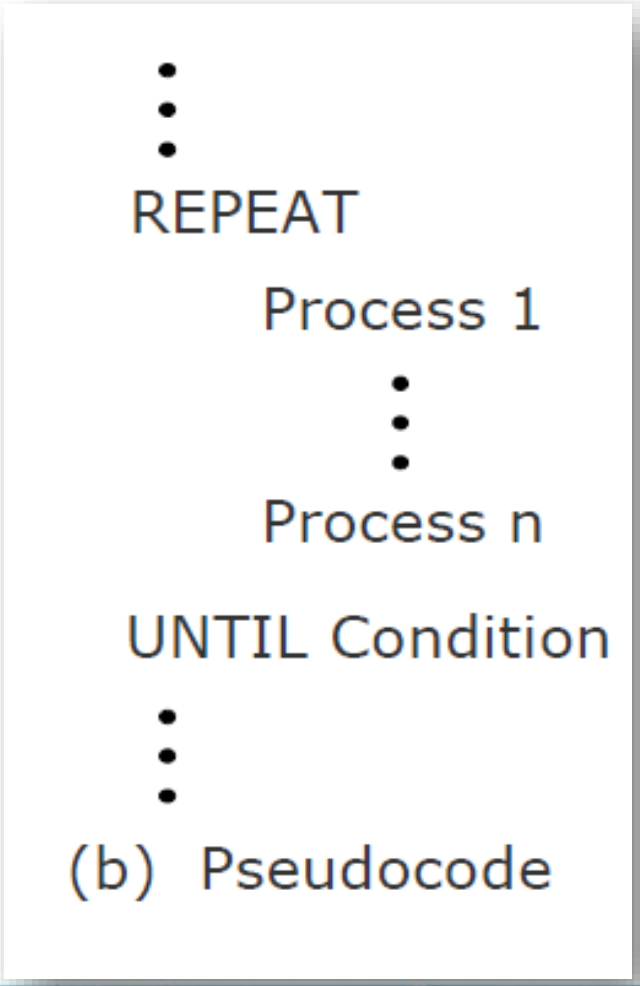




Iteration Logic (REPEAT…UNTIL) Structure







Pseudocode Examples

- ✓ **Read** a number and print it out.

```
Read A
Print A
```

- ✓ **Read** a number and find its **square** out.

```
Read A
B = A^2
Print B
```

- ✓ **Read** a number, **check** if it is **odd** or **even**.

```
Read A
    IF (A/2 gives a remainder)
            THEN Print "It's Odd"
            ELSE Print "It's Even"
    ENDIF
```

- ✓ Read **three** numbers, and find the **Largest** Number

```
Read A
Read B
Read C
IF (A>B)
        THEN IF (A>C)
                    THEN Print A
                    ELSE Print C
        ENDIF
        ELSE IF (B>C)
                    THEN Print B
                    ELSE Print C
        ENDIF
ENDIF
```

✓ Find $\sum_{A=1}^{5} A$

```
Sum = 0
A = 1
WHILE (A <> 6)
        Sum = Sum + A
        A = A + 1
ENDWHILE
Print Sum
```

(A != 6)

OR

```
Sum = 0
A = 1
REPEAT
        Sum = Sum + A
        A = A + 1
UNTIL (A == 6)
Print Sum
```

- ✓ Read a number (Number >1) and **check** if it's a **Prime** number .

```
Read A
B = A - 1
IsPrime=True
WHILE (B != 1)
     IF (A/B gives no remainder)
          THEN IsPrime= False
     ENDIF
B = B - 1
ENDWHILE
IF (IsPrime == true)
     THEN Print "Prime"
     ELSE Print "Not Prime"
ENDIF
```

Logic value

Advantages of Pseudocode

- **Converting** a pseudocode to a programming language is much **more easier** than converting a flowchart to a programming language

- As **compared** to a flowchart, it is easier to modify the pseudocode of a program logic when program modifications are necessary

- **Writing** of pseudocode involves much **less time** and **effort** than drawing an equivalent flowchart as it has only a **few rules** to follow

Introducing C++ Programming Language

C++ is the most used and most popular programming language developed by **Bjarne Stroustrup**. C++ is a **high-level** and **object-oriented programming** language. This language allows developers to write clean and efficient code for **large applications** and **software development**, **game** development, and **operating system** programming. It is an *expansion* of the *C programming* language to include Object Oriented Programming (**OOPs**) and is used to develop programs for computers.

Why Learn C++?

- C++ is one of the most used and **popular** programming languages.
- C++ is used in making *operating systems*, *embedded* systems, and *Graphical User Interfaces.*
- It is an *object-oriented* programming language that implements all the OOPs concepts such as *Abstraction*, *Encapsulation*, and *Inheritance*, which gives a clear structure to programs and allows code to be reused, lowering development costs and providing *security*.
- It is *portable* and can be used to create applications that can be *adapted* to multiple platforms.
- C++ is *easy* to learn so that you can choose it as your first programming language.
- It makes programming easy for programmers to switch to C++ because its syntax is similar to C, Java, and C#.

What is Compiler?

Computers *understand* only one language and that language consists of sets of instructions made of **ones** and **zeros**. This computer language is appropriately called *machine language*. A single instruction to a computer could look like this:

00000	10011110

A particular computer's machine language program that allows a user to input two numbers, adds the two numbers together, and displays the total could include these machine code instructions:

00000	10011110
00001	11110100
00010	10011110
00011	11010100
00100	10111111
00101	00000000

To make programming *easier*, **high-level** languages have been *developed*. High level programs also make it *easier* for programmers to *inspect* and *understand*, each other's programs easier.

This is a portion of code written in **C++** that accomplishes the exact same purpose:

```
int a, b, sum;
cin >> a;
cin >> b;
sum = a + b;
cout << sum << endl;
```

Even if you cannot really understand the code above, you should be able to appreciate how much easier it will be to program in the C++ language as opposed to machine language.

Because a computer can only understand *machine* language and humans wish to write in high level languages.

High level languages have to be *re-written* (translated) into *machine* language at some point. This is done by special programs called ***compilers***, ***interpreters***, or ***assemblers*** that are built into the various programming applications.

C++ is designed to be a *compiled* language, meaning that it is generally translated into machine language that can be understood directly by the system, making the generated program highly efficient. For that, a set of tools are needed, known as the development toolchain, whose core are a *compiler* and its *linker*.

Console Programs

Console programs are programs that use *text* to *communicate* with the user and the environment, such as *printing* text to the screen or *reading* input from a keyboard.

Console programs are **easy** to *interact* with, and generally have a *predictable* behavior that is identical across all platforms. They are also simple to implement and thus are very useful to learn the basics of a programming language:

The examples in these tutorials are all console programs. The way to compile console programs depends on the particular tool you are using. The easiest way for beginners to compile C++ programs is by using an *Integrated Development Environment* (**IDE**).

An **IDE** generally integrates several development tools, including a *text editor* and tools to compile programs directly from it. Here you have instructions on how to compile and run console programs using different free Integrated Development Interfaces (**IDEs**):

IDE	OS	Console programs
Code::blocks	Windows/Linux/MacOS	Compile console programs using Code::blocks
Visual Studio Express	Windows	Compile console programs using VS Express 2013
Dev-C++	Windows	Compile console programs using Dev-C++

Structure of a Program

The best way to learn a programming language is by **writing programs**. Typically, the first program beginners write is a program called **"Hello World"**, which simply prints "Hello World" to your *computer screen*. Although it is very *simple*, it contains all the *fundamental* components ***C++*** programs have:

```
// my first program in C++
#include <iostream>

int main()
{
std::cout << "Hello World!";
}
```



```
Hello World!
```

The **left** panel above shows the ***C++ code*** for this program.

The **right** panel shows the ***result*** when the program is *executed* by a computer.

The ***grey*** numbers to the **left** of the panels are *line numbers* to make *discussing* programs and researching *errors* easier. They are not part of the program.

Let's *examine* this program line by line:

Line 1: // my first program in C++

Two slash signs indicate that the rest of the line is a ***comment*** inserted by the *programmer* but which has no effect on the behavior of the program. Programmers use them to include short *explanations* or *observations* concerning the code or program. In this case, it is a *brief* introductory description of the program.

Line 2: ***# include <iostream>***

Lines beginning with a ***hash*** sign ***(#)*** are directives read and interpreted by what is known as the *preprocessor*. They are **special** lines interpreted before the compilation of the program itself begins. In this case, the directive #include <**iostream**>, instructs the preprocessor to include a section of standard C++ code, known as **header** iostream, that allows to perform *standard* **input** and **output** operations, such as *writing* the output of this program (**Hello World**) to the screen.

*Line 3****: A blank line.***

Blank lines have no effect on a program. They simply improve readability of the code.

Line 4: ***int main ()***

This line initiates the **declaration** of a function. Essentially, a function is a group of code statements which are given a name: in this case, this gives the name "**main**" to the group of code statements that follow. Their definition is introduced with a *succession* of a type (**int**), a name (main) and a pair of parentheses (**()**), *optionally* including parameters.

The function named ***main*** is a special function in all C++ programs; it is the function called when the program is run. The *execution* of all C++ programs begins with the ***main*** function, regardless of where the function is actually located within the code.

Lines 5 and 7: ***{*** *and* ***}***

The open brace ({) at line **5** indicates the *beginning* of main's function definition, and the closing brace (}) at line **7**, indicates its *end*. Everything between these braces is the *function's body* that defines what happens when main is called. All functions use braces to indicate the beginning and end of their definitions.

Line 6: ***std****::****cout*** ***<< "Hello World!";***

This line is a C++ statement. A statement is an ***expression*** that can actually produce some effect. It is the meat of a program, specifying its actual behavior. Statements are *executed* in the same order that they appear within a function's body. This statement has three parts: First, ***std::cout***, which identifies the standard character output device (usually, this is the computer screen).

Second, the insertion operator (<<), which indicates that what *follows* is inserted into *std::cout*.

Finally, a *sentence* within quotes ("***Hello world!***"), is the content inserted into the standard *output*.

Notice that the statement ends with a semicolon (**;**). This character marks the end of the statement, just as the period ends a sentence in English. All C++ statements **must** end with a ***semicolon*** character. One of the most *common* syntax **errors** in C++ is forgetting to end a statement with a *semicolon*.

There is a line *containing* a **comment** (beginning with //). There is a line with a directive for the preprocessor (beginning with #). There is a line that defines a function (in this case, the ***main*** function). And, finally, a line with a statements ending with a ***semicolon*** (the insertion into ***cout***), which was within the *block* delimited by the *braces* (**{ }**) of the **main** function.

The program has been structured in *different* lines and properly indented, in order to make it *easier* to understand for the humans reading it. But C++ does not have *strict* rules on indentation or on how to *split* instructions in *different* lines. *For example*, instead of

```
#include <iostream>

int main ()
{
std::cout << " Hello World!";
}
```

It could be written:

```
int main () { std::cout << "Hello World!"; }
```

All in a ***single*** line, and this would have had *exactly* the same meaning as the preceding code.

In C++, the *separation* between statements is specified with an ending semicolon (**;**), with the separation into different lines not mattering at all for this purpose. Many statements can be written in a single line, or each statement can be in its own line. The *division* of code in different lines serves only to make it *more legible* and *schematic* for the humans that may read it, but has no effect on the actual behavior of program.

Let's add an additional statement to our first program:

```
#include <iostream>

int main ()
{
std::cout << "Hello World! ";
std::cout << "This is  a C++ program";
}
```

```
Hello World! This is a C++ program
```

In this case, the program performed **two** insertions into **std::cout** in *two* different statements. Once again, the separation in different lines of code simply gives greater *readability* to the program, since main could have been perfectly valid defined in this way:

```
1. #include <iostream>
2.
3. int main (){ std::cout << " Hello World! "; std::cout << " This is a C++
   program "; }
```

A different way to write code could have also been *divided* into *more* code lines instead:

```
#include <iostream>

int main ()
{
std::cout << "Hello World!"; std::cout
<< "This is a C++ program";
}
```

And the result would again have been *exactly* the *same* as in the previous examples.

Comments:

Comments do **not** affect the operation of the program; however, they provide an *important* tool to document directly within the source code what the program *does* and *how* it operates.

C++ supports ***two ways*** of commenting code:

```
// line comment
/* block comment */
```

The ***first*** of them, known as *line comment*, discards everything from where the pair of slash signs (//) are found up to the *end* of that same line. The ***second*** one, known as *block comment*, discards everything between the /* characters and the first appearance of the */ characters, with the possibility of including ***multiple lines.***

Let's add comments to our *second* program:

```
/* my second program in C++    with more comments */
#include <iostream>
int main ()
{
std::cout << "Hello World! ";      // prints Hello World!
std::cout << "This is C++ program"; // prints I'm a C++ program
}
```

If *comments* are included within the *source* code of a program without using the comment characters combinations //, /* or */, the *compiler* takes them as if they were C++ expressions, most likely causing the compilation to *fail* with one, or *several*, ***error*** messages.

Using namespace std:

If you have seen C++ code before, you may have seen **cout** being used instead of **std::cout**. Both name the same object: the *first* one uses its ***unqualified*** name (**cout**), while the *second* ***qualifies*** it directly within the namespace **std** (as **std::cout**).

cout is part of the *standard* library, and all the elements in the standard C++ library are *declared* within what is called a ***namespace***: the namespace ***std***.

In order to refer to the elements in the std namespace a program shall either qualify each and every use of elements of the library (as we have done by *prefixing* **cout** with **std::**), or introduce visibility of its components. The most typical way to introduce visibility of these components is by means of using ***declarations***:

```
using namespace std;
```

The above declaration *allows* ***all*** elements in the **std namespace** to be accessed in an *unqualified* manner (without the **std::** prefix). With this in mind, the last example can be rewritten to make unqualified uses of **cout** as:

```
/* my second program in C++    with more comments */
#include <iostream>
using namespace std;
int main ()
{
cout << "Hello World! ";      // prints Hello World!
cout << "This is a C++ program"; // prints I'm a C++ program
}
```

```
Hello World!  This is a C++ program
```

Both ways of accessing the elements of the **std namespace** (*explicit* qualification and using declarations) are valid in C++ and produce the exact same behavior. For simplicity, and to improve readability.

Standard output (cout)

On most program environments, the standard output by default is the *screen*, and the C++ stream object defined to access it is **cout**.

For formatted output operations, **cout** is used together with the **insertion operator**, which is written as "**<<**"

```
cout << "Output sentence";      // prints Output sentence on screen
cout << 120;                    // prints number 120 on screen
cout << x;                      // prints the value of x on screen
```

The **<<** operator inserts the data that follows it into the stream that precedes it. In the examples above, it inserted the literal string Output sentence, the number **120**, and the value of variable **x** into the standard output stream **cout**. Notice that the *sentence* in the first statement is enclosed in *double quotes* (**"**) because it is a ***string literal***, while in the last one, **x** is not. The double quoting is what makes the difference; when the *text* is enclosed between them, the text is printed literally; when they are not, the text is interpreted as the *identifier* of a *variable*, and its value is printed instead. ***For example***, these two sentences have very different results:

```
cout << "Hello";        // prints Hello
cout << Hello;          // prints the content of variable Hello
```

Multiple insertion operations **(<<)** may be chained in a *single statement*:

```
cout << "This " << " is a " << "single C++ statement";
```

This last statement would print the text This is a single C++ statement. Chaining insertions is especially useful to mix literals and variables in a *single statement*:

```
cout << "I am " << age << " years old and my zipcode is " << zipcode;
```

Assuming the **age** variable contains the value **24** and the **zipcode** variable contains **90064**, the output of the previous statement would be:

```
I am 24 years old and my zipcode is 90064
```

What **cout** does not do automatically is add line breaks at the end, unless instructed to do so. ***For example***, take the following *two* statements inserting into **cout**:

```
cout << "This is a sentence."; cout << "This is another sentence.";
```

The output would be in a *single line*, without any line breaks in between. Something like:

```
This is a sentence. This is another sentence.
```

Using `endl` :

`endl` is a manipulator used with the standard output stream (**`cout`**) to insert a **newline** character and flush the output buffer. It is part of the <**`iostream`**> library.

```
cout << "Hello World! " <<  endl;
cout << "I'm a C++ program";
```

```
Hello World!
This is a C++ program
```

In C++, **`\n`** and **`\t`** are **escape sequences** used to represent special characters in strings.

- **`\n`** is used to insert a **new line**. It moves the cursor to the beginning of the next line.
- **`\t`** is used to insert a **tab** character. It adds a **horizontal tab** space, which is often used for alignment. As shown in example:

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

int main()
{
    // Using \n for a new line
    cout << "Hello, World!\n";
    cout << "Welcome to C++ programming.\n";

    // Using \t for tab spaces
    cout << "Name:\t Ali Kemal\n";
    cout << "Age:\t 25\n";
}
```

```
Hello, World!
Welcome to C++ programming.
Name:    Ali Kemal
Age:     25
```

Exercises

Exercise 1

Write a C++ program that outputs the following text on screen:

```
Oh what
a happy day!
Oh yes,
what a happy day!
```

Use the manipulator **endl** where appropriate.

Exercise 2

The following program contains several **errors**:

```
*/ Now you should not forget your glasses //
#include <stream>
int main
{
cout << "If this text",
cout >> " appears on your display, ";
cout << " endl;"
cout << 'you can pat yourself on '
<< " the back!" << endl.
)
```

Resolve the errors and run the program to test your changes.

Exercise 3

Write a **C++** Code to display the **matrix** shape:

1 2
3 4

Variables and types:

Programming is not limited only to printing simple texts on the screen. In order to go a little further on and to become able to write programs that perform useful tasks that really save us work, we need to introduce the concept of **variables**.

Let's imagine that I ask you to remember the number **5**, and then I ask you to also memorize the number **2** at the same time. You have just stored two different values in your memory (**5** and **2**). Now, if I ask you to ***add*** **1** to the first number I said, you should be retaining the numbers **6** (that is **5+1**) and **2** in your memory. Then we could, *for example*, ***subtract*** these values and obtain **4** as result. The whole process described above is a simile of what a computer can do with *two variables*. The same process can be expressed in C++ with the following set of statements:

```
a = 5;
b = 2;
a = a + 1;
result = a - b;
```

We can now define *variable* as a portion of memory to store a value. Each *variable* needs a *name* that *identifies* it and *distinguishes* it from the others. *For example*, in the previous code the variable names were **a, b**, and **result**, but we could have called the variables *any* names we could have come up with, as long as they were valid C++ identifiers.

Identifiers:

A valid ***identifier*** is a *sequence* of one or *more letters*, *digits*, or *underscore* characters (_). *Spaces*, *punctuation* marks, and *symbols* **cannot** be part of an identifier. In addition, identifiers shall *always* begin with a ***letter***. They can also begin with an *underline* character **(_),** but such identifiers are-on most cases- considered reserved for compiler-specific keywords or external identifiers, as well as *identifiers* containing *two* successive underscore characters anywhere. In no case can they *begin* with a *digit*.

C++ uses a number of ***keywords*** to identify *operations* and data *descriptions*; therefore, identifiers created by a programmer ***cannot*** match these *keywords*. The standard reserved keywords that cannot be used for programmer created identifiers are:

```
alignas, alignof, and, and_eq, asm, auto, bitand, bitor, bool,
break, case, catch, char, char16_t, char32_t, class, compl,
const, constexpr, const_cast, continue, decltype, default,
delete, do, double, dynamic_cast, else, enum, explicit, export,
extern, false, float, for, friend, goto, if, inline, int, long,
mutable, namespace, new, noexcept, not, not_eq, nullptr,
operator, or, or_eq, private, protected, public, register,
```

reinterpret_cast, return, short, signed, sizeof, static, static_assert, static_cast, struct, switch, template, this, thread_local, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while, xor, xor_eq.

Specific *compilers* may also have additional specific reserved keywords. Very important: The C++ language is a "***case sensitive***" language. That means that an identifier written in ***capital*** letters is not equivalent to another one with the same name but written in ***small*** letters. Thus, for example, the **`RESULT`** variable is not the same as the **`result`** variable or the **`Result`** variable, these are ***three*** different identifiers.

Fundamental data types:

Fundamental data types are basic types implemented directly by the language that represent the basic storage units supported natively by most systems. They can mainly be classified into:

1) **Character types:** They can represent a *single* character, such as **'A'** or **'$'**. The most basic type is **char**, which is a *one-byte* character. Other types are also provided for wider characters.
2) **Numerical integer types:** They can store a whole number value, such as **7** or **1024**. They exist in a *variety* of *sizes*, and can either be *signed* or *unsigned*, depending on whether they support *negative* values or not.
3) **Floating-point types:** They can represent real values, such as **3.14** or **0.01**, with different levels of *precision*, depending on which of the *three* floating point types is used.
4) **Boolean type:** The *boolean* type, known in C++ as bool, can only represent one of *two* states, **true** or **false**. Here is the complete list of fundamental types in C++:

Data Type	Size (in Bytes)	Meaning
`signed int` or int	4	used for integers (range -2,147,483,648 to 2,147,483,647)
`unsigned int`	4	can only store positive integers (range 0 to 4,294,967,295)
`short`	2	used for small integers (range -32768 to 32767)
`unsigned short`	2	used for small positive integers (range 0 to 65,535)
`long`	at least 4	used for large integers (range –2,147,483,648 to 2,147,483,647)
`unsigned long`	4 or 8	used for large positive integers (range 0 to 4,294,967,295))
`long long`	8	used for very large integers (range -2^63 to 2^63 – 1)
`unsigned long long`	8	used for very large positive integers or 0 (range 0 to 2^64 - 1)
`float`	4	used for floating-point numbers
`double` and `long double`	8, 12, or 16	used for large floating-point numbers
`signed char`	1	used for characters (guaranteed range -127 to 127)
`unsigned char`	1	used for characters (range 0 to 255)
`char`	1	Character
`wchar_t`	2	Wide Character
`bool`	1	Boolean

`void`	**0**	**No storage**

Type sizes above are expressed in **bits**; the more bits a type has, the more distinct values it can represent, but at the same time, also consumes more space in memory:

Size	Unique representable values	Notes
8-bit	**256**	**= 2^8**
16-bit	**65 536**	**= 2^{16}**
32-bit	**4 294 967 296**	**= 2^{32} (~4 billion)**
64-bit	**18 446 744 073 709 551 616**	**= 2^{64} (~18 billion billion)**

For **integer** types, having more representable values means that the range of values they can represent is greater; *for example*, a ***16-bit unsigned integer*** would be able to represent ***65536*** distinct values in the range ***0*** to ***65535***, while its ***signed*** counterpart would be able to represent, on most cases, values between ***-32768*** and ***32767***. Note that the range of positive values is approximately **halved** in signed types compared to unsigned types, due to the fact that ***one*** of the ***16 bits*** is used for the ***sign***.

The types described above (**characters**, **integers**, **floating-point**, and **boolean**) are collectively known as *arithmetic types*. But fundamental types exist: **void**, which identifies the lack of type;

Declaration of variables:

C++ is a strongly-typed language, and requires every variable to be declared with its type before its first use. The syntax to declare a new variable in C++ is straightforward: we simply write the type followed by the variable name (i.e., its identifier). ***For example:***

```
int a;
float mynumber;
```

If declaring **more** than one variable of the ***same*** type, they can all be declared in a single statement by separating their identifiers with commas. ***For example***:

```
int a, b, c;
```

To see what variable declarations, look like in action within a program, as in ***example***:

```
// operating with variables
#include <iostream>
using namespace std;
int main ()
{
```

```
// declaring variables:
  int a, b;
  int result;
// process:
  a = 5; b = 2;
  a = a + 1;
  result = a - b;
// print out the result:
  cout << result;
// terminate the program:
  return 0;
}
```



```
4
```

Initialization of variables:

In **C++**, there are **three** ways to initialize variables. They are all ***equivalent*** and are reminiscent of the evolution of the language over the years:

The **first** one, known as *c-like* initialization (because it is *inherited* from the **C** language), consists of appending an equal sign followed by the value to which the variable is initialized:

type identifier = ***initial_value***, such as:

```
int x = 0;
```

A **second** method, known as *constructor* initialization (introduced by the **C++** language), *encloses* the initial value between **parentheses** (**()**):

type identifier (initial_value); such as:

```
int x(0);
```

Finally, a **third** method, known as *uniform* initialization, similar to the above, but using **curly braces** ({}) instead of *parentheses* (this was introduced by the revision of the C++ standard, in 2011):

type identifier {initial_value}; such as:

```
int x{0};
```

All ***three*** ways of initializing variables are ***valid*** and equivalent in C++.

```
// initialization of variables

#include <iostream>
using namespace std;

int main ()
{
  int a=5;               // initial value: 5
  int b(3);              // initial value: 3
  int c{2};              // initial value: 2
  int result;            // initial value undetermined

  a = a + b;
  result = a - c;
  cout << result;

  return 0;
}
```



Literal Constants:

Literal constants (or simply ***literals***) in **C++** are values used *directly* in the code to represent **fixed** data values. These constants are **not** stored in variables, but are used ***directly*** in expressions or function calls. C++ provides *several* types of literal constants, each with a specific type:

1) **Integer Literals:** these represent integer values. For example:

```
int x = 5;          // 5 is an integer literal
int dec = 42;       // decimal literal
int oct = 052;      // octal literal (prefix 0)
int hex = 0x2A;     // hexadecimal literal (prefix 0x or 0X).
```

2) **Floating-Point Literals:** these represent real numbers, with or without a fractional part. For example:

```
double pi = 3.14159;          // 3.14159 is a floating-point literal
float pi = 3.14f;             // 'f' suffix indicates float type
double largeNumber = 1.2e3; // equivalent to 1.2 * 10^3
```

3) **Character and String Literals:** Represent individual characters. For example

```
char letter = 'A';                    // 'A' is a character literal
const char* str = "Hello, World!";    // "Hello, World!" is a string
```

4) **Boolean Literals:** Represent the truth values **true** and **false**. For example:

```
bool flag = true;  // true is a boolean literal
```

Typed constant expressions:

Sometimes, it is just convenient to give a name to a constant value:

```
const double pi = 3.1415926;
const char tab = '\t';
```

We can then use these **names** instead of the **literals** they were defined to:

```
#include <iostream>
using namespace std;
const double pi = 3.14159;
const char newline = '\n';
int main ()
{
  double r=5.0;         // radius
  double circumference;
  circumference = 2 * pi * r;
  cout << circumference;
  cout << newline;
}
```



```
31.4159
```

Preprocessor definitions (#define):

Another mechanism to name constant values is the use of preprocessor definitions. They have the following form:

#define identifier replacement

For example

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

```
#define PI 3.14159
#define NEWLINE '\n'
int main ()
{
  double r=5.0;         // radius
  double circumference;
  circumference = 2 * PI * r;
  cout << circumference;
  cout << NEWLINE;
}
```



```
31.4159
```

Note that the **#define** lines are preprocessor directives, and as such are single-line instructions that -*unlike C++ statements*- do not require semicolons **(;)** at the end; the directive extends automatically until the end of the line.

Standard input (cin):

In most program environments, the standard input by default is the keyboard, and the C++ stream object defined to access it is **cin**.

For formatted input operations, **cin** is used together with the extraction operator, which is written as "**>>**". This operator is then *followed* by the *variable* where the extracted data is stored.

```
int age;
cin >> age;
```

The first statement *declares* a variable of type int called **age**, and the second extracts from **cin** a value to be stored in it. This operation makes the program wait for input from **cin**; generally, this means that the program will *wait* for the user to enter some sequence with the keyboard. In this case, note that the characters introduced using the keyboard are only transmitted to the program when the **ENTER** (or **RETURN**) key is pressed. Once the statement with the extraction operation on **cin** is reached, the program will *wait* for as long as needed until some input is introduced. The extraction operation on **cin** uses the type of the variable after the **>>** operator to determine how it *interprets* the characters **read** from the *input*; if it is an **integer**, the format expected is a series of **digits**, if a **string** a sequence of **characters**, etc.

```
// i/o example
#include <iostream>
using namespace std;
int main ()
{
        int i;
       cout << "Please enter an integer value: ";
       cin >> i;
       cout << "The value you entered is " << i;
       cout << " and its double is " << i*2 << ".\n";
       return 0;
}
```

Extractions on **cin** can also be chained to request more than one datum in a single statement:

```
cin >> a >> b;
```

This is equivalent to:

```
cin >> a;
cin >> b;
```

In both cases, the user is expected to introduce **two** values, one for variable **a**, and another for variable **b**. Any kind of *space* is used to separate two consecutive input operations; this may either be a space, a tab, or a new-line character.

cin and strings:

The extraction operator can be used on **cin** to get strings of characters in the same way as with fundamental data types:

```
string mystring;
cin >> mystring;
```

However, **cin** extraction always considers spaces (whitespaces, tabs, newline...) as terminating the value being extracted, and thus extracting a string means to always extract a single word, not a phrase or an entire sentence. To get an **entire line** from **cin**, there exists a function, called **getline**, that takes the stream (**cin**) as *first* argument, and the ***string*** variable as ***second***.

For example:

```
// cin with strings
#include <iostream>
#include <string>
using namespace std;
int main ()
{
     string mystr;
     cout << "What's your name? ";
     getline (cin, mystr);
     cout << "Hello " << mystr << ".\n";
     cout << "What is your favorite Colour? ";
     getline (cin, mystr);
     cout << "I like " << mystr << " too!\n";
     return 0;
}
```

```
What's your name? Sarmed
Hello Sarmed.
What is your favorite Colour? White
I like White too!
```

Example: Using **cout** command to display C++ data types

```
// cout usage to display Data type
#include <iostream>
using namespace std;

int main()
{
  // Basic output: strings and numbers
  cout << "Hello, world!" << endl;
  cout << 123 << endl;
  cout << 3.14159 << endl;

  // Combining different data types
  int age = 30;
  cout << "My age is: " << age << endl;
```

```
 // Outputting variables directly
 double temperature = 25.5;
cout << "The temperature is: " << temperature << " degrees Celsius." << endl;

 // Outputting characters
 char grade = 'A';
 cout << "Your grade is: " << grade << endl;

 // Outputting multiple items on the same line
 int x = 10, y = 20;
 cout << "x = " << x << ", y = " << y << endl;

 // Using escape sequences
 cout << "This is a tab:\tAnd this is a newline:\n" << endl;
 cout << "This is a backslash: \\" << endl;
 cout << "This is a double quote: \"" << endl;

 //Outputting boolean values.
 bool isTrue = true;
 cout << "The boolean value is: " << isTrue << endl;

 //Outputting without endl (no newline)
 cout << "This is part one. ";
 cout << "This is part two.";
 cout << endl;    //adds a newline at the end of the previous line.

 //outputting multiple lines with one cout.
 cout << "This is line 1\nThis is line 2\nThis is line 3\n";

 //Outputting hexadecimal and octal numbers.
 int num = 255;
cout << "Decimal: " << num << endl;
cout << "Hexadecimal: " << hex << num << endl;
cout << "Octal: " << oct << num << endl;
cout << dec;       //reset to decimal.

 return 0;
}
```

```
Hello, world!
123
3.14159
My age is: 30
The temperature is: 25.5 degrees Celsius.
Your grade is: A
x = 10, y = 20
This is a tab:  And this is a newline:

This is a backslash: \
This is a double quote: "
The boolean value is: 1
This is part one. This is part two.
This is line 1
This is line 2
This is line 3
Decimal: 255
Hexadecimal: ff
Octal: 377
```

Example: Using the **`sizeof`** Operator. This program tells you how much space each of the **12** fundamental types uses:

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

int main()
{
        // prints the storage sizes of the fundamental types:
cout << "Number of bytes used:\n";
cout << "\t  char: " << sizeof(char) << endl;
cout << "\t  short: " << sizeof(short) << endl;
cout << "\t  int: " << sizeof(int) << endl;
cout << "\t  long: " << sizeof(long) << endl;
cout << "\t  unsigned char: " << sizeof(unsigned char) << endl;
cout << "\t  unsigned short: " << sizeof(unsigned short) << endl;
cout << "\t  unsigned int: " << sizeof(unsigned int) << endl;
cout << "\t  unsigned long: " << sizeof(unsigned long) << endl;
cout << "\t  signed char: " << sizeof(signed char) << endl;
cout << "\t  float: " << sizeof(float) << endl;
```

```
cout << "\t  double: " << sizeof(double) << endl;
cout << "\t  long double: " << sizeof(long double) << endl;
}
```

```
Number of bytes used:
          char: 1
          short: 2
          int: 4
          long: 4
          unsigned char: 1
          unsigned short: 2
          unsigned int: 4
          unsigned long: 4
          signed char: 1
          float: 4
          double: 8
          long double: 16
```

Example: Print Number Entered by User

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

int main()
{
     int number;

     cout << "Enter an integer: ";
     cin >> number;

     cout << "You entered " << number;
     return 0;
}
```

```
Enter an integer: 23
You entered 23
```

Example: Program to Add Two Integers, which its output display "***num1+num2=sum***"

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

int main()
{

  int first_number, second_number, sum;
```

```
    cout << "Enter two integers: ";
    cin >> first_number >> second_number;

    // sum of two numbers in stored in variable sumOfTwoNumbers
    sum = first_number + second_number;

    // prints sum
    cout << first_number << " + " <<  second_number << " = " << sum;

    return 0;
}
```

```
Enter two integers: 42 52
42 + 52 = 94
```

OR

```
Enter two integers: 42
52
42 + 52 = 94
```

Test: **Explain** the following:

1) Try enter a **real number** (float type) instead of an **integer number** for **both first number and second number**.
2) What are the **maximum** digits of **integer** to be entered to get **correct** result?

Exercises

Exercise 1

What are the **errors** in the following**?**

```
1)    int a(2.5);
2)    int goto = 200;
3)    short big = 40000;
4)    double he's(1.2E+5);
5)    const long large;
6)    char st='My program';
7)    float val = 12345.12345;
8)    define G=9.81;
```

Exercise 2

Write and run a program that prints the **sum**, **difference**, **product**, and **quotient**, of **two** integers that are input ***interactively***. Using output statements such as "**`The sum =`** ".

Exercise 3

Write a program to find the **volume** of *cylinder*, by entering the dimensions and defining PI as **constant (**with **6** decimal places**)**.

Using the statement " **`The Cylinder Volume is:`** " as output.

Operators

Once introduced to variables and constants, we can begin to operate with them by using operators. What follows is a complete list of operators. At this point, it is likely not necessary to know all of them, but they are all listed here to also serve as reference.

Assignment operator (=):

The assignment operator assigns a value to a variable.

`x = 5;` This *statement* assigns the *integer* value **5** to the variable **x**.
`x = y;` This *statement* assigns to variable **x** the value contained in variable **y**.

The value of **x** at the moment this *statement* is executed is **lost** and **replaced** by the value of **y**. *For example*, let's have a look at the following code:

```
// assignment operator
#include <iostream>
using namespace std;
int main ()
{
int a, b;             // a:?,  b:?
a = 10;               // a:10, b:?
b = 4;                // a:10, b:4
a = b;                // a:4,  b:4
b = 7;                // a:4,  b:7
cout << "a:"; cout << a;
cout << " b:"; cout << b;
}
```

```
a:4 b:7
```

This program *prints* on screen the final values of **a** and **b** (**4** and **7**, respectively). Notice how a was not affected by the final modification of **b**, even though we declared **a = b** earlier.

```
y = 2 + (x = 5);
```

In this expression, **y** is assigned the result of adding **2** and the value of another assignment expression (which has itself a value of 5). It is roughly equivalent to:

```
x = 5;
y = 2 + x;
```

With the *final* result of assigning **7** to **y**.

The following expression is also valid in C++:

```
x = y = z = 5;
```

It assigns **5** to the all *three* variables: **x, y** and **z**; always from *right-to-left*.

Arithmetic operators:

The **five** arithmetical operations supported by C++ are:

Operator	Description
+	addition
-	subtraction
*	multiplication
/	division
%	Modulus

Operations of **addition**, **subtraction**, **multiplication** and **division** correspond literally to their respective mathematical operators. The last one, **Modulus** operator, represented by a percentage sign (**%**), gives the *remainder* of a *division* of **two** values. For example:

```
x = 11 % 3;
```

results in variable **x** containing the value **2**, since dividing **11** by **3** results in **3**, with a *remainder* of **2**.

Compound assignment:

Compound assignment operators *modify* the current value of a variable by performing an **operation** on it. They are *equivalent* to assigning the result of an operation to the *first* operand:

Expression	Equivalent To...
`y += x;`	`y = y + x;`
`x -= 5;`	`x = x - 5;`
`x /= y;`	`x = x / y;`
`price *= units + 1;`	`price = price * (units+1);`

and the same for all other compound assignment operators. ***For example***:

```
// Compound operator assignment
#include <iostream>
using namespace std;

int main ()
{
```

```
int a, b=3, c=5;
a = b;
a+=2;               // equivalent to a=a+2
b-=1;               // equivalent to b=b-1
c*=a+b;             // equivalent to c=c*(a+b)
cout << "a= "<< a <<endl;
cout << "b= "<< b <<endl;
cout << "c= "<< c;

}
```



```
a= 5
b= 2
c= 35
```

Increment and decrement:

Some expression can be *shortened* even more: the ***increase*** operator (**++**) and the ***decrease*** operator (**--**) increase or reduce by **one** the value stored in a *variable*. They are equivalent to **+=1** and to **-=1**, respectively. Thus:

```
++x;
```

```
x+=1;
```

```
x=x+1;
```

Are **all** equivalent in its functionality; the ***three*** of them *increase* by *one* the value of **x**.

A peculiarity of this operator is that it can be used both as a ***prefix*** and as a ***suffix***. That means that it can be written either *before* the variable name **(++x)** or *after* it **(x++)**. Although in simple expressions like **x++** or **++x**, both have exactly the same meaning; in other expressions in which the result of the increment or decrement operation is evaluated, they may have an **important difference** in their meaning: In the case that the increase operator is used as a ***prefix*** **(++x)** of the value, the expression evaluates to the ***final value of x***, once it is already increased. On the other hand, in case that it is used as a ***suffix*** (**x++**), the value is also increased, but the expression evaluates to the value that ***x had before being increased***. Notice the difference:

Example 1	Example 2
`x = 3; y = ++x;`	`x = 3; y = x++;`
`// x contains 4, y contains 4`	`// x contains 4, y contains 3`

In ***Example 1***, the value assigned to **y** is the value of **x** ***after being increased***. While in ***Example 2***, it is the value **x** ***had before been increased***.

Relational and comparison operators:

Two expressions can be *compared* using ***relational*** and ***equality*** operators. *For example*, to know if **two** values are *equal* or if one is *greater than* the other. The result of such an operation is either **TRUE** or **FALSE** (i.e., a **Boolean value**). The ***relational*** operators in C++ are:

Operator	Description
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

There are some examples:

```
(7 == 5)      // evaluates to FALSE
(5 > 4)       // evaluates to TRUE
(3 != 2)      // evaluates to TRUE
(6 >= 6)      // evaluates to TRUE
(5 < 5)       // evaluates to FALSE
```

It's **not** just numeric **constants** that can be compared, but just any value, including, of course, **variables**.

Suppose that **a=2, b=3** and **c=6**, then:

```
(a == 5)      // evaluates to FALSE, since a is not equal to 5
(a*b >= c)    // evaluates to TRUE, since (2*3 >= 6) is TRUE
(b+4 > a*c)   // evaluates to FALSE, since (3+4 > 2*6) is FALSE
((b=2) == a) // evaluates to TRUE
```

Be careful! The assignment operator (operator **=**, with *one equal* sign) is **not** the same as the equality comparison operator (operator **==**, with *two* equal signs); the **first** one **(=)** *assigns* the *value* on the **right-hand** to the *variable* on its **left**, while the other (**==**) compares

whether the values on both sides of the **operator** are **equal**. Therefore, in the last expression **`((b=2) == a)`**, we **first** assigned the value **2** to **`b`** and then we **compared** it to **`a`** (that also stores the value **2**), yielding **TRUE**.

Logical operators:

The operator **`!`** is the **C++** operator for the Boolean operation **NOT**. It has only one **operand**, to its *right*, and inverts it, producing **FALSE** if its operand is **TRUE**, and **True** if its operand is **FALSE**. Basically, it returns the *opposite* Boolean value of evaluating its operand. ***For example***:

```
!(5 == 5)   // evaluates to false because the expression at its right (5 == 5) is TRUE
!(6 <= 4)   // evaluates to true because (6 <= 4) would be FALSE
!true       // evaluates to FALSE
!false      // evaluates to TRUE
```

The **logical** operators **`&&`** and **`||`** are used when evaluating **two** expressions to obtain a single relational result. The operator **`&&`** corresponds to the Boolean logical operation **AND**, which yields **TRUE** if both its operands are **TRUE**, and **FALSE** otherwise. The following panel shows the result of operator **`&&`** evaluating the expression **`a&&b`**:

&& OPERATOR (and)		
a	b	a && b
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

The operator **`||`** corresponds to the **Boolean** logical operation **OR**, which yields **TRUE** if either of its operands is **TRUE**, thus being **FALSE** only when both operands are **FALSE**. Here are the possible results of **`a||b`**:

|| OPERATOR (or)		
a	b	a || b
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

For example:

```
((5 == 5) && (3 > 6))  // evaluates to FALSE ( TRUE && FALSE )
((5 == 5) || (3 > 6))  // evaluates to TRUE ( TRUE || FALSE )
```

When using the **logical** operators, C++ only evaluates what is necessary from ***left*** to ***right*** to come up with the combined relational result, ignoring the rest. Therefore, in the *last example* `((5==5)||(3>6))`, C++ evaluates ***first*** whether `5==5` is **TRUE**, and if so, it never checks whether `3>6` is **TRUE** or *not*. This is known as *short-circuit* evaluation, and works like this for these operators:

Operator	Short-Circuit
`&&`	if the *left-hand* side expression is **FALSE**, the combined result is **FALSE** (the *right-hand* side expression is never evaluated).
`||`	if the *left-hand* side expression is **TRUE**, the combined result is **TRUE** (the *right-hand* side expression is never evaluated).

This is mostly important when the *right-hand* expression has side effects, such as *altering* values:

Conditional ternary operator `(?)` :

The **conditional** operator evaluates an expression, *returning* one value if that expression evaluates to **TRUE**, and a different one if the expression evaluates as **FALSE**. Its syntax is:

```
condition? result1:result2
```

If condition is **TRUE**, the entire expression evaluates to **`result1`**, and otherwise to **`result2`**.

```
7==5 ? 4 : 3       // evaluates to 3, since 7 is not equal to 5.
7==5+2 ? 4 : 3     // evaluates to 4, since 7 is equal to 5+2.
5>3 ? a : b        // evaluates to the value of a, since 5 is greater than 3.
a>b ? a : b        // evaluates to whichever is greater, a or b.
```

For example:

```
// conditional operator
#include <iostream>
using namespace std;

int main ()
{
  int a,b,c;
  a=2; b=7;

  c = (a>b) ? a : b;
  cout << c << '\n';
}
```



In this example, **a** was **2**, and **b** was **7**, so the expression being evaluated (**a>b**) was not **true**, thus the **first** value specified after the question mark was *discarded* in favor of the **second** value (the one after the colon) which was **b** (with a value of **7**).

Comma operator (,)

The comma operator (**,**) is used to *separate* **two** or more expressions that are included where only one expression is *expected*. When the set of expressions has to be evaluated for a value, only the **right-most** expression is ***considered***. ***For example***, the following code:

```
a = (b=3, b+2);
```

would **first** assign the value **3** to **b**, and then assign **b+2** to variable **a**. So, at the end, variable **a** would contain the value **5** while variable b would contain value **3**.

Explicit type casting operator

Type casting operators allow to **convert** a value of a *given* type to *another* type. There are *several* ways to do this in **C++. The simplest** one, which has been **inherited** from the C **language**, is to precede the expression to be converted by the new type *enclosed* between *parentheses* **(())**:

```
int i;
float f = 3.14;
i = (int) f;
```

The previous code **converts** the floating-point number **3.14** to an *integer* value (**3**); the *remainder* is **lost**. Here, the typecasting operator was (**int**). *Another* way to do the same thing in **C++**.

```
i = int (f);
```

Precedence of operators

A single expression may have *multiple* operators. ***For example***:

```
x = 5 + 7 % 2;
```

In C++, the above expression always assigns **6** to variable **x**, because the **%** operator has a ***higher*** precedence than the **+** operator, and is always *evaluated* **before**. Parts of the expressions can be *enclosed* in ***parenthesis*** to override this precedence order, or to make explicitly clear the intended effect. Notice the difference:

```
x = 5 + (7 % 2);    // x = 6 (same as without parenthesis)
x = (5 + 7) % 2;    // x = 0
```

From **greatest** to **smallest** priority, C++ operators are evaluated in the following order:

Precedence	Operators	Description	Associativity
1	`::`	Scope resolution	Left to right
2	`++ -- () [] . ->`	Postfix increment/decrement, function call, array subscript, member access	Left to right
3	`++ -- + - ! ~ * & sizeof typeid`	Prefix increment/decrement, unary operators	Right to left
5	`* / %`	Multiplication, division, modulo	Left to right
6	`+ -`	Addition, subtraction	Left to right
7	`<< >>`	Bitwise shift operators	Left to right
8	`< <= > >=`	Relational operators	Left to right
9	`== !=`	Equality operators	Left to right
10	`&`	Bitwise AND	Left to right
11	`^`	Bitwise XOR	Left to right
12	`|`	Bitwise OR	Left to right
13	`&&`	Logical AND	Left to right
14	`||`	Logical OR	Left to right

When an expression has two operators with the *same precedence* level, grouping determines which one is evaluated first: either ***left-to-right*** or ***right-to-left***.

Example: Evaluating **operands** in **logical** expressions.

```
#include <iostream>
using namespace std;
int main()
{
cout << boolalpha; // Outputs boolean values as true or false
bool FT = false;
int y = 5;
FT = 7 || (y = 0);
cout << "Result of (7 || (y = 0)): " << FT << endl;
cout << "Value of y: " << y << endl;
int a, b, c;
a = b = c = 0;
FT = ++a || ++b && ++c;
cout << '\n'
<< " FT = " << FT
<< ", a = " << a
<< ", b = " << b
<< ", c = " << c << endl;
a = b = c = 0;
FT = ++a && ++b || ++c;
cout << " FT = " << FT
<< ", a = " << a
<< ", b = " << b
<< ", c = " << c << endl;
return 0;
}
```

```
Result of (7 || (y = 0)): true
Value of y: 5

 FT = true, a = 1, b = 0, c = 0
 FT = true, a = 1, b = 1, c = 0
```

Exercises

Exercise 1

What values do the following arithmetic expressions have?

a) `3/10`
b) `11%4`
c) `15/2.0`
d) `3 + 4 % 5`
e) `6 * 7 % 3`
f) `-7 % 4 * 3`

Exercise 2

The `int` variable `x and y` contains the number `7` and `0`. Calculate the value of the following logical expressions:

a) `x < 10 && x >= -1`
b) `!x && x >= 3`
c) `x++ == 8 || x == 7`
d) `x <= y || y >=0`
e) `x > -2 && y == 0`
f) `x && !y`
g) `!(x+1) || y - 1 > 0`

Flow control statements

A simple **C++** statement is each of the individual instructions of a program, like the variable declarations and expressions seen in previous sections. They always end with a semicolon (**;**), and are executed in the same order in which they appear in a program.

But programs are not limited to a ***linear sequence*** of statements. During its process, a program may ***repeat*** segments of code, or take ***decisions*** and ***bifurcate***. For that purpose, **C++** provides ***flow control*** statements that serve to specify what has to be done by our program, when, and under which circumstances.

Many of the flow control statements explained in this section require a generic (sub)statement as part of its syntax. This statement may either be a simple **C++** statement, - such as a single instruction, terminated with a semicolon (**;**) - or a *compound statement*. A compound statement is a group of statements (each of them terminated by its own semicolon), but all grouped together in a ***block***, *enclosed* in curly **braces**: **{}**:

```
{ statement1; statement2; statement3; }
```

The entire ***block*** is considered a single statement (composed itself of multiple sub statements). Whenever a generic statement is part of the syntax of a flow control statement, this can either be a simple statement or a compound statement.

