|  |
| --- |
| **FURTHER EDUCATION AND TRAINING CERTIFICATE: INFORMATION TECHNOLOGY: SYSTEMS DEVELOPMENT**  **ID 78965 LEVEL 4 – CREDITS 165** |
| **LEARNER GUIDE**  **SