Computer Systems:

All computer systems consist of similar hardware devices and software components.

- **Hardware**: Hardware refers to the physical components that a computer is made of. A computer is not an individual device, but a system of devices. A typical computer system consists of the following major components:
  1. The central processing unit (CPU)
  2. Main memory (RAM)
  3. Secondary storage devices
  4. Input devices
  5. Output devices

- **Software**: Software refers to the programs that run on a computer. There are two general categories of software: operating systems and application software. An operating system is a set of programs that manages the computer's hardware devices and controls their processes. Application software refers to program that the computer useful to the user. These programs solve specific problems or perform general operations that satisfy the needs of the user.

Programming Languages

There are two categories of programming languages: low level and high level. A low level language is close to the level of the computer, which means it resembles the numeric machine language of the computer more than natural language of humans. The easiest languages for people to learn are high level language. They are called "high level" because they are closer to the level of human readability than computer-readability like C++.

How C++ Programming Works

The C++ programming language is a popular and widely used programming language for creating **computer programs**. Programmers around the world embrace C++ because it gives maximum control and efficiency to the programmer.

If you are a programmer, or if you are interested in becoming a programmer, there are a couple of benefits you gain from learning C++:

- You will be able to read and write code for a large number of platforms -- everything from microcontrollers to the most advanced scientific systems can be written in C++, and many modern operating systems are written in C++.
- C++ is an object oriented language. C++ is an extension of C, C++ also includes several other improvements to the C language, including an extended set of library routines.

We will walk through the entire language and show you how to become a C++ programmer, starting at the beginning. You will be amazed at all of the different things you can create once you know C++!

What is C++?

C++ is a **computer programming language**. That means that you can use C++ to create lists of instructions for a computer to follow. C++ is one of thousands of programming languages currently in use. It gives programmers maximum control and efficiency. C++ is an easy language to learn. It is a bit more cryptic in its style than some other languages, but you get beyond that fairly quickly.

C++ is what is called a **compiled language**. This means that once you write your C++ program, you must run it through a C++ **compiler** to turn your program into an **executable** that the computer can run (execute). The C++ program is the human-readable form, while the executable that comes out of the compiler is the machine-readable and executable form.

When a C++ program is written, it must be typed/ into the computer and saved to a file. A text editor, which is similar to a word processing program, is used for this task. The statements written by the programmer are called **source code**, and the file they are saved in is called the source file. After the source code is saved to a file, the process of translating it to machine language can begin.

During the first phase of this process, a program called the **preprocessor** reads the source code. The preprocessor searches for special lines that begin with the (#) symbol. These lines contain commands that cause the preprocessor to modify the source code in some way as we will see later.

During the next phase the compiler steps through the preprocessed source code, translating each source code instruction into the appropriate machine language instruction. This process will uncover any syntax errors that may be in the program. Syntax errors are illegal uses of keywords, operators, punctuation, and other language elements. If the program is free of syntax errors, the compiler stores the translated machine language instructions, which are called **object code**, in an **object file**.

Although an **object file** contains machine language instructions, it is not a complete program, because C++ is conveniently equipped with a library of prewritten code for performing common operations or sometimes difficult tasks. For example, the library contains mathematical functions, such as calculating the square root of a number. The collection of code, called the **run time library**, is extensive. Programs almost always use some part of it. When the compiler generates an **object file**, however, it does not include machine code for any run time library routines the programmer might have used. During the last phase of the translation process, another program called the linker combines the object file with the necessary library routines. Once the linker has finished with this step, an **executable file** is created. The executable file contains machine language instructions, or **executable code**, and is ready to run on the computer.

Libraries

Libraries are very important in C++ because the C++ language supports only the most basic features that it needs. C++ does not even contain I/O functions to read from the keyboard and write to the screen. Anything that extends beyond the basic language must be written by a programmer. The resulting chunks of code are often placed in **libraries** to make them easily reusable. We have seen the standard I/O, or **stdio**, library already: Standard libraries exist for

standard I/O, math functions, string handling, time manipulation, and so on. You can use libraries in your own programs to split up your programs into modules. This makes them easier to understand, test, and debug, and also makes it possible to reuse code from other programs that you write.

The Simplest C++ Program

Let's start with the simplest possible C++ program and use it both to understand the basics of C++ and the C++ compilation process.

```
#include <iostream.h>

int main()
{
    cout << "Hello";
    return 0;
}
```

When executed, this program instructs the computer to print out the line "Hello" -- then the program quits. You can't get much simpler than that!

You should see the output "Hello" when you run the program. Here is what happened when you compiled the program:

If you mistype the program, it either will not compile or it will not run. If the program does not compile or does not run correctly, edit it again and see where you went wrong in your typing. Fix the error and try again.

The Simplest C++ Program: What's Happening?

- **#include <iostream.h>** tells the preprocessor to include the iostream standard file. This specific file (iostream) includes the declarations of the basic standard input-output library in C++, and it is included because its functionality is going to be used later in the program.
- The line **int main()** declares the main function. Every C++ program must have a function named **main** somewhere in the code. At run time, program execution starts at the first line of the main function.
- In C++, the **{** and **}** symbols mark the beginning and end of a block of code. In this case, the block of code making up the main function contains two lines.
- cout << "Hello"; This line is a C++ statement. A statement is a simple or compound expression that can actually produce some effect. In facts, this statement performs the only action that generates a visible effect in our first program. cout represents the standard output stream in C++, and the meaning of the entire statement is to insert a sequence of characters (in this case, the Hello sequence of characters) into the standard output stream (which usually is the screen).
- The **return 0;** line causes the function to return an error code of 0 (no error) to the shell that started execution.

- We can little update the previous program to become

```
#include <iostream.h>
#include <conio.h>
int main()
{
clrscr();
cout<<"Hello";
getch();
return 0;
}
```

  - **#include <conio.h>**. This line **includes** the "console I/O library" into your program.
  - clrscr() clears the current text window and places the cursor in the upper left-hand corner (at position 1,1).
  - getch() reads a single character directly from the keyboard, without echoing to the screen.

- **To output multiple lines**

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

int main()
{
    clrscr();
    cout << "Hello";
    cout<<"nice to meet you";
    getch();
    return 0;
}
```

The output becomes:

```
Hellonice to meet you
```

You can use \n for newline at the end of the 1st message, so that the 2nd message appear in the next line, and the program becomes:

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

int main()
{
    clrscr();
    cout << "Hello\n";
    cout<<"nice to meet you";
    getch();
    return 0;
}
```

The output becomes:

```
Hello
nice to meet you
```

which is the same as to

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

int main()
{
    clrscr();
    cout << "Hello\nnice to meet you";
    getch();
    return 0;
}
```

- You may also use the endl manipulator to add a new line:

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

int main()
{
    clrscr();
    cout << "Hello"<<endl<<"nice to meet you";
    getch();
    return 0;
}
```

Comments

Comments may be of the form:

// comment

or

/* comment */

The first form allows a trailing comment on a single line, while the second form allows comments that span multiple lines.

Comments may appear anywhere.

Function Declarations

Provides the full definition of a function, while
storage_class type identifier (formal_argument_list);

provides a *prototype* of a function which may be used to provide code which uses the function with the minimal essential parts of the definition to permit compilation. In the prototype the *formal_argument_list* must contain the types, but need not contain names for the formal arguments.

In older versions of C++, different syntax was used to declare formal arguments, placing them after the closing parenthesis and before the function body.

Example:

```
void main(void);

void main(void)

{

}
```

Constants and Variables

Constants are data items whose values cannot change while the program is running.

Literals

Literals are used to express particular values within the source code of a program. We have already used these previously to give concrete values to variables or to express message we wanted our programs to print out, for example, when we wrote:

A=5;

The 5 in the piece of code was a literal constants.

Literal constants can be divided in integer numerals, floating-point numerals, character, strings and Boolean values.

Integer Numerals

1776

707

-273

They are numerical constants that identify decimal values. Notice that to express a numerical constant we do not have to write quotes (") nor any special character. There is no doubt that it is a constant: whenever we write 1776 in a program, we will be referring to the value 1776.

In addition to decimal numbers (those that all of us are used to use every day) C++ allows the use as literal constants of octal numbers (base 8) and hexadecimal numbers (base 16). If we want to express an octal number we have to precede it with a 0 (zero character). And in order to express a hexadecimal number we have to precede it with the characters 0x (zero, x). For example, the following literal constants are all equivalent to each other:

```
75          //decimal
0113        //octal
0x4b        //hexadecimal
```

All of these represent the same number: 75 expressed as a base-10 numeral, octal numeral and hexadecimal numeral, respectively.

Floating point numbers

They express numbers with decimals and/or exponents. They can include either a decimal point, an e character (that express "by ten at the xth height", where x is an integer value that follows the e character), or both a decimal point and an e character:

3.14159 // 3.14159

6.02e23 // 6.02×10^{23}

1.6e-19 // 1.6×10^{-19}

3.0 // 3.0

Character and string literals

There also exist non-numerical constants, like:

'Z'

"Hello"

The 1st expression represent single character constant, and the 2nd expression represent string literals composed of several character. Notice that to represent a single character we enclose it between quotes (') and to express a string (which generally consists of more than one character) we enclose it between double quotes (").

When writing both single character and string literals, it is necessary to put the quotation marks surrounding them to distinguish them from possible variable identifiers or reserved keywords. Notice the difference between these two expressions:

x

'x'

x alone would refer to a variable whose identifier is x, whereas 'x' (enclosed within single quotation marks) would refer to the character constant 'x'.

Characters and string literals have certain peculiarities, like the escape codes. These are special characters that are difficult or impossible to express otherwise in the source code of program, like newline (\n) or tab (\t). All of them are preceded by a backslash (\). Here you have a list of some of such escape codes:

\n	Newline
\r	Carriage return
\t	Tab
\v	Vertical tab
\b	Backspace
\f	Form feed (page feed)
\a	Alert (beep)
\'	Single quote (')
\"	Double quote (")
\?	Question mark (?)
\\	Backslash (\)

For example:

'\n'

'\t'

"Left \t righr"

"one\ntwo\nthree"

String literals can extend to more than a single line of code by putting a backslash sign (\) at the end of each unfinished line.

"string expressed in \

two lines"

#define (directive)

Defines a macro

Syntax: #define <id1>[(<id2>, ...)] <token string>

The #define directive defines a macro.

Macros provide a mechanism for token replacement with or without a set of formal, function-line parameters.

All subsequent instances of the identifier <id1> in the source text will be replaced by the text defined by <token string>.

If <id1> is followed IMMEDIATELY by a "(", the identifiers following the "(" (<id2>, etc.) are treated like parameters to a function instead of as part of <token string>.

All instances of <id2> in <token string> will be replaced with the actual text defined for <id2> when <id1> is referenced in the source.

To continue the definitions on another line, end the current line with \

```
Example:
 #define MAXINT 32767
 #define ctrl(x) ((x) \
   - 64) /* continued from preceding line */
```

It is legal but ill-advised to use Turbo C++ keywords as macro identifiers.

Example

```
#include <iostream.h>
#define x 5

int main()
{
    cout << "x:="<<x;    //x=5
    return 0;
}
```

In the above program, the value of x is fixed and can't be changed later

Variables

As a programmer, you will frequently want your program to "remember" a value. For example, if your program requests a value from the user, or if it calculates a value, you will want to remember it somewhere so you can use it later. The way your program remembers things is by using **variables**. Variables represent storage locations in the computer's memory. For example:

```
int b;
```

This line says, "I want to create a space called b that is able to hold one integer value." A variable has a **name** (in this case, b) and a **type** (in this case, int, an integer). You can store a value in b by saying something like:

```
b = 5;
```

You can use the value in b by saying something like:

```
cout << b
The output will be:
5

While writing:
cout << "b"
The output will be:
b

and writing:
cout << "b=" << b
The output will be:
b=5

To display a text line No.= with the value of b use:
cout << "No.=" << b
The output will be:
No.=5
```

In C++, there are several standard types for variables:

- **Integer: int**
- **Floating point: float**
- **Character: char**

An int is a 2-byte integer value. A float is a 4-byte floating point value. A char is a 1-byte single character (like "a" or "A") or any other 8-bit quantity, when a character is stored in memory, it is actually the numeric code is stored. A string is declared as an array of characters.

There are a number of derivative types:

- **double** (8-byte floating point value)
- **short** (2-byte integer)
- **unsigned short** or **unsigned int** (positive integers, no sign bit)

Data Types

Type	Length	Range
unsigned char	8 bits	0 to 255
char	8 bits	-128 to 127
enum	16 bits	-32,768 to 32,767
unsigned int	16 bits	0 to 65,535
short int	16 bits	-32,768 to 32,767
int	16 bits	-32,768 to 32,767
unsigned long	32 bits	0 to 4,294,967,295
long	32 bits	-2,147,483,648 to 2,147,483,647
float	32 bits	3.4 * (10**-38) to 3.4 * (10**+38)
double	64 bits	1.7 * (10**-308) to 1.7 * (10**+308)
long double	80 bits	3.4 * (10**-4932) to 1.1 * (10**+4932)
void	-	valueless

Note: void data type specifies a valueless expression.

Examples:

```
char a='d';

int b=50;

float c=5.3;
```

Q) write a program to calculate the area of circle, radius=4.5

```
#include <iostream.h>
#include <conio.h>
#define pi 3.1415
int main()
{
 float r=4.5,area;
 clrscr();
 area=r*r*pi;
 cout << "Area of circle="<<area;
 getch();
}
```

Arithmetic Operators

1- Plus and minus operators (+ and -)

+ cast-expression: Value of the operand after any required integral promotions.

- cast-expression: Negative of the value of the operand after any required integral promotions.

Example: y = -y;

Example:

z=x+y; c=a-b;

2-Multiplicative operators (* / %)

There are three multiplicative operators:

* (multiplication), / (division), % (modulus or remainder)

```
void main()
{
    int a=5;
    a=a%2;
    cout <<"a="<<a; //a=1 remainder of division a/2
 }
```

The / operator performs integer division if both operands are integers, and performs floating point division otherwise. For example:

```
void main()
{
    float a;
    a=10/3;
    cout <<"a="<<a;
}
```

This code prints out a floating point value since **a** is declared as type **float**, but **a** will be 3.0 because the code performed an integer division.

Typecasting

C++ allows you to perform type conversions on the fly. You do this especially often when using pointers. Typecasting also occurs during the assignment operation for certain types. For example, in the code above, the integer value was automatically converted to a float.

You do typecasting in C++ by placing the type name in parentheses and putting it in front of the value you want to change. Thus, in the above code, replacing the line **a=10/3;** with **a=(float)10/3;** produces 3.33333 as the result because 10 is converted to a floating point value before the division, also you can say a=10.0/3;

Precedence of Operators

Operator precedence in C++ is also similar to that in most other languages. Division and multiplication occur first, then addition and subtraction. The result of the calculation 5+3*4 is 17, not 32, because the * operator has higher precedence than + in C++. You can use parentheses to change the normal precedence ordering: (5+3)*4 is 32. The 5+3 is evaluated first because it is in parentheses. We'll get into precedence later -- it becomes somewhat complicated in C++ once pointers are introduced.