Chapter one

Problem-Solving Using Computers

1.1. Basics of Program Development

In order to solve a given problem, computers must be given the correct instruction about how they can solve it. The terms **computer programs**, **software programs**, or just **programs** are the instructions that tells the computer what to do. Computer requires programs to function, and a computer programs does nothing unless its instructions are executed by a CPU. **Computer programming** (often shortened to **programming** or **coding**) is the process of writing, testing, debugging/troubleshooting, and maintaining the source code of computer programs. Writing computer programs means writing instructions that will make the computer follow and run a program based on those instructions.

A computer program (also known as source code) is often written by professionals known as **Computer Programmers** (simply **programmers**). Source code is written in one of programming languages. A **programming language** is an artificial language that can be used to control the behavior of a machine, particularly a computer. Programming languages, like natural language (such as Amharic), are defined by syntactic and semantic rules which describe their structure and meaning respectively. The syntax of a language describes the possible combinations of symbols that form a syntactically correct program. The meaning given to a combination of symbols is handled by semantics.

The process in which all z steps are done starting from the feasibility study up to the development of the exact system is known as **program development**. The life cycle of program development will be discussed in section 1.4 in detail.

1.2. Flowcharting, Algorithms, Pseudo Code

Computer solves varieties of problems that can be expressed in a finite number of steps leading to a precisely defined goal by writing different programs. A program is not needed only to solve a problem but also it should be reliable, (maintainable) portable and efficient. In computer programming two facts are given more weight:

• The first part focuses on defining the problem and logical procedures to follow in solving it.

• The second introduces the means by which programmers communicate those procedures to the computer system so that it can be executed.

There are system analysis and design tools, particularly flowcharts and structure chart that can be used to define the problem in terms of the steps to its solution. The programmer uses programming language to communicate the logic of the solution to the computer. Before a program is written, the programmer must clearly understand what data are to be used, the desired result, and the procedure to be used to produce the result. The procedure, or solution, selected is referred to as an algorithm.

1.2.1. Algorithm

Any computing problem can be solved by executing a series of actions in a specific order. A procedure for solving a problem in terms of

I. the actions to execute and

2. the order in which these actions execute

is called an algorithm. An algorithm is defined as a step-by-step sequence of instructions that must terminate and describe how the data is to be processed to produce the desired outputs. Simply, algorithm is a sequence of instructions. Algorithms are a fundamental part of computing. There are three commonly used tools to help to document program logic (the algorithm). These are flowcharts, structured chart, and Pseudo Code.

1.2.2. Pseudo Code

Pseudo code (derived from pseudo and code) is a compact and informal high-level description of a computer algorithm that uses the structural conventions of programming languages, but typically omits details such as subroutines, variables declarations and system-specific syntax. The programming language is augmented with natural language descriptions of the details, where convenient, or with compact mathematical notation. The purpose of using pseudo code is that it may be easier for humans to read than conventional programming languages, and that it may be a compact and environment-independent generic description of the key principles of an algorithm. No standard for pseudo code syntax exists, as a program in pseudo code is not an executable program. As the name suggests, pseudo code generally does not actually obey the syntax rules of any particular language; there is no systematic standard form, although any particular writer will generally borrow the appearance of a particular language.

Example**:**

Original Program Specification:

Write a program that obtains two integer numbers from the user. It will print out the sum of those numbers.

Pseudo code:

```
Prompt the user to enter the first integer
Prompt the user to enter a second integer
Compute the sum of the two user inputs
Display an output prompt that explains the answer as the sum
Display the result
```

1.2.3. Flowcharts

A flowchart (also spelled flow-chart and flow chart) is a schematic representation of an algorithm or a process. The advantage of flowchart is it doesn't depend on any particular programming language, so that it can be used to translate an algorithm to more than one programming language. Flowchart uses different symbols (geometrical shapes) to represent different processes. The following table shows some of the common symbols.

Example 2: Write an algorithm description and draw a flow chart to check a number is negative or not.

Algorithm description

```
1/ Read a number x
2/ If x is less than zero write a message negative
else write a message not negative
```

Sometimes there are conditions in which it is necessary to execute a group of statements repeatedly. Until some condition is satisfied. This condition is called a loop. Loop is a sequence of instructions, which is repeated until some specific condition occurs. A loop normally consists of four parts. These are:

Initialization: - Setting of variables of the computation to their initial values and setting the counter for determining to exit from the loop.

Computation: - Processing

Test: - Every loop must have some way of exiting from it or else the program would endlessly remain in a loop.

Increment: - Re-initialization of the loop for the next loop.

Example 3: - Write the algorithmic description and draw a flow chart to find the following sum.

Sum = 1+2+3+…. + 50

Algorithmic description

```
1. Initialize sum too and counter to 1
1.1. If the counter is less than or equal to 50
• Add counter to sum
• Increase counter by 1
• Repeat step 1.1
Else
• Exit
2. Write sum
```

1.3. Software Engineering

Software Engineering is an approach to developing software that attempts to treat it as a formal process more like traditional engineering than the craft that many programmers believe it is. Software engineering (SE) is the application of a systematic, disciplined, quantifiable approach to the design, development, operation, and maintenance of software, and the study of these approaches; that is, the application of engineering to software. A C++ Software Engineer is responsible for developing and/or implementing the new features to improve the existing programs and software.

1.4. Program Development Life Cycle

The Program Development Life Cycle is a conceptual model used in project management that describes the stages involved in a computer system development project from an initial feasibility study through maintenance of the completed application.

1.4.1. Feasibility Study

The first step is to identify a need for the new system. This will include determining whether a business problem or opportunity exists, conducting a feasibility study to determine if the proposed solution is cost effective, and developing a project plan.

A preliminary analysis, determining the nature and scope of the problems to be solved is carried out. Possible solutions are proposed, describing the cost and benefits. Finally, a preliminary plan for decision making is produced.
The process of developing a large information system can be very costly, and the investigation stage may require a preliminary study called a feasibility study, which includes e.g. the following components:

a. Organizational Feasibility

- How well the proposed system supports the strategic objectives of the organization.

b. Economic Feasibility

- Cost savings
- Increased revenue
- Decreased investment
- Increased profits

c. Technical Feasibility

- Hardware, software, and network capability, reliability, and availability

d. Operational Feasibility

- End user acceptance
- Management support
- Customer, supplier, and government requirements

1.4.2. Requirements analysis

Requirements analysis is the process of analyzing the information needs of the end users, the organizational environment, and any system presently being used, developing the functional requirements of a system that can meet the needs of the users. Also, the requirements should be recorded in a document, email, user interface storyboard, executable prototype, or some other form. The requirements documentation should be referred to throughout the rest of the system development process to ensure the developing project aligns with user needs and requirements.
End users must be involved in this process to ensure that the new system will function adequately and meets their needs and expectations.

1.4.3. Designing solution

After the requirements have been determined, the necessary specifications for the hardware, software, people, and data resources, and the information products that will satisfy the functional requirements of the proposed system can be determined. The design will serve as a blueprint for the system and helps detect problems before these errors or problems are built into the final system.

The created system design, but must be reviewed by users to ensure the design meets users' needs.

1.4.4. Testing designed solution

A smaller test system is sometimes a good idea in order to get a "proof-of-concept" validation prior to committing funds for large scale fielding of a system without knowing if it really works as intended by the user.

1.4.5. Implementation

The real code is written here. Systems implementation is the construction of the new system and its delivery into production or day-to-day operation. The key to understanding the implementation phase is to realize that there is a lot more to be done than programming. Implementation requires programming, but it also requires database creation and population, and network installation and testing. You also need to make sure the people are taken care of with effective training and documentation. Finally, if you expect your development skills to improve over time, you need to conduct a review of the lessons learned.

1.4.6. Unit testing

Normally programs are written as a series of individual modules, these subjects to separate and detailed test.

1.4.7. Integration and System testing

This step brings all the pieces together into a special testing environment, then checks for errors, bugs and interoperability. The system is tested to ensure that interfaces between modules work (integration testing), the system works on the intended platform and with the expected volume of data (volume testing) and that the system does what the user requires (acceptance/beta testing).

1.4.8. Maintenance

What happens during the rest of the software's life: changes, correction, additions, moves to a different computing platform and more. This, the least glamorous and perhaps most important step of all, goes on seemingly forever.

1.5. Overview of Computer Programming Languages

Available programming languages come in a variety of forms and types. Thousands of different programming languages have been developed, used, and discarded. Programming languages can be divided in to two major categories: low-level and high-level languages.

1.5.1. Low-level languages

Computers only understand one language and that is binary language or the language of 1s and 0s. Binary language is also known as machine language, one of low-level languages. In the initial years of computer programming, all the instructions were given in binary form. Although the computer easily understood these programs, it proved too difficult for a normal human being to remember all the instructions in the form of 0s and 1s. Therefore, computers remained mystery to a common person until other languages such as assembly language was developed, which were easier to learn and understand.

Assembly language is a symbolic representation of machine code, which allows symbolic designation of memory locations.

1.5.2. High-level languages

Although programming in assembly language is not as difficult and error prone as stringing together ones and zeros, it is slow and cumbersome. In addition it is hardware specific. The lack of portability between different computers led to the development of high-level languages—so called because they permitted a programmer to ignore many low-level details of the computer's hardware.

Another most fundamental ways programming languages are characterized (categorized) is by programming paradigm. A **programming paradigm** provides the programmer's view of code execution. The most influential paradigms are examined in the following sections, in approximate chronological order.

