INTERNATIONAL COLLEGE of YAYASAN MELAKA

Open Distance Learning

TCIT1023 Programming Fundamentals

ROSNIZA BINTI A. RAHIM

Module Writer: Rosniza Binti A. Rahim

Developed by: International College of Yayasan Melaka

TABLE OF CONTENTS

UNDERSTANDING COURSE GUIDE

Refer and understand this *Course Guide* carefully from the beginning to the end. It describes the course a n d how you use the course material. It suggests the learning time to complete the course successfully. Referring the *Course Guide* will help you to clarify important contents that you might miss or overlook.

ABOUT THE COURSE

TCIT 1023 Programming Fundamentals is subject for Diploma Information Technology that offered by School of Engineering and Computing Technologyin ICYM. This course is worth 3 credit hours and should be covered to14 weeks

You should be acquainted with learning independently and being able to optimize the learning modes and environment available to you. Make sure refer right course material and understand the course requirements as well as how the course is conducted.

LEARNING TIME SCHEDULE

It is a standard ICYM practice that learner accumulate 40 study hours for every credit hour. As for this three-credit hour course, you are expected to spend 120 study hours. Table 1 gives an estimation of how the 120 study hours could be accumulated.

Table 1: Estimation of Student Learning Time

Distribution of Student Learning Time by Chapter	CLO	Teaching and Learning Activities					
		Face to Face				Non-Face to Face (Independent Learning)	Total
		L	T	P	O		
Chapter 1	1	2	1			3	
Chapter 2	2	4	2			6	
Chapter 3	3	4	2	4		2	
Chapter 4	2	2	1	2		1	
Chapter 5	3	4	2	4		2	
Chapter 6	3	4	2	4		4	
Chapter 7	2	4	2	4		2	
Chapter 8	2	4	2	4		2	
						Sub-Total SLT	86
Continuous Assessment		%	Face to Face			Non-Face to Face (Independent Learning)	
			Physical	Online			
1	Test	20		3		5	
2	Lab Skill	10		4		3	
3	Project Assignment	20		1		5	
4	Participant			3		2	
						Sub-Total SLT	26
Final Assessment		%	Face to Face			Non-Face to Face (Independent Learning)	
			Physical	Online			
1	Final Examination	50	3			5	
						Sub-Total SLT	8
						GRAND-Total SLT	**120**

COURSE LEARNING OUTCOME

By the end of this course, you should be able to:

1. List and categorized programming languages (PLO1, C1)
2. Discuss the usage of C++ syntax for sequence, selection and repetition statement. (PLO2, C2)
3. Solve an application using C++ programming language to solve given problems within a given time. (PLO3, C3A)

COURSE SYNOPSIS

This course is divided into 8 topics. The synopsis for each topic can be listed as follows:

Topic 1 explain the concepts of program and programming and how the programming can be done in certain phase. The discussion about the programming Languages and program Development Tools 4GL.

Topic 2 explain the algorithms where student should understand how to represent the algorithms in correct ways. They will exposed to the right technique in creating algorithms for sequence, selection and repetition structures of programming. There are two types of problem solving technique, either using the flow chart or the pseudocode.

Topic 3 the Dev C++ software that will be used by he students for the entire module, is shown through sample coding. The student should understand the development environment for C++ language and try out their first program. The concepts of data types and variable in programming also introduced in this topic.

Topic 4 there are many types of operator such as arithmetic, relational and logical operators. This topic provides an introduction to these operators and also its usage.

Topic 5 introduced to the selection control structure which uses if and switch statement. The example are given, so student can write program using this control structures. The menu and usage of the break statement is also introduced. The difference between selection structures isgiven to students so that students can use it correctly.

Topic 6 This topic gives three types of repetition structure that is used in C++ program, which are while, do-while and for statements. Examples are given to students so that students can write the repetition structure with the correct syntax

Topic 7 This topic will be introduced with the Functions. Functions are sub programs in a C++ Program. To write a C++ program which is long, functions are necessary so that every sub-program can be written separately and tested separately. After all the sub programs are written, it can be combined to make a complete program.

Topic 8 C++ provides the array, which stores fixed-size sequential collection of elements of the same type. An arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

LEARNING GUIDANCE

The learning guidance is important to understand before you go through this module. Understanding the learning guidance will help you to organize your study of this course in a more objective and effective way. Generally, learning guidance for each topic is as follows:

Learning Outcomes: This part is to measurable, observable, and specific statement that clearly indicates what you should know and be able to do because of learning in each chapter. By go through each topic, you can continuously gauge your understanding of the topic.

Self-Learning Material: To aid you in your subsequent learning and to report on what you have learned. The activities are in-text questions (ITO) and self-assessment questions (SAQ), assignment on each chapter of the material to monitor and develop your own learning.

Activity: Question and activity within module can be constructed to put back the dialogue between student and module in learning activity. With the given question or task, you are encouraged to read the description or explanation within a module, so you can answer the question or solve the problem proposed.
You are encouraged to read since you realize that without reading the description or explanation, you will not be able to answer the question or the assignment. Text question is applied to you to pay attention to a certain problem rather than to assess the learning progress.
Self-assessment question is such a task that requires written answer

form you. If you complete the task, you are asking to check your answer with the answer key provided in the module.
Self -assessment is be developed in various form of test questions, there are easy question, fill in the blank, multiple choices, true-false and matching.

Summary: You will find this part at the end of each topic. This component helps you to recap the whole topic. By going through the summary, you should be able to gauge your knowledge retention level. Should you find points in the summary that you do not fully understand, it would be a good idea for you to revisit the details in the module.

Key Terms: This component can be found at the end of each topic. You should go through this component to remind yourself of important terms or jargon used throughout the module. Should you find terms here that you are not able to explain, you should look for the terms in the module.

References: The References section is where a list of relevant and useful textbooks, journals, articles, electronic contents, or sources can be found. The list can appear in a few locations such as in the *Course Guide* (at the References section), at the end of every topic or at the back of the module. You are encouraged to read or refer to the suggested sources to obtain the additional information needed and to enhance your overall understanding of the course.

ASSESSMENT METHOD
Please refer to ICYM E Learning

TOPIC 1

Introduction to Computer System

LEARNING OUTCOMES

By the end of topic, you should be able to:

1. Define the meaning of programming languages.
2. List the program development Tools 4GL

Introduction to Programming Language

To create a program, programmers sometimes write, or code, a program's instructions using a programming language. A programming language is a set of words, abbreviations, and symbol that enables a programmer to communicate instructions to a computer. Several hundreds programming languages exist today. Each language has its own rules for writing the instructions. Languages are often designed for specific purposes, such as scientific applications, business solutions, or Web page development.

Two types of languages are low-level and high-level. A low-level language is close to the level of the computer, which means it resembles the numeric machine language of the computer more than the natural language of humans. A low-level language is a programming language that is machine dependent. A machine dependent language runs on only one particular type of computers.

The easiest languages for people to learn are high-level languages. They are call "high-level" because they are closer to the level of humanreadability than computer-readability. High-level languages often are machine independent. A machine independent language can run on many different types of computers and operating systems.

A **computer program** is a set of instructions that directs a computer to perform a task.

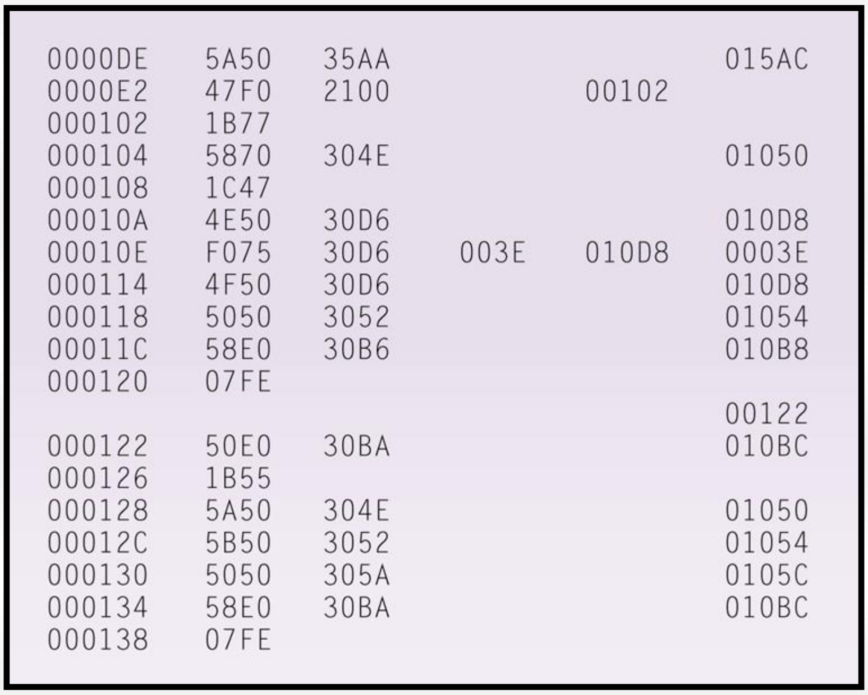
A **computer programmer** creates and modifies computer programs

1.1 Low-Level languages

Two types of **low-level languages** are machine languages and **assembly languages**. Machine language, known as the first generation of programming languages, is the only language the computer directly recognizes. Machine language instructions use a series of binary digits (1s and 0s) or a combination of numbers and letters that represents binary digits. The binary digits correspond to the on and off electrical states.

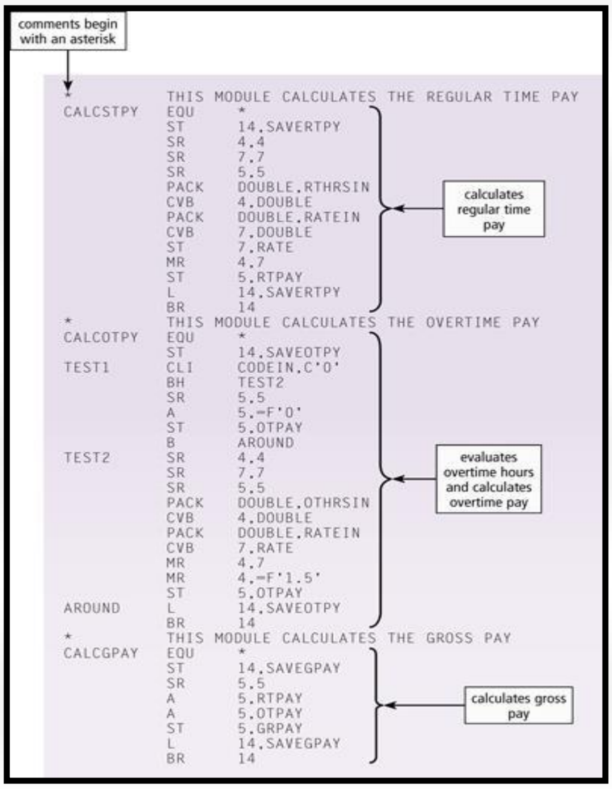
Coding in machine language is tedious and time-consuming With an assembly language, the second generation of programming languages, a programmer writes instructions using symbolic instructions codes. Assembly languages also use symbolic addresses. A symbolic address is a meaningful name that identifies a storage location. Assembly languages can be difficult to learn. In addition, programmers must convert an assembly language program into machine language before the computer can execute, or run, the program.

That is, the computer cannot execute the assembly source program. A source program is the program that contains the language instructions, or code, to be converted to machine language. To convert the assembly language source program into machine language, programmers use a program called assembler.



```
0000DE   5A50   35AA                   015AC
0000E2   47F0   2100           00102
000102   1B77
000104   5870   304E                   01050
000108   1C47
00010A   4E50   30D6                   010D8
00010E   F075   30D6   003E    010D8   0003E
000114   4F50   30D6                   010D8
000118   5050   3052                   01054
00011C   58E0   30B6                   010B8
000120   07FE
                                       00122
000122   50E0   30BA                   010BC
000126   1B55
000128   5A50   304E                   01050
00012C   5B50   3052                   01054
000130   5050   305A                   0105C
000134   58E0   30BA                   010BC
000138   07FE
```

A sample machine language program, coded using hexadecimal number system


comments begin with an asterisk
calculates regular time pay
evaluates overtime hours and calculates overtime pay
calculates gross pay

```
*          THIS MODULE CALCULATES THE REGULAR TIME PAY
CALCSTPY   EQU    *
           ST     14,SAVERTPY
           SR     4,4
           SR     7,7
           SR     5,5
           PACK   DOUBLE,RTHRSIN
           CVB    4,DOUBLE
           PACK   DOUBLE,RATEIN
           CVB    7,DOUBLE
           ST     7,RATE
           MR     4,7
           ST     5,RTPAY
           L      14,SAVERTPY
           BR     14
*          THIS MODULE CALCULATES THE OVERTIME PAY
CALCOTPY   EQU    *
           ST     14,SAVEOTPY
TEST1      CLI    CODEIN,C'O'
           BH     TEST2
           SR     5,5
           A      5,=F'0'
           ST     5,OTPAY
           B      AROUND
TEST2      SR     4,4
           SR     7,7
           SR     5,5
           PACK   DOUBLE,OTHRSIN
           CVB    4,DOUBLE
           PACK   DOUBLE,RATEIN
           CVB    7,RATE
           MR     4,7
           MR     4,=F'1.5'
           ST     5,OTPAY
AROUND     L      14,SAVEOTPY
           BR     14
*          THIS MODULE CALCULATES THE GROSS PAY
CALCGPAY   EQU    *
           ST     14,SAVEGPAY
           SR     5,5
           A      5,RTPAY
           A      5,OTPAY
           ST     5,GRPAY
           L      14,SAVEGPAY
           BR     14
```

An assembly language payroll program

Checkpoint 1.1

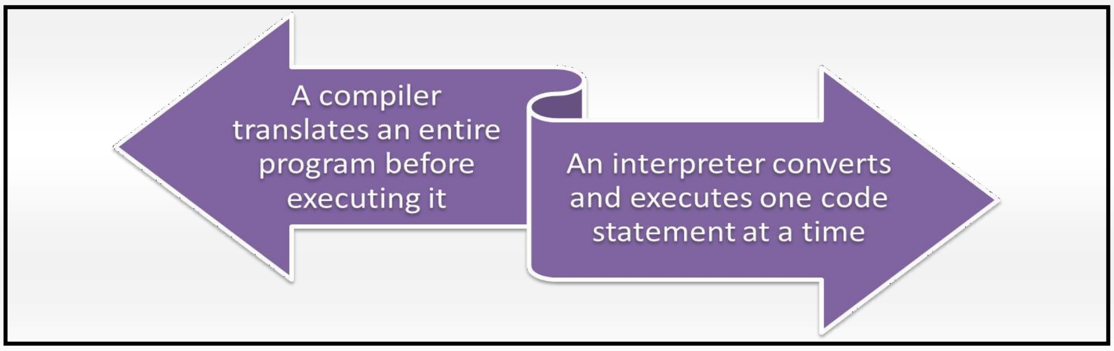
Instructions: Find the true or false statement below. Then, rewrite the remaining false statements so they are true.

1. Low-level languages often are machine independent.
2. Machine language and assembly language are types of low-level language.
3. The easiest languages for people to learn are high-level languages

1.2 Procedural languages

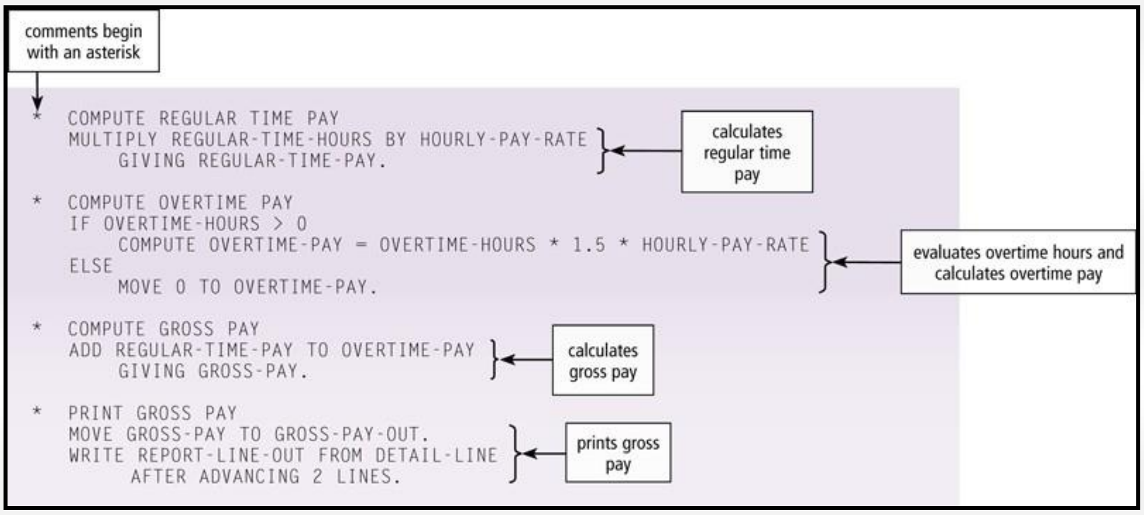
The disadvantages of machine language and assembly language (low-level) languages led to the development of procedural languages in the late 1950s and 1960s. In a procedural language, the programmer writes instructions that tell the computer what to accomplish and how to do it.

With a procedural language, often called a third-generation language (3GL), a programmer uses a series of English-like words to write instructions. For example, ADD stands for addition and PRINT means to print.


A compiler translates an entire program before executing it
An interpreter converts and executes one code statement at a time

C The C programming language, developed in the early 1970s by Dennis Ritchie at Bell Laboratories, originally was designed for writing system software. Today, many programs are written in C. C run on almost any type of computer with any operating system, but it is used most often with the UNIX and Linux operating system.

COBOL COBOL (COmmonBusiness-Oriented Language) is designed for business applications, but easy to read because of the English-like statements.


comments begin with an asterisk
calculates regular time pay
evaluates overtime hours and calculates overtime pay
calculates gross pay
prints gross pay

```
*  COMPUTE REGULAR TIME PAY
   MULTIPLY REGULAR-TIME-HOURS BY HOURLY-PAY-RATE
       GIVING REGULAR-TIME-PAY.

*  COMPUTE OVERTIME PAY
   IF OVERTIME-HOURS > 0
       COMPUTE OVERTIME-PAY = OVERTIME-HOURS * 1.5 * HOURLY-PAY-RATE
   ELSE
       MOVE 0 TO OVERTIME-PAY.

*  COMPUTE GROSS PAY
   ADD REGULAR-TIME-PAY TO OVERTIME-PAY
       GIVING GROSS-PAY.

*  PRINT GROSS PAY
   MOVE GROSS-PAY TO GROSS-PAY-OUT.
   WRITE REPORT-LINE-OUT FROM DETAIL-LINE
        AFTER ADVANCING 2 LINES.
```

An excerpt from a COBOL payroll program. The code shows the computations for regular time pay, overtime pay, and gross pay; the decision to evaluate the overtime hours; and the output of the gross pay.

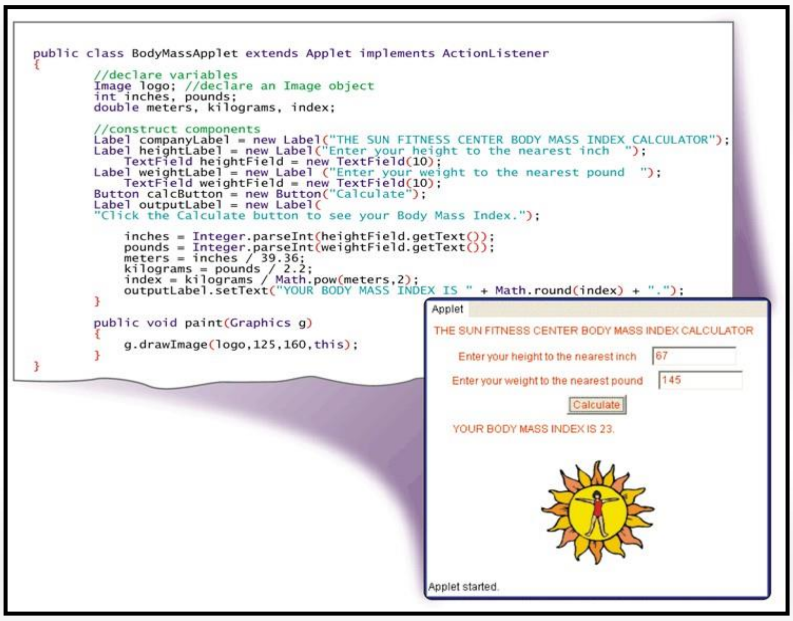
OOP An object-oriented programming (OOP) language allows programmers the ability to reuse and modify existing objects. Other advantages of OOP include:

- Objects can be reused
- Programmers create application faster
- Works well in RAD environment
- Most program development tools are **IDE**s

RAD
(Rapid Application Development) is a method of developing software, in which the programmer writes and implements a program in segments instead of waiting until the entire program is completed.

IDE
(integrated development environment) includes tools for building graphical interfaces, an editor for entering program code, a compiler and/or interpreter, and a debugger (to remove errors, which is discussed later in the chapter).

Java Java is an object-oriented programming language developed by Sun Microsystems. Java may be used to develop programs that run over the Internet, in a Web browser. Java uses a just-in-time compiler to convert the machine-independent code into machine-dependent code that is executed immediately. Programmers use various Java Platform implementations, developed by Sun Microsystems, which provide development tools for creating programs for all sizes of computers.



```
public class BodyMassApplet extends Applet implements ActionListener
{
        //declare variables
        Image logo; //declare an Image object
        int inches, pounds;
        double meters, kilograms, index;

        //construct components
        Label companyLabel = new Label("THE SUN FITNESS CENTER BODY MASS INDEX CALCULATOR");
        Label heightLabel = new Label("Enter your height to the nearest inch  ");
            TextField heightField = new TextField(10);
        Label weightLabel = new Label ("Enter your weight to the nearest pound  ");
            TextField weightField = new TextField(10);
        Button calcButton = new Button("Calculate");
        Label outputLabel = new Label(
        "Click the Calculate button to see your Body Mass Index.");

            inches = Integer.parseInt(heightField.getText());
            pounds = Integer.parseInt(weightField.getText());
            meters = inches / 39.36;
            kilograms = pounds / 2.2;
            index = kilograms / Math.pow(meters,2);
            outputLabel.setText("YOUR BODY MASS INDEX IS " + Math.round(index) + ".");
        }

        public void paint(Graphics g)
        {
            g.drawImage(logo,125,160,this);
        }
}
```

```
Applet
THE SUN FITNESS CENTER BODY MASS INDEX CALCULATOR
Enter your height to the nearest inch   67
Enter your weight to the nearest pound   145
Calculate
YOUR BODY MASS INDEX IS 23.
Applet started.
```

A portion of Java program and the window the program displays

1.3 Object-Oriented Programming Languages and Program Development Tools

The Microsoft .NET Framework allows almost any type of program to run on the Internet or an internal business network, as well as computers and mobile devices. Similarly ASP.NET is a Web application framework that provides the tools necessary for the creation of dynamic Web sites.

Using .NET and/or ASP.NET, programmers easily can develop Web applications, Web services, and Windows programs. Examples of languages that support .NET include C++, C#, Visual Basic, Delphi, and Power Builder. The following sections discuss each of these languages.

i. C++ is an object-oriented programming language that is an extension of the C programming language. Additional features for working with objects, classes, events, and other object-oriented concepts.Programmers commonly use C++ to develop database and Web applications.

ii. C# is based on C++ and was developed by Microsoft.

iii. Visual Studio is Microsoft's suite of program development tools
   - ☼ Visual Basic is based on the BASIC programming language
   - ☼ Visual C++ is based on C++
   - ☼ Visual C# combines the programming elements of C++ with an easier, rapid-development environment

iv. A visual programming language is a language that uses a visual or graphical interface for creating all source code.

v. Borland's Delphi is a powerful program development tool that is ideal for building large-scale enterprise and Web applications in a RAD environment.

vi. PowerBuilder is a powerful program development RAD tool. Best suited for Web-based, .NET, and large-scale enterprise object-oriented applications.

1.4 Others Programming Languages and Program Development Tools 4GL

A 4GL (fourth-generation language) is a nonprocedural language that enables users and programmers to access data in a database. With a nonprocedural language, the programmer writes English-like instructions or interacts with a graphical environment to retrieve data from files or a database. Many object-oriented program development tools use 4GLs. One popular 4GL is SQL. SQL is a query language that allows users to manage, update, and retrieve data in a relational DBMS.

Classic Programming Languages in addition to the programming languages discussed on the previous languages discussed on the previous pages, programmers sometimes use the language listed, which were more popular in the past than today.

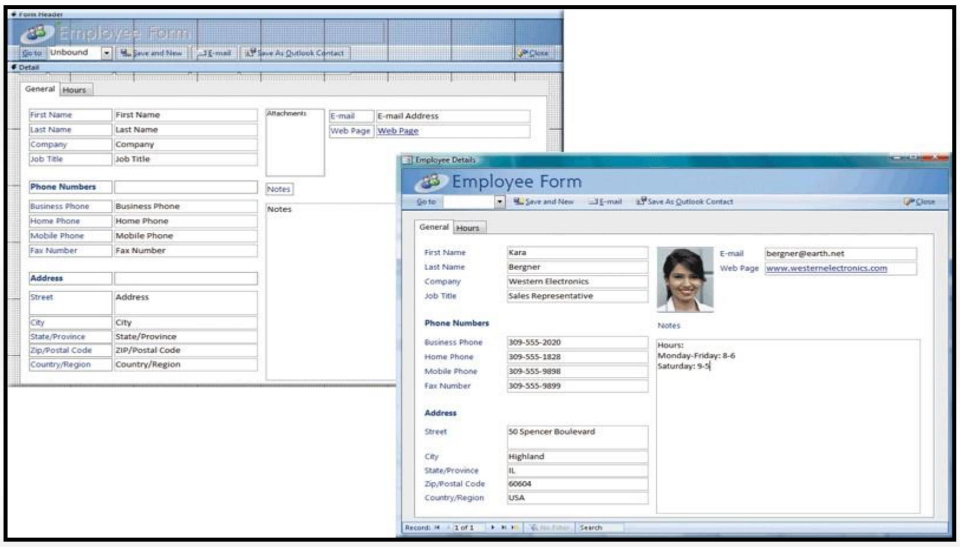
Ada	Derived from Pascal, developed by the U.S. Department of Defense, named after Augusta Ada Lovelace Byron, who is thought to be the first female computer programmer.
ALGOL	ALGOLrithmic Language, the first structured procedural language
APL	A Programming Language a scientific language designed to manipulate tables of numbers.
BASIC	Beginners All-purpose Symbolic Instructions Code, developed by John Kemeny and Thomas Kurtz as a simple, interactive problem-solving language.
Forth	Similar to C, used for small computerized devices.
FORTRAN	FORmulaTRANslator, one of the first high-level programming languages used for scientific applications.
HyperTalk	An object-oriented programming language developed by Apple to manipulate cards that can contain text, graphics, and sound.
LISP	LISt Processing, a language used for artificial intelligence applications.
Logo	An educational tool used to teach programming and problem solving to children
Modula-2	A successor to Pascal used for developing system software.
Pascal	Developed to teach students structured programming concept, named in honor of Blaise Pascal, a French mathematician who developed one of the earliest calculating machines.
PILOT	Programmed Inquiry Learning Or Teaching, used to write computer-aided instruction programs.
PL/1	Programming Language One, a business and scientific language that combines many features of FORTRAN and COBOL.
Prolog	PROgrammingLOGic, used for development of artificial intelligence applications
RPG	Report Program Generator, used to assist businesses with generating reports and access/update data in databases.
Smalltalk	Object-oriented programming language.

Classic Programming Languages

Macros A Macro is a series of statements that instructs a program how to complete a task. Macros allow users to automate routine, repetitive, or difficult tasks in application software such as word processing, spreadsheet, or database programs. That is, users can create simple programs within the software by writing macros. You usually create a macro in one of two ways:

i. record the macro – if you want to automate a routine or repetitive task such as formatting or editing. A macro recorder is similar to movie camera because both record all actions until turned off.
ii. write the macro – when you become familiar with programming techniques, you can write your own macros instead of recording them.

Application Generators An Application Generator is a program that creates source code or machine code form a specification of the required functionality. When using an application generator, a programmer or user works with menu-driven tools and graphical user interfaces to define the desired specifications. Application generators most often are bundled with or are included as part of a DBMS. An application generator typically consists of a report writer, form, and menu generator.



First Name	First Name
Last Name	Last Name
Company	Company
Job Title	Job Title
Phone Numbers	
Business Phone	Business Phone
Home Phone	Home Phone
Mobile Phone	Mobile Phone
Fax Number	Fax Number
Address	
Street	Address
City	City
State/Province	State/Province
Zip/Postal Code	ZIP/Postal Code
Country/Region	Country/Region
E-mail	E-mail Address
Web Page	Web Page
Notes	Notes

First Name	Kara
Last Name	Bergner
Company	Western Electronics
Job Title	Sales Representative
Phone Numbers	
Business Phone	309-555-2020
Home Phone	309-555-1828
Mobile Phone	309-555-9898
Fax Number	309-555-9899
Address	
Street	50 Spencer Boulevard
City	Highland
State/Province	IL
Zip/Postal Code	60604
Country/Region	USA
E-mail	bergner@earth.net
Web Page	www.westernelectronics.com

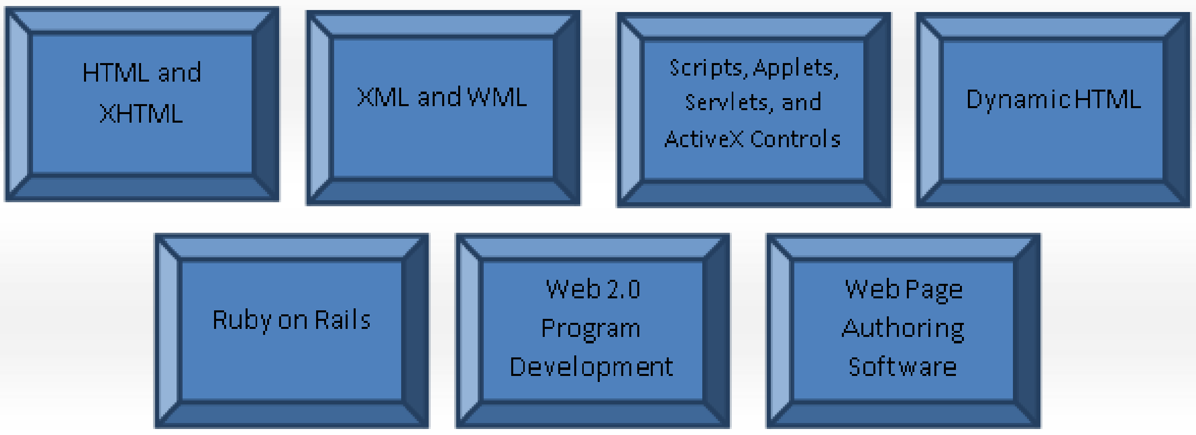
Notes

Hours:
Monday-Friday: 8-6
Saturday: 9-5

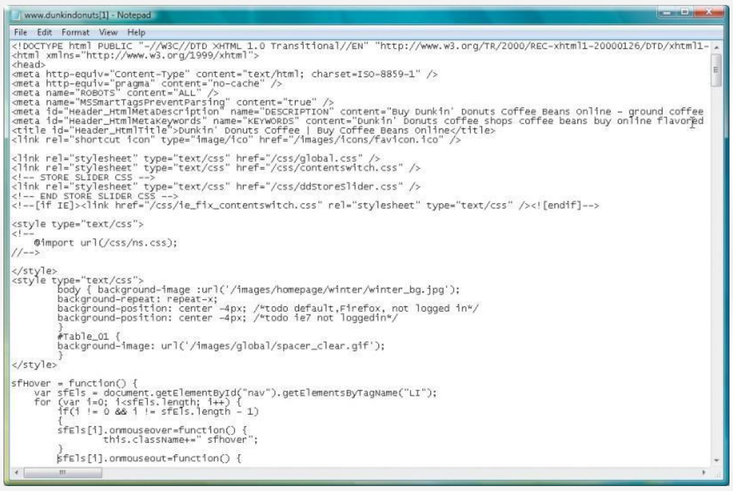
Application Generator : A form design and the resulting filled-in form created with Microsoft Access

1.5 Web Page Development

The designers of Web pages, known of **Web developers,** use a variety of techniques to create Web pages. The following sections discuss these techniques.

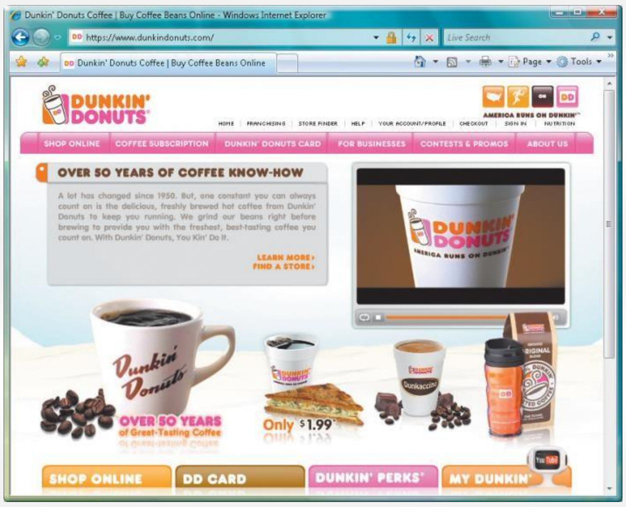

HTML and XHTML
XML and WML
Scripts, Applets, Servlets, and ActiveX Controls
Dynamic HTML
Ruby on Rails
Web 2.0 Program Development
Web Page Authoring Software

HTML and XHTML **HTML (Hypertext Markup Language)** is a special formatting language that programmers use to format documents for display on the Web. XHTML (extensible HTML) is a markup language that allows Web sites to be displayed more easily on mobile devices.



```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/2000/REC-xhtml1-20000126/DTD/xhtml1-
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
<meta http-equiv="pragma" content="no-cache" />
<meta name="ROBOTS" content="ALL" />
<meta name="MSSmartTagsPreventParsing" content="true" />
<meta id="Header_HtmlMetaDescription" name="DESCRIPTION" content="Buy Dunkin' Donuts Coffee Beans Online - ground coffee
<meta id="Header_HtmlMetaKeywords" name="KEYWORDS" content="Dunkin' Donuts coffee shops coffee beans buy online flavored
<title id="Header_HtmlTitle">Dunkin' Donuts Coffee | Buy Coffee Beans Online</title>
<link rel="shortcut icon" type="image/ico" href="/images/icons/favicon.ico" />

<link rel="stylesheet" type="text/css" href="/css/global.css" />
<link rel="stylesheet" type="text/css" href="/css/contentswitch.css" />
<!-- STORE SLIDER CSS -->
<link rel="stylesheet" type="text/css" href="/css/ddStoreSlider.css" />
<!-- END STORE SLIDER CSS -->
<!--[if IE]><link href="/css/ie_fix_contentswitch.css" rel="stylesheet" type="text/css" /><![endif]-->

<style type="text/css">
<!--
    @import url(/css/ns.css);
//-->

</style>
<style type="text/css">
        body { background-image :url('/images/homepage/winter/winter_bg.jpg');
        background-repeat: repeat-x;
        background-position: center -4px; /*todo default,Firefox, not logged in*/
        background-position: center -4px; /*todo ie7 not loggedin*/
        }
        #Table_01 {
        background-image: url('/images/global/spacer_clear.gif');
        }
</style>

sfHover = function() {
    var sfEls = document.getElementById("nav").getElementsByTagName("LI");
    for (var i=0; i<sfEls.length; i++) {
        if(i != 0 && i != sfEls.length - 1)
        {
        sfEls[i].onmouseover=function() {
                this.className+=" sfhover";
        }
        sfEls[i].onmouseout=function() {
```

Portion of XHTML program



Portion of resulting Web page

The portion of XHTML code in the top figure generates a portion of a Web page shown in the bottom figure

XML and WML **XML (Extensible Markup Language)** is an increasingly popular format for sharing data that allows Web developers to create customized tags and use predefined tags to display content appropriately on various devices. XML separates the Web page content from its format, allowing the Web browser to display the contents of a Web page in a form appropriate for the display device. For example, a smart phone, a PDA, and a notebook computer all could display the same XML page or use different formats or sections of the XML page.

Wireless devices use a subset of XML called WML. WML (wireless markup language) allows Web developers to design pages specifically for microbrowsers. Many smart phones and other mobile devices use WML as their markup language.

Scripts, Applets, Servlets, and ActiveX Controls Markup Languages tell a browser how to display text and images, set up lists and option buttons, and establish links on a Web page. By adding dynamic content and interactive elements such as scrolling messages, animated graphics, forms, pop-up windows, and interaction, Web pages become much more interesting. To add these elements, Web developers write small programs called scripts, applets, servlets, and ActiveX controls. These programs run inside of another program. This is different from programs discussed thus far, which are executed by the operating system. In this case, the Web browser executes these short programs.

One reason for using scripts, applets, servlets, and ActiveX controls is to add special multimedia effects to Web pages. Examples include animated graphics, scrolling messages, calendars, and advertisements. Another reason to use these programs is to include interactive capabilities on Web pages.

Scripting Languages Programmers write scripts, applets, servlets, or ActiveX controls using a variety of languages. These include some of the languages previously discussed, such as Java, C++, C#, and Visual Basic. Some programmers use scripting languages. A scripting language is an interpreted language that typically is easy to learn and use. Popular scripting languages include JavaScript, Perl, PHP, Rexx, Tcl, and VBScript.

- ☼ **JavaScript** is an interpreted language that allows a programmer to add dynamic content and interactive elements to a Web page. These elements include alert messages, scrolling text, animations, drop-down menus, data input forms, pop-up windows, and interactive quizzes.
- ☼ **Perl** (Perl Extraction and Report Language) originally was developed by Larry Wall at NASA's Propulsion Laboratory as a procedural language similar to C and C++. The latest release of Perl, however, is an interpreted scripting language. Because Perl has powerful text processing capabilities, it has become a popular language for writing scripts.
- ☼ **PHP** which stands for PHP: Hypertext, Preprocessor is a free, open source scripting language. PHP is similar to C, Java, and Perl.
- ☼ **Rexx** (REstructuredeXtendedeXecutor) was developed by Mike Cowlishaw at IBM as a procedural interpreted scripting language for both the professional programmer and the nontechnical user.
- ☼ **Tcl** (Tool Command Language) is an interpreted scripting language created by Dr. John Outershout and maintained by Sun Microsystem Laboratories.

- ☼ **VBScript** (Visual Basic, Scripting Edition) is a subset of the Visual Basic language that allows programmers to add intelligence and interactivity to Web pages. As with JavaScript, Web developers embed VBScript code directly into an HTML document.

Dynamic HTML **Dynamic HTML (DHTML)** is a newer type of HTML that allows Web developers to include more graphical interest and interactivity in a Web page. Typically, Web pages created with DHTML are more animated and responsive to user interaction. Colors change, font sizes grow, objects appear and disappear as a user moves the mouse, and animations dance around the screen.

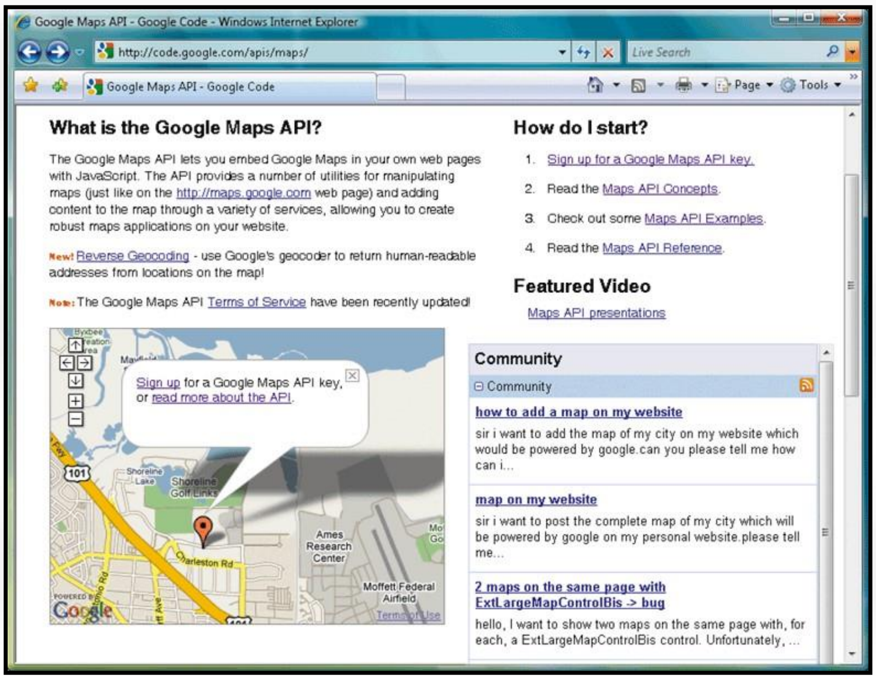
Ruby on Rails **Ruby on Rails** is an open source framework that provides technologies for developing object-oriented, database-driven Web sites. Ruby on Rails is designed to make Web developers more productive by providing them an easy-to-use environment and eliminating time-consuming steps in the Web development process.

Web 2.0 Program Development Web 2.0 sites often use RSS. Ajax which stands for Asynchronous JavaScript and XML, is a method of creating interactive Web applications designed to provide immediate response to user requests. Instead of refreshing entire Web pages, Ajax works with the Web browser to update only changes to the Web page. This technique saves time because the Web application does not spend time repeating sending unchanged information across network.

Ajax combines several programming tools: JavaScript or other scripting language, HTML or XHTML, and XML. Examples of Web sites that use Ajax are Google Maps and Flickr.

Most Web 2.0 sites also use APIs so that Web developers can create their own Web applications. An API (application program interface) is a collection of tools that programmers use to interact with an environment such as a

Web site or operating system. Mapping Web sites, for example, include APIs that enable programmers to integrate maps into their Web sites.


Google Maps API - Google Code - Windows Internet Explorer
http://code.google.com/apis/maps/
Live Search
Google Maps API - Google Code
Page
Tools

What is the Google Maps API?

The Google Maps API lets you embed Google Maps in your own web pages with JavaScript. The API provides a number of utilities for manipulating maps (just like on the http://maps.google.com web page) and adding content to the map through a variety of services, allowing you to create robust maps applications on your website.

New! Reverse Geocoding - use Google's geocoder to return human-readable addresses from locations on the map!

Note: The Google Maps API Terms of Service have been recently updated!

Sign up for a Google Maps API key, or read more about the API.

How do I start?

1. Sign up for a Google Maps API key.
2. Read the Maps API Concepts.
3. Check out some Maps API Examples.
4. Read the Maps API Reference.

Featured Video

Maps API presentations

Community

Community

how to add a map on my website

sir i want to add the map of my city on my website which would be powered by google.can you please tell me how can i...

map on my website

sir i want to post the complete map of my city which will be powered by google on my personal website.please tell me...

2 maps on the same page with ExtLargeMapControlBis -> bug

hello, I want to show two maps on the same page with, for each, a ExtLargeMapControlBis control. Unfortunately, ...

Google Maps provides tools for programmers to integrate APIs into their Web sites

Web Page Authoring Software You do not need to learn HTML to develop a Web page. You can use Web page authoring software to create sophisticated Web pages that include graphical images, video, audio, animation, and other special effects. Web page authoring software generates HTML and XHTML tags from your Web page design.

Four Web page authoring programs are:

Dreamweaver | **Expression Web** | **Flash** | **SharePoint Designer**

☼ **Dreamweaver**, by Adobe System, is a Web page authoring program that allows Web developers to create, maintain, and manage professional Web sites.

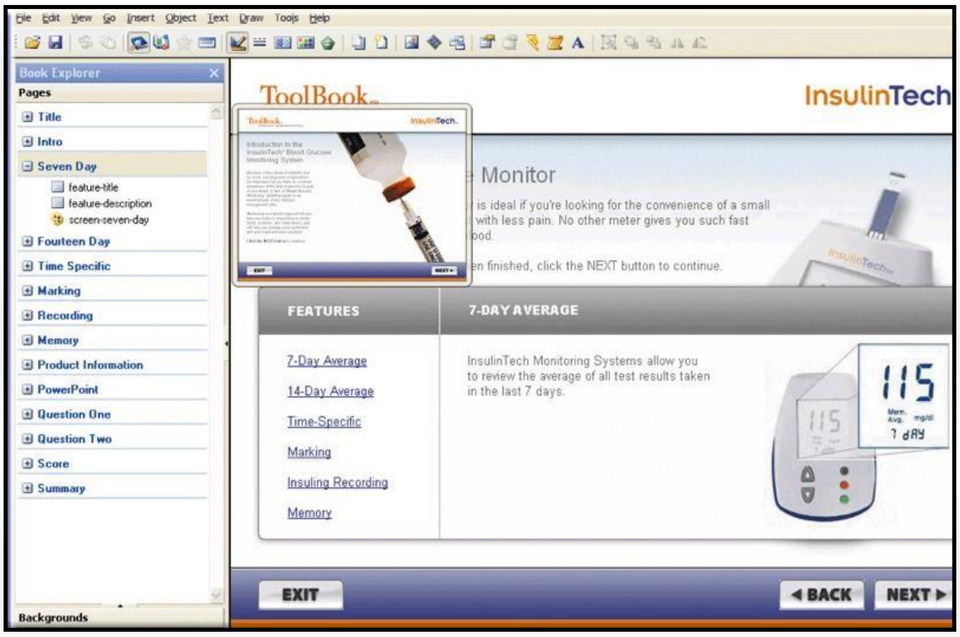
☼ **Expression Web**, is Microsoft's Web page authoring program that enables Web developers to create professional, dynamic, interactive

Web sites. Expression Web developers to combine interactive content with text, graphics, audio, and video.

- ☼ **Flash,** by Adobe Systems, is a Web page authoring program that enables Web developers to combine interactive content with text, graphics, audio, and video.
- ☼ **SharePoint Designer** is a Web page authoring program that is part of the Microsoft Office and SharePoint families of products.

1.5 Multimedia Program Development

- ☼ Multimedia authoring software allows programmers to combine text, graphics, animation, audio, and video in an interactive presentation. Many developers use multimedia authoring software for computer-based training (CBT) and Web-based training (WBT). Popular multimedia authoring software includes **ToolBook** and **Director**. Many businesses and colleges use ToolBook to create content for distance learning courses.



is ideal if you're looking for the convenience of a small with less pain. No other meter gives you such fast ood.

en finished, click the NEXT button to continue.

InsulinTech Monitoring Systems allow you to review the average of all test results taken in the last 7 days.

Checkpoint 1.2

Instructions: Find the true or false statement below. Then, rewrite the remaining false statements so they are true.

1. The idea of inheritance makes object-oriented programming more reusable then code generated by top-down design.
2. Java is the ideal development language, which is why other programming languages are beginning to lose their importance.
3. Prototyping is a form of rapid application development (RAD), which enables programmers to build software that executes incredibly

Quiz Yourself Online: Do Self Test in the E-Learning

ACTIVITY

1. **A program written in a high -level language is translated into machine code. This is so that it can be processed by a computer. Name one type of translator that can be used. Describe how your answer to part(b) translate this program.**
2. **High level languages can be compiled or interpreted, give two difference between a compiler and an interpreter.**

Self Assessment

Exercise 1 eLearning

KEY TERM

Programming language	**Source program**
computer programmer	**assembler**
Computer program	**Third-generation language**
Machine languages	**Web developers**
Assembly language	**Scripting language**

SUMMARY

- Introduced to the concept of programming fundamentals.
- Two common types of low level programming languages are assembly language and machine language.
- Introduced with other programming languages and program development Tools 4GL

REFERENCEES

1. Tony Gaddis, Starting Out with C++ - Early Objects (10th edition) ,Pearson, 2020.

TOPIC 2

Program Development

LEARNING OUTCOMES

By the end of topic, you should be able to:

1. List the six process steps involved in developing a program.
2. Solve problems by using two algorithm representative techniques.
3. Build a flow chart using the sequential, selective and repetitive control structure.
4. Build a pseudocode using the sequential, selective and repetitive control structure.

2.1 System Development Life Cycle

A **system** is a set of components that interact to achieve a common goal. An **information system (IS**) is a collection of hardware, software, data, people, and procedures that work together to produce quality information. An information system supports daily, short-term, and long-range activities of users.

The type of information that users need often changes. When this occurs, the information system must meet the new requirements. In some cases, members of the system development team modify the current information system. In other cases, they develop an entirely new information system.

System development is a set of activities used to build an information system. System development activities often are grouped into larger categories called phases. This collection of phases sometimes is called the system development life cycle (SDLC). Many SDLCs contain five phases:

1. Planning
2. Analysis
3. Design
4. Implementation
5. Operation, Support, and Security

Each system development phase consists of a series of activities, and the phases form a loop. In theory, the five system development phases often appear sequentially, as shown in next figure. In reality, activities within adjacent phases often interact with one another – making system development a dynamic iterative process.

Planning Phase

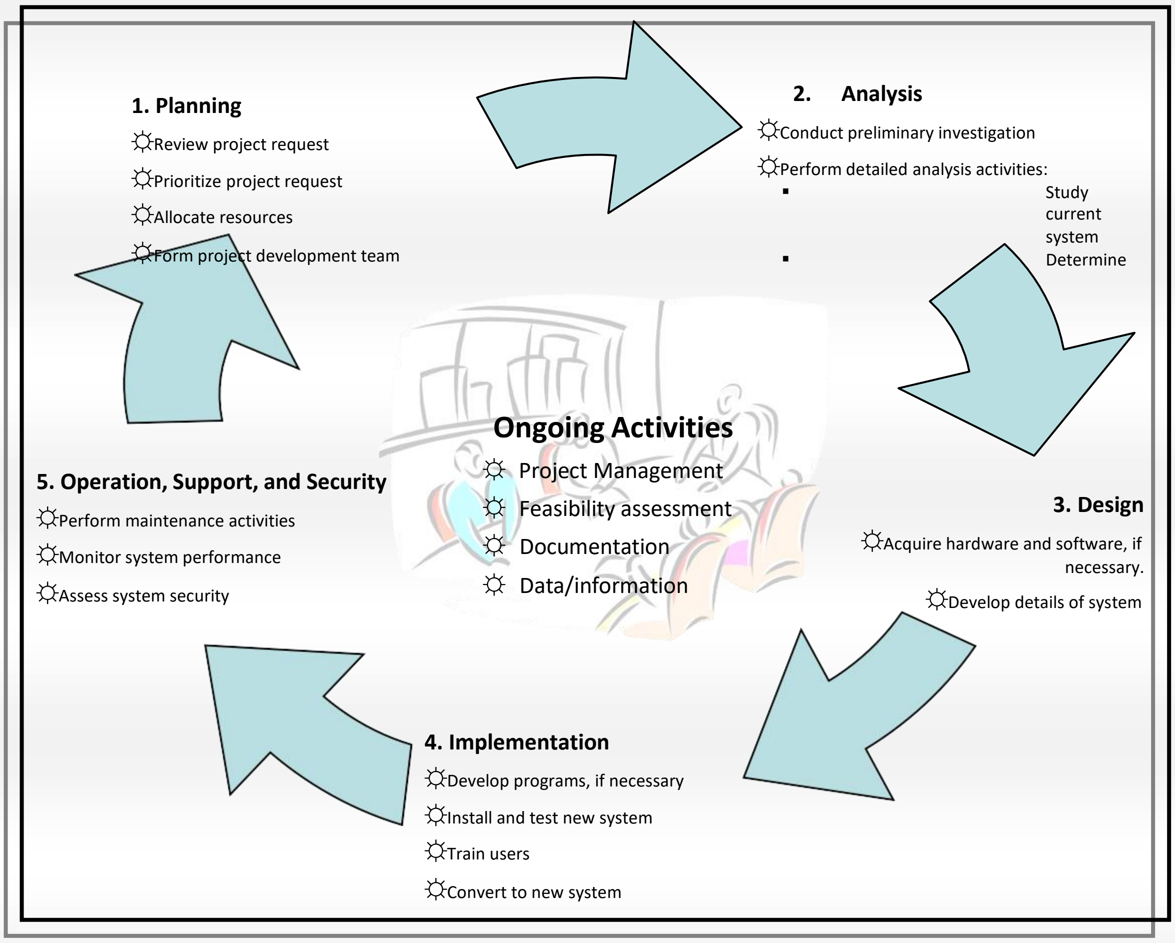
The **planning phase** for a project begins when the steering committee receives a project request. During the planning phase, four major activities are performed:

(i) Review and approve the project request;

(ii) Prioritize the project requests;

(iii) Allocate resources such as money, people, and equipment to approved projects; and

(iv) Form a project development team for each approved project.


1. Planning
Review project request
Prioritize project request
Allocate resources
Form project development team
2. Analysis
Conduct preliminary investigation
Perform detailed analysis activities:
Study current system
Determine
3. Design
Acquire hardware and software, if necessary.
Develop details of system
4. Implementation
Develop programs, if necessary
Install and test new system
Train users
Convert to new system
5. Operation, Support, and Security
Perform maintenance activities
Monitor system performance
Assess system security
Ongoing Activities
Project Management
Feasibility assessment
Documentation
Data/information

System development consists of five phases that form a loop. Several ongoing activities also take place throughout system development.

Analysis Phase

The analysis phase consists of two major activities:

(i) Conduct a preliminary investigation (feasibility study)	To determine the exact nature of the problem or improvement and decide whether it is worth pursuing.
(ii) Perform detailed analysis (logical design)	Study how the current system works, determine the users' wants, needs, and requirements; and recommend a solution.

Design Phase

The design phase consists of two activities:

(i) If necessary, acquire hardware and software and

(ii) Develop all of the details of the new or modified information system.

Implementation Phase

The purpose of the implementation phase is to construct, or build the new or modified system and then deliver it to the users. Members of the system development team perform four major activities in this phase:

(i) Develop programs

(ii) Install and test the new system

(iii) Train users, and

(iv) Convert to the new system

Operation, Support, and Security Phase

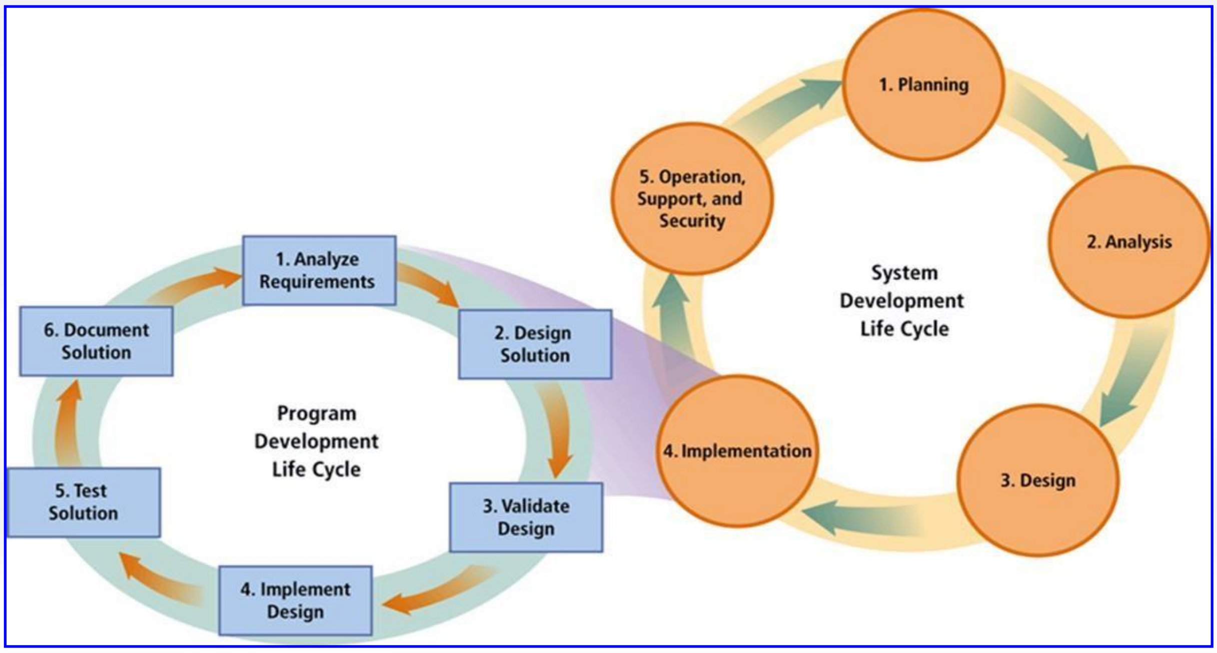
The purpose of the operation, support, and security phase is to provide ongoing assistance for an information system and its users after the system is implemented.

2.2 Program Development

Program development consists of a series of steps programmers use to build computer programs.

The program development life cycle (PDLC) guides computer programmers through the development of a program. The program development life cycle consists of six steps.

1. Analyze Requirements
2. Design Solution
3. Validate Design
4. Implement Design
5. Test Solution
6. Document Solution



The Program Development Life Cycle consists of six steps that form a loop. The program development life cycle is part of the implementation phase of the system development life cycle.

Program development is an ongoing process within system development. Each time someone identifies errors in or improvements to a program and requests program notifications, the Analyze Requirements step begins again. When programmers correct errors (called bugs) or add enhancements to an existing program, they are said to be maintaining the program. Program maintenance is an ongoing activity that occurs after a program has been delivered to users.

What Initiates Program Development?

As shown in figure before, system development consists of five phases: planning; analysis; design; implementations; and operation, support, and security. During the analysis phase, the development team recommends how to handle software needs. Choices include modifying existing programs, purchasing packaged software, building custom software in-house, or outsourcing some or all of the IT operation.

If the organization opts for in-house development, the design and implementation phases of system development become quite extensive. In the design phase, the analyst creates a detailed set of requirements for the programmers. Once the programmers receive the requirements, the implementation phase begins. At this time, the programmer analyzes the requirements of the problem to be solved. Thus, program development begins at the start of the implementation phase in system development.

The scope of the requirements largely determines how many programmers work on the program development. If the scope is large, a programming team that consists of group programmers may develop the programs. If the specifications are simple, a single programmer might complete all the development tasks. Whether a single programmer or a programming team, all the programmers involved must interact with users and members of the development team throughout program development

By following the steps in program development, programmers create programs that are correct (produce accurate information) and maintainable (easy to modify).

2.3 Algorithm

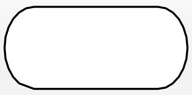
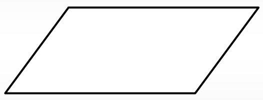
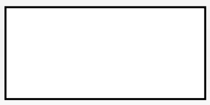
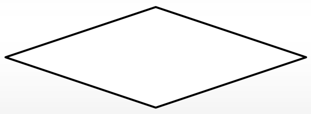
Algorithm is a set of logical sequential steps used to solve the problem.

2.3.1 Flow chart

A **flow chart** is a graphical or symbolic representation of a process

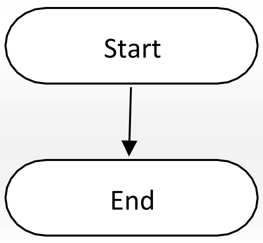
Flow chart is made up of geometry nodes that represent processes. These nodes are joined by arrows that will show the flow or continuity of the activities. It is not only used in programming but also in other matters such as loan approval. It will describe the whole process from the application, checking for eligibility, if not eligible, applying again, and if eligible, approval of application, and finally end of process.

There are many nodes/symbols that are used in a flow chart, but there are used in a flow chart, but there are four symbols that should be basically identified as shown in table below.

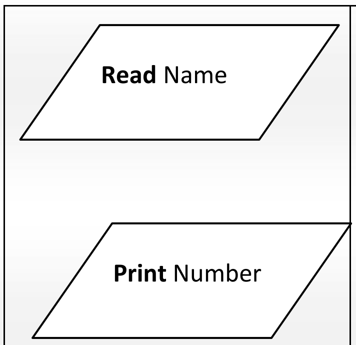
Symbol	Meaning
	Start/End
	Input/Output
	Process
	Condition

Flow Chart Symbols

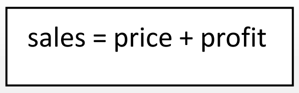
i) Start/End

	This symbol is used at the beginning and at the end of the flow chart. For the whole flow chart, there can only be one **Start** symbol and one **End** symbol. One arrow will exit from the Start symbol, after traversing through many different symbols; it will enter the End symbol. Only one arrow can exit from the Start symbol, but many arrows can enter the End symbol.

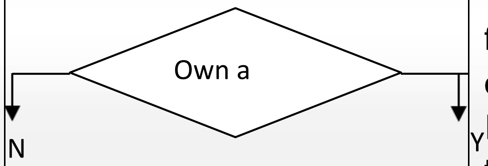
ii) Input/Output

	**Input** means data will be entered into the computer. For example, name, age or address. Text that is typed is called input. **Output** is response from the computer. Text that is displayed on the screen is called output. For example, the message: "WELCOME" is output **Input/Output** symbols are used when interacting between user and computer. It is used to receive information from users and providing information to the users.

iii) Process

	**Process** symbol is used to represent an activity that is being executed by the computer. It does not involve input and output. For example, sales=price + profit. Probably before this, the user has entered the price and the profit values (input). Now, the computer will get the sales value. Prior to that, the calculation will be done by the computer to get the sales value. This calculation process is represented by the process symbol.

iv) Condition

	When a problem has two choices, the data flow will have two branches. There is a condition that will determine which branch to be selected. Usually, this symbol is used with the statement "True/False" or "Yes/No". For example, the question, "Do you own a car?" There are two answers only: "Yes" or "No". This symbol is not used to get input but is used to process the input. The question is in the input symbol, but the input from the user will be processed by the condition symbol. Only one arrow will enter the symbol, and there must be exactly two arrows exiting from it, one representing Yes/True and the other No/False.

All these symbols must be joined by an arrow. It is not possible to have a symbol without any arrows. These arrows can only start at the Start symbol, and end at the End symbol. Input/Output, process and condition symbols must have in and out arrows. This means, when we follow the flow of the arrows from the start till the end, there will not be a dead end. We will move from one symbol to another until the end symbol is reached. Do not allow any symbol without an out arrow from it except the End symbol.

It is reminded once again, only one Start symbol and only one out arrow from it. This is where the activities start. Once traversing through the many symbols, the arrow might branch off, nevertheless all the arrows must end at the End symbol.

2.3.2 Pseudocode

Pseudocode consists of short, English phrases used to explain specific tasks within a program's algorithm

Pseudocode is the representation of algorithm that resembles actual program code. It does not use symbols to represent sequential steps, but steps are written using natural language that explains the processing involved in solving problems. The main reason in representing the problem in this form is that it is more systematic and its logic is easier to understand.

To represent algorithm using pseudocode, we need to use the writing rules as:

a) Every step in an algorithm should not have more than two actions.

b) Steps in an algorithm are executed in sequence.

c) The word Start shows that the process has started and the word End or Stop is used to show that the process has ended.

d) The action that is allowed includes declaring variable names to identify the set of variables that have a corresponding data type. Types of data might be integer, real, character or others.

e) To give an initial value to a variable.

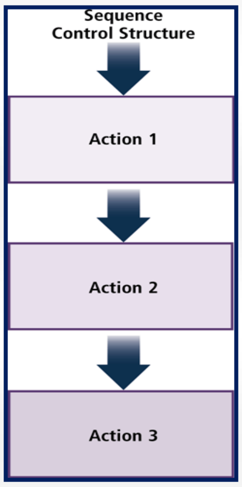
f) To use arithmetic symbols to state the addition, subtraction, multiplication, division operations and brackets to show operation priority.

2.4 Control Structure

When programmers are required to design the logic of a program, they typically use control structures to describe the tasks a program is to perform. A control structure, also known as a construct, depicts the logical order of program instructions. Three basic control structures are sequence, selection, and repetition.

2.4.1 Sequence Control Structure

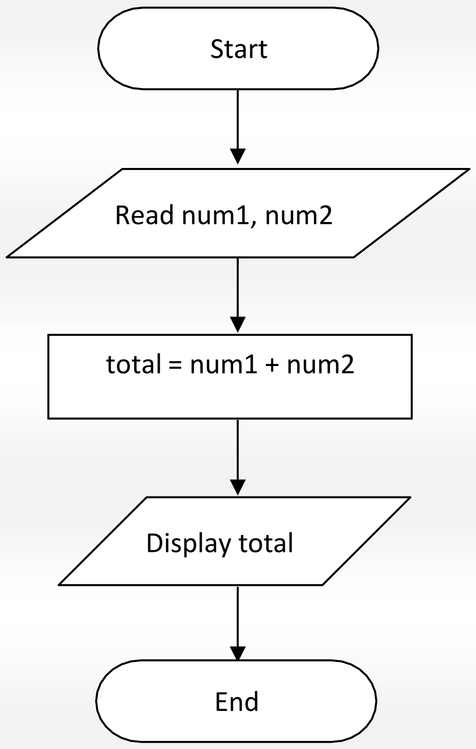
A sequence control structure shows one or more actions following each other in order. Actions include inputs, processes, and outputs. All actions must be executed; that is none to be skipped. Examples of actions are reading a record, calculating averages or totals, and printing totals.



The sequence control structure shows one or more actions followed by another

Example : User enters **two integer numbers**. Computer will add these two numbers and display its **total**.

Flow chart:	**Pseudocode:**

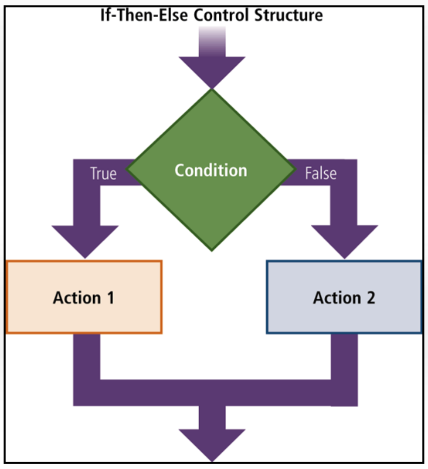
 Start
Read num1, num2
total = num1 + num2
Display total
End
	Start
	    Read num1, num2
	    total = num1 + num2
	    Display total
	End

2.4.2 Selection Control Structure

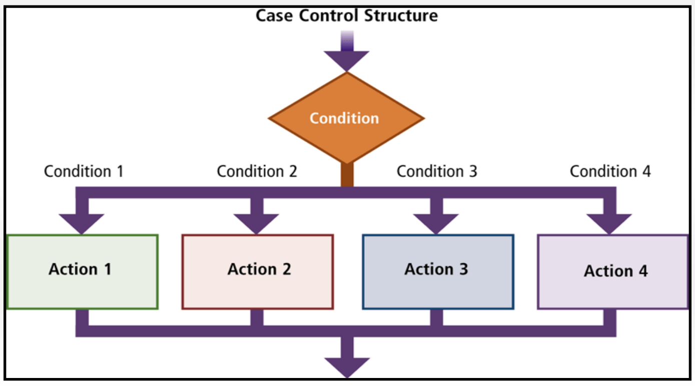
A **selection control structure** tells the program which action to take, based on a certain condition. Two common types of selection control structures are the **if-then-else** and the **case**.

When a program evaluates the condition in an if-then-else control structure, it yields one of the two possibilities: **TRUE** or **FALSE**. If the result of the condition is **true**, then the program performs one action. If the result is **false**, the program performs a different action.

For example, the if-then-else control structure can determine if an employee should receive overtime pay. A possible condition might be the following: Is Hours Worked greater than 40? If the response is yes (true), then the action would calculate overtime pay. If the response is no (false), then the action would set overtime pay equal to 0.



The if-then-else control structure directs the program toward one course of action or another based on the evaluation of a condition.

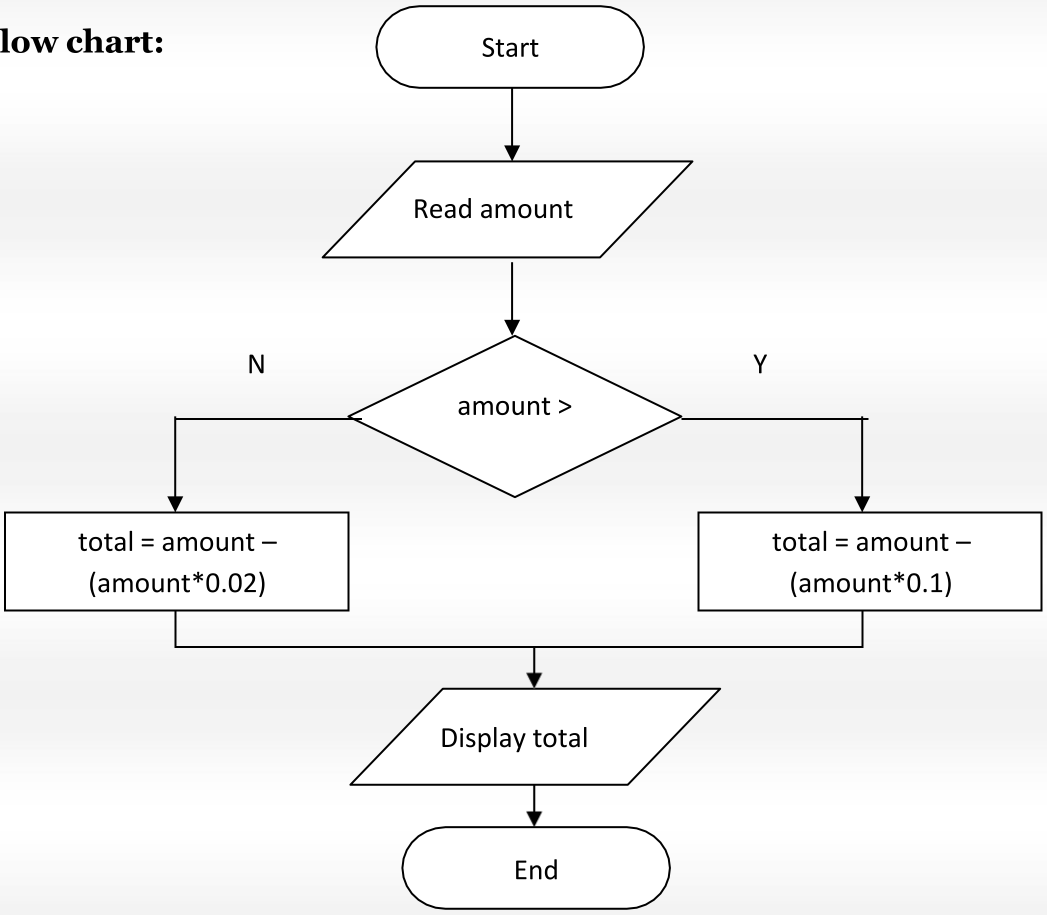


The case control structure allows for more than two alternatives when a condition is evaluated

With the case control structure, a condition can yield one of three or more possibilities. The size of beverage, for example, might be one of these options: small, medium, large, or extra large. A case control structure would determine the price of the beverage on the size purchase.

Example : Discount of 10% is given to the customer, if the amount purchased is more than RM1,000. If the amount purchased is less than RM1,000, discount of 2% is given.

Flow chart:



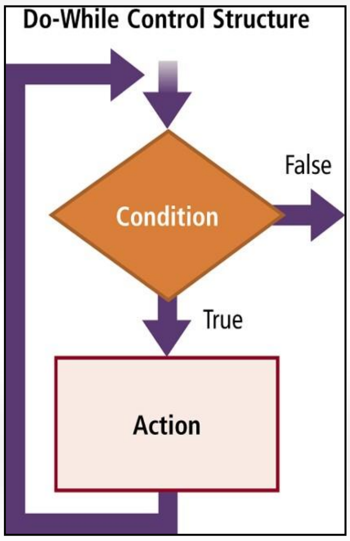
Pseudocode:

```
Start
        Read amount
        if (amount > 1000)
                total = amount – (amount*0.1)
        if not
                total = amount – (amount*0.02)
        end if
        Display total
Stop
```

2.4.3 Repetition Control Structure

The repetition control structure enables a program to perform one or more actions repeatedly as long as a certain condition is met. Many programmers refer to this construct as a loop. Two forms of the repetition control structure are the **do.. while** and **do.. until**.

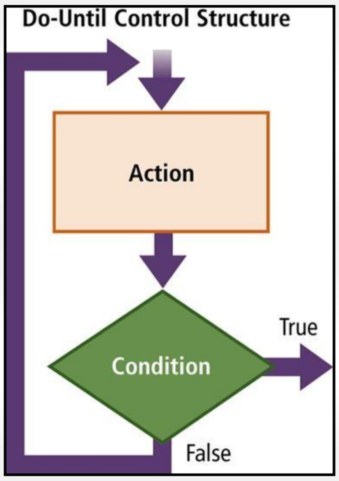
A **do-while** control structure repeats one or more times as long as a specified condition is true. This control structure tests a condition at the beginning of a loop. If the result of the condition is true, the program executes the action(s) inside the loop. Then, the program loops back and tests the condition again. If the result of the condition is still true, the program executes the action(s) inside the loop again. This looping process continues until the condition being tested becomes false. At that time, the program stops looping and moves to another set of actions.



The do-while control structure normally is used when the occurrence of an event is not quantifiable or predictable. For example, programmers frequently use the do-while control structure to process all records in a file. A payroll program using a do-while control structure to process all records in a file. A payroll program using a do-while control structure loops once for each employee. This program stop looping after it processes the last employee's record.

The do-while control structure tests the condition at the beginning of the loop. It exits the loop when the result of the condition is false

The do-until control structure is similar to the do-while but has two major differences: where it tests the condition and when it stops looping. First, the do-until control structures tests the condition at the end of the loop. The action(s) in a do-until control structure thus always will execute at least once.

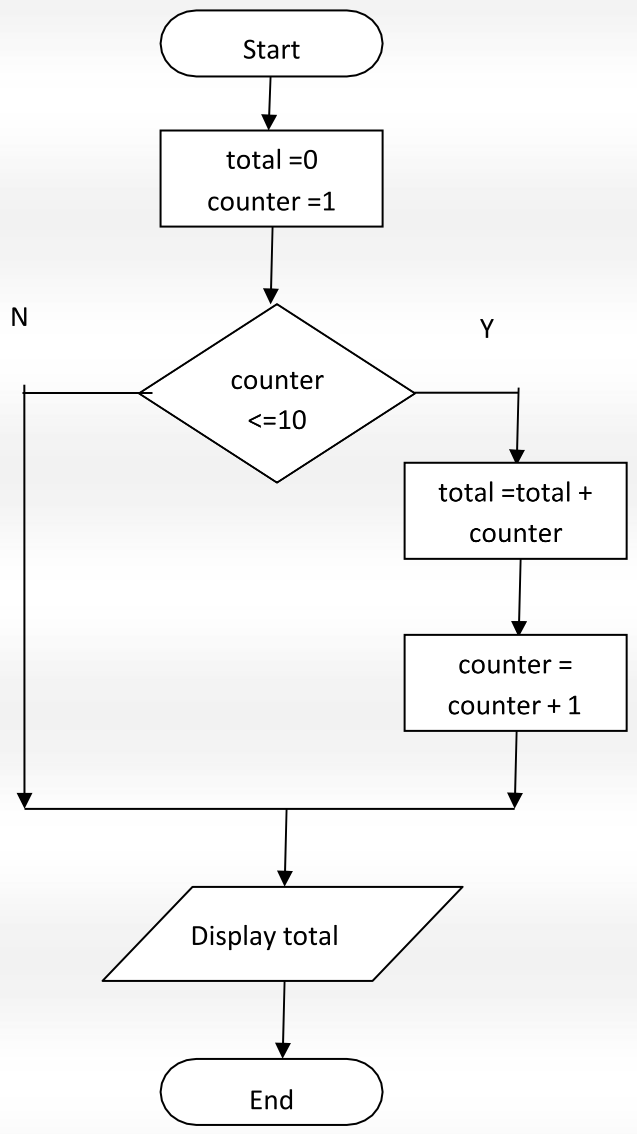


The loop in a do-while control structure, by contrast might not execute at all. That is, if the condition immediately is false, the action or actions in the do-while loop never execute. Second, do-until control structure continues looping until the condition is true – and then stops. This is different from the do-while control structure, which continues to loop while the condition is true.

The do-until control structure tests the condition at the end of the loop. It exits the loop when the result of the condition is true.

Example : Add the integer numbers from 1 to 10and display the total.

Flow chart:



Pseudocode:

```
Start
    total =0
    counter =1
    while (counter <=10)
        total = total + counter
        counter = counter +1
    end while
    Display total
End
```

Checkpoint 2.1

Instructions: Find the true and false statement below. Then, rewrite the remaining false statements so they are true.

1. Three basic control structures are sequence, selection, and pseudocode.
2. Programmers must convert an assembly language program into machine language before the computer can execute, or run, the program.
3. Programmers use a sequence control structure to show program modules graphically.
4. A sequence control structure tells the program which action to take, based on a certain condition.

Quiz Yourself Online: Do Self-Test in the E-Learning

ACTIVITY

Write the pseudocode two input two values for each staff in a private company. The two values are the salary and a character value that represent the staff performance for that particular year. The staff performance will be represent by 'C' for excellent, 'B' for good and 'L' for non-performer. The pseudocode should be able to calculate and display the salary increment for each staff, where the increment will be calculate based on the performance: 12% increment for the excellent staff, 8% for the good staff and there is no increment for the non-performer.

