

# 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 tell the computer what to do. Computer requires programs to function, and a computer program 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 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 <sup>solves</sup> 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.*

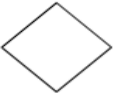



Pseudo code:

*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.

<u>Symbol</u>	<b>Name</b>	<b>Function</b>
	<b>Terminal</b>	Used to represent the start of the end of a program
	<b>Input/ output</b>	Used to represent data input or data output from a computer
	<b>Processing</b>	Usually encloses operations or (command block) a group of operations( a process)

	<b>Decision block</b>	it usually contains a question within it there are typically two output paths: one if the answer to the question is yes (true) , and the other if the answer is no ( false)
	<b>Flow line</b>	is used to indicate the direction of logical flow ( a path from one operation to another
	<b>On-page</b>	is used for connecting two points in connector a flow chart without drawing flow lines In one page.
	<b>Off page connector</b>	It is used an exit to or any entry from another part of the flowchart on another page

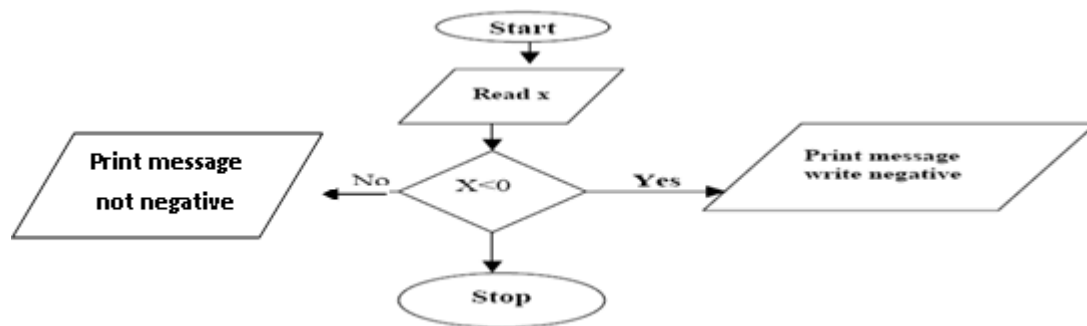
Example 2: Write an algorithm description and draw a flow chart to check a number is negative or not.

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.

$$\text{Sum} = 1+2+3+\dots + 50$$

Algorithmic description

1. Initialize sum too and counter to 1

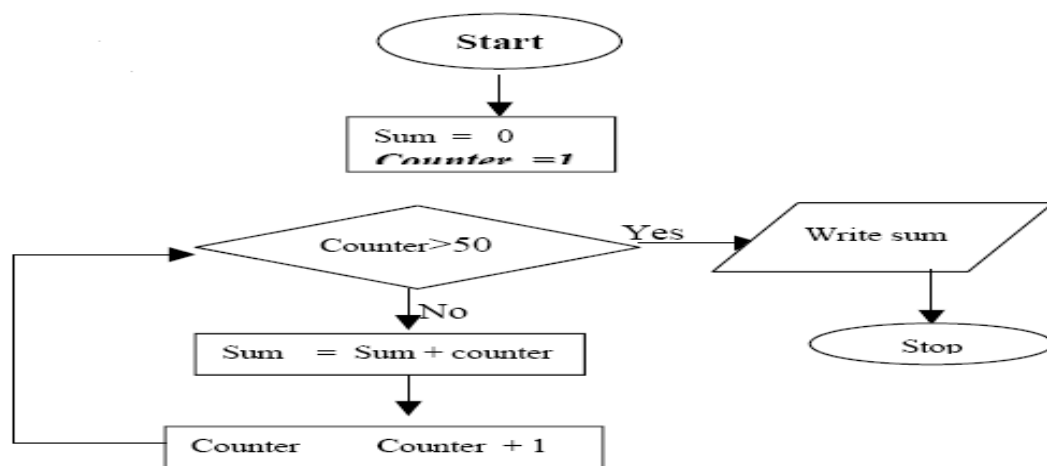
1.1. If the counter is less than or equal to 50

- 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 into 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 were 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

<i>Paradigm</i>	<i>Key concept</i>	<i>Program</i>	<i>Program execution</i>	<i>Result</i>
Imperative	Command (instruction)	Sequence of commands	Execution of commands	Final state of computer memory
Functional	Function	Collection of functions	Evaluation of functions	Value of the main function
Logic	Predicate	Logic formulas: axioms and a theorem	Logic proving of the theorem	Failure or Success of proving
Object-oriented	Object	Collection of classes of objects	Exchange of messages between the objects	Final state of the objects' states