Procedural Programming Languages

The imperative (procedural) programming paradigm is the oldest and the most traditional one. It has grown from machine and assembler languages. An imperative program consists of explicit commands (instructions) and calls of procedures (subroutines) to be consequently executed; they carry out operations on data and modify the values of program variables (by means of assignment statements), as well as external environment.

Within this paradigm variables are considered as containers for data similar to memory cells of computer memory.

The 'first do this, next do that' is a short phrase which really in a nutshell describes the spirit of the imperative paradigm. The basic idea is the command, which has a measurable effect on the program state. The phrase also reflects that the order to the commands is important. 'First do that, then do this' would be different from 'first do this, then do that'.

Languages that use this paradigm - Fortran, Algol, Pascal, Basic, C

Functional Programming Languages

The functional paradigm is in fact an old style too, since it has arisen from evaluation of algebraic formulae, and its elements were used in first imperative algorithmic languages such as Fortran. Pure functional program is a collection of mutually related (and possibly recursive) functions. Each function is an expression for computing a value and is defined as a composition of standard (built-in) functions. Execution of functional program is simply application of all functions to their arguments and thereby computation of their values.

In this paradigm we express computations as the evaluation of mathematical functions. Functional programming paradigms treat values as single entities. Unlike variables, values are never modified. Instead, values are transformed into new values. Computations of functional languages are performed largely through applying functions to values

Languages that use this paradigm - Lisp, Refal, Planner, Scheme;

Logic programming paradigms

In this paradigm we express computation in exclusively in terms of mathematical logic. While the functional paradigm emphasizes the idea of a mathematical function, the logic paradigm focuses on predicate logic, in which the basic concept is a relation. Logic languages are useful for expressing problems where it is not obvious what the functions should be.

Within the logic paradigm, program is thought of as a set of logic formulae: axioms (facts and rules) describing properties of certain objects, and a theorem to be proved. Program execution is a process of logic proving (inference) of the theorem through constructing the objects with the described properties. The logic paradigm fits extremely well when applied in problem domains that deal with the extraction of knowledge from basic facts and relations

Let us consider now how we can define the brother relation in terms of simpler relations and properties father, mother, and male. Using the Prolog logic language one can say:

```
brother(X,Y) /* X is the brother of Y */
/* if there are two people F and M for which*/
father(F,X), /* F is the father of X */
father(F,Y), /* and F is the father of Y */
mother(M,X), /* and M is the mother of X */
mother(M,Y), /* and M is the mother of Y */
male(X). /* and X is male */
```

Languages that use this paradigm - Prolog

Object-Oriented Programming Languages

Object-oriented programming is the newest and most powerful paradigms. OO programming paradigm is not just a few new features added to a programming language, but it a new way of thinking about the process of decomposing problems and developing programming solutions. In object- oriented programs, the designer specifies both the data structures and the types of operations that can be applied to those data structures. This pairing of a piece of data with the operations that can be performed on it is known as an object. An object encapsulates passive data and active operations on these data: it has a storage fixing its state (structure) and a set of methods (operations on the storage) describing behavior of the object. A program thus becomes a collection of cooperating objects, rather than a list of instructions. Objects can store state information and interact with other objects, but generally each object has a distinct, limited role.

Languages that follow this paradigm - Smalltalk, Eiffel, C++, and object Pascal

Table 1. Features of programming paradigms

1.6. The C++ Compilation Process

C++ systems generally consist of three parts: a program**-development environment, the language and the C++ Standard Library**. The following discussion explains a typical C++ program development environment.

C++ programs typically go through six phases to be executed. These are: edit, preprocess, compile, link, load and execute. The first phase consists of editing a file. This is accomplished with an editor program.

The programmer types a C++ program with the editor and makes corrections if necessary. Next, the programmer gives the command to compile the program. The compiler translates the C++ program into machine language code (also referred to as object code). In a C++ system, a preprocessor program executes automatically before the compiler's translation phase begins. The C++ preprocessor obeys commands called preprocessor directives, which indicate that certain manipulations are to be performed on the program before compilation. The preprocessor is invoked by the compiler before the program is converted to machine language. The next phase is called linking. C++ programs typically contain references to functions and data defined elsewhere, such as in the standard libraries or in the private libraries. The object code produced by the C++ compiler typically contains "holes" due to these missing parts. A linker links the object code with the code for the missing functions to produce an executable image (with no missing pieces). If the program compiles and links correctly, an executable image is produced.

The next phase is called loading. Before a program can be executed, the program must first be placed in memory. This is done by the loader, which takes the executable image from disk and transfers it to memory. Additional components from shared libraries that support the program are also loaded. Finally, the computer, under the control of its CPU, executes the program one instruction at a time.

A typical C++ Environment

1.7. Introduction to the Preprocessor

This chapter introduces the preprocessor. Preprocessing occurs before a program is compiled. Some possible actions are **inclusion of other files in the file being compiled, definition of symbolic constants and macros, conditional compilation of program code and conditional execution of preprocessor directives**. All preprocessor directives begin with#, and only whitespace characters may appear before a preprocessor directive on a line. **Preprocessor directives are not C++ statements, so they do not end in a semicolon (;)**. Preprocessor directives are processed fully before compilation begins.

1.7.1. The # include preprocessor directive

The # include preprocessor directive causes a copy of a specified file to be included in place of t h e directive. The two forms of the # include directive are

# include < filename >

# include "filename"

The difference between these is the location the preprocessor searches for the file to be included. If the filename is enclosed in angle brackets (< and>)-used for standard library header files-, the preprocessor searches for the specified file in an implementationdependent manner, normally through predesignated directories. If the file name is enclosed in quotes, the preprocessor searches first in the same directory as the file being compiled, then in the same implementation-dependent manner as for a file name enclosed in angle brackets. This method is normally used to include programmer-defined header files.

1.7.2. The # define preprocessor Directive: Symbolic Constants

The #define preprocessor directive creates symbolic constants-constants represented as symbols-and macros-operations defined as symbols. The # define preprocessor directive format is

#define identifier replacement-text

When this line appears in a file, all subsequent occurrences (except those inside a string) of identifier in that file will be replaced by replacement-text before the program is compiled. For example,

#define PI 3.14159

replaces all subsequent occurrences of the symbolic constant PI with the numeric constant 3.14159. Symbolic constants enable the programmer to create a name for a constant and use the name throughout the program. Later, if the constant needs to be modified throughout the program, it can be modified once in the # define preprocessor directive-and when the program is recompiled, all occurrences of the constant in the program will be modified.

1.7.3. The # define preprocessor Directive: Macros

A macro is an operation defined in a #define preprocessor directive. As with symbolic constants, the macro-identifier is replaced with the replacement-text before the program is compiled. Macros may be defined with or without arguments. A macro without arguments is processed like a symbolic constant. In a macro with arguments, the arguments are substituted in the replacement-text, then the macro is expanded-i.e., the replacement text replaces the macro-identifier and argument list in the program.

Consider the following macro definition with one argument for the area of a circle:

```
#define CIRCLE_AREA (x) ( PI * (x) * ( x ) )
```

Wherever CIRCLE_A REA (x) appears in the file, the value of x is substituted for x in the replacement text, the symbolic constant PI is replaced by its value (defined previously) and the macro is expanded in the program. For example, the statement

```
area = CIRCLE_AREA (4);
```

is expanded to

```
area = (3.14159 * (4 ) * (4)) ;
```

1.7.4. Conditional Compilation

Conditional compilation enables the programmer to control the execution of preprocessor directives and the compilation of program code. Each of the conditional preprocessor directives evaluates a constant integer expression that will determine whether the code will be compiled. **Cast expressions, sizeof expressions and enumeration constants cannot be evaluated in preprocessor directives**. The conditional preprocessor construct is much like the if selection structure. Consider the following preprocessor code:

```
# ifndef NULL
#define NULL 0
#endif
```

These directives determine if the symbolic constant **NULL** is already defined. The expression defined (NULL) evaluates to 1 if NULL is defined, and 0 otherwise. If the result is 0, **!defined (NULL)** evaluates to 1, and NULL is defined. Otherwise, the #define directive is skipped. Every **#if** construct ends with **#endif**. Directives **#ifdef** and **#ifndef** are shorthand for #if defined (name) and **#if !defined (name).** A multiple-part conditional preprocessor construct may be tested using the **#elif** (the equivalent of else if in an if structure) and the #else (the equivalent of else in an if structure) directives.

1.7.5. The #error and #pragma preprocessor directives

The #error directive

#error tokens prints an implementation-dependent message including the tokens speci fied i n the directive. The tokens are sequences of characters separated by spaces. For example, *#error 1 - Out of range error* contains six tokens. In one popular C++ compiler, for example, when a #error directive is processed, the tokens in the directive are displayed as an error

message, preprocessing stops and the program does not compile.

The #pragma directives

#pragma tokens causes an implementation-defined action. A pragma not recognized by the implementation is ignored. A particular C++ compiler, for example, might recognize pragmas that enable the programmer to take advantage of that compiler's specific capabilities.

1.7.6. The # and ## operators

The # and ## preprocessor operators are available in C++ and ANSIC. The # operator causes a replacement-text token to be converted to a string surrounded by quotes. Consider the following macro definition:

#define HELLO (x) cout « " Hello , " # x « end1 ;

When HELLO (John) appears in a program file, it is expanded to

cout « " Hello , " " John " « endl ;

The string " John " rep l aces #x in the replacement text. Strings separated by whitespace are concatenated during preprocessing, so the above statement is equivalent to

cout « " He l l o , John " « endl ;

Note that the # operator must be used in a macro with arguments, because the operand of # refers to an argument of the macro.

The ## operator concatenates two tokens. Consider the following macro definition:

#define TOKENCONCAT (x , y) x ## y

When TOKENCONCAT appears in the program, its arguments are concatenated and used to replace the macro. For example, TOKENCONCAT

1.7.7. Line numbers

The #line preprocessor directive causes the subsequent source code lines to be renumbered starting with the specified constant integer value. The directive

#line 100

starts line numbering from 100, beginning with the next source code line. A file name can be included in the #l in e directive. The directive

#line 100 " file1.cpp "

indicates that lines are numbered from 100, beginning with the next source code line and that the name of the file for the purpose of any compiler messages is "file1.cpp". The directive could be used to help make the messages produced by syntax errors and compiler warnings more meaningful. The line numbers do not appear in the source file.

1.7.8. Predefined symbolic Constants

The identifiers for each predefined symbolic constant begin and end with two underscores. These identifiers and the defined preprocessor operator cannot be used in #define or #undef directives.

There are six predefined symbolic constants:

1.7.9. Assertions

The *assert* macro-defined in the <cassert> header file-tests the value of an expression. If the value of the expression is 0 (false), then assert prints an error message and calls function abort (of the general utilities library-<cstdlib>) to terminate program execution. This is a useful debugging tool for testing whether a variable has a correct value. For example, suppose variable x should never be larger than 10 in a program. An assertion may be used to test the value of x and print a n error message if the value of x is incorrect. The statement would be

```
assert ( x <= 10 ) ;
```

If x is greater than 10 when the preceding statement is encountered in a program, an error message containing the line number and file name is printed, and the program terminates. The programmer may then concentrate on this area of the code to find the error. I f the symbolic constant NDEBUG is defined, subsequent assertions will be ignored. Thus, when assertions are no longer needed (i.e., when debugging is complete), the line

```
#define NDEBUG
```

is inserted in the program file rather than deleting each assertion manually. Most C + + compilers now include exception handling. C++ programmers prefer using exceptions rather than assertions. But assertions are still valuable for C++ programmers who work with C legacy code.

Chapter Two

Basic Concepts of C++ Programming

2.1. Variables, Constants, Initializing Variables

2.1.1. Variables

A variable is a symbolic name for a memory location in which data can be stored and subsequently recalled. Variables are used for holding data values so that they can be utilized in various computations in a program.

- A variable is a reserved place in memory to store information in.
- Variables are used for holding data values so that they can be used in various computations in a program.

All variables have three important properties:

- Data Type: a type which is established when the variable is defined. (e.g. integer, real, character etc). Data type describes the property of the data and the size of the reserved memory
- Name: a name which will be used to refer to the value in the variable. A unique identifier for the reserved memory location
- Value: a value which can be changed by assigning a new value to the variable.

2.1.1.1. Variable Declaration

Variables can be created in a process known as declaration. Declaring a variable means defining (creating) a variable. You create or define a variable by stating its type, followed by one or more spaces, followed by the variable name and a semicolon. The variable name can be virtually any combination of letters, but cannot contain spaces and the first character must be a letter or an underscore.

Syntax: Datatype Variable_Name;

Variable names cannot also be the same as keywords used by C++. Legal variable names include x, J23f, and myAge. Good variable names tell you what the variables are for; using good names makes it easier to understand the flow of your program. The following statement defines an integer variable called myAge:

int myAge;

IMPORTANT-Variables must be declared before used!

As a general programming practice, avoid such horrific names as J23qrsnf, and restrict single-letter variable names (such as x or i) to variables that are used only very briefly. Try to use expressive names such as myAge or howMany.

A point worth mentioning again here is that C++ is case-sensitive. In other words, uppercase and lowercase letters are considered to be different. A variable named age is different from Age, which is different from AGE.

- The name of a variable sometimes is called an identifier which should be unique in a program.
- Certain words are reserved by C++ for specific purposes and cannot be used as identifiers.

Creating More Than One Variable at a Time

You can create more than one variable of the same type in one statement by writing the type and then the variable names, separated by commas. For example:

Int myAge, myWeight; // two int variables

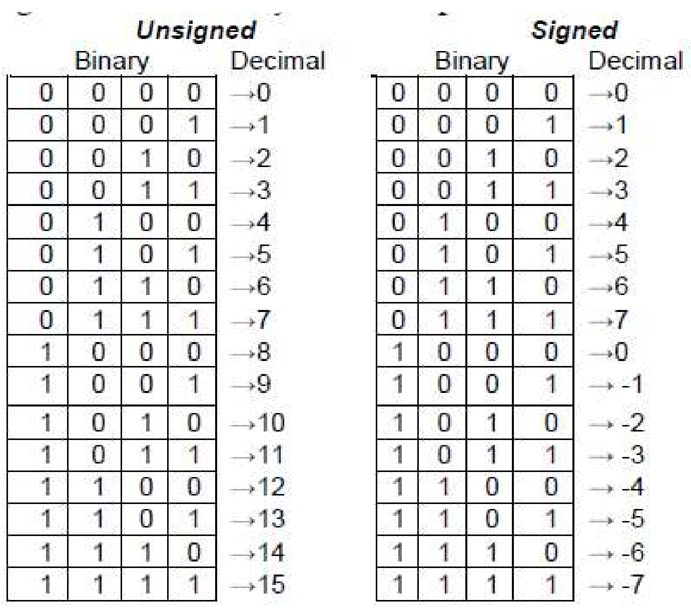
long area, width, length; // three longs

As you can see, myAge and myWeight are each declared as integer variables. The second line declares three individual long variables named area, width, and length. However keep in mind that you cannot mix types in one definition statement.

2.5.1.2. Signed and Unsigned.

- Signed integers are either negative or positive. Unsigned integers are always positive.
- Because both signed and unsigned integers require the same number of bytes, the largest number (the magnitude) that can be stored in an unsigned integer is twice as the largest positive number that can be stored in a signed integer.

E.g.: Lets us have only 4 bits to represent numbers



Unsigned					Signed				
Binary				Decimal	Binary				Decimal
0	0	0	0	→0	0	0	0	0	→0
0	0	0	1	→1	0	0	0	1	→1
0	0	1	0	→2	0	0	1	0	→2
0	0	1	1	→3	0	0	1	1	→3
0	1	0	0	→4	0	1	0	0	→4
0	1	0	1	→5	0	1	0	1	→5
0	1	1	0	→6	0	1	1	0	→6
0	1	1	1	→7	0	1	1	1	→7
1	0	0	0	→8	1	0	0	0	→0
1	0	0	1	→9	1	0	0	1	→ -1
1	0	1	0	→10	1	0	1	0	→ -2
1	0	1	1	→11	1	0	1	1	→ -3
1	1	0	0	→12	1	1	0	0	→ -4
1	1	0	1	→13	1	1	0	1	→ -5
1	1	1	0	→14	1	1	1	0	→ -6
1	1	1	1	→15	1	1	1	1	→ -7

- In the above example, in case of unsigned, since all the 4 bits can be used to represent the magnitude of the number the maximum magnitude that can be represented will be 15 as shown in the example.
- If we use signed, we can use the first bit to represent the sign where if the value of the first bit is 0 the number is positive if the value is 1 the number is negative. In this case we will be left with only three bits to represent the magnitude of the number. Where the maximum magnitude will be 7.
- Because you have the same number of bytes for both signed and unsigned integers, the largest number you can store in an unsigned integer is twice as big as the largest positive number you can store in a signed integer. An unsigned short integer can handle numbers from 0 to 65,535. Half the numbers represented by a signed short are negative, thus a signed short can only represent numbers from -32,768 to 32,767.

Example: A demonstration of the use of variables.

```
2: #include <iostream.h>

3:

4: intmain()

5: {

6: unsigned short int Width = 5, Length;
7: Length = 10;
8:
9: // create an unsigned short and initialize with result
10: // of multiplying Width by Length
11: unsigned short intArea = Width * Length;
12:
13: cout<< "Width:" << Width << "\n";
14: cout<< "Length: " << Length <<endl;
15: cout<< "Area: " << Area <<endl;
16: return 0;
17: }
```

Output:
```
Width:5
Length: 10
Area: 50
```

Line 2 includes the required include statement for the iostream's library so that cout will work. Line 4 begins the program.

On line 6, Width is defined as an unsigned short integer, and its value is initialized to 5. Another unsigned short integer, Length, is also defined, but it is not initialized. On line 7, the value 10 is assigned to Length.

On line 11, an unsigned short integer, Area, is defined, and it is initialized with the value obtained by multiplying Width times Length. On lines 13-15, the values of the variables are printed to the screen. Note that the special word endl creates a new line.

2.1.2. Constants

- A constant is any expression that has a fixed value.
- Like variables, constants are data storage locations in the computer memory. But, constants, unlike variables their content cannot be changed after the declaration.

- Constants must be initialized when they are created by the program, and the programmer can't assign a new value to a constant later.
- In C++, we have two ways to declare a symbolic constant. These are using the #define and the const key word.

2.1.2.1. Defining constants with #define:

- The #define directive makes a simple text substitution.
- The define directive can define only integer constants

E.g.: #define studentPerClass 15

- In our example, each time the preprocessor sees the word studentPerClass, it inserts 15 into the text.

2.1.2.2. Defining constants with the const key word:

- Here, the constant has a type, and the compiler can ensure that the constant is used according to the rules for that type.

E.g.: const unsigned short intstudentPerClass = 15;

2.1.3. Initializing Variables

- When a variable is assigned a value at the time of declaration, it is called variable initialization.
- This is identical with declaring a variable and then assigning a value to the variable immediately after declaration.
- The syntax: DataType variable name = initial value;

e.g. int a = 0;

or: int a;

a=0;

2.2. Data Types

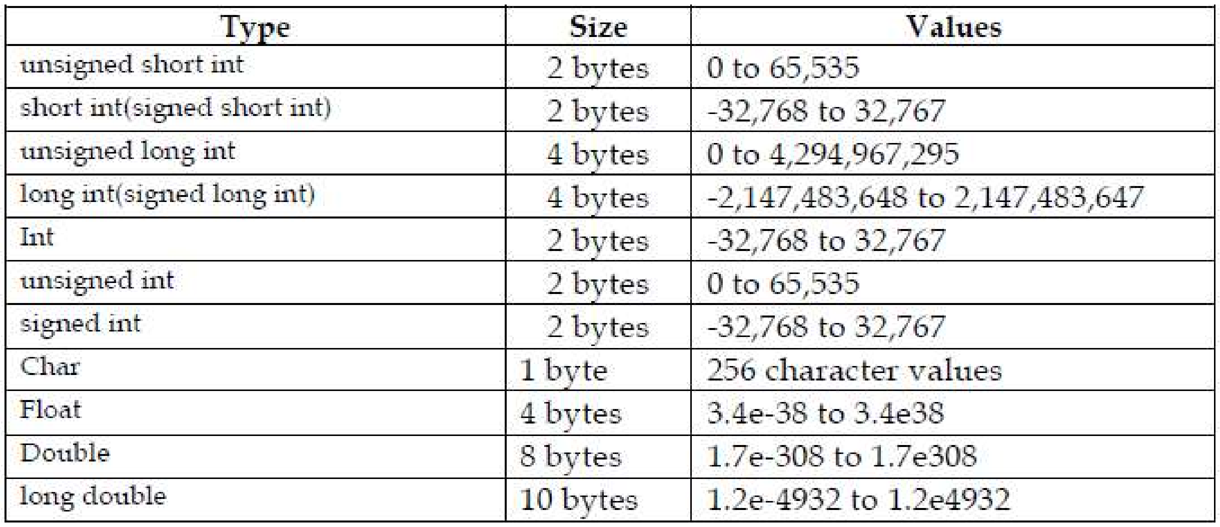
When you define a variable in C++, you must tell the compiler what kind of variable it is: an integer, a character, and so forth. This information tells the compiler how much room to set aside and what kind of value you want to store in your variable.

Basic (fundamental) data types in C++ can be conveniently divided into numeric and character types. Numeric variables can further be divided into integer variables and floating-point variables. Integer variables will hold only integers whereas floating number variables can accommodate real numbers.

Both the numeric data types offer modifiers that are used to vary the nature of the data to be stored. The modifiers used can be short, long, signed and unsigned.

The data types used in C++ programs are described in the following table. This table shows the variable type, how much room it takes in memory, and what kinds of values can be stored in these variables. The values that can be stored are determined by the size of the variable types.

Table 2.1. Data types and their ranges



Type	Size	Values
unsigned short int	2 bytes	0 to 65,535
short int(signed short int)	2 bytes	-32,768 to 32,767
unsigned long int	4 bytes	0 to 4,294,967,295
long int(signed long int)	4 bytes	-2,147,483,648 to 2,147,483,647
Int	2 bytes	-32,768 to 32,767
unsigned int	2 bytes	0 to 65,535
signed int	2 bytes	-32,768 to 32,767
Char	1 byte	256 character values
Float	4 bytes	3.4e-38 to 3.4e38
Double	8 bytes	1.7e-308 to 1.7e308
long double	10 bytes	1.2e-4932 to 1.2e4932

2.3. Assigning Values to Variables

You assign a value to a variable by using the assignment operator (=). Thus, you would assign 5 to Width by writing:

int Width;

Width = 5;

You can combine these steps and initialize Width when you define it by writing:

Int Width=5;

Initialization looks very much like assignment, and with integer variables, the difference is minor. The essential difference is that initialization takes place at the moment you create the variable.

Just as you can define more than one variable at a time, you can initialize more than one variable at creation. For example:

// create two int variables and initialize them

int width = 5, length = 7;

This example initializes the integer variable width to the value 5 and the length variable to the value 7. It is possible to even mix definitions and initializations:

int myAge = 39, yourAge, hisAge = 40;

This example creates three type int variables, and it initializes the first and third.

2.4. Expressions, Comments, Statements, Identifier, Keywords

2.4.1. Expressions

An expression is a computation which yields a value. It can also be viewed as any statement that evaluates to a value (returns a value).

E.g.: the statement 3+2; returns the value 5 and thus is an expression.

- Some examples of an expression:

E.g.1:

3.2 returns the value 3.2

PI float constant that returns the value 3.14 if the constant is defined.

secondsPerMinute integer constant that returns 60 if the constant is declared

E.g.2: complicated expressions:

x = a + b;

y = x = a + b;

The second line is evaluated in the following order:

1. add a to b.

2. assign the result of the expression a + b to x.

3. assign the result of the assignment expression x = a + b to y.

2.4.2. Comments

- A comment is a piece of descriptive text which explains some aspect of a program.

- Program comments are text totally ignored by the compiler and are only intended to inform the reader how the source code is working at any particular point in the program.

C++ provides two types of comment delimiters:

- Single Line Comment: Anything after // {double forward slash} (until the end of the line on which it appears) is considered a comment.
    - E.g.: cout<<var1; //this line prints the value of var1
- Multiple Line Comment: Anything enclosed by the pair /* and */ is considered a comment.

E.g.:

```
/*this is a kind of comment where
Multiple lines can be enclosed in
one C++ program */
```

Comments should be used to enhance (not to hinder) the readability of a program. The following points, in particular, should be noted:

- A comment should be easier to read and understand than the code which it tries to explain. A confusing or unnecessarily-complex comment is worse than no comment at all.
- Over-use of comments can lead to even less readability. A program which contains so much comment that you can hardly see the code can by no means be considered readable.
- Use of descriptive names for variables and other entities in a program, and proper indentation

2.4.3. Statements

Statements represent the lowest-level building blocks of a program. Roughly speaking, each statement represents a computational step which has a certain side-effect. (A side-effect can be thought of as a change in the program state, such as the value of a variable changing because of an assignment.) Statements are useful because of the side-effects they cause, the combination of which enables the program to serve a specific purpose. A running program spends all of its time executing statements. The order in which statements are executed is called flow control (or control flow). This term reflect the fact that the currently executing statement has the control of the CPU, which when completed will be handed over (flow) to another statement. Flow control in a program is typically sequential, from one statement to the next, but may be diverted to other paths by branch statements.

Flow control is an important consideration because it determines what is executed during a run and what is not, therefore affecting the overall outcome of the program. Like many other procedural languages, C++ provides different forms of statements for different purposes. Declaration statements are used for defining variables. Assignment-like statements are used for simple, algebraic computations. Branching statements are used for specifying alternate paths of execution, depending on the outcome of a logical condition. Loop statements are used for specifying computations which need to be repeated until a certain logical condition is satisfied. Flow control statements are used to divert the execution path to another part of the program.

- In C++, a statement controls the sequence of execution, evaluates an expression, or does nothing (the null statement).
- All C++ statements end with a semicolon.

E.g.: x = a + b; //The meaning is: assign the value of the sum of a and b to x.

- White spaces: white spaces characters (spaces, tabs, new lines) can't be seen and generally ignored in statements. White spaces should be used to make programs more readable and easier to maintain.
- Blocks: a block begins with an opening French brace ({) and ends with a closing French brace (}).

2.4.4. Identifiers

An identifier is name associated with a function or data object and used to refer to that function or data object. An identifier must:

- Start with a letter or underscore
- Consist only of letters, the digits 0-9, or the underscore symbol _
- Not be a reserved word

For the purposes of C++ identifiers, the underscore symbol, _, is considered to be a letter. Its use as the first character in an identifier is not recommended though, because many library functions in C++ use such identifiers. Similarly, the use of two consecutive underscore symbols, _ _, is forbidden.

The following are valid identifiers

Length	days_in_year	DataSet1	Profit95
Int	_Pressure	first_one	first_1

Although using _Pressure is not recommended.

The following are invalid:

days-in-year	1data	int	first.val
throw	my__best	No##	bestWish!

Although it may be easier to type a program consisting of single character identifiers, modifying or correcting the program becomes more and more difficult. The minor typing effort of using meaningful identifiers will repay itself many fold in the avoidance of simple programming errors when the program is modified.

2.4.5. Keywords

Reserved/Key words have a unique meaning within a C++ program. These symbols, the reserved words, must not be used for any other purposes. All reserved words are in lowercase letters. The following are some of the reserved words of C++.

Table 2.2. Keywords in C++

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

Notice that main is not a reserved word. However, this is a fairly technical distinction, and for practical purposes you are advised to treat main, cin, and cout as if they were reserved as well.

2.5. Operators and Operator Precedence

Operators

- An operator is a symbol that makes the machine to take an action.
- Different Operators act on one or more operands and can also have different kinds of operators.
- C++ provides several categories of operators, including the following:

Assignment operator (=).

- ✓ The assignment operator causes the operand on the left side of the assignment statement to have its value changed to the value on the right side of the statement.
- ✓ Syntax: Operand1=Operand2;
- ✓ Operand1 is always a variable
- ✓ Operand2 can be one or combination of:

• A literal constant: Eg: x=12;

• A variable: Eg: x=y;

• An expression: Eg: x=y+2;

Compound assignment operators (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=).

- ✓ Compound assignment operator is the combination of the assignment operator with other operators like arithmetic and bit wise operators.

- ✓ The assignment operator has a number of variants, obtained by combining it with other operators. E.g.: value += increase; is equivalent to value = value + increase;

a -= 5; is equivalent to a= a– 5;

a /= b; is equivalent to a= a/ b;

price *= units + 1 is equivalent to price = price * (units + 1);

- ✓ And the same is true for the rest.

Arithmetic operators (+, -, *, /, %).

- ✓ Except for remainder or modulo (%), all other arithmetic operators can accept a mix of integers and real operands. Generally, if both operands are integers then, the result will be an integer. However, if one or both operands are real then the result will be real.
- ✓ When both operands of the division operator (/) are integers, then the division is performed as an integer division and not the normal division we are used to.
- ✓ Integer division always results in an integer outcome.
- ✓ Division of integer by integer will not round off to the next integer

E.g.:

9/2 gives 4 not 4.5

-9/2 gives -4 not -4.5

- ✓ To obtain a real division when both operands are integers, you should cast one of the operands to be real.   E.g.: int cost = 100;

Int volume = 80;

Double unitPrice = cost/(double)volume;

- ✓ The module(%) is an operator that gives the remainder of a division of two integer values. For instance, 13 % 3 is calculated by integer dividing 13 by 3 to give an outcome of 4 and a remainder of 1; the result is therefore 1.

E.g.: a = 11 % 3 ,a is 2

Relational operator (==, !=, > , <, >=, <=).

- ✓ In order to evaluate a comparison between two expressions, we can use the relational operator.
- ✓ The result of a relational operator is a bool value that can only be true or false according to the result of the comparison. E.g.:

(7 = = 5) would return false or returns 0

(5 > 4) would return true or returns 1

- ✓ The operands of a relational operator must evaluate to a number. Characters are valid operands since they are represented by numeric values. For E.g.:

'A' < 'F' would return true or 1. it is like (65 < 70)

Logical Operators (!,&&, ||):

- ✓ Logical negation (!) is a unary operator, which negates the logical value of its operand. If its operand is non zero, it produce 0, and if it is 0 it produce 1.
- ✓ Logical AND (&&) produces 0 if one or both of its operands evaluate to 0 otherwise it produces 1.
- ✓ Logical OR (||) produces 0 if both of its operands evaluate to 0 otherwise, it produces 1. E.g.:

!20 //gives 0

10 && 5 //gives 1

10 || 5.5 //gives 1

10 && 0 // gives 0

N.B. In general, any non-zero value can be used to represent the logical true, whereas only zero represents the logical false.

Increment/Decrement Operators: (++) and (--)

The auto increment (++) and auto decrement (--) operators provide a convenient way of, respectively, adding and subtracting 1 from a numeric variable. E.g.:

if a was 10 and if a++ is executed then a will automatically changed to 11.

Prefix and Postfix:

- ✓ The prefix type is written before the variable. Eg (++ myAge), whereas the postfix type appears after the variable name (myAge ++).
- ✓ Prefix and postfix operators can not be used at once on a single variable:

E.g.: ++age-- or --age++ or ++age++ or - - age - - is invalid

- ✓ In a simple statement, either type may be used. But in complex statements, there will be a difference.
- ✓ The prefix operator is evaluated before the assignment, and the postfix operator is evaluated after the assignment.

E.g. int k = 5;

(auto increment prefix) y= ++k + 10; //gives 16 for y

(auto increment postfix) y= k++ + 10; //gives 15 for y

(auto decrement prefix) y= --k + 10; //gives 14 for y

(auto decrement postfix) y= k-- + 10; //gives 15 for y

Conditional Operator (?:)

The conditional operator takes three operands. It has the general form:

Syntax:

operand1 ?operand2 : operand3

- ✓ First operand1 is a relational expression and will be evaluated. If the result of the evaluation is nonzero (which means TRUE), then operand2 will be the final result. Otherwise, operand3 is the final result.

E.g.: General Example

Z=(X<Y? X : Y)

This expression means that if X is less than Y the value of X will be assigned to Z otherwise (if X>=Y) the value of Y will be assigned to Z.

E.g.:

int m=1,n=2,min;

min = (m < n? m : n);

The value stored in min is 1.

E.g.:

(7 = = 5 ? 4: 3) returns 3since 7 is not equal to 5

Comma Operator (,).

- ✓ Multiple expressions can be combined into one expression using the comma operator.
- ✓ The comma operator takes two operands. Operand1,Operand2
- ✓ The comma operator can be used during multiple declaration, for the condition operator and for function declaration, etc
- ✓ It the first evaluates the left operand and then the right operand, and returns the value of the latter as the final outcome.

E.g.

int m,n,min;

int mCount = 0, nCount = 0;

min = (m < n ? (mCount++ , m) : (nCount++ , n));

- ✓ Here, when m is less than n, mCount++ is evaluated and the value of m is stored in min. otherwise, nCount++ is evaluated and the value of n is stored in min.

The sizeof() Operator.

- ✓ This operator is used for calculating the size of any data item or type.
- ✓ It takes a single operand (e.g. 100) and returns the size of the specified entity in bytes. The outcome is totally machine dependent. E.g.:

a = sizeof(char)

b = sizeof(int)

c = sizeof(1.55) etc

Explicit type casting operators.

- ✓ Type casting operators allows you to convert a datum of a given type to another data type.

E.g.

int i;

float f = 3.14;

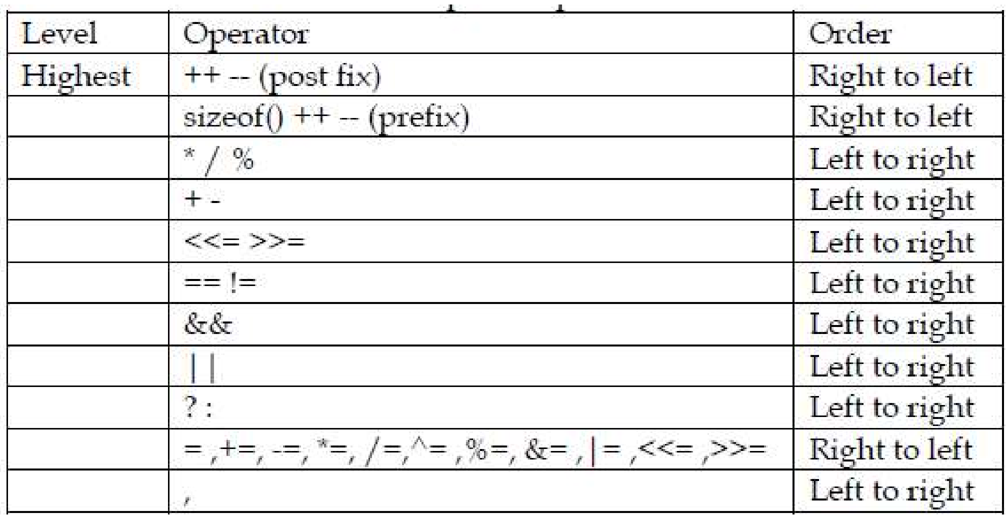
i = (int)f; equivalent to i = int(f);

Then variable i will have a value of 3 ignoring the decimal point

Operator Precedence

The order in which operators are evaluated in an expression is significant and is determined by precedence rules. Operators in higher levels take precedence over operators in lower levels.

Table 2.3.operator precedence



Level	Operator	Order
Highest	++ -- (post fix)	Right to left
	sizeof() ++ -- (prefix)	Right to left
	* / %	Left to right
	+ -	Left to right
	<<= >>=	Left to right
	== !=	Left to right
	&&	Left to right
	||	Left to right
	? :	Left to right
	= ,+=, -=, *=, /=,^= ,%=, &= ,|= ,<<= ,>>=	Right to left
	,	Left to right

E.g.

a = = b + c * d

c * d is evaluated first because * has a higher precedence than + and = =.

The result is then added to b because + has a higher precedence than = =

And then == is evaluated.

- ✓ Precedence rules can be overridden by using brackets.

E.g. rewriting the above expression as:

a = = (b + c) * d causes + to be evaluated before *.

- ✓ Operators with the same precedence level are evaluated in the order specified by the column on the table of precedence rule.

E.g. a = b += c the evaluation order is right to left, so the first b += c is evaluated followed by a = b.

2.6. Debugging and Programming Errors

Programming is a complex process, and since it is done by human beings, it often leads to errors.This makes debugging a fundamental skill of any programmer as debugging is an intrinsic part of programming. Programming errors are called bugs and going through the code, examining it and looking for something wrong in the implementation (bugs) and correcting them is called debugging. Often times the program doesn’t work as planned, and won’t compile properly. Even the best programmers make mistakes, being able to identify what you did wrong is essential. There are two types of errors that exist; those that the C++ compiler can catch on its own, and those that the compiler can’t catch. Errors that C++ can catch are known as compiler-time errors. Compiler-time errors should be relatively easy to fix, because the compiler points you to where the problem is. All that garbage that’s spit out by the compiler has some use. Here’s an example.

```
1 int main()
2 {
3 return 0
4 }
```

Your compiler should generate an error something like… \main.cpp(3) : error C2143: syntax error : missing ';' before '}' Compiler errors differ from compiler to compiler, but it’s all going to generally be the same. Now lets take this compiler error apart. The first part of it \main.cpp(4) says that the error is in the file main.cpp, on line 4. After that is error C2143: That’s the compiler specific error code. After that the error states syntax error : Which tells you that you messed up some syntax. So you must not have typed something right. Then it tells you missing ‘;’ before ‘}’ There’s a missing semi-colon before a closing bracket. Acknowledging a compiler error should be as easy as that. The other type of error that C++ doesn’t catch is called a run-time error. Run-time errors are errors often much more tricky to catch.

There’s several ways that one can debug a program. The two that I use most often are the WRITE technique, and single-step debugging. The WRITE technique involves creating output statements for all of the variables, so you can see the value of everything.

For smaller program, the WRITE technique works reasonably well, but as things get larger, it’s harder to output all your variables, and it just starts to seem like a waste of time. Instead, we’ll rely on the debugger. A debugger is a tool built into most development environments (and although they differ, most debuggers work on the same principles.). A programmer controls the debugger through commands by the means of the same interface as the editor. You can access these commands in menu items or by using hotkeys. The debugger allows the programmer to control the execution of his/her program. He/she can execute one step at a time in the program, he/she can stop the program at any point, and he/she can examine the value of variables.

2.7. Basic Input and Output in C++, Formatted Input-Output

The most common way in which a program communicates with the outside world is through simple, character-oriented Input/Output (IO) operations. C++ provides two useful operators for this purpose: >> for input and << for output. The following example illustrates the use of >> for input and << for output.

```
#include <iostream.h>
int main (void)
{
```

```
4: int workDays = 5;
5: float workHours = 7.5;
6: float payRate, weeklyPay;
7:
8: cout<< "What is the hourly pay rate? ";
9: cin>>payRate;
10:
11: weeklyPay = workDays * workHours * payRate;
12: cout<< "Weekly Pay = ";
13: cout<<weeklyPay;
14: cout<< '\n';
15:}
```

Line 8 outputs the prompt 'What is the hourly pay rate? ' to seek user input. Line 9 reads the input value typed by the user and copies it to payRate. The input operator >> takes an input stream as its left operand (cin is the standard C++ input stream which corresponds to data entered via the keyboard) and a variable (to which the input data is copied) as its right operand.

When run, the program will produce the following output (user input appears in bold):

```
What is the hourly payrate? 33.55

WeeklyPay=1258.125
```

Both << and >> return their left operand as their result, enabling multiple input or multiple output operations to be combined into one statement.

- Cout is an object used for printing data to the screen.
- To print a value to the screen, write the word cout, followed by the insertion operator also called output redirection operator (<<) and the object to be printed on the screen.
- Syntax: Cout<<Object;

- The object at the right hand side can be:

• A literal string: “Hello World”

• A variable: a place holder in memory

- Cin is an object used for taking input from the keyboard.
- To take input from the keyboard, write the word cin, followed by the input redirection operator (>>) and the object name to hold the input value.
- Syntax: Cin>>Object
- Cin will take value from the keyboard and store it in the memory. Thus the cin statement needs a variable which is a reserved memory place holder.
- Both << and >> return their right operand as their result, enabling multiple input or multiple output operations to be combined into one statement. The following example will illustrate how multiple input and output can be performed: E.g.:

Cin>>var1>>var2>>var3;

Here three different values will be entered for the three variables. The input should be separated by a space, tan or newline for each variable. ‘

Cout<<var1<<”, “<<var2<<” and “<<var3;

Here the values of the three variables will be printed where there is a “,” (comma) between the first and the second variables and the “and” word between the second and the third.

Sample C++ program

```
#include <iostream.h>

int main()
{
        cout << "Hello World!\n";
        return 0;
}
```

- Every C++ program has a *main()* function
- All functions begin with an opening brace ({) and end with a closing brace (})
- When your program starts, *main*() is called automatically
- *main*(), like all functions, must state what kind of value it will return

- cout is used in C++ to print strings and values to the screen
- ANSI-compliant programs declare main() to return an int to the operating system when your program completes. Some programmers signal an error by returning the value 1.

Keywords

- have a unique meaning within a C++ program
- must not be used for any other purposes
- All reserved words are in lower-case letters.

Identifiers

- An identifier is name associated with a function or data object and used to refer to that function or data object.
- Identifiers can be used to identify variable or constants or functions.
- An identifier must:
  - Start with a letter or underscore
  - Consist only of letters, the digits 0-9, or the underscore symbol _
  - Not be a reserved word

Variables

- A symbolic name for a memory location in which data can be stored and subsequently recalled
- Variables are used for holding data values so that they can be utilized in various computations in a program
- All variables have two important attributes:
  - ✓ A **type**, which is, established when the variable is defined (e.g., integer, float, character). Once defined, the type of a C++ variable cannot be changed
  - ✓ A value, which can be changed by assigning a new value to the variable

Variable Declaration

- Declaring a variable means defining (creating) a variable
- You create or define a variable by stating its type, followed by one or more spaces, followed by the variable name and a semicolon
- The variable name can be virtually any combination of letters, but cannot contain spaces and the first character must be a letter or an underscore
- Variable names cannot also be the same as keywords used by C++.
- Variables must be declared before used!
- C++ is case-sensitive - A variable named age is different from Age, which is different from AGE.
- You can create more than one variable of the same type in one statement by writing the type and then the variable names, separated by commas.
- The location of the declaration within the program determines the ***scope*** of the variable - the scope of a variable extends from its point of declaration to the end of the immediate block in which it is declared or which it controls.

Initializing variables

- You initialize a variable by using the assignment operator (=)
- initialization takes place at the moment you create the variable
- You can initialize more than one variable at creation.
- It is possible to even mix definitions and initializations
- Some compilers may simply leave “garbage” in the uninitialized variable, producing output like This …….
- The syntax is *specifier type name initializer;*

Assigning Values to Variables

- You assign a value to a variable by using the assignment operator (=)

Constants

- An object whose value cannot be changed
- Constants are declared by preceding its type specifier with the keyword **const**, like this: const int N = 22;
- Constants must be initialized when they are declared.
- Constants are usually defined for values like that will be used more than once in a program but not changed.
- It is customary to use all capital letters in constant identifiers to distinguish them from other kinds of identifiers

Literals

- Literals are constant values which can be a number, a character of a string. For example the number 129.005, the character ‘A’ and the string “hello world” are all literals. There is no identifier that identifies them.

Expressions

- a computation which yields a value. It can also be viewed as any statement that evaluates to a value (returns a value) E.g.: the statement 3+2; returns the value 5 and thus is an expression.
- ⏲ Some examples of an expression:

  E.g.1:

  3.2 returns the value 3.2
  PI float constant that returns the value 3.14 if the constant is defined.
  secondsPerMinute integer constant that returns 60 if the constant is declared

  E.g.2: complicated expressions:

  x = a + b;
  y = x = a + b;

The second line is evaluated in the following order:
1. add a to b.
2. assign the result of the expression a + b to x.
3. assign the result of the assignment expression x = a + b to y.

Statements

- Represent the lowest-level building blocks of a program
- each statement represents a computational step
- The order in which statements are executed is called flow control (or control flow)
- Different forms of statements:
  - ✓ Declaration statements - for defining variables
  - ✓ Assignment like statements - are used for simple, algebraic computations
  - ✓ Branching statements –are used for specifying alternate paths of execution, depending on the outcome of a logical condition.
  - ✓ Loop statements - for specifying computations which need to be repeated until a certain logical condition is satisfied
  - ✓ Flow control statements – to divert the execution path to another part of the program

Comment

- is a piece of descriptive text which explains some aspect of a program
- are totally ignored by the compiler and are only intended for human readers
- C++ provides two types of comment delimiters:
  - ✓ Anything after // (until the end of the line on which it appears) is considered a comment.
  - ✓ Anything enclosed by the pair /* and */ is considered a comment

Data types

- A data type is specified during variable declaration in order to tell the compiler how much room to set aside and what kind of value you want to store in your variable
- The data types supported by C++ can be classified as **basic** (fundamental) data types, **user defined** data types, **derived** data types and **empty** data types.

Basic data types:
- divided into **numeric** and **character** types
- Numeric variables can further be divided into integer variables and floating-point variables
- Both the numeric data types offer modifiers - **short**, **long**, **signed** and **unsigned**

Type	Size	Values
unsigned short int	2 bytes	0 to 65,535(0 to 2^16 -1)
short int(signed short int)	2 bytes	-32,768 to 32,767(-(2^16)/2 to (2^16/2) -1)
unsigned long int	4 bytes	0 to 4,294,967,295(0 to 2^32 -1)
long int(signed long int)	4 bytes	-2,147,483,648 to 2,147,483,647
Int	2 bytes	-32,768 to 32,767

unsigned int	2 bytes	0 to 65,535
signed int	2 bytes	-32,768 to 32,767
Char	1 byte	256 character values
Float	4 bytes	3.4e-38 to 3.4e38
Double	8 bytes	1.7e-308 to 1.7e308
long double	10 bytes	1.2e-4932 to 1.2e4932

Signed and Unsigned

- Signed integers are either negative or positive. Unsigned integers are always positive
- To any variable, do not assign a value that is beyond its range!

