



Wolaita Sodo University
Department of Computer Science
Computer Programming Course Module

Prepared and Compiled by:

Dawit Uta (MTech.)

Reviewed by:

1. Arba Asha (MSc.), HoD, Department of Computer Science
2. Mesay Wana (MSc.)
3. Melaku Bayih (MSc.)

February, 2023

Contents

Chapter One: Introduction	4
1.1 Introduction to programming?	4
1.2 Problem solving techniques	4
1.3 Programming Languages	8
1.3.1 Generations of programming languages	9
1.3.2 Language Translators	10
1.3.3 C++ Program Compilation	13
Chapter Two: Basics of programming	16
2.1 The Structure of a C++ Program	16
2.2 C++ IDE	17
2.3 A brief look at cout and cin	18
2.4 Keywords, Identifiers, Comments, Inputs, Outputs, Parts of a program	18
2.5 Data Types	25
2.6 Variables	27
2.7 Constants	29
2.8 Operators	31
2.8.1 Assignment operator	31
2.8.2 Arithmetic operators	32
2.8.5 Relational Operators	32
2.8.6 Logical Operators	34
2.8.7 The sizeof() Operator	37
2.8.8 The Increment and Decrement Operators	37
2.8.9 Operator Precedence	38
2.9 Type Conversion	41
2.10 Programming Errors	43
Chapter Three: Control Statements	45
3.1 The if statement	45
3.2 The if/else statement	49
3.3 The if/else --- if statement	51
3.3.1 Using a trailing else	53
3.3.2 Nested if Statements	54

3.4	The switch statement.....	56
3.5	Introduction to Loops.....	62
3.5.1	The while Loop.....	62
3.5.2	The do-while Loop.....	72
3.5.3	The for Loop	75
3.5.4	The continue and goto Statement.....	86
Chapter 4: Function and Passing argument to function		92
4.1	Definitions of Functions.....	92
4.2	Declaration and calling functions	92
4.3	Function prototypes	95
4.4	Sending data into a function.....	99
4.5	Passing data by value	102
4.6	Using reference variables as parameters.....	103
4.7	Recursion	110
4.7.1	The recursive factorial function	112
4.7.2	The recursive GCD function	113
Chapter 5: Arrays, Pointers & Strings		115
5.1	Introduction.....	115
5.2	One Dimensional Array	116
5.2.1	Accessing Array Elements.....	117
5.2.2	Initialization of arrays	118
5.3	Multidimensional arrays	122
5.4	Address and pointer	123
5.5	Pointer and array	125
5.6	Pointer and String	126
5.7	Strings representation and manipulation.....	127
5.7.1	String Output.....	128
5.7.2	String Input	128
5.7.3	Avoiding buffer over flow	130
5.7.4	String constants.....	130
5.7.5	Copying string.....	131
5.7.6	Concatenating strings.....	131
5.7.7	Comparing strings.....	133

5.8	Dynamic memory:.....	133
Chapter 6: Structures		137
6.1	Specifying simple structure.....	137
6.2	Initializing and accessing structure variable	138
6.3	Arrays of Structures and accessing array of structures	139
Chapter Seven: File and File Management		143
7.1	Introduction.....	143
7.2	Stream	143
7.3	Operation with File.....	143
7.4	Types of Disk File Access.....	144
7.4.1	Sequential File Concepts	144
7.4.2	Random Access File Concepts	153
7.5	Command Line Argument.....	157
References		159

Chapter One: Introduction

1.1 Introduction to programming?

Programming is a skill that can be acquired by a computer professional that gives him/her the knowledge of making the computer perform the required operation or task.

Why do we need to learn computer programming?

Computer programming is critical if one wants to know how to make the computer perform a task. Most users of a computer only use the available applications on the computer. These applications are produced by computer programmers. Thus if someone is interested to make such kind of applications, he/she needs to learn how to talk to the computer, which is learning computer programming.

Programming refers to the process of making a machine or an object performs a particular task by giving it the necessary instructions (programs) and inputs.

It is the process of solving a problem using computers.

A program is a detailed set of instructions written in a programming language that directs the computer to solve a problem. For the instructions to be carried out, a computer must execute a program, that is, the computer reads the program, and then follows the steps encoded in the program in a precise order until completion. Back in the real world, the following examples show how program instructions are used to solve problems.

Consider a recipe book. Each recipe is a program of its own. It has a list of ingredients (the input data) and a list of instructions detailing exactly what to do with those ingredients. When you follow a recipe, you are executing that program.

Therefore computers solve problems by following the program instructions in the method of solution. However, we need to write a program that consists of instructions corresponding to the steps of the solution such that the computer can interpret and execute them.

What is programming language?

Programming Language: is a set different category of written symbols that instruct computer hardware to perform specified operations required by the designer.

What skills do we need to be a programmer?

For someone to be a programmer, in addition to basic skills in computer, needs to have the following major skills:

- **Programming Language Skill:** knowing one or more programming language to talk to the computer and instruct the machine to perform a task.
- **Problem Solving Skill:** skills on how to solve real world problem and represent the solution in understandable format.
- **Algorithm Development:** skill of coming up with sequence of simple and human understandable set of instructions showing the step of solving the problem. Those set of steps should not be dependent on any programming language or machine.

1.2 Problem solving techniques

A problem can be defined as a question or situation that presents uncertainty, perplexity, or difficulty. In this unit we will try to clarify the meaning of the term problem solving, identify

the steps involved, define a methodology for problem solving, and see how we can make use of the computer as a tool in the process.

For example: The retail shop's sales function. Selling a product in the shop is the task here. That is the problem is to sell or how a sales person sells a product in the shop. For a sale to happen in a shop, when a buyer brings products to the sales counter, you should go through the following:

- ✓ First, you should determine what the unit price of the product is.
- ✓ After that you have to count the quantity, (how many of the product) and
- ✓ Calculate the total amount to be paid by the customer.
- ✓ Then tell the total price, get the money from the customer and
- ✓ Then calculate the change that you have to give back to the customer.

Problem Solving Using Computers

Computers can be programmed to do many complicated tasks at very high speeds. Also, they can store large quantities of information. Therefore, we can use computers as a tool in problem solving if one or more of the following conditions for a problem hold true:

- ✓ It has extensive input.
- ✓ It has extensive output.
- ✓ Its method of solution is too complicated to implement manually.
- ✓ If done manually, it takes an excessively long time to solve.
- ✓ We expect to use the same method of solution often in the future to solve the same problem with different inputs.

Techniques of problem solving are:




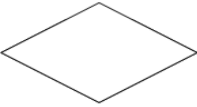


i. Pseudo code

- ✓ Pseudocode is a semiformal, English-like language with a limited vocabulary that can be used to design and describe algorithms.
- ✓ The main purpose of a pseudocode is to define the procedural logic of an algorithm in a simple, easy-to-understand manner for its readers, who may or may not be proficient in computer programming languages.
- ✓ A pseudocode language is a more appropriate algorithm description language than any programming language.
- ✓ A pseudocode can be used for:
 - Designing algorithms
 - Communicating algorithms to users
 - Debugging logic errors in program
 - Documenting programs for future maintenance and expansion purposes

ii. Flowcharting:

A flowchart is a graph consisting of geometrical shapes that are connected by flow lines.

The geometrical shapes in a flowchart represent the types of statements in an algorithm. The details of statements are written inside the shapes.

Symbol	Name	Use
	Terminal	Indicates the beginning and end of a program
	Process	A calculation or assigning of a value to a variable
	Input/Output (I/O)	Any statement that causes data to be input to a program (INPUT, READ) or output from the program such as printing on the display screen or printer
	Decision	Program decisions. Allows alternate courses of action based on a condition. A decision indicates a question that can be answered yes or no (or true or false)
	Connector	Can be used to eliminate lengthy flow lines. Its use indicates that one symbol is connected to another
	Flow lines	Used to connect symbols and indicate the sequence of operations. The flow is assumed to go from top to bottom and from left to right.

Examples:

Example-1: For the previous example, the retail's shop sales:

Pseudocode:

Start

Input Unit Price, Quantity, Customer Paying Amount

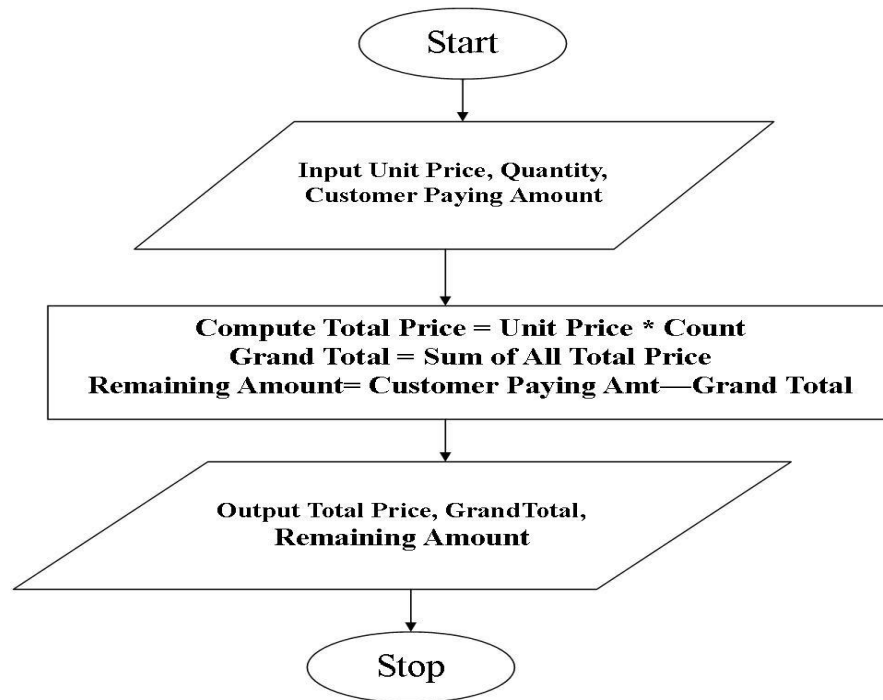
Compute Total Price = Unit Price * Count

Compute Grand Total = Sum of All Total Price

Compute Remaining Amount= Customer Paying Amt—Grand Total

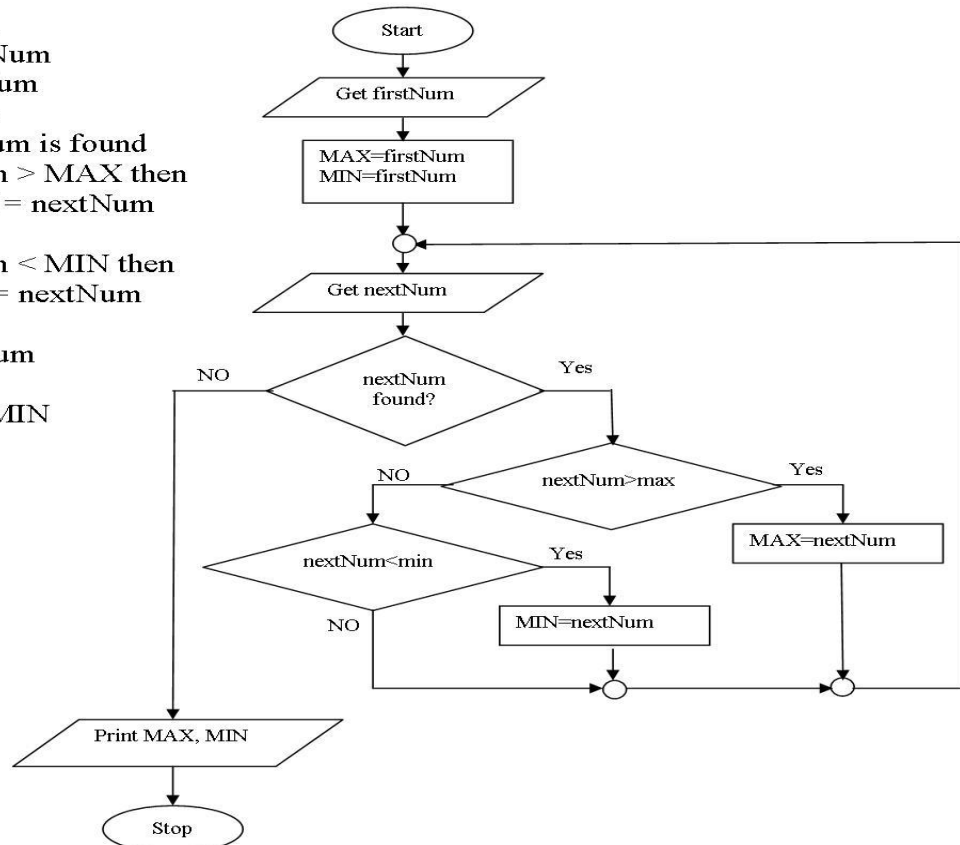
Output Total Price, Grand Total, Remaining Amount

Stop

Flowchart:**Example-2: Algorithm for computing the minimum and the maximum values****Pseudocode:**

```

Start
Get firstNum
MAX = firstNum
MIN = firstNum
Get nextNum
While nextNum is found
    If nextNum > MAX then
        MAX = nextNum
    EndIf
    If nextNum < MIN then
        MIN = nextNum
    EndIf
    Get nextNum
EndWhile
Print MAX, MIN
Stop
  
```

Flowchart:

- iii. **Algorithm:** An algorithm is a sequence of a finite number of steps arranged in a specific logical order which is used to develop the solution for a problem
- ✓ An algorithm must satisfy the following requirements:
 - **Unambiguousness:** It must not be ambiguous. In most cases we develop an algorithm so that it can be translated into a program to be executed by computers. Computers cannot cope with ambiguities. Therefore, every step in an algorithm must be clear as to what it is supposed to do and how many times it is expected to be executed.
 - **Generality:** It must have generality. A procedure that prints the message "One inch is 2.54 centimeters" is not an algorithm; however, one that converts a supplied number of inches to centimeters is an algorithm.
 - **Correctness:** It must be correct and must solve the problem for which it is designed.
 - **Finiteness:** It must execute its steps and terminate in finite time. An algorithm that never terminates is unacceptable.
 - ✓ In general, algorithms have input and produce some output. It is conceivable that an algorithm may have no input. An example could be one that generates a fixed report form. However, such algorithms are rather trivial and quite rare.
 - ✓ Several techniques have been developed expressly for the representation of algorithms. Pseudocodes and flowcharts can be used to develop algorithms.

1.3 Programming Languages

Early programming languages were designed for specific kinds of tasks. Modern languages are more general-purpose. In any case, each language has its own characteristics, vocabulary, and syntax.

- ✓ A programming language is an artificial language designed to express computations that can be performed by a machine, particularly a computer. Programming languages can be used to create programs that control the behavior of a machine or to express algorithms precisely.
- ✓ Many programming languages have some form of written specification of their syntax (form) and semantics (meaning).

Types [Levels] of Programming Languages

There is only one programming language that any computer can actually understand and execute: its own native binary machine language. This is the lowest possible level of language in which it is possible to write a computer program. All other languages are said to be high level or low level according to how closely they can be said to resemble machine code.

1. *Low-level languages*

- ✓ Low-level languages have the advantage that they can be written to take advantage of any peculiarities in the architecture of the central processing unit (CPU) which is the "brain" of any computer. Thus, a program written in a low-level language can be extremely efficient, making optimum use of both computer memory and processing time.
- ✓ However, to write a low-level program takes a substantial amount of time, as well as a clear understanding of the inner workings of the processor itself. Therefore, low-level programming is typically used only for very small programs, or for segments of code that are highly critical and must run as efficiently as possible.

Characteristics of LOW Level Languages:

- They are machine oriented: an assembly language program written for one machine will not work on any other type of machine unless they happen to use the same processor chip.
- Each assembly language statement generally translates into one machine code instruction, therefore the program becomes long and time-consuming to create.

2. High-level languages

- ✓ High-level languages permit faster development of large programs.
- ✓ The final program as executed by the computer is not as efficient, but the savings in programmer time generally far outweigh the inefficiencies of the finished product. This is because the cost of writing a program is nearly constant for each line of code, regardless of the language.

Characteristics of HIGH Level Languages:

- They are not machine oriented: they are portable, meaning that a program written for one machine will run on any other machine for which the appropriate compiler or interpreter is available.
- They are problem oriented: most high level languages have structures and facilities appropriate to a particular use or type of problem. For example, FORTRAN was developed for use in solving mathematical problems. Some languages, such as PASCAL were developed as general-purpose languages.
- Statements in high-level languages usually resemble English sentences or mathematical expressions and these languages tend to be easier to learn and understand than assembly language.
- Each statement in a high level language will be translated into several machine code instructions.

1.3.1 Generations of programming languages

1st Generation - Machine language

In the early days of computer programming all programs had to be written in machine code. For example a short (3 instruction) program might look like this:

0111 0001 0000 1111

1001 1101 1011 0001

1110 0001 0011 1110

- ✓ Machine code has several significant disadvantages associated it:
 - It is not intuitively obvious what a machine code instruction does simply from its encoding, consequently it is very difficult to read and write machine code.
 - The writing of machine code is extremely time consuming and error prone.
 - Many different machine codes exist (one for each make and type of computer).
- ✓ Machine code executes directly without translation since it is the actual pattern of 0s and 1s understood by the computer's memory.

2nd Generation - Assembly language

- ✓ Assembler languages were initially developed to address the disadvantages associated with machine code programming. They used symbolic codes instead lists of binary instructions.
- ✓ Consequently programming became more "friendly". An example of assembly code is given below:

```
MOV AX, 01
MOV BX, 02
ADD AX, BX
```

- ✓ In assembler language each line of the program corresponds to one instruction in machine code. For a program written in assembler language to be executable it must be translated into machine code using a translating program called an *assembler*.
- ✓ Although use of assembly languages offers some advantages there are still a number of significant disadvantages associated with their use:
 - Each model of computer has its own assembly language associated with it.
 - Assembler programming still requires great attention to detail and hence remains both time consuming and tedious.
 - Because of (2) the risk of program error is not significantly reduced.
- ✓ Note that there are some computer applications, such as interfacing with peripherals, where assembler language is still a necessity

3rd Generation - High Level Languages

- ✓ High Level languages are so-called because they are independent of the particular computer. They are not machine oriented.

4th Generation - Fourth Generation Languages (4GLS)

- ✓ 4GLs are the most modern high level languages today. The user must concentrate more on what is to be done to solve the problem while the how part is entirely (or most of it) left in the hands of the 4GL.

5th Generation:

- ✓ These are used in artificial intelligence (AI) and expert systems; also used for accessing databases.
- ✓ 5GLs are “natural” languages whose instruction *closely resembles* human speech. E.g. “get me Jone Brown’s sales figure for the 1997 financial year”.
- ✓ 5GLs require very powerful hardware and software because of the complexity involved in interpreting commands in human language.

1.3.2 Language Translators

- ✓ Any program written in a language other than machine language needs to be translated to machine language. The set of instructions that do this task are known as translators. All computer programs run on machine code that is executed directly on computer architecture. Machine code is not easily read or programmed directly by humans. Essentially, machine

code is a long series of bits (i.e. ones and zeroes). In order to program, humans write code in a language that is then translated in to machine code.

Source Code is the code that is input to a translator.

Executable code is the code that is output from the translator.

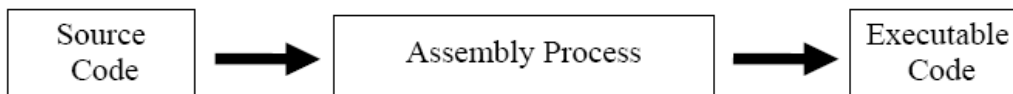
- ✓ There are 3 types of system software used for translating the code that a programmer writes into a form that the computer can execute (i.e. machine code). These are:
 1. Assemblers
 2. Compilers
 3. Interpreters

1. Assemblers

No matter how close assembly language is to machine code, the computer still cannot understand it. The assembly-language program must be translated into machine code.

Assembly languages are more easily translated in to machine code than high-level programs languages. Each assembly language statement directly corresponds to one or more machine instructions.

Another way to think about this is that assembly language code is simply an abbreviated form of machine code. Thus, to transform a program from an assembly language to machine code all that must be done is that the instructions must be converted from their mnemonic abbreviations into their equivalent string of ones and zeroes. This transformation process is known as *assembling* and is accomplished by an *assembler*.



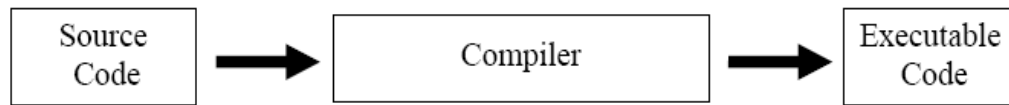
The assembler program recognizes the character strings that make up the symbolic names of the various machine operations, and substitutes the required machine code for each instruction.

The final result is a machine-language program that can run on its own at any time; the assembler and the assembly-language program are no longer needed. To help distinguish between the "before" and "after" versions of the program, the original assembly-language program is also known as the source code, while the final machine-language program is designated the object code.

If an assembly-language program needs to be changed or corrected, it is necessary to make the changes to the source code and then re-assemble it to create a new object program.

2. Compilers

Each instruction in the compiler language can correspond to many machine instructions. Once the program has been written, it is translated to the equivalent machine code by a program called a compiler. Once the program has been compiled, the resulting machine code is saved separately, and can be run on its own at any time.



- ✓ A compiler is a computer program that converts an entire program written in a high-level language (called *source code*) and translates it into an executable form (called *object code*).
- ✓ Updating or correcting a compiled program requires that the original (source) program be modified appropriately and then recompiled to form a new machine-language (object) program.
- ✓ Typically, the compiled machine code is less efficient than the code produced when using assembly language. This means that it runs a bit more slowly and uses a bit more memory than the equivalent assembled program. To offset this drawback, however, we also have the fact that it takes much less time to develop a compiler-language program, so it can be ready to go sooner than the assembly-language program. E.g. C++, Pascal, FORTRAN, etc.

Advantages of a Compiler

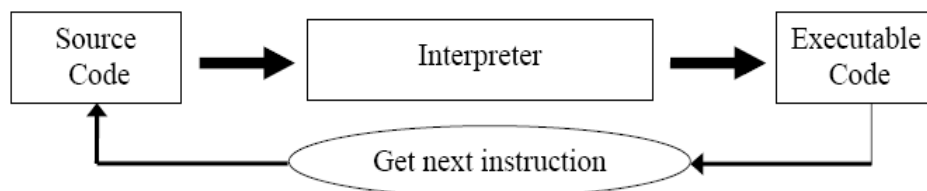
- Fast in execution
- The object/executable code produced by a compiler can be distributed or executed without having to have the compiler present.
- The object program can be used whenever required without the need to of re-compilation.

Disadvantages of a Compiler

- Debugging a program is much harder. Therefore not so good at finding errors
- When an error is found, the whole program has to be re-compiled

3. Interpreters

- ✓ An interpreter is a computer program that takes source code and converts each line in succession.



- ✓ At each step it executes the high-level statement. In other words, it doesn't have to examine the entire program before it can begin executing code.
- ✓ Thus, programs that are interpreted lend themselves to interactive programming. However, programs that are interpreted will generally run much slower than programs that are compiled.
- ✓ An Interpreter translates high-level source code into executable code. However the difference between a compiler and an interpreter is that **an interpreter translates one line at a time and then executes it**: no object code is produced, and so the program has to be

interpreted each time it is to be run. If the program performs a section code 1000 times, then the section is translated into machine code 1000 times since each line is interpreted and then executed. E.g. QBASIC, Lisp etc.

Advantages of an Interpreter

- Good at locating errors in programs
- Debugging is easier since the interpreter stops when it encounters an error.
- If an error is deducted there is no need to retranslate the whole program. This can enormously speed up the development and testing process.

Disadvantages of an Interpreter

- Because the interpreter has to scan the user's program one line at a time and execute internal portions of itself in response, execution of an interpreted program is much slower than for a compiled program.
- No object code is produced, so a translation has to be done every time the program is running.
- For the program to run, the Interpreter must be present. i.e. both the interpreter and the user's program reside in memory at the same time.

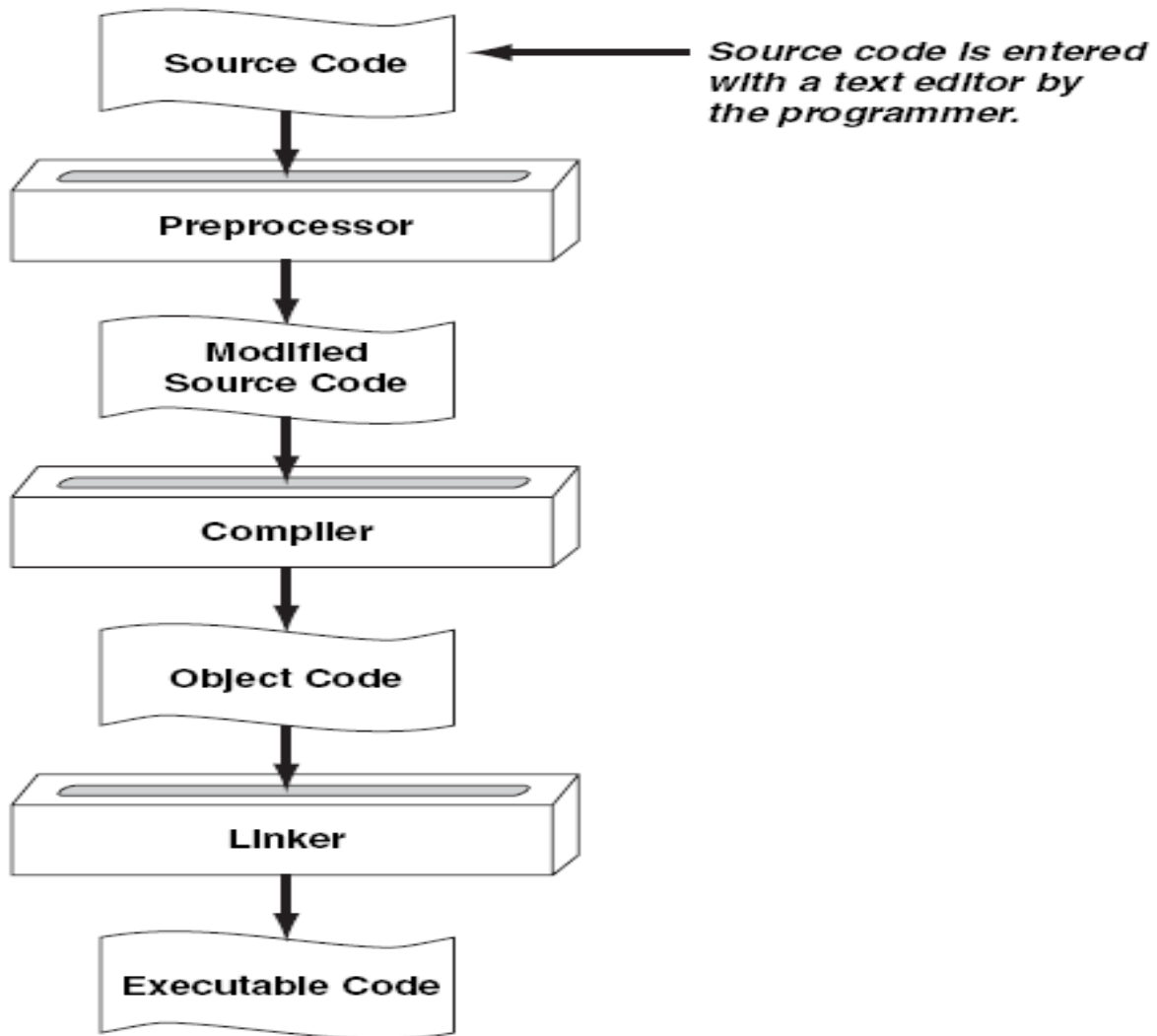
1.3.3 C++ Program Compilation

- ✓ When a C++ program is written, it must be typed into the computer and saved to a file. A *text editor*, which is similar to a word processing program, is used for this task. The statements written by the programmer are called **source code**, and the file they are saved in is called the **source file**. After the source code is saved to a file, the process of translating it to machine language can begin.
- ✓ **During the first phase of this process**, a program called the **preprocessor** reads the source code. The preprocessor searches for special lines that begin with the # symbol. These lines contain commands that cause the preprocessor to modify the source code in some way.
- ✓ **During the next phase** the **compiler** steps through the preprocessed source code, translating each source code instruction into the appropriate machine language instruction. This process will uncover any *syntax errors* that may be in the program. Syntax errors are illegal uses of key words, operators, punctuation, and other language elements. If the program is free of syntax errors, the compiler stores the translated machine language instructions, which are called **object code**, in an **object file**.
- ✓ Although an object file contains machine language instructions, it is not a complete program. Here is why: C++ is conveniently equipped with a library of prewritten code for performing common operations or sometimes-difficult tasks. For example, the library contains hardware-specific code for displaying messages on the screen and reading input from the keyboard. It also provides routines for mathematical functions, such as calculating the square root of a number. This collection of code, called the **run-time library**, is extensive. Programs almost always use some part of it. When the compiler generates an

object file, however, it does not include machine code for any run-time library routines the programmer might have used.

- ✓ ***During the last phase*** of the translation process, another program called the ***linker*** combines the object file with the necessary library routines. Once the linker has finished with this step, an ***executable file*** is created. The executable file contains machine language instructions, or ***executable code***, and is ready to run on the computer.

The following illustrates the process of translating a source file into an executable file.



Mechanics of creating a program (C++)

C++ programs typically go through *five phases* to be executed: these are *edit*, *preprocess*, *compile*, *link*, *load*:

Edit: this is accomplished with an editor program. The programmer types C++ statements with the editor and makes corrections if necessary.

The programs source file is then stored on *secondary storage device* such as a disk with a “.cpp” file name.

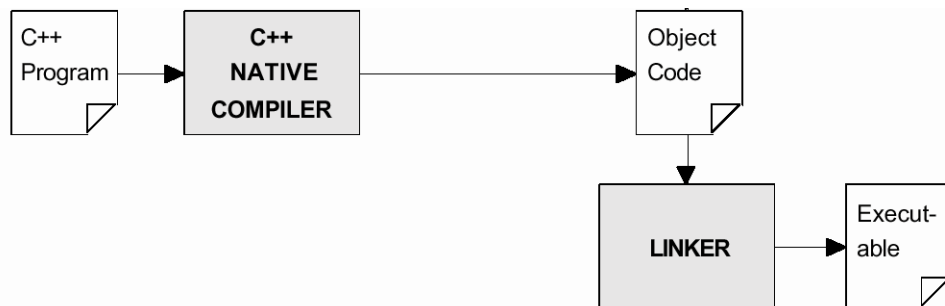
After the program is edited, C++ is principally compiled in three phases:

preprocessing, ***translation to object code***, and ***linking*** (the last two phases are what is generally thought of as the "compilation" process).

Preprocess: In a C++ system, a *preprocessor* program executes automatically before the compiler's translation phase begins. The C++ preprocessor obeys command 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 C++ **preprocessor** goes over the program text and carries out the instructions specified by the preprocessor directives (e.g., #include). The result is a modified program text which no longer contains any directives.

Compile: Then, the C++ compiler **translates** the program code. The compiler may be a true C++ compiler which generates native (assembly or machine) code. The outcome may be incomplete due to the program referring to library routines which are not defined as a part of the program. For example, the << operator which is actually defined in a separate IO library.

Linking: C++ programs typically contain references to functions and data defined elsewhere, such as in the standard 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 function to produce an *executable image* (with no missing pieces). Generally, the **linker** completes the object code by linking it with the object code of any library modules that the program may have referred to. The final result is an executable file ***Loading***: the loader takes the executable file 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.



In practice all these steps are usually invoked by a single command and the user will not even see the intermediate files generated.

Chapter Two: Basics of programming

2.1 The Structure of a C++ Program.

To understand the basic parts of a simple program in C++, let's have a look at the following code:

```
#include<iostream.h>
#include<conio.h>
void main()
{
    cout<<"\n Hello World!";
    getch();
}
```

Any C++ program file should be saved with file name extension “**.CPP**”

Type the program directly into the editor, and save the file as `hello.cpp`, compile it and then run it. It will print the words **Hello World!** on the computer screen.

The first character is the #. This character is a signal to the preprocessor. Each time you start your compiler, the preprocessor runs through the program and looks for the pound (#) symbols and act on those lines before the compiler runs. The **include** instruction is a preprocessor instruction that directs the compiler to include a copy of the file specified in the angle brackets in the source code. If the path of the file is not specified, the preprocessor looks for the file under `c:\tc\include\` folder or in **include** folder of the location where the editor is stored.

The effects of line 1, i.e. `include<iostream.h>` is to include the file `iostream.h` into the program as if the programmer had actually typed it. When the program starts, **main()** is called automatically. Every C++ program has a **main()** function.

The return value type for **main()** here is void, which means main function will not return a value to the caller (which is the operating system). The main function can be made to return a value to the operating system.

The Left French brace “{” signals the beginning of the main function body and the corresponding Right French Brace “}” signals the end of the main function body. Every Left French Brace needs to have a corresponding Right French Brace. The lines we find between the braces are statements or said to be the body of the function. A statement is a computation step which may produce a value or interact with input and output streams. The end of a single statement ends with semicolon (;).