## 1.6. The C++ Compilation Process

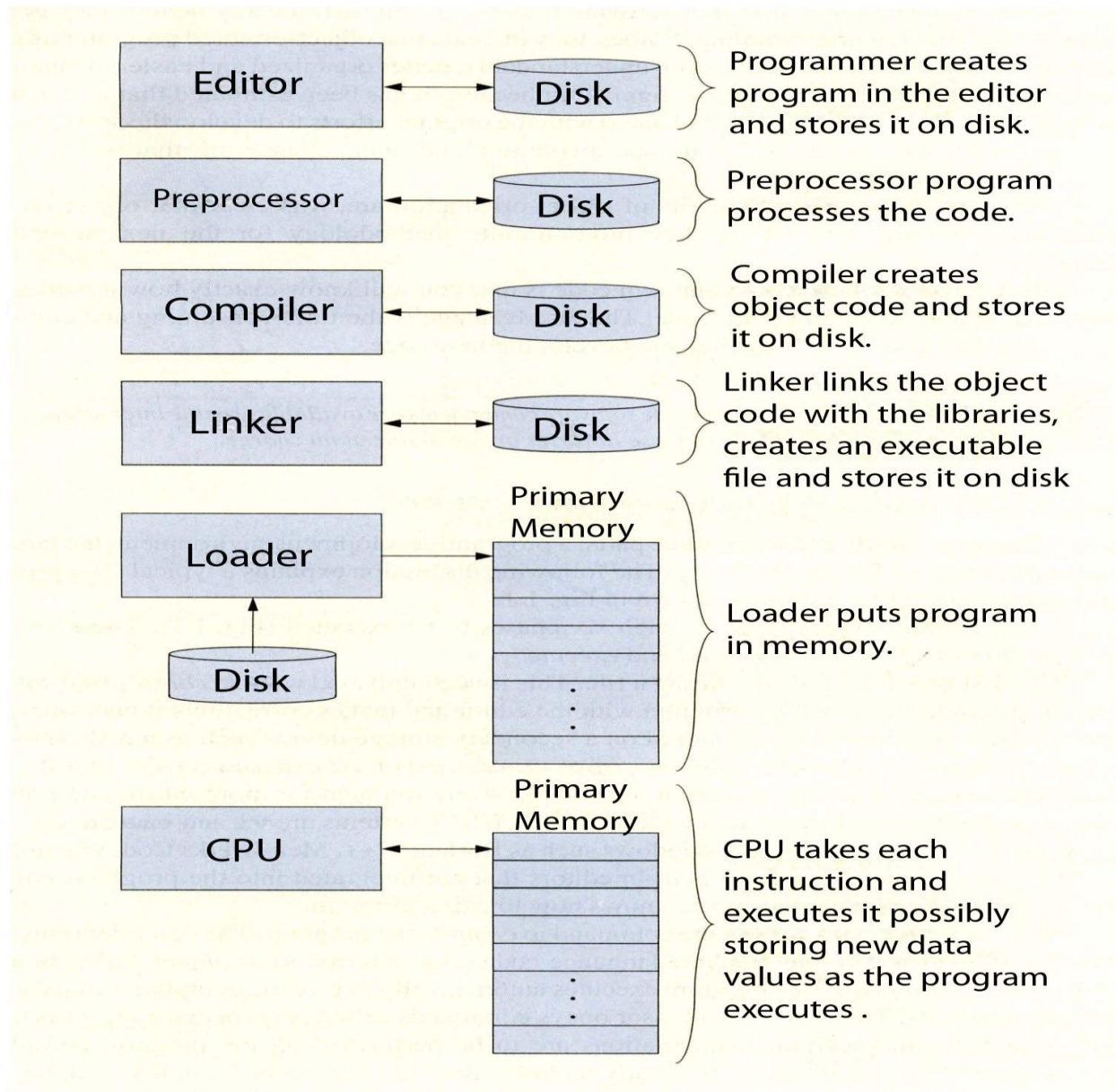
C++ systems generally consist of three parts: a program-**development environment, the language and the C++ Standard Library**. The following discussion explains a typical C++ program development environment.

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 the 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 implementation-dependent 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 specified in 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 << endl ;
```

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

```
cout << " Hello , " " John " << endl ;
```

The string " John " replaces **#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 `#line` 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:

Symbolic constant	Description
<code>__LINE__</code>	The line number of the current source code line (an integer constant).
<code>__FILE__</code>	The presumed name of the source file (a string).
<code>__DATE__</code>	The date the source file is compiled (a string of the form " <code>Mmm dd yyyy</code> " such as " <code>Aug 19 2002</code> ").
<code>__STDC__</code>	Indicates whether the program conforms to the ANSI C standard. Contains value 1 if there is full conformance and is undefined otherwise.
<code>__TIME__</code>	The time the source file is compiled (a string literal of the form " <code>hh:mm:ss</code> ").
<code>__TIMESTAMP__</code>	The date and time of the last modification to the source file (a string of the form " <code>Ddd Mmm Date hh:mm:ss yyyy</code> ", such as " <code>Mon Aug 19 12:01:55 2002</code> ").

### 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 an 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. If 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.