Characters and Numbers

- Character variables (type char) are typically 1 byte, enough to hold 256 values

Operators

- C++ provides operators for composing **arithmetic**, **relational**, **logical**, **bitwise**, and **conditional** expressions. Plus, **assignment**, **increment**, and **decrement** operators.

Chapter Three: Flow of Control

3.1 Introduction

A running program spends all of its time executing instructions or statements in that program. The order in which statements in a program are executed is called ***flow*** of that program. Programmers can control which instruction to be executed in a program, which is called ***flow control***. This term reflects the fact that the currently executing statement has the control of the CPU, which when completed will be handed over (flow) to another statement. Flow control in a program is typically ***sequential***, from one statement to the next. But we can also have execution that might be divided to other paths by ***branching statements***. Or perform a block of statement repeatedly until a condition fails by ***Repetition*** or ***looping***. Flow control is an important concept in programming because it will give all the power to the programmer to decide what to do to execute during a run and what is not, therefore, affecting the overall outcome of the program.

3.2 Boolean values

A boolean type is an integral type whose variables can have only two values: **false** and **true**. These values are stored as the integers **0** and **1**. The boolean type in Standard C++ is named **bool**.

Example, Boolean Variables

```
int main()
{ // prints the value of a boolean variable:
    bool flag=false;
    cout << "flag = " << flag << endl;
    flag = true;
    cout << "flag = " << flag << endl;
}
```

A ***boolean expression*** is a condition that is either true or false.

Example:-

```
if (n%d) cout << "n is not a multiple of d"; // (n%d) is a Boolean expression.
```

The output statement will execute precisely when n%d is not zero, and that happens precisely when d does not divide n evenly, because n%d is the remainder from the integer division.

3.3 Conditional statements

3.3.1 The *If* statement/selection structure

It is sometimes desirable to make the execution of a statement dependent upon a condition being satisfied. The if statement provides a way of expressing this, the general form of which is:

```
if (expression)
    statement;
```

The pseudocode statement:-

```
If student's grade is greater than or equal to 60
        Print "Passed"
```

determines whether the condition "student ' s grade is greater than or equal to 60" is true or f a l s e . If the condition is true, then "Passed" is printed and the next pseudocode statement in order i s "performed" (remember that pseudocode is not a real programming language). If the condition is false, the print statement is ignored and the next pseudocode statement in order i s performed. Note that the second line of this selection structure is indented. Such indentation is optional, but it is highly recommended because it emphasizes the inherent structure of structured program s. When you convert your pseudocode into C++ code, the C++ compiler ignores whitespace characters (like blanks, tabs and newlines) used for indentation and vertical spacing.

The preceding pseudocode *If* statement can be wri tten in C++ as

```
if ( grade > = 6 0 )
        cout « " Passed " ;
```

3.3.2 The *If/else* statement/selection structure

The if/else selection structure allows the programmer to specify an action to perform when the condition is true and a different action to perform when the condition is false. For example, the pseudocode statement

```
If student's grade is greater than or equal to 60
        Print "Passed "
else
        Print " Failed "
```

prints ***Passed*** if the student ' s grade is greater than or equal to 60, but prints ***Failed*** if the student' s grade is less than 60. I n either case, after printing occurs, the next pseudocode statement in sequence is "performed."

The preceding pseudocode if/else structure can be written in C++ as:

```
if ( grade > = 6 0 )
        cout « " Passed " ;
else
        cout « "Failed" ;
```

Note that the body of the else is also indented. Whatever indentation convention you choose should be applied consistently throughout your programs. It is difficult to read programs that do not obey uniform spacing conventions.

C++ provides the conditional operator (? :), which is closely related to the if/else structure. The conditional operator is C++'s only ternary operator-it takes three operands.

The above if/else statement can be replaced with the following line of code:-

```
cout « ( grade >= 60 ? " Passed " : " Failed " ) ;
```

3.3.3 **Nested *if/else*** statement/selection structure

Like compound statements, selection statements can be used wherever any other statement can be used. So a selection statement can be used within another selection statement. This is called nesting statements.

When **if..else** statements are nested, the compiler uses the following rule to parse the compound statement:

Match each **else** *with the last unmatched* **if**.

A frequently-used form of nested if statements involves the else part consisting of another if-else statement. For example: nested if..else statements to check if a character is a digit, upperletter, lower letter or special character:

```
int main{
    if (ch >= '0' && ch <= '9')
    kind = digit;
    else {
        if (ch >= 'A' && ch <= 'Z')
            kind = upperLetter;
        else {
            if (ch >= 'a' && ch <= 'z')
                kind = lowerLetter;
```

```
                        else
                            kind = special;
                    }
                }
            }
```

3.3.3 The *else if* Construct/structure

Nested if..else statements are often used to test a sequence of parallel alternatives, where only the else clauses contain further nesting. In that case, the resulting compound statement is usually formatted by lining up the else if phrases to emphasize the parallel nature of the logic.

The above nested if else code can also be written using the ***else if*** construct as shown below:

```
int main{
        if (ch >= '0' && ch <= '9')
            kind = digit;
        else if (cha >= 'A' && ch <= 'Z')
            kind = capitalLetter;
        else if (ch >= 'a' && ch <= 'z')
            kind = smallLetter;
        else
            kind = special;

      }
```

3.3.4 The **switch** Statement

Another C++ statement that implements a selection control flow is the switch statement (*multiple-choice statement)*. The switch statement provides a way of choosing between a set of alternatives based on the value of an expression. The switch statement has four components: -

- ***Switch***
- ***Case***
- ***Default***
- ***Break,*** Where Default and Break are Optional.

The General Syntax might be:

```
switch(expression)
  {
      case constant1:
             statements;
             •
             •
             •
      case constant n:
             statements;
      default:
             statements;
  }
```

First *expression* (called the switch **tag**) is evaluated, and the outcome is compared to each of the *constant*s (called case **labels**), in the order they appear, until a match is found. The *statements* following the matching case are then executed. Note the plural: each case may be followed by zero or more statements (not just one statement). Execution continues until either a break statement is encountered or all intervening statements until the end of the switch statement are executed. The final default case is optional and is exercised if none of the earlier cases provide a match.

For example, suppose we have parsed a binary arithmetic operation into its three components and stored these in variables operator, operand1, and operand2. The following switch statement performs the operation and stores the result in result.

```
switch (operator) {
    case '+':   result = operand1 + operand2;
                break;
    case '-':   result = operand1 - operand2;
                break;
    case '*':   result = operand1 * operand2;
                break;
    case '/':   result = operand1 / operand2;
                break;
    default:cout << "unknown operator: " << ch << '\n';
                break;
}
```

As illustrated by this example, it is usually necessary to include a break statement at the end of each case. The break terminates the switch statement by jumping to the very end of it. There are, however, situations in which it makes sense to have a case without a break. For example, if we extend the above statement to also allow x to be used as a multiplication operator, we will have:

```
switch (operator) {
    case '+':   result = operand1 + operand2;
                break;
    case '-':   result = operand1 - operand2;
                break;
    case 'x':
    case '*':   result = operand1 * operand2;
                break;
    case '/':   result = operand1 / operand2;
                break;
    default:cout << "unknown operator: " << ch << '\n';
                break;
}
```

Because case 'x' has no break statement (in fact no statement at all!), when this case is satisfied, execution proceeds to the statements of the next case and the multiplication is performed.

It should be obvious that any switch statement can also be written as multiple if-else statements. The above statement, for example, may be written as:

```
if (operator == '+')
    result = operand1 + operand2;
else if (operator == '-')
    result = operand1 - operand2;
else if (operator == 'x' || operator == '*')
    result = operand1 * operand2;
else if (operator == '/')
    result = operand1 / operand2;
else
    cout << "unknown operator: " << ch << '\n';
```

However, the switch version is arguably neater in this case. In general, preference should be given to the switch version when possible. The if-else approach should be reserved for situation where a switch cannot do the job

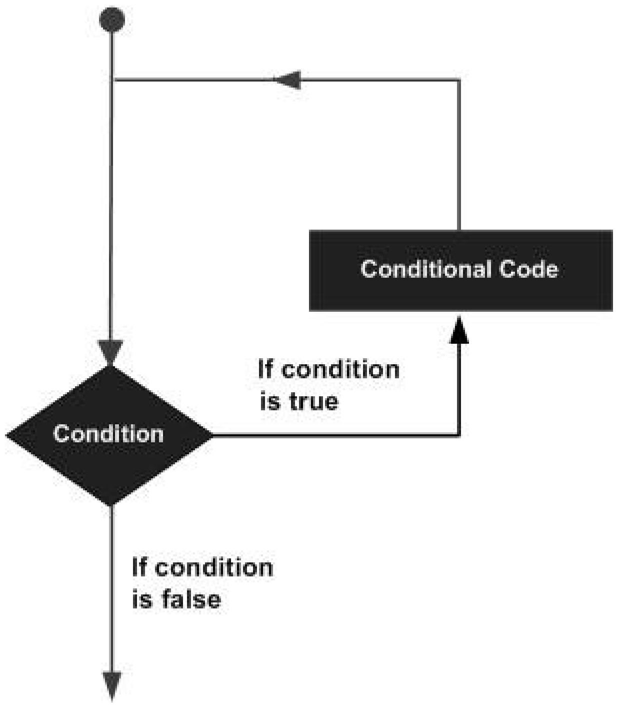
Chapter Four

Looping

Loop Basics

In general, statements are executed sequentially; the first statement in a function is executed first, followed by the second, and so on. Programming languages provide various control structures that allow for more complicated execution paths. There may be a situation, when you need to execute a block of code several times--***looping***.

A **loop statement** allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages:



There are two types of loops

Count controlled loops

- Repeat a statement or block a specified number of times
- Count-controlled loops contain
  - An initialization of the loop control variable
  - An expression to test if the proper number of repetitions has been completed
  - An update of the loop control variable to be executed with each iteration of the body

Event-controlled loops

- Repeat a statement or block until a condition within the loop body changes that causes the repetition to stop
- Types of Event-Controlled Loops

  1. Sentinel controlled:

  Keep processing data until a special value that is not a possible data value is entered to indicate that processing should stop

  2. End-of-file controlled:

  Keep processing data as long as there is more data in the file

  3. Flag controlled:

  Keep processing data until the value of a flag changes in the loop body

C++ programming language provides the following types of loop to handle looping requirements.

For Loop

The ***for*** statement (also called **for loop**) is similar to the while statement, but has two additional components: an expression which is evaluated only once before everything else, and an expression which is evaluated once at the end of each iteration. The general form of the for statement is:

`for` ***(expression$_1$; expression$_2$; expression$_3$)***
    statement;

(Where expression1 is *initialization*; expression2 is *condition*, and expression3 is *update. initialization, condition, and update are optional expressions, and statement is any executable statement.)*

The *initialization* expression is used to declare and/or initialize control variable(s) for the loop; it is evaluated first, before any iteration occurs. The *condition* expression is used to determine whether the loop should continue iterating; it is evaluated immediately after the initialization; if it is true, the statement is executed. The *update* expression is used to update the control variable(s); it is evaluated after the statement is executed. So the sequence of events that generate the iteration is:

1. evaluate the *initialization* expression;
2. if the value of the *condition* expression is false, terminate the loop;
3. execute the *statement*;
4. evaluate the *update* expression;
5. repeat steps 2–4.

The general for loop is equivalent to the following while loop:

```
expression₁;
while (expression₂) {
     statement;
     expression₃;
}
```

The most common use of for loops is for situations where a variable is incremented or decremented with every iteration of the loop. The following for loop, for example, calculates the sum of all integers from 1 to n.

```
int main()
 {
     int n;
     cout << "Enter a positive integer: ";
     cin >> n;
     long sum=0;
     for (int i=1; i <= n; i++)
         {sum += i;
         cout << "The sum of the first " << n << " integers is " << sum
     <<'\n';}
 }
```

This is preferred to the while-loop version we saw earlier. In this example, i is usually called the loop variable.

C++ allows the first expression in a for loop to be a variable definition. In the above loop, for example, i can be defined inside the loop itself:

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

Contrary to what may appear, the scope for i is not the body of the loop, but the loop itself. Scope-wise, the above is equivalent to:

```
inti;
for (i = 1; i<= n; ++i)
    sum += i;
```

Any of the three expressions in a for loop may be empty. For example, removing the first and the third expression gives us something identical to a while loop:

```
for (; i != 0;)      // is equivalent to:  while (i != 0)
    something;       //                         something;
```

Removing all the expressions gives us an infinite loop. This loop's condition is assumed to be always true:

```
for (;;)             // infinite loop
    something;
```

For loops with multiple loop variables are not unusual. In such cases, the **comma operator** is used to separate their expressions:

```
for (i = 0, j = 0; i + j < n; ++i, ++j)
    something;
```

Because loops are statements, they can appear inside other loops. In other words, loops can be **nested**. For example,

```
for (inti = 1; i<= 3; ++i)
    for (int j = 1; j <= 3; ++j)
        cout << '(' << i << ',' << j << ")\n";
```

produces the product of the set {1,2,3} with itself, giving the output:

```
(1,1)
(1,2)
(1,3)
(2,1)
(2,2)
(2,3)
(3,1)
(3,2)
(3,3)
```

Scope of variables

```
int main()
         {
             int n;
             cout << "Enter a positive integer: ";
             cin >> n;
             long sum=0;//global variable
             for (int i=1; i < n/2; i++) // the scope of this i is this loop
                 sum += i;
             for (int i=n/2; i <= n; i++) // the scope of this i is this loop
                 sum += i;
             cout << "The sum of the first " << n << " integers is "
             << sum << endl;
         }
```

Examples:

a)This program finds the maximum of a sequence of input numbers using **a Sentinel to Control a for Loop**:

```
int main()
{
    int n, max;
    cout << "Enter positive integers (0 to quit): ";
    cin >> n;
    for (max = n; n > 0; )
    {   if (n > max) max = n;
        cin >> n;
    }
    cout << "max = " << max << endl;
}
```

b) Using a Flag to Break Out of a Nest of Loops

```
int main()
{
    const int N=5;
    bool done=false;
    for (int i=0; i<N; i++)
    {for (int j=0; j<N && !done; j++)
    {for (int k=0; k<N && !done; k++)
        if (i+j+k>N) done = true;
        else cout << i+j+k << " ";
    cout << "* ";
    }
    cout << "." << endl; // inside the i loop, outside the j loop
    done = false;
    }
}
```

```
0 1 2 3 4 * 1 2 3 4 5 * 2 3 4 5 .
1 2 3 4 5 * 2 3 4 5 .
2 3 4 5 .
3 4 5 .
4 5 .
```

While Loop

The while statement (also called **while loop**) provides a way of repeating a statement while a condition holds. It is one of the three flavors of **iteration** in C++. The general form of the while statement is:

```
while (expression)
    statement;
```

First *expression* (called the **loop condition**) is evaluated. If the outcome is nonzero then *statement* (called the **loop body**) is executed and the whole process is repeated. Otherwise, the loop is terminated.

For example, suppose we wish to calculate the sum of all numbers from 1 to some integer denoted by n. This can be expressed as:

```
i = 1;
sum = 0;
while (i<= n)
    sum += i;
```

For n set to 5, Table 2.9 provides a trace of the loop by listing the values of the variables involved and the loop condition.

Table 5.1 **While loop trace**

Iteration	**i**	**n**	**i<= n**	**sum += i++**
First	1	5	1	1
Second	2	5	1	3
Third	3	5	1	6
Fourth	4	5	1	10
Fifth	5	5	1	15
Sixth	6	5	0	

It is not unusual for a while loop to have an empty body (i.e., a null statement). The following loop, for example, sets n to its greatest odd factor.

```
while (n % 2 == 0 && n /= 2)
    ;
```

Here the loop condition provides all the necessary computation, so there is no real need for a body. The loop condition not only tests that n is even, it also divides n by two and ensures that the loop will terminate should n be zero.

Do while Loop

The do statement (also called **do loop**) is similar to the while statement, except that its body is executed first and then the loop condition is examined. The general form of the do statement is:

```
do
    statement;
while(expression);
```

First *statement*is executed and then *expression* is evaluated. If the outcome of the latter is nonzero then the whole process is repeated. Otherwise, the loop is terminated.

The do loop is less frequently used than the while loop. It is useful for situations where we need the loop body to be executed at least once, regardless of the loop condition. For example, suppose we wish to repeatedly read a value and print its square, and stop when the value is zero. This can be expressed as the following loop:

```
do {
    cin>> n;
    cout<< n * n << '\n';
} while (n != 0);
```

Unlike the while loop, the do loop is never used in situations where it would have a null body. Although a do loop with a null body would be equivalent to a similar while loop, the latter is always preferred for its superior readability.

Exiting from a Loop

The continue statement

- The continue statement terminates the current iteration of a loop and instead jumps to the next iteration.
- It is an error to use the continue statement outside a loop.
- In while and do while loops, the next iteration commences from the loop condition.
  - In a "for" loop, the next iteration commences from the loop's third expression.

  E.g.:
  ```
  for(int n=10;n>0;n--)
  { if(n==5)
  continue; //causes a jump to n—
  cout<<n<< ",";
  }
  ```
- When the continue statement appears inside nested loops, it applies to the loop immediately enclosing it, and not to the outer loops. For example, in the following set of nested loops, the continue statement applies to the "for" loop, and not to the "while" loop.

  E.g.:
  ```
  while(more)
  {
  for(i=0;i<n;i++)
  { cin>>num;
  if(num<0)
  continue; //causes a jump to : i++
  }
  }
  ```

4.3.2. The *break* statement

- A break statement may appear inside a loop (***while***, ***do***, or ***for***) or a switch statement. It causes a jump out of these constructs, and hence terminates them.
- Like the continue statement, a break statement only applies to the "loop" or "switch" immediately enclosing it. It is an error to use the break statement outside a loop or a switch statement.

  E.g.:
  ```
  for(n=10;n>0;n--)
  { cout<<n<< ",";
  if(n = = 3)
  { cout<< "count down aborted!!";
  break;
  }
  }
  ```

4.3.3. The 'goto' Statement

The goto statement provides the lowest-level of jumping. It has the general form:

```
goto label;
```

where *label* is an identifier which marks the jump destination of goto. The label should be followed by a colon and appear before a statement within the same function as the goto statement itself. For example, the role of the break statement in the for loop in the previous section can be emulated by a goto:

```
for(n=10;n>0;n--){
        cout<<n<< ",";
        if(n = = 3) {
                cout<< "count down aborted!!";
                goto out;
        }
}
out:: cout<<"out of loop";
```