The statement in the above example causes the sting “Hello World!” to be sent to the “cout” stream which will display it on the computer screen.

2.2 C++ IDE



















What is an IDE and why is it useful?

An IDE is a coding environment that includes both an editor and a language-specific toolchain. A typical IDE allows the user to edit source code and build executables. A good IDE has the editor and the additional tools set up in such a way as to make a developer more productive. IDEs have built-in functions like debugging, auto-completion, compilation, and syntax highlighting, all of which make programming easier and faster. For example, hitting a keyboard shortcut to compile and run a C++ project

is much more convenient than switching to a terminal emulator, running multiple commands on the command line, and switching back to the editor to locate a specific line causing an error. IDEs can be great time-savers for seasoned C++ developers. Newer programmers can benefit from IDEs because they simplify the development. The best IDEs may vary per language, so let's look specifically into the best C++ IDEs.

Integrated development environment Software / C++

From sources across the web

 Eclipse Common Public License	 Visual Studio Proprietary software	 Dev-C++ GNU General Public License
 Code::Blocks GNU General Public License	 CLion proprietary license	 CodeLite GNU General Public License
 NetBeans Freeware	 Microsoft Visual C++ Freeware	 Qt Creator GNU General Public License
 Xcode Freeware	 C++Builder Proprietary software	 Atom MIT License
 KDevelop GNU General Public License	 GNAT Programming Studio GNU General Public License	 Turbo C++ Freeware
 Anjuta	 MonoDevelop	 Borland C++

- ❖ **Code::blocks** is a cross-platform, free, and open-source IDE for C/C++ development that includes compiling, auto code compilation, code coverage, profiling, debugging.
- ❖ **Dev C++** is a fully-featured open-source IDE for C++. it only supports the Windows operating system. Despite this limitation, it includes support for GCC-based compilers like Cygwin and MinGW. International language support, code compilation, syntax-highlighting editor.
- ❖ **CodeLite** is an open source, free, cross platform IDE, specialized in C, C++, Rust, Python, PHP and JavaScript (mainly for backend developers using Node.js) programming languages which runs best on all major Platforms (OSX, Windows and Linux)
- ❖ **Quincy** is freeware open-source. It is a **simple programming environment for C/C++ on Windows**. It contains an editor, a compiler, a debugger, and graphics and GUI toolkits. Because of its simple interface, Quincy is ideal for learning C or C++ programming.

2.3 A brief look at cout and cin

- 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, tab 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.

2.4 Keywords, Identifiers, Comments, Inputs, Outputs, Parts of a program

Keywords

Some words are reserved by C++, and you may not use them as variable names. These are keywords used by the compiler to control your program. Keywords include if, while, for, and main. The key words make up the "core" of the language and have specific purposes. They are words that have a special meaning. They may only be used for their intended purpose. They are also known as reserved words.

- ✓ For example, the following expressions are always considered key words according to the ANSI-C++ standard and therefore they must not be used as identifiers:

asm	auto	break	bool	case
catch	char	class	const	const_cast
continue	default	delete	do	double
dynamic_cast	else	enum	explicit	extern
false	float	for	friend	goto
if	inline	int	long	mutable
namespace	new	operator	private	protected
public	register	reinterpret_cast	return	short
signed	sizeof	static	static_cast	struct
switch	template	this	throw	true
try	typedef	typeid	typename	union
unsigned	using	virtual	void	volatile
wchar_t	while			

- ✓ Additionally, alternative representations for some operators do not have to be used as identifiers since they are reserved words under some circumstances: **and**, **not**, **not_eq**, **or**, **or_eq**, **xor**, **xor_eq**
- ✓ **Note:** The C++ language is "case sensitive", that means that an identifier written in capital letters is not equivalent to another one with the same name but written in small letters. Thus, for example the variable **RESULT** is not the same as the variable **result** nor the variable **Result**.

SPECIAL CHARACTERS

- ✓ In the sample program you encountered several sets of special characters. The following table provides a short summary of how they were used.

CHARACTER	NAME	DESCRIPTION
//	Double slash	Marks the beginning of a comment.
#	Pound sign	Marks the beginning of a preprocessor directive.
< >	Opening and closing brackets	Encloses a filename when used with the <code>#include</code> directive.
()	Opening and closing parentheses	Used in naming a function, as in <code>int main()</code> .
{ }	Opening and closing braces	Encloses a group of statements, such as the contents of a function.
" "	Opening and closing quotation marks	Encloses a string of characters, such as a message that is to be printed on the screen.
;	Semicolon	Marks the end of a complete programming statement.

Escape Sequence

- ✓ `\n` is an example of an *escape sequence*. Escape sequences are written as a backslash character (`\`) followed by one or more control characters and are used to control the way

output is displayed. There are many escape sequences in C++. The newline escape sequence (`\n`) is just one of them.

- ✓ When `cout` encounters `\n` in a string, it doesn't print it on the screen but interprets it as a special command to advance the output cursor to the next line. You have probably noticed inserting the escape sequence requires less typing than inserting `endl`. That's why many programmers prefer it.
- ✓ Escape sequences give you the ability to exercise greater control over the way information is output by your program. The following table lists a few of them.

ESCAPE SEQUENCE	NAME	DESCRIPTION
<code>\n</code>	Newline	Causes the cursor to go to the next line for subsequent printing.
<code>\t</code>	Horizontal tab	Causes the cursor to skip over to the next tab stop.
<code>\a</code>	Alarm	Causes the computer to beep.
<code>\b</code>	Backspace	Causes the cursor to back up, or move left one position.
<code>\r</code>	Return	Causes the cursor to go to the beginning of the current line, not the next line.
<code>\\</code>	Backslash	Causes a backslash to be printed.
<code>\'</code>	Single quote	Causes a single quotation mark to be printed.
<code>\"</code>	Double quote	Causes a double quotation mark to be printed.

- ✓ Do not confuse the backslash (`\`) with the forward slash (`/`). An escape sequence will not work if you accidentally start it with a forward slash. Also, do not put a space between the backslash and the control character.

Identifiers

- ✓ An *identifier* is a programmer-defined name that represents some element of a program. Variable names are examples of identifiers.
- ✓ You should always choose names for your variables that give an indication of what the variables are used for. You may be tempted to declare variables with names like this:

`int x;`

- The rather nondescript name, `x`, gives no clue as to the variable's purpose. Here is a better example.

`int itemsOrdered;`

- The name `itemsOrdered` gives anyone reading the program an idea of the variable's use. This way of coding helps produce self-documenting programs, which means you get an understanding of what the program is doing just by reading its code. Because real-world programs usually have thousands of lines, it is important that they be as self-documenting as possible.

- You probably have noticed the mixture of uppercase and lowercase letters in the variable name *itemsOrdered*. Although all of C++'s key words must be written in lowercase, you may use uppercase letters in variable names.
- The reason the O in *itemsOrdered* is capitalized is to improve readability. Normally “items ordered” is two words. Unfortunately you cannot have spaces in a variable name, so the two words must be combined into one. When “items” and “ordered” are stuck together you get a variable declaration like this:

```
int itemsordered;
```
- Capitalization of the second word and succeeding words makes *itemsOrdered* easier to read and is the convention we use for naming variables in this book. However, this style of coding is not required. You are free to use all lowercase letters, all uppercase letters, or any combination of both. In fact, some programmers use the underscore character to separate words in a variable name, as in the following.

```
int items_ordered;
```
- ✓ A valid identifier is a sequence of one or more letters, digits or underline symbols (_). The length of an identifier is not limited, although for some compilers only the 32 first characters of an identifier are significant (the rest are not considered).
- ✓ Neither spaces nor marked letters can be part of an identifier. Only letters, digits and underline characters are valid. In addition, variable identifiers should always begin with a letter. They can also begin with an underline character (_), but this is usually reserved for external links. In no case they can begin with a digit.
- ✓ Another rule that you have to consider when inventing your own *identifiers* is that they cannot match any **key word** of the C++ language nor your compiler's specific ones since they could be confused with these.

Legal Identifiers

- ✓ Regardless of which style you adopt, be consistent and make your variable names as sensible as possible.
- ✓ Here are some specific rules that must be followed with all identifiers.
 - The first character must be one of the letters a through z, A through Z, or an underscore character (_).
 - After the first character you may use the letters a through z or A through Z, the digits 0 through 9, or underscores.
 - Uppercase and lowercase characters are distinct. This means *ItemsOrdered* is not the same as *itemsordered*.
- ✓ The following table lists variable names and indicates whether each is legal or illegal in C++.

VARIABLE NAME	LEGAL OR ILLEGAL
dayOfWeek	Legal.
3dGraph	Illegal. Variable names cannot begin with a digit.
_employee_num	Legal.
June1997	Legal.
Mixture#3	Illegal. Variable names may only use letters, digits, and underscores.

- ✓ You may choose your own variable names in C++, as long as you do not use any of the C++ *key words*. The key words make up the “core” of the language and have specific purposes.

Comments

- ✓ Comments are pieces of source code discarded from the code by the compiler. They do nothing. Their purpose is only to allow the programmer to insert notes or descriptions embedded within the source code.
- ✓ C++ supports two ways to insert comments:
 - *//* line comment
 - */** block comment **/*
- ✓ The first of them, the line comment, discards everything from where the pair of slash signs (*//*) is found up to the end of that same line. The second one, the block comment, discards everything between the */** characters and the next appearance of the **/* characters, with the possibility of including several lines.
- ✓ We are going to add comments to our second program:

```

/* my second program in C++
   with more comments */
#include <iostream.h>
int main ()
{
    cout << "Hello World! "; // says Hello World!
    cout << "I'm a C++ program"; // says I'm a C++ program
    return 0;
}

```

- ✓ If you include comments within the source code of your programs without using the comment characters combinations *//*, */** or **/*, the compiler will take them as if they were C++ instructions and, most likely causing one or several error messages.

Basic Input/Output

- ✓ One of the primary jobs of a computer is to produce output. When a program is ready to send information to the outside world, it must have a way to transmit that information to an output device.
- ✓ The *console* is the basic interface of computers, normally it is the set composed of the keyboard and the screen. The keyboard is generally the standard *input* device and the *monitor* is the standard output device.
- ✓ In the *iostream* C++ library, standard *input* and *output* operations for a program are supported by two data streams: **cin** for input and **cout** for output.
- ✓ Therefore **cout** (the standard output stream) is normally directed to the screen and **cin** (the standard input stream) is normally assigned to the keyboard.
- ✓ By handling these two streams you will be able to interact with the user in your programs since you will be able to show messages on the screen and receive his/her input from the keyboard.

Output (cout)

- ✓ The **cout** stream is used in conjunction with the overloaded operator << (a pair of "less than" signs).

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

- ✓ When the << symbol is used this way, it is called the *stream-insertion operator*, since it inserts the data that follows it into the stream that precedes it. The information immediately to the right of the operator is sent to cout and then displayed on the screen. In the examples above it inserted the constant string Output sentence, the numerical constant 120 and the variable x into the output stream **cout**.
- ✓ **Note:** The stream insertion operator is always written as two less-than signs with no space between them. Because you are using it to send a stream of data to the cout object, you can think of the stream insertion operator as an arrow that must point toward cout.
- ✓ Notice that the first of the two sentences is enclosed between double quotes (") because it is a string of characters.
- ✓ Whenever we want to use constant strings of characters we must enclose them between double quotes (") so that they can be clearly distinguished from variables. For example, these two sentences are very different:

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

- ✓ The *insertion operator* (<<) may be used more than once in a same sentence:

```
cout << "Hello, " << "I am " << "a C++ sentence";
```

this last sentence would print the message **Hello, I am a C++ sentence** on the screen.

- ✓ The utility of repeating the insertion operator (<<) is demonstrated when we want to print out a combination of variables and constants or more than one variable:

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

- ✓ If we suppose that variable age contains the number 24 and the variable zipcode contains 90064 the output of the previous sentence would be:

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

- ✓ It is important to notice that **cout** does not add a line break after its output unless we explicitly indicate it, therefore, the following sentences will be shown followed in screen:

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

even if we have written them in two different calls to **cout**.

- ✓ So, in order to perform a line break on output we must explicitly order it by inserting a new-line character, that in C++ can be written as **\n**:

```
cout << "First sentence.\n ";
cout << "Second sentence.\nThird sentence.";
```

Produces the following output:

```
First sentence.
Second sentence.
Third sentence.
```

- ✓ Additionally, to add a new-line, you may also use the **endl** manipulator. For example:


```
cout << "First sentence." << endl;
cout << "Second sentence." << endl;
```

Would print out:

```
First sentence.
Second sentence.
```

- ✓ The **endl** manipulator has a special behavior when it is used with buffered streams: they are flushed. But anyway **cout** is unbuffered by default.
- ✓ You may use either the `\n` escape character or the **endl** manipulator in order to specify a line jump to **cout**. Notice the differences of use shown earlier.

Input (cin).

- ✓ Handling the standard input in C++ is done by applying the overloaded operator of *extraction* (`>>`) on the **cin** stream. This must be followed by the variable that will store the data that is going to be read. For example:

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

declares the variable age as an int and then waits for an input from cin (keyborad) in order to store it in this integer variable.

- ✓ **cin** can only process the input from the keyboard once the RETURN key has been pressed. Therefore, even if you request a single character **cin** will not process the input until the user presses RETURN once the character has been introduced.
- ✓ You must always consider the *type* of the variable that you are using as a container with **cin** extraction. If you request an integer you will get an integer, if you request a character you will get a character and if you request a string of characters you will get a string of characters.

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

Output:

Please enter an integer value: 702

The value you entered is 702 and its double is 1404.

- ✓ The user of a program may be one of the reasons that provoke errors even in the simplest programs that use **cin** (like the one we have just seen). Since if you request an integer value and the user introduces a name (which is a string of characters), the result may cause your program to misoperate since it is not what we were expecting from the user. So when you use the data input provided by **cin** you will have to trust that the user of your program will be totally cooperative and that he will not introduce his name when an integer value is requested.
 - ✓ You can also use **cin** to request more than one datum input from the user:
 `cin >> a >> b;`
 is equivalent to:
 `cin >> a;`
 `cin >> b;`
 - ✓ In both cases the user must give two data, one for variable **a** and another for variable **b** that may be separated by any valid blank separator: a space, a tab character or a newline.
-

2.5 Data Types

- ✓ Computer programs collect pieces of data from the real world and manipulate them in various ways. There are many different types of data. In the realm of numeric information, for example, there are whole numbers and fractional numbers. There are negative numbers and positive numbers. And there are numbers so large, and others so small, that they don't even have a name. Then there is textual information. Names and addresses, for instance, are stored as groups of characters.
- ✓ When you write a program you must determine what types of information it will be likely to encounter. If you are writing a program to calculate the number of miles to a distant star, you'll need variables that can hold very large numbers. If you are designing software to record microscopic dimensions, you'll need to store very small and precise numbers. Additionally, if you are writing a program that must perform thousands of intensive calculations, you'll want variables that can be processed quickly. The data type of a variable determines all of these factors.
- ✓ When programming, we store the variables in our computer's memory, but the computer must know what we want to store in them since storing a simple number, a letter or a large number is not going to occupy the same space in memory.
- ✓ Although C++ offers many data types, in the very broadest sense there are only two:
 - numeric and
 - character.
- ✓ Numeric data types are broken into two additional categories:
 - integer and
 - floating-point.
- ✓ Integers are whole numbers like 12, 157, -34, and 2.
- ✓ Floating-point numbers have a decimal point, like 23.7, 189.0231, and 0.987. Additionally, the integer and floating point data types are broken into even more classifications.
- ✓ Your primary considerations for selecting a numeric data type are:
 - The largest and smallest numbers that may be stored in the variable
 - How much memory the variable uses
 - Whether the variable stores signed (both positive and negative) or unsigned (only positive) numbers
 - The number of decimal places of precision the variable has

Integer Data Types

- ✓ Includes *short*, *unsigned short*, *int*, *unsigned int*, *long*, *unsigned long* data types

Floating-Point Data Types

- ✓ Whole numbers are not adequate for many jobs. If you are writing a program that works with dollar amounts or precise measurements, you need a data type that allows fractional values. In programming terms, these are called *floating-point* numbers.
- ✓ Internally, floating-point numbers are stored in a manner similar to *scientific notation*. Take the number 47,281.97. In scientific notation this number is 4.728197×10^4 . (10^4 is equal to 10,000, and $4.728197 \times 10,000$ is 47,281.97)
- ✓ Computers typically use *E notation* to represent floating-point values. In E notation, the number 47,281.97 would be 4.728197E4. The part of the number before the E is the mantissa, and the part after the E is the power of 10. When a floating-point number is stored in memory, it is stored as the mantissa and the power of 10.
- ✓ The following table shows other numbers represented in scientific and E notation.

DECIMAL NOTATION	SCIENTIFIC NOTATION	E NOTATION
247.91	2.4791×10^2	2.4791E2
0.00072	7.2×10^{-4}	7.2E-4
2,900,000	2.9×10^6	2.9E6

- ✓ In C++ there are three data types that can represent floating-point numbers. They are *float*, *double*, *long double*.
- ✓ A *char* variable (used to hold characters) is most often one byte long.
- ✓ The size of a variable is the number of bytes of memory it uses. Typically, the larger a variable is, the greater the range it can hold.
- ✓ Our computer's memory is organized in bytes. A byte is the minimum amount of memory that we can manage. A byte can store a relatively small amount of data, usually an integer between 0 and 255 or one single character. But in addition, the computer can manipulate more complex data types that come from grouping several *bytes*, such as long numbers or numbers with decimals.
- ✓ Next you have a list of the existing fundamental data types in C++, as well as the range of values that can be represented with each one of them:

Name	Bytes*	Description	Range*
char	1	character or integer 8 bits length.	signed: -128 to 127 unsigned: 0 to 255
short	2	integer 16 bits length.	signed: -32768 to 32767 unsigned: 0 to 65535
long	4	integer 32 bits length.	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295

Int	*	Integer. Its length traditionally depends on the length of the system's Word type , 2 (4) bytes.	See short, long
float	4	floating point number.	3.4e + / - 38 (7 digits)
double	8	double precision floating point number.	1.7e + / - 308 (15 digits)
long double	10	long double precision floating point number.	1.2e + / - 4932 (19 digits)

- *Values of columns Bytes and Range may vary depending on your system.

2.6 Variables

- ✓ In order to go a little further on and to become able to write programs that perform useful tasks that really save us work we need to introduce the concept of the **variable**.
- ✓ In programming a *variable* is a named storage location for holding data. Variables allow you to store and work with data in the computer's memory. Part of the job of programming is to determine how many variables a program will need and what type of information each will hold.
- ✓ Let's think that I ask you to retain the number 5 in your mental memory, and then I ask you to also memorize the number 2. You have just stored two values in your memory. Now, if I ask you to add 1 to the first number I said, you should be retaining the numbers 6 (that is 5+1) and 2 in your memory. Values that we could now subtract and obtain 4 as the result. All this process that you have made is a simile of what a computer can do with two variables. This same process can be expressed in C++ with the following instruction set:

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

- ✓ Obviously this is a very simple example since we have only used two small integer values, but consider that your computer can store millions of numbers like these at the same time and conduct sophisticated mathematical operations with them.
- ✓ Therefore, we can define a variable as a portion of memory to store a determined value.
- ✓ Each variable needs an identifier that distinguishes it from the others, for example, in the previous code the variable identifiers were **a**, **b** and **result**, but we could have called the variables any names we wanted to invent, as long as they were valid identifiers.

Declaration of variables

- ✓ In order to use a variable in C++, we must first declare it specifying which of the data types above we want it to be.
- ✓ The syntax to declare a new variable is to write the *data type specifier* that we want (like **int**, **short**, **float**...) followed by a *valid variable identifier*.
- ✓ For example:

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

Are valid declarations of variables. The first one declares a variable of type **int** with the identifier **a**. The second one declares a variable of type **float** with the identifier **mynumber**. Once declared, variables **a** and **mynumber** can be used within the rest of their scope in the program.

- ✓ If you need to declare several variables of the same type and you want to save some writing work you can declare all of them in the same line separating the identifiers with commas. For example:

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

declares three variables (**a**, **b** and **c**) of type **int**, and has exactly the same meaning as if we had written:

```
int a;
```

```
int b;
```

```
int c;
```

- ✓ Integer data types (**char**, **short**, **long** and **int**) can be signed or unsigned according to the range of numbers that we need to represent.
- ✓ Thus to specify an integer data type we do it by putting the keyword **signed** or **unsigned** before the data type itself. For example:

```
unsigned short NumberOfSons;
```

```
signed int MyAccountBalance;
```

- ✓ By default, if we do not specify **signed** or **unsigned** it will be assumed that the type is **signed**, therefore in the second declaration we could have written:

```
int MyAccountBalance;
```

with exactly the same meaning and since this is the most usual way, few source codes include the keyword **signed** as part of a compound type name.

- ✓ The only exception to this rule is the **char** type that exists by itself and it is considered a different type than **signed char** and **unsigned char**.
- ✓ Finally, **signed** and **unsigned** may also be used as simple types, meaning the same as **signed int** and **unsigned int** respectively. The following two declarations are equivalent:

```
unsigned MyBirthYear;
```

```
unsigned int MyBirthYear;
```

- ✓ To see what variable declaration looks like in action in a program, use the following examples:

Example-1

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

Example-2

```
// a C++ program with a variable
int main()
{
    int number;
    number = 5;
    cout << "The value of number is " << "number" <<
endl;
    cout << "The value of number is " << number << endl;
    number = 7;
    cout << "Now the value of number is " << number <<
endl;
    return 0;    }
```

Program Output

The value of number is number
The value of number is 5
Now the value of number is 7

Initialization of variables

- ✓ When declaring a variable, its value is undetermined by default. But you may want a variable to store a concrete value the moment that it is declared. In order to do that, you have to append an equal sign followed by the value wanted to the variable declaration:

type identifier = initial_value ;

- ✓ For example, if we want to declare an **int** variable called **a** that contains the value **0** at the moment in which it is declared, we could write:

int a = 0;

- ✓ Here is the next line, in the above example:

number = 5;

This is called an *assignment*. The = sign is an operator that copies the value on its right (5) into the variable named on its left (number). This line does not print anything on the computer's screen. It runs silently behind the scenes, storing a value in RAM. After this line executes, number will be set to 5.

- ✓ **Note:** The item on the left hand side of an assignment statement *must* be a variable. It would be incorrect to say 5 = number;
- ✓ Additionally to this way of initializing variables (known as C-like), C++ has added a new way to initialize a variable: by enclosing the initial value between parenthesis ():

type identifier (initial_value) ;

- ✓ For example:

int a (0);

- ✓ Both ways are valid and equivalent in C++.

2.7 Constants

- ✓ Like variables, constants are data storage locations. Unlike variables, and as the name implies, constants don't change. You must initialize a constant when you create it, and you cannot assign a new value later.
- ✓ C++ has two types of constants: literal and symbolic.

1. Literal Constants

- ✓ A literal constant is a value typed directly into your program wherever it is needed. For example

```
int myAge = 39;
```

myAge is a variable of type *int*; *39* is a literal constant. You can't assign a value to *39*, and its value can't be changed.

- ✓ A constant is any expression that has a fixed value. They can be divided in Integer Numbers, Floating-Point Numbers, Characters and Strings.

a) Integer Constants

Values 1776, 707, -273 are numerical constants that identify integer decimal numbers. Notice that to express a numerical constant we do not need to write quotes (") nor any special character. There is no doubt that it is a constant: whenever we write **1776** in a program we will be referring to the value 1776.

b) Floating Point Constants

- ✓ They express numbers with decimals and/or exponents. They can include a decimal point, an **e** character (that expresses "by ten at the Xth height", where X is the following integer value) or both.

```
3.14159 // 3.14159
6.02e23 // 6.02 x 1023
1.6e-19 // 1.6 x 10-19
3.0     // 3.0
```

these are four valid numbers with decimals expressed in C++. The first number is PI, the second one is the number of Avogadro, the third is the electric charge of an electron (an extremely small number) -all of them approximated- and the last one is the number 3 expressed as a floating point numeric literal.

c) Characters and strings

- ✓ There also exist non-numerical constants, like: 'z', 'p', "Hello world", "How do you do?". The first two expressions represent single characters, and the following two represent strings of several characters.
- ✓ Notice that to represent a single character we enclose it between single quotes (') and to express a string of more than one character we enclose them between double quotes (").
- ✓ When writing both single characters and strings of characters in a constant way, it is necessary to put the quotation marks to distinguish them from possible variable identifiers or reserved words.
- ✓ Notice this: *x* and '*x*'. Here, **x** refers to variable **x**, whereas '**x**' refers to the character constant '**x**'.

2. Symbolic Constants

- ✓ A symbolic constant is a constant that is represented by a name, just as a variable is represented.
- ✓ Unlike a variable, however, after a constant is initialized, its value can't be changed.
- ✓ For example:

- If your program has one integer variable named `students` and another named `classes`, you could compute how many students you have, given a known number of classes, if you knew there were 15 students per class:


```
students = classes * 15;
```
 - In this example, 15 is a literal constant. Your code would be easier to read, and easier to maintain, if you substituted a symbolic constant for this value:


```
students = classes * studentsPerClass
```
 - If you later decided to change the number of students in each class, you could do so where you define the constant `studentsPerClass` without having to make a change every place you used that value.
- ✓ There are two ways to declare a symbolic constant in C++.

a. Defining Constants with `#define`






- ✓ The old, traditional, and now obsolete way is with a preprocessor directive, `#define`.
- ✓ To define a constant the traditional way, you would enter this:
- ```
#define studentsPerClass 15
```
- ✓ Note that `studentsPerClass` is of no particular type (`int`, `char`, and so on). `#define` does a simple text substitution. Every time the preprocessor sees the word `studentsPerClass`, it puts in the text `15`.
- ✓ Because the preprocessor runs before the compiler, your compiler never sees your constant; it sees the number 15.

#### b. Defining Constants with `const`

- ✓ Although `#define` works, there is a new, much better way to define constants in C++:
- ```
const unsigned short int studentsPerClass = 15;
```
- ✓ This example also declares a symbolic constant named `studentsPerClass`, but this time `studentsPerClass` is typed as an unsigned short int.
- ✓ This method has several advantages in making your code easier to maintain and in preventing bugs. The biggest difference is that this constant has a type, and the compiler can enforce that it is used according to its type.
- ✓ **NOTE:** Constants cannot be changed while the program is running. If you need to change `studentsPerClass`, for example, you need to change the code and recompile.

2.8 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
-  Arithmetic operator
-  Relational operator
-  Logical operator
-  Increment/decrement operator

2.8.1 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.

2.8.2. 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`

2.8.5 Relational Operators

- ✓ Relational operators allow you to compare numeric values and determine if one is greater than, less than, equal to, or not equal to another.
- ✓ Computers are good at performing calculations, but they are also quite adept at comparing values to determine if one is greater than, less than, or equal to, the other. These types of operations are

valuable for tasks such as examining sales figures, determining profit and loss, checking a number to ensure it is within an acceptable range, and validating the input given by a user.

- ✓ Numeric data is compared in C++ by using relational operators. Characters can also be compared with these operators, because characters are considered numeric values in C++. Each relational operator determines if a specific relationship exists between two values. For example, the greater-than operator ($>$) determines if a value is greater than another. The equality operator ($==$) determines if two values are equal.
- ✓ The following table lists all of C++'s relational operators.

RELATIONAL OPERATORS	MEANING
$>$	Greater than
$<$	Less than
$>=$	Greater than or equal to
$<=$	Less than or equal to
$==$	Equal to
$!=$	Not equal to

- ✓ **Note:** All the relational operators are binary operators with left-to-right associativity. Recall that associativity is the order in which an operator works with its operands.
- ✓ All of the relational operators are binary. This means they use two operands. Here is an example of an expression using the greater-than operator:

$x > y$

This expression is called a *relational expression*. It is used to determine if x is greater than y .

- ✓ The following expression determines if x is less than y :

$x < y$

- ✓ Relational expressions are Boolean expressions, which means their value can only be *true* or *false*.
- ✓ If x is greater than y , the expression $x > y$ will be true, while the expression $y == x$ will be false.
- ✓ The $==$ operator determines if the operand on its left is equal to the operand on its right. If both operands have the same value, the expression is true. Assuming that a is 4, the following expression is true:

$a == 4$

- ✓ But the following is false:

$a == 2$

- ✓ **WARNING!** Notice the equality operator is two $=$ symbols together. Don't confuse this operator with the assignment operator, which is one $=$ symbol. The $==$ operator determines if a variable is equal to another value, but the $=$ operator assigns the value on the operator's right to the variable on its left.
- ✓ A couple of the relational operators actually test for two relationships. The $>=$ operator determines if the operand on its left is greater than *or* equal to the operand on the right. Assuming that a is 4, b is 6, and c is 4, both of the following expressions are true:

$b >= a$

$a >= c$

- ✓ But the following is false:

$a \geq 5$

- ✓ The \leq operator determines if the operand on its left is less than *or* equal to the operand on its right. Once again, assuming that a is 4, b is 6, and c is 4, both of the following expressions are true:

$a \leq c$

$b \leq 10$

- ✓ But the following is false:

$b \leq a$

- ✓ The last relational operator is \neq , which is the not-equal operator. It determines if the operand on its left is not equal to the operand on its right, which is the opposite of the $=$ operator.
- ✓ As before, assuming a is 4, b is 6, and c is 4, both of the following expressions are true:

$a \neq b$

$b \neq c$

- ✓ These expressions are true because a is *not* equal to b and b is *not* equal to c. But the following expression is false because a *is* equal to c:

$a \neq c$

- ✓ The following table shows other relational expressions and their true or false values. (**Assume x is 10 and y is 7.**)

EXPRESSION	VALUE
$x < y$	false, because x is not less than y.
$x > y$	true, because x is greater than y.
$x \geq y$	true, because x is greater than or equal to y.
$x \leq y$	false, because x is not less than or equal to y.
$y \neq x$	true, because y is not equal to x.

2.8.6 Logical Operators

- ✓ Logical operators connect two or more relational expressions into one or reverse the logic of an expression. In the previous section you saw how a program tests two conditions with two if statements. In this section you will see how to use logical operators to combine two or more relational expressions into one.
- ✓ The following table lists C++'s logical operators.

OPERATOR	MEANING	EFFECT
$\&\&$	AND	Connects two expressions into one. Both expressions must be true for the overall expression to be true.
$\ \ $	OR	Connects two expressions into one. One or both expressions must be true for the overall expression to be true. It is only necessary for one to be true, and it does not matter which.
$!$	NOT	Reverses the "truth" of an expression. It makes a true expression false, and a false expression true.

The && Operator

- ✓ The && operator is known as the logical AND operator. It takes two expressions as operands and creates an expression that is true only when both sub-expressions are true.
- ✓ Here is an example of an if statement that uses the && operator:


```
if (temperature < 20 && minutes > 12)
    cout << "The temperature is in the danger zone.";
```
- ✓ Notice that both of the expressions being ANDed together are complete expressions that evaluate to true or false. First `temperature < 20` is evaluated to produce a true or false result. Then `minutes > 12` is evaluated to produce a true or false result. Then, finally, these two results are ANDed together to arrive at a final result for the entire expression. The `cout` statement will only be executed if temperature is less than 20 AND minutes is greater than 12. If either relational test is false, the entire expression is false and the `cout` statement is not executed.
- ✓ The following table shows a truth table for the && operator. The truth table lists all the possible combinations of values that two expressions may have, and the resulting value returned by the && operator connecting the two expressions. As the table shows, both sub-expressions must be true for the && operator to return a true value.

EXPRESSION	VALUE OF THE EXPRESSION
false && false	false (0)
false && true	false (0)
true && false	false (0)
true && true	true (1)

The || Operator

- ✓ The || operator is known as the logical OR operator. It takes two expressions as operands and creates an expression that is true when either of the sub-expressions are true.
- ✓ Here is an example of an if statement that uses the || operator:


```
if (temperature < 20 || temperature > 100)
    cout << "The temperature is in the danger zone.";
```
- ✓ The `cout` statement will be executed if temperature is less than 20 OR temperature is greater than 100. If either relational test is true, the entire expression is true and the `cout` statement is executed.
- ✓ **Note:** The two things being ORed should both be logical expressions that evaluate to true or false. It would not be correct to write the if condition as `if (temperature < 20 || > 100)`
- ✓ **Note:** There is no || key on the computer keyboard. Use two | symbols. This symbol is on the backslash key. Press Shift and backslash to print it.

EXPRESSION	VALUE OF THE EXPRESSION
false false	false (0)
false true	true (1)
true false	true (1)
true true	true (1)

The ! Operator

- ✓ The ! operator performs a logical NOT operation. It takes an operand and reverses its truth or falsehood. In other words, if the expression is true, the ! operator returns false, and if the expression is false, it returns true. Here is an if statement using the ! operator:

```
if (!(temperature > 100))
    cout << "You are below the maximum temperature.\n";
```

- ✓ First, the expression (temperature > 100) is tested to be true or false. Then the ! operator is applied to that value. If the expression (temperature > 100) is true, the ! operator returns false. If it is false, the ! operator returns true. In the example, it is equivalent to asking “is the temperature not greater than 100?”
- ✓ The following table shows a truth table for the ! operator.

EXPRESSION	VALUE OF THE EXPRESSION
!false	true (1)
!true	false (0)

Conditional operator (?)

- ✓ The conditional operator evaluates an expression returning a value if that expression is true and a different one if the expression is evaluated as false. Its format is:

```
condition ? result1 : result2
```

- ✓ If condition is true the expression will return result1, if it is not it will return result2.

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

```
// conditional operator
```

```
#include <iostream>
using namespace std;
int main ()
{
    int a,b,c;
    a=2;
    b=7;
    c = (a>b) ? a : b;
    cout << c;
```

```
    return 0;
}
```

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

2.8.7 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
```

2.8.8 The Increment and Decrement Operators

- ✓ ++ and -- are operators that add and subtract 1 from their operands.
- ✓ To *increment* a value means to increase it by one, and to *decrement* a value means to decrease it by one. Both of the following statements increment the variable num:

```
num = num + 1;
num += 1;
```

And num is decremented in both of the following statements:

```
num = num - 1;
num -= 1;
```

- ✓ C++ provides a set of simple unary operators designed just for incrementing and decrementing variables. The increment operator is ++. The decrement operator is --.
- ✓ The following statement uses the ++ operator to increment num:

```
num++;
```

- ✓ And the following statement decrements num:

```
num--;
```

- ✓ **Note:** The expression num++ is pronounced “num plus plus,” and num— is pronounced “num minus minus.”
- ✓ Our examples so far show the increment and decrement operators used in *postfix mode*, which means the operator is placed after the variable. The operators also work in *prefix mode*, where the operator is placed before the variable name:

```
++num;
--num;
```

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 cannot be used at once on a single variable: Eg: ++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

Explicit type casting operator

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

✓ For example:

`int i;`

`float f = 3.14;`

`i = (int) f;`

✓ The type casting operator (**int**) converts the float number **3.14** to an integer value **3**.

2.8.9 Operator Precedence

✓ When making complex expressions with several operands, we may have some doubts about which operand is evaluated first and which later. For example, in this expression:

`a = 5 + 7 % 2`

we may doubt if it really means:

`a = 5 + (7 % 2)` with result **6**, or

`a = (5 + 7) % 2` with result **0**

The correct answer is the first of the two expressions, with a result of **6**.

✓ There is an established order with the priority of each operator, and not only the arithmetic ones (those whose preference we may already know from mathematics) but for all the operators which can appear in C++.

✓ From greatest to lowest priority, the priority order is as follows:

Priority	Operator	Description	Associativity
1	<code>() [] -> . sizeof</code>		Left
2	<code>++ --</code>	increment/decrement	Right
	<code>!</code>	unary NOT	
	<code>& *</code>	Reference and Dereference (pointers)	
	<code>(type)</code>	Type casting	
	<code>+ -</code>	Unary plus and minus	
3	<code>* / %</code>	arithmetical operations	Left
4	<code>+ -</code>	arithmetical operations	Left
5	<code>< <= > >=</code>	Relational operators	Left
6	<code>== !=</code>	Relational operators	Left

7	&&	Logic operators	Left
8	?:	Conditional	Right
9	= += -= *= /= %= >>= <<= &= ^= =	Assignment	Right

- ✓ The following table shows the precedence of the arithmetic operators. The operators at the top of the table have higher precedence than the ones below it (Highest to Lowest)

(unary negation) -
* / %
+ -

- ✓ The multiplication, division, and modulus operators have the same precedence. This is also true of the addition and subtraction operators. Table 3-2 shows some expressions with their values.

EXPRESSION	VALUE
5 + 2 * 4	13
10 / 2 - 3	2
8 + 12 * 2 - 4	28
4 + 17 % 2 - 1	4
6 - 3 * 2 + 7 - 1	6

- ✓ Associativity is the order in which an operator works with its operands. Associativity is either *left to right* or *right to left*. The associativity of the division operator is left to right, so it divides the operand on its left by the operand on its right.

Grouping with Parentheses

- ✓ Parts of a mathematical expression may be grouped with parentheses to force some operations to be performed before others.
- ✓ All these precedence levels for operators can be manipulated or become more legible using parenthesis signs (and), as in this example:

$a = 5 + 7 \% 2;$

might be written as:

$a = 5 + (7 \% 2);$ or

$a = (5 + 7) \% 2;$

according to the operation that we wanted to perform.

- ✓ So if you want to write a complicated expression and you are not sure of the precedence levels, always include parenthesis. It will probably also be more legible code.
- ✓ In the following statement, the sum of a, b, c, and d is divided by 4.

$average = (a + b + c + d) / 4;$

- ✓ Without the parentheses, however, d would be divided by 4 and the result added to a, b, and c.
- ✓ The following table shows more expressions and their values.

EXPRESSION	VALUE
$(5 + 2) * 4$	28
$10 / (5 - 3)$	5
$8 + 12 * (6 - 2)$	56
$(4 + 17) \% 2 - 1$	0
$(6 - 3) * (2 + 7) / 3$	9

EXPRESSIONS

Converting Algebraic Expressions to Programming Statements

- ✓ In algebra it is not always necessary to use an operator for multiplication. C++, however, requires an operator for any mathematical operation. The following table shows some algebraic expressions that perform multiplication and the equivalent C++ expressions.

ALGEBRAIC EXPRESSION	OPERATION	C++ EQUIVALENT
6B	6 times B	$6 * B$
(3)(12)	3 times 12	$3 * 12$
4xy	4 times x times y	$4 * x * y$

- ✓ When converting some algebraic expressions to C++, you may have to insert parentheses that do not appear in the algebraic expression. For example, look at the following expression:

$$x = \frac{a + b}{c}$$

- ✓ To convert this to a C++ statement, $a + b$ will have to be enclosed in parentheses:

$$x = (a + b) / c;$$

- ✓ The following table shows more algebraic expressions and their C++ equivalents.

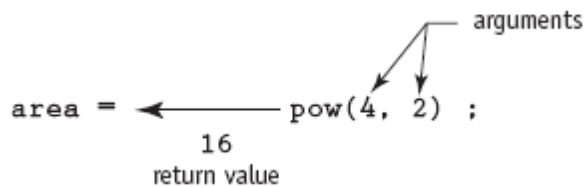
ALGEBRAIC EXPRESSION	C++ EXPRESSION
$y = 3\frac{x}{2}$	$y = x / 2 * 3;$
$z = 3bc + 4$	$z = 3 * b * c + 4;$
$a = \frac{3x+2}{4a-1}$	$a = (3 * x + 2) / (4 * a - 1)$

- ✓ Unlike many programming languages, C++ does not have an exponent operator. Raising a number to a power requires the use of a *library function*. The C++ library isn't a place where you check out books, but a collection of specialized functions. Think of a library function

as a “routine” that performs a specific operation. One of the library functions is called `pow`, and its purpose is to raise a number to a power. Here is an example of how it’s used:

```
area = pow(4, 2);
```

- ✓ This statement contains a *call* to the `pow` function. The numbers inside the parentheses are *arguments*. Arguments are information being sent to the function. `pow` always raises the first argument to the power of the second argument. In this example, 4 is raised to the power of 2. The result is *returned* from the function and used in the statement where the function call appears. In this case, the value 16 is returned from `pow` and assigned to the variable `area`. This is illustrated in the following figure.



- ✓ The statement `area = pow(4,2)` is equivalent to the following algebraic statement:

$$area = 4^2$$
- ✓ Here is another example of a statement using the `pow` function. It assigns 3 times 6^3 to `x`:
`x = 3 * pow(6, 3);`
- ✓ And the following statement displays the value of 5 raised to the power of 4:
`cout << pow(5, 4);`

2.9 Type Conversion

Explicit Type Conversion

- ✓ Type casting operators allows you to convert a datum of a given type to another.
- ✓ The first way to do this in C++ is to precede the expression to be converted by the new type enclosed between parenthesis `()`:

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

The previous code converts the float number **3.14** to an integer value (**3**). Here, the type casting operator was **(int)**.

- ✓ Another way to do the same thing in C++ is using the constructor form: preceding the expression to be converted by the type and enclosing the expression between parenthesis:

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

Both ways of type casting are valid in C++.

Implicit Type Conversion

- ✓ When an operator’s operands are of different data types, C++ will automatically convert them to the same data type. This can affect the results of mathematical expressions.

- ✓ If a floating-point value is assigned to an int variable, what value will the variable receive? If an int is multiplied by a float, what data type will the result be? What if a double is divided by an unsigned int? Is there any way of predicting what will happen in these instances?
- ✓ The answer is **yes**. C++ follows a set of rules when performing mathematical operations on variables of different data types. It's helpful to understand these rules to prevent subtle errors from creeping into your programs.
- ✓ Just like officers in the military, data types are ranked. One data type outranks another if it can hold a larger number. For example, a float outranks an int and a double outranks a float. The following table lists the data types in order of their rank, from highest to lowest.

long double
double
float
unsigned long
long
unsigned int
int

- ✓ One exception to the ranking in the above is when an int and a long are the same size. In that case, an unsigned int outranks long because it can hold a higher value.
- ✓ When C++ is working with an operator, it strives to convert the operands to the same type. This implicit, or automatic, conversion is known as *type coercion*.
- ✓ When a value is converted to a higher data type, it is said to be *promoted*. To *demote* a value means to convert it to a lower data type.
- ✓ Let's look at the specific rules that govern the evaluation of mathematical expressions.

Rule 1: *char, short, and unsigned short are automatically promoted to int.*

- ✓ You will notice that char, short, and unsigned short do not appear in the above table. That's because anytime they are used in a mathematical expression, they are automatically promoted to an int.
- ✓ The only exception to this rule is when an unsigned short holds a value larger than can be held by an int. This can happen on systems where a short is the same size as an int. In this case, the unsigned short is promoted to unsigned int.

Rule 2: *When an operator works with two values of different data types, the lower-ranking value is promoted to the type of the higher-ranking value.*

- ✓ In the following expression, assume that years is an int and interestRate is a double:
 years * interestRate

- ✓ Before the multiplication takes place, years will be promoted to a double.

Rule 3: *When the final value of an expression is assigned to a variable, it will be converted to the data type of that variable.*

- ✓ This means that if the variable receiving the value is of a lower data type than the value it is receiving, the value will be demoted to the type of the variable. If the variable's data type does not have enough storage space to hold the value, part of the value will be lost, and the variable could receive an inaccurate result.
- ✓ If the variable receiving the value is an integer and the value being assigned to it is a floating-point number, the value will be *truncated* before being stored in the variable. This means everything after the decimal point will be discarded:

```
int x = 3.75; // 3.75 will be truncated to integer 3
// x will be assigned a 3
```

- ✓ If the variable receiving the value has a higher data type than the value being assigned to it, there is no problem. In the following statement, assume that `area` is a long int, while `length` and `width` are both int:

```
area = length * width;
```

Because `length` and `width` are both an int, they will not be converted to any other data type. The result of the multiplication, however, will be converted to long so it can be stored in `area`.

- ✓ **WARNING!** Remember, when both operands of a division are integers, the fractional part will be truncated, or thrown away. Watch out for situations where an expression results in a fractional value being assigned to an integer variable. As discussed, the variable will receive a truncated value. Here is an example:

```
int x, y = 4;  
double z = 2.7;  
x = y * z;
```

- ✓ In the expression `y * z`, `y` will be promoted to double and 10.8 will result from the multiplication. Because `x` is an integer, however, 10.8 will be truncated and 10 will be stored in `x`.

2.10 Programming Errors

- ✓ Another challenge that awaits you as a problem solver is program debugging.
- ✓ **Debugging** is the process of finding and correcting errors in computer programs. No matter how careful you are as a programmer, most programs you write will contain errors. Either they won't compile or they won't execute properly. This situation is something that happens very frequently to every programmer.
- ✓ You should take program debugging as a challenge, develop your debugging skills, and enjoy the process. Program debugging is another form of problem solving
- ✓ There are three types of programming errors:
 1. **Design errors**
 2. **Syntax errors**
 3. **Run-time errors**

Design Errors

- ✓ **Design Errors** occur during the analysis, design, and implementation phases. We may choose an incorrect method of solution for the problem to be solved, we may make mistakes in translating an algorithm into a program, or we may design erroneous data for the program.
- ✓ Design errors are usually difficult to detect. Debugging them requires careful review of problem analysis, algorithm design, translation, and test data.

Syntax Errors

- ✓ **Syntax Errors** are violations of syntax rules, which define how the elements of a programming language must be written.
- ✓ They occur during the implementation phase and are detected by the compiler during the compilation process.
- ✓ In fact, another name for syntax errors is compilation errors.
- ✓ If your program contains violations of syntax rules, the compiler issues diagnostic messages. Depending on how serious the violation is, the diagnostic message may be a warning message or an error message.

- ✓ A **warning message** indicates a minor error that may lead to a problem during program execution. These errors do not cause the termination of the compilation process and may or may not be important.
- ✓ However, some warning diagnostics, if unheeded, may result in execution errors. Therefore, it is good practice to take warning message seriously and eliminate their causes in the program.
- ✓ If the syntax violation is serious, the compiler stops the compilation process and produces **error messages**, telling you about the nature of the errors and where they are in the program.
- ✓ Because of the explicit help we get from compilers, debugging syntax errors is relatively easy.

Run-time Errors

- ✓ **Run-time Errors** are detected by the computer while your program is being executed.
- ✓ They are caused by program instructions that require the computer to do something illegal, such as attempting to store inappropriate data or divide a number by zero.
- ✓ When a run-time error is encountered, the computer produces an error message and terminates the program execution. You can use these error messages to debug run-time errors

Chapter Three: Control Statements

A program is usually not limited to a linear sequence of instructions. During its process it may bifurcate, repeat code or take decisions. For that purpose, C++ provides control structures that serve to specify what has to be done to perform our program.

3.1 The if statement

- ✓ The *if statement* can cause other statements to execute only under certain conditions.
- ✓ You might think of the statements in a procedural program as individual steps taken as you are walking down a road. To reach the destination, you must start at the beginning and take each step, one after the other, until you reach the destination. The programs you have written so far are like a “path” of execution for the program to follow.
- ✓ Wouldn't it be useful, though, if a program could have more than one “path” of execution? What if the program could execute some statements only under certain circumstances?
- ✓ That can be accomplished with the *if statement*, as illustrated by the following program. The user enters three test scores and the program calculates their average. The *cout* statements inside the braces are only executed under the condition that *average* equals 100.

```
// This program averages 3 test scores.
#include <iostream.h>
int main()
{
    int score1, score2, score3;
    double average;
    cout << "Enter 3 test scores and I will average them: ";
    cin >> score1 >> score2 >> score3;
    average = (score1 + score2 + score3) / 3.0;
    cout << "Your average is " << average << endl;
    if (average == 100)
    {
        cout << "Congratulations! ";
        cout << "That's a perfect score!\n";
    }
    return 0;
}
```

Program output with example input shown in bold

Enter 3 test scores and I will average them: 80 90 70[Enter]

Your average is 80.0

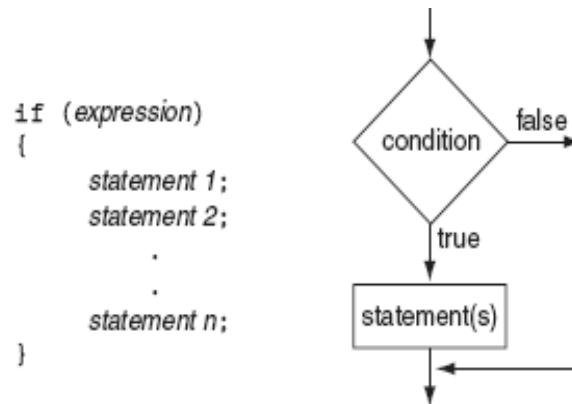
Program output with other example input shown in bold

Enter 3 test scores and I will average them: 100 100 100[Enter]

Your average is 100.0

Congratulations! That's a perfect score!

- ✓ The following figure shows the general format of the *if statement* and a flowchart visually depicting how it works.



- ✓ The *if statement* is simple in the way it works. If the expression inside the parentheses is *true*, the statements inside the braces are executed. Otherwise, they are skipped.
- ✓ This block of statements is *conditionally executed* because the statements only execute under the condition that the expression in parentheses is *true*.
- ✓ This is similar to the way we mentally test conditions every day.

If the car is low in gas, stop at a service station and get gas.

If it's raining outside, go inside.

If you're hungry, get something to eat.

- ✓ If the block of statements to be conditionally executed contains only one statement, the braces can be omitted. For example, in the above program if the two *cout* statements were combined into one statement, they could be written as shown here.

```
if (average == 100)
```

```
cout << "Congratulations! That's a perfect score!\n";
```

Programming Style and the *if Statement*

- ✓ Even though *if statements* usually span more than one line, they are technically one long statement.
- ✓ For instance, the following *if statements* are identical except in style:

```
if (a >= 100)
```

```
cout << "The number is out of range.\n";
```

```
if (a >= 100) cout << "The number is out of range.\n";
```

- ✓ The first of these two *if statements* is considered to be better style because it is easier to read. By indenting the conditionally executed statement or block of statements you are causing it to stand out visually. This is so you can tell at a glance what part of the program the *if statement* executes. This is a standard way of writing *if statements* and is the method you should use.
- ✓ Here are two important style rules you should adopt for writing *if statements*:
 - The conditionally executed statement(s) should begin on the line after the *if statement*.

- The conditionally executed statement(s) should be indented one “level” from the if statement.

✓ **Note:** In most editors, each time you press the tab key, you are indenting one level.

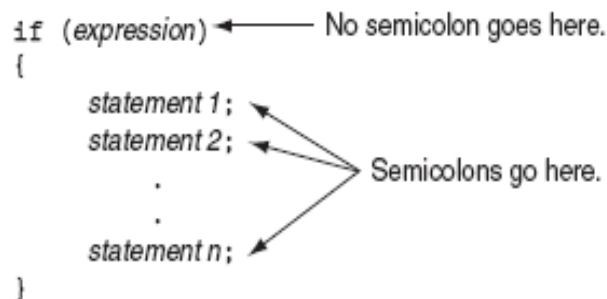
Three common errors to watch out for

✓ When writing if statements, there are three common errors you must watch out for.

1. Misplaced semicolons
2. Missing braces
3. Confusing = with ==

Be Careful with Semicolons

✓ *Semicolons* do not mark the end of a line, but the end of a complete C++ statement. The *if statement* isn't complete without the conditionally executed statement that comes after it. So, you must not put a semicolon after the *if (expression)* portion of an *if statement*.



- ✓ If you inadvertently put a semicolon after the if part, the compiler will assume you are placing a *null statement* there. The *null statement* is an empty statement that does nothing. This will prematurely *terminate the if statement*, which disconnects it from the block of statements that follows it. These statements will then always execute.
- ✓ For example, notice what would have happened in the above program if the if statement had been prematurely terminated with a semicolon, as shown here.

```

if (average == 100); // Error. The semicolon terminates the if statement prematurely.
{
    cout << "Congratulations! ";
    cout << "That's a perfect score!\n";
}
  
```

Output of revised the above program with example input shown in bold

Enter 3 test scores and I will average them: **80 90 70**[Enter]

Your average is 80.0

Congratulations! That's a perfect score!

- ✓ Because the *if statement* ends when the premature semicolon is encountered, the *cout* statements inside the braces are considered to be separate statements following the if, rather than statements belonging to the *if*. Therefore, they always execute, regardless of whether average equals 100 or not.
- ✓ **Note:** Indentation and spacing are for the human reader of a program, not the computer. Even though the *cout* statements following the *if statement* in this example are indented, the semicolon still terminates the if statement.

Don't Forget the Braces

- ✓ If you intend to conditionally execute a block of statements, rather than just one statement, with an if statement, don't forget the braces. Without a set of braces, the if statement *only executes the very next statement*. Any following statements are considered to be outside the if statement and will always be executed, even when the if condition is false.
- ✓ For example, notice what would have happened in the above program if the braces enclosing the two cout statements to be conditionally executed had been omitted.

```
if (average == 100)
cout << "Congratulations! "; // There are no braces.
cout << "That's a perfect score!\n"; // This is outside the if.
```

Output of the above program revised a second time with example input shown in bold

Enter 3 test scores and I will average them: 80 90 70[Enter]

Your average is 80.0

That's a perfect score!

- ✓ With no braces around the set of statement to be conditionally executed, only the first of these statements is considered to belong to the *if statement*. Because the condition in our test case (average == 100) was false, the *Congratulations!* message was skipped. However the *cout* statement that prints *That's a perfect score!* was executed, as it would be every time, regardless of whether average equals 100 or not.

Not All Operators Are "Equal"

- ✓ Earlier you saw a warning not to confuse the equality operator (==) with the assignment operator (=), as in the following statement:

```
if (x = 2) // Caution here!
cout << "It is True!";
```

- ✓ This statement does not determine *if x is equal to 2*; it *assigns x the value 2*! Furthermore, the *cout* statement will *always* be executed because the expression *x = 2* evaluates to 2, which C++ considers *true*.
- ✓ This occurs because the value of an assignment expression is the value being assigned to the variable on the left side of the = operator. Therefore the value of the expression *x = 2* is 2.
- ✓ Earlier you learned that C++ stores the value true as 1. But it actually considers all nonzero values, not just 1, to be *true*. Thus 2 represents a *true* condition.
- ✓ Let's examine this more closely by looking at yet another variation of the original Program. This time notice what would have happened if the equal-to relational operator in the if condition had been replaced by the assignment operator, as shown here.

```
if (average = 100) // Error. This assigns 100 to average.
{
cout << "Congratulations! ";
cout << "That's a perfect score!\n";
}
```

Output of above program revised a third time with example input shown in bold

Enter 3 test scores and I will average them: 80 90 70[Enter]

Your average is 80.0

Congratulations! That's a perfect score!

- ✓ Rather than being *compared to 100*, average is *assigned the value 100* in the if statement. This causes the if test to evaluate to 100, which is considered *true*. Therefore the two *cout* statements will execute every time, regardless of what test scores are entered by the user.

More about Truth

- ✓ You have seen that a relational expression has the value 1 when it is true and 0 when false. You have also seen that while 0 is considered false, all values other than 0 are considered true. This means that any value, even a negative number, represents *true* as long as it is not 0.
- ✓ Here is a summary of the rules you have seen so far:
 - When a relational expression is *true* it has the value 1.
 - When a relational expression is *false* it has the value 0.
 - An expression that has the value 0 is considered *false* by the if statement.
 - An expression that has any value other than 0 is considered *true* by the if statement.
- ✓ Relational expressions are not the only conditions that may be tested. For example, the following is a legal if statement in C++:

```
if (value)
```

```
    cout << "It is True!";
```

- This if statement does not test a relational expression, but rather the contents of a variable. If the variable, *value*, contains any number other than 0, the message "*It is True!*" will be displayed.
- If *value* is set to 0, however, the *cout* statement will be skipped.
- ✓ Here is another example:


```
if (x + y)
    cout << "It is True!";
```

 - In this statement the sum of *x* and *y* is tested. If the sum is 0, the expression is *false*; otherwise it is *true*.
- ✓ You may also use the *return value* of a function call as a conditional expression. Here is an example that uses the *pow* function:

```
if (pow(a, b))
```

```
    cout << "It is True!";
```

- This if statement uses the *pow* function to raise *a* to the power of *b*. If the result is anything other than 0, the *cout* statement is executed.

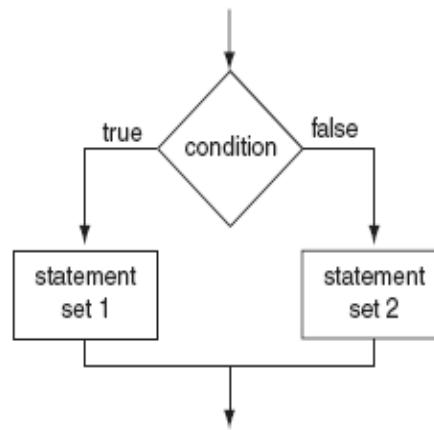
3.2 The if/else statement

- ✓ The *if/else statement* will execute one set of statements when the *if expression* is *true*, and another set when the expression is *false*.
- ✓ The if/else statement is an expansion of the if statement. The following figure shows the general format of this statement and a flowchart visually depicting how it works.

```

if (expression)
{
    statement set 1;
}
else
{
    statement set 2;
}

```



- ✓ As with the if statement, an expression is evaluated. If the expression is true, a block containing one or more statements is executed. If the expression is false, however, a different group of statements is executed.
- ✓ The following program uses the if/else statement along with the modulus operator to determine if a number is odd or even.

*/*This program uses the modulus operator to determine if a number is odd or even. If the number // is evenly divisible by 2, it is an even number. A remainder indicates it is odd.*/*

#include <iostream>

using namespace std;

int main()

{

int number;

cout << "Enter an integer and I will tell you if it\n";

cout << "is odd or even. ";

cin >> number;

if (number % 2 == 0)

cout << number << " is even.\n";

else

cout << number << " is odd.\n";

return 0;

}

Program output with example input shown in bold

Enter an integer and I will tell you if it

is odd or even. 17[Enter]

17 is odd.

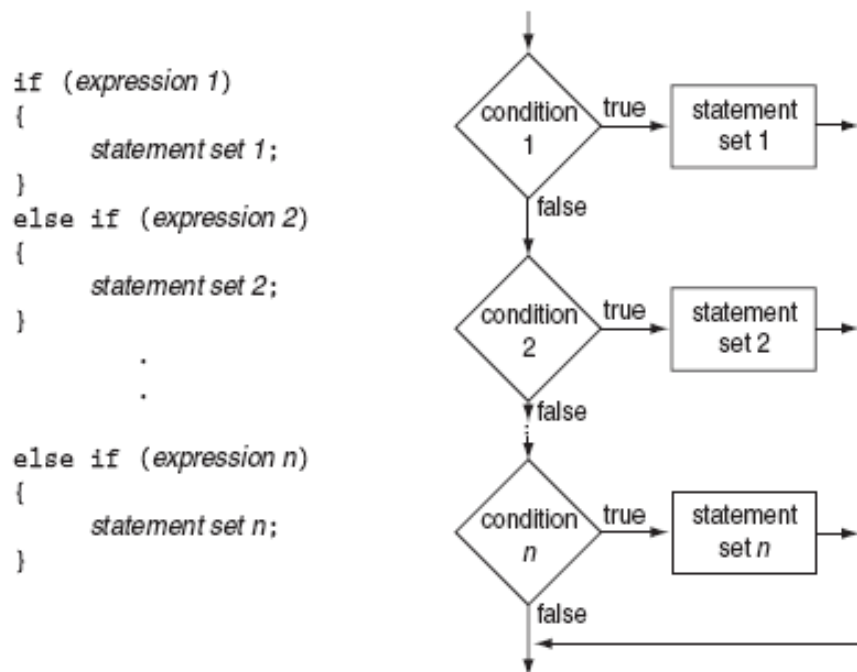
- ✓ The *else* part at the end of the if statement specifies a statement that is to be executed when the expression is *false*. When *number % 2* does not equal 0, a message is printed indicating the number is odd.
- ✓ Note that the program will only take *one of the two paths* in the if/else statement.
- ✓ If you think of the statements in a computer program as steps taken down a road, consider the if/else statement as a junction in the road. Instead of being a momentary detour, like an

if statement, the if/else statement causes program execution to follow one of two exclusive paths.

- ✓ Notice the programming style used to construct the if/else statement. The word *else* is at the same level of indentation as *if*. The statement whose execution is controlled by else is indented one level. This visually depicts the two paths of execution that may be followed.
- ✓ As with the *if* part, if you don't use braces the *else* part controls a single statement. If you wish to execute more than one statement with the *else* part, place these statements inside a set of braces.

3.3 The if/else --- if statement

- ✓ The *if/else if statement* is a chain of if statements. They perform their tests, one after the other, until one of them is found to be true.
- ✓ The following figure shows its format and a flowchart visually depicting how it works.



- ✓ We make certain mental decisions by using sets of different but related rules. For example, we might decide the type of coat or jacket to wear by consulting the following rules:
 - if it is very cold, wear a heavy coat,*
 - else, if it is chilly, wear a light jacket,*
 - else, if it is windy, wear a windbreaker,*
 - else, if it is hot, wear no jacket.*
- The purpose of these rules is to determine which type of outer garment to wear. If it is cold, the first rule dictates that a heavy coat must be worn. All the other rules are then ignored. If the first rule doesn't apply, however (if it isn't cold), then the second rule is consulted. If that rule doesn't apply, the third rule is consulted, and so forth.
- ✓ This type of decision making is also very common in programming. In C++ it is accomplished through the *if/else if statement*.

- ✓ This construction is like a chain of if/else statements. The *else* part of one statement is linked to the *if* part of another. When put together this way, the chain of if/elses becomes one long statement. The following program shows an example. The user is asked to enter a numeric test score and the program displays the letter grade earned.

```
// This program uses an if/else if statement to assign a letter grade (A, B, C, D, or F) to
a numeric
// test score.
#include <iostream.h>
int main()
{
    int testScore;
    char grade;
    cout << "Enter your numeric test score and I will\n";
    cout << "tell you the letter grade you earned: ";
    cin >> testScore;
    if (testScore < 60)
        grade = 'F';
    else if (testScore < 70)
        grade = 'D';
    else if (testScore < 80)
        grade = 'C';
    else if (testScore < 90)
        grade = 'B';
    else if (testScore <= 100)
        grade = 'A';
    cout << "Your grade is " << grade << ".\n";
    return 0;
}
```

Program output with example input shown in bold

```
Enter your numeric test score and I will
tell you the letter grade you earned: 88[Enter]
Your grade is B.
```

- ✓ The *if/else if statement* has a number of notable characteristics. Let's analyze how it works in the above program. First, the relational expression *testScore < 60* is tested.

```
if (testScore < 60)
    grade = 'F';
```

- ✓ If *testScore* is less than 60, the letter 'F' is assigned to *grade* and the rest of the linked if statements are skipped. If *testScore* is not less than 60, the *else* part takes over and causes the next *if statement* to be executed.

```
else if (testScore < 70)
    grade = 'D';
```

- ✓ The first *if statement* filtered out all of the scores less than 60, so when this next *if statement* executes, *testScore* will have a value of 60 or greater. If *testScore* is less than 70, the letter 'D' is assigned to *grade* and the rest of the if/else if statement is ignored. This chain of events continues until one of the conditional expressions is found *true* or the *end of the statement* is encountered. In either case, the program resumes at the statement immediately following the if/else if statement, which is the *cout* statement that prints the *grade*.
- ✓ Each if statement in the structure depends on all the *if statements* before it being *false*.
- ✓ The statements following a particular *else if* are executed when the *conditional expression* following the else if is *true* and all previous conditional expressions are *false*.

3.3.1 Using a trailing else

- ✓ A trailing *else*, placed at the end of an *if/else if statement*, provides a default set of actions when *none* of the if expressions are *true*. What happens in the current version of the program if the user enters a test score greater than 100?
- ✓ The *if/else if statement* handles all scores through 100, but none greater. If the user were to enter 104, for example, the program would not give any letter grade because there is no code to handle a score greater than 100. Assuming that any grade over 100 is invalid, we can fix the program by placing an *else* at the end of the *if/else if statement*.

*// This program uses an if/else if statement to assign a letter grade (A, B, C, D, or F) to a
 // numeric test score. A trailing else has been added to catch test scores > 100.*

```
#include <iostream.h>
int main()
{
  int testScore;
  char grade;
  int goodScore = 1;
  cout << "Enter your numeric test score and I will\n";
  cout << "tell you the letter grade you earned: ";
  cin >> testScore;
  if (testScore < 60)
    grade = 'F';
  else if (testScore < 70)
    grade = 'D';
  else if (testScore < 80)
    grade = 'C';
  else if (testScore < 90)
    grade = 'B';
  else if (testScore <= 100)
```

```

grade = 'A';
else
goodScore = 0;
if (goodScore)
    cout << "Your grade is " << grade << ".\n";
else
{
    cout << testScore << " is an invalid score.\n";
    cout << "Please enter a score that is no greater than 100.\n";
}
return 0;
}

```

Program output with example input shown in bold

Enter your numeric test score and I will
 tell you the letter grade you earned: **104**[Enter]
 104 is an invalid score.
 Please enter a score that is no greater than 100.

- ✓ The trailing *else* catches any value that “falls through the cracks.” It provides a default response when none of the *ifs* find a *true* condition.

3.3.2 Nested if Statements

- ✓ A *nested if statement* is an if statement in the conditionally executed code of another if statement.
- ✓ Anytime an *if statement* appears inside another *if statement*, it is considered *nested*.
- ✓ In actuality, the if/else if structure is a nested if statement. Each *if* (after the first one) is nested in the *else* part of the previous *if*.
- ✓ You may also nest *ifs* inside the *if* part, as shown in the following program. Suppose the program is used to determine if a bank customer qualifies for a special interest rate on loans, intended for people who recently graduated from college and are employed.

```

// This program demonstrates a nested if statement.
#include <iostream.h>
int main()
{
    char employed, recentGrad;
    cout << "Answer the following questions with either Y for Yes or N for No.\n";
    cout << "Are you employed? ";
    cin >> employed;
    cout << "Have you graduated from college in the past two years? ";
    cin >> recentGrad;
    if (employed == 'Y')

```

```

{
if (recentGrad == 'Y')
{
    cout << "You qualify for the special ";
    cout << "interest rate.\n";
}
}
return 0;
}

```

Program output with example input shown in bold

Answer the following questions with either Y for Yes or N for No.

*Are you employed? **Y[Enter]***

*Have you graduated from college in the past two years? **Y[Enter]***

You qualify for the special interest rate.

Program output with other example input shown in bold

Answer the following questions with either Y for Yes or N for No.

*Are you employed? **Y[Enter]***

*Have you graduated from college in the past two years? **N[Enter]***

- ✓ Because the first *if statement* conditionally executes the second one, both the *employed* and *recentGrad* variables must be set to 'Y' for the message to be printed informing the user he or she qualifies for the special interest rate.
- ✓ This type of nested *if statement* is good for narrowing choices down and categorizing data. The only way the program will execute the *second if statement* is for the *conditional expression* of the *first* one to be *true*.
- ✓ **Note:** When you are debugging a program with nested *if/else statements*, it's important to know which *if statement* each *else* belongs to. The rule for matching each *else* with an *if* is that an *else* goes with the *last if statement* that doesn't have its own *else*. This is easier to see when the *if statements* are properly indented. Each *else* should be lined up with the *if* it belongs to. These visual cues are important because *nested if statements* can be very long and complex.

// This program demonstrates a nested if statement.

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

```
int main()
```

```
{
```

```
    char employed, recentGrad;
```

```
    cout << "Answer the following questions with either Y for Yes or N for No.\n";
```

```
    cout << "Are you employed? ";
```

```
    cin >> employed;
```

```
    cout << "Have you graduated from college in the past two years? ";
```

```
    cin >> recentGrad;
```

```
    if (employed == 'Y')
```



```

{ //Nested if
if (recentGrad == 'Y') // Employed and a recent grad
{
cout << "You qualify for the special interest rate.\n";
}
else // Employed but not a recent grad
{
cout << "You must have graduated from college in the past two years to qualify.\n";
}
}
else // Not employed
{
cout << "You must be employed to qualify.\n";
}
return 0;
}

```

Program output with example input shown in bold

Answer the following questions with either Y for Yes or N for No.

Are you employed? **N[Enter]**

Have you graduated from college in the past two years? **Y[Enter]**

You must be employed to qualify.

Program output with other example input shown in bold

Answer the following questions with either Y for Yes or N for No.

Are you employed? **Y[Enter]**

Have you graduated from college in the past two years? **N[Enter]**

You must have graduated from college in the past two years to qualify.

Program output with other example input shown in bold

Answer the following questions with either Y for Yes or N for No.

Are you employed? **Y[Enter]**

Have you graduated from college in the past two years? **Y[Enter]**

You qualify for the special interest rate.

3.4 The switch statement

- ✓ The *switch statement* lets the value of a *variable* or *expression* determine where the program will branch to. A branch occurs when one part of a program causes another part to execute.
- ✓ The *if/else if statement* allows your program to *branch into one of several possible paths*. It performs a series of tests (usually relational) and branches when one of these tests is *true*.
- ✓ The *switch statement* is a similar mechanism. It, however, tests the *value* of an integer expression and then uses that *value* to determine which set of statements to branch to.
- ✓ Here is the format of the switch statement:
switch (integer expression)

```

{
  case constant expression:
    // Place one or more statements here.
  case constant expression:
    // Place one or more statements here.
    // case statements may be repeated as many times as necessary.
  case constant expression:
    // Place one or more statements here.
  default:
    // Place one or more statements here.
}

```

- ✓ The first line of the statement starts with the word *switch*, followed by an *integer expression* inside parentheses. This can be either of the following:
 - A *variable* of any of the integer data types (including *char*)
 - An *expression* whose value is of any of the integer data types
- ✓ On the next line is the beginning of a block containing several *case statements*. Each *case* statement is formatted in the following manner:

case constant expression:

// Place one or more statements here.

- ✓ After the word *case* is a *constant expression* (which must be of an integer type), followed by a colon. The *constant expression* can be either an *integer literal* or an *integer named constant*.
 - The *expression* cannot be a *variable* and it cannot be a *Boolean expression* such as $x < 22$ or $n == 25$.
- ✓ The *case* statement marks the beginning of a section of statements. These statements are branched to if the value of the switch expression matches that of the case expression.
- ✓ **WARNING!** The expressions of each *case statement* in the block must be unique.
- ✓ An optional *default* section comes after all the *case statements*.
 - This section is branched to if none of the case expressions match the switch expression.
 - Thus it functions like a trailing *else* in an if/else if statement.
- ✓ The following program shows how a simple switch statement works.

```

/* This program demonstrates the use of a switch statement. The program simply tells
the user what character they entered.*/
#include <iostream.h>
int main()
{
  char choice;
  cout << "Enter A, B, or C: ";
  cin >> choice;
  switch (choice)
  {
    case 'A':

```

```

    cout << "You entered A.\n";
    break;
    case 'B':
        cout << "You entered B.\n";
        break;
    case 'C':
        cout << "You entered C.\n";
        break;
    default:
        cout << "You did not enter A, B, or C!\n";
    }
    return 0;
}

```

Program output with example input shown in bold

Enter A, B, or C: **B[Enter]**

You entered B.

Program output with different example input shown in bold

Enter A, B, or C: **F[Enter]**

You did not enter A, B, or C!

- ✓ The *first case statement* is case 'A:', the second is *case 'B':*, and the third is *case 'C':*. These statements mark where the program is to branch to if the variable choice contains the values 'A', 'B', or 'C'. (Remember, *character variables* and *constants* are considered integers.)
- ✓ The *default* section is branched to *if* the user enters anything other than A, B, or C.
- ✓ Notice the *break* statements that are in the *case 'A'*, *case 'B'*, and *case 'C'* sections.
- ✓ The *break* statement causes the program to exit the switch statement. The next statement executed after encountering a *break* statement will be whatever statement follows the *closing brace* that terminates the switch statement.
- ✓ A *break* statement is needed whenever you want to “*break out of*” a switch statement because it is not automatically exited after carrying out a set of statements the way an if/else if statement is.
- ✓ The *case* statements show the program where to start executing in the block and the *break* statements show the program where to stop.
- ✓ Without the *break* statements, the program would execute all of the lines from the matching case statement to the end of the block.
- ✓ **Note:** The *default* section (or the last case section, if there is no default) does not need a break statement. Some programmers prefer to put one there anyway, for consistency.
- ✓ The following program is demonstrates what happens if the break statements are omitted.

```

/*This program demonstrates how a switch statement works if there are no break
statements.*/
#include <iostream.h>
int main()

```

```

{
char choice;
cout << "Enter A, B, or C: ";
cin >> choice;
// The following switch statement is missing its break statements!
switch (choice)
{
case 'A':
cout << "You entered A.\n";
case 'B':
cout << "You entered B.\n";
case 'C':
cout << "You entered C.\n";
default :
cout << "You did not enter A, B, or C!\n";
}
return 0;
}

```

Program output with example input shown in bold

Enter A, B, or C: **A[Enter]**

You entered A.

You entered B.

You entered C.

You did not enter A, B, or C!

Program output with different example input shown in bold

Enter A, B, or C: **C[Enter]**

You entered C.

You did not enter A, B, or C!

- ✓ Without the *break* statement, the program “falls through” all of the statements below the one with the matching case expression. Sometimes this is what you want.
- ✓ The following program lists the features of three TV models a customer may choose from. The model 100 has remote control. The model 200 has remote control and stereo sound. The model 300 has remote control, stereo sound, and picture-in-a-picture capability. The program uses a switch statement with carefully omitted breaks to print the features of the selected model.

// This program is carefully constructed to use the "fall through"

// feature of the switch statement.

```

#include <iostream>
using namespace std;
int main()
{
int modelNum;

```

```

cout << "Our TVs come in three models:\n";
cout << "The 100, 200, and 300. Which do you want? ";
cin >> modelNum;
cout << "That model has the following features:\n";
switch (modelNum)
{
case 300:
cout << "\tPicture-in-a-picture\n";
case 200:
cout << "\tStereo sound\n";
case 100:
cout << "\tRemote control\n";
break;
default :
cout << "You can only choose the 100, ";
cout << "200, or 300.\n";
}
return 0;
}

```

Program output with example input shown in bold

Our TVs come in three models:
The 100, 200, and 300. Which do you want? **100[Enter]**
That model has the following features:
Remote control

Program output with different example input shown in bold

Our TVs come in three models:
The 100, 200, and 300. Which do you want? **200[Enter]**
That model has the following features:
Stereo sound
Remote control

Program output with different example input shown in bold

Our TVs come in three models:
The 100, 200, and 300. Which do you want? **300[Enter]**
That model has the following features:
Picture-in-a-picture
Stereo sound
Remote control

Program output with different example input shown in bold

Our TVs come in three models:
The 100, 200, and 300. Which do you want? **500[Enter]**
That model has the following features:
You can only choose the 100, 200, or 300.

- ✓ Another example of how useful this “fall through” capability can be is when you want the program to *branch to the same set of statements for multiple case expressions*. For instance, the following program asks the user to select a grade of dog food. The available choices are A, B, and C. The switch statement will recognize either upper or lowercase letters.

/ The switch statement in this program uses the "fall through" feature to catch both uppercase and lowercase letters entered by the user.*/*

```
#include <iostream.h>
int main()
{
    char feedGrade;
    cout << "Our dog food is available in three grades:\n";
    cout << "A, B, and C. Which do you want pricing for? ";
    cin >> feedGrade;
    switch(feedGrade)
    {
        case 'a':
        case 'A':
            cout << "30 cents per pound.\n";
            break;
        case 'b':
        case 'B':
            cout << "20 cents per pound.\n";
            break;
        case 'c':
        case 'C':
            cout << "15 cents per pound.\n";
            break;
        default :
            cout << "That is an invalid choice.\n";
    }
    return 0;
}
```

Program output with example input shown in bold

*Our dog food is available in three grades:
A, B, and C. Which do you want pricing for? **b[Enter]**
20 cents per pound.*

Program output with different example input shown in bold

*Our dog food is available in three grades:
A, B, and C. Which do you want pricing for? **B[Enter]**
20 cents per pound.*

- ✓ When the user enters 'a' the corresponding case has no statements associated with it, so the program falls through to the next case, which corresponds with 'A'.

```
case 'a':
case 'A':
cout << "30 cents per pound.\n";
break;
```

- ✓ The same technique is used for 'b' and 'c'.

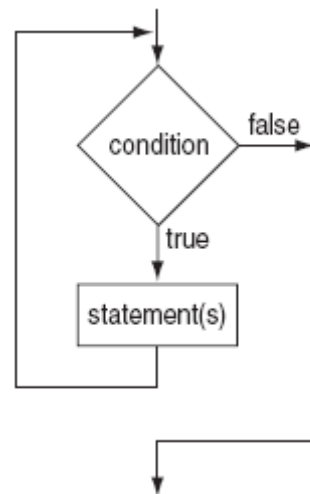
3.5 Introduction to Loops

- ✓ A *loop* is a control structure that causes a statement or group of statements to repeat. C++ has three looping control structures: the while loop, the do-while loop, and the for loop. The difference between each of these is how they control the repetition.

3.5.1 The while Loop

- ✓ The while loop has two important parts: (1) an expression that is tested for a true or false value, and (2) a statement or block that is repeated as long as the expression is true. Figure below shows the general format of the while loop and a flowchart visually depicting how it works.

```
while (expression)
{
    statement;
    statement;
    // Place as many statements here
    // as necessary
}
```



- ✓ Notice there is no semicolon after the expression in parentheses. As with the if statement, the while loop is not complete without the statements that follow it.
- ✓ The expression inside parentheses is tested and if it has a true value the statements in the body of the loop are executed. (If there is only one statement in the loop body, the braces may be omitted.) This cycle of testing the loop expression and executing the statements in the body of the loop is repeated until the expression in parentheses is false.
- ✓ The while loop works like an if statement that executes over and over. As long as the expression in the parentheses is true, the conditionally-executed statements will repeat.
- ✓ For example, we are going to make a program to count down using a *while* loop:

```
// custom countdown using while
#include <iostream.h>
int main ()
{
    int n;
```

```

cout << "Enter the starting number > ";
cin >> n;
while (n>0) {
    cout << n << ", ";
    --n;
}
cout << "FIRE!";
return 0;
}

```

Output:

Enter the starting number > 8
8, 7, 6, 5, 4, 3, 2, 1, FIRE!

- ✓ When the program starts the user is prompted to insert a starting number for the countdown. Then the *while* loop begins, if the value entered by the user fulfills the condition **n>0** (that **n** be greater than **0**), the block of instructions that follows will execute an indefinite number of times while the condition (**n>0**) remains true.
- ✓ All the process in the program above can be interpreted according to the following script: beginning in **main**:

1. User assigns a value to **n**.
2. The while instruction checks if (**n>0**). At this point there are two possibilities:
 - **true**: execute *statement* (step 3.)
 - **false**: jump *statement*. The program follows in step 5..

3. Execute *statement*:

```

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

```

(prints out **n** on screen and decreases **n** by 1).

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

- ✓ We must consider that the loop has to end at some point, therefore, within the block of instructions (loop's *statement*) we must provide some method that forces *condition* to become false at some moment, otherwise the loop will continue looping forever. In this case we have included **--n**; that causes the *condition* to become **false** after some loop repetitions: when **n** becomes **0**, that is where our countdown ends.
- ✓ Program 5-3 shows a cin statement inside a loop.

Program 5-3

// This program demonstrates a simple while loop.


```

#include <iostream.h>
int main()
{
    int number = 0;
    cout << "This program will let you enter number after\n";
    cout << "number. Enter 99 when you want to quit the ";
    cout << "program.\n";
    while (number != 99)
    cin >> number;
    cout << "Done\n";
    return 0;
}

```

Program Output with Example Input Shown in Bold

This program will let you enter number after
number. Enter 99 when you want to quit the program.

1[Enter]

2[Enter]

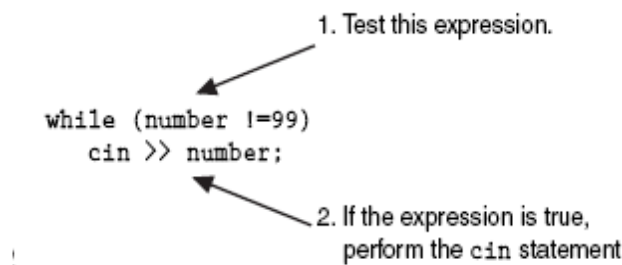
30[Enter]

75[Enter]

99[Enter]

Done

- ✓ This program repeatedly reads values from the keyboard until the user enters 99. The loop controls this action by testing the variable number. As long as number does not equal 99, the loop repeats. The cin statement, inside the loop, puts the user's input into number, where, at the beginning of the next cycle it will be tested again. (Figure 5-2 illustrates the role of the test expression and the body of the loop.) Each repetition is known as an *iteration*.



- ✓ **Note:** Many programmers choose to enclose the body of a loop in braces, even if it only has one statement:

```

while (number != 99)
{
    cin >> number;
}

```

- ✓ This is a good programming practice since the braces visually offset the body of the loop. The braces are not required for only one statement, however, so the choice of using them in this manner is yours.

while Is a Pretest Loop

- ✓ The while loop is known as a *pretest* loop, which means it tests its expression before each iteration.
- ✓ Notice the variable definition of number in Program 5-3:

```
int number = 0;
```

number is initialized to 0. If number had been initialized to 99, as shown in the following program segment, the loop would never execute the cin statement:

```
int number = 99;
while (number != 99)
    cin >> number;
```

- ✓ An important characteristic of the while loop is that the loop will never iterate if the test expression is false to start with. If you want to be sure a while loop executes the first time, you must initialize the relevant data in such a way that the test expression starts out as true.

Terminating a Loop

- ✓ In all but rare cases, loops must contain within themselves a way to terminate. This means that something inside the loop must eventually make the test expression false. The loop in Program 5-3 stops when the user enters 99 at the keyboard, which is subsequently stored in the variable number by the cin object.
- ✓ If a loop does not have a way of stopping, it is called an *infinite loop*. Infinite loops keep repeating until the program is interrupted. Here is an example:

```
int test = 0;
while (test < 10)
    cout << "Hello\n";
```

- ✓ This loop will execute forever because it does not contain a statement that changes test. Each time the test expression is evaluated, test will still be equal to 0. Here is another version of the loop. This one will stop after it has executed 10 times:

```
int test = 0;
while (test < 10)
{
    cout << "Hello\n";
    test++;
}
```

- ✓ This loop increments test after each time it prints "Hello\n". When test reaches 10, the expression test < 10 is no longer true, so the loop will stop.
- ✓ It's also possible to create an infinite loop by accidentally placing a semicolon after the test expression. Here is an example:

```
int test = 0;
while (test < 10); // Error: Note the semicolon here.
{
    cout << "Hello\n";
    test++;
}
```

- ✓ The semicolon after the test expression is assumed to be a null statement and disconnects the while statement from the block that comes after it. To the compiler, the loop looks like this:

```
while (test < 10);
```

- ✓ This while loop will forever execute the null statement, which does nothing. The program will appear to have “hung,” or “gone into space” because there is nothing to display screen output or show activity.
- ✓ Another common pitfall with loops is accidentally using the = operator when you intend to use the == operator. The following is an infinite loop because the test expression assigns 1 to remainder each time it is evaluated rather than testing if remainder is equal to 1:

```
while (remainder = 1) // Error: Notice the assignment.
{
    cout << "Enter a number: ";
    cin >> num;
    remainder = num % 2;
}
```

- ✓ Remember, any nonzero value is evaluated as true.

Programming Style and the while Loop

- ✓ It's possible to create loops that look like this:

```
while (number != 99) cin >> number;
```

as well as this:

```
while (test < 10) { cout << "Hello\n"; test++; }
```

- ✓ Avoid this style of programming, however. The programming style you should use with the while loop is similar to that of the if statement:
 - If there is only one statement repeated by the loop, it should appear on the line after the while statement and be indented one additional level.
 - If the loop repeats a block, the block should begin on the line after the while statement and each line inside the braces should be indented.
- ✓ In general, you'll find a similar style being used with the other types of loops presented in this chapter.

Counters

- ✓ A counter is a variable that is regularly incremented or decremented each time a loop iterates.
- ✓ Sometimes it's important for a program to keep track of the number of iterations a loop performs. For example, Program 5-4 displays a table consisting of the numbers 1 through 10 and their squares, so its loop must iterate 10 times.

Program 5-4

// This program displays the numbers 1 through 10 and their squares.

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

```
int main()
```

```
{
```

```
    int num = 1; // Initialize counter
```

```

cout << "Number Number Squared\n";
cout << "-----\n";
while (num <= 10)
{
cout << num << "\t\t" << (num * num) << endl;
num++; // Increment counter
}
return 0;
}

```

Program Output

Number Number Squared

```

-----
1          1
2          4
3          9
4         16
5         25
6         36
7         49
8         64
9         81
10        100

```

- ✓ In Program 5-4, the variable `num`, which starts at 1, is incremented each time through the loop.
- ✓ When `num` reaches 11 the loop stops. `num` is used as a *counter* variable, which means it is regularly incremented in each iteration of the loop. In essence, `num` keeps count of the number of iterations the loop has performed. Since `num` is controlling when to stay in the loop and when to exit from the loop, it is called the *loop control variable*.
- ✓ **Note:** It's important that `num` be properly initialized. Remember, variables defined inside a function have no guaranteed starting value.
- ✓ In Program 5-4, `num` is incremented in the last statement of the loop. Another approach is to combine the increment operation with the relational test, as shown in Program 5-5.

Program 5-5

// This program displays the numbers 1 through 10 and their squares.

```

#include <iostream.h>
int main()
{
int num = 0;
cout << "Number Number Squared\n";
cout << "-----\n";
while (num++ < 10)

```


```
cout << num << "\t\t" << (num * num) << endl;
return 0;
}
```

Program Output

Number Number Squared

```
-----
1          1
2          4
3          9
4         16
5         25
6         36
7         49
8         64
9         81
10        100
```

- ✓ Notice that `num` is now initialized to 0, rather than 1, and the relational expression uses the `<` operator instead of `<=`. This is because of the way the increment operator works when combined with the relational expression.
- ✓ The increment operator is used in postfix mode, which means it adds one to `num` after the relational test. When the loop first executes, `num` is set to 0, so 0 is compared to 10. The `++` operator then increments `num` immediately after the comparison. When the `cout` statement executes, `num` has the value 1.



`num` is compared to 10, then it is incremented. When the `cout` statement executes, `num` is 1 greater than it was in the relational test.

```
while (num++ < 10)
{
    cout << num << "\t\t" << (num * num) << endl;
}
```

- ✓ Inside the loop, `num` always has a value of 1 greater than the value previously compared to 10.
- ✓ That's why the relational operator is `<` instead of `<=`. When `num` is 9 in the relational test, it will be 10 in the `cout` statement.

Letting the User Control the Loop

- ✓ Sometimes the user has to decide how many times a loop should iterate. Program 5-6 averages a set of three test scores for any number of students. The program asks the user how many students there are scores for. This number is stored in `numStudents`. Each time the while loop is executed, a counter variable is incremented until it exceeds this number, causing the loop to iterate once for each student. This counter is the loop control variable.

Program 5-6

```

// This program averages a set of test scores for multiple
// students. It lets the user specify how many.
#include <iostream.h>
int main()
{
    int numStudents, count;
    cout << "This program will give you the average of three\n";
    cout << "test scores per student.\n";
    cout << "How many students do you have test scores for? ";
    cin >> numStudents;
    cout << "Enter the scores for each of the students.\n";
    count = 1; // Initialize the loop control variable
    while (count <= numStudents)
    {
        int score1, score2, score3;
        double average;
        cout << "\nStudent " << count << ": ";
        cin >> score1 >> score2 >> score3;
        average = (score1 + score2 + score3) / 3.0;
        cout << "The average is " << average << "\n";
        count++; // Increment the loop control variable
    }
    return 0;
}

```

Program Output with Example Input Shown in Bold

This program will give you the average of three
test scores per student.

How many students do you have test scores for? **3[Enter]**

Enter the scores for each of the students.

Student 1: **75 80 82[Enter]**

The average is 79.00

Student 2: **85 85 90[Enter]**

The average is 86.67

Student 3: **60 75 88[Enter]**

The average is 74.33

Keeping a Running Total

- ✓ Some programming tasks require a running total to be kept. Program 5-7, for example, calculates a company's total sales over a period of time by taking daily sales figures as input and keeping a running total of them as they are gathered.
- ✓ A *running total* is a sum of numbers that accumulates with each iteration of a loop. The variable used to keep the running total is called an *accumulator*.

Program 5-7

*// This program takes daily sales figures over a period of time
// and calculates their total.*

```
#include <iostream.h>
int main()
{
    int days,
    count; // Counter that controls the loop
    double total = 0.0; // Initialize accumulator
    cout << "For how many days do you have sales figures? ";
    cin >> days;
    count = 1; // Initialize counter
    while (count <= days)
    {
        double sales;
        cout << "Enter the sales for day " << count << ": ";
        cin >> sales;
        total += sales; // Accumulate running total
        count++; // Increment counter
    }
    cout << "The total sales are $" << total << endl;
    return 0;
}
```

Program Output with Example Input Shown in Bold

For how many days do you have sales figures? **5[Enter]**
 Enter the sales for day 1: **489.32[Enter]**
 Enter the sales for day 2: **421.65[Enter]**
 Enter the sales for day 3: **497.89[Enter]**
 Enter the sales for day 4: **532.37[Enter]**
 Enter the sales for day 5: **506.92[Enter]**
 The total sales are \$2448.15

- ✓ The daily sales figures are stored in sales (a variable defined inside the body of the while loop). The contents of sales is then added to total. The variable total was initialized to 0, so the first time through the loop it will be set to the same value as sales. During each iteration after the first, total will be increased by the amount in sales. After the loop has finished, total will contain the total of all the daily sales figures entered.

Sentinels

- ✓ A *sentinel* is a special value that marks the end of a list of values.
- ✓ Program 5-7, in the previous section, requires the user to know in advance the number of days there are sales figures for. Sometimes the user has a list that is very long and doesn't

know how many items there are. In other cases, the user might be entering several lists and it is impractical to require that every item in every list be counted.

- ✓ A technique that can be used in these situations is to ask the user to enter a sentinel at the end of the list. A *sentinel* is a special value that cannot be mistaken as a member of the list and signals that there are no more values to be entered. When the user enters the sentinel, the loop terminates.
- ✓ In Program 5-3 the number 99 was used as an end sentinel. Program 5-8 provides another example of using an end sentinel. This program calculates the total points earned by a soccer team over a series of games. It allows the user to enter the series of game points, then enter -1 to signal the end of the list.

Program 5-8

```
// This program calculates the total number of points a
// soccer team has earned over a series of games. The user
// enters a series of point values, then -1 when finished.
#include <iostream.h>
int main()
{
    int game = 1, points, total = 0;
    cout << "Enter the number of points your team has earned\n";
    cout << "so far in the season, then enter -1 when finished.\n\n";
    cout << "Enter the points for game " << game << ": ";
    cin >> points;
    while (points != -1)
    { total += points;
      cout << "Enter the points for game " << ++game << ": ";
      cin >> points;
    }
    cout << "\nThe total points are " << total << endl;
    return 0;
}
```

Program Output with Example Input Shown in Bold

Enter the number of points your team has earned

so far in the season, then enter -1 when finished.

Enter the points for game 1: **7[Enter]**

Enter the points for game 2: **9[Enter]**

Enter the points for game 3: **4[Enter]**

Enter the points for game 4: **6[Enter]**

Enter the Points for game 5: **8[Enter]**

Enter the points for game 6: **-1[Enter]**

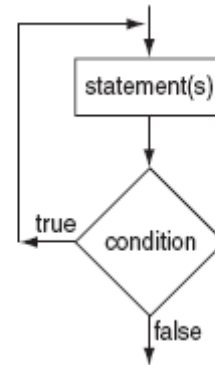
The total points are 34

- ✓ Notice how the first `cin` statement was placed before the loop. This is so the while loop will not try to test the value of points until a first value has been read in for it. The value `-1` was chosen for the sentinel because it is not possible for a team to score negative points.
- ✓ In addition to the while loop, C++ also offers the do-while and for loops. Each loop is appropriate for different programming problems.

3.5.2 The do-while Loop

- ✓ The do-while loop looks similar to a while loop turned upside down. Figure 5-4 shows its format and a flowchart visually depicting how it works.

```
do
{   statement;
    statement;
    // Place as many statements
    // here as necessary.
} while (expression);
```



- ✓ As with the while loop, if there is only one conditionally executed statement in the loop body, the braces may be omitted.
- ✓ **Note:** The do-while loop must be terminated with a semicolon after the closing parenthesis of the test expression.
- ✓ Besides the way it looks, the difference between the do-while loop and the while loop is that do-while is a *posttest* loop. It tests its expression after each iteration is complete. This means do-while always performs at least one iteration, even if the test expression is false from the start. For example, in the following while loop the `cout` statement will not execute at all:

```
int x = 1;
while (x < 0)
cout << x << endl;
```

- ✓ But the `cout` statement in the following do-while loop will execute once because the do-while loop does not evaluate the expression `x < 0` until the end of the iteration.

```
int x = 1;
do
cout << x << endl;
while (x < 0);
```

- ✓ You should use do-while when you want to make sure the loop executes at least once.
- ✓ For example, the following program echoes any number you enter until you enter 0.

```
// number echoer
#include <iostream.h>
int main ()
```

```

{
    unsigned long n;
    do {
        cout << "Enter number (0 to end): ";
        cin >> n;
        cout << "You entered: " << n << "\n";
    } while (n != 0);
    return 0;
}

```

Output:

Enter number (0 to end): 12345

You entered: 12345

Enter number (0 to end): 160277

You entered: 160277

Enter number (0 to end): 0

You entered: 0

- ✓ The *do-while* loop is usually used when the condition that has to determine its end is determined within the loop statement, like in the previous case, where the user input within the block of instructions is what determines the end of the loop. If you never enter the **0** value in the previous example the loop will never end.
- ✓ Program 5-11, another version of the test-averaging program, uses a do-while loop to repeat as long as the user wishes.

Program 5-11

// This program averages 3 test scores. It repeats as

// many times as the user wishes.

#include <iostream.h>

int main()

{

int score1, score2, score3;

double average;

char again;

do

{ cout << "Enter 3 scores and I will average them: ";

cin >> score1 >> score2 >> score3;

average = (score1 + score2 + score3) / 3.0;

cout << "The average is " << average << ".\n";

cout << "Do you want to average another set? (Y/N) ";

cin >> again;

} while (again == 'Y' || again == 'y');

return 0;

}

Program Output with Example Input Shown in Bold

Enter 3 scores and I will average them: **80 90 70**[Enter]

The average is 80.

Do you want to average another set? (Y/N) **y**[Enter]

Enter 3 scores and I will average them: **60 75 88**[Enter]

The average is 74.333336.

Do you want to average another set? (Y/N) **n**[Enter]

- ✓ The variable again is set by cin inside the body of the loop. Because the test occurs after the body has executed, it doesn't matter that again isn't initialized.
- ✓ Notice the use of the || operator in the test expression. Any expression that can be evaluated as true or false may be used to control a loop.

Using do-while with Menus

- ✓ The do-while loop is a good choice for repeating a menu. Recall Program 4-27, which displays a menu of health club packages. Program 5-12 is a modification of that program that uses a do-while loop to repeat the program until the user selects item 4 from the menu.

Program 5-12

*// This menu-driven program uses a switch statement to carry out the
 // appropriate set of actions based on the menu choice entered. A
 // do-while loop allows the program to repeat until the user selects
 // menu choice 4.*

```
#include <iostream.h>
int main()
{
    int choice, months;
    double charges;
    do
    { // Display the menu choices
      cout << "\n\tHealth Club Membership Menu\n\n";
      cout << "1. Standard Adult Membership\n";
      cout << "2. Child Membership\n";
      cout << "3. Senior Citizen Membership\n";
      cout << "4. Quit the Program\n\n";
      cout << "Enter your choice: ";
      cin >> choice;
      if (choice >= 1 && choice <= 3)
      { cout << "For how many months? ";
        cin >> months;
        // Set charges based on user input
        switch (choice)
        {
            case 1: charges = months * 40.0;
```

```

    break;
    case 2: charges = months * 20.0;
    break;
    case 3: charges = months * 30.0;
    }
    // Display the monthly charges
    cout << fixed << showpoint << setprecision(2);
    cout << "The total charges are $" << charges << endl;
    }
    else if (choice != 4)
    { cout << "The valid choices are 1 through 4.\n";
      cout << "Run the program again and select one of these.\n";
    }
    } while (choice != 4);
    return 0;
    }

```

Program Output with Example Input Shown in Bold

Health Club Membership Menu

1. Standard Adult Membership
2. Child Membership
3. Senior Citizen Membership
4. Quit the Program

Enter your choice: **1[Enter]**

For how many months **12[Enter]**

The total charges are \$480.00

Health Club Membership Menu

1. Standard Adult Membership
2. Child Membership
3. Senior Citizen Membership
4. Quit the Program

Enter your choice: **4[Enter]**

3.5.3 The for Loop

- ✓ The third type of loop in C++ is the for loop. It is ideal for situations that require a counter because it has built-in expressions that initialize and update variables. Here is the format of the for loop.

```

for (initialization; test; update)
{
    statement;
    statement;
    // Place as many statements here
    // here as necessary.
}

```

}

- ✓ As with the other two loops, if there is only one statement in the loop body, the braces may be omitted.
- ✓ The for loop has three expressions inside the parentheses, separated by semicolons. (Notice there is no semicolon after the third expression.) The first expression is the *initialization expression*. It is typically used to initialize a counter or other variable that must have a starting value. This is the first action performed by the loop and it is only done once.
- ✓ The second expression is the *test expression*. Like the test expression in the while and do-while loops, it controls the execution of the loop. As long as this expression is true, the body of the for loop will repeat. The for loop is a pretest loop, so it evaluates the test expression before each iteration. It repeats *statements* while *condition(test)* remains true, like the *while* loop. So this loop is specially designed to perform a repetitive action with a counter.
- ✓ The third expression is the *update expression*. It executes at the end of each iteration. Typically, it increments a counter or other variable that must be modified in each iteration.

It works the following steps:

1. *initialization* is executed. Generally it is an initial value setting for a counter variable. This is executed only once.
2. *test* is checked, if it is **true** the loop continues, otherwise the loop finishes and *statement* is skipped.
3. *statement* is executed. As usual, it can be either a single instruction or a block of instructions enclosed within curly brackets { }.
4. finally, whatever is specified in the *update* field is executed and the loop gets back to step 2.

Here is an example of countdown using a *for* loop.

```
// countdown using a for loop
#include <iostream.h>
int main ()
{
    for (int n=10; n>0; n--) {
        cout << n << ", ";
    }
    cout << "FIRE!";
    return 0;
}
```

Output:

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

- ✓ The *initialization* and *update* fields are optional. They can be avoided but not the semicolon signs among them. For example we could write: **for (;n<10;)** if we want to specify no *initialization* and no *update*; or **for (;n<10;n++)** if we want to include an *update* field but not an *initialization*.

- ✓ Optionally, using the comma operator (,) we can specify more than one instruction in any of the fields included in a **for** loop, like in *initialization*, for example. The comma operator (,) is an instruction separator, it serves to separate more than one instruction where only one instruction is generally expected.

- ✓ For example, suppose that we wanted to initialize more than one variable in our loop:

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

This loop will execute 50 times if neither **n** nor **i** are modified within the loop:

```
for ( n=0, i=100 ; n!=i ; n++, i-- )
```

The diagram shows the components of the for loop:

- Initialization:** Points to the first part of the loop header: `n=0, i=100`.
- Condition:** Points to the middle part of the loop header: `n!=i`.
- Increase:** Points to the last part of the loop header: `n++, i--`.

n starts with **0** and **i** with **100**, the condition is (**n!=i**) (that **n** be not equal to **i**). Because **n** is increased by one and **i** decreased by one, the loop's condition will become false after the 50th loop, when both **n** and **i** will be equal to 50.

- ✓ Program 5-13 is another version of Program 5-5, modified to use the for loop instead of the while loop.

Program 5-13

// This program uses a for loop to display the numbers 1-10 and

// their squares.

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

```
int main()
```

```
{
```

```
int num;
```

```
cout << "Number Number Squared\n";
```

```
cout << "-----\n";
```

```
for (num = 1; num <= 10; num++)
```

```
cout << num << "\t\t" << (num * num) << endl;
```

```
return 0;
```

```
}
```

Program Output

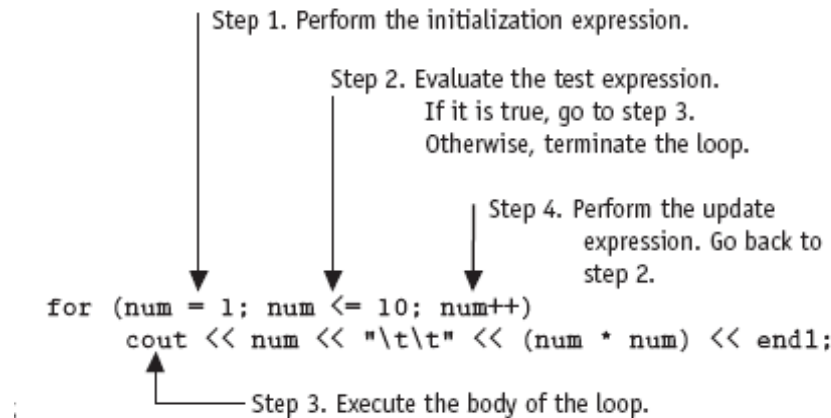
Number Number Squared

```
-----
```

1	1
2	4
3	9
4	16
5	25
6	36
7	49
8	64

9	81
10	100

- ✓ Figure 5-5 describes the mechanics of the for loop in Program 5-13.



- ✓ **WARNING!** Be careful not to place a statement in the body of the for loop that duplicates the update expression. The following loop, for example, increments *x* twice for each iteration:

```

for (x = 1; x <= 10; x++)
{
    cout << x << (x * x) << endl;
    x++; // Wrong!
}
  
```

- ✓ **Note:** Because the for loop performs a pretest, it's possible that it will never iterate. Here is an example:

```

for (x = 11; x < 10; x++)
    cout << x << endl;
  
```

- ✓ Because the variable *x* is initialized to a value that makes the test expression false from the beginning, this loop terminates as soon as it begins.

Omitting the for Loop's Expressions

- ✓ Although it is generally considered bad programming style to do so, one or more of the for loop's expressions can be omitted. The initialization expression may be omitted from inside the for loop's parentheses if it has already been performed or if no initialization is needed. Here is an example of the loop in Program 5-11 with the initialization being performed prior to the loop:

```

int num = 1;
for ( ; num <= 10; num++)
    cout << num << "\t\t" << (num * num) << endl;
  
```

- ✓ **Note:** The semicolon is still required, even though the initialization expression is missing.

- ✓ You may also omit the update expression if it is being performed elsewhere in the loop or if none is needed. The following for loop works just like a while loop:

```
int num = 1;
for ( ; num <= 10; )
{
    cout << num << "\t\t" << (num * num) << endl;
    num++;
}
```

- ✓ You can even go so far as to omit all three expressions from the for loop's parentheses. Be warned, however, that if you leave out the test expression, the loop has no built-in way of terminating. Here is an example:

```
for ( ; ; )
    cout << "Hello World\n";
```

- ✓ Because this loop has no way of stopping, it will display "Hello World\n" forever (or until something interrupts the program).

Other Forms of the Update Expression

- ✓ You are not limited to using increment statements in the update expression. Here is a loop that displays all the even numbers from 2 through 100 by adding 2 to its counter:

```
for (number = 2; number <= 100; number += 2)
    cout << number << endl;
```

- ✓ And here is a loop that counts backward from 10 down to 0:

```
for (number = 10; number >= 0; number--)
    cout << number << endl;
```

- ✓ The following loop has no formal body. The combined increment operation and cout statement in the update expression perform all the work of each iteration:

```
for (number = 1; number <= 10; cout << number++);
```

Using Initialization and Update Lists

- ✓ If your loop needs to perform more than one statement as part of the initialization, separate the statements with commas. Program 5-14 is a version of Program 5-8, modified to let the user input sales figures for one week. It initializes two variables in the for loop's initialization.

Program 5-14

```
// This program takes daily sales figures for one week
// and calculates their total.
#include <iostream.h>
int main()
{
    const int NUM_DAYS = 7; // Number of days to process
    int count;
    double total;
    for (count = 1, total = 0.0; count <= NUM_DAYS; count++)
    {
```



```

double sales;
cout << "Enter the sales for day " << count << ": ";
cin >> sales;
total += sales;
}
cout << "The total sales are $" << total << endl;
return 0;
}

```

Program Output with Example Input Shown in Bold

```

Enter the sales for day 1: 489.32[Enter]
Enter the sales for day 2: 421.65[Enter]
Enter the sales for day 3: 497.89[Enter]
Enter the sales for day 4: 532.37[Enter]
Enter the sales for day 5: 506.92[Enter]
Enter the sales for day 6: 489.01[Enter]
Enter the sales for day 7: 476.55[Enter]
The total sales are $3413.71

```

- ✓ In the for loop, count is initialized to 1, then total is initialized to 0.0. You may place more than one statement in the update expression as well.

```

double sales;
for (count = 1, total = 0.0; count <= NUM_DAYS; count++, total += sales)
{
    cout << "Enter the sales for day " << count << ": ";
    cin >> sales;
}

```

- ✓ In the update expression of this loop, count is incremented, then the value in sales is added to total at the end of each iteration. The two statements are separated by a comma.
- ✓ **Note:** In the preceding program segment, the definition of the variable sales was moved out of the body of the loop. The scope of a variable defined in the block does not extend outside the braces. In order for the update expression to see sales, its definition had to be placed prior to the for loop.
- ✓ Connecting multiple statements with commas works well in the initialization and update expressions, but don't try to connect multiple relational expressions this way in the conditional expression. If you wish to perform more than one conditional test, build an expression with the && or || operators, like this:

```

for (count = 1, total = 0; count <= 10 && total < 500; count++)
{
    double amount;
    cout << "Enter the amount of purchase #" << count << ": ";
    cin >> amount;
    total += amount;
}

```

```
}
```

Defining a Variable in the Initialization Expression

- ✓ Not only may variables be initialized in the initialization expression, but they may be defined there as well. This is generally considered good programming style. Here is an example:

```
for (int num = 1; num <= 10; num++)
    cout << num << "\t\t" << (num * num) << endl;
```

- ✓ In this loop, the variable `num` is both defined and initialized in the initialization expression. It makes sense to define variables like counters in this manner when they are only used in a loop.
- ✓ This makes their purpose clearer.

Deciding Which Loop to Use

- ✓ Although most repetitive algorithms can be written with any of the three types of loops, each works best in different situations.
- ✓ Each of C++'s three loops are ideal to use in different situations. Here's a short summary of when each loop should be used.

The while Loop

- ✓ The while loop is a *pretest* loop. It is ideal in situations where you do not want the loop to iterate if the condition is false from the beginning. It is also ideal if you want to use a sentinel.

```
cout << "This program finds the square of any integer.\n";
cout << "\nEnter an integer, or -99 to quit: ";
cin >> num;
while (num != -99)
{ cout << num << " squared is " << pow(num, 2) << endl;
  cout << "\nEnter an integer, or -99 to quit ";
  cin >> num;
}
```

The do-while Loop

- ✓ The do-while loop is a *posttest* loop. It is ideal in situations where you always want the loop to iterate at least once.

```
cout << "This program finds the square of any integer.\n";
do
{ cout << "\nEnter an integer: ";
  cin >> num;
  cout << num << " squared is " << pow(num, 2) << endl;
  cout << "Do you want to square another number? (Y/N) ";
  cin >> doAgain;
} while (doAgain == "Y" || doAgain == "y");
```

The for Loop

- ✓ The for loop is a *pretest* loop that first executes an initialization expression. In addition, it automatically executes an update expression at the end of each iteration. It is ideal for

situations where a counter variable is needed. The for loop is primarily used when the exact number of required iterations is known.

```
cout << "This program finds the squares of the integers "
<< "from 1 to 8.\n\n";
for (num = 1; num <= 8; num++)
{
    cout << num << " squared is " << pow(num, 2) << endl;
}
```

Nested Loops

- ✓ A loop that is inside another loop is called a *nested loop*.
- ✓ The first loop is called the *outer loop*. The one nested inside it is called the *inner loop*. This is illustrated by the following two while loops. Notice how the inner loop must be completely contained within the outer one.

```
while (some condition) // Beginning of the outer loop
{ ---
    while (some condition) // Beginning of the inner loop
    { ---
        ---
    } // End of the inner loop
} // End of the outer loop
```

- ✓ Nested loops are used when, for each iteration of the outer loop, something must be repeated a number of times. Here are some examples from everyday life:
 - For *each* batch of cookies to be baked we must put *each* cookie on the cookie sheet.
 - For *each* salesperson, we must add up *each* sale to determine total commission.
 - For *each* teacher we must produce a class list for *each* of their classes.
 - For *each* student we must add up *each* test score to find the student's test average.
- ✓ Whatever the task, the inner loop will go through all its iterations each time the outer loop is done. This is illustrated by Program 5-15, which handles this last task, finding student test score averages. Any kind of loop can be nested within any other kind of loop. This program uses two for loops.

Program 5-15

```
// This program uses nested loops to average a set of test scores
// for a group of students. It asks the user for the number of
// students and the number of test scores per student.
#include <iostream.h>
int main()
{
    int numStudents, // Number of students
    numTests, // Number of tests per student
    total; // Accumulates total score for each student
    double average; // Average test score for each student
    cout << "This program averages test scores.\n";
```

```

cout << "How many students are there? ";
cin >> numStudents;
cout << "How many test scores does each student have? ";
cin >> numTests;
cout << endl;
for (int student = 1; student <= numStudents; student++)//Outer loop
{
    total = 0;
    for (int test = 1; test <= numTests; test++) //Inner loop
    {
        int score;
        cout << "Enter score " << test << " for "
        << "student " << student << ": ";
        cin >> score;
        total += score;
    } //End inner loop
    average = total / numTests;
    cout << "The average for student " << student;
    cout << " is " << average << ".\n\n";
} //End outer loop
return 0;
}

```

Program Output with Example Input Shown in Bold

This program averages test scores.

How many students are there? **2[Enter]**

How many test scores does each student have? **3[Enter]**

Enter score 1 for student 1: **84[Enter]**

Enter score 2 for student 1: **79[Enter]**

Enter score 3 for student 1: **97[Enter]**

The average for student 1 is 86.

Enter score 1 for student 2: **92[Enter]**

Enter score 2 for student 2: **88[Enter]**

Enter score 3 for student 2: **94[Enter]**

The average for student 2 is 91.

- ✓ Let's trace what happened in Program 5-15, using the sample data shown. In this case, for each of two students, each of three scores were input and summed. First the outer loop was entered and student was set to 1. Then, once the total accumulator was initialized to zero for that student, the inner loop was entered. While the outer loop was still on its first iteration and student was still 1, the inner loop went through all of its iterations, handling tests 1, 2, and 3 for that student. It then exited the inner loop and calculated and output the average for student 1. Only then did the program reach the bottom of the outer loop and go back up to

do its second iteration. The second iteration of the outer loop processed student 2. For *each* iteration of the outer loop, the inner loop did *all* its iterations.

- ✓ It might help to think of each loop as a rotating wheel. The outer loop is a big wheel that is moving slowly. The inner loop is a smaller wheel that is spinning quickly. For every rotation the big wheel makes, the little wheel makes many rotations. Since, in our example, the outer loop was done twice, and the inner loop was done three times for each iteration of the outer loop, the inner loop was done a total of six times in all. This corresponds to the six scores input by the user. The following points summarize this.
 - An inner loop goes through all of its iterations for each iteration of an outer loop.
 - Inner loops complete their iterations faster than outer loops.
 - To get the total number of iterations of an inner loop, multiply the number of iterations of the outer loop by the number of iterations done by the inner loop each time the outer loop is done.

Breaking Out of a Loop

- ✓ The `break` statement causes a loop to terminate early.
- ✓ Sometimes it's necessary to stop a loop before it goes through all its iterations. The `break` statement, which was used with `switch` in Chapter 4, can also be placed inside a loop.
- ✓ When it is encountered, the loop stops and the program jumps to the statement immediately following the loop.
- ✓ The `while` loop in the following program segment appears to execute 10 times, but the `break` statement causes it to stop after the fifth iteration.

```
int count = 0;
while (count++ < 10)
{
    cout << count << endl;
    if (count == 5)
        break;
}
```

- ✓ **WARNING!** Use the `break` statement with great caution. Because it bypasses the loop condition to terminate the loop, it violates the rules of structured programming and makes code more difficult to understand, debug, and maintain.
- ✓ For this reason, we do not recommend using it to exit a loop. Because it is part of the C++ language, however, we discuss it briefly in this section.
- ✓ Using `break` we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. For example, we are going to stop the count down before it naturally finishes (an engine failure maybe):

```
// break loop example
#include <iostream.h>
int main ()
{
    int n;
```

```

for (n=10; n>0; n--) {
    cout << n << ", ";
    if (n==3)
    {
        cout << "countdown aborted!";
        break;
    }
}
return 0;
}

```

Output:

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

- ✓ Program 5-16 uses the break statement to interrupt a for loop. The program asks the user for a number and then displays the value of that number raised to the powers of 0 through 10. The user can stop the loop at any time by entering Q.

Program 5-16

// This program raises the user's number to the powers of 0 through 10.

```

#include <iostream.h>
#include <math.h>
int main()
{
    int value;
    char choice;
    cout << "Enter a number: ";
    cin >> value;
    cout << "This program will raise " << value;
    cout << " to the powers of 0 through 10.\n";
    for (int count = 0; count <= 10; count++)
    {
        cout << value << " raised to the power of ";
        cout << count << " is " << pow(value, count);
        cout << "\nEnter Q to quit or any other key ";
        cout << "to continue. ";
        cin >> choice;
        if (choice == 'Q' || choice == 'q')
            break;
    }
    return 0;
}

```

Program Output with Example Input Shown in Bold

Enter a number: **2**[Enter]

This program will raise 2 to the powers of 0 through 10.

2 raised to the power of 0 is 1

Enter Q to quit or any other key to continue. **C[Enter]**

2 raised to the power of 1 is 2

Enter Q to quit or any other key to continue. **C[Enter]**

2 raised to the power of 2 is 4

Enter Q to quit or any other key to continue. **Q[Enter]**

Using break in a Nested Loop

- ✓ In a nested loop, the break statement only interrupts the loop it is placed in. The following program segment displays five rows of asterisks on the screen. The outer loop controls the number of rows and the inner loop controls the number of asterisks in each row. The inner loop is designed to display 20 asterisks, but the break statement stops it during the eleventh iteration.

```
for (int row = 0; row < 5; row++)
{
    for (int star = 0; star < 20; star++)
    {
        cout << '*';
        if (int star == 10)
            break;
    }
    cout << endl;
}
```

The output of this program segment is

```
*****
*****
*****
*****
*****
```

3.5.4 The continue and goto Statement

The continue statement

- ✓ The continue statement causes a loop to stop its current iteration and begin the next one.
- ✓ The continue statement causes the current iteration of a loop to end immediately. When continue is encountered, all the statements in the body of the loop that appear after it are ignored, and the loop prepares for the next iteration.
- ✓ In a while loop, this means the program jumps to the test expression at the top of the loop. As usual, if the expression is still true, the next iteration begins. In a do-while loop, the program jumps to the test expression at the bottom of the loop, which determines if the next iteration will begin. In a for loop, continue causes the update expression to be executed, and then the test expression to be evaluated.
- ✓ The following program segment demonstrates the use of continue in a while loop:

```
int testVal = 0;
while (testVal++ < 10)
```

```

{
    if (testVal == 4)
        continue;
    cout << testVal << " ";
}

```

- ✓ This loop looks like it displays the integers 1 through 10. When testVal is equal to 4, however, the continue statement causes the loop to skip the cout statement and begin the next iteration.
- ✓ The output of the loop is
1 2 3 5 6 7 8 9 10
- ✓ **WARNING!** As with the break statement, the continue statement violates the rules of structured programming and makes code more difficult to understand, debug, and maintain. For this reason, you should use continue with great caution.
- ✓ The *continue* instruction causes the program to skip the rest of the loop in the present iteration as if the end of the *statement* block would have been reached, causing it to jump to the following iteration. For example, we are going to skip the number 5 in our countdown:

```

// break loop example
#include <iostream.h>
int main ()
{
    for (int n=10; n>0; n--) {
        if (n==5) continue;
        cout << n << ", ";
    }
    cout << "FIRE!";
    return 0;
}

```

Output:

10, 9, 8, 7, 6, 4, 3, 2, 1, FIRE!

- ✓ Program 5-17 shows a practical application of the continue statement. The program calculates the charges for DVD rentals where current releases cost \$3.50 and all others cost \$2.50. If a customer rents several DVDs, every third one is free. The continue statement is used to skip the part of the loop that calculates the charges for every third DVD.

Program 5-17

// This program calculates DVD rental charges.

// Every third rental is free.

```
#include <iostream>
```

```
#include <iomanip>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    int dvdCount = 1, numDVDs;
```



```

double total = 0.0;
char current;
cout << "How many DVDs are being rented? ";
cin >> numDVDs;
do
{ if ((dvdCount % 3) == 0)
{
    cout << "DVD #" << dvdCount << " is free!\n";
    continue;
}
    cout << "Is DVD #" << dvdCount;
    cout << " a current release (Y/N)? ";
    cin >> current;
    if (current == 'Y' || current == 'y')
        total += 3.50;
    else
        total += 2.50;
} while (dvdCount++ < numDVDs);
cout << fixed << showpoint << setprecision(2);
cout << "The total is $" << total << endl;
return 0;
}

```

Program Output with Example Input Shown in Bold

```

How many DVDs are being rented? 6[Enter]
Is DVD #1 a current release (Y/N)? y[Enter]
Is DVD #2 a current release (Y/N)? n[Enter]
DVD #3 is free!
Is DVD #4 a current release (Y/N)? n[Enter]
Is DVD #5 a current release (Y/N)? y[Enter]
DVD #6 is free!
The total is $12.00

```

The goto instruction.

- ✓ It allows making an absolute jump to another point in the program. You should use this feature carefully since its execution ignores any type of nesting limitation.
- ✓ The destination point is identified by a label, which is then used as an argument for the goto instruction. A label is made of a valid identifier followed by a colon (:).
- ✓ This instruction does not have a concrete utility in structured or object oriented programming aside from those that low-level programming fans may find for it. For example, here is our countdown loop using **goto**:

```

// goto loop example
#include <iostream.h>
int main ()

```

```

{
    int n=10;
    loop:
    cout << n << ", ";
    n--;
    if (n>0) goto loop;
    cout << "FIRE!";
    return 0;
}

```

Output:

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

The *exit* function.

- ✓ *exit* is a function defined in [cstdlib](#) (stdlib.h) library.
- ✓ The purpose of *exit* is to terminate the running program with an specific exit code. Its prototype is:


```
void exit (int exit code);
```
- ✓ The *exit code* is used by some operating systems and may be used by calling programs. By convention, an *exit code* of 0 means that the program finished normally and any other value means an error happened.

Using Loops for Data Validation

- ✓ Loops can be used to create input routines that repeat until acceptable data is entered.
- ✓ Loops are especially useful for validating input. They can test whether an invalid value has been entered and, if so, require the user to continue entering inputs until a valid one is received. In Chapter 4 we used if statements to validate user inputs, but they do not provide an ideal solution. Examine the following code, which uses an if statement to try to ensure that the user enters a number between 1 and 100.

```

cout << "Enter a number in the range 1 - 100: ";
cin >> number;
if (number < 1 || number > 100)
{
    cout << "ERROR: The value must be in the range 1 - 100: ";
    cin >> number;
}

```

- ✓ Notice that if the user enters a good number initially, the lines of code in the body of the if statement are never executed. That is what we want to happen. If the user enters an invalid number, they will be asked to enter a new value. But what if the second number is also invalid? It will never be checked and so will be accepted.
- ✓ A better approach is to use a while loop to validate input data. The following code demonstrates this. Like the previous code, this code is validating that the user enters a number between 1 and 100.

```

cout << "Enter a number in the range 1 - 100: ";
cin >> number;

```

```

while (number < 1 || number > 100)
{
    cout << "ERROR: The value must be in the range 1 - 100: ";
    cin >> number;
}

```

- ✓ With this new code, if the user enters a good value initially, the loop will never be executed. This is what we want. If the user enters an invalid number, however, the loop will be entered and the user will not be able to leave it until a valid input has been entered. Every input will be checked.
- ✓ Program 5-18 demonstrates using while loops to perform input data validation. This program calculates the number of soccer teams a youth league can create, based on the desired team size and the total number of players.

Program 5-18

```

// This program calculates the number of soccer teams that a
// youth league may create from the number of available players.
// Input validation is performed with while loops.
#include <iostream.h>
int main()
{
    int players, teamPlayers, numTeams, leftOver;
    // Get the number of players per team.
    cout << "How many players do you wish per team?\n";
    cout << "(Enter a value in the range 9 - 15): ";
    cin >> teamPlayers;
    while (teamPlayers < 9 || teamPlayers > 15) // Validate input
    {
        cout << "Team size should be 9 to 15 players.\n";
        cout << "How many players do you wish per team? ";
        cin >> teamPlayers;
    }
    // Get the number of players available.
    cout << "How many players are available? ";
    cin >> players;
    while (players < 0) // Validate input
    {
        cout << "Please enter a positive number: ";
        cin >> players;
    }
    // Perform calculations and display results.
    numTeams = players / teamPlayers;
    leftOver = players % teamPlayers;
    cout << "There will be " << numTeams << " teams with ";

```

```
cout << leftOver << " players left over.\n";  
return 0;  
}
```

Program Output with Example Input Shown in Bold

How many players do you wish per team?

(Enter a value in the range 9 - 15): **4[Enter]**

Team size should be 9 to 15 players.

How many players do you wish per team? **12[Enter]**

How many players are available? **-142[Enter]**

Please enter a positive number: **142[Enter]**

There will be 11 teams with 10 players left over.

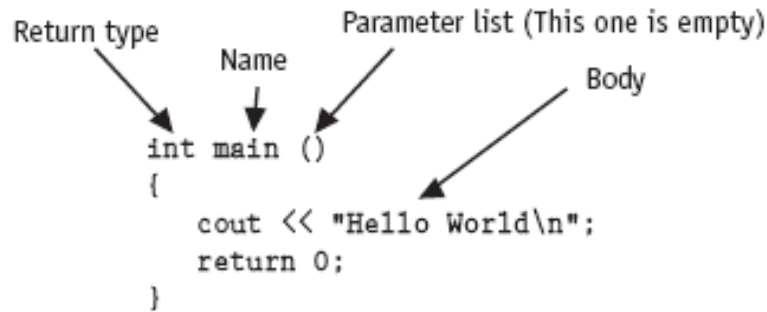
Chapter 4: Function and Passing argument to function

4.1 Definitions of Functions

- ✓ A *function* is a block of instructions that is executed when it is called from some other point of the program.
 - ✓ A function is a collection of statements that performs a specific task. So far you have used functions in two ways:
 - you have created a function called *main* in every program you've written, and
 - you have called library functions such as *pow* and *sqrt*.
 - ✓ One reason to use functions is that they break a program up into small, manageable units. Each unit is a module, programmed as a separate function.
 - ✓ Imagine a book that has a thousand pages, but isn't divided into chapters or sections. Trying to find a single topic in the book would be very difficult. Real-world programs can easily have thousands of lines of code, and unless they are modularized, they can be very difficult to modify and maintain.
 - ✓ Another reason to use functions is that they simplify programs. If a specific task is performed in several places in a program, a function can be written just once to perform that task, and then be executed anytime it is needed.
-

4.2 Declaration and calling functions

- ✓ A *function call* is a statement that causes a function to execute.
- ✓ A *function definition* contains the statements that make up the function. When creating a function, you must write its *definition*.
- ✓ All function definitions have the following parts:
 - **Name:** Every function must have a name. In general, the same rules that apply to variable names also apply to function names.
 - **Parameter list:** The program module that calls a function can send data to it. The parameter list is the list of variables that hold the values being passed to the function.
 - **Body:** The body of a function is the set of statements that carry out the task the function is performing. These statements are enclosed in a set of braces.
 - **Return type:** A function can send a value back to the program module that called it. The return type is the data type of the value being sent back.
- ✓ The following figure shows the definition of a simple function with the various parts labeled. Notice that the function's return type is actually listed first.



- ✓ **Note:** The line in the definition that reads `int main ()` is called the *function header*.

Void functions

- ✓ It isn't necessary for all functions to return a value, however. Some functions simply perform one or more statements and then terminate. These are called *void functions*. The `displayMessage` function shown here is an example:

```
void displayMessage()  
{  
    cout << "Hello from the function displayMessage.\n";  
}
```

- ✓ The function's name is `displayMessage`. This name is descriptive, as function names should be. It gives an indication of what the function does: It displays a message. Notice the function's return type is `void`. This means the function does not return a value to the part of the program that executed it. Also notice the function has no return statement. It simply displays a message on the screen and exits.

Calling a function

- ✓ A function is executed when it is *called*.
- ✓ Function ***main*** is called automatically when a program starts, but all other functions must be executed by *function call* statements.
- ✓ When a function is called, the program branches to that function and executes the statements in its body. Let's look at the following program, which contains two functions: *main* and *displayMessage*.

```
#include <iostream.h>  
  
void displayMessage()  
{  
    cout << "Hello from the function displayMessage.\n";  
}  
  
int main()  
{  
    cout << "Hello from main.\n";
```

```

displayMessage(); // Call displayMessage
cout << "Back in function main again.\n";
return 0;
}

```

- ✓ The function `displayMessage` is called by the following line in `main`:

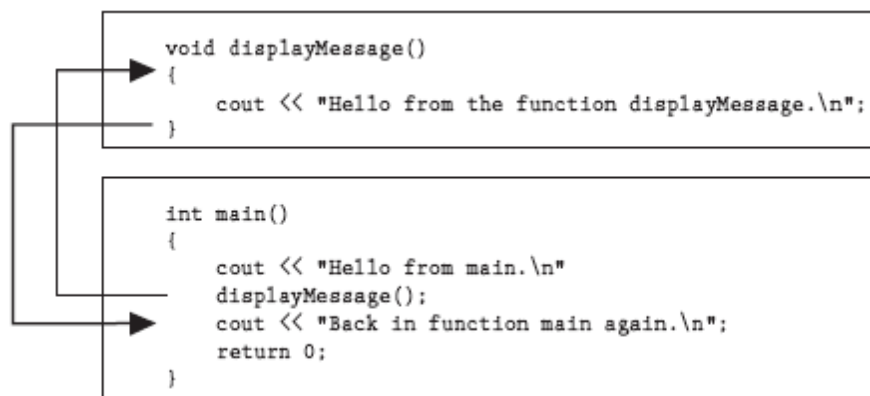
`displayMessage();`

- This line is the **function call**. It is simply the name of the function followed by a set of parentheses and a semicolon.
- Let's compare this with the function header:

Function Header → `void displayMessage()`

Function Call → `displayMessage();`

- ✓ The function header is part of the **function definition**. It declares the function's return type, name, and parameter list. It is not terminated with a semicolon because the definition of the function's body follows it.
- ✓ The function call is a statement that executes the function, so it is terminated with a semicolon like all other C++ statements.
- ✓ The function call does not list the return type, and, if the program is not passing data into the function, the parentheses are left empty.
- ✓ Even though the program starts executing at `main`, the function `displayMessage` is defined first. This is because the compiler must know the function's return type, the number of parameters, and the type of each parameter before it is called. One way to ensure the compiler will know this information is to place the function definition before all calls to that function.
- ✓ Notice how the above program flows. It starts, of course, in function `main`. When the call to `displayMessage` is encountered, the program branches to that function and performs its statements. Once `displayMessage` has finished executing, the program branches back to function `main` and resumes with the line that follows the function call. This is illustrated in the following figure.



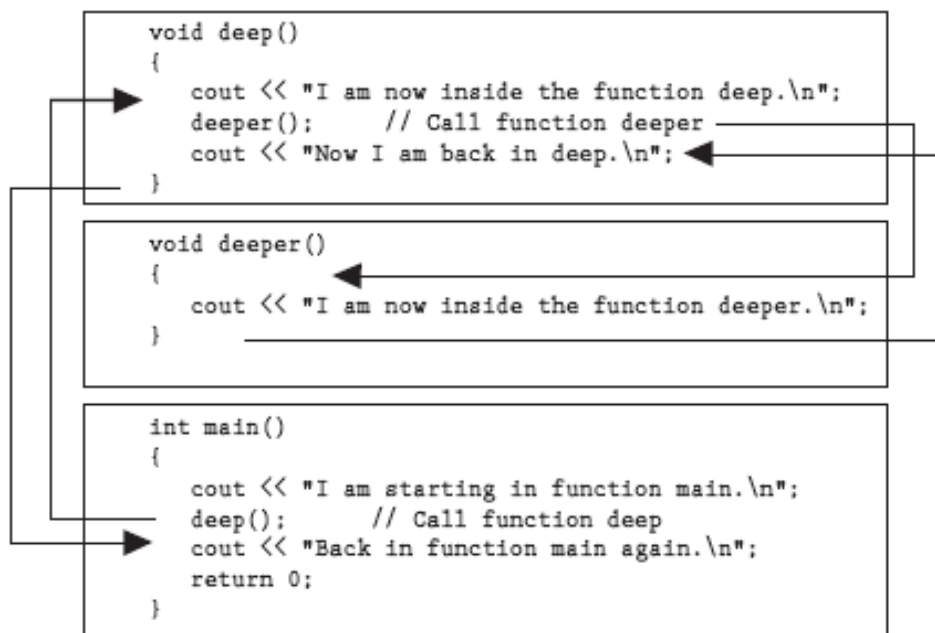
- ✓ Function call statements may be used in control structures like loops, *if statements*, and *switch statements*. The following program segment places the `displayMessage` function call inside a loop.

```

void displayMessage()
{
    cout << "Hello from the function displayMessage.\n";
}
int main()
{
    cout << "Hello from main.\n";
    for (int count = 0; count < 5; count++)
        displayMessage(); // Call displayMessage
    cout << "Back in function main again.\n";
    return 0;
}

```

- ✓ It is possible to have many functions and function calls in a program. Each call statement causes the program to branch to a function and then back to main when the function is finished.
- ✓ Functions may also be called in a hierarchical, or layered, fashion. For example, this can be demonstrated by a program which has three functions: *main*, *deep*, and *deeper*. The following figure shows the paths taken by the program. The function *main* only calls the function *deep*. In turn, *deep* calls *deeper*.



4.3 Function prototypes

- ✓ A *function prototype* eliminates the need to place a function definition before all calls to the function.
- ✓ Before the compiler encounters a call to a particular function, it must already know certain things about the function. In particular, it must know the number of parameters the function uses, the type of each parameter, and the return type of the function. Parameters

allow information to be sent to a function. Certain return types allow information to be returned from a function.

- ✓ One way of ensuring that the compiler has this required information is to place the function definition before all calls to that function, like in the above program segment.
- ✓ Another method is to declare the function with a *function prototype*. Here is a prototype for the `displayMessage` function:
`void displayMessage();`
- ✓ This prototype looks similar to the function header, except there is a semicolon at the end. The statement tells the compiler that the function `displayMessage` has a `void` return type, meaning it doesn't return a value, and it uses no parameters.
- ✓ **Note:** Function prototypes are also known as *function declarations*.
- ✓ **WARNING!** You must either place the function definition or the function prototype ahead of all calls to the function. Otherwise the program will not compile. Function prototypes are usually placed near the top of a program so the compiler will encounter them before any function calls.
- ✓ The following program segment uses *function prototype* to declare functions

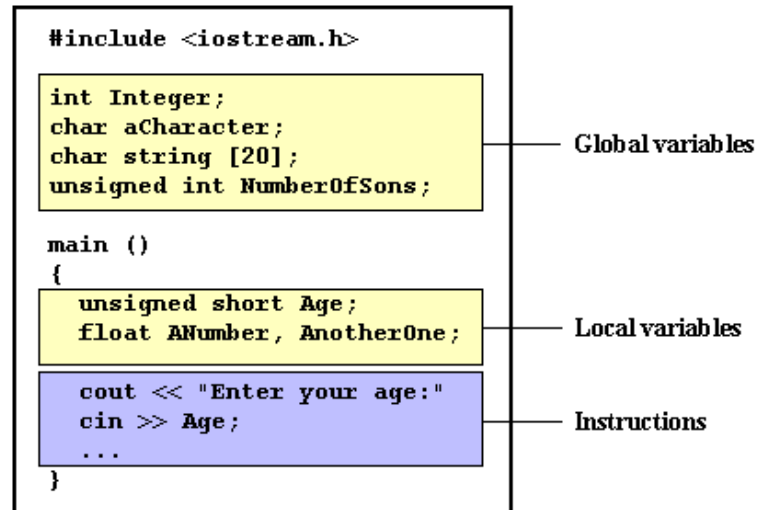
//Function prototypes

```
void first();
void second();
int main()
{
    cout << "I am starting in function main.\n";
    first(); // Call function first
    second(); // Call function second
    cout << "Back in function main again.\n";
}
void first()
{
    cout << "I am now inside the function first.\n";
}
void second()
{
    cout << "I am now inside the function second.\n";
}
```

- ✓ The compiler encounters the calls to the functions *first* and *second* in *main* before it has read the definition of those functions. Because of the function prototypes, however, the compiler already knows the return type and parameter information of *first* and *second*. There should be a prototype for each function in a program except *main*. A prototype is never needed for *main* since it is the starting point of the program.

Local and global variables

- ✓ A local variable is defined inside a function and is not accessible outside the function. A global variable is defined outside all functions and is accessible to all functions in its scope.



Local variables

- ✓ Just as you've defined variables inside function `main`, you may also define them inside other functions. Variables defined inside a function are *local* to that function. They are hidden from the statements in other functions, which normally cannot access them.
- ✓ The following program segment shows that because the variables defined in a function are hidden, other functions may have separate, distinct variables with the same name.

```
int main()
{
    int num = 1; // Local variable
    cout << "In main, num is " << num << endl;
    anotherFunction();
    cout << "Back in main, num is still " << num << endl;
    return 0;
}

void anotherFunction()
{
    int num = 20; // Local variable
    cout << "In anotherFunction, num is " << num << endl;
}
```

- ✓ Even though there are two variables named `num`, the program can only “see” one of them at a time. When the program is executing in `main`, the `num` variable defined in `main` is visible. When `anotherFunction` is called, however, only variables defined inside it are visible, so the `num` variable in `main` is hidden.

Global variables

- ✓ Although local variables are safely hidden from other functions, they do not provide a convenient way of sharing data. When large amounts of data must be accessible to all the functions in a program, global variables are an easy alternative.
- ✓ A global variable is any variable defined outside all the functions in a program. The scope of a global variable is the portion of the program from the variable definition to the end.
- ✓ The following program segment shows two functions, `main` and `anotherFunction`, which access the same global variable, `num`.

```
int num = 2; // Global variable
int main()
{
    cout << "In main, num is " << num << endl;
    anotherFunction();
    cout << "Back in main, num is " << num << endl;
}
void anotherFunction()
{
    cout << "In anotherFunction, num is " << num << endl;
    num = 50;
    cout << "But, it is now changed to " << num << endl;
}
```

- ✓ In the above program segment, `num` is defined outside of all the functions. Because its definition appears before the definitions of `main` and `anotherFunction`, both functions have access to it.
- ✓ Unless you explicitly initialize numeric global variables, they are automatically initialized to zero. The variable `globalNum` in the following program is never set to any value by a statement, but because it is global, it is automatically set to zero.

```
int globalNum; // Global variable. Automatically set to zero.
int main()
{
    cout << "globalNum is " << globalNum << endl;
    return 0;
}
```

- ✓ **Note:** Remember that local variables are not automatically initialized like global variables are. The programmer must handle this.

- ✓ If a function has a local variable with the same name as a global variable, only the local variable can be seen by the function.
- ✓ Also, when two or more functions modify the same variable, you must be careful that what one function does will not upset the correctness of another function. Although global variables make it easy to share data, they should be used carefully and sparingly.

4.4 Sending data into a function

- ✓ When a function is called, the program may send values into the function.
- ✓ Values that are sent into a function are called *arguments*. You're already familiar with how to use arguments in a function call. In the following statement the function `pow` is being called with two arguments, 2 and 4, passed to it:

```
result = pow(2, 4);
```

- ✓ A *parameter* is a special variable that holds a value being passed as an argument into a function.
- ✓ By using parameters, you can design your own functions that accept data this way. Here is the definition of a function that uses a parameter:

```
void displayValue(int num)
{
    cout << "The value is " << num << endl;
}
```

- ✓ Notice the integer variable definition inside the parentheses (`int num`). The variable `num` is a parameter. This enables the function `displayValue` to accept an integer value as an argument.
- ✓ The values that are passed into a function are called *arguments*, and the variables that receive those values are called *parameters*.
- ✓ The following program demonstrates a function with a parameter.

```
#include <iostream.h>
void displayValue(int); // Function prototype
int main()
{
    cout << "I am passing 5 to displayValue.\n";
    displayValue(5); // Call displayValue with argument 5
    cout << "Now I am back in main.\n";
    return 0;
}
// Definition of function displayValue
void displayValue(int num)
{
    cout << "The value is " << num << endl;
}
```

- ✓ Notice the function prototype for `displayValue`:

```
void displayValue(int); // function prototype
```

- ✓ It is not necessary to list the name of the parameter variable inside the parentheses. Only its data type is required. This function prototype could optionally have been written as `void displayValue(int num);`
- ✓ However, the compiler ignores the name of the parameter variable in the function prototype. In main, the `displayValue` function is called with the argument 5 inside the parentheses. The number 5 is passed into `num`, which is `displayValue`'s parameter. This is illustrated in the following figure:

```
displayValue(5);           // function call
    |
    |
    v
void displayValue(int num) // function header
{
    cout << "The value is " << num << endl;
}
```

- ✓ Any argument listed inside the parentheses of a function call is copied into the function's parameter variable. In essence, parameter variables are initialized to the value of their corresponding arguments.
- ✓ The following program fragment shows the function `displayValue` being called several times with a different argument being passed each time.

```
displayValue(5); // Call displayValue with argument 5
displayValue(10); // Call displayValue with argument 10
displayValue(2); // Call displayValue with argument 2
displayValue(16); // Call displayValue with argument 16
```

- ✓ Each time the function is called, `num` takes on a different value. Any expression whose value could normally be assigned to `num` may be used as an argument. For example, the following function call would pass the value 8 into `num`:

```
displayValue(3 + 5);
```

- ✓ If you pass an argument whose type is not the same as the parameter's type, the argument will be promoted or demoted automatically. For instance, because `displayValue`'s parameter is type `int`, the argument in the following function call would be truncated, causing the value 4 to be passed to `num`:

```
displayValue(4.7);
```

- ✓ Often, it's useful to pass several arguments into a function. The following program segment shows the function of a function with three parameters.

```
void showSum(int, int, int); // Function prototype
```

```
int main()
{
    int value1, value2, value3;
    cin >> value1 >> value2 >> value3;
```

```
showSum(value1, value2, value3); // Call showSum with 3 arguments.
}
```

```
void showSum(int num1, int num2, int num3)
{
    cout << (num1 + num2 + num3) << endl;
}
```

- ✓ In the function header for `showSum`, the parameter list contains three variable definitions separated by commas:

```
void showSum(int num1, int num2, int num3)
```

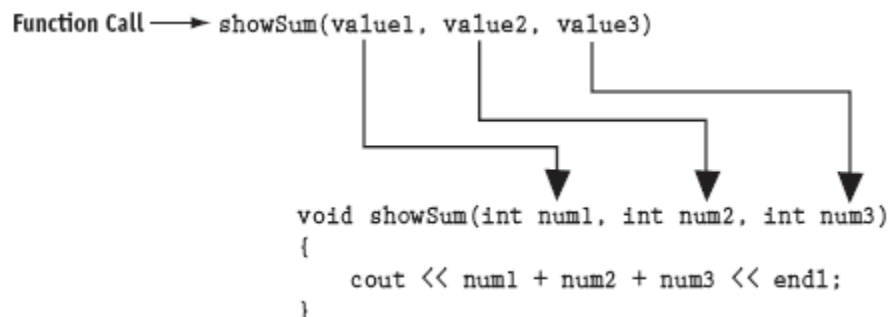
- ✓ **WARNING!** Each variable must have a data type listed before its name. You can't leave out the data type of any variable in the parameter list. A compiler error would occur if the parameter list were declared as `int num1, num2, num3` instead of `int num1, int num2, int num3`. In the function call, the variables `value1`, `value2`, and `value3` are passed as arguments:

```
showSum(value1, value2, value3);
```

- ✓ Notice the syntax difference between the *function header* and the *function call* when passing variables as arguments into parameters. In a function call, you do *not* include the variable's data types inside the parentheses. For example, it would be an error to write this function call as

```
showSum(int value1, int value2, int value3); // Wrong!
```

- ✓ When a function with multiple parameters is called, the arguments are passed to the parameters in order. This is illustrated in the following figure.



- ✓ The following function call will cause 5 to be passed into the `num1` parameter, 10 to be passed into `num2`, and 15 to be passed into `num3`:

```
showSum(5, 10, 15);
```

- ✓ However, the following function call will cause 15 to be passed into the `num1` parameter, 5 to be passed into `num2`, and 10 to be passed into `num3`:

```
showSum(15, 5, 10);
```

- ✓ **Note:** Like all variables, parameters have a scope. The scope of a parameter is limited to the body of the function which uses it.

4.5 Passing data by value

- ✓ When an argument is passed into a parameter by value, only a copy of the argument's value is passed. Changes to the parameter do not affect the original argument.
- ✓ Parameters are special-purpose variables that are defined inside the parentheses of a function definition. Their purpose is to hold the information passed to them by the arguments, which are listed inside the parentheses of a function call.
- ✓ Normally when information is passed to a function it is *passed by value*. This means the parameter receives a copy of the value that is passed to it. If a parameter's value is changed inside a function, it has no effect on the original argument.
- ✓ For example, suppose that we called the function **addition** using the following code :

```
int x=5, y=3, z;
```

```
z = addition ( x , y );
```

- What we did in this case was to call function **addition** passing the values of **x** and **y**, that means **5** and **3** respectively, not the variables themselves.

```
int addition (int a, int b)
```

```

      ↑5      ↑3
z = addition ( x , y );
```

- This way, when function **addition** is being called the value of its variables **a** and **b** become **5** and **3** respectively, but any modification of **a** or **b** within the function **addition** will not affect the values of **x** and **y** outside it, because variables **x** and **y** were not passed themselves to the the function, only their values.
- ✓ The following program segment demonstrates this concept.

```
void changeThem(int, double);
```

```
int main()
```

```
{
```

```
int whole = 12;
```

```
double real = 3.5;
```

```
cout << " whole is " << whole << " and real is " << real << endl;
```

```
changeThem(whole, real); // Call changeThem with 2 arguments
```

```
cout << " whole is " << whole << " and real is " << real << endl;
```

```
}
```

```
void changeThem(int i, double d)
```

```
{
```

```
i = 100;
```

```
d = 27.5;
```

```
cout << " i is " << i << " and d is " << d << endl;
```

```
}
```

- ✓ Even though the parameters **i** and **f** are changed in the function **changeThem**, the arguments **whole** and **real** are not modified. The parameters **i** and **f** only contain copies of **whole** and **real**. The **changeThem** function does not have access to the original arguments.

4.6 Using reference variables as parameters

- ✓ A *reference variable* is an alias for another variable. When used as a parameter, a reference variable allows a function to access the parameter's original argument. Any change to the parameter is actually made to the original argument.
- ✓ When arguments are normally passed to a function by value, parameters receive only a copy of the value sent to them, which they store in the function's local memory. Any changes made to the parameter's value do not affect the value of the original argument.
- ✓ Sometimes, however, we want a function to be able to change a value in the calling function (i.e., the function that called it). This can be done by making the parameter a *reference variable*. A reference variable is an alias for another variable. Any change made to the reference variable is actually performed on the variable it is an alias for.
- ✓ When we use a reference variable as a parameter, it becomes an alias for the corresponding variable in the argument list. Any change made to the parameter is actually made to the variable in the calling function. When data is passed to a parameter in this manner, the argument is said to be *passed by reference*.
- ✓ When passing a variable *by reference* we are passing the variable itself and any modification that we do to that parameter within the function will have effect in the passed variable outside it.

```
void duplicate (int& a,int& b,int& c)
               ↑x   ↑y   ↑z
               ↓x   ↓y   ↓z
duplicate (  x   ,   y   ,   z   );
```

- ✓ To express it another way, we have associated **a**, **b** and **c** with the parameters used when calling the function (**x**, **y** and **z**) and any change that we do on **a** within the function will affect the value of **x** outside. Any change that we do on **b** will affect **y**, and the same with **c** and **z**.
- ✓ Reference variables are defined like regular variables, except there is an ampersand (&) in front of the name. For example, the following function definition makes the parameter refVar a reference variable:

```
void doubleNum(int &refVar)
{
refVar *= 2;
}
```

- ✓ **Note:** The variable refVar is called “a reference to an int.” This function doubles refVar by multiplying it by 2. Since refVar is a reference variable, this action is actually performed on the variable that was passed to the function as an argument.

- ✓ When prototyping a function with a reference variable, be sure to include the ampersand after the data type. Here is the prototype for the doubleNum function:

```
void doubleNum(int &);
```

- ✓ Some programmers prefer not to put a space between the data type and the ampersand. The following prototype is equivalent to the one above:

```
void doubleNum(int&);
```

- ✓ **Note:** The ampersand must appear in both the prototype and the header of any function that uses a reference variable as a parameter. It does not appear in the function call.
- ✓ The following program segment demonstrates the use of a parameter that is a reference variable.

//Function prototype. The parameter is a reference variable.

```
void doubleNum(int &);
int main()
{
    int value = 4;
    cout << "In main, value is " << value << endl;
    doubleNum(value);
    cout << "Now back in main, value is " << value << endl;
}
void doubleNum (int &refVar)
{
    refVar *= 2;
}
```

- ✓ The parameter refVar in “points” to the value variable in function main. When a program works with a reference variable, it is actually working with the variable it references, or points to.
- ✓ **Note:** Only variables may be passed by reference. If you attempt pass a non-variable argument, such as a literal, a constant, or an expression, into a reference parameter, an error will result. Using the doubleNum function as an example, the following function calls will both generate an error.

```
doubleNum(5); // Error
```

```
doubleNum(value + 10); // Error
```

- ✓ If a function uses more than one reference variable as a parameter, be sure to place the ampersand before each reference variable name. Here is the prototype and definition for a function that uses four reference variable parameters:

// Function prototype with four reference variables as parameters.

```
void addThree(int &, int &, int &, int &);
```

// Definition of addThree. All four parameters are reference variables.

```
void addThree(int &sum, int &num1, int &num2, int &num3)
```

```
{
    cout << "Enter three integer values: ";
    cin >> num1 >> num2 >> num3;
```

```
sum = num1 + num2 + num3;
}
```

Default arguments

- ✓ Default arguments are passed to parameters automatically if no argument is provided in the function call.
- ✓ It's possible to assign *default arguments* to function parameters. A default argument is passed to the parameter when the actual argument is left out of the function call. The default arguments are usually listed in the function prototype.

✓ Here is an example:

```
void showArea(double = 20.0, double = 10.0);
```

- ✓ Default arguments are literal values or constants with an = operator in front of them, appearing after the data types listed in a function prototype. Because parameter names are optional in function prototypes, the example prototype could also be declared as

```
void showArea(double length = 20.0, double width = 10.0);
```

- ✓ In both example prototypes, the function showArea has two double parameters. The first is assigned the default argument 20.0 and the second is assigned the default argument 10.0.

✓ Here is the definition of the function:

```
void showArea(double length, double width)
{
    double area = length * width;
    cout << "The area is " << area << endl;
}
```

- ✓ The default argument for length is 20.0 and the default argument for width is 10.0. Because both parameters have default arguments, they may optionally be omitted in the function call, as shown here:

```
showArea();
```

- ✓ In this function call, both default arguments will be passed to the parameters. Parameter length will receive the value 20.0 and width will receive the value 10.0. The output of the function will be

The area is 200

- ✓ The default arguments are only used when the actual arguments are omitted from the function call. In the following call, the first argument is specified, but the second is omitted:

```
showArea(12.0);
```

- ✓ The value 12.0 will be passed to length, while the default value 10.0 will be passed to width. The output of the function will be

The area is 120

- ✓ Of course, all the default arguments may be overridden. In the following function call, arguments are supplied for both parameters:

```
showArea(12.0, 5.5);
```

- ✓ The output of this function call will be

The area is 66

- ✓ **Note:** A function's default arguments should be assigned in the earliest occurrence of the function name. This will usually be the function prototype. However, if a function does not have a prototype, default arguments may be specified in the function header.

- ✓ The showArea function could be defined as follows:

```
void showArea(double length = 20.0, double width = 10.0)
{
    double area = length * width;
    cout << "The area is " << area << endl;
}
```

- ✓ Although C++'s default arguments are very convenient, they are not totally flexible in their use. When an argument is left out of a function call, all arguments that come after it must be left out as well. In the showArea function, it is not possible to omit the first argument without omitting the second argument. For example, the following function call would be illegal:

showArea(, 5.5); // Illegal function call!

- ✓ It's possible for a function to have some parameters with default arguments and some without. For example, in the following function, only the last parameter has a default argument:

```
// Function prototype
void calcPay(int empNum, double payRate, double hours = 40.0);
// Definition of function calcPay
void calcPay(int empNum, double payRate, double hours)
{
    double wages;
    wages = payRate * hours;
    cout << "Gross pay for employee number ";
    cout << empNum << " is " << wages << endl;
}
```

- ✓ When calling this function, arguments must always be specified for the first two parameters (empNum and payRate) because they have no default arguments. Here are examples of valid calls:

calcPay(769, 15.75); // Uses default argument for hours

calcPay(142, 12.00, 20); // Specifies number of hours

- ✓ When a function uses a mixture of parameters with and without default arguments, the parameters with default arguments must be declared last. In the calcPay function, hours could not have been declared before either of the other parameters. The following prototypes are illegal:

// Illegal prototype

void calcPay(int empNum, double hours = 40.0, double payRate);

// Illegal prototype

```
void calcPay(double hours = 40.0, int empNum, double payRate);
```

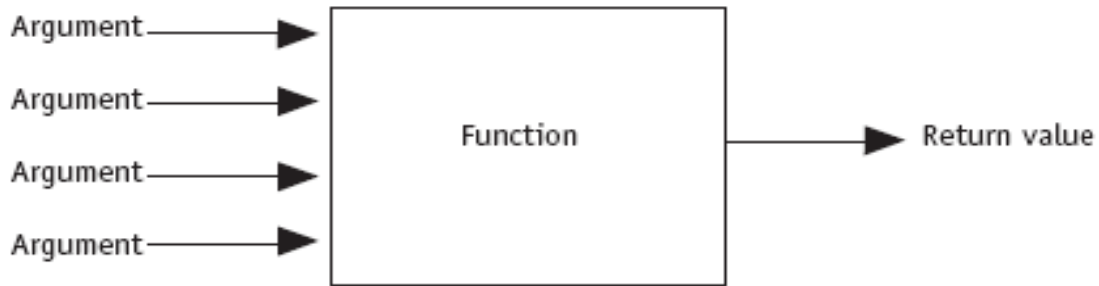
- ✓ Here is a summary of the important points about default arguments:
 - The value of a default argument must be a literal value or a named constant.
 - When an argument is left out of a function call (because it has a default value), all the arguments that come after it must also be left out.
 - When a function has a mixture of parameters both with and without default arguments, the parameters with default arguments must be defined last.
-

The return statement

- ✓ The return statement causes a function to end immediately.
- ✓ When the last statement in a function has finished executing, the function terminates. The program returns to the module that called it and continues executing from the point immediately following the function call. It is possible, however, to force a function to return to where it was called from before its last statement has been executed.
- ✓ When the return statement is encountered, the function immediately terminates and the program returns.
- ✓ The following function, *divide*, shows the quotient of *arg1* divided by *arg2*. If *arg2* is set to zero, the function returns without performing the division.

```
void divide(double arg1, double arg2)
{
  if (arg2 == 0.0)
  {
    cout << "Sorry, I cannot divide by zero.\n";
    return;
  }
  cout << "The quotient is " << (arg1 / arg2) << endl;
}
```

- ✓ Although several arguments can be passed into a function, only one value can be returned from it. Think of a function as having multiple communications channels for receiving data (parameters), but only one channel for sending data (the return value). This is illustrated in the following figure.
- ✓ A function may send a value back to the part of the program that called the function.



- ✓ The data type of the return value precedes the function name in the function header and prototype. The following prototype declares a function named *square* that accepts an integer argument and returns an integer:

```
int square(int);
```

- ✓ Here is the definition of the function:

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

- ✓ This function only has one line, which is a return statement. When a function returns a value, it must have a *return* statement. The expression that follows the return key word is evaluated, converted to the data type the function returns, and sent back to the part of the program that called the function. This is demonstrated in the following program segment.

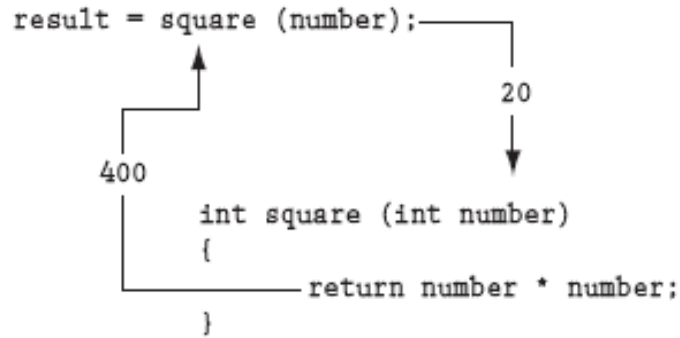
```
int main()
{
    int number, result;
    cout << "Enter a number and I will square it: ";
    cin >> number;
    result = square(number);
    cout << number << " squared is " << result << endl;
}

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

- ✓ Here is the line that calls the square function:

```
result = square(number);
```

- ✓ An expression is something that has a value. If a function returns a value, a call to that function is an expression. The statement above assigns the value returned from *square* to the variable *result*. For example, when 20 is passed as an argument into *square*, 20 times 20, or 400, is returned and assigned to *result*.
- ✓ The following figure illustrates how information is passed to and returned from the function.



✓ Actually, the *result* variable is unnecessary in program segment. The return value of the *square* function could have been displayed by the `cout` object, as shown here:

```
cout << number << " squared is " << square(number) << endl;
```

✓ It is also possible to use a value returned by a function in a relational test or in an arithmetic expression. For example, the following two statements are both perfectly legal:

```
if (square(number) > 100)
```

```
cout << "big square\n";
```

here, *square* function is called and the returned value is used in a relational test

```
sum = 1000 + square(number);
```

here,

square function is called and the returned value is used in an arithmetic expression.

✓ The following program segment uses a `getRadius` function to get the radius of the circle from the user and return that value back to main. This function accepts no arguments and returns a double.

```
int main()
{
    const double PI = 3.14159;
    double rad;
    cout << "This program calculates the area of a circle.\n";
    rad = getRadius();
    cout << "The area is " << PI * square(rad) << endl;
}
double getRadius()
{
    double radius;
    cout << "Enter the radius of the circle: ";
    cin >> radius;
    return radius;
}
double square(double number)
{
```

```
return number * number;
```

```
}
```

- ✓ The return type of a function should be the type of the data you wish to return from the function. If a function is returning a double value that is being assigned to a variable, then that variable should also be a double. If the double value being returned by the square function were assigned to an int variable, the value would be truncated.

- ✓ This is illustrated in the following example:

```
int result;
```

```
result = square(2.7);
```

- ✓ The square function returns the value 7.29, but it is truncated to 7 when it is stored in result.
 - ✓ **Note:** Remember, If you give a function a return type other than void, you must have a return statement in that function.
-

4.7 Recursion

- ✓ A recursive function is one that calls itself.
- ✓ You have seen instances of functions calling other functions. Function A can call function B, which can then call Function C. It's also possible for a function to call itself. A function that calls itself is a *recursive function*. Look at this message function:

```
void message()
```

```
{
```

```
cout << "This is a recursive function.\n";
```

```
message();
```

```
}
```

- ✓ This function displays the string “*This is a recursive function.*”, and then calls itself. Each time it calls itself, the cycle is repeated. Can you see a problem with the function? There's no way to stop the recursive calls. This function is like an infinite loop because there is no code to stop it from repeating.
- ✓ To be useful, a recursive function must have a way of controlling the number of recursive calls. The following is a modification of the message function. It passes an integer argument that holds the number of times the function is to call itself.

```
void message(int times)
```

```
{
```

```
if (times > 0)
```

```
{
```

```
cout << "This is a recursive function.\n";
```

```
message(times - 1);
```

```
}
```

```
}
```

- ✓ This function contains an if statement that controls the repetition. As long as the times argument is greater than zero, it will display the message and call itself again. Each time it

calls itself, it passes `times-1` as the argument. For example, let's say a program calls the function with the following statement:

```
message(5);
```

- ✓ The argument, 5, will cause the function to call itself six times. The first time the function is called, the if statement will display the message and call itself with 4 as the argument.
- ✓ Each time the function *message* is called, a new instance of the *times* parameter is created in memory. The first time the function is called, the *times* parameter is set to 5. When the function calls itself, a new instance of *times* is created, and the value 4 is passed into it. This cycle repeats until zero is passed to the function.
- ✓ The function will call itself six times, so the *depth of recursion* is six. When the function reaches its sixth call, the *times* parameter will be set to 0. At that point, the if statement will stop the recursive chain of calls and the sixth instance of the function will return.
- ✓ Control of the program will return from the sixth instance of the function to the point in the fifth instance directly after the recursive function call:

```
if (times > 0)
{
    cout << "This is a recursive function.\n";
    message(times - 1);
}
```

 *Control returns here.*

- ✓ Because there are no more statements to be executed after the function call, the fifth instance of the function returns control of the program back to the fourth instance. This repeats until all instances of the function return.
- ✓ The following program segment demonstrates the recursive message function.

```
void message(int);
```

```
int main()
```

```
{
```

```
message(5);
```

```
return 0;
```

```
}
```

```
void message(int times)
```

```
{
```

```
if (times > 0)
```

```
{
```

```
cout << "This is a recursive function.\n";
```

```
message(times - 1);
```

```
}
```

```
}
```


4.7.1 The recursive factorial function

- ✓ The recursive factorial function accepts an argument and calculates its factorial.
- ✓ Its base case is when the argument is 0. Let's use an example from mathematics to examine an application of recursion. In mathematics, the notation $n!$ represents the factorial of the number n . The factorial of a number is defined as

$$n! = 1 \times 2 \times 3 \times \dots \times n; \quad \text{if } n > 0$$

$$= 1; \text{ if } n = 0$$

- ✓ The rule states that when n is greater than 0, its factorial is the product of all the positive integers from 1 up to n . For instance, $6!$ can be calculated as $1 \times 2 \times 3 \times 4 \times 5 \times 6$. The rule also specifies the base case: the factorial of 0 is 1.
- ✓ We can define the factorial of a number using recursion as follows:

$$\text{factorial}(n) = n \times \text{factorial}(n - 1) \text{ if } n > 0$$

$$= 1; \text{ if } n = 0$$

- ✓ The C++ implementation of this recursive definition is

```
int factorial(int num)
{
    if (num == 0) // base case
        return 1;
    else
        return num * factorial(num - 1);
}
```

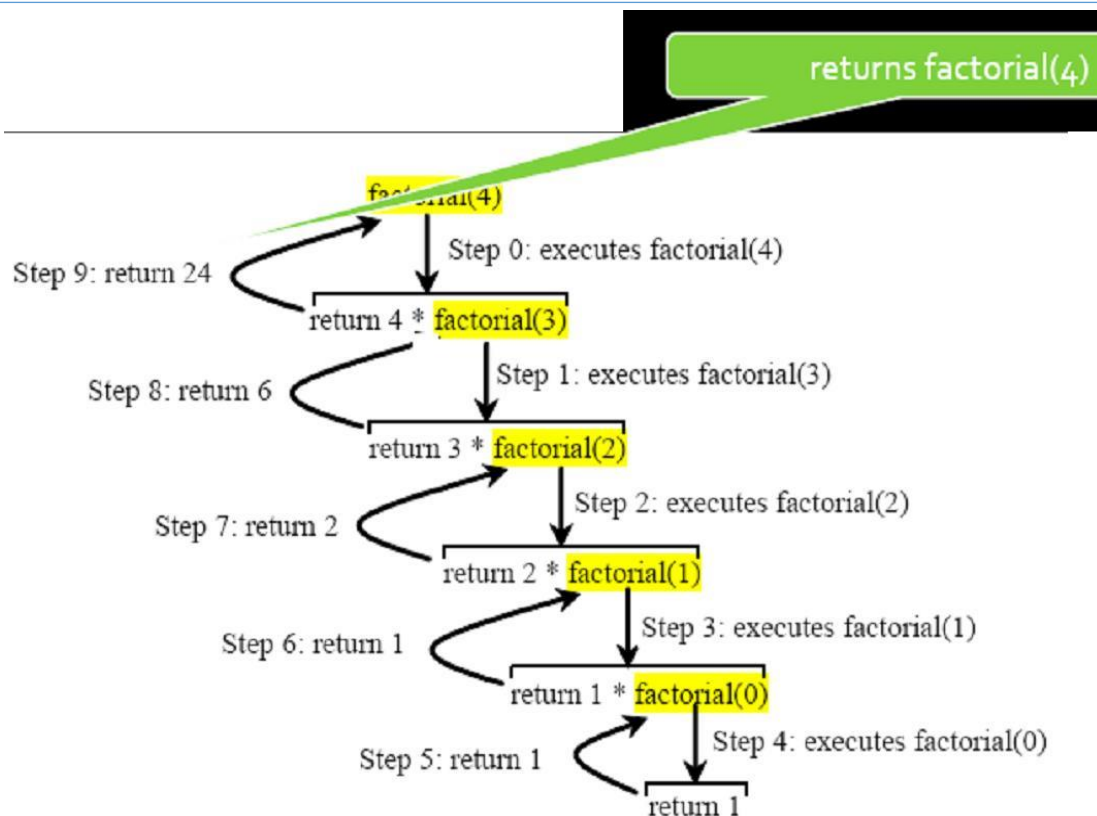
- ✓ Consider a program that displays the value of $4!$ with the following statement:

```
cout << factorial(4) << endl;
```

- ✓ The first time the function is called, `num` is set to 4. The if statement will execute the following line:

```
return num * factorial(num - 1);
```

- ✓ Although this is a return statement, it does not immediately return. Before the return value can be determined, the value of `factorial(num - 1)` must be determined. The function is called recursively until the fifth call, in which the `num` parameter will be set to zero.
- ✓ The diagram in following figure illustrates the value of the argument and the return value during each call of the function.



4.7.2 The recursive GCD function

- ✓ The gcd function uses recursion to find the greatest common divisor (gcd) of two numbers. Our next example of recursion is the calculation of the greatest common divisor, or gcd, of two numbers. Using Euclid's algorithm, the gcd of two positive integers, x and y , is
 $\text{gcd}(x, y) = y$; if y divides x evenly
 $= \text{gcd}(y, \text{remainder of } x/y)$; otherwise
- ✓ This definition states that the gcd of x and y is y if x/y has no remainder. Otherwise, the answer is the gcd of y and the remainder of x/y .
- ✓ The following program segment shows the recursive C++ implementation to calculate the greatest common divisor (gcd) of two numbers.

```
int gcd(int x, int y)
{
    if (x % y == 0) // base case
        return y;
    else
        return gcd(y, x % y);
}
```

The exit() function

- ✓ The exit() function causes a program to terminate, regardless of which function or control mechanism is executing.
- ✓ A C++ program stops executing when a return statement in function main is encountered. When other functions end, however, the program does not stop. Control of the program goes back to the place immediately following the function call.
- ✓ Sometimes, however, the programmer wishes, under certain conditions, to terminate a program in a function other than main. To accomplish this, the exit function is used.
- ✓ When the exit function is called, it causes the program to stop, regardless of which function contains the call. The following program demonstrates this.

```
void someFunction()
```

```
{  
    cout << "This program terminates with the exit function.\n Bye!\n";  
    exit(0);  
    cout << "This message will never be printed."  
    cout << " The program has already terminated.\n";  
}
```

- ✓ To use the exit function, be sure to include the stdlib header file. Notice the function takes an integer argument. This argument is the exit code you wish the program to pass back to the computer's operating system. In the program, the exit code zero is passed, which commonly indicates a successful exit.