Selection statements: `if` and `else`:

The **`if`** keyword is used to execute a ***statement*** or ***block***, if, and only if, a condition is *fulfilled*. Its syntax is:

```
if (condition) statement;
```

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.
For example, the following code fragment prints the message (**`x is 100`**), only if the value stored in the **`x`** variable is indeed **`100`**:

```
if (x == 100) cout << "x is 100";
```

If **`x`** is ***not*** exactly **`100`**, this statement is *ignored*, and nothing is printed. If you want to include more than a single statement to be executed when the condition is *fulfilled*, these statements shall be *enclosed* in **braces** ({}), forming a block:

```
if (x == 100)
{
   cout << "x is ";
   cout << x;
}
```

As usual, indentation and line breaks in the code have no effect, so the above code is equivalent to:

```
if (x == 100) { cout << "x is "; cout << x; }
```

Selection statements with **if** can also specify what happens when the condition is **not** fulfilled, by using the **else** keyword to introduce an alternative statement. Its *syntax* is:

```
if (condition) statement1 else statement2
```

where **statement1** is executed in case condition is ***true***, and in case it is ***not***, **statement2** is executed. **For example**:

```
if (x == 100) cout << "x is 100";
  else cout << "x is not 100";
```

This prints **x** is **100**, if indeed **x** has a value of **100**, but if it **not**, and only if it does not, it prints **x is not 100** instead.

Several **if + else** structures can be ***concatenated*** with the intention of checking a range of values. **For example**:

```
if (x > 0)
  cout << "x is positive"; else if (x < 0)
  cout << "x is negative"; else cout << "x is 0";
```

This prints whether **x** is ***positive***, ***negative***, or ***zero*** by ***concatenating*** two **ifelse** structures. Again, it would have also been possible to execute more than a single statement per case by grouping them into blocks enclosed in ***braces***: **{}**.

Example: **Check** Whether Number is **Even** or **Odd** using **if else**

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

int main()
{
  int n;

  cout << "Enter an integer: ";
  cin >> n;

  if ( n % 2 == 0)
    cout << n << " is even.";
  else
    cout << n << " is odd.";

  return 0;
}
```



```
Enter an integer: 20
20 is even.

Enter an integer: 123
123 is odd.
```

Hint: Replace `if else` statement using ***ternary*** operators:

```
(n % 2 == 0) ? cout << n << " is even." :  cout << n << " is odd.";
```

Example: Find **Largest** Number Using `if...else` Statement

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

int main()
{
    double n1, n2, n3;

    cout << "Enter three numbers: ";
    cin >> n1 >> n2 >> n3;
    if(n1 >= n2 && n1 >= n3)
        cout << "Largest number: " << n1;

    else if(n2 >= n1 && n2 >= n3)
        cout << "Largest number: " << n2;
    else
        cout << "Largest number: " << n3;
    return 0;
}
```



```
Enter three numbers: 12.235 11.005 3.94523
Largest number: 12.235

Enter three numbers: 17.025 21.235 21.632
Largest number: 21.632
```

Example: Find roots of a **Quadratic Equation** $ax^2 + bx + c = 0$

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

int main()
{

    float a, b, c, x1, x2, discriminant, realPart, imaginaryPart;
    cout << "Enter coefficients a, b and c: ";
    cin >> a >> b >> c;
    discriminant = b*b - 4*a*c;

    if (discriminant > 0)
{
        x1 = (-b + sqrt(discriminant)) / (2*a);
        x2 = (-b - sqrt(discriminant)) / (2*a);
        cout << "Roots are real and different." << endl;
        cout << "x1 = " << x1 << endl;
        cout << "x2 = " << x2 << endl;
    }

    else if (discriminant == 0) {
        cout << "Roots are real and same." << endl;
        x1 = -b/(2*a);
        cout << "x1 = x2 =" << x1 << endl;
    }

    else
{
        realPart = -b/(2*a);
        imaginaryPart =sqrt(-discriminant)/(2*a);
        cout << "Roots are complex and different."  << endl;
        cout << "x1 = " << realPart << "+" << imaginaryPart << "i" << endl;
        cout << "x2 = " << realPart << "-" << imaginaryPart << "i" << endl;
    }
    return 0;
}
```

Example: This program allows the user to input either a "**Y**" or a "**y**" for "**yes**":

```
#include <iostream>
using namespace std;
int main()
{
     char ans;
     cout << "Are you enrolled (y/n): ";
     cin >> ans;
      if (ans == 'Y' || ans == 'y')
            cout << "You are enrolled.\n";
      else
            cout << "You are not enrolled.\n";
}
```

```
Are you enrolled (y|n): N
You are not enrolled.
```

Example: This program requests the user's language and then prints a **greeting** in that language:

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

int main()
{
char language;
cout << "Engl., Fren., Ger., Ital., or Rus.? (e|f|g|i|r): ";
cin >> language;
if (language == 'e') cout << "Welcome to Programming Course.";
else if (language == 'f') cout << "Bon jour, Cours de programmation.";
else if (language == 'g') cout << "Guten tag, Programmierkurs.";
else if (language == 'i') cout << "Bon giorno, al corso di programmazione";
else if (language == 'r') cout << "Dobre utre, Kurs programmirovaniya.";
else cout << "Sorry; we don't speak your language.";
return 0
}
```

```
Engl., Fren., Ger., Ital., or Rus.? (e|f|g|i|r): i
Bon giorno, al corso di programmazione.
```

Exercises

Exercise 1

Convert **seconds** to *Hours*, *minutes* and *seconds*.

Exercise 2

Check **Leap** Year Using **if...else**

Exercise 3

Fine the value of **Y** using output statement "**The Y** = ".

$$Y = \begin{cases} 10x^2 + 35x - 20 & 1 \le x \le 3 \\ -5x + 10 & 3 < x \le 4 \\ -10 & x > 4 \end{cases}$$

Exercise 4

Find the circle equation $(x-h)^2 + (y-k)^2 = r^2$ passing through two points (x_1, y_1) , (x_2 , y_2) where $h = \frac{x_1+x_2}{2}$, $k = \frac{y_1+y_2}{2}$

Exercise 5

Find the **Middle** number of **three** Numbers.

Exercise 6

Check the **grade** of input **mark** using **else if** statement

Iteration statements (loops)

Loops repeat a statement a certain number of times, or while a condition is fulfilled. They are introduced by the keywords **while**, **do**, and **for**.

The while loop:

The simplest kind of loop is the while-loop. Its syntax is:

```
while (expression) statement
```

The **while-loop** simply ***repeats*** statement while expression is **true**. If, after any execution of statement, expression is no longer true, the loop ends, and the program continues right after the loop. ***For example***, let's have a look at a countdown using a while-loop:

```
// custom countdown using while
#include <iostream>
using namespace std;
int main ()
{
int n = 10;
while (n>0)
      {

      cout << n << ", ";
      --n;

      }
  cout << "liftoff!\n";
}
```

10, 9, 8, 7, 6, 5, 4, 3, 2, 1, liftoff!

The *first* statement in main sets **n** to a value of **10**. This is the first number in the *countdown*. Then the **while-loop** begins: **if** this value fulfills the condition **n>0** (that **n** is *greater than zero*), then the block that follows the condition is ***executed***, and ***repeated*** for as long as the condition (**n>0**) remains being true.

The whole process of the previous program can be interpreted according to the following script (beginning in main):

1) **n** is assigned a value
2) The **while** condition is checked (**n>0**). At this point there are **two** possibilities:
   condition is **true**: the statement is executed (to **step 3**)
   condition is **false**: ignore statement and continue after it (to **step 5**)
3) Execute statement: **cout << n << ", ";**

`--n;` (**prints** the value of **n** and decreases **n** by **1**)

4) **End** of block. Return automatically to **step 2**.
5) **Continue** the program right after the block: print **liftoff!** and end the program.

A thing to consider with **while-loops** is that the loop should ***end*** at some point, and thus the statement shall alter values checked in the condition in some way, so as to force it to become *false* at some point. Otherwise, the loop will continue looping *forever*. In this case, the loop includes **--n**, that *decreases* the value of the variable that is being evaluated in the condition (**n**) by one - this will eventually make the condition (**n>0**) *false* after a certain number of loop iterations. To be more specific, after **10** iterations, **n** becomes **0**, making the condition no longer true, and ending the while-loop.

The do-while loop

A very similar loop is the **do-while** loop, whose syntax is:

```
do statement while (condition);
```

It behaves like a **while-loop**, except that ***condition*** is evaluated *after* the execution of statement instead of *before*, *guaranteeing* at least *one execution* of statement, even if condition is ***never*** fulfilled. ***For example***, the following example program *echoes* any text the user introduces until the user *enters* **goodbye**:

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

int main ()
{
  string str;
   do {
        cout << "Enter text: ";
        getline (cin,str);
        cout << "You entered: " << str << '\n';
        }
   while (str != "goodbye");
}
```

```
Enter text: hello
You entered: hello
Enter text: who's there?
You entered: who's there?
Enter text: goodbye
You entered: goodbye
```

The **do-while** loop is usually *preferred* over a **while-loop** when the statement needs to be executed at least **once**, such as when the condition that is checked to end of the loop is determined within the loop statement itself. In the previous example, the user input within the block is what will determine if the loop *ends*. And thus, even if the user wants to end the loop as soon as possible by entering **goodbye**, the block in the loop needs to be executed at least once to prompt for input, and the condition can, in fact, only be determined after it is executed.

The `for` loop

The for loop is designed to iterate a number of times. Its syntax is:

```
for (initialization; condition; increase) statement;
```

Like the **while-loop**, this loop **repeats** statement while condition is ***true***. But, in addition, the **for loop** provides specific locations to contain an *initialization* and an *increase* expression, executed *before* the loop begins the *first time*, and after each iteration, respectively. Therefore, it is especially useful to use ***counter*** variables as condition.

It works in the following way:

1) **initialization** is executed. Generally, this *declares* a *counter* variable, and sets it to some *initial* value. This is executed a ***single time***, at the beginning of the loop.
2) condition is ***checked***. If it is ***true***, the loop *continues*; otherwise, the loop *ends*, and statement is *skipped*, going directly to **step 5**.
3) statement is *executed*. As usual, it can be either a *single statement* or a *block* enclosed in curly braces { }.
4) **increase** is executed, and the loop gets back to **step 2**.
5) the loop ***ends***: execution *continues* by the next statement after it.

Here is the countdown ***example*** using a **for loop**:

```
// countdown using a for loop
#include <iostream>
using namespace std;

int main ()
{
     for (int n=10; n>0; n--)
     {
     cout << n << ", ";
     }
 cout << "liftoff!\n";
}
```

10, 9, 8, 7, 6, 5, 4, 3, 2, 1, liftoff!

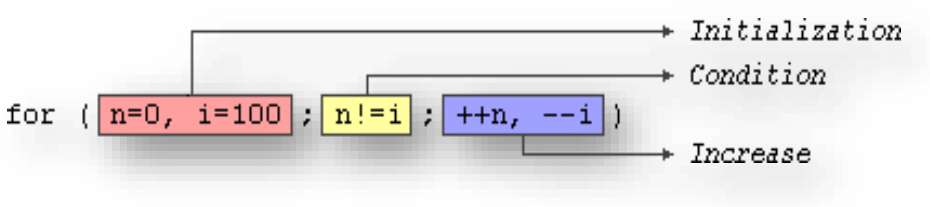
The **three** fields in a **for-loop** are *optional*. They can be left empty, but in all cases the ***semicolon*** signs between them are *required*. ***For example***,

- `for (;n<10;)` is a ***loop*** without *initialization* or *increase* (equivalent to a **while-loop**)
- `for (;n<10;++n)` is a ***loop*** with *increase*, but **no** *initialization* (maybe because the *variable* was already *initialized* ***before*** the loop). A loop with no condition is equivalent to a ***loop*** with ***true*** as condition (i.e., an ***infinite loop***).

Because *each* of the fields is executed in a particular time in the life cycle of a loop, it may be useful to execute more than a ***single expression*** as any of *initialization*, *condition*, or *statement*. Unfortunately, these are not statements, but rather, *simple expressions*, and thus cannot be **replaced** by a *block*. As expressions, they can, however, make use of the *comma* operator (`,`): This operator is an expression *separator*, and can separate *multiple* expressions where only one is generally expected. ***For example***, using it, it would be possible for a **for loop** to handle ***two counter*** variables, *initializing* and *increasing* both:

```
for ( n=0, i=100 ; n!=i ; ++n, --i )
{
   // whatever here...
}
```

This loop will execute ***50 times*** if neither `n` or `i` are *modified* within the loop:



`n` starts with a value of **0**, and `i` with **100**, the condition is `n!=i` (i.e., that `n` is *not equal* to `i`). Because `n` is *increased* by ***one***, and `i` *decreased* by ***one*** on each *iteration*, the loop's condition will become ***false*** after the ***50th*** iteration, when both `n` and `i` are *equal* to **50**.

Example: Sum of Natural Numbers (N) using `for` **loop**

$$\sum_{i=0}^{n} i$$

```
#include <iostream>
using namespace std;
int main()
{
    int n, sum = 0;
    cout << "Enter a positive integer: ";
    cin >> n;

    for (int i = 1; i <= n; ++i) {
        sum += i;
    }

    cout << "Sum = " << sum;
    return 0;
}
```



```
Enter a positive integer: 10
Sum = 55
```

Notice: **Rewrite** the program using **`do-while`** or **`while`** loop.

Example: Find the **Factorial** of N (**N!**)

```
#include <iostream>
using namespace std;
int main()
{
    int n;
    long factorial = 1;
    cout << "Enter a positive integer: ";
    cin >> n;

    if (n < 0)
        cout << "Error! Factorial of a negative number doesn't exist.";
    else
   {
        for(int i = 1; i <= n; ++i)
        {
            factorial *= i;
   }
        }
cout << "Factorial of " << n << "! = " << factorial;
    }
```

```
Enter a positive integer: 10
Factorial of 10! = 3628800
Enter a positive integer: -6
Error! Factorial of a negative number doesn't exist.
```

Example: Program to Generate **Fibonacci** Sequence Up to a Certain Number

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

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

int main()
{
    int a = 0, b = 1, c = 0, n;

    cout << "Enter a positive number: ";
    cin >> n;
    cout << "Fibonacci Series: " << a << ", " << b;

    c = a + b;

    while(c <= n)
    {
        cout << ", "<< c;
        a = b;
        b = c;
        c = a + b;
    }
    return 0;
}
```

```
Enter a positive number: 100
Fibonacci Series: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89
```

Example: Display **Multiplication Table** up to **10**

```
#include <iostream>

using namespace std;
int main()
{
    int n;

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

    for (int i = 1; i <= 10; ++i) {
        cout << n << " * " << i << " = " << n * i << endl;
    }
    return 0;
}
```

```
Enter a positive integer: 10
10 * 1 = 10
10 * 2 = 20
10 * 3 = 30
10 * 4 = 40
10 * 5 = 50
10 * 6 = 60
10 * 7 = 70
10 * 8 = 80
10 * 9 = 90
10 * 10 = 100
```

Example: C++ Program to ***Reverse*** an Integer.

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

int main() {

  int n, reversed_number = 0, remainder;

  cout << "Enter an integer: ";
  cin >> n;

  while(n != 0)

  {
    remainder = n % 10;
    reversed_number = reversed_number * 10 + remainder;
    n /= 10;
  }

  cout << "Reversed Number = " << reversed_number;

  return 0;
}
```



```
Enter an integer: 12345
Reversed Number = 54321
```

Example: Compute ***Power*** X^y Manually.

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

int main()
{
    int exponent;
    float base, result = 1;
    cout << "Enter base and exponent respectively:  ";
    cin >> base >> exponent;

    cout << base << "^" << exponent << " = ";

    while (exponent != 0)
    {
        result *= base;
        --exponent;
    }
    cout << result;

    return 0;
}
```



```
Enter base and exponent respectively:  5 6
5^6 = 15625
```

Example: Using a `do..while` Loop to Compute the product

$$\prod_{i=1}^{n} i$$

```
#include <iostream>
using namespace std;
int main()
{
 int n, i=1;
 cout << "Enter a positive integer: ";
 cin >> n;
 long Prd=1;
  do
  Prd *= i++;
  while (i <= n);
 cout << "The Prod of Number (" << n << ") is = " << Prd;
}
```

```
Enter a positive integer: 9
The Prod of Number (9) is = 362880
```

Nested Loops:

A **nested** loop means a loop statement inside another loop statement. That is why nested loops are also called "***loop inside loops*** ". Nested `for` loop refers to any type of loop that is defined inside a '`for`' loop. The syntax:

```
for ( initialization; condition; increment ) {

   for ( initialization; condition; increment ) {

       statement(s) of inside loop
   }

     Statement(s) of outer loop
}
```

For example, this program prints a ***multiplication*** table:

```
#include <iomanip>
#include <iostream>
using namespace std;
int main()
{ for (int x=1; x <= 10; x++)
  { for (int y=1; y <= 10; y++)
     cout << setw(4) << x*y;
     cout << endl;
  }
}
```

```
   1   2   3   4   5   6   7   8   9  10
   2   4   6   8  10  12  14  16  18  20
   3   6   9  12  15  18  21  24  27  30
   4   8  12  16  20  24  28  32  36  40
   5  10  15  20  25  30  35  40  45  50
   6  12  18  24  30  36  42  48  54  60
   7  14  21  28  35  42  49  56  63  70
   8  16  24  32  40  48  56  64  72  80
   9  18  27  36  45  54  63  72  81  90
  10  20  30  40  50  60  70  80  90 100
```

Also, using nested **while** loop refers to any type of loop that is defined inside a '**while** loop. The syntax:

```
while(condition) {

   while(condition) {

      statement(s) of inside loop
   }

   Statement(s) of outer loop
}
```

For example: Print a Pattern

```
1
1 2
1 2 3
……. n
```

```
#include <iostream>
using namespace std;
int main()
{
    int n;
    cout<< "Enter n value:" ;
    cin>>n;
    int i = 1;

    while (i <= n) {
        int j = 1;
        while (j <= i) {
            cout<<j<<" ";
            j++;
        }
        i++;
        cout<<endl;
    }
    return 0;
}
```



```
Enter n value:7
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
1 2 3 4 5 6 7
```

And using nested **do-while** loop refers to any type of loop that is defined inside a **do-while** loop. The syntax:

```
do{

   do{

      statement(s) of inside loop
   }while(condition);

   Statement(s) of outer loop
}while(condition);
```

Note: There is **no** rule that a ***loop*** must be nested inside its own **type**. In fact, there can be *any* type of loop nested inside *any* type and to *any* level. such as:

```
do{

   while(condition) {

      for ( initialization; condition; increment ) {

          statement(s) of inside for loop
      }

       statement(s) of inside while loop
   }

    statement(s) of outer do-while loop

}while(condition);
```

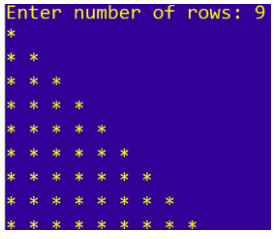
Exercises

Exercise 1: Generate **Fibonacci** Series with nth. Terms.

Exercise 2: Check **Palindrome** Number, such as (*11, 262, 9009, 87078…*)

Exercise 3: Display all **Factors** of a Number

Exercise 4: Program to Print a ***Half-Pyramid*** of "*"



Exercise 5:
Find the sine of angle (x in radian), solve until the difference of summation for last two terms is ≤ ***0.00001***

$$\boldsymbol{\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots = \sum_{n=0}^{\infty} \frac{(-1)^n\, x^{2n+1}}{(2n+1)!}}$$

Jump statements

Jump statements allow altering the flow of a program by performing jumps to **specific locations**.

The break statement:

break *leaves* a loop, even if the condition for its end is not fulfilled. It can be used to **end** an *infinite* loop, or to *force* it to end before its natural end. ***For example***, let's stop the countdown before its natural end:

```
// break loop example
 #include <iostream>
 using namespace std;
 int main ()
 {
   for (int n=10; n>0; n--)
   {
      cout << n << ", ";
     if (n==3)
     {
      cout << "countdown aborted!";
      break;
     }
   }
 }
```



10, 9, 8, 7, 6, 5, 4, 3, countdown aborted!

Example: Reads a *sequence* of *positive* integers, **terminated** by 0, and find their **average**:

```
#include <iostream>
using namespace std;
int main()
{
 int n, count=0, sum=0;
       for (;;)  // "forever"
       {
       cout << "read a Positive Number (0 to Quit) :";
       cin >> n;
       if (n <= 0) break;
       ++count;
       sum += n;
       }
cout << "\n The average of those " << count << " positive numbers is "
<< float(sum)/count << endl;
}
```

```
read a Positive Number (0 to Quit) :10
read a Positive Number (0 to Quit) :22
read a Positive Number (0 to Quit) :31
read a Positive Number (0 to Quit) :17
read a Positive Number (0 to Quit) :41
read a Positive Number (0 to Quit) :-11

 The average of those 5 positive numbers is 24.2
```

The continue statement:

The **continue** statement causes the program to ***skip*** the rest of the loop in the *current* iteration, as if the end of the statement ***block*** had been reached, causing it to *jump* to the *start* of the *following* iteration. ***For example***, let's **skip** number **5** in our countdown:

```
// continue loop example
#include <iostream>
using namespace std;
int main ()
{
  for (int n=10; n>0; n--)
     {
     if (n==5) continue;
     cout << n << ", ";
      }
  cout << "liftoff!\n";
}
```



```
10, 9, 8, 7, 6, 4, 3, 2, 1, liftoff!
```

Example: Using **continue** statement to **pass** all even numbers between **1** to **25**

```
#include <iostream>
using namespace std;
int main()
{
for( int i = 0; i < 25; i++ )
     {
     if( i % 2 == 0 )
     continue;
     cout<<i<<",";
     }
cout<<" Done ";

}
```



1,3,5,7,9,11,13,15,17,19,21,23, Done

Selecting with switch statement:

The syntax of the **switch** statement is a bit peculiar. Its purpose is to *check* for a *value* among a number of possible constant expressions. It is something *similar* to concatenating **if-else** statements, but **limited** to constant expressions. Its most typical syntax is:

```
switch (expression)
{
case constant1: statementList1;
case constant2: statementList2;
case constant3: statementList3;
:
:
case constantN: statementListN;
default: statementList0;
}
```

This evaluates the expression and then looks for its value among the **case constants**.
If the value is found among the constants listed, then the statements in the corresponding **statementList** are executed. Otherwise, if there is a **default** (which is *optional*), then the program branches to its **statementList**. The *expression* must evaluate to an ***integral*** type and the ***constants*** must be ***integral*** constants. ***For example,*** using a **switch** Statement to ***Select*** a Range of Scores.

```
#include <iostream>
using namespace std;
int main()
{
int score;
cout << "Enter your test score: "; cin >> score;
switch (score/10)
     {
     case 10:
     case 9: cout << "Your grade is an A." << endl; break;
     case 8: cout << "Your grade is a B." << endl; break;
     case 7: cout << "Your grade is a C." << endl; break;
     case 6: cout << "Your grade is a D." << endl; break;
     case 5: cout << "Your grade is a E." << endl; break;
     case 4:
     case 3:
     case 2:
     case 1:
     case 0: cout << "Your grade is an F." << endl; break;
     default: cout << "Error: score is invalid.\n";
     }
cout << "Goodbye." << endl;
}
```



```
Enter your test score: 83
Your grade is a B.
Goodbye.
```

Exercises

Exercise 1: Write and run a program that simulates a **simple calculator**. It reads **two integers** and a **character**. If the character is a **+**, the ***sum*** is *printed*; if it is a **–**, the ***difference*** is *printed*; if it is a *****, the ***product*** is *printed*; if it is a **/**, the ***quotient*** is *printed*; and if it is a **%**, the ***remainder*** is *printed*. Use a `switch` statement.