KEY TERM

System	**Pseudocode**
Planning phase	**data**
Program development	**Sequence**
Algorithm	**Selection**
Flowchart	**Repetition**

SUMMARY.

- Algorithm can be divided into flow chart and pseudocode.
- Flow chart shows the flow or the sequence of activities clearly and logically.
- Pseudocode can help in writing program code quickly, as it resembles program code.
- Input is all the information that is relevant and needed to execute a process.
- Output is the result that is needed.
- Selection structure is a structure design that gives a few choices during execution.
- Repetition structure is a structure where one block of statements is executed repeatedly.

REFERENCEES

2. Tony Gaddis, Starting Out with C++ - Early Objects (10th edition) ,Pearson, 2020.

TOPIC 3

Program Coding and Simple Input/Output

LEARNING OUTCOMES

By the end of topic, you should be able to:

1. Write simple C++ programs
2. Compile and execute C++ program and
3. Correct errors found in the programs.

3.1 The parts of a C++ Program

C++ programs have parts and components that serve specific purposes. We will begin by looking at Program 3-1.

PROGRAMS 3-1

```
// A simple C++ program
#include <iostream.h>

int main ( )

{
 cout<< "Programming is great fun!";
 return 0;
}

```

The output of the program is shown below. This is what appears on the screen when the program runs.

Program Output

```
Programming is great fun!
```

Comments help explain what's going on

Let's examine the program line by line. Here's the first line.

// A simple C++ program

The // marks the beginning of a **comment**. The compiler ignores everything from the double slash to the end of the line. That means you can type anything you want on that line and the compiler will never complain. Although comments are not required, they are very important to programmers.

Line 2 :

#include <iostream.h>

Because this line starts with a #, it is called **preprocessor directive.** The preprocessor reads your program before it is compiled and only executes those lines beginning with a # symbol.

The # include directive causes the preprocessor to include the contents of another file that is to be included. The word inside the brackets, iostream, is the name of the file that is to be included. The iostream file contains code that allows a C++ program to display output on the screen and read input from the keyboard. Because this program uses cout to display screen output, the iostream file must be included. The contents of the iostream file are included in the program at the point the #include statement appears. The iostream file is called a header file, so it should be included atthe head, or top, of the program.

Line 4 :

int main ( )

This marks the beginning of a function. A **function** can be taught of as a group of one or more programming statements that collectively has a name. The name of this function is main, and the set of parentheses that follows the name indicate that it is a function. The word int stands for "integer". It indicates that the function sends an integer value back to the operating system when it is finished executing.

Although most C++ programs have more than one function, every C++ program must have a function called main. It is the starting point of the program. If you are ever reading someone else's C++ program and want to find where it starts, just look for a function named main.

Line 5 :

{

This is called a left-brace, or an opening brace, and it is associated with the beginning of the function main. All the statements that make up a function are enclosed in a set of braces. If you look the third line down from the opening brace you'll see the closing brace. Everything between the two braces is the contents of the function main.

After the opening brace you see the following statement in Line 6:

cout<< "Programming is great fun!";

This line displays a message on the screen. The message "**Programming is great fun!"**is printed without the quotation marks. In programming terms, the group of characters inside the quotation marks is called a **string literal** or **string constants**.

At the end of the line is a semicolon (;). Just as a period marks the end of a sentence, a semicolon marks the end of a complete statement in C++.

Line 8 reads:

return 0;

This sends the integer value 0 back to the operating system upon the program's completion. The value 0 usually indicates that a program executed successfully.

Line 9 contains the closing brace:

}

This brace marks the end of the main function. Since main is the only function in this program, it also marks the end of the program.

In the sample program you encountered several sets of special characters. Table 3-1 provides a short summary of how they were used.

Table 3-1 Special characters

Character	Name	Description
//	Double Slash	Marks the beginning of a comment
#	Pound Sign	Marks the beginning of a preprocessor directive
<>	Opening and closing brackets	Enclose a filename when used with the #include directive
( )	Opening and closing parentheses	Used in naming a function, as in int main ( )
{ }	Opening and closing braces	Encloses a group of statements, such as the contents of a function
" "	Opening and Closing quotation marks	Enclose 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

➲ Checkpoint

1. The following C++ program will not compile because the lines have been mixed up.

```
int main ( )

}

// A crazy mixed up program

return 0;

#include <iostream.h>

cout<< "In 1942 Columbus sailed the ocean blue.";

{
```

When the lines are properly arranged the program should display the following on the screen:

In 1942 Columbus sailed the ocean blue.

Rearrange the lines in the correct order. Test the program by entering it on the computer, compiling it, and running it.

3.2 The cout Object

In this section you will learn to write programs that produce output on the screen. The simplest type of screen output that a program can display is console output, which is merely plain text.

Program 3-2 is one way to write the same program.

PROGRAMS 3-2

```
// A simple C++
program #include
<iostream.h>

int main()
{
cout<< "Programming is " << "great
fun!"; return 0;

}
```

```
Program Output
Programming is great fun!
```

Program 3-3 shows another way to accomplish the same thing.

PROGRAMS 3-3

```
// A simple C++ program
#include <iostream.h>

int main()
{
cout<< "Programming is ";
cout<< "great fun!";
return 0;
}
```

```
Program Output
Programming is great
fun!
```

An important concept to understand about Program 3-3 is that, although the output is broken up into two programming statements, this program will display the message on a single line. Unless you specify otherwise, the information you send to cout is displayed in a continuous stream. Sometimes this can produce less-than desirable results. Program 3-4 is an example.

The layout of the actual output looks nothing like the arrangement of the strings in the source code.

First, notice there is no space displayed between the words "sellers" and "during", or between "June:" and "Computer". cout display messages exactly as they are sent. If spaces are to be displayed, they must appear in the strings.

PROGRAMS 3-4

```
// An unruly printing program
#include <iostream.h>

int main()
{
cout<< "The following items were top sellers";
cout<< "during the month of June:";
cout<< "Computer games";
cout<< "Coffee";
cout<< "Aspirin";
return 0;
}
```

Program Output

```
The following items were top sellersduring the month of June:Computer
gamesCoffeeAspirin
```

Second, even though the output is broken into five lines in the source code, it comes out as one long line of output. Because the output is too long to fit on one line on the screen, it wraps around to a second line when displayed. The reason the output comes out as one long line is because **cout** does not start a new line unless told to do so. There are two ways to instruct **cout** a stream manipulator called **endl** (which is pronounced "end-line" or "end-L") Program 3-5 is an example.

PROGRAMS 3-5

```
// A well-adjusted printing
program #include <iostream.h>

int main()
{
cout<< "The following items were top sellers"
<<endl; cout<< "during the month of June:" <<endl;
cout<< "Computer games" <<endl;
cout<< "Coffee" <<endl;
cout<< "Aspirin" <<endl;
return 0;
}
```

Program Output

```
The following items were top
sellers during the month of June:
Computer games
Coffee
Aspiri
n
```

Another way to cause cout to go to anew line is to insert an escape sequence in the string itself. An escape sequence starts with a backslash character (\), and is followed by one or more control characters. It allows you to control the way output is displayed by embedding commands within the string itself. Program 3-6 is an example.

The **newline escape sequence** is \n. 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.

PROGRAMS 3-6

```
// Yet another well-adjusted printing program
#include <iostream.h>

int main()
{
cout<< "The following items were top sellers\n";
cout<< "during the month of June:\n";
cout<< "Computer games\nCoffee";
cout<< "\nAspirin\n";
return 0;
}
```

Program Output

```
The following items were top sellers
during the month of June:
Computer games
Coffee
Aspirin
```

There are many escape sequences in C++. They give you the ability to exercise greater control over the way information is output by your program. Table 3-2 lists a few of them.

Escape Sequence	Name	Description
`\n`	Newline	Causes the cursor to go to the next line for subsequent printing.
`\t`	Horizontal Tab	Causes the cursor to skip over to the next tab stop.
`\a`	Alarm	Causes the computer to beep.
`\b`	Backspace	Causes the cursor to back up, or move left one position.
`\\`	Backslash	Causes a backslash to be printed.

Table 3-2 List of Escape Sequence

➲ Checkpoint

1. The following C++ program will not compile because the lines have been mixed up.

```
cout<< "Success\n";
cout<< " Success\n\n";
int main ( )
cout<< "Success";
}
//It's a mad, mad program
#include <iostream.h>
cout<<"Success \n";
{
return 0;
```

When the lines are properly arranged, the program should display the following on the screen:

Program Output

```
Success
Success Success

Success
```

2. Study the following program and show what it will print on the screen.

```
//The Works of Wolfgang
#include <iostream.h>
int main ( )
{
cout<< "The works of Wolfgang\ninclude the following";
cout<< "\nThe Turkish March" <<endl;
cout<< "and Symphony No. 40";
cout<< "in G minor." <<endl;
return 0;
}
```

3.3 Variables

Variables are an example of an identifier that is declared by the user. This name must be appropriate, easily understood and gives a clear meaning towards the value that it represents. This name is used to name the memory space that will store the variable's value.

Variables need to be declared before using. To declare the variable, we need to identify the name and type to represent a value.

When a variable has a certain data type, it means the variable can only represent data from that type only. For example, for an integer type variable, C++ assumes only whole numbers can be represented.

3.3.1 Rules to Name a Variable

In C++ language, there is a guideline in choosing a name for a variable which is:

a. Variable name can only contain letters, digits and underscores '_'.
b. Variable names cannot start with a digit (numbers).
c. Reserved words in C++ cannot be used as variable names.
d. Identifies that have a space or symbols (! @ # $ % & *) are not valid.
e. The size of the variable name cannot exceed 32 characters.

Next, there are some examples of variable names that are valid given in Table 3-3.

Valid Variable Names	Invalid Variable Names
tot3	2number
grand_Total	se-cond
_total	$price
GrandTotal	student name

grand_total	double
GRANDTOTAL	1*2

Table 3-3 Valid and Invalid Variable Names

It is not necessary for only lower case letters to be used to name a variable. Capital letters could also be used. Nevertheless, C++ is case sensitive, where lower case letters are different than capital letters. Observe how each of these variable name below are different in C.

total Total toTAL TOtal toTAl

The best way to name the variable is with a name that best represents the value that is store in it. Therefore, sometimes the variables name can be made of two words, like student_name or student name. These names cannot be separated by a space as it would be against the first rule of naming variables.

➲ Checkpoint

1. For each of the variable names below, state why it is not valid.
   a. float
   b. percentage%
   c. printf
   d. one&two
   e. integer
   f. student age
   g. value 10.5
   h. 1value
   i. answer 1
   j. X-2

3.4 C++ reserved Words

Reserved words are words that have a specific meaning in C++ and cannot be used for other reasons. All reserved words come in lower case letters.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Table 3-4 Reserved Words in C++

3.5 Variable Declaration

Variables can be used to keep input data and the calculation results or logic manipulations. Values that are kept by variables can change throughout program execution.

Syntax:

```
data_type variable_name;
```

Or

```
data_type variable_name = initial value;
```

Next are some examples of variable declarations given in Table 3-5:

Data Type	Variable Name	Values Stored
int	marks1, marks2, marks3;	98, 23, 77
double	averageMarks;	66.6
char	grade;	A
unsigned short integer	age;	36

Table 3-5Variable Declaration Examples

When declaring variables, the compiler will be notified of four items:

1. Variable name
2. Variable type
3. Size of cell of variable in the memory
4. Variable storage class

Different variables are used to keep different types of data. Therefore, in the variable declaration, it has to be mentioned what data type the variable will contain.

3.6 Literals

A variable is called a "variable" because its value may be changed. A literal, on the other hand, is a value that does not change during the program's execution. Program 3-7contains both literals and a variable.

PROGRAMS 3-7

```
// This program has literals and a variable.
#include <iostream.h>

int main()
{
int apples;

apples = 20;
cout<< "Today we sold " << apples << " bushels of apples.\n";
return 0;
}
```

Program Output
```
Today we sold 20 bushels of apples
```

Of course, the variable is **apples**. It is defined as an integer. Table 3-6 lists the literals found in the program

Literal	Type of Literal
`20`	Integer Literal
`"Today we sold "`	String Literal
`" bushels of apples.\n"`	String Literal
0	Integer Literal

Table 3-6 Literals and Type of Literals

➲ Checkpoint

1. Examine the following program.

```
//This program uses variables and literals
#include <iostream.h>
int main ( )
{
int little;
int big;

little = 2;
big = 2000;
cout<< "The little number is" << little <<endl;
cout<< "The big number is" << big <<endl;
return 0;
}
```

List all the variables and literals that appear in the program.

2. What will the following program display on the screen?

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

number = 712;
cout<< "The value is " << "number" <<endl;
return 0;
}
```

3.7 Integer Data types

There are many different types of data. Variables are classified according to their data type, which determines the kind of information that may be stored in them. **Integer variables can only hold whole numbers.**

Program Output

```
We have made a long journey of 4276 miles.
Our checking account balance is -20
About 18900 days ago Columbus stood on this spot.
```

```
int main()
{
int checking;
unsignedint miles;
long days;

checking = -20;
miles = 4276;
days = 189000;
cout<< "We have made a long journey of " << miles;
cout<< " miles.\n";
cout<< "Our checking account balance is " << checking;
cout<< "\nAbout " << days << " days ago Columbus ";
cout<< "stood on this spot.\n";
return 0;
}
```

3.8 The char Data Types

chardata type gets its name from the word "character". As its name suggests, it is **primarily forstoring characters.**

PROGRAMS 3-9

```
// This program uses character literals.
#include <iostream>

int main()
{
char letter;

letter = 'A';
cout<< letter <<endl;
letter = 'B';
cout<< letter <<endl;
return 0;
}
```

Program Output

```
A
B
```

3.9 Floating-Point Data Types

Floating-point data types are used to define variables that **can hold real numbers**.

3.10 Variable Assignments

We can assign values to a variable by using the following statement:

variable = value;

Consider the next example:

```
/*1*/    int number1, number2, multipliedNumber;
/*2*/    number1 = 10;
/*3*/    number2 = 25;
/*4*/    multipliedNumber = number1 * number2;
```

When statement **/*1*/**is declared, the memory cell assigned to each variable is:

?	**?**	**?**
number1	**number2**	**multipliedNumber**

Statement **/*2*/**assigns 10 to variable number1 and statement **/*3*/**assigns 25 to variable number2**.**

10	**25**	**?**
number1	**number2**	**multipliedNumber**

Nextstatement **/*4*/** is executed. When assignement statement multipliedNumber = number1 * number2; is executed, the right side of '=' will be calculated first, that is by doing multiplication on values kept by variables number1 and number2. The result is 250 and this value will be assignment to the variable multipliedNumber that is on the left of "=". Next is the result of the execution of statement **/*4*/**.

10	**25**	**250**
number1	**number2**	**multipliedNumber**

➲ Checkpoint

1. Explain why the declaration is not valid:

	Declaration	Explanation
i.	`int;`	
ii.	`float a;`	
iii.	`x, y, z;`	

2. Correct the declaration below:

	Declaration	Correction
i.	`int x = 'A';`	
ii.	`float y = 1;`	

3. Write the declaration for these values:

   i. Student marks : 78

   ii. Weight of boxes of books : 4.5kg

   iii. Temperature at a country in the west during winter: -4°C.

   iv. Student grade is A.

3.11 Interactivity

So far you have written programs with built-in data. Without giving the user an opportunity to enter his or her own data, you have initialized the variables with the necessary starting values. These types of programs are limited to performing their task with only a single set of starting data. If you decide to change the initial value of any variable, the program must be modified and recompiled.

In reality, most programs ask for values that will be assigned to variables. This means the program does not have to be modified if the user wants to run it several times with different sets of data. For example, a program that calculates payroll for a small business might ask the user to enter the name of the employee, the hours worked, and the hourly pay rate. When the paycheck for that employee has been printed, the program could start over again and ask for the name, hours worked, and hourly pay rate of the next employee.

3.11.1 The cin object

Just as **cout** is C++’s standard output object, **cin** is the standard input object. It reads input from the console (or keyboard) as shown in Program 3-10.

PROGRAMS 3-10

```
// This program asks the user to enter the length and width of
// a rectangle. It calculates the rectangle's area and displays
// the value on the screen.
#include <iostream.h>

int main()
{
int length, width, area;

cout<< "This program calculates the area of a ";
cout<< "rectangle.\n";
cout<< "What is the length of the rectangle? ";
cin>> length;
cout<< "What is the width of the rectangle? ";
cin>> width;
area = length * width;
cout<< "The area of the rectangle is " << area << ".\n";
return 0;
}
```

Program Output

```
This program calculates the area of a rectangle.
What is the length of the rectangle? 10 [Enter]
What is the width of the rectangle? 20 [Enter]
The area of the rectangle is 200.
```

Instead of calculating the area of one rectangle, this program can be used to get the area of any rectangle. The values that are stored in the length and width variables are entered by the user when the program is running. Look at this line:

```
cout<< "What is the length of the rectangle? ";
cin>> length;
```

In that line, the cout object is used to display the question **"What is the length of the rectangle? ";** This question is known as a prompt, and it tells the user what data he or she should enter. Your program should always display a **prompt** before it uses **cin** to read input. This way, the user will know that he or she must type a value at the keyboard.

The line uses **cin** object to read a value from keyboard. The **>>**symbol is the **stream extraction operator**. It gets characters from the stream object on its left and stores them in the variable whose name appears on its right. In this line, characters are taken from the **cin** object (which gets from the keyboard) and are stored in the length variable.

Gathering input from the user is normally a two-step process:

1. Use the **`cout`** object to display a prompt on the screen.
2. Use the **`cin`** object to read a value from the keyboard.

The prompt should ask the user a question, or tell the user to enter a specific value. For example, the code we just examined from Program 3-10 displays the following prompt:

```
cout<< "What is the length of the rectangle? ";
```

When the user sees the prompt, he or she knows to enter the rectangle’s length. After the prompt is displayed, the program uses the cin object to read a value from the keyboard and store the value in the length variable.

NOTE : You must include the `iostream` file in any program that uses `cin`.

3.11.2 Entering Multiple Values

The **cin** object may be used to gather multiple values at once. Look at Program 3-11 which is a modified version of program 3-10. Line **cin>> length >> width;** waits for the user to enter two values. The first is assigned to **length** and the second to **width**.

In the example output, the user entered **10** and **20**, so 10 is stored in length and 20 is stored in width.

Notice, the user separate the numbers by spaces as they entered. This is how **cin** knows where each number begins and ends. It doesn’t matter how many spaces are entered between the individual numbers. For example, the user could have entered

10   20

NOTE :The **[enter]** key is pressed after the last number is entered.

cin will also read multiple values of different data types.

PROGRAMS 3-11

```
// This program asks the user to enter the length and width of
// a rectangle. It calculates the rectangle's area and displays
// the value on the screen.
#include <iostream.h>

int main()
{
int length, width, area;

cout<< "This program calculates the area of a ";
cout<< "rectangle.\n";
cout<< "Enter the length and width of the rectangle ";
cout<< "separated by a space.\n";
cin>> length >> width;   ←
area = length * width;
cout<< "The area of the rectangle is " << area <<endl;
return 0;
}
```

➲ Checkpoint

1. A program has the following variable definitions.
   long miles;
   int feet;
   float inches;

   Write one cin statement that reads a value into each of these variables.

2. The following program will run, but the user will have difficulty understanding what to do. How would you improve the program?

```
//This program multiplies two numbers and displays the result
#include <iostream.h>
int main ( )
{
double first, second, product;

cin>> first >> second;
product = first * second;
cout<< product;
return 0;
}
```

3. Complete the following program skeleton so it asks for the user's weight (in pounds) and displays the equivalent weight in kilograms.

```
//To change weight from pound to kilogram
#include <iostream.h>
int main ( )
{
 double pounds, kilograms;

   // Write code here that prompts the user to enter his or her weight and reads
  the input into the pounds variable

 // The following line does the conversion.

 kilograms = pounds / 2.2;

   //  Write code here that displays the user's weight in kilograms.

return 0;
```

4. Write C++ statement(s) that accomplish the following:

   i. Declare **int** variables **x**, **y**, and **sum**.
   ii. Prompt the user to input two integer numbers, and save the number in variable named **x** and **y**.
   iii. Adds the two integer numbers, **x** and **y**, and save it in a variable named **sum**.
   iv. Print the **sum** of the numbers

5. Based on scenario below (i-v), write the C++ statement for the following program.

   i. Read an integer value and store it to a variable named **Lebar**.
   ii. Print an output : **Your Salary for this month is : RM2567.00**. The salary value is stored in the SALARY variable.

Instructions: Find the true and false statement below. Then, rewrite the remaining false statements so they are true.

1. The line starts with a # called preprocessor directive.
2. The iostream file contains code that allows a C++ program to display output on the screen and read input from the keyboard.
3. \t is a newline to go to the next line for subsequent printing.
4. goto is a reserved word.

Quiz Yourself Online: Do Self Test in the E-Learning

ACTIVITY

Examine the following program

```
//This program uses variables and literals
#include <iostream.h>
int main ( )
{
int little;
int big;

little = 2;
big = 2000;
cout<< "The little number is" << little <<endl;
cout<< "The big number is" << big <<endl;
return 0;
}
```

List all the variables and literals that appear in the program.

KEY TERM

Preprocessor directive	**literals**
Function	**Variable assignments**
String literal	**Interactivity**
Variables	
Reserved words	

SUMMARY.

- C++ language is a high level language.
- Learned writing simple C++ program, how to compile and execute those programs.

REFERENCEES

Tony Gaddis, Starting Out with C++ - Early Objects (10th edition) ,Pearson, 2020.

TOPIC 4

Mathematical Expressions, Relational Operators and Logical Operators.

LEARNING OUTCOMES

By the end of topic, you should be able to:

1. Write the arithmetic expressions using C++ syntax
2. Use the operators for arithmetic, Relational and Logic.
3. Evaluate the expressions to be used in assignment statement.

4.1 Operators

Symbols like +, -, *, /, <, >, != are known as operators. In C++ language there are many types of operators. There are arithmetic operators, relational operators, logic operators, increment and decrement operators and pointers. This chapter will explain in detail about these operators.

Data stored in memory can be modified using operators. The operators are divided into three distinct types, refer to Table 4.1.

Operators	Symbol
Arithmetic	+, -, *, /, %
Relational	<, <=, >, >=, ==, !=
Logic	&&, ||, !

Table 4.1 Operators and Symbol

Operators are use to connect operands into expressions. These expressions, when ended with a semicolon (;) becomes a statement.

Expression: basic_pay + allowances – expenses

Statement: net_pay = basic_pay + allowances – expenses;

4.2 Arithmetic Operators

Arithmetic operators are fixed to one variable only. A list of unary operators are given in Table 8.2.

Operator	Function
+	Positive Operator
-	Negative Operator
++	Increment Operator
--	Decrement Operator
!	Not Operator
&	Address Operator
*	Value of Address Operator

Table 4.2 Unary Operators Symbols

Increment/Decrement Operator

If the increment/decrement operator is fixed at the end (postfix) of a variable, the original value of the variable is used, and then only is the value updated (incremented or decremented).

If the increment/decrement operator is fixed at the beginning (prefix) of a variable, the original value of the valuable will be updated, (incremented or decremented) and the updated value will be used.

PROGRAMS 4-1

```
// This program demonstrates the ++ and -- operators.
#include <iostream.h>

int main()
{
intnum = 4;   // num starts out with 4.

   // Display the value in num.
cout<< "The variable num is " <<num<<endl;
cout<< "I will now increment num.\n\n";

   // Use postfix ++ to increment num.
num++;
cout<< "Now the variable num is " <<num<<endl;
cout<< "I will increment num again.\n\n";
```

```
// Use prefix ++ to increment num.
   ++num;
cout<< "Now the variable num is " <<num<<endl;
cout<< "I will now decrement num.\n\n";

   // Use postfix -- to decrement num.
num--;
cout<< "Now the variable num is " <<num<<endl;
cout<< "I will decrement num again.\n\n";

   // Use prefix -- to increment num.
   --num;
cout<< "Now the variable num is " <<num<<endl;
return 0;
}
```

Program Output
```
The variable num is 4
I will now increment num.

Now the variable num is 5
I will increment num again.

Now the variable num is 6
I will increment num again.

Now the variable num is 5
I will increment num again.

The variable num is 4
```

NOTE :The expression num++ is pronounced “num plus plus”, and num—is pronounced “num minus minus”.

➲ Checkpoint

1. What is the output of the following program?

```
// This program demonstrates the prefix and postfix
// modes of the increment and decrement operators.
#include <iostream.h>

int main()
{
  int num = 4;

  cout << num << endl;   // Displays 4
  cout << num++ << endl; // Displays 4, then adds 1 to num
  cout << num << endl;  // Displays 5
  cout << ++num << endl; // Adds 1 to num, then displays 6
  cout << endl;         // Displays a blank line

  cout << num << endl;   // Displays 6
  cout << num-- << endl; // Displays 6, then subtracts 1 from num
  cout << num << endl;  // Displays 5
  cout << --num << endl; // Subtracts 1 from num, then displays 4

  return 0;
}
```

Program Output

4.3 Binary Arithmetic Operators

Operands that are fixed with arithmetic operators must be of numeric types. Therefore, the operands must be of type **int, float, double,** or even **char**. Operators is put in between two operands. The modulus operator can only be used with the **int** type data only.

Operator	Function
+	Positive Operator
-	Negative Operator
++	Increment Operator
--	Decrement Operator
!	Not Operator
&	Address Operator
*	Value of Address Operator

Table 4.3 Binary Operators Symbols

PROGRAMS 4-2

```
#include <iostream.h>

int main()
{
intx, y, z;
   x = 10, y = 13;

   z = x + y;
      cout<< z <<endl;
   y = y - x;
      cout<<y<<endl;
   x = y * z;
      cout<<x<<endl;
   z = x / 20;
      cout<< z <<endl;
   y = z % x;
      cout<<y<<endl;
return 0;
}
```

```
Program Output
23
3
69
3
3
```

Integer Division

Integer division happens when both the operands are of integer type. Operands can be a constant or a variable. If the division is not a round number (that is it has a decimal value), therefore the remainder of the division is ignored. The result is truncated to an integer value.

Real Division

Real division is done when one or both of the operands have a floating point value. Division operation done is the same as normal arithmetic division. The result of the division is a floating point value.

Value a	Value b	Integer Division	Real Division
10	3	10 / 3 = 3	10.0 / 3.0 = 3.333333
5	10	5 / 10 = 0	5.0 / 10.0 = 0.500000
6	6	6 / 6 = 1	6.0 / 6.0 = 1.000000

Table 4.4 Division Calculations

Operation	Division Result	Remainder
5 % 3	1	2
7 % 2	3	1
12 % 2	6	0

Table 4.5 Modulos Operations

If all of the operands are **int**, the result is **int**, but if any one of the operands is float, it will result in a float value.

4.4 Arithmetic Expressions

Arithmetic expressions can be written by combining one or more arithmetic operations. The precedence levels of C++ arithmetic expressions would determine the arithmetic operations sequence in solving the expressions. If you do not want the arithmetic operations evaluated using the precedence levels, use the brackets as the expressions in the brackets will be evaluated first. Table 4-5 shows the precedence levels.

Precedence Level	**Operation**
High	( ), *, /, %
Low	+ , -

Table 4-6 Precedence Levels.

How precedence levels work?

$$\mathbf{1 \quad + \quad \underbrace{2 \quad * \quad 3}}$$

$$\mathbf{1 \quad + \quad 6 \quad = \; 7}$$

Therefore **: 1 + 2 * 3 is 7**

➲ Checkpoint

1. Complete the table below by writing the value of each expression in the “Value” column.

Expression	Value
6+3*5	
12/2-4	
9+14*2-6	
5+19%3-1	
(6+2)*3	
14/(11 – 4)	
9+12*(8 – 3)	
(6+17)%2-1	
(9 – 3) * (6 + 9) / 3	

4.5 Assignment Statement

Assignment of data to the variable can be written as:

variable = expression;

For the statement above, the left part can only be a variable. On the right, it can be made up of a combination of variables and constants.

Combined Assignment Operators

Quite often, programs have assignment statements of the following form:

number = number + 1 ;

The expression on the right side of the assignment operator gives the value of number plus 1. The result is then assigned to number, replacing the value that was previously stored there. Effectively, this statement adds 1 to number. In a similar fashion, the following statement subtracts 5 from number.

number = number – 5;

Table 4-7 shows other examples of statements written this way.

Assume x = 6

Statement	What It Does	Value of x after the statement
`x = x + 4;`	Adds 4 to x	10
`x = x - 3;`	Subtracts 3 from x	3
`x = x * 10;`	Multiplies x by 10	60
`x = x / 4;`	Divides x by 2	3
`x = x % 4;`	Makes x the remainder of x/4	2

These types of operations are very common in programming. For convenience, C++ offers a special set of operators designed specifically for these jobs. Table 4-8 shows the **combined assignment operators**, also known as **compound operators**, and **arithmetic assignment operators**.

Operator	Example Usage	Equivalent to
`+=`	x += 5;	`x = x + 5;`
`-=`	y -= 2	`y = y - 2;`
`*=`	z *= 10;	`z = z * 2;`
`/=`	a /= b;	`a = a / b;`
`%=`	c %= 3;	`c = c % 3;`

PROGRAMS 4-3

```
// This program tracks the inventory of three widget stores
// that opened at the same time. Each store started with the
// same number of widgets in inventory. By subtracting the
// number of widgets each store has sold from its inventory,
// the current inventory can be calculated.
#include <iostream.h>

int main()
{
intbegInv,     // Begining inventory for all stores
sold,       // Number of widgets sold
       store1,  //  Store  1's  inventory
       store2,  //  Store  2's  inventory
       store3;  // Store 3's inventory

   // Get the beginning inventory for all the stores.
cout<< "One week ago, 3 new widget stores opened\n";
cout<< "at the same time with the same beginning\n";
cout<< "inventory. What was the beginning inventory? ";
cin>>begInv;

   // Set each store's inventory.
   store1 = store2 = store3 = begInv;

   // Get the number of widgets sold at store 1.
cout<< "How many widgets has store 1 sold? ";
cin>> sold;
   store1 -= sold; // Adjust store 1's inventory.

   // Get the number of widgets sold at store 2.
cout<< "How many widgets has store 2 sold? ";
cin>> sold;
   store2 -= sold; // Adjust store 2's inventory.

   // Get the number of widgets sold at store 3.
cout<< "How many widgets has store 3 sold? ";
cin>> sold;
   store3 -= sold; // Adjust store 3's inventory.
```

```
// Display each store's current inventory.
cout<< "\nThe current inventory of each store:\n";
cout<< "Store 1: " << store1 <<endl;
cout<< "Store 2: " << store2 <<endl;
cout<< "Store 3: " << store3 <<endl;
return 0;
}
```

```
Program Output with Example Input Shown in Bold
One week ago, 3 new widget stores opened
at the same time with the same beginning
inventory, What was the beginning inventory? 100 [Enter]
How many widget has store 1 sold? 25 [Enter]
How many widget has store 2 sold? 15 [Enter]
How many widget has store 3 sold? 45 [Enter]

The current inventory of each store:
Store 1: 75
Store 2: 85
Store 3: 55
```

➲ Checkpoint

1. Write a multiple assignment statement that assigns **0** to the variables **total, subtotal, tax,** and **shipping**.

2. Write statements using combined assignment operators to perform the following:

   a. Add 6 to x.

   b. Subtract 4 from amount.

   c. Multiply y by 4.

   d. Divide total by 27.

   e. Store in x the remainder of x divided by 7.

   f. Add y * 5 to x.

   g. Subtract discount times 4 from total.

   h. Multiply increase by salesRep times 5.

   i. Divide profit by shares minus 1000.

3. What will the following program display?

```
#include <iostream.h>
int main ( )
{
intunus, duo, tres;
unus = duo = tres = 5;
unus += 4;
duo *= 2;
tres -= 4;
unus /= 3;
duo += tres;
cout<<unus<<endl;
cout<< duo <<endl;
cout<<tres<<endl;
return 0;
}
```

4. Complete the table below by writing the value of each expression in the “Value” column.

Expression	Value
i. (9 – 3) * (6 + 9)	
ii. 5 + 19 % 3 - 1	
iii. (6 + 17) % 2 - 1	

5. Determine the value of the following expressions:

Expression	Value
i. 3 * 4 / 6 + 6	
ii. 10 - (1 + 7 * 3)	
iii. 3.0 + 4.0 * 6.0	
iv. 50 % 10	

Instructions: Find the true and false statement below. Then, rewrite the remaining false statements so they are true.

1. An operator is a symbol used for performing operations on operands.
2. a = x + y; In the above statement, x and y are operands while + is an addition operator.
3. % modulus operator means returns remainder after division.
4. ++increment operator means decrease an integer value by 1.

Quiz Yourself Online: Do Self Test in the E-Learning

ACTIVITY

Write a multiple assignment statement that assigns **0** to the variables **total, subtotal, tax,** and **shipping**.

KEY TERM

Operators

Integer Division

Real division

Variables

Precedence level

Assignment statements

Arithmetic

Relational

Logic

SUMMARY.

- Special arithmetic operators are normally used to write programs.
- C++ arithmetic expressions are a combination of one or more arithmetic operations
- Other than arithmetic operators, C++ also provide operators that are use to compare data that is known as relational operators.

REFERENCEES

Tony Gaddis, Starting Out with C++ - Early Objects (10th edition) ,Pearson, 2020.

TOPIC 5

Making Decisions

LEARNING OUTCOMES

By the end of topic, you should be able to:

1. Write C++ symtax for if selection, if/else selection adn nested if selection structures.
2. Differentiate between if and switch statements
3. Compare between the if and switch statements based on the probelm given.

5.1 Relational Operators

Relational operators allow you to compare numeric and char values and determine whether one is greater than, less than, equal to, or not equal to another. Numeric data is compared in C++ by using relational operators. Each relational operator determines whether 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. Table 5-1 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

Table 5-1 Relational Operators

➲ **Checkpoint**

1. Assuming x is 5, y is 6, and z is 8, indicate by circling the T or F whether each of the following relational expressions is true or false:

a.	x == 5	**T**	**F**
b.	7 <= (x+2)	**T**	**F**
c.	z < 4	**T**	**F**
d.	(2 + x) != y	**T**	**F**
e.	z != 4	**T**	**F**
f.	x >= 9	**T**	**F**
g.	x <= (y * 2)	**T**	**F**

5.2 Logic Operator

The logic operator is used to combine a few expressions that have relational operators. It is useful to do complex comparisons to make decisions.

Operator	Function	Explanation
&&	And	Is **true** if and only if both expressions are **true**
| |	Or	Is **true** if one or both the expressions are **true**
!	Not	Is **true** if the expression is **false** and vice versa

Table 5-2 Logical Operators

5.2.1 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.";
```

In the statement above the two relational expressions are combined into a single expression. The **cout** statement will only be executed if temperature is less than 20 AND minutes are greater than 12. If either relational test is false, the entire expression is false and the **cout** statement is not executed.

Expression	Value of Expression
true && true	false (0)
false && true	false (0)
false && false	false (0)
true && true	true (1)

Table 5-3 Truth table for the && operator

5.2.2 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 is 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.

Expression	Value of Expression
true || true	true (1)
false || true	true (1)
false || false	false (0)
true || true	true (1)

Table 5-4 Truth table for the || operator

5.2.3 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 false, it returns true. Here is an if statement using **!** 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?"

Expression	Value of Expression
! true	false (0)
! false	true (1)

Table 5-4 Truth table for the ! operator

➲ Checkpoint

1. The following truth table shows various combinations of the value true and false connected by a logical operator. Complete the table by indicating if the result of such a combination is TRUE or FALSE.

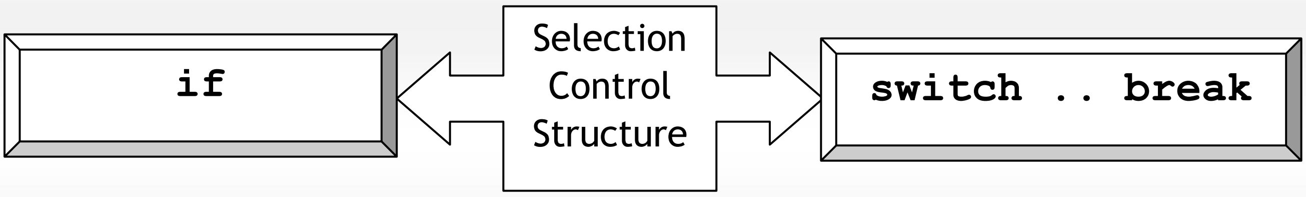
Logical Expression	Result (true or false)
true && false	
true && true	
false && true	
false && false	
true || false	
true || true	
false || true	
false || false	
! true	
! false	

2. Assume the variables a =2, b=4, and c=6. Indicate by circling the T or F if each of the following conditions is true or false:

a == 4 || b > 2	**T**	**F**
6 <= c && a > 3	**T**	**F**
1 != b && c != 3	**T**	**F**
a >= -1 || a <= b	**T**	**F**
! (a > 2)	**T**	**F**

5.3 Introduction to Selection Control Structure

In topic 3, you have learnt to solve problems using the sequence, selection and repetition control structures. C++ language has a few selection structures that control the flow which is if, switch and break.



5.4 If statement

The **if statement** is used to represent the selection structure. The selection structure allows us to choose and execute only one of the many choices available based on a certain condition. Selection structure is divided into 3 main types, which are if selection, if/else selection and nested if selection.

if | **if/else** | **Nested if**

5.4.1 If statement

Syntax for the if selection structure is written as:

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

The if statement above will only be executed when the value of the expression is true. If the statement is false, the statement will not be executed. Expression can only be made up of relational or logical expressions that would have the value of true or false.

```
// This program averages three test scores
#include <iostream.h>
#include <iomanip.h>

int main()
{
   int score1, score2, score3;  // To hold three test scores
   double average;              // TO hold the average score

   // Get the three test scores.
   cout << "Enter 3 test scores and I will average them: ";
   cin >> score1 >> score2 >> score3;

   // Calculate and display the average score.
   average = (score1 + score2 + score3) / 3.0;
   cout << fixed << showpoint << setprecision(1);
   cout << "Your average is " << average << endl;

   // If the average is greater than 95, congratulate the user.
   if (average > 95)
      cout << "Congratulations! That's a high 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 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 high score!
```

Expanding the **if** Statement

The **if** statement can conditionally execute a block of statements enclosed in braces.
What if you want an if statement to conditionally execute a group of statements, not just one line?

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

Program 5-2

```
// This program averages 3 test scores.
// It demonstrates an if statement executing a block of statements.
#include <iostream.h>
#include <iomanip.h>

int main()
{
   int score1, score2, score3; // To hold three test scores
   double average;              // TO hold the average score

   // Get the three test scores.
   cout << "Enter 3 test scores and I will average them: ";
   cin >> score1 >> score2 >> score3;

   // Calculate and display the average score.
   average = (score1 + score2 + score3) / 3.0;
   cout << fixed << showpoint << setprecision(1);
   cout << "Your average is " << average << endl;

   // If the average is greater than 95, congratulate the user.
   if (average > 95)
   {
      cout << "Congratulations!\n";
      cout << "That's a high score.\n";
      cout << "You deserve a pat on the back!\n";
   }
   return 0;
}
```

Program Output with 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 high score.
You deserve a pat on the back!
```

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
```

5.4.2 The if/else Statement

The if/else statement will execute one group of statements if the expression is true, or another group of statements if the expression is false. The if/else statement is an expansion of the if statement. Here is its format:

```
if (expression)
        statement or block;
else
        statement or block;
```

As with the if statement, an expression is evaluated. If the expression is true, a statement or block is executed. If the expression is false, however, a separate group of statements is executed. Program 5-3 uses the if/else statement along with the modulus operator to determine if a number is odd or even.

Program 5-3

```
// 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.h>

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.
```

To test more than one condition, an if statement can be nested inside another if statement.

```
// This program demonstrates the nested if statement.
#include <iostream.h>

int main()
{
   char employed,     // Currently employed, Y or N
        recentGrad;   // Recent graduate, Y or N

   // Is the user employed and a recent graduate?
   cout << "Answer the following questions\n";
   cout << "with either Y for Yes or ";
   cout << "N for No.\n";
   cout << "Are you employed? ";
   cin >> employed;
   cout << "Have you graduated from college ";
   cout << "in the past two years? ";
   cin >> recentGrad;

   // Determine the user's loan qualifications.
   if (employed == 'Y')
   {
      if (recentGrad == 'Y') //Nested if
      {
         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 Different 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]
```

Program 5-5

```
// This program demonstrates the nested if statement.
#include <iostream.h>

int main()
{
   char employed,     // Currently employed, Y or N
        recentGrad;   // Recent graduate, Y or N

   // Is the user employed and a recent graduate?
   cout << "Answer the following questions\n";
   cout << "with either Y for Yes or ";
   cout << "N for No.\n";
   cout << "Are you employed? ";
   cin >> employed;
   cout << "Have you graduated from college ";
   cout << "in the past two years? ";
   cin >> recentGrad;

   // Determine the user's loan qualifications.
   if (employed == 'Y')
   {
      if (recentGrad == 'Y') // Nested if
      {
         cout << "You qualify for the special ";
         cout << "interest rate.\n";
      }
      else // Not a recent grad, but employed
      {
         cout << "You must have graduated from ";
         cout << "college in the past two\n";
         cout << "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 Different 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 Different 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.
```

5.5 The if/else if Statement

The if/else if statement tests a series of conditions. It is often simpler to test a series of conditions with the if/else if statement than with a set of nested if/else statements.

```
if (expression_1)
{
        statement;
        statement;
        etc.
}
else if (expression_2)
{
        statement;
        statement;
        etc.
}
Insert as many else if clauses as necessary
else
{
        statement;
        statement;
        etc.
}
```

If `expression_1` is true these statements are executed, and the rest of the structure is ignored.

Otherwise, if `expression_2` is true these statements are executed, and the rest of the structure is ignored.

These statements are executed if none of the expressions above are true

Program 5-6

```
// 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;  // To hold a numeric test score
   char grade;     // To hold a letter grade

   // Get the numeric test score.
   cout << "Enter your numeric test score and I will\n";
   cout << "tell you the letter grade you earned: ";
   cin >> testScore;

   // Determine the letter grade.
   if (testScore < 60)
      cout << "Your grade is F.\n";
   else if (testScore < 70)
      cout << "Your grade is D.\n";
   else if (testScore < 80)
      cout << "Your grade is C.\n";
   else if (testScore < 90)
      cout << "Your grade is B.\n";
   else
      cout << "Your grade is A.\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: 78 [Enter]
Your grade is C.
```

Program Output with Different Example Input Shown in Bold

```
Enter your numeric test score and I will
tell you the letter grade you earned: 84 [Enter]
Your grade is B.
```

5.6 Menus

You can use nested if/else statements or the if/else if statement to create menu-driven programs. A menu-driven program allows the user to determine the course of action by selecting it from a list of actions.

Program 5-7

```
// This program displays a menu and asks the user to make a
// selection. An if/else if statement determines which item
// the user has chosen.
#include <iostream.h>
#include <iomanip.h>

int main()
{
   int choice;        // Menu choice
   int months;        // Number of months
   double charges;    // Monthly charges

   // Constants for membership rates
   const double ADULT = 40.0;
   const double SENIOR = 30.0;
   const double CHILD = 20.0;

   // Display the menu and get a choice.
   cout << "\t\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;

   // Set the numeric ouput formatting.
   cout << fixed << showpoint << setprecision(2);

   // Respond to the user's menu selection.
   if (choice == 1)
   {
      cout << "For how many months? ";
      cin >> months;
      charges = months * ADULT;
      cout << "The total charges are $" << charges << endl;
   }
else if (choice == 2)
   {
      cout << "For how many months? ";
      cin >> months;
      charges = months * CHILD;
      cout << "The total charges are $" << charges << endl;
   }
   else if (choice == 3)
   {
      cout << "For how many months? ";
      cin >> months;
      charges = months * SENIOR;
      cout << "The total charges are $" << charges << endl;
   }
   else if (choice == 4)
   {
       cout << "Program ending.\n";
   }
   else
   {
      cout << "The valid choices are 1 through 4. Run the\n";
      cout << "program again and select one of those.\n";
   }
   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: 3 [Enter]
For how many months? 6[Enter]
The total charges are $180.00
```

➲ Checkpoint

1. What is the output of the following program?

```
#include <iostream.h>
int main ( )
{
        int funny = 7, serious = 15;

        funny = serious % 2;
        if (funny != 1)
        {
          funny = 0;
          serious = 0;
        }
        else if (funny == 2)
        {
          funny = 10;
          serious = 10;
        }
        else
        {
          funny = 1;
          serious = 1;
        }
        cout << funny << " " << serious << endl;
        return 0;
}
```

2. What is the output of the following program fragment:

```
pendapatan_net = 3000;
if  (pendapatan_net> 10000.00)
     city_tax = 0.1 * pendapatan_net;
else
     city_tax = 0.0;
cout<< "city tax = RM " <<city_tax;
```

3. The following program is used in a bookstore to determine how many discount coupons a customer gets. Complete the table that appears after the program.

```
#include <iostream.h>
int main ( )
{
        int numBooks, numCoupons;

        cout << "How many books are being purchased? ";
        cin >> numBooks;
        if (numBooks < 1)
                numCoupons = 0;
        else if (numBooks < 3)
                numCoupons = 1;
        else if (numBooks < 5)
                numCoupons = 2;
        else
                numCoupons = 3;
        cout << "The number of coupons to give is " << numCoupons << endl;
        return 0;
}
```

If the customer purchases this many books	This many coupons are given
1	
2	
3	
4	
5	
10	

4. Write a complete C++ program that calculates the user's body mass index (BMI) and categorizes it as underweight, normal, overweight, or obese, based on the following table from the United State Centers for Disease Control:

BMI	Weight Status
Below 18.5	Underweight
18.5 – 24.9	Normal
25.0 – 29.9	Overweight
30.0 and above	Obese

To calculate BMI based on weight in pounds *(wt_lb)* and height in inches *(ht_in),* use this formula :

$$\frac{703 \times wt\_lb}{ht\_in^2}$$

5.7 The switch Statement

The switch statement lets the value of a variable or expression determines where the program will branch. 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 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 (IntegerExpression)
{
  case ConstantExpression:
     //place one or more
     //statement here

  case ConstantExpression:
     //place one or more
     //statement here

  //case statements may be repeated as many
  //times as necessary

  default :
     //place one or more
     //statements here
}
```

The expression following the word `case` must be an integer or constant. It cannot be a variable, and it cannot be expressions such as `x < 22` or `n == 50`.

Program 5-8 shows how simple switch statement works.

Program 5-8

```
// The switch statement in this program tells the user something
// he or she already knows: what they just 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 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 literals 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.

```
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";
}
```

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.

Program 5-9 is a modification of Program 5-8, without the **break** statements.

Program 5-9

```
// The switch statement in this program tells the user something
// he or she already knows: what they just 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";
      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 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. Program 5-10 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.

Program 5-10

```
// This program is carefully constructed to use the "fallthrough"
// feature of the switch statement.
#include <iostream.h>

int main()
{
   int modelNum;  // Model number

   // Get a model number from the user.
   cout << "Our TVs come in three models:\n";
   cout << "The 100, 200, and 300. Which do you want? ";
   cin >> modelNum;

   // Display the model's features.
   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 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 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 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, Program 5-11 asks the user to select a grade of pet food. The available choices are A, B, and C. The **switch** statement will recognize either upper or lowercase letters.

Program 5-11

```
// 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;

   // Get the desired grade of feed.
   cout << "Our pet food is available in three grades:\n";
   cout << "A, B, and C. Which do you want pricing for? ";
   cin >> feedGrade;

   // Display the price.
   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 pet 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 Example Input Shown in Bold

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

➲ Checkpoint

1. Explain why you cannot convert the following **`if/else if`** statement into a switch statement.

```
if (temp == 100)
      x = 0;
else if (population > 1000)
      x = 1;
else if (rate < .1)
      x = -1;
```

2. What is wrong with the following switch statement?

```
switch (temp)
{
  case temp < 0 : cout << "Temp is negative. \n";
                       break;
  case temp == 0:  cout << "Temp is zero.\n";
                       break;
  case temp > 0 :     cout << "Temp is positive. \n";
                       break;
}
```

3. What will the following program display?

```
int funny = 7, serious =5;
funny = serious * 2;
switch (funny)
{
  case  0 : cout << "That is funny.\n";
           break;
  case 30 : cout <<"That is serious.\n";
           break;
  case 32 : cout <<"That is seriously funny.\n";
           break;
  default : cout << funny << endl;
}
```

4. Complete the following program skeleton by writing a switch statement that displays “one” if the user has entered 1, “two” if the user has entered 2, and “three” if the user has entered 3. If the number other than 1,2, or 3 is entered, the program should display an error message.

```
#include <iostream.h>

int main ( )
{
  int userNum;
  cout << “Enter one of the numbers 1,2, or 3: “;
  cin >> userNum;
  //
  //  Write the switch statement here.
  //
  return 0;
}
```

5. Rewrite the following program. Use a switch statement instead of the **`if/else if`** statement.

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

  cout <<”Which formula do you want to see ? \n\n”;
  cout <<”1.  Area of a circle \n”;
  cout <<”2. Area of a rectangle \n”;
  cout <<”3. Area of a cylinder \n”;
  cout <<”4. None of them! \n”;
  cin >> selection;
  if (selection == 1)
     cout << “Pi times radius squared \n”;
  else if (selection == 2)
     cout << “Length times width \n”;
  else if (selection == 3)
    cout << “Pi times radius squared times height\n”;
  else if (selction == 4)
    cout << “Well okay then, good bye! \n”;
  else
    cout << “Not good with numbers, eh? \n”;
  return 0;
}
```

Instructions: Find the true and false statement below. Then, rewrite the remaining false statements so they are true.

1. Three basic control structures are sequence, selection, and pseudocode.
2. Programmers must convert an assembly language program into machine language before the computer can execute, or run, the program.
3. Programmers use a sequence control structure to show program modules graphically.
4. A sequence control structure tells the program which action to take, based on a certain condition.

Quiz Yourself Online: Do Self Test in the E-Learning

ACTIVITY

What will the following program display?

```
int funny = 7, serious =5;

funny = serious * 2;

switch (funny)
{
  case  0 : cout << "That is funny.\n";
               break;
  case 30 : cout <<"That is serious.\n";
               break;
  case 32 : cout <<"That is seriously funny.\n";
               break;
  default : cout << funny << endl;
}
```

KEY TERM

if	**Menu**
Switch .. break	**assembler**
If/else	**Selection**
Nested if	
If/else if	

SUMMARY

- There are three forms of the if statement if, if-else and nested if-else
- Use the switch and break statements
- The differences between these control structures are also given.

REFERENCEES

Tony Gaddis, Starting Out with C++ - Early Objects (10th edition) ,Pearson, 2020.

TOPIC 6

Introduction to Computer System

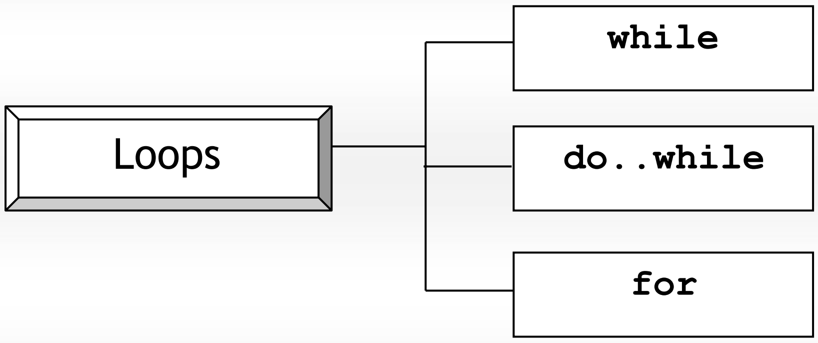
LEARNING OUTCOMES

By the end of topic, you should be able to:

1. Implement C++ program using repetition structure such as for, while and do-while.
2. Differentiate between for, while and do-while statements

6.1 Introduction to Loop

A loop is part of a program that repeats. It 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.



6.2 The while Loop

The while loop has two important parts:

1. An expression that is tested for a true of false value, and
2. A statement or block that is repeated as long as the expression is true.

Here is the general format of the while loop:

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

In the general format, expression is any expression that can be evaluated as true or false, and statement is any valid C++ statement. The first line shown in the format is sometimes called the loop header. It consists of the key word while followed by an expression enclosed in parentheses.

Here's how the loop works: the expression is tested, and if it is true, the statement is executed. This cycle repeats until the expression is false.

The statement that is repeated is known as the body of the loop. It is also considered a conditionally executed statement, because it is executed only under the condition that the expression is true.

Notice there is no semicolon after the expression in parentheses. Like the **if** statement, the while loop is not complete without the statement that follows it.

If you wish the while loop to repeat a block of statements, its format is:

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

The while loop works like an if statement that executes over and over. As long as the expression inside the parentheses is true, the conditionally executed statement or block will repeat. Program 6-1 uses the while loop to print "Hello" five times.

Programs 6-1

```
1  // This program demonstrates a simple while loop.
2  #include <iostream.h>
3
4  int main()
5  {
6     int number = 1;
7
8     while (number <= 5)
9     {
10       cout << "Hello\n";
11       number++;
12    }
13    cout << "That's all!\n";
14    return 0;
15 }
```

Program Output

```
Hello
Hello
Hello
Hello
Hello
That's all!
```

Let's take a closer look at this program. In line 7 an integer variable, **number**, is defined and initialized with the value 1. In line 9 the **while** loop begins with this statement:

while (number <=5)

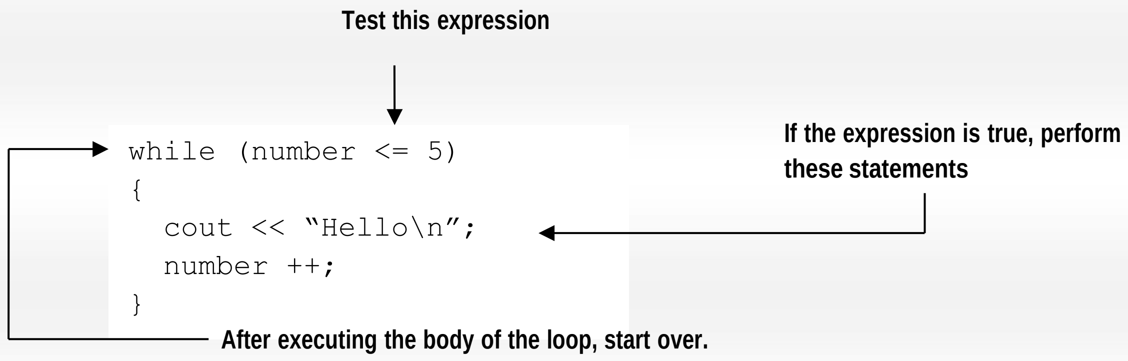
This statement tests the variable number to determine whether it is less than or equal to 5. If it is, then the statements in the body of the loop (line 11 and 12) are executed:

cout << "Hello \n";

number ++;

The statement in line 11 prints the word "Hello". The statement in line 12 uses the increment operator to add one to number. This is the last statement in the body of the loop, so after it executes, the loop starts over. It tests the expression **number <= 5** again, and if it is true, the statements in the body of the loop are executed again. This cycle repeats until the expression **number <= 5** is false.

Each repetition of loop is known as iteration. This loop will perform five iterations because the variable number is initialized with the value 1, and it is incremented each time the body of the loop is executed. When the expression number <= 5 is tested and found to be false, the loop will terminate and the program will resume execution at the statement that immediately follows the loop.


Test this expression
If the expression is true, perform these statements
After executing the body of the loop, start over.

```
while (number <= 5)
{
   cout << "Hello\n";
   number ++;
}
```

The while loop 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 in Line 6 of Program 6-1:

int number = 1;

The number variable is initialized with the value 1. If the number had been initialized with a value that is greater than 5, as shown in the following program segment, the loop would never execute:

```
int number = 6;
while (number <= 5)
{
  cout << “Hello\n”;
  number++;
}
```

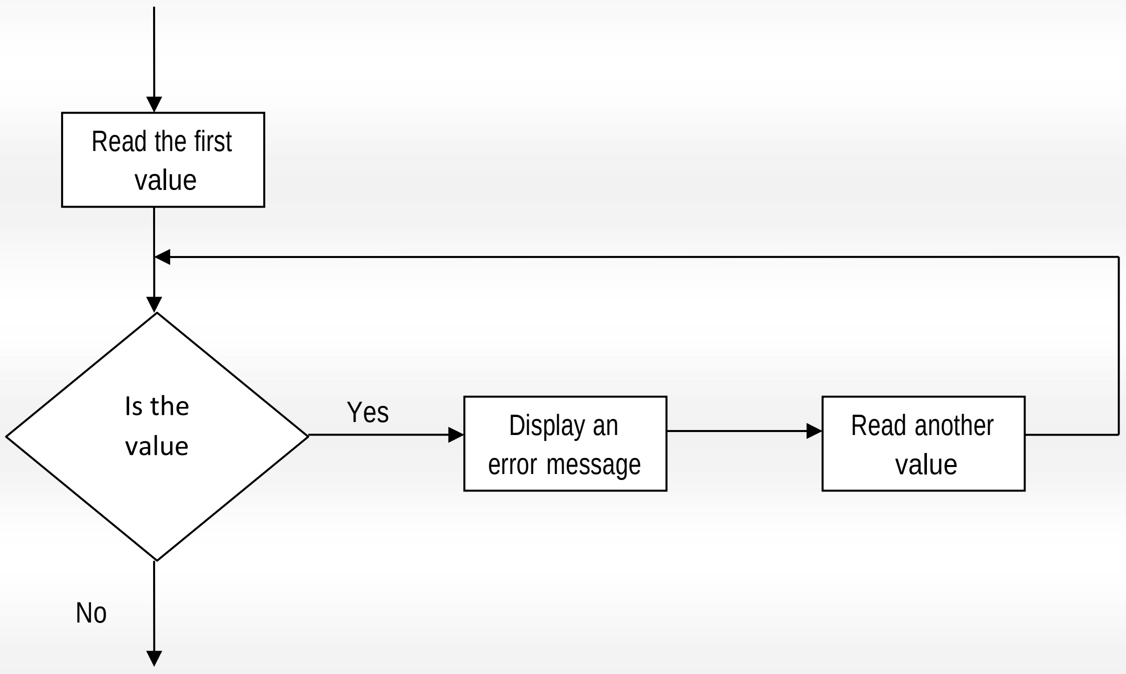
An important characteristic of the while loop is that the loop will never iterate if the test expression is false to start with. If you want to be sure that a while loop executes the first time, you must initialize the relevant data in such a way that the test expression starts out as true.

Using the while Loop for Input Validation

The **while** loop is especially useful for validating input. If an invalid value is entered, a loop can require that the user re-enter it as many times as necessary. For example, the following loop asks a number in the range of 1 through 100:

```
cout << “Enter a number in the range 1 – 100:”;
cin >> number;
while (number < 1 || number > 100)
{
  cout << “ERROR : Enter a value in the range 1 -100: “;
  cin >> number;
}
```

This code first allows the user to enter a number. This takes place just before the loop. If the input is valid, the loop will not execute. If the input is invalid, however, the loop will display an error message and require the user to enter another number. The loop will continue to execute until the user enters a valid number. The general logic of performing input validation is shown in Figure 6-1.



The read operation that takes place just before the loop is called a priming read. It provides the first value for the loop to test. Subsequent values are obtained by the loop.

Programming 6-2 calculates the number of soccer teams a youth league may create, based on a given number of players and a maximum number of players per team. The program uses while loops to validate user input.

Programs 6-2

```
/ This program calculates the number of soccer teams
// that a youth league may create from the number of
// available players. Input validation is demonstrated
// with while loops.
#include <iostream.h>

int main()
{
   int players,       // Number of available players
       teamPlayers,   // Number of desired players per team
       numTeams,      // Number of teams
       leftOver;      // Number of players left over

   // 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;

   // Validate the input.
   while (teamPlayers < 9 || teamPlayers > 15)
   {
      cout << "You should have at least 9 but no\n";
      cout << "more than 15 per team.\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;

// Validate the input.
   while (players <= 0)
   {
      cout << "Please enter a positive number: ";
      cin >> players;
   }

   // Calculate the number of teams.
   numTeams = players / teamPlayers;
// Calculate the number of leftover players.
   leftOver = players % teamPlayers;

   // Display the results.
   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]
You should have at least 9 but no
more than 15 per team.
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.
```

➲ Checkpoint

1. Write an input validation loop that asks the user to enter a number in the range 10 through 25.

2. Write an input validation loop that asks the user to enter ‘Y’, ‘y’, ‘N’, or ‘n’.

3. Write an input validation loop that asks the user to enter “Yes” or “No”.

6.3 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 control or keep track of the number of iterations a loop performs. For example, Program 6-3 displays a table consisting of the numbers 1 through 10 and their squares, so, its loop must iterate 10 times.

Programs 6-3

```
// This program displays the numbers 1 through 10 and
// their squares.
#include <iostream.h>

int main()
{
   int num = 1; //Initialize the counter.

   cout << "Number Number Squared\n";
   cout << "-----------------------------------\n";
   while (num <= 10)
   {
      cout << num << "\t\t" << (num * num) << endl;
      num++; //Increment the 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 6-3, 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.

6.4 The do-while Loop

The do-while loop is a posttest loop which means its expression is tested after each iteration.

The do-while loop looks something like an inverted while loop. Here is the do-while loop's format when the body of the loop contains only a single statement:

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

Here is the format of the do-while loop when the body of the loop contains multiple statements:

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

NOTE : The do-while loop must be terminated with a semicolon.

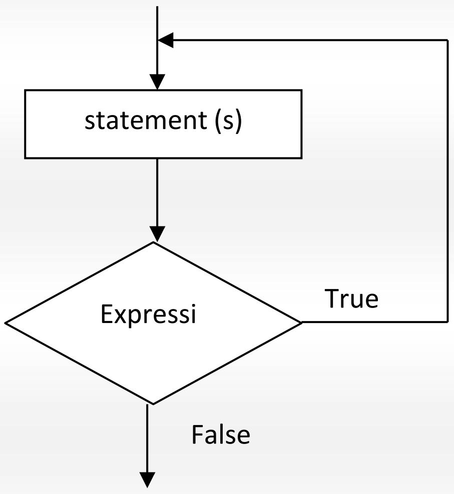
The **do-while** loop is a **post-test loop**. This means it does not test its expression until it has completed an iteration. As a result, the **do-while** loop always performs at least one iteration, even if the expression is false to begin with. This differs from the behavior of a while loop, which you will recall is a **pretest loop**. 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);
```

Figure 6-2 illustrates the logic of the **do-while** loop.



You should use the **do-while** loop when you want to make sure **the loop executes at least once**. For example, Program 6-4 averages a series of three test scores for a student. After the average is displayed, it asks the user if he or she wants to average another set of test scores. The program repeats as long as the user enters Y for yes.

Programs 6-4

```
// This program averages 3 test scores. It repeats as
// many times as the user wishes.
#include <iostream.h>

int main()
{
   int score1, score2, score3; // Three scores
   double average;             // Average score
   char again;                 // To hold Y or N input

   do
   {
      // Get three scores.
      cout << "Enter 3 scores and I will average them: ";
      cin >> score1 >> score2 >> score3;

      // Calculate and display the average.
      average = (score1 + score2 + score3) / 3.0;
      cout << "The average is " << average << ".\n";

      // Does the user want to average another set?
      cout << "Do you want to average another set? (Y/N) ";
      cin >> again;
   } while (again == 'Y' || again == 'y');
   return 0;
}
```

Program Output

```
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 74.3333
Do you want to average another set? (Y/N) n [Enter]
```

When this program was written, the programmer had no way of knowing the number of times the loop would iterate. This is because the loop asks the user if he or she wants to repeat the process. This type of loop is known as a **user-controlled loop**, because it allows the user to decide the number of iterations.

Using do-while with Menus

The do-while loop is a good choice for repeating a menu. Program 6-5 uses a do-while loop to repeat the program until the user selects item from the menu.

Programs 6-5

```
/ This program displays a menu and asks the user to make a
// selection. A do-while loop repeats the program until the
// user selects item 4 from the menu.
#include <iostream>
#include <iomanip>
int main()
{
   int choice;        // Menu choice
   int months;        // Number of months
   double charges;    // Monthly charges

   // Constants for membership rates
   const double ADULT = 40.0;
   const double SENIOR = 30.0;
   const double CHILD = 20.0;

   // Set up numeric output formatting.
   cout << fixed << showpoint << setprecision(2);

   do
   {
      // Display the menu.
      cout << "\n\t\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;
// Validate the menu selection.
      while (choice < 1 || choice > 4)
      {
         cout << "Please enter 1, 2, 3, or 4: ";
         cin >> choice;
      }

      // Validate and process the user's choice.
      if (choice != 4)
      {
         // Get the number of months.
         cout << "For how many months? ";
         cin >> months;

         // Respond to the user's menu selection.
         switch (choice)
         {
            case 1: charges = months * ADULT;
                     break;
            case 2: charges = months * CHILD;
                     break;
            case 3:  charges = months * SENIOR;
         }

         // Display the monthly charges.
         cout << "The total charges are $";
         cout << charges << endl;
      }
   } while (choice != 4);
   return 0;
}
```

Program Output

```
           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]
```

➲ Checkpoint

1. What will the following program segments display?

a.

```
int count = 10;
do
  cout << "Hello World \n";
while (count ++ <1);
```

b.

```
int v = 0;
do
  cout << v++;
while (v < 5);
```

c.

```
int count=0, funny=1, serious=0, limit=4;
do
{
  funny++;
  serious +=2;
}while(count++ < limit);
cout << funny << " " << serious << " ";
cout << count << endl;
```

6.5 The for Loop

The **for** loop is ideal for performing a known number of iterations.

In general, there are two categories of loops: conditional loops and count-controlled loops. A conditional loop executes as long as a particular condition exists. For example, an input validation loop executes as long as the input value is invalid. When you write a conditional loop, you have no way of knowing the number of times it will iterate.

Sometimes you know the exact number of iterations that a loop must perform. A loop that repeats a specific number of times is known as a count-controlled loop. For example, if the loop asks the user to enter the sales amounts for each month in the year, it will iterate twelve times. In essence, the loop counts to twelve and asks the user to enter a sales amount each time it takes a count. A count-controlled loop must possess three elements:

1. It must initialize a counter variable to a starting value.
2. It must test the counter variable by comparing it to a maximum value. When the counter variable reaches its maximum value, the loop terminates.
3. It must update the counter variable during each iteration. This is usually done by incrementing the variable.

```
for (initialization; test; update)
     statement;
```

The first line of the **for** loop is the loop *header*. After the keyword for, there are three expressions inside the parentheses, separated by semicolons. (Notice there is not a semicolon after the third expression). The first expression is the *initialization expression*. It is normally used to initialized a counter variable to its starting value. This is the first action performed by the loop, and it is done once. The second expression is the *test expression*. This is an expression that 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. The third expression is the *update expression*. It executes at the end of each iteration. Typically, this is a statement that increments the loop's counter variable.

Here is an example of a simple for loop that prints "Hello" five times:

```
    for (count = 1; count <= 5; count ++)
      cout << "Hello" << endl;
```

Figure 6-3 illustrates the sequence of events that takes place during the loop's execution. Notice that Steps 2 through 4 are repeated as long as the test expression is **true**.

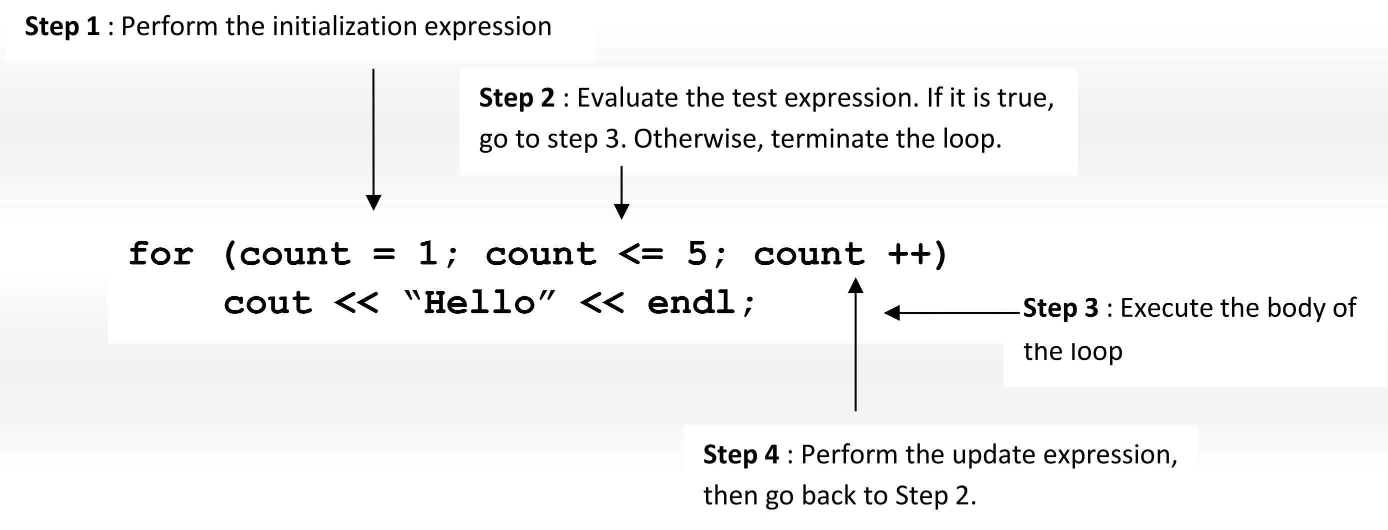
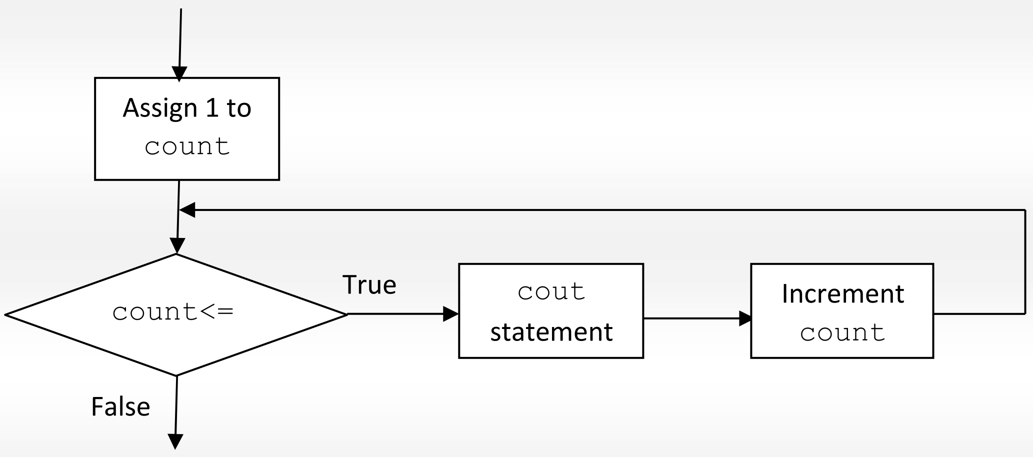

Step 1 : Perform the initialization expression
Step 2 : Evaluate the test expression. If it is true, go to step 3. Otherwise, terminate the loop.
for (count = 1; count <= 5; count ++)
cout << "Hello" << endl;
Step 3 : Execute the body of the loop
Step 4 : Perform the update expression, then go back to Step 2.

Figure 6-4 shows the loop's logic in the form of a flowchart.



Notice how the counter variable, count, is used to control the number of times that the loop iterates. During the execution of the loop, this variable takes on the values 1 through 5, and when the test expression count <= 5 is false, the loop terminates. Also notice that in this example the count variable is used only in the loop header, to control the number of loop iterations. It is not used for any other purpose. It is also possible to use the counter variable within the body of the loop. For example, look at following code:

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

The counter variable in this loop is **number**. In addition to controlling the number of iterations, it is also used in the body of the loop. This loop will produce the following output:

```
1 2 3 4 5 6 7 8 9 10
```

Program 6-6 shows another example of a **for** loop that uses its counter variable within the body of the loop. This is yet another program that displays a table showing the numbers 1 through 10 and their squares.

Programs 6-6

```
// This program demonstrates a user controlled for loop.
#include <iostream.h>

int main()
{
   int num;        // Loop counter variable
   int maxValue;   // Maximum value to display

   // Get the maximum value to display.
   cout << "I will display a table of numbers and\n";
   cout << "their squares. How high should I go? ";
   cin >> maxValue;

   cout << "\nNumber    Number Squared\n";
   cout << "-------------------------\n";

   for (num = 1; num <= maxValue; num++)
      cout << num << "\t\t" << (num * num) << endl;
   return 0;
}
```

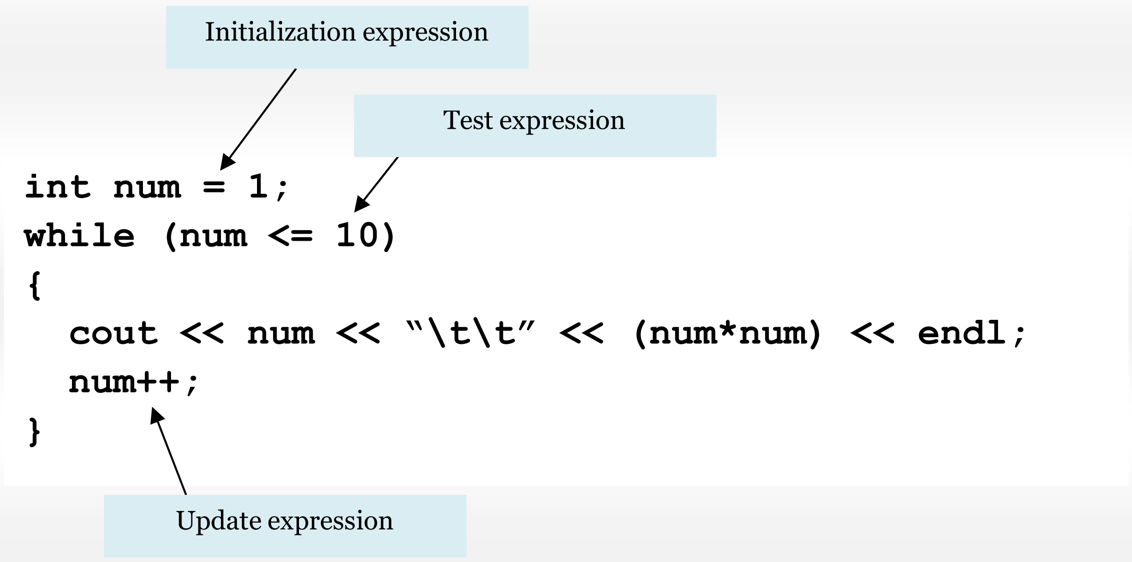
Program Output

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

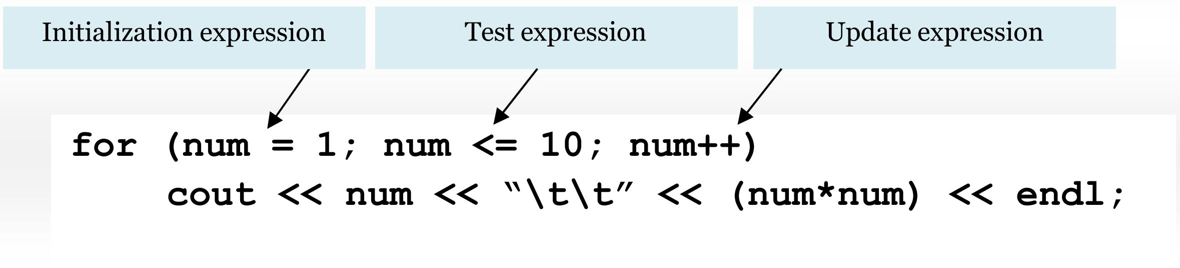
Using the for Loop Instead of while or do-while

You should use the **for** loop instead of the **while** or **do-while** loop in any situation that clearly requires an initialization, uses a false condition to stop the loop, and requires an update to occur at the end of each loop iteration.

Figure below shows how the **while** loop and the **for** loop in each have initialization, test, and update.



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



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

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 (num=2; num<=100; num+=2)
     cout << num << endl;
```

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

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

Creating a User Controlled for Loop

Sometimes you want the user to determine the maximum value of the counter variable in a **for** loop, and therefore determine the number of times the loop iterates. Instead of displaying the numbers 1 through 10 and their squares, this program allows the user to enter the maximum value to display.

Before the loop, this program asks the user to enter the highest value to display. This value is stored in the **maxValue** variable:

```
cout << "I will display a table of numbers and \n";
cout << "their squares. How high should I go? ";
cin >> maxValue;
```

The **for** loop's test expression then uses this value as the upper limit for the control variable:

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

In this loop, the **num** variable takes on the values 1 through **maxValue**, and then the loop terminates.

Programs 6-7

```
// This program demonstrates a user controlled for loop.
#include <iostream.h>

int main()
{
   int num;        // Loop counter variable
   int maxValue;   // Maximum value to display

   // Get the maximum value to display.
   cout << "I will display a table of numbers and\n";
   cout << "their squares. How high should I go? ";
   cin >> maxValue;

   cout << "\nNumber    Number Squared\n";
   cout << "-------------------------------------\n";

   for (num = 1; num <= maxValue; num++)
      cout << num << "\t\t" << (num * num) << endl;
   return 0;
}
```

Program Output

```
I will display a table of numbers and
their squares. How high should I go? 5 [Enter]
Number          Number Squared
---------------------------------------------------
1                    1
2                    4
3                    9
4                    16
5                    25
```

➲ Checkpoint

1. Name the three expressions that appear inside the parentheses in the **`for`** loop's header.

2. You want to write a **`for`** loop that displays "**`I love to program`**" 50 times. Assume that you will use a counter variable named **`count`**.
   a. What initialization expression will you use?
   b. What test expression will you use?
   c. What update expression will you use?
   d. Write the loop.

3. What will the following program segments display?
   a.
   ```
   for (int count = 0; count < 6; count ++)
       cout << (count + count);
   ```
   b.
   ```
   for (int value = -5; value < 5; value++)
       cout << value;
   ```
   c.
   ```
   int x;
   for ( x=5; x<=14; x+=3)
       cout << x << endl;
   cout << x << endl;
   ```

4. Write a **`for`** loop that displays your name 10 times.

5. Write a **`for`** loop that displays all of the odd numbers, 1 through 49.

6. Write a **`for`** loop that displays every fifth number, zero through 100.

6.6 Keeping a Running Total

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.

Programs that calculate the total of a series of numbers typically use two elements:

- A loop that reads each number in the series
- A variable that accumulates the total of numbers as they are read.

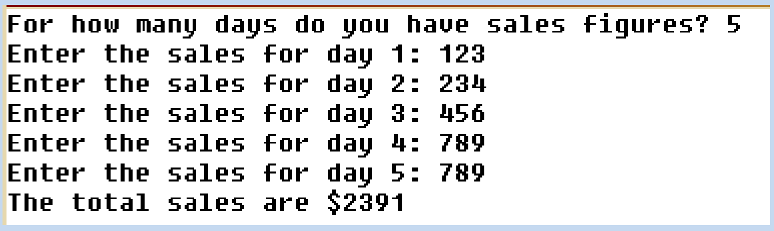
The variable that is used to accumulate the total of the numbers is called an accumulator. It is often said that the loop keeps a running total because it accumulates the total as it reads each number in the series.

Let’s look at a program that calculates a running total.

Programs 6-8

```
// This program takes daily sales figures over a period of time
// and calculates their total.
#include  <iostream.h>
#include <iomanip.h>

int main()
{
   int days;              // Number of days
   double total = 0.0;   // Accumulator, initialized with 0

   // Get the number of days.
   cout << "For how many days do you have sales figures? ";
   cin >> days;

   // Get the sales for each day and accumulate a total.
   for (int count = 1; count <= days; count++)
   {
      double sales;
      cout << "Enter the sales for day " << count << ": ";
      cin >> sales;
      total += sales;    // Accumulate the running total.
   }

   // Display the total sales.
   cout << fixed << showpoint << setprecision(2);
   cout << "The total sales are $" << total << endl;
   return 0;
}
```

Program Output



```
For how many days do you have sales figures? 5
Enter the sales for day 1: 123
Enter the sales for day 2: 234
Enter the sales for day 3: 456
Enter the sales for day 4: 789
Enter the sales for day 5: 789
The total sales are $2391
```

6.7 Sentinels

A sentinel is a special value that marks the end of a list of values. Program 6-8, in the previous section, requires the user to know in advance the number of days he or she wishes to enter 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.

Programs 6-9

```
// 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,    // Game counter
       points,      // To hold a number of points
       total = 0;   // Accumulator

   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;
      game++;
      cout << "Enter the points for game " << game << ": ";
      cin >> points;
   }
   cout << "\nThe total points are " << total << endl;
   return 0;
}
```

Program Output

```
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
```

The value -1 was chosen for the sentinel in this program because it is not possible for a team to score negative points. Notice that this program performs a priming read in the highlighted line to get the first value. This makes it possible for the loop to immediately terminate if the user enters -1 as the first value. Also note that the sentinel value is not included in the running total.

➲ Checkpoint

1. Describe the difference between pretest loops and posttest loops?

2. What is the difference between while loop and the do-while loop?

3. Which loop should you use in situations where you wish the loop to repeat until the test expression is false, but the loop should execute at least one time?

4. Which loop should you use when you know the number of required iterations?

5. Write a **for** loop that displays the following set of numbers:

```
0, 10, 20, 30, 40, 50 …. 1000
```

6. Write a loop that asks the user to enter a number. The loop should iterate 10 times and keep a running total of the numbers entered.

7. Convert the following **while** loop to a **do while** loop:

```
int x=1;
while (x > 0)
{
  cout << "Enter a number :  ";
  cin >> x;
}
```

8. Convert the following **do-while** loop to a **while** loop:

```
char sure;
do
{
  cout <<"Are you sure you want to quit?";
  cin >> sure;
} while (sure != 'Y' && sure != 'N');
```

9. Convert the following **while** loop to a **for** loop:

```
int count = 0;
while (count < 50)
{
  cout << “count is” << count << endl;
  count ++;
}
```

10. Convert the following **for** loop to a **while** loop:

```
for (int x =50; x>0; x--)
{
  cout << x << “seconds to go.\n”;
}
```

11. The following program has errors. Find as many as you can.

```
//this program adds two numbers entered by the user
int  main ( )
{
  int num1, num2;
  char again;

  while (again == ‘y’|| again == ‘Y’)
              cout << “Enter a number: “;
              cin >> num1;
              cout << “Enter another number:  “;
              cin >> num2;
              cout << “Their sum is << (num1 + num2) << endl;
              cout << “Do you want to do this again?  “;
              cin >> again;
  return 0;
}
```

12. Ocean Levels

Assuming the ocean's level is currently rising at about 1.5 millimeters per year, write a program that displays a table showing the number of millimeters that the ocean will have risen each year for the next 25 years.

13. **Calories Burned**
    Running on a particular treadmill you burn 3.9 calories per minute. Write a program that uses a loop to displays a table showing the number of calories burned after 10, 15, 20, 25, and 30 minutes.

Instructions: Find the true or false statement below. Then, rewrite the remaining false statements so they are true.

1. The idea of inheritance makes object-oriented programming more reusable then code generated by top-down design.
2. Java is the ideal development language, which is why other programming languages are beginning to lose their importance.
3. Prototyping is a form of rapid application development (RAD), which enables programmers to build software that executes incredibly

Quiz Yourself Online: Do Self Test in the E-Learning

ACTIVITY

1. **Write a C++ program to print alphabets from a to z using for loop. How to print alphabets using loop in C++ programming. Logic to print alphabets form a to z using for loop in C++ programming.**
2. **Write a C program to print all odd numbers from 1 to n using for loop. How to print odd numbers from 1 to n using loop in C programming. Logic to print odd numbers in a given range in C programming.**

KEY TERM

while	**Pretest loop**
Do-while	**User controlled loop**
for	**Sentinels**
Counters	
Post test loop	

SUMMARY

- Introduced to the three repetition constructs while, do-while and for statements
- The initialization, expression and counter which are important for all repetition constructs are explained.

TOPIC 7

Functions

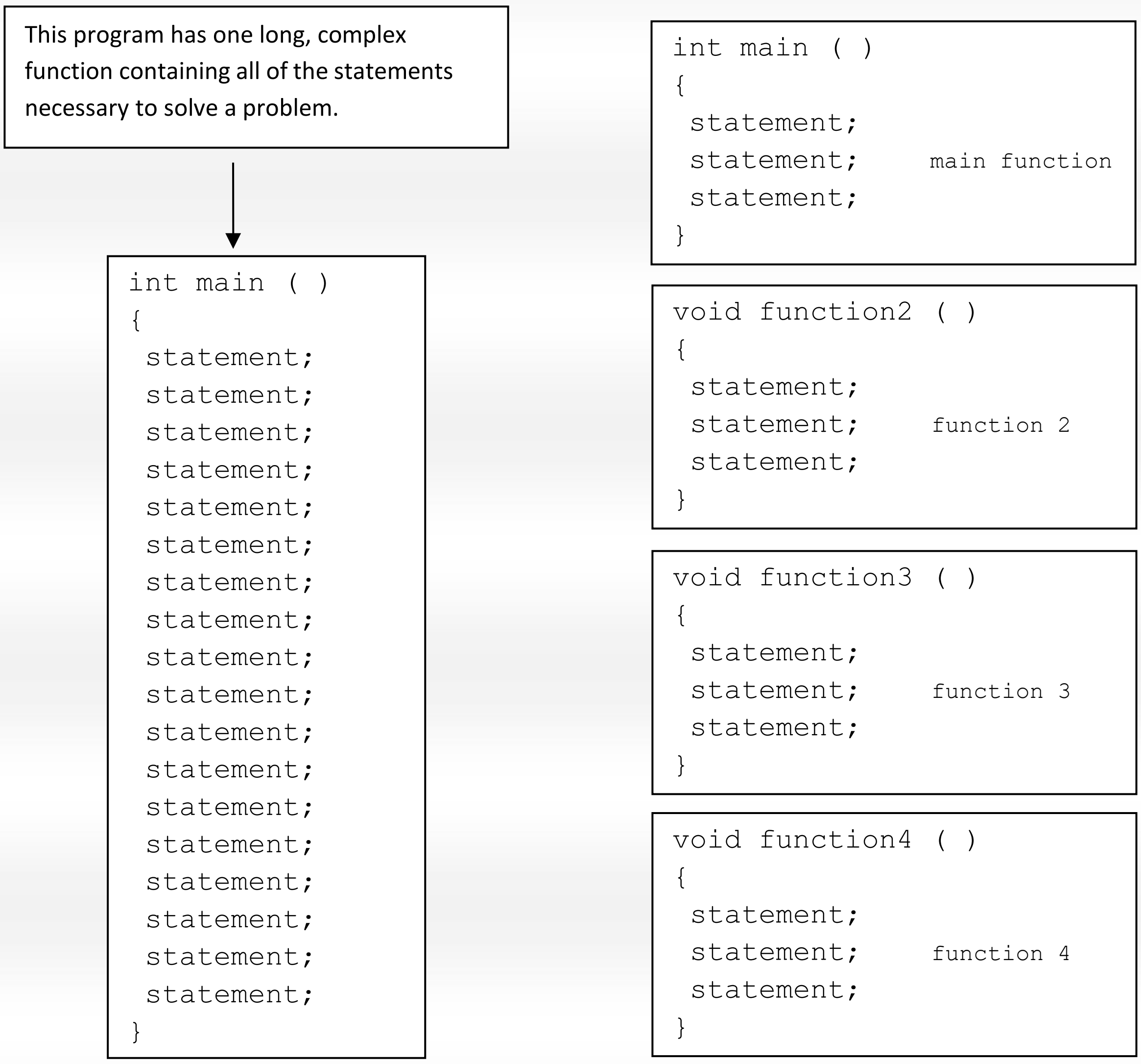
LEARNING OUTCOMES

By the end of topic, you should be able to:

1. Define functions.
2. Write function prototypes
3. Write C++ program that invoke functions

C++ is a procedural language. It has functional features that enable us to structure our programs better. So far, the program examples given are easy and small. In fact, all program statements can be put into one function which is the main () function. This structure is not suitable for large programs because it makes the programs hard to read and understood. Proper code arrangement is needed so that it can clarify logical structure and flow of program execution. This will make sure programs are easier to read, understand and maintained.

Figure 7-1 illustrates this idea by comparing two programs: one that uses a long complex function containing all of the statements necessary to solve a problem, and another that divides a problem into smaller problems, each of which are handled by separate function.


This program has one long, complex function containing all of the statements necessary to solve a problem.

```
int main ( )
{
 statement;
 statement;
 statement;
 statement;
 statement;
 statement;
 statement;
 statement;
 statement;
 statement;
 statement;
 statement;
 statement;
 statement;
 statement;
 statement;
 statement;
 statement;
}
```

```
int main ( )
{
 statement;
 statement;     main function
 statement;
}
```

```
void function2 ( )
{
 statement;
 statement;     function 2
 statement;
}
```

```
void function3 ( )
{
 statement;
 statement;     function 3
 statement;
}
```

```
void function4 ( )
{
 statement;
 statement;     function 4
 statement;
}
```

Figure 7-1

Another reason to write functions is that they simplify programs. If a specific task is performed in several places in a program, a function can be written once to perform that task, and then be executed anytime it is needed. This benefit of using functions is known as code reuse because you are writing the code to perform that task, and then reusing it each time you need to perform the task.

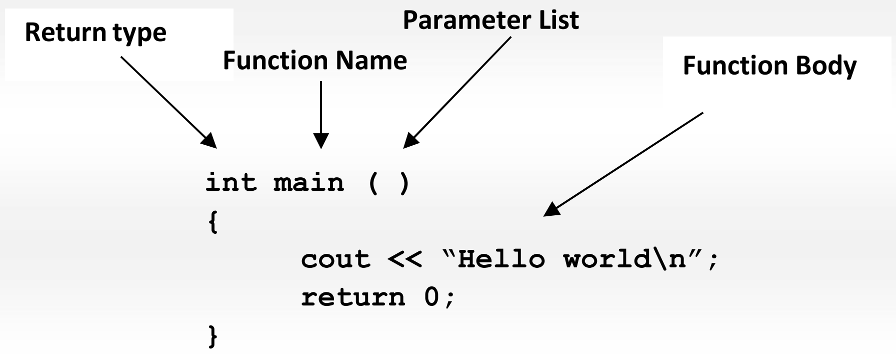
7.1 Defining and Calling Functions

CONCEPT : 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:

Return type	:	A function can send a value to the part of the program that executed it. The return type is the data type of the value that is sent from the function.
Name	:	You should give each function a descriptive name. In general, the same rules that apply to variable names also apply to function names.
Parameter list	:	The program can send data into a function. The parameter list is a list of variables that hold the values being passed to the function.
Body	:	The body of a function is the set of statements that perform the function’s operation. They are enclosed in a set of braces.

Figure 7-2 shows the definition of a simple function with the various parts labeled.



```
int main ( )
{
      cout << “Hello world\n”;
      return 0;
}
```

Figure 7-2

The line in the definition that reads **int main ( )** is called the function header.

void Functions

You already know that a function can return a value. The main function in all of the programs you have seen is declared to return an **int** value to the operating system. The return 0; statement causes the value 0 to be returned when the main function finishes executing.

It isn't necessary for all functions to return a value, however. Some functions simply perform one or more statements which follow terminate. These are called void functions. The displayMessage function which follows is an example.

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

The function's name is displayMessage. This name gives an indication of what the function does: It displays a message. You should always give function names that reflect their purpose. Notice that the function's return type is void. This means the functions 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 execute the statements in its body. Let's look at Program 6-1, which contains two functions: **main** and **displayMessage**.

The function displayMessage is called by the following statement line 18:

displayMessage();

Programs 7-1

```
// This program has two functions: main and displayMessage
#include <iostream.h>

//****************************************
// Definition of function displayMessage  *
// This function displays a greeting.     *
//****************************************
void displayMessage()
{
   cout << "Hello from the function displayMessage.\n";
}
//****************************************
// Function main                          *
//****************************************
int main()
{
   cout << "Hello from main.\n";
   displayMessage();
   cout << "Back in function main again.\n";
   return 0;
 }
```

Program Output

```
Hello from main.
Hello from the function displayMessage.
Back in function main again.
```

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

Function Header	→	**void displayMessage()**	is a part of 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.
Function Call	→	**displayMessage();**	Is a statement that executes the function, so it is

terminated with a semicolon
like all other C++ statements.

Program 7-2 have many functions: main, first, and second.

In lines ** and ** of **Program 7-2**, function main contains a call to first and a call to second:

first ( );
second ( );
Each call statement causes the program to branch to a function and then back to main when the function is finished.

Program Output

```
I am starting in function main.
I am now inside the function first.
I am now inside the function second.
Back in function main again.
```

Pro

```
1
2
3
4   //*****************************************
5   // Definition of function first          *
6   // This function displays a message.      *
7   //*****************************************
8
9   void first()
10  {
11    cout << "I am now inside the function first.\n";
12  }
13
14  //*****************************************
15  // Definition of function second         *
16  // This function displays a message.      *
17 //*****************************************
18
19 void second()
20  {
21    cout << "I am now inside the function second.\n";
22  }
23
24  //*****************************************
25  // Function main                          *
26 //*****************************************
27
28 int main()
29  {
30    cout << "I am starting in function main.\n";
31    first();    // Call function first
32    second();   // Call function second
33    cout << "Back in function main again.\n";
34    return 0;
35  }
```

➲ Checkpoint

1. Is the following a function header or a function call?

   `calcTotal ( );`

2. Is the following a function header or a function call?

   `void showResults ( )`

3. What will the output of the following program be if the user enters 10?

```
#include <iostream.h>

void func1( )
{
       cout << "Able was I\n";
}

void func2( )
{
       cout << "I saw Elba\n";
}
int main ( )
{
       int input;
       cout << "Enter a number: ";
       cin >> input;
       if (input < 10)
       {
         func1();
         func2();
       }
       else
       {
         func2();
         func1();
       }
return 0;
}
```

7.2 Function Prototypes

CONCEPT : 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 the function's return type, the number of parameters it uses, and the type of each parameter.

One way ensuring that the compiler has this information is to place the function definition before all calls to that function. Another method is to declare the function with a function prototype. Here is a prototype for the **displayMessage** function in Program 7-1:

```
void displayMessage ( );
```

The prototype looks similar to the function header, except there is a semicolon at the end. The statement above tells the compiler that the function has a void return type (it doesn't return a value) and uses no parameters.

You must place either 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. Program 7-3 is a modification of Program 7-1. The definitions of the functions first and second have been placed after main, and a function prototype has been placed after the **#include <iostream.h>** statement.

When the compiler is reading Program 7-3, it encounters the calls to the functions first and second in lines 12 and 13 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.

Programs 7-3

```
// This program has three functions: main, First, and Second.
#include <iostream>


// 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";
   return 0;
}

//*************************************
// Definition of function first.      *
// This function displays a message.  *
//*************************************

void first()
{
   cout << "I am now inside the function first.\n";
}

//*************************************
// Definition of function second.     *
// This function displays a message.  *
//*************************************

void second()
{
   cout << "I am now inside the function second.\n";
}
```

Program Output

```
I am starting in function main.
I am now inside the function first.
I am now inside the function second.
Back in function main again.
```

7.3 Sending Data into a Function

CONCEPT : When a function is called, the program may send values into the function.

Values that are sent into a function are called arguments. By using parameters, you can design your own functions that accept data this way. A parameter is a special variable that holds a value being passed into a function. 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 values as an argument. **Program 7-4** is a complete program using this function.

Programs 7-4

```
1  // This program demonstrates a function with a parameter.
2  #include <iostream.h>
3
4
5  // Function Prototype
6  void displayValue(int);
7
8  int main()
9  {
10    cout << "I am passing 5 to displayValue.\n";
11    displayValue(5);  // Call displayValue with argument 5
12    cout << "Now I am back in main.\n";
13    return 0;
14 }
15
16 //*********************************************************
17 // Definition of function displayValue.                  *
18 // It uses an integer parameter whose value is displayed. *
19 //*********************************************************
20
21 void displayValue(int num)
22 {
23   cout << "The value is " << num << endl;
24 }
```

Program Output

```
I am passing 5 to displayValue.
The value is 5
Now I am back in main.
```

First, notice the function prototype for displayValue in line 6:

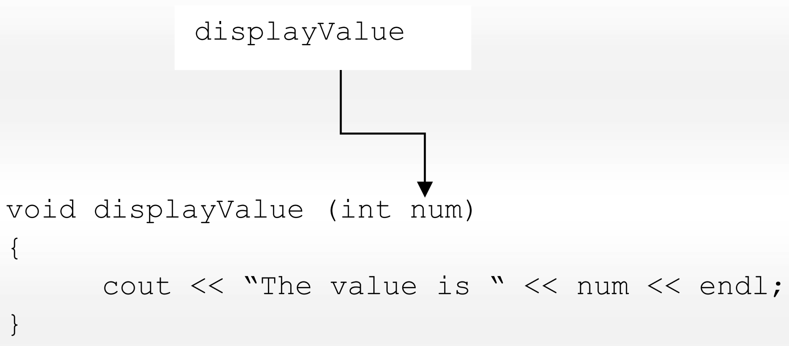
void displayValue(int);

It is not necessary to list the name of the parameter variable inside the parentheses. Only its data type is required. The function prototype shown above 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 figure 7-1.



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

Figure 7-1

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. Program 7-5 shows the function displayValue being called several times with a different argument being passed each time.

Program 7-5

```
// This program demonstrates a function with a parameter.
#include <iostream.h>

// Function Prototype
void displayValue(int);

int main()
{
   cout << "I am passing several values to displayValue.\n";
   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
   cout << "Now I am back in main.\n";
   return 0;
}

//***********************************************************
// Definition of function displayValue.                     *
// It uses an integer parameter whose value is displayed. *
//***********************************************************

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

Program Output

```
I am passing several values to displayValue.
The value is 5
The value is 10
The value is 2
The value is 16
Now I am back in main.
```

Each time the function is called in Program 7-5, **num** takes on a different value. Any expression whose value could normally be assigned to **num** may be used as an argument.

Often, it's useful to pass several arguments into a function. Program 7-6 shows the definition of a function with three parameters.

Program 7-6

```
// This program demonstrates a function with three parameters.
#include <iostream.h>

// Function Prototype
void showSum(int, int, int);

int main()
{
   int value1, value2, value3;

   // Get three integers.
   cout << "Enter three integers and I will display ";
   cout << "their sum: ";
   cin >> value1 >> value2 >> value3;

   // Call showSum passing three arguments.
   showSum(value1, value2, value3);
   return 0;
}

//***************************************************************
// Definition of function showSum.                              *
// It uses three integer parameters. Their sum is displayed.  *
//***************************************************************

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

Program Output

```
Enter three integers and I will display their sum: 4 8 7 [Enter]
19
```

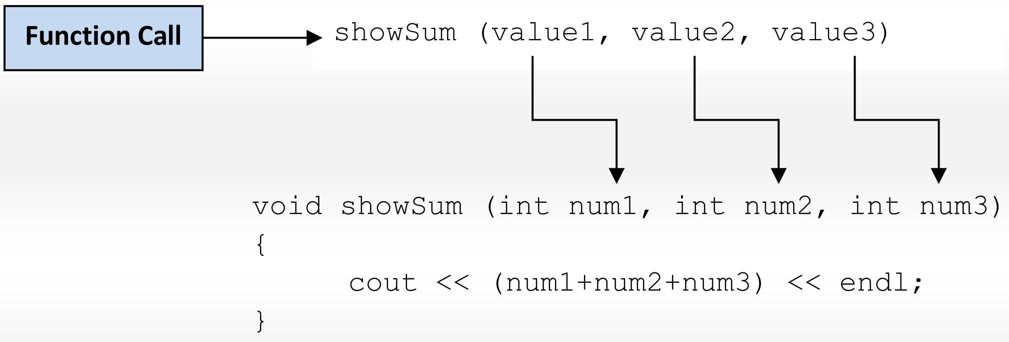
In the function header for showSum, the parameter list contains three variable definitions separated by commas:

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

In the function call, the variables value1, value2, and value3 are passed as arguments:

showSum(value1, value2, value3);

When a function with multiple parameters is called, the arguments are passed to the parameters in order. This is illustrated in Figure 7-2.



```
showSum (value1, value2, value3)

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

Figure 7-2

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, 10 to be passed into **num2**, and 10 to be passed into **num3**.

showSum (15, 5, 10);

The function prototype must list the data type of each parameter.

Like all variables, parameters have a scope. The scope of a parameter is limited to the body of the function that uses it.

7.4 Passing Data by Value

CONCEPT : When an argument is passed into a parameter, only a copy of the argument's value is passed. Changes to the parameter do not affect the original argument.

As you've seen in this chapter, parameters are special purpose variables that are defined inside the parentheses of a function definition. They are separate and distinct from the arguments that are listed inside the parentheses of a function call. The values that are stored in the parameter variables are copies of the arguments. Normally, when a parameter's value is changed inside a function it has no effect on the original argument. Program 7-7 demonstrates this concept.

Program 7-7

```
// This program demonstrates that changes to a function parameter
// have no affect on the original argument.
#include <iostream.h>

// Function Prototypevoid
changeMe(int);

int main()
{
   int number = 12;

   // Display the value in number.
   cout << "number is " << number << endl;

   // Call changeMe, passing the value in number
   // as an argument.
   changeMe(number);

   // Display the value in number again.
   cout << "Now back in main again, the value of ";cout <<
   "number is " << number << endl;
   return 0;
}
```

```
//*****************************************************************
// Definition of function changeMe.                               *
// This function changes the value of the parameter myValue.      *
//*****************************************************************

void changeMe(int myValue)
{
   // Change the value of myValue to 0.
   myValue = 0;

   // Display the value in myValue.
   cout << "Now the value is " << myValue << endl;
}
```

Program Output

```
number is 12
Now the value is 0
Now back in main again, the value of number is 12
```

Even though the parameter variable **myValue** is changed in the **changeMe** function, the argument number is not modified. The **myValue** variable contains only a copy of the number variable.

The **changeMe** function does not have access to the original argument. When only a copy of an argument is passed to a function, it is said to be passed by value. This is because the function receives a copy of the argument's value, and does not have access to the original argument.

Figure 7-3 illustrates that a parameter variable's storage location in memory is separate from the original argument.

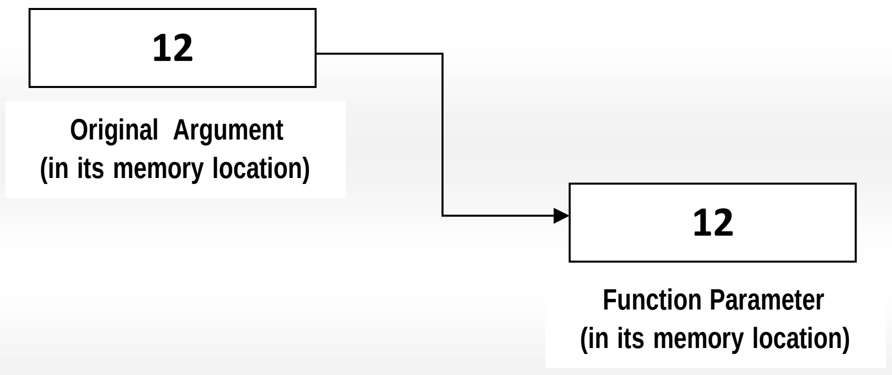


Figure 7-3

➲ Checkpoint

1. Indicate which of the following is the function prototype, the function header, and the function call:

   void showNum (double num)

   void showNum (double);

   showNum (45.67);

2. Write a function named timesTen. The function should have an integer parameter named number. When timesTen is called, it should display the product of number times ten. (Note : just write the function. Do not write a complete program).

3. Write a function prototype for the timesTen function you wrote in Question 2.

4. What is the output of the following program?

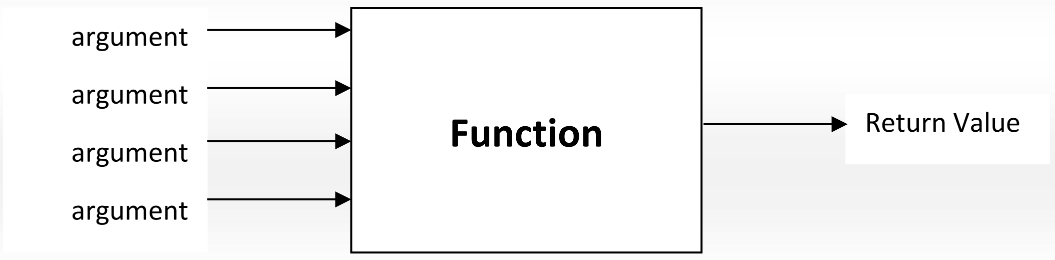
```
#include <iostream.h> void
func1 (double, int);int main
( )
{
        int x = 0; double y
        = 1.5;
        cout << x << “ “ << y << endl;
        func1 (y,x);
        cout << x << “ “ << y << endl;
        return 0;
}

void func1 (double a, int b)
{
        cout << a << “ “ << b << endl;
        a = 0.0;
        b = 10;
        cout << a << “ “ << b << endl;
}
```

7.5 Returning a Value form a Function

CONCEPT : A function may send a value back to the part of the program that called the function.

You've seen that data may be passed into a function by way of parameter variables. Data may also be returned from a function, back to the statement that called it. Functions that return a value are appropriately known as value-returning functions.

Although several arguments may be passed into a function, only one value may be returned from it. Think of a function as having multiple communication channels for receiving data (parameters), but only one channel fro sending data (the return value). This is illustrated in Figure 7-4.

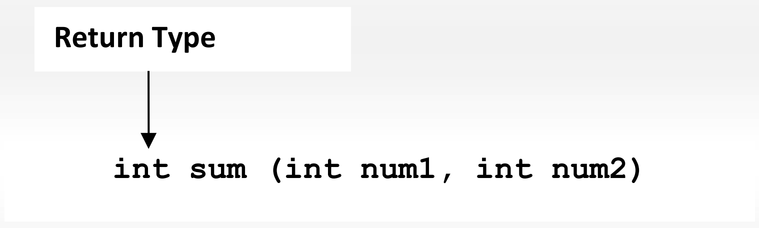


Defining a Value-Returning Function

When you are writing value-returning function, you must decide what type of value the function will return. This is because you must specify the data type of the return value in the function header, and in the function prototype. Recall that a void function, which does not return a value, uses the key word void as its return type in the function header. A value-returning function will use **int**, **double, bool**, or any other valid data type in its header. Here is an example of a function that returns an **int** value :

```
int sum(int num1, int num2)
{
  int result;
  result = num1 + num2;
  return result;
}
```

The name of this function is sum. Notice in the function header that the return type is **int**, as illustrated in Figure 7-5.



```
int sum (int num1, int num2)
```

Figure 7-5

This code defines a function named sum that accepts two **int** arguments. The arguments are passed into the parameter variables **num1** and **num2**. Inside the function, a variable **result**, is defined. Variables that are defined inside a function are called local variables. After the variable definition, the parameter variables **num1** and **num2** are added, and their **sum** is assigned to the **result** variable. The last statement in the function is

return result;

This statement causes the function to end, and it sends the value of the **result** variable back to the statement that called the function. A value-returning function must have a return statement written in the following general format:

```
return   expression;
```

In the general format, **expression** is the value to be returned. It can be any expression that has a value, such as a variable, literal or mathematical expression. The value of the expression is converted to the data type that the function returns, and is sent back to the statement that called the function. In this case, the **sum** function returns the value in the **result** variable.

However, we could have eliminated the result variable and returned the expression **num1 + num2**, as shown in the following code:

```
int sum(int num1, int num2)
{
  return num1 + num2;
}
```

When writing the prototype for a value returning function, follow the same conventions that we have covered earlier. Here is the prototype for the sum function:

int sum(int, int);

Calling a Value-Returning Function

Program 7-8

```
// This program uses a function that returns a value.
#include <iostream>

// Function prototype
int sum(int, int);

int main()
{
  int value1 = 20,    // The first value
      value2 = 40,    // The second value
      total;          // To hold the total

  // Call the sum function, passing the contents of
  // value1 and value2 as arguments. Assign the return
  // value to the total variable.
  total = sum(value1, value2);

  // Display the sum of the values.
  cout << "The sum of " << value1 << " and "
       << value2 << " is " << total << endl;
  return 0;
}

//*****************************************************
// Definition of function sum. This function returns  *
// the sum of its two parameters.                      *
//*****************************************************

int sum(int num1, int num2)
{
  return num1 + num2;
}
```

Program Output

```
The sum of 20 and 40 is 60
```

Here is the statement in line 16 that calls the **sum** function, passing **value1** and **value2** as arguments.

total = sum (value1, value2);

This statement assigns the value returned by the sum function to the total variable. In this case, the function will return 60. Figure 7-6 shows how the arguments are passed into the function and how a value is passed back from the function.

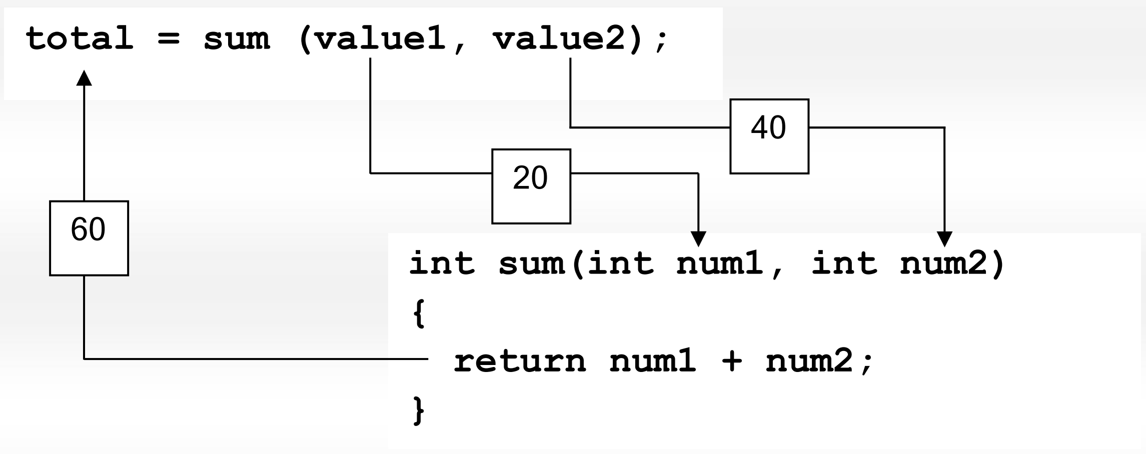


Figure 7-6

When you call a value-returning function, you usually want to do something meaningful with the value it returns. Program 7-7 shows a function's return value being assigned to a variable. This is commonly how return values are used, but you can do many other things with them. For example, the following code shows a mathematical expression that uses a call to the **sum** function:

```
int x=10, y=15;
double average;
average = sum(x,y) / 2.0;
```

Figure 7-7

In the last statement, the sum function is called with x and y as its arguments. The function's return value, which is 25, is divided by 2.0. The result, 12.5, is assigned to average.

Here is another example:

```
int x=10, y=15;
cout << "The sum is" << sum (x,y) << endl;
```

This code sends the sum function's return value to cout so it can be displayed on the screen. The message "The sum is 25" will be displayed.

➲ Checkpoint

1. What is the output of the following function?

```
#include <iostream.h>
void summer (int, int);
int fall (int, int);

int main ( )
{
 int Num1 = 2,Num2 = 5, x = 3 ;
 summer (Num1, Num2);
 cout << Num1 <<"  " << Num2 << x << endl;
 x = fall(Num1, Num2);
 cout << Num1 <<"  " << Num2 << x << endl;
}
void summer (int a, int b)
{
      int Num1;
      Num1 = b + 12;
      a = 2 * b + 5;
      b = Num1 + 4;
}
int fall (int u, int v)
{
      int Num2;
      Num2 = u + v;
      return Num2;
}
```

2. Fill in the blanks with the most suitable command/token:
   Isikan tempat kosong dengan arahan/token yang paling sesuai:

```
#include <iostream.h>
int______________________(int, int) //the function prototype
int main ( )
{
   int________________, secnum, max;
   cout << "Enter a number : ";
   cin >> firstnum;
   cout << "Great! Please enter a second number :  ";
   cin >> secnum;
   ___________= findMax (firstnum, secnum); //the function is called here
   cout << "\nThe maximum of the two number is" << max << endl;
   return 0;
}
____________findMax (int x, int y)
{                     //start of function body
   int maxnum;
   if (x >= y)
     maxnum = x;
   else
     maxnum = y;
   return______________; //return statement
}
```

3. Give the function prototype and function header for each of the following functions:
Berikan prototaip fungsi dan kepala fungsi bagi fungsi-fungsi berikut:

a. Function smallest that takes three integers, x, y, and z and returns an integer.
Fungsi smallest yang mengambil tiga nilai integer, x, y, dan z serta memulangkan nilai integer.

Function Prototype ***Prototaip Fungsi***	
Function Header ***Kepala Fungsi***	

b. Function *instructions* that does not receive any parameter and does not return a value.
Fungsi instructions yang tidak menerima sebarang parameter dan tidak memulangkan sebarang nilai.

Function Prototype ***Prototaip Fungsi***	
Function Header ***Kepala Fungsi***	

Instructions: Find the true or false statement below. Then, rewrite the remaining false statements so they are true.

1. The idea of inheritance makes object-oriented programming more reusable then code generated by top-down design.
2. Java is the ideal development language, which is why other programming languages are beginning to lose their importance.
3. Prototyping is a form of rapid application development (RAD), which enables programmers to build software that executes incredibly

Quiz Yourself Online: Do Self Test in the E-Learning

ACTIVITY

Write a program to find the factorial of a given number by using a function in C++ programming language. The question is about to print he factorial of a number y creating a function. The factorial of a number n means the product of all the numbers form 1 to n. How to create a function in C++ programming language to print the factorial of a number.

KEY TERM

Return type	**Function definition**
Function Name	**Function prototype**
Parameter List	**Function header**
Function Body	
Calling Function	

SUMMARY

- Introduced about functions which are small programs that can do a specific task
- How to define and call functions.
- Some functions that require parameters when called. Parameters are input to the function

REFERENCEES

Tony Gaddis, Starting Out with C++ - Early Objects (10th edition) ,Pearson, 2020.

TOPIC 8

Arrays

LEARNING OUTCOMES

By the end of topic, you should be able to:

1. Define the concepts of array.
2. Define the initialized array
3. Use the array contents for processing it.

8.1 Introduction

CONCEPT : An array allow you to store and work with multiple values of the same data type.

The variables you have worked with so far are designed to hold only one value at a time. Each of the variable definitions in **Figure 8-1** causes only enough memory to be reserved to hold one value of the specified data type.

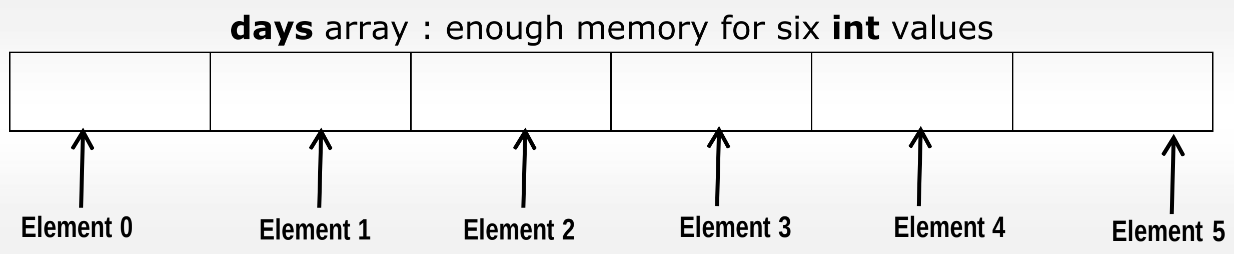
int count;	Enough memory for 1**int**
	12314
float price;	Enough memory for 1**float**
	56.981
char letter;	Enough memory for 1**char**
	A

Figure 8-1

An array works like a variable that can store a group of values, all of the same type. The values are stored together in consecutive memory locations. Here is a definition of an array of integers:

```
int days [6];
```

The name of this array is **days**. The number inside the brackets is the array's *size declarator*. It indicates the number of elements, or values, the array can hold. The **days** array can store six elements, each one an integer. This is depicted in **Figure 8-2**.



An array's size declarator must be a constant integer expression with a value greater than zero. It can be either a literal, as in the previous example, or a named constant, as shown in the following:

Const int NUM_DAYS = 6;

int days [NUM_DAYS];

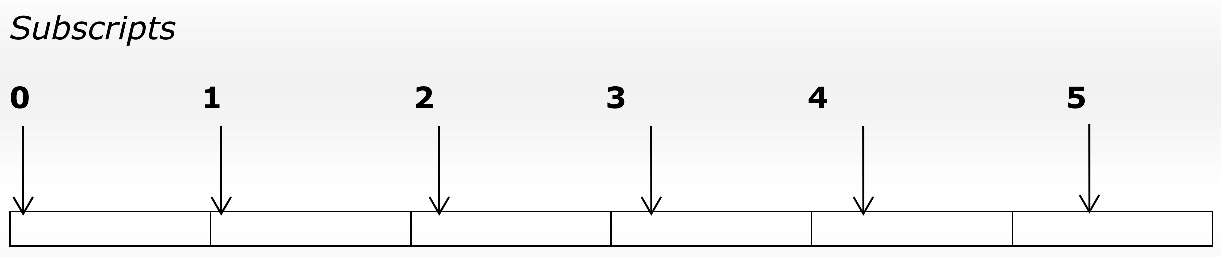
Arrays of any data type can be defined. The following are all valid array definitions:

```
float temperatures [100]; // array of 100 floats
char name [41];                  // array of 41 characters
long units [50];          // array of 50 long integers
double sizes [1200];      // array of 1200 doubles
```

8.2 Accessing Array elements

CONCEPT: the individual elements of an array are assigned unique subscripts. These subscripts are used to access the elements.

Even though an entire array has only one name, the elements may be accessed and used as individual variables. This is possible because each element is assigned a number known as a subscript. A subscript is used as an index to pinpoint a specific element within an array. The first element is assigned the subscript 0, the second element is assigned 1, and so forth. The six elements in the array hours would have the subscripts 0 through 5. This is shown in Figure 8-3.



int hours[6];

Figure 8-3

NOTE: subscript numbering in C++ always starts at zero. The subscript of the last element in an array is one less that the total number of elements in the array. This means that in the array shown in Figure 8-3, the element **`hours[6]`** does not exist. **`hours[5]`** is the last element in the array.

8.3 Inputting and Outputting Array Contents

Array elements may be used with the **cin** and **cout** objects like any othervariable. Program 8-1 shows the array hours being used to store and display values entered by the user.

Program 8-1

```
// This program asks for the number of hours worked
// by six employees. It stores the values in an array.
#include <iostream>

int main()
{
constint NUM_EMPLOYEES = 6;
int hours[NUM_EMPLOYEES];

   // Get the hours worked by each employee.
cout<< "Enter the hours worked by "
<< NUM_EMPLOYEES << " employees: ";
cin>> hours[0];
cin>> hours[1];
cin>> hours[2];
cin>> hours[3];
cin>> hours[4];
cin>> hours[5];

   // Display the values in the array.
cout<< "The hours you entered are:";
cout<< " " << hours[0];
cout<<  "  "  << hours[1];
cout<<  "  "  << hours[2];
cout<<  "  "  << hours[3];
cout<<  "  "  << hours[4];
cout<< " " << hours[5] <<endl;
return 0;
}
```

Program Output with Example Input Shown in Bold

```
Enter the hours worked by 6 employees: 20 12 40 30 30 15 [Enter]
The hours you entered are: 20 12 40 30 30 15
```

Figure 8-4 shows the contents of the array hours with the values entered by the user in the example output above.

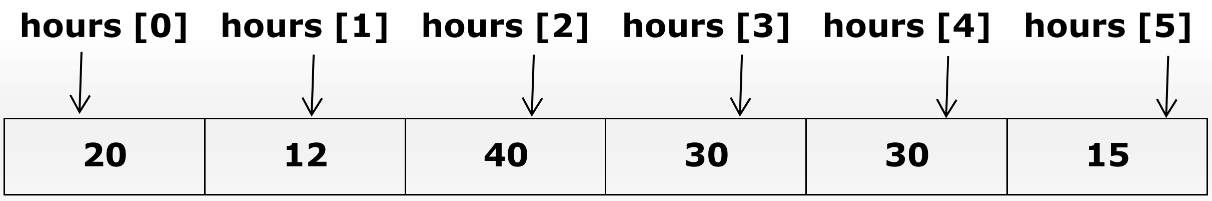


Figure 8-4

Even though the size declaratory of an array definition must be a constant or a literal, subscript numbers can be stored in variables. This makes it possible to use a loop to "cycle through" an entire array, performing the same operation on each element. For example, look at the following program code:

Program 8-2

```
// This program asks for the number of hours worked
// by six employees. It stores the values in an array.
#include <iostream.h>

int main()
{
constint NUM_EMPLOYEES = 6; // Number of employees
inthours[NUM_EMPLOYEES];    // Each employee's hours
int count;                   // Loop counter

   // Input the hours worked.
   for (count = 0; count < NUM_EMPLOYEES; count++)
   {
cout<< "Enter the hours worked by employee "
<< (count + 1) << ": ";
cin>>hours[count];
   }

   // Display the contents of the array.
cout<< "The hours you entered are:";
   for (count = 0; count < NUM_EMPLOYEES; count++)
cout<< " " <<hours[count];
cout<<endl;
  return 0;
}
```

```
Program Output with Example Input Shown in Bold
Enter the hours worked byemployee 1: 20[Enter]
Enter the hours worked by employee 2: 12 [Enter]
Enter the hours worked by employee 3: 40 [Enter]
Enter the hours worked by employee 4: 30 [Enter]
Enter the hours worked by employee 5: 30 [Enter]
Enter the hours worked by employee 4: 15[Enter]
The hours you entered are: 20 12 40 30 30 15
```

The first for loop, in line 12 through 17, prompts the user for each employee’s hours. Take a closer look at lines 14 through 16:

```
cout<< "Enter the hours worked by employee "
<< (count + 1) << ": ";
cin>> hours[count];
```

Notice that the **cout** statement uses the expression **count + 1** to display the employee number, but the **cin** statement uses count as the array subscript. This is because the hours for employee number 1 are stored in **hours[0**], the hours for employee number 2 are stored in **hours[1],** and so forth.

The loop in lines 20 through 23 also uses the count variable to step through the array, displaying each element.

➲ Checkpoint

1. Define the following arrays:
    - **a.** **empNums**, a 100-element array of ints.
    - **b.** **payRates**, a 25-element array of floats.
    - **c.** **cityName**, a 26-element array of chars.

2. What's wrong with the following array definitions?
    - **a.** **int readings [-1];**
    - **b.** **float measurements [4.5];**
    - **c.** **int size;**
    - **d.** **char name[size];**

3. What would the valid subscript values be in four-element array of integers?

4. What is the output of the following code?

```
int values[5], count;
for (count = 0; count < 5; count++)
        values [count] = count + 1;
for (count = 0; count < 5; count++)
        cout<< values [count] <<endl;
```

8.4 Array initialization

CONCEPT : Arrays may be initialized when they are defined.

Like regular variables, C++ allows you to initialize an array's elements when you create the array. Here is an example:

constint MONTHS = 12;
int days [MONTH] = {31,28,31,30,31,30,31,31,30,31,30,31};

The series of values inside the braces and separated with commas is called an initialization list. These values are stored in the array elements in the order they appear in the list. The first value, 31, is stored in days[0], the second value, 28, is stored in days[1], and so forth. Figure 8-5 shows the contents of the array after initialization.

Subscripts

0	1	2	3	4	5	6	7	8	9	10	11
31	28	31	30	31	30	31	31	30	31	30	31

Figure 8-5

Program 8-3 demonstrates how an array may be initialized

Program 8-3

```
// This program displays the number of days in each month.
#include <iostream.h>

int main()
{
constint MONTHS = 12;
int days[MONTHS] = { 31, 28, 31, 30,
                        31, 30, 31, 31,
                        30, 31, 30, 31};

for (int count = 0; count < MONTHS; count++)
   {
cout<< "Month " << (count + 1) << " has ";
cout<< days[count] << " days.\n";
   }
return 0;
}
```

Program Output with Example Input Shown in Bold

```
Month 1 has 31 days.
Month 2 has 28 days.
Month 3 has 31 days.
Month 4 has 30 days.
Month 5 has 31 days.
Month 6 has 30 days.
Month 7 has 31 days.
Month 8 has 31 days.
Month 9 has 30 days.
Month 10 has 31 days.
Month 11 has 30 days.
Month 12 has 31 days.
```

8.5 Processing Array Contents

CONCEPT: Individual array elements are processed like any other type of variable.

Processing array elements is no different than processing other variables. For example, the following statement multiplies hours[3] by the variable rate:

pay = hours[3] * rate;

Program 8-4 demonstrates the use of array elements in a simple mathematical statement. A loop steps through each element of the array, using the elements to calculate the gross of five employees.

Program 8-4

```
// This program stores, in an array, the hours worked by
// employees who all make the same hourly wage.
#include <iostream.h>
#include <iomanip.h>

int main()
{
constint NUM_EMPLOYEES = 5;
int hours[NUM_EMPLOYEES];
doublepayrate;

   // Input the hours worked.
cout<< "Enter the hours worked by ";
cout<< NUM_EMPLOYEES << " employees who all\n";
cout<< "earn the same hourly rate.\n";
for (int index = 0; index < NUM_EMPLOYEES; index++)
   {
cout<< "Employee #" << (index + 1) << ": ";
cin>> hours[index];
   }

   // Input the hourly rate for all employees.
cout<< "Enter the hourly pay rate for all the employees: ";
cin>>payrate;

   // Display each employee's gross pay.
cout<< "Here is the gross pay for each employee:\n";
cout<< fixed <<showpoint<<setprecision(2);
for (index = 0; index < NUM_EMPLOYEES; index++)
   {
doublegrossPay = hours[index] * payrate;
cout<< "Employee #" << (index + 1);
cout<< ": $" <<grossPay<<endl;
   }
return 0;
}
```

```
Program Output with Example Input Shown in Bold
Enter the hours worked by 5 employees who all
earn the same hourly rate.
Employee #1 : 5 [Enter]
Employee #2 :10[Enter]
Employee #3 :15[Enter]
Employee #4 :20[Enter]
Employee #5 :40[Enter]
Enter the hourly pay rate for all the employees: 12.75 [Enter]
Here is the gross pay for each employee:
Employee #1 :$63.75
Employee #2 :$127.50
Employee #3 :$191.25
Employee #4 :$255.00
Employee #5 :$510.00
```

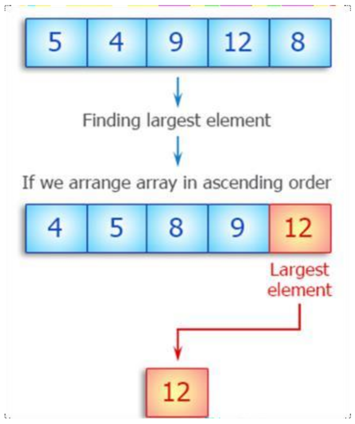
Instructions: Find the true or false statement below. Then, rewrite the remaining false statements so they are true.

1. The valid indexes for the array show, int myArray[20]; is 0-19.
2. The correct way to create an array named markScore that will hold 5 mark scores, is float markScore[5].
3. A two dimension array can also thought of as A table and an array of arrays.
4. The declare an array of 5 characters, and initializes them to some known values, is char array[5]={‘a’,’b’,’c’,’d’,’e’};

Quiz Yourself Online: Do Self Test in the E-Learning

ACTIVITY

Write a C++ program to find the largest elements of a given array of integers.



KEY TERM

Element

Array contents

Array initialization

Subscripts

Inputting Array contents

Outputting Array Contents

Enough memory

SUMMARY

- ☐ An array allow to store and work with multiple values of the same data type.
- ☐ Elements of an array assign unique subscripts which use to access the elements.
- ☐ Declare the use arrays of characters.
- ☐ Use both array notation to access elements of an array.

REFERENCEES

Tony Gaddis, Starting Out with C++ - Early Objects (10th edition) ,Pearson, 2020.