Chapter 5: Arrays, Pointers & Strings

5.1 Introduction

A collection of identical data objects, which are stored in consecutive memory locations under a common heading or a variable name. In other words, an array is a group or a table of values referred to by the same name. The individual values in array are called elements. Array elements are also variables.

Set of values of the same type, which have a single name followed by an index. In C++, square brackets appear around the index right after the name. A block of memory representing a collection of many simple data variables stored in a separate array element, and the computer stores all the elements of an array consecutively in memory.

Properties of arrays:

Arrays in C++ are zero-bounded; that is the index of the first element in the array is 0 and the last element is N-1, where N is the size of the array.

It is illegal to refer to an element outside of the array bounds, and your program will crash or have unexpected results, depending on the compiler. Array can only hold values of one type

Array declaration

Declaring the name and type of an array and setting the number of elements in an array is called dimensioning the array. The array must be declared before one uses it like other variables. In the array declaration one must define:

1. The type of the array (i.e. integer, floating point, char etc.)
2. Name of the array,
3. The total number of memory locations to be allocated or the maximum value of each subscript. i.e. the number of elements in the array.

So the general syntax for the declaration is:

```
DataType name [array size];
```

The expression array size, which is the number of elements, must be a constant such as 10 or a symbolic constant declared before the array declaration, or a constant expression such as `10*sizeof(int)`, for which the values are known at the time compilation takes place.

Note: array size cannot be a variable whose value is set while the program is running.

Thus to declare an integer with size of 10 having a name of num is:

```
int num [10];
```

This means: ten consecutive two byte memory location will be reserved with the name num.

That means, we can store 10 values of type **int** without having to declare 10 different variables each one with a different identifier. Instead of that, using an array we can store 10 different values of the same type, **int** for example, with a unique identifier.

What is an Array?: An **array** is a data structure which allows a collective name to be given to a group of elements which ***all have the same type***. An individual element of an array is identified by its own unique ***index*** (or **subscript**).

An array can be thought of as a collection of numbered boxes each containing one data item. The number associated with the box is the index of the item. To access a particular item the index of the box associated with the item is used to access the appropriate box. The index **must** be an integer and indicates the position of the element in the array. Thus the elements of an array are **ordered** by the index.

5.2 One Dimensional Array

Declaration of Arrays

An array declaration is very similar to a variable declaration. First a type is given for the elements of the array, then an identifier for the array and, within square brackets, the number of elements in the array. The number of elements **must be an integer**.

For example data on the average temperature over the year in Ethiopia for each of the last 100 years could be stored in an array declared as follows:

```
float annual_temp[100];
```

This declaration will cause the compiler to allocate space for 100 consecutive float variables in memory. The number of elements in an array must be fixed at compile time. It is best to make the array size a constant and then, if required, the program can be changed to handle a different size of array by changing the value of the constant,

```
const int NE = 100;
```

```
float annual_temp[NE];
```

then if more records come to light it is easy to amend the program to cope with more values by changing the value of NE. This works because the compiler knows the value of the constant NE at compile time and can allocate an appropriate amount of space for the array. It would not work if an ordinary variable was used for the size in the array declaration since at compile time the compiler would not know a value for it.

Initializing Arrays

When declaring an array of local scope (within a function), if we do not specify the array variable will not be initialized, so its content is undetermined until we store some values in it. If we declare a global array (outside any function) its content will be initialized with all its elements filled with zeros. Thus, if in the global scope we declare:

```
int day [5];
```

every element of day will be set initially to 0:

	0	1	2	3	4
day	0	0	0	0	0

But additionally, when we declare an Array, we have the possibility to assign initial values to each one of its elements using curly brackets { }. For example:

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

The above declaration would have created an array like the following one:

	0	1	2	3	4
day	16	2	77	40	12071

The number of elements in the array that we initialized within curly brackets { } must be equal or less than the length in elements that we declared for the array enclosed within

square brackets []. If we have less number of items for the initialization, the rest will be filled with zero.

For example, in the example of the day array we have declared that it had 5 elements and in the list of initial values within curly brackets { } we have set 5 different values, one for each element. If we ignore the last initial value (12071) in the above initialization, 0 will be taken automatically for the last array element.

Because this can be considered as useless repetition, C++ allows the possibility of leaving empty the brackets [], where the number of items in the initialization bracket will be counted to set the size of the array.

```
int day [] = { 1, 2, 7, 4, 12,9 };
```

The compiler will count the number of initialization items which is 6 and set the size of the array day to 6 (i.e.: day[6])

You can use the initialization form only when defining the array. You cannot use it later, and cannot assign one array to another once. I.e.

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

```
int ar [4];
```

```
ar[]={1,2,3,4};//not allowed
```

```
arr=ar;//not allowed
```

Note: when initializing an array, we can provide fewer values than the array elements. E.g. `int a [10] = { 10, 2, 3 };` in this case the compiler sets the remaining elements to zero.

5.2.1 Accessing Array Elements

Given the declaration above of a 100-element array the compiler reserves space for 100 consecutive floating point values and accesses these values using an index/subscript that takes values from 0 to 99. The first element in an array in C++ always has the index 0, and if the array has *n* elements the last element will have the index n-1.

An **array element** is accessed by writing the identifier of the array followed by the subscript in square brackets. Thus to set the 15th element of the array above to 1.5 the following assignment is used:

```
annual_temp[14] = 1.5;
```

Note that since the first element is at index 0, then the ith element is at index i-1. Hence in the above the 15th element has index 14.

An array element can be used anywhere an identifier may be used. Here are some examples assuming the following declarations:

```
const int NE = 100,
```

```
    N = 50;
```

```
int i, j, count[N];
```

```
float annual_temp[NE];
```

```
float sum, av1, av2;
```

A value can be read into an array element directly, using `cin`

```
cin >> count[i];
```

The element can be increased by 5,

```
count[i] = count[i] + 5;
```

or, using the shorthand form of the assignment

```
count[i] += 5;
```

Array elements can form part of the condition for an if statement, or indeed, for any other logical expression:

```
if (annual_temp[j] < 10.0)
    cout << "It was cold this year " << endl;
```

for statements are the usual means of accessing every element in an array. Here, the first NE elements of the array annual_temp are given values from the input stream cin.

```
for (i = 0; i < NE; i++)
    cin >> annual_temp[i];
```

The following code finds the average temperature recorded in the first ten elements of the array.

```
sum = 0.0;
for (i = 0; i < 10; i++)
    sum += annual_temp[i];
av1 = sum / 10;
```

Notice that it is good practice to use named constants, rather than literal numbers such as 10. If the program is changed to take the average of the first 20 entries, then it all too easy to forget to change a 10 to 20. If a const is used consistently, then changing its value will be all that is necessary.

For example, the following example finds the average of the last k entries in the array. k could either be a variable, or a declared constant. Observe that a change in the value of k will still calculate the correct average (provided $k \leq NE$).

```
sum = 0.0;
for (i = NE - k; i < NE; i++)
    sum += annual_temp[i];
av2 = sum / k;
```

Important - C++ does not check that the subscript that is used to reference an array element actually lies in the subscript range of the array. Thus C++ will allow the assignment of a value to annual_temp[200], however the effect of this assignment is unpredictable. For example it could lead to the program attempting to assign a value to a memory element that is outside the program's allocated memory space. This would lead to the program being terminated by the operating system. Alternatively it might actually access a memory location that is within the allocated memory space of the program and assign a value to that location, changing the value of the variable in your program which is actually associated with that memory location, or overwriting the machine code of your program. Similarly reading a value from annual_temp[200] might access a value that has not been set by the program or might be the value of another variable. It is the programmer's responsibility to ensure that if an array is declared with n elements then no attempt is made to reference any element with a subscript outside the range 0 to n-1. Using an index, or subscript, that is out of range is called Subscript Overflow. Subscript overflow is one of the commonest causes of erroneous results and can frequently cause very strange and hard to spot errors in programs.

5.2.2 Initialization of arrays

The initialization of simple variables in their declaration has already been covered. An array can be initialized in a similar manner. In this case the initial values are given as a list enclosed

in curly brackets. For example initializing an array to hold the first few prime numbers could be written as follows:

```
int primes[] = {1, 2, 3, 5, 7, 11, 13};
```

Note that the array has not been given a size, the compiler will make it large enough to hold the number of elements in the list. In this case primes would be allocated space for seven elements. If the array is given a size then this size must be greater than or equal to the number of elements in the initialization list. For example:

```
int primes[10] = {1, 2, 3, 5, 7};
```

would reserve space for a ten element array but would only initialize the first five elements.

Example Program: Printing Outliers in Data

The requirement specification for a program is:

A set of positive data values (200) are available. It is required to find the average value of these values and to count the number of values that are more than 10% above the average value.

Since the data values are all positive a negative value can be used as a sentinel to signal the end of data entry. Obviously this is a problem in which an array must be used since the values must first be entered to find the average and then each value must be compared with this average.

Hence the use of an array to store the entered values for later re-use.

An initial algorithmic description is:

```
initialize.
enter elements into array and sum elements.
evaluate average.
scan array and count number greater than
    10% above average.
output results.
```

This can be expanded to the complete algorithmic description:

```
set sum to zero.
set count to zero.
set nogt10 to zero.
enter first value.
while value is positive
{
    put value in array element with index count.
    add value to sum.
    increment count.
    enter a value.
}
average = sum/count.
for index taking values 0 to count-1
    if array[index] greater than 1.1*average
        then increment nogt10.
```

output average, count and nogt10.

In the above, the variable **nogt10** is the number greater than 10% above the average value. It is easy to argue that after exiting the while loop, count is set to the number of positive numbers entered. Before entering the loop count is set to zero and the first number is entered, that is count

is one less than the number of numbers entered. Each time round the loop another number is entered and count is incremented hence count remains one less than the number of numbers entered. But the number of numbers entered is one greater than the number of positive numbers so count is therefore equal to the number of positive numbers.

A **main()** program written from the above algorithmic description is given below:

```
void main()
{
    const int NE = 200; // maximum no of elements in array
    float sum = 0.0;    // accumulates sum
    int count = 0;      // number of elements entered
    int nogt10 = 0;     // counts no greater than 10% above average
    float x;            // holds each no as input
    float indata[NE];   // array to hold input
    float average;      // average value of input values
    int i;              // control variable
    // Data entry, accumulate sum and count
    // number of +ve numbers entered
    cout << "Enter numbers, -ve no to terminate: " << endl;
    cin >> x;
    while (x >= 0.0)
    {
        sum = sum + x;
        indata[count] = x;
        count = count + 1;
        cin >> x;
    }
    // calculate average
    average = sum/count;
    // Now compare input elements with average
    for (i = 0; i < count; i++)
    {
        if (indata[i] > 1.1 * average)
            nogt10++;
    }
    // Output results
    cout << "Number of values input is " << n;
    cout << endl
        << "Number more than 10% above average is "
        << nogt10 << endl;
}
```

Since it was assumed in the specification that there would be less than 200 values the array size is set at 200. In running the program less than 200 elements may be entered, if n elements where $n < 200$ elements are entered then they will occupy the first n places in the array `indata`. It is

common to set an array size to a value that is the maximum we think will occur in practice, though often not all this space will be used.

Example Program: Test of Random Numbers

The following program simulates the throwing of a dice by using a random number generator to generate integers in the range 0 to 5. The user is asked to enter the number of trials and the program outputs how many times each possible number occurred.

An array has been used to hold the six counts. This allows the program to increment the correct count using one statement inside the loop rather than using a switch statement with six cases to choose between variables if separate variables had been used for each count. Also it is easy to change the number of sides on the dice by changing a constant. Because C++ arrays start at subscript 0 the count for an *i* occurring on a throw is held in the *i*-1th element of this count array. By changing the value of the constant `die_sides` the program could be used to simulate a `die_sides`-sided die without any further change.

```
#include <iostream.h>
#include <stdlib.h> // time.h and stdlib.h required for
#include <time.h>   // random number generation
void main()
{
    const int die_sides = 6; // maxr-sided die
    int count[die_sides];    // holds count of each
                           // possible value
    int no_trials,           // number of trials
        roll,               // random integer
        i;                  // control variable
    float sample;            // random fraction 0 .. 1

    // initialize random number generation and count
    // array and input no of trials
    srand(time(0));
    for (i=0; i < die_sides; i++)
        count[i] = 0;
    cout << "How many trials? ";
    cin >> no_trials;

    // carry out trials
    for (i = 0; i < no_trials; i++)
    {
        sample = rand()/float(RAND_MAX);
        roll = int ( die_sides * sample);
        // returns a random integer in 0 to die_sides-1
        count[roll]++; // increment count
    }

    // Now output results
    for (i = 0; i < die_sides; i++)
    {
```

```

        cout << endl << "Number of occurrences of "
            << (i+1) << " was " << count[i];
    }
    cout << endl;
}

```

5.3 Multidimensional arrays

An array may have more than one dimension. Each dimension is represented as a subscript in the array. Therefore a two dimensional array has two subscripts, a three dimensional array has three subscripts, and so on.

Arrays can have any number of dimensions, although most of the arrays that you create will likely be of one or two dimensions.

A chess board is a good example of a two-dimensional array. One dimension represents the eight rows, the other dimension represents the eight columns.

Suppose the program contains a class named square. The declaration of array named board that represents would be

```
Square board[8][8];
```

The program could also represent the same data with a one dimensional, 64-square array. For example, it could include the statement

```
Square board[64];
```

Such a representation does not correspond as closely to the real-world object as the two dimensional array, however.

Suppose that when the game begins. The king is located in the fourth position in the first row. Counting from zero that position corresponds to `board[0][3]` in the two dimensional array, assuming that the first subscript corresponds to the row, and the second to the column.

Initializing Multidimensional Arrays

To initialize a multidimensional arrays, you must assign the list of values to array elements in order, with last array subscript changing while the first subscript holds steady. Therefore, if the program has an array **int theArray[5][3]**, the first three elements go into **theArray[0]**; the next three into **theArray[1]**; and so forth.

The program initializes this array by writing

```
int theArray[5][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
```

for the sake of clarity, the program could group the initializations with braces, as shown below.

```
int theArray[5][3] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9}, {10, 11, 12}, {13, 14, 15} };
```

The compiler ignores the inner braces, which clarify how the numbers are distributed.

Each value should be separated by comma, regardless of whether inner braces are included. The entire initialization must set must appear within braces, and it must end with a semicolon.

Omitting the Array Size

If a one-dimensional array is initialized, the size can be omitted as it can be found from the number of initializing elements:

```
int x[] = { 1, 2, 3, 4 } ;
```

This initialization creates an array of four elements.

Note however:

```
int x[][] = { {1,2}, {3,4} } ; // error is not allowed.
```

and must be written

```
int x[2][2] = { {1,2}, {3,4} } ;
```

Example of multidimensional array:

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

```
void main(){
```

```
    int SomeArray[5][2] = {{0,0},{1,2}, {2,4},{3,6}, {4,8}}
```

```
    for ( int i=0; i<5; i++)
```

```
    for (int j = 0; j<2;j++)
```

```
    {
```

```
        cout<<"SomeArray["<<i<<"]["<<j<<"]: ";
```

```
        cout<<endl<<SomeArray[i][ j];
```

```
    }
```

```
}
```

5.4 Address and pointer

We have already seen how variables are memory cells that we can access by an identifier. But these variables are stored in concrete places of the computer memory. For our programs, the computer memory is only a succession of 1 *byte* cells (the minimum size for a datum), each one with a unique address. A pointer is a variable which stores the address of another variable. The only difference between pointer variable and regular variable is the data they hold. There are two pointer operators in C++:

- **&** the address of operator
- ***** the dereference operator

Whenever you see the **&** used with pointers, think of the words “address of.” The **&** operator always produces the memory address of whatever it precedes. The ***** operator, when used with pointers, either declares a pointer or dereferences the pointer’s value. The dereference operator can be literally translated to “*value pointed by*”.

A **pointer** is simply the address of an object in memory. Generally, objects can be accessed in two ways: directly by their symbolic name, or indirectly through a pointer. The act of getting to an object via a pointer to it, is called **dereferencing** the pointer. Pointer variables are defined to point to objects of a specific type so that when the pointer is dereferenced, a typed object is obtained.

At the moment in which we declare a variable this one must be stored in a concrete location in this succession of cells (the memory). We generally do not decide where the variable is to be placed - fortunately that is something automatically done by the compiler and the operating system on runtime, but once the operating system has assigned an address there are some cases in which we may be interested in knowing where the variable is stored.

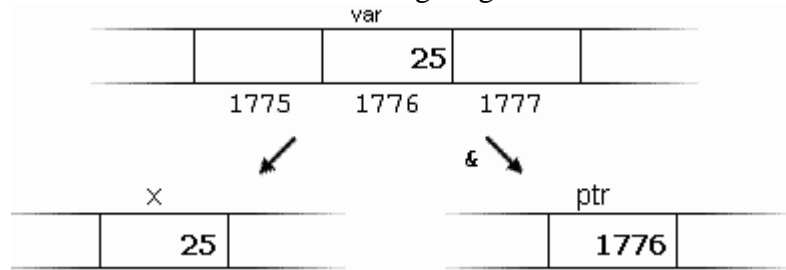
This can be done by preceding the variable identifier by an *ampersand sign* (**&**), which literally means, “*address of*”. For example: `ptr = &var;`

This would assign to variable **ptr** the address of variable **var**, since when preceding the name of the variable **var** with the *ampersand* (**&**) character we are no longer talking about the content of the variable, but about its address in memory.

We are going to suppose that **var** has been placed in the memory address **1776** and that we write the following:

```
var=25;
x=var;
ptr = &var;
```

The result will be the one shown in the following diagram:



We have assigned to **x** the content of variable **var** as we have done in many other occasions in previous sections, but to **ptr** we have assigned the address in memory where the operating system stores the value of **var**, that we have imagined that it was **1776** (it can be any address). The reason is that in the allocation of **ptr** we have preceded **var** with an *ampersand* (&) character.

The variable that stores the address of another variable (like ptr in the previous example) is what we call a pointer.

Declaring Pointers:

Is reserving a memory location for a pointer variable in the heap.

Syntax:

```
type * pointer_name ;
```

to declare a pointer variable called p_age, do the following:

```
int * p_age;
```

Whenever the dereference operator, *, appears in a variable declaration, the variable being declared is always a pointer variable.

Assigning values to pointers:

p_age is an integer pointer. The type of a pointer is very important. p_age can point only to integer values, never to floating-point or other types.

To assign p_age the address of a variable, do the following:

```
int age = 26;
int * p_age; p_age = &age; OR
int age = 26;
int * p_age = & age;
```

Both ways are possible.

If you wanted to print the value of age, do the following:

```
cout<<age;//prints the value of age Or by using pointers you can do it as follows
cout<<*p_age;//dereferences p_age;
```

The dereference operator produces a value that tells the pointer where to point. Without the *, (i.e cout<<p_age), a cout statement would print an address (the address of age). With the *, the cout prints the value at that address.

You can assign a different value to age with the following statement:

```
age = 13; //assigns a new value to variable age
```

*p_age = 13 //assigns 13 as a value to the memory p_age points at. **N.B:** the * appears before a pointer variable in only two places: when you declare a pointer variable and when you dereference a pointer variable (to find the data it points to).

The following program is one you should study closely. It shows more about pointers and the pointer operators, & and *, than several pages of text could do.

```
#...
#...
void main() {
    int num = 123; // a regular integer variable
    int *p_num; //declares an integer pointer
    cout<< "num is "<<num<<endl;
    cout<< "the address of num is "<<&num<<endl;
    p_num = &num; // puts address of num in p_num;
    cout<< "*p_num is "<<*p_num<<endl; //prints value of num
    cout<< "p_num is "<<p_num<<endl; //prints value of P_num getch();
}
```

Pointer to void

Note that we can't assign the address of a float type variable to an integer pointer variable and similarly the address of an integer variable cannot be stored in a float or character pointer.

```
float y;
int x;
int *ip;
float *fp;
ip = &y; //illegal statement
fp = &x; //illegal statement
```

That means, if a variable type and pointer to type is same, then only we can assign the address of variable to pointer variable. And if both are different type then we can't assign the address of variable to pointer variable but this is also possible in C++ by declaring pointer variable as a void as follows:

```
void *p;
```

Let us see an example:

```
void *p; int x; float y;
p = &x; //valid assignment p = &y; //valid assignment
```

The difficulty on void pointers is that, void pointers can not be de referenced.

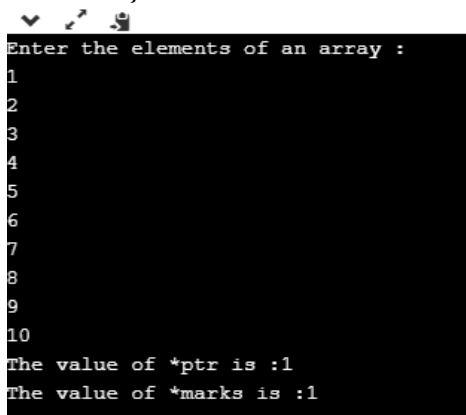
They are aimed only to store address and the dereference operator is not allowed for void pointers.

5.5 Pointer and array

Array and pointers are closely related to each other. In C++, the name of an array is considered as a pointer, i.e., the name of an array contains the address of an element. C++ considers the array name as the address of the first element. For example, if we create an array, i.e., marks which hold the 20 values of integer type, then marks will contain the address of first element,

i.e., marks[0]. Therefore, we can say that array name (marks) is a pointer which is holding the address of the first element of an array.

```
#include <iostream>
using namespace std;
int main()
{
    int *ptr; // integer pointer declaration
    int marks[10]; // marks array declaration
    cout << "Enter the elements of an array :" << endl;
    for(int i=0;i<10;i++)
    {
        cin>>marks[i];
    }
    ptr=marks; // both marks and ptr pointing to the same element..
    cout << "The value of *ptr is :" <<*ptr<<endl;
    cout << "The value of *marks is :" <<*marks<<endl;
}
```



```
Enter the elements of an array :
1
2
3
4
5
6
7
8
9
10
The value of *ptr is :1
The value of *marks is :1
```

In the above code, we declare an integer pointer and an array of integer type. We assign the address of marks to the ptr by using the statement ptr=marks; it means that both the variables 'marks' and 'ptr' point to the same element, i.e., marks[0]. When we try to print the values of *ptr and *marks, then it comes out to be same. Hence, it is proved that the array name stores the address of the first element of an array.

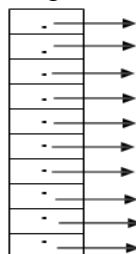
This is the out put

Array of pointers

If you have to reserve many pointers for many different values, you might want to declare an array of pointers. The following reserves an array of 10 integer pointer variables:

```
int *iptr[10]; //reserves an array of 10 integer pointers
```

The above statement will create the following structure in RAM



iptr[4] = &age; // makes iptr[4] point to address of age.

5.6 Pointer and String

If you declare a character table with 5 rows and 20 columns, each row would contain the same number of characters. You can define the table with the following statement.

```
char names[5][20] = {"George"}, {"Mesfin"}, {"John"}, {"Kim"}, {"Barbara"};
```

The above statement will create the following table in memory:

G	e	o	r	g	e	\0													
M	e	s	f	i	n	\0													
J	o	h	n	\0															
K	i	m	\0																
B	a	r	b	a	r	a	\0												

Notice that much of the table is waster space. Each row takes 20 characters, even though the data in each row takes far fewer characters.

To fix the memory-wasting problem of fully justified tables, you should declare a single-dimensional array of character pointers. Each pointer points to a string in memory and the strings do not have to be the same length.

Here is the definition for such an array:

```
char *name[5] = {"George"}, {"Mesfin"}, {"John"}, {"Kim"}, {"Barbara"};
```

This array is a single-dimension array. The asterisk before names makes this array an array of pointers. Each string takes only as much memory as is needed by the string and its terminating zero. At this time, we will have this structure in memory:

- To print the first string, we should use:
cout<<*names; //prints George.
- To print the second use:
cout<< *(names+1); //prints Michael
- Whenever you dereference any pointer element with the * dereferencing operator, you access one of the strings in the array.

5.7 Strings representation and manipulation

String in C++ is nothing but a sequence of character in which the last character is the null character '\0'. The null character indicates the end of the string. Any array of character can be converted into string type in C++ by appending this special character at the end of the array sequence.

In C++ strings of characters are held as an **array of characters**, one character held in each array element. In addition a special **null character**, represented by '\0', is appended to the end of the string to indicate the end of the string. Hence if a string has n characters then it requires an n+1 element array (at least) to store it. Thus the character 'a' is stored in a single byte, whereas the single-character string "a" is stored in two consecutive bytes holding the character 'a' and the null character.

A string variable s1 could be declared as follows:

```
char s1[10];
```

The string variable s1 could hold strings of length up to nine characters since space is needed for the final null character. Strings can be initialized at the time of declaration just as other variables are initialized. For example:

```
char s1[] = "example";
```


char s2[20] = "another example"
would store the two strings as follows:

s1 |e|x|a|m|p|l|e|\0|

s2 |a|n|o|t|h|e|r| |e|x|a|m|p|l|e|\0|?|?|?|?

In the first case the array would be allocated space for eight characters, that is space for the seven characters of the string and the null character. In the second case the string is set by the declaration to be twenty characters long but only sixteen of these characters are set, i.e. the fifteen characters of the string and the null character. Note that the length of a string does not include the terminating null character.

5.7.1 String Output

A string is output by sending it to an output stream, for example:

```
cout << "The string s1 is " << s1 << endl;
```

would print

The string s1 is example

The **setw(width)** I/O manipulator can be used before outputting a string, the string will then be output right-justified in the field width. If the field width is less than the length of the string then the field width will be expanded to fit the string exactly. If the string is to be left-justified in the field then the **setiosflags** manipulator with the argument **ios::left** can be used.

5.7.2 String Input

When the input stream **cin** is used space characters, **newline** etc. are used as separators and terminators. Thus when inputting numeric data **cin** skips over any leading spaces and terminates reading a value when it finds a white-space character (**space**, **tab**, **newline** etc.). This same system is used for the input of strings, hence a string to be input cannot start with leading spaces, also if it has a space character in the middle then input will be terminated on that space character. The null character will be appended to the end of the string in the character array by the stream functions. If the string s1 was initialized as in the previous section, then the statement

cin << s1; would set the string s1 as follows when the string **"first"** is entered (without the double quotes)

|f|i|r|s|t|\0|e|\0|

Note that the last two elements are a relic of the initialization at declaration time. If the string that is entered is longer than the space available for it in the character array then C++ will just write over whatever space comes next in memory. This can cause some very strange errors when some of your other variables reside in that space!

To read a string with several words in it using **cin** we have to call **cin** once for each word. For example to read in a name in the form of a Christian name followed by a surname we might use code as follows:

```
char christian[12], surname[12];
cout << "Enter name ";
cin >> christian;
cin >> surname;
cout << "The name entered was "
    << christian << " "
```

```
<< surname;
```

The name would just be typed by the user as, for example, **Ian Aitchison** and the output would then be

```
The name entered was Ian Aitchison
```

```
Enter a string: Law is a bottomless pit.
```

```
You entered:    Law
```

Where did the rest of the phrase go?

It turns the insertion operator >> consider a space to be a terminating character.

Thus it will read strings consisting of a single word, but anything typed after a space is thrown away.

To read text containing blanks we use another function, **cin::get()**.

```
#include<iostream.h>
void main()
{
    const int max=80;
    char str[max];
    cout<<"\n Enter a string:";
    cin.get(str,max); // max avoid buffer overflow
    cout<<"\n You entered : "<<str;
}
```

Reading multiple lines

We have solved the problem of reading strings with embedded blanks, but what about strings with multiple lines? It turns out that the **cin::get()** function can take a third argument to help out in this situation.

This argument specifies the character that tells the function to stop reading. The default value of this argument is the **newline("\n")** character, but if you call the function with some other character for this argument, the default will be overridden by the specified character.

In the next example, we call the function with a dollar sign ('\$') as the third argument

```
//reads multiple lines, terminates on '$' character
#include<iostream.h>
void main(){
    const int max=80;
    char str[max];
    cout<<"\n Enter a string:\n";
    cin.get(str, max, '$');    //terminates with $
    cout<<"\n You entered:\n"<<str; }
```

now you can type as many lines of input as you want. The function will continue to accept characters until you enter the terminated character \$ (or until you exceed the size of the array. Remember, you must still press Enter key after typing the '\$' character .

5.7.3 Avoiding buffer over flow

The strings in the program invites the user to type in a string. What happens if the user enters a string that is longer than the array used to hold it? There is no built-in mechanism in C++ to keep a program from inserting array elements outside an array.

However, it is possible to tell the >> operator to limit the number of characters it places in an array.

```
//avoids buffer overflow with cin.width
#include<iostream.h>
#include<iomanip.h>    //to use setw
void main(){
    const int MAX=20;
    char str[MAX];
    cout<<"\n Enter a string: ";
    cin>>setw(MAX)>>str;
    cout<<"\n You entered :"<<str;
}
```

5.7.4 String constants

You can initialize a string to a constant value when you define it. Here's an example'

```
#include<iostream.h>
void main(){
    char str[] = "Welcome to C++ programming language";
    cout<<str;
}
```

if you tried to the string program with strings that contain more than one word , you may have unpleasant surprise. Copying string the hard way

The best way to understand the true nature of strings is to deal with them character by character

```
#include<iostream.h>
#include<string.h>    //for strlen()
void main()
{
    const int max=80;
    char str1[]=" Oh, Captain, my Captain!"
        our fearful trip is done";
    char str2[max];
    for(int i=0; i<strlen(str1);i++)
        str2[i]=str1[i];
    str2[i]='\0';
    cout<<endl;
    cout<<str2;
}
```

5.7.5 Copying string

Ofcourse you don't need to use a **for** loop to copy a string. As you might have guesses, a library function will do it for you. You can copy strings using **strcpy** or **strncpy** function. We assign strings by using the string copy function **strcpy**. The prototype for this function is in **string.h**.

strcpy(destination, source);

strcpy copies characters from the location specified by source to the location specified by destination. It stops copying characters after it copies the terminating null character.

The return value is the value of the destination parameter.

You must make sure that the destination string is large enough to hold all of the characters in the source string (including the terminating null character).

Example:

```
#include <iostream.h>
#include <string.h>
void main(){
    char me[20] = "David";
    cout << me << endl;
    strcpy(me, "YouAreNotMe");
    cout << me << endl ;
    return;
}
```

There is also another function **strncpy**, is like **strcpy**, except that it copies only a specified number of characters.

strncpy(destination, source, int n);

It may not copy the terminating null character.

Example

```
#include <iostream.h>
#include <string.h>
void main() {
    char str1[] = "String test";
    char str2[] = "Hello";
    char one[10];
    strncpy(one, str1, 9);
    one[9] = '\0';
    cout << one << endl;
    strncpy(one, str2, 2);
    cout << one << endl;
    strcpy(one, str2);
    cout << one << endl;
}
```

5.7.6 Concatenating strings

In C++ the + operator cannot normally be used to concatenate string, as it can in some languages such as BASIC; that is you can't say

str3 = str1 + str2;

You can use strcat() or strncat

The function strcat concatenates (appends) one string to the end of another string.

strcat(destination, source);

- The first character of the source string is copied to the location of the terminating null character of the destination string.
- The destination string must have enough space to hold both strings and a terminating null character.

Example:

```
#include <iostream.h>
#include <string.h>
void main() {
    char str1[30];
    strcpy(str1, "abc");
    cout << str1 << endl;
    strcat(str1, "def");
    cout << str1 << endl;
    char str2[] = "xyz";
    strcat(str1, str2);
    cout << str1 << endl;
    str1[4] = '\0';
    cout << str1 << endl;
}
```

The function strncat is like strcat except that it copies only a specified number of characters.

strncat(destination, source, int n);

It may not copy the terminating null character.

Example:

```
#include <iostream.h>
#include <string.h>
void main() {
    char str1[30];
    strcpy(str1, "abc");
    cout << str1 << endl;
    strncat(str1, "def", 2);
    str1[5] = '\0';
    cout << str1 << endl;
    char str2[] = "xyz";
    strcat(str1, str2);
    cout << str1 << endl;
    str1[4] = '\0';
    cout << str1 << endl;
}
```

5.7.7 Comparing strings

Strings can be compared using `strcmp` or `strncmp` functions

The function `strcmp` compares two strings.

`strcmp(str1, str2);`

`strcmp` returns: `< 0` if str1 is less than str2
 `= 0` if str1 is equal to str2
 `> 0` if str1 is greater than str2

Example:

```
#include <iostream.h>
#include <string.h>
void main() {
    cout << strcmp("abc", "def") << endl;
    cout << strcmp("def", "abc") << endl;
    cout << strcmp("abc", "abc") << endl;
    cout << strcmp("abc", "abcdef") << endl;
    cout << strcmp("abc", "ABC") << endl;
}
```

The function **`strncmp`** is like **`strcmp`** except that it compares only a specified number of characters.

`strncmp(str1, str2, int n);`

`strncmp` does not compare characters after a terminating null character has been found in one of the strings.

Example:

```
#include <iostream.h>
#include <string.h>
void main()
{
    cout << strncmp("abc", "def", 2) << endl;
    cout << strncmp("abc", "abcdef", 3) << endl;
    cout << strncmp("abc", "abcdef", 2) << endl;
    cout << strncmp("abc", "abcdef", 5) << endl;
    cout << strncmp("abc", "abcdef", 20) << endl;
}
```

5.8 Dynamic memory:

Until now, in our programs, we have only had as much memory as we have requested in declarations of variables, arrays and other objects that we included, having the size of all of them to be fixed before the execution of the program. But, What if we need a variable amount of memory that can only be determined during the program execution (runtime)? For example, in case that we need a user input to determine the necessary amount of space. The answer is *dynamic memory*, for which C++ integrates the operators *new* and *delete*.

- Pointers are useful for creating **dynamic** objects during program execution.

Unlike normal (global and local) objects which are allocated storage on the runtime stack, a dynamic object is allocated memory from a different storage area called the **heap**. Dynamic objects do not obey the normal scope rules. Their scope is explicitly controlled by the programmer.

a) *The New Operator*

In C++ *new* operator can create space dynamically i.e at run time, and similarly *delete* operator is also available which releases the memory taken by a variable and return memory to the operating system.

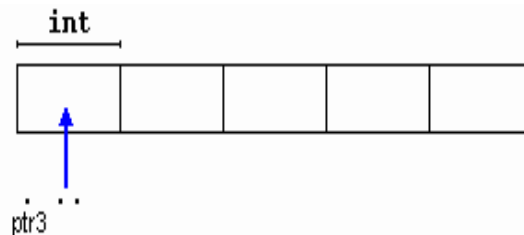
When the space is created for a variable at compile time this approach is called static. If space is created at run time for a variable, this approach is called dynamic. See the following two lines:

```
int a[10]; //creation of static array
int *a;
a = new int[10]; //creation of dynamic array
```

Lets have another example:

```
int * ptr3;
ptr3 = new int [5];
```

In this case, the operating system has assigned space for 5 elements of type **int** in the heap and it has returned a pointer to its beginning that has been assigned to **ptr3**. Therefore, now, **ptr3** points to a valid block of memory with space for 5 **int** elements.



You could ask what is the difference between declaring a normal array and assigning memory to a pointer as we have just done. The most important one is that the size of an array must be a constant value, which limits its size to what we decide at the moment of designing the program before its execution, whereas the dynamic memory allocation allows assigning memory during the execution of the program using any variable, constant or combination of both as size.

The dynamic memory is generally managed by the operating system, and in the multi-task interfaces can be shared between several applications, so there is a possibility that the memory exhausts. If this happens and the operating system cannot assign the memory that we request with the operator **new**, a null pointer will be returned. For that reason it is recommendable to always verify if after a call to instruction **new** the returned pointer is null:

```
int * ptr3;
ptr3 = new int [5];
```

```

if (ptr3 == NULL) {
    // error assigning memory. Take measures.
};

```

if ptr3 is **NULL**, it means that there is no enough memory location in the heap to be given for ptr3.

b) Operator delete

Since the necessity of dynamic memory is usually limited to concrete moments within a program, once this one is no longer needed it shall be freed so that it become available for future requests of dynamic memory. For this exists the operator **delete** , whose form is:

delete pointer ;

or

delete [] pointer ;

- The first expression should be used to delete memory allocated for a single element, and the second one for memory allocated for multiple elements (arrays).
- In most compilers both expressions are equivalent and can be used without distinction, although indeed they are two different operators and so must be considered for operator overloading.
- In the following simple example, a program that memorizes numbers, does not have a limited amount of numbers that can be introduced, thanks to the concept and power of pointer that we request to the system as much space as it is necessary to store all the numbers that the user wishes to introduce.

```

#include <iostream.h>
#include <stdlib.h>
#include <conio.h>
int main ()
{
    char input [100];
    int i,n;
    long * num;// total = 0;
    cout << "How many numbers do you want to type in? ";
    cin.getline (input,100);
    i=atoi (input);
    num= new long[i];
    if (num == NULL)
    {
        cout<<"\nno enough memory!";
        getch();
        exit (1);
    }
    for (n=0; n<i; n++)
    {

```



```
        cout << "Enter number: ";
        cin.getline (input,100);
        num[n]=atol (input);
    }
    cout << "You have entered: ";
    for (n=0; n<i; n++)
        cout << num[n] << ", ";
    delete[] num;
    getch();
    return 0;
}
```

NULL is a constant value defined in C++ libraries specially designed to indicate null pointers. In case that this constant is not defined you can do it yourself by defining it to 0:

Chapter 6: Structures

6.1 Specifying simple structure

A structure is a collection of one or more variable types grouped together that can be referred using a single name (group name) and a member name.

You can refer to a structure as a single variable, and you also can initialize, read and change the parts of a structure (the individual variables that make it up).

Each element (called a member) in a structure can be of different data type.

The General Syntax of structures is:

```
Struct [structure tag]
{
    Member definition;
    Member definition;
    ...
    Member definition;
}[one or more structure variables];
```

Let us see an example

E.g.:

```
struct Inventory
{
    char description[15];
    char part_no[6];
    int quantity;
    float cost;
}; // all structures end with semicolon
```

Another example might be:

```
struct Student
{
    char
    ID[8];
    char
    FName[15];
    char LName[15];
    char Sex;
    int age;
    float CGPA;
};
```

The above “Student” structure is aimed to store student record with all the relevant details. After the definition of the structure, one can declare a structure variable using the structure tag. If we need two variables to have the above structure property then the declaration would be:

```
struct Inventory inv1,inv2; //or
```

```
struct Student Stud1,stud2,Stud3;
```

Structure tag is not a variable name. Unlike array names, which reference the array as variables, a structure tag is simply a label for the structure's format.

The structure tag `Inventory` informs C++ that the tag called `Inventory` looks like two character arrays followed by one integer and one float variables. A structure tag is actually a newly defined data type that you, the programmer, defined.

6.2 Initializing and accessing structure variable

To access or refer to the members/variables of a structure we need to use the **dot operator** (`.`)

The General syntax to access members of a structure variable would be:

VarName.Member

Where *VarName* is the variable name of the structure variable And *Member* is variable name of the members of the structure

Eg:

For the above student structure:

```
struct Student Stud; //declaring Stud to have the property of the Student structure
strcpy(Stud.FName,"Abebe"); //assigned Abebe as First Name
Stud.CGPA=3.21; //assignes 3.21 as CGPA value of Abebe
cout<<Stud.FName; //display the name
cout<<Stud.CGPA; // display the CGPA of Abebe
```

Initializing Structure Data

You can initialize members when you declare a structure, or you can initialize a structure in the body of the program. Here is a complete program.

```
struct cd_collection
{
    char title[25];
    char artist[20];
    int num_songs;
    float price;
    char date_purchased[9];
} cd1 = {"Red Moon Men","Sams and the Sneeds", 12, 11.95,"08/13/93"};
cout<<"\nhere is the info about cd1"<<endl;
cout<<cd1.title<<endl;
cout<<cd1.artist<<endl;
cout<<cd1.num_songs<<endl;
cout<<cd1.price<<endl;
cout<<cd1.date_purchased<<endl;
```

A better approach to initialize structures is to use the dot operator(`.`). the dot operator is one way to initialize individual members of a structure variable in the body of your program. The syntax of the dot operator is :

structureVariableName.memberName

here is an example:

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
void main()
{
    clrscr();
    struct cd_collection{
        char title[25];
        char artist[20];
        int num_songs;
        float price;
        char date_purchased[9];
    }cd1;
    //initialize members here
    strcpy(cd1.title,"Red Moon Men");
    strcpy(cd1.artist,"Sams");
    cd1.num_songs= 12;
    cd1.price = 11.95f;
    strcpy(cd1.date_purchased,"22/12/02");
    //print the data
    cout<<"\nHere is the info"<<endl;
    cout<<"Title : "<<cd1.title<<endl;
    cout<<"Artist : "<<cd1.artist<<endl;
    cout<<"Songs :
    "<<cd1.num_songs<<endl; cout<<"Price :
    "<<cd1.price<<endl;
    cout<<"Date purchased : "<<cd1.date_purchased;
    getch();
}
```

6.3 Arrays of Structures and accessing array of structures

Arrays of structures are good for storing a complete employee file, inventory file, or any other set of data that fits in the structure format.

Consider the following structure declaration:

```
struct Company
{
    int employees;
    int registers;
    double sales;
    }store[1000];
```

In one quick declaration, this code creates 1,000 *store* structures with the definition of the **Company** structure, each one containing three members.

NB. Be sure that your computer does not run out of memory when you create a large number of structures. Arrays of structures quickly consume valuable information.

You can also define the array of structures after the declaration of the structure.

```
struct Company
{
    int employees;
    int registers;
    double sales;
}; // no structure variables defined yet

#include<iostream.h>
...
void main()
{
    struct Company store[1000]; //the variable store is array of the structure Company
    ...
}
```

Accessing or referencing the array structure

The **dot operator** (.) works the same way for structure array element as it does for regular variables. If the number of employees for the fifth store (store[4]) increased by three, you could update the structure variable like this:

```
store[4].employees += 3;
```

Unlike in the case of arrays, where the whole content of an array could not be copied to another one using a simple statement, in structures, you can assign complete structures to one another by using array notation.

To assign all the members of the 20th store to the 45th store, you would do this:

```
store[44] = store[19]; //copies all members from 20th store to 45th
```

Here is a complete C++ code that shows you how to use array of structures, and how to pass and return structures to functions.

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
#include<iomanip.h>
struct inventory
{
    long storage; int
    accesstime; char
    vendorcode;
    float cost;
    float price;
};
void disp_menu(void);
struct inventory enter_data();
void see_data(inventory disk[125],int num_items);
```

```
void main()
{ clrscr();
  inventory disk[125];
  int ans;
  int num_items = 0; //total number of items in the inventory

  do{
    do{
      disp_menu();
      cin>>ans;
    }while(ans<1 || ans>3);

    switch(ans)
    { case 1:
      disk[num_items] = enter_data();
      num_items++;
      break;
      case 2:
      see_data(disk,num_items);
      break;
      default :
      break;
    }
    }while(ans != 3);
  return;
} //end main

void disp_menu()
{
  cout<<"\n\n*** Disk Drive Inventory System ***\n\n";
  cout<<"Do you want to : \n\n";
  cout<<"t1. Enter new item in inventory\n\n";
  cout<<"t2. See inventory data\n\n"; cout<<"t3.
  Exit the program\n\n"; cout<<"What is your
  choice ? ";
  return;
}
```

```
inventory enter_data()
{
    inventory disk_item; //local variable to fill with input
    cout<<"\n\nWhat is the next drive's storage in bytes? ";
    cin>>disk_item.storage;
    cout<<"\nWhat is the drive's access time in ms ? ";
    cin>>disk_item.accesstime;
    cout<<"What is the drive's vendor code (A, B, C, or D)? ";
    disk_item.vendorcode = getchar();
    cout<<"\nWhat is the drive's cost? ";
    cin>>disk_item.cost;
    cout<<"\nWhat is the drive's price? ";
    cin>>disk_item.price;
    return (disk_item);
}

void see_data(inventory disk[125], int num_items)
{
    int ctr;
    cout<<"\n\nHere is the inventory listing:\n\n";
    for(ctr=0;ctr<num_items;ctr++)
    {
        cout<<"Storage: "<<disk[ctr].storage<<"\n";
        cout<<"Access time: "<<disk[ctr].accesstime<<endl;
        cout<<"Vendor code: "<<disk[ctr].vendorcode<<"\n";
        cout<<"Cost : $ "<<disk[ctr].cost<<"\n"; cout<<"Price: $ "
            <<disk[ctr].price<<endl;
    }
    return;
}
```

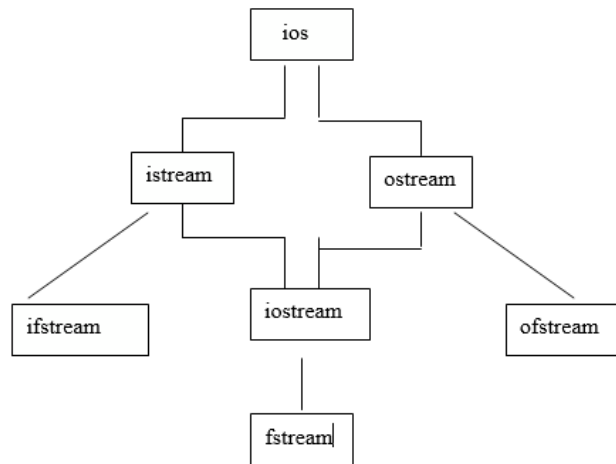
Chapter Seven: File and File Management

7.1 Introduction

- The data created by the user and assigned to variables with an assignment statement is sufficient for some applications. With large volume of data most real-world applications use a better way of storing that data. For this, disk files offer the solution.
- When working with disk files, C++ does not have to access much RAM because C++ reads data from your disk drive and processes the data only parts at a time.

7.2 Stream

- Stream is a general name given to *flow of data*. In C++, there are different types of streams. Each stream is associated with a particular class, which contains member function and definition for dealing with file. Lets have a look at the figure:



- According to the above hierarchy, the class `iostream` is derived from the two classes' `istream` and `ostream` and both `istream` and `ostream` are derived from `ios`. Similarly the class `fstream` is derived from `iostream`.

Generally two main header files are used `iostream.h` and `fstream.h`. The classes used for input and output to the video display and key board are declared in the header file `iostream.h` and the classes used for disk file input output are declared in `fstream.h`.

- Note that when we include the header file `fstream.h` in our program then there is no need to include `iostream.h` header file. Because all the classes which are in `fstream.h` they are derived from classes which are in `iostream.h` therefore, we can use all the functions of `iostream` class.

7.3 Operation with File

- First we will see how files are opened and closed. A file can be defined by following class `ifstream`, `ofstream`, `fstream`, all these are defined in `fstream.h` header file.
 - if a file object is declared by `ifstream` class, then that object can be used for *reading* from a file.

- if a file object is declared by *ofstream* class, then that object can be used for *writing* onto a file.
- If a file object is declared by *fstream* class then, that object can be used for both *reading from* and *writing to* a file

7.4 Types of Disk File Access

- Your program can access files either in *sequential* manner or *random* manner. The access mode of a file determines how one can read, write, change, add and delete data from a file.
- A sequential file has to be accessed in the same order as the file was written. This is analogous to cassette tapes: you play music in the same order as it was recorded.
- Unlike the sequential files, you can have random-access to files in any order you want. Think of data in a random-access file as being similar to songs on compact disc (CD): you can go directly to any song you want to play without having to play or fast-forward through the other songs.

7.4.1 Sequential File Concepts

- You can perform three operations on sequential disk files. You can *create* disk files, *add* to disk files, and *read* from disk files.

7.4.1.1: Opening and Closing Sequential Files

- When you open a disk file, you only have to inform C++, the file name and what you want to do with it. C++ and your operating system work together to make sure that the disk is ready, and they create an entry in your file directory for the filename (if you are creating a file). When you

- close a file, C++ writes any remaining data to the file, releases the file from the program, and updates the file directory to reflect the file's new size.
- You can use either of the two methods to open a file in C++:
 - using a **Constructor** or
 - using the **open function**
 - The following C++ statement will create an object with the name **fout** of **ofstream** class and this object will be associated with file name "hello.txt".
ofstream fout ("hello.txt");
 - This statement uses the constructor method.
 - The following C++ statement will create an object with the name **fout** of **ofstream** class and this object will be associated with file name "hello.txt".
ofstream fout;
fout.open("hello.txt");
 - If you open a file for writing (out access mode), C++ creates the file. If a file by that name already exists, C++ *overwrite* the old file with no warning. You must be careful when opening files not to overwrite existing data that you want.
 - If an error occurs during opening of a file, C++ does not create a valid file pointer (file object). Instead, C++ creates a file pointer (object) equal to zero. For example if you open a file for output, but use an invalid disk name, C++ can't open the file and therefore makes the file object equal to zero.
 - You can also determine the file access mode when creating a file in C++. If you want to use the open function to open a file then the syntax is:
fileobject.open(filename,accessmode);
 - File name is a string containing a valid file name for your computer.
 - Accessmode is the sought operation to be taken on the file and must be one of the values in the following table.

Mode	Description
app	Opens file for appending
ate	Seeks to the end of file while opening the file
in	Opens the file for reading
out	Opens the file for writing
binary	Opens the file in binary mode

- You should always check for the successful opening of a file before starting file manipulation on it. You use the fail() function to do the task:

- Lets have an example here: ifstream indata;

```
indata.open("c:\\myfile.txt",ios::in);
if(indata.fail())
{
    //error description here
}
```
- In this case, the open operation will fail (i.e the fail function will return true), if there is no file named myfile.txt in the directory C:\
- After you are done with your file manipulations, you should use the close function to release any resources that were consumed by the file operation. Here is an example

```
indata.close();
```
- The above close() statement will terminate the relationship between the ifstream object indata and the file name "c:\\myfile.txt", hence releasing any resource needed by the system.

7.4.1.2: Writing to a sequential File

- The most common file I/O functions are
 - get() and put()
- You can also use the output redirection operator (<<) to write to a file.
- The following program creates a file called names.txt in C:\ and saves the name of five persons in it:

```
#include<fstream.h>
#include<stdlib.h>
ofstream fp;
void main()
{
    fp.open("c:\\names.txt",ios::out);
    if(fp.fail())
    {
        cerr<< "\nError opening file";
        getch();
        exit(1);
    }
    fp<< "Abebe Alemu"<<endl;
    fp<< "Lemelem Berhanu"<<endl;
    fp<< "Tesfaye Mulugeta"<<endl;
    fp<< "Mahlet Kebede"<<endl; fp<<
    "Assefa Bogale"<<endl; fp.close();
} //end main
```

Writing characters to sequential files:

- A character can be written onto a file using the *put()* function. See the following code:

```
#include<fstream.h>
#include<stdlib.h>// for exit() function
...
void main()
{
    char c;
    ofstream outfile;
    outfile.open("c:\\test.txt",ios::out);

    if(outfile.fail())
    {
        cerr<< "\nError opening test.txt";
        getch();
        exit(1);
    }

    for(int i=1;i<=15;i++)
    {
        cout<< "\nEnter a character : ";
        cin>>c;
        outfile.put(c);
    }
    output.close();
} //end main
```

- The above program reads 15 characters and stores in file test.txt.
- You can easily add data to an existing file, or create new files, by opening the file in *append access mode*.
- Files you open for append access mode (using *ios::app*) do *not have to exist*. If the file exists, C++ appends data to the end of the file (as is done when you open a file for write access).

- The following program adds three more names to the *names.txt* file created in the earlier program.

```
#include<fstream.h>
#include<stdlib.h>
...
void main()
{
    ofstream outdata;
    outdata.open("c:\\names.txt",ios::app);
    if(outdata.fail())
    {
        cerr<< "\nError opening names.txt";
        getch();
        exit(1);
    }
    outdata<< "Berhanu Teka"<<endl;
    outdata<< "Zelalem Assefa"<<endl;
    outdata<< "Dagim Sheferaw"<<endl;
    outdata.close();
} //end main
```

- If the file *names.txt* *does not exist*, C++ creates it and stores the three names to the file.
- Basically, you have to change only the `open()` function's access mode to turn a file-creation program into a file-appending program.

7.4.1.3: Reading from a File

- Files you open for read access (using `ios::in`) *must exist already*, or C++ gives you an error message. You can't read a file that does not exist. `Open()` returns zero if the file does not exist when you open it for read access.
- Another event happens when you read files. Eventually, you read all the data. Subsequently reading produces error because there is *no more* data to read. C++ provides a solution to the end-of-file occurrence.
- If you attempt to read a file that you have completely read the data from, C++ returns the value zero. To find the end-of-file condition, be sure to check for zero when reading information from files.

- The following code asks the user for a file name and displays the content of the file to the screen.

```
#include<fstream.h>
#include<conio.h>
#include<stdlib.h>
void main()
{
    clrscr();
    char name[20],filename[15];
    ifstream indata;
    cout<<"\nEnter the file name : ";
    cin.getline(filename,15);
    indata.open(filename,ios::in);
    if(indata.fail())
    {
        cerr<<"\nError opening file : "<<filename;
        getch();
        exit(1);
    }
    while(!indata.eof())// checks for the end-of-file
    {
        indata>>name;
        cout<<name<<endl;
    }
    indata.close();
    getch();
}
```

Reading characters from a sequential file

- You can read a characters from a file using *get()* function. The following program asks for a file name and displays *each character* of the file to the screen. NB. A space among characters is considered as a character and hence, the exact replica of the file will be shown in the screen.

```
#include<fstream.h>
#include<conio.h>
#include<stdlib.h>
void main()
{
    char c,filename[15];
    ifstream indata;
    cout<<"\nEnter the file name : ";
```

```

cin.getline(filename,15);
indata.open(filename,ios::in); if(indata.fail())//
check id open succeeded
{
    cerr<<"\nError opening file : "<<filename;
    getch();
    exit(1);
}
while(!indata.eof())// check for eof
{
    indata.get(c);
    cout<<c;
}
indata.close();
getch();
}

```

7.4.1.4: File Pointer and their Manipulators

- Each file has two pointers one is called *input pointer* and second is *output pointer*. The input pointer is called *get pointer* and the output pointer is called *put pointer*.
 - When input and output operation take places, the appropriate pointer is automatically set according to the access mode.
 - For example when we open a file in reading mode, file pointer is automatically set to the start of the file.
 - When we open a file in append mode, the file pointer is automatically set to the end of file.
 - In C++ there are some manipulators by which we can control the movement of the pointer. The available manipulators are:
 1. seekg()
 2. seekp()
 3. tellg()
 4. tellp()
1. seekg(): this moves get pointer i.e input pointer to a specified location.
For eg. infile.seekg(5); move the file pointer to the byte number 5 from starting point.
 2. seekp(): this move put pointer (output pointer) to a specified location for example: outfile.seekp(5);
 3. tellg(): this gives the current position of get pointer (input pointer)
 4. tellp(): this gives the current position of put pointer (output pointer)
eg. ofstream fileout;

```
fileout.open("c:\\test.txt",ios::app);
int length = fileout.tellp();
```

- By the above statement in length, the total number bytes of the file are assigned to the integer variable length. Because the file is opened in append mode that means, the file pointer is the last part of the file.
- Now lets see the seekg() function in action

```
#include<fstream.h>
#include<conio.h>
#include<stdlib.h>
void main()
{
    clrscr();
    fstream fileobj;
    char ch; //holds A through Z
    //open the file in both output and input mode
    fileobj.open("c:\\alph.txt",ios::out|ios::in);
    if(fileobj.fail())
    {
        cerr<<"\nError opening alph.txt";
        getch();
        exit(1);
    }
    //now write the characters to the file
    for(ch = 'A'; ch <= 'Z'; ch++)
    {
        fileobj<<ch;
    }
    fileobj.seekg(8L,ios::beg);//skips eight letters, points to I
    fileobj>>ch;
    cout<<"\nThe 8th character is : "<<ch;
    fileobj.seekg(16L,ios::beg);//skips 16 letters, points to Q
    fileobj>>ch;
    cout<<"\nThe 16th letter is : "<<ch;
    fileobj.close();
    getch();
}
```

- To point to the end of a data file, you can use the seekg() function to position the file pointer at the last byte. This statement positions the file pointer to the last byte in the file. ***Fileobj.seekg(0L,ios::end);***

- This seekg() function literally reads “move the file pointer 0 bytes from the end of the file.” The file pointer now points to the end-of-file marker, but you can seekg() backwards to find other data in the file.
- The following program is supposed to read “c:\alph.txt” file backwards, printing each character as it skips back in the file.
- Be sure that the seekg() in the program seeks two bytes backwards from the *current* position, not from the beginning or the end as the previous programs. The for loop towards the end of the program needs to perform a “skip-two-bytes-back”, read-one-byte-forward” method to skip through the file backwards.

```
#include<fstream.h>
#include<conio.h>
#include<stdlib.h>

void main()
{
    clrscr();
    ifstream indata;
    int ctr=0;
    char inchar;

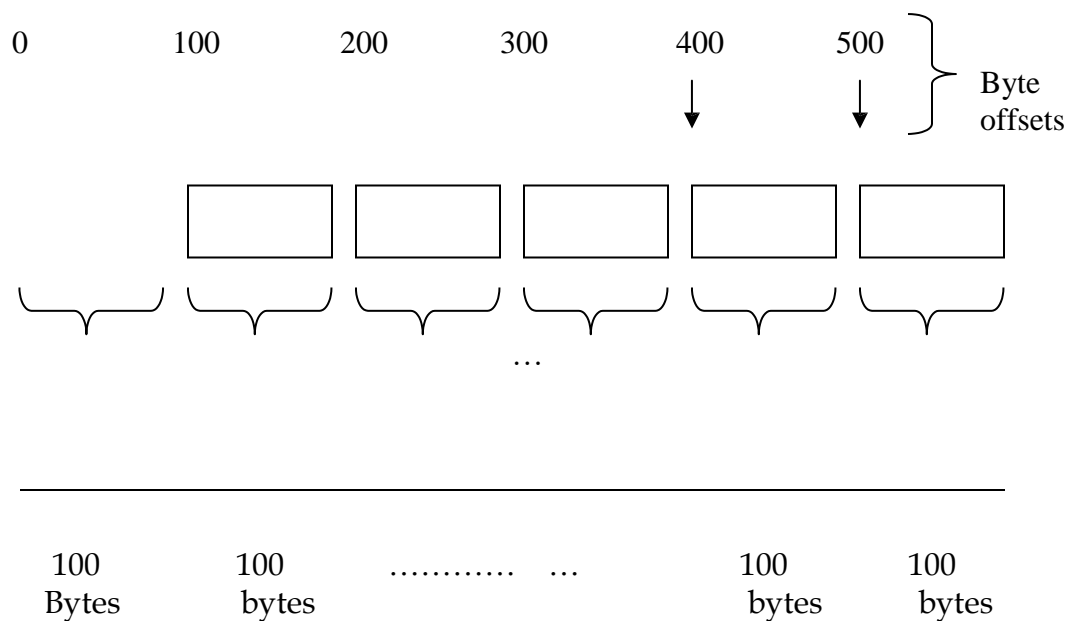
    indata.open("c:\\alph.txt",ios::in);
    if(indata.fail())
    {
        cerr<<"\nError opening alph.txt";
        getch();
        exit(1);
    }
    indata.seekg(-1L,ios::end);//points to the last byte in the file
    for(ctr=0;ctr<26;ctr++)
    {
        indata>>inchar;
        indata.seekg(-2L,ios::cur);
        cout<<inchar;
    }
    indata.close();
    getch();
}
```

Text Files And Binary Files (Comparison)

- The default access mode for file access is *text mode*. A text file is an ASCII file, compatible with most other programming languages and applications. Programs that read ASCII files can read data you create as C++ text files.
- If you specify binary access, C++ creates or reads the file in binary format. Binary data files are “*squeezed*”- that is, they take less space than text files. The disadvantage of using binary files is that other programs can’t always read the data files. Only C++ programs written to access binary files can read and write them. The advantage of binary files is that you save disk space because your data files are more compact.
- The binary format is a *system-specific* file format. In other words, not all computers can read a binary file created on another computer.

7.4.2 Random Access File Concepts

- Random access enables you to read or write any data in your disk file with out having to read and write every piece of data that precedes it.
- Generally you read and write file records. A record to a file is analogues to a C++ structure. A record is a collection of one or more data values (called fields) that you read and write to disk. Generally you store data in the structures and write structures to disk.
- When you read a record from disk, you generally read that record into a structure variable and process it with your program.
- Most random access files are fixed-length records. Each record (a row in the file) takes the same amount of disk space.
- With fixed length records, your computer can better calculate where on the disk the desired record is located.



7.4.2.1 Opening Random-Access Files

- There is really no difference between sequential files and random files in C++. The difference between the files is not physical, but lies in the method that you use to access them and update them.
- The ostream member function write outputs a fixed number of bytes, beginning at a specific location in memory, to the specified stream.
- When the stream is associated with a file, function write writes the data at the location in specified by the “put” file position pointer.
- The istream member function read inputs a fixed number of bytes from the specified stream into an area in memory beginning at the specified address.
- When the stream is associated with a file, function read inputs bytes at the location in the file specified by the “get” file poison pointer.
- Syntax of write: *fileobject.write((char*) & NameOfObject, sizeof(name of object))*
- Function write expects data type const char* as its first argument. The second argument of write is an integer of type size specifying the number of bytes to be written.

Writing randomly to a random access file

- Here is an example that shows how to write a record to a random access file.

```
#include<fstream.h>
#include<conio.h>
#include<stdlib.h>
struct stud_info{
    int id;
    char name[20];
    char fname[20];
    float CGPA;
}student;

void main()
{
    clrscr();
    char filename[15]; ofstream
    outdata; cout<<"\nenter file name :
    "; cin>>filename;
    outdata.open(filename,ios::out);
    if(outdata.fail())
    {
```

```

        cerr<<"\nError opening "<<filename;
        getch();
        exit(1);
    }
    //stud_info student;
    //accept data here
    cout<<"\nEnter student id : ";
    cin>>student.id;
    cout<<"\nEnter student name : ";
    cin>>student.name;
    cout<<"\nEnter student father name : ";
    cin>>student.fname; cout<<"\nEnter
    student CGPA : ";
    cin>>student.CGPA;

    //now write to the file outdata.seekp(((student.id)-1)
    * sizeof(student)); outdata.write((char*) &student,
    sizeof(student)); outdata.close();
    cout<<"\nData has been saved";
    getch();
}

```

- The above code uses the combination of ostream function seekp and write to store data at exact locations in the file.
- Function seekp sets the put file-position pointer to a specific position in the file, then the write outputs the data.
- 1 is subtracted from the student id when calculating the byte location of the record. Thus, for record 1, the file position pointer is set to the byte 0 of the file.
- The istream function read inputs a specified number of bytes from the current position in the specified stream into an object.
- The syntax of read : read((char*)&name of object, sizeof(name of object));
- Function read requires a first argument of type char *. The second argument of write is an integer of type size specifying the number of bytes to be read.

- Here is a code that shows how to read a random access record from a file.

```
#include<fstream.h>
#include<conio.h>
#include<stdlib.h>
struct stud_info{
    int studid;
    char name[20];
    char fname[20];
    float CGPA;
};
void main()
{
    clrscr();
    ifstream indata; char filename[15];
    cout<<"\nEnter the file name : ";
    cin>>filename;
    indata.open(filename,ios::in);
    if(indata.fail())
    {
        cerr<<"\nError opening "<<filename;
        getch();
        exit(1);
    }
    stud_info student;
    cout<<"\nEnter the id no of the student : ";
    int sid;
    cin>>sid;
    indata.seekg((sid-1) * sizeof(student));
    indata.read((char*) &student, sizeof(student));
    cout<<"\nhere is the information";
    cout<<"\nstudent id : "<<student.studid;
    cout<<"\nstudent name : "<<student.name;
    cout<<"\nstudent fname : "<<student.fname;
    cout<<"\nstudent CGPA : "<<student.CGPA;
    indata.close();
    getch();
}
```

7.5 Command Line Argument

- Command line argument means facility by which you can supply arguments to the main() function. These arguments are supplied to the program when the main function is called from the command line. Eg. c:\> file-name arg1 arg2
- Where file-name is the name of file(the program) and arg1, arg2 are arguments passed to the program.
- If we want to pass arguments to the main function, then main function should be written as follow.
Return type main(int argc, char * argv[])
- The first argument argc represents the number of arguments in a command line. The second argument argv is an array of character type pointers that points to the command line arguments. Argc is known as argument counter and argv is called argument vector.
- C:\> student A B. the value of argc is 3 (student, A & B) and the argv would be an array of three pointers to string as follows:
Argv[0] – points to student Argv[1] – points to A Argv[2] – points to B
- note that argv[0] always represents the command name that invokes the program.

- Here is a sample command line argument code

```
#include<fstream.h>
#include<conio.h>
#include<stdlib.h>
void main(int argc, char *argv[])
{
clrscr(); char ch; if(argc < 3)
{
cerr<<"\ntoo few parameters";
getch();
exit(1);
}
else if(argc > 3)
{
cerr<<"\ntoo many parametes";
getch();
exit(1);
}
else//number of parameters okay
{
ofstream outdata;
ifstream indata;
outdata.open(argv[2],ios::out);
if(outdata.fail())
{
cerr<<"\nlow disk space to create file : "<<argv[2];
getch();
exit(1);
}
```

```
    }
    indata.open(argv[1],ios::in);
    if(indata.fail())
    {
        cerr<<"\nfile : "<<argv[1]<<" does not exist";
        getch();
        exit(1);
    }
    //now start copying the file
    while(!indata.eof())
    {
        indata.get(ch);
        outdata.put(ch);
    }
    indata.close(); outdata.close();
    cout<<"\nfinished copying";
}
}
```

References

1. Ravichandran; "Problem Solving with C++", Tata Mc. Grew Hill Company
2. Thinking in C++, Volume 2: Practical Programming, Bruce Eckel, President, MindView, Inc., Chuck Allison, Utah Valley State Colleg
3. E.Balagurusamy, "Programming with C++", Tata Mc. Grew Hill Company