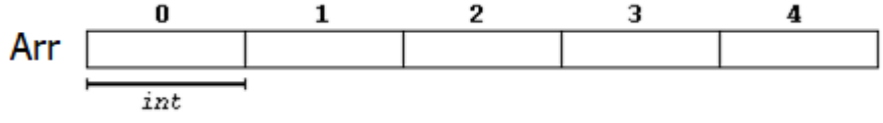
Exercise 2: Write a program where a user has only 3 attempts to enter the correct password, such as "**1234**"

Exercise 3: Write a program to **pass** all numbers range between **2** to **79** that are **not** divided by 3 or 5, and **count** the numbers that been **passed**.

Arrays

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

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



where each *blank* panel represents an element of the array. In this case, these are values of type **`int`**. These elements are numbered from **`0`** to **`4`**, being **0** the first and **4** the last;

In **C++**, the ***first*** element in an ***array*** is always numbered with a ***zero*** (*not a one*), no matter its length. Like a regular variable, an array ***must*** be *declared before* it is ***used***. A typical declaration for an array in C++ is:

```
type name [elements];  such as  int r[7];
```

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

Therefore, the **`Arr`** array, with **five** elements of type int, can be *declared* as: **`int Arr[5];`**

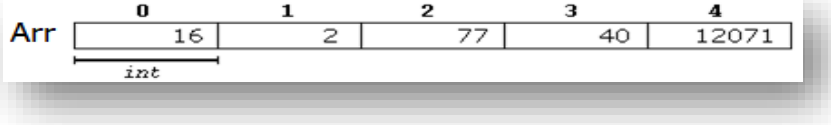
NOTE: The elements field within square brackets **`[]`**, representing the number of elements in the array, must be a ***constant*** expression, since arrays are blocks of ***static*** memory whose size must be determined at compile time, before the program runs.

Initializing arrays:

By default, regular arrays of local scope (*for example*, those declared ***within*** a function) are left uninitialized. This means that none of its elements are set to any particular value; their contents are undetermined at the point the array is declared. But the elements in an array can be *explicitly* initialized to specific values when it is declared, by enclosing those initial values in *braces* **`{}`**. ***For example:***

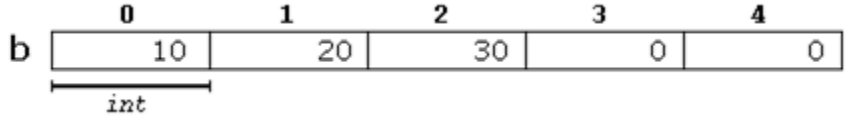
```
int Arr[5] = { 16, 2, 77, 40, 12071 };
```

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



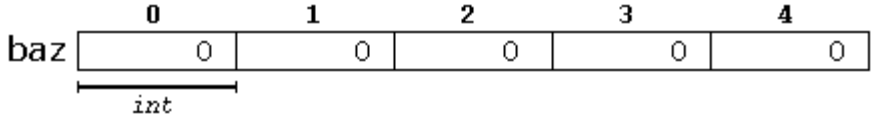
The number of values between braces `{}` shall not be ***greater*** than the number of elements in the array. ***For example***, in the example before, `Arr` was declared having **5** elements (as specified by the number enclosed in square brackets, `[]`), and the braces `{}` contained *exactly* **5**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

```
int b[5] = { 10, 20, 30 };
```
Will create an array like this:



The *initializer* can even have *no values*, just the ***braces***: `int baz[5] = {};`

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



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

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

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

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

Example: Display the array **R** with **5** integer values:

```
#include <iostream>
using namespace std;
int main()
{
    int R[5] = {10, 20, 30, 40, 50};
    cout << "The values in array R are:" << endl;
    for(int i = 0; i < 5; i++)
    {
        cout << R[i] << endl;
    }
    return 0;
```

```
The values in array R are:
10
20
30
40
50
```

Or display the array elements ***horizontally***: `cout << R[i] <<'\t';`

```
The values in array R are:
10      20      30      40      50
```

Example: Generate an array **Ar** with **10** ***random integer*** value its range ***{-10 → 10}***

Hint: Using `rand()`,`srand()` and `time()` functions to generate ***random*** value.
To get the value in ***range***, using the equation:

`Ar[i]=(rand()%(max-min+1)) + min`

Where `Ar[i]` is array element, `max` is maximum range and `min` is minimum range.
So the equation become: `Ar[i] = (rand() % 21) - 10;`

```
#include <iostream>
#include <cstdlib>      // For rand() and srand()
#include <ctime>        // For time()
using namespace std;

int main()
{
  int Ar[10];
  srand(time(0));
  for (int i = 0; i < 10; i++)
  {
    Ar[i] = (rand() % 21) - 10;
    cout << "Ar[" << i << "] = " << Ar[i] << endl;
  }
}
```

```
Ar[0] = -1
Ar[1] = -5
Ar[2] = -4
Ar[3] = -8
Ar[4] = 4
Ar[5] = -10
Ar[6] = -6
Ar[7] = 7
Ar[8] = 6
Ar[9] = -5
```

Example: Generate an array **ODD** with **50** odd values from ***1*** to ***100*** and find the ***sum*** and ***average*** of them.

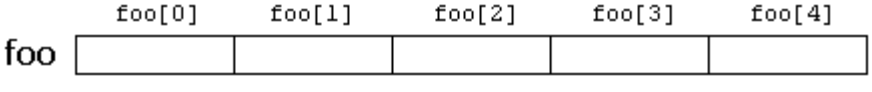
```
#include <iostream>
using namespace std;
int main()
{
    int ODD[50],count=0, sum=0;

    for(int i = 1; i < 100; i+=2)
    {
        ODD[count]=i;
        cout << ODD[count] <<'\t';
        sum+= ODD[count];
        count++;
     }
float avr=sum/count;
cout<<"\n\n the sum of odd No. form (1-100) =" << sum;
cout<<"\n and The Average is =" << avr;

return 0;
}
```

```
1       3       5       7       9       11      13      15      17      19
        21      23      25      27      29      31      33      35      37
        39      41      43      45      47      49      51      53      55
        57      59      61      63      65      67      69      71      73
        75      77      79      81      83      85      87      89      91
        93      95      97      99

 the sum of odd No. form (1-100) =2500
 and The Average is =50
```

Accessing the values of an array:

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

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



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

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

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

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

Therefore, the expression **foo[2]** is itself a variable of type **int**.

It is important to be able to clearly *distinguish* between the *two* uses that ***brackets*** **[]** have related to arrays. They perform two different tasks: **one** is to specify the ***size*** of arrays when they are *declared*; and the **second** one is to specify ***indices*** for concrete array elements when they are accessed. Such as:

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

Some other valid operations with arrays:

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

Example: Find the ***R*** for **x** values which are ***{1, -3, 7, 2, -4, 5, 9, -8}***

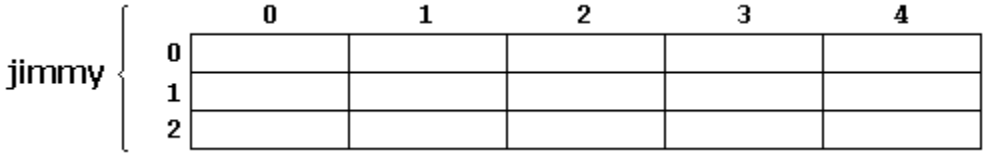
$$R = \frac{1}{n}\sum_{i=0}^{n}(x_i^2 + 1)$$

```
#include <iostream>
using namespace std;
int main()
{
    int x[]={1, -3, 7,  2, -4, 5, 9, -8};
    int n,sum=0;
    for(n=0 ; n<8 ;n++)
    {
        sum+=(x[n]*x[n]+1);
    }
    float R=float(sum)/n;
    cout<<"The R value = "<<R;
    return 0;
}
```

```
The R value = 32.125
```

Multidimensional arrays

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

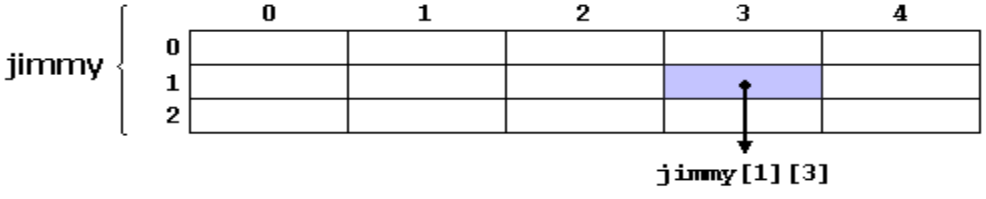


jimmy represents a bidimensional array of **3** per **5** elements of type int. The **C++** syntax for this is:

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

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

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



(***remember*** that array indices always begin with ***zero***).

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

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

Declares an array with an element of type **char** for each second in a century. This amounts to more than **3 billion char!**. So, this declaration would consume more than **3 gigabytes** of memory! At the end, multidimensional arrays are just an abstraction for programmers, since the same results can be achieved with a simple array, by multiplying its indices:

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

Notice numerical array with ***one****-dimension* called **vector,** and **matrix** is a numerical ***two****-dimensional* array. Matrices are specifically designed for ***linear algebra*** and have a well-defined set of mathematical operations, such as: (addition, subtraction, multiplication, transpose, determinant, inverse and so on).

For example, display the **jimmy** (**3 X 5**) array its values range {1→15} as shown in figure:

jimmy

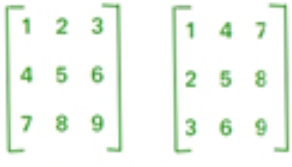
	0	1	2	3	4
0	1	2	3	4	5
1	2	4	6	8	10
2	3	6	9	12	15

```
#include <iostream>
using namespace std;
#define WIDTH 5
#define HEIGHT 3

int main ()
{
int jimmy [HEIGHT][WIDTH];
int n,m;
  for (n=0; n<HEIGHT; n++)
  {
  for (m=0; m<WIDTH; m++)
    {
    jimmy[n][m]=(n+1)*(m+1);
    cout<<jimmy[n][m]<<'\t';
    }
    cout<<endl;
  }
return 0
}
```



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

Example: **Read** and **display** Matrix **M** and find its **transpose** (M^T) such as



Input Output

```
#include <iostream>
using namespace std;
int main()
{
     int M[3][3],T[3][3],n=1;
     cout << "the Input Matrix" << endl;
     for (int i=0;i<3;i++){
          for(int j=0;j<3;j++){
               M[i][j]=n;
               cout<< M[i][j] << '\t';
               n++;
          }
          cout << endl;
     }
```

```
cout<< "the Transpose Matrix" <<endl;
      for (int i=0;i<3;i++){
            for(int j=0;j<3;j++){
                  T[i][j]=M[j][i];
                  cout << T[i][j] <<'\t';
            }
            cout<<endl;
      }
      return 0;
}
```

```
the Input Matrix
1       2       3
4       5       6
7       8       9
the Transpose Matrix
1       4       7
2       5       8
3       6       9
```

There are many **matrices** using in mathematics, such as (**Identity**, **UpperTri**, **LowerTri**, **Zeros**, **Ones**, and so on) as shown. And other matrices base on **formula** to generate it like (**Pascal**, **Hankel**, and so on)

Unit matrix $[I]=\begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix}$ $a_{ij}=1 \quad for\ i=j$ $a_{ij}=0 \quad for\ i\neq j$	*Upper triangular matrix* $\begin{bmatrix}a_{11}&a_{12}&a_{13}\\0&a_{22}&a_{23}\\0&0&a_{33}\end{bmatrix}$ $a_{ij}=0 \quad for\ i<j$	*Lower triangular matrix* $\begin{bmatrix}a_{11}&0&0\\a_{21}&a_{22}&0\\a_{31}&a_{32}&a_{33}\end{bmatrix}$ $a_{ij}=0 \quad for\ i>j$
Diagonal matrix $\begin{bmatrix}a_{11}&0&0\\0&a_{22}&0\\0&0&a_{33}\end{bmatrix}$ $a_{ij}=0 \quad for\ i\neq j$	*Symetric matrix* $\begin{bmatrix}a_{11}&b&c\\b&a_{22}&d\\c&d&a_{33}\end{bmatrix}$ $a_{ij}=a_{ji}$	*Anti symetric matrix* $\begin{bmatrix}0&-b&-c\\b&0&d\\c&-d&0\end{bmatrix}$ $a_{ii}=0$ $a_{ij}=-a_{ji}$
Square matrix $A=\begin{bmatrix}a_{11}&a_{12}\\a_{21}&a_{22}\end{bmatrix}$ *number of rows equal to the number of culomns* $A(n\times n)$.	*General size matrix* $\begin{bmatrix}a_{11}&a_{12}&a_{13}&a_{14}\\a_{21}&a_{22}&a_{23}&a_{24}\end{bmatrix}$ *A is a* (2×4) *matrix*	

Example: Generate the following matrices (NXN)

1) **Identity Matrix I**

```
int main()
{
     int I[100][100],n; // لحجز حيز بالذاكره للمصفوفة مؤقت ويكون كبير
     cout<<"Enter size of Matrix (n):";
     cin>>n;
     cout<< "\n The Identity Matrix " <<endl;
     for (int i=0;i<n;i++){
          for(int j=0;j<n;j++){
               I[i][j]=(i==j)? 1:0;
               cout << I[i][j] << '\t';
          }
          cout<<endl;
     }
     return 0;
}
```

```
Enter size of Matrix (n):6

 The Identity Matrix
1       0       0       0       0       0
0       1       0       0       0       0
0       0       1       0       0       0
0       0       0       1       0       0
0       0       0       0       1       0
0       0       0       0       0       1
```

2) **Upper Triangle Matrix** for $U = \begin{bmatrix} 2 & -1 & 3 & 4 \\ 6 & 8 & -5 & 7 \\ 1 & -2 & -3 & 5 \\ 9 & -5 & 10 & -8 \end{bmatrix}$

```
int main()
{
int U[4][4] = { { 2,-1,3,4}, {6,8,-5,7}, { 1,-2,-3,5}, {9,-5,10,-8} }

cout<< "\n The U Matrix " <<endl;
     for (int i=0;i<4;i++){
          for(int j=0;j<4;j++){
               cout<< setw(3) << U[i][j] <<'\t';
          }
          cout<<endl;
     }

cout<< "\n The Upper Triangle Matrix " <<endl;
for (int i=0;i<4;i++){
          for(int j=0;j<4;j++){
               U[i][j]=(i<=j)? U[i][j]:0;
               cout<< setw(3)<< U[i][j] << '\t';
```

```
The U Matrix
 2      -1       3       4
 6       8      -5       7
 1      -2      -3       5
 9      -5      10      -8

The Upper Triangle Matrix
 2      -1       3       4
 0       8      -5       7
 0       0      -3       5
 0       0       0      -8
```

```
            }
            cout<<endl;
      }
      return 0;
}
```

3) $A_{ij} = \frac{i+1}{j+2}$

```
int main()
{
float A[100][100];
      cout<< "Enter size of Matrix (n):";
      cin>>n;
      cout<< "\n The Matrix (A)" <<endl;
      for (int i=0;i<n;i++){
            for(int j=0;j<n;j++){
               A[i][j]= (i+1.0)/(j+2.0);
               cout<<fixed<<setw(4)<< setprecision(2)<<A[i][j]<<'\t';
            }
            cout<<endl;
      }
      return 0;
}
```

```
Enter size of Matrix (n):5

 The Matrix (A)
0.50    0.33    0.25    0.20    0.17
1.00    0.67    0.50    0.40    0.33
1.50    1.00    0.75    0.60    0.50
2.00    1.33    1.00    0.80    0.67
2.50    1.67    1.25    1.00    0.83
```

Exercises

Exercise 1: Generate an array ***R*** with ***20*** *random integer* values range {***1→100***} and find the ***maximum*** value and its ***position*** within the array.

Exercise 2: Sort the array **A** with values ***(55, 23, 99, 61, 48, 85, 30, 77, 12)*** using the ***Bubble Sort algorithm***.

Exercise 3: Find the ***standard deviation*** of a data set of ***n*** numbers $x_0, \ldots, x_{n-1}$ defined by the formula: for given sample of ***x*** = [1, 9,3,7,0,2,8,5,4]

$$s = \sqrt{\frac{\sum_{i=0}^{n-1} (x_i - \bar{x})^2}{n-1}}$$

Where $\bar{x}$ is the ***mean*** of the data. This formula says: *square* each deviation ***(x[i] - mean)***; sum those squares; divide that *square root* by ***n-1***; take the *square root* of that sum.

Exercise 4: Generate (**N X N**) matrix for the following:

$$H\begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix} \quad B\begin{bmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \end{bmatrix} \quad D\begin{bmatrix} 1 & 1 & \ldots & N \\ 2 & 2 & \ldots & N \\ 3 & 3 & \ldots & N \\ \vdots & \vdots & \ldots & N \\ N & N & \ldots & N \end{bmatrix}$$

Exercise 5: Generate a matrix **K**(**N X M**) with *random* values range ***{-20 →30}*** display it, then change the ***negative*** values to ***0*** and ***positive*** values to ***1***

Exercise 6: Generate a matrix **T**(**N X M**) with *random* values range ***{0→50}*** and find the **sum** of matrix and the **average**.

Functions

These are the functions other than the **Standard Library Functions**. These are created by the users and the user has the flexibility to choose the **function name**, the statements that will be executed, the parameters that will be passed to the user & the return type of the function.

Functions allow to structure programs in segments of code to perform individual tasks. In **C++,** a function is a group of statements that is given a **name**, and which can be called from some point of the program. The most common syntax to define a function is:

```
type name ( parameter1, parameter2, ...) { statements }
```

Where:

- **type** is the type of the value returned by the function.
- **name** is the identifier by which the function can be **called**.
- **parameters** (as many as needed): Each parameter consists of a **type** followed by an identifier, with each parameter being *separated* from the next by a *comma*. Each parameter looks very much like a regular variable declaration (for example: **int x**), and in fact acts within the function as a regular variable which is *local* to the function. The purpose of parameters is to allow *passing arguments* to the function from the location where it is called from.
- **statements** is the function's *body*. It is a block of statements surrounded by *braces* **{ }** that specify what the function actually does.

Let's have a look at an example:

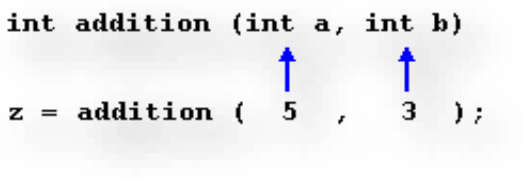
```
// function example
#include <iostream>
using namespace std;
int addition (int a, int b)
{
int r;
r=a+b;
return r;
}

int main ()
{
int z;
  z = addition (5,3);
  cout << "The result is " << z;
}
```



```
The result is 8
```

This program is *divided* in **two** functions: **addition** and **main**. Remember that no matter the order in which they are defined, a C++ program *always* starts by calling **main**. In fact, **main** is the only function called *automatically*, and the code in any other function is only executed if its function is called from main (directly or indirectly).

In the example above, **main** begins by declaring the variable **z** of type **int**, and right after that, it performs the first function call: it calls **addition**. The call to a function follows a structure very similar to its declaration. In the example above, the call to addition can be compared to its definition just a few lines earlier:

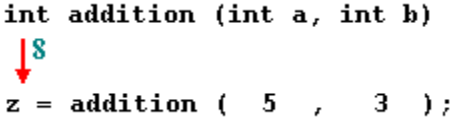


The parameters in the function declaration have a clear correspondence to the arguments passed in the function call. The call passes two values, **5** and **3**, to the function; these correspond to the parameters **a** and **b**, declared for function **addition**.

Then, inside **addition**, another **local** variable is declared (**int r**), and by means of the expression **r=a+b**, the result of a plus b is assigned to **r**; which, for this case, where **a** is **5** and **b** is **3**, means that **8** is assigned to **r**.

The *final* statement within the function: `return r;`

Ends function **addition**, and returns the control *back* to the point where the function was *called*; in this case: to function **main**. At this precise moment, the program *resumes* its course on main returning exactly at the same point at which it was interrupted by the call to addition. But additionally, because addition has a ***return*** type, the call is evaluated as having a value, and this value is the value specified in the return statement that ended addition: in this particular case, the value of the *local* variable **r**, which at the moment of the return statement had a value of **8**.



Therefore, the call to **addition** is an expression with the value returned by the function, and in this case, that value, **8**, is assigned to **z**. It is as if the entire function call (**addition(5,3)**) was *replaced* by the value it *returns* (i.e., **8**).

Then **main** simply prints this value by calling: `cout << "The result is " << z;`

A function can actually be called ***multiple*** times within a program, and its ***argument*** is naturally *not limited* just to literals:

```
// function example
#include <iostream>
using namespace std;
int subtraction (int a, int b)
{   int r;   r=a-b;   return r;  }

int main ()
{
int x=5, y=3, z;
z = subtraction (7,2);
cout << "The first result is " << z << '\n';
cout << "The second result is " << subtraction (7,2)<< '\n';
cout << "The third result is " << subtraction (x,y) <<'\n';
z= 4 + subtraction (x,y);
cout << "The fourth result is " << z << '\n';
}
```



```
The first result is 5
The second result is 5
The third result is 2
The fourth result is 6
```

For example, create a function to find the **square** or **cubic** for given number which are *not include* in **cmath** library,

```
int square(int x)
{
return x*x;
}
```

```
int cube(int x)
{
return x*x*x;
}
```

```
// create square and cubic functions
#include <iostream>
using namespace std;
int square(int x)
{
return x*x;
}

int cube(int x)
{
return x*x*x;
}
```

```
int main()
{
     int n;
     cout<<"Enter Number : ";
     cin>>n;
     cout<< "The square of ("<< n <<")is :"<<square(n)<<endl;
     cout<< "The cubic of ("<< n <<")is :"<<cube(n)<<endl;
}
```

```
Enter Number : 8
The square of (8) is : 64
The cubic of (8) is : 512
```

Example: create a function to find the following:

1) The **volume** of **sphere** $V = \frac{4}{3}\pi r^3$

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

double Vsphere(double R)
{
double V=(4.0/3.0)*3.14159*R*R*R;
return V;
}

int main()
{
     double r;
     cout<<"Enter radius : ";
     cin>>r;
     cout<< "\n The vol. of sphere = " << Vsphere(r) <<endl;
}
```

```
Enter radius : 25

 The vol. of sphere = 65449.8
```

2) Convert **Fahrenheit** to **Celsius** temperature using formula $C = \frac{(F-32)}{1.8}$

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

float F2C(int F)
{
float C=(F-32)/1.8;
return C;
}
```

```
int main()
{
     int F;
     cout<< "Enter Fahrenheit temperature:  ";
     cin>>F;
     cout<< "\n The Celsius temperature is: " << F2C(F)<<endl;
}
```

```
Enter Fahrenheit temperature:  70

 The Celsius temperature is: 21.1111
```

3) Find the **Factorial** $n! = (n)(n-1)\cdots(3)(2)(1)$

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

long Factorial(int n)
{
long f=1;
for (int i=n;i>0;i--)
f*=i;
return f;
}

int main()
{
     int N;
     cout<< "Enter the Number: ";
     cin>> N;
     if(N<0)
     cout<<"Error result";
     else
     cout<< "the factorial of ("<< N << ") = "<<Factorial(N)<<endl;


}
```

Example: From an **exercise 5** (page 51). Find the sine of angle (x in radian), solve until the difference of summation for last two terms is ≤ ***0.00001***

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots = \sum_{n=0}^{\infty} \frac{(-1)^n\, x^{2n+1}}{(2n+1)!}$$

```
#include <iostream>
#include <cmath>
#include <iomanip>
using namespace std;
# define pi 3.14159

 long double factorial(int n)
 {
    long double F = 1;
    for (int i = 1; i <= n; ++i)
        F *= i;
    return F;
 }

int main()
{
        int angle, n=0;
        double Er=1, S=0, SS, x;
        cout<< "Enter the angle in Degree : ";
        cin>>angle;
        int angle=angle%360;
        x=angle*pi/180;
        while (Er>0.00001)
        {
                SS=S;
                int sign = (n % 2 == 0) ? 1 : -1;
                int power = 2 * n + 1;
                S+=sign*pow(x,power)/factorial(power);
                Er=fabs(S-SS);
                n++;
        }
cout<<fixed<<"\n The sine of angle("<<angle<<") = "<<setprecision(5)<<S<<n;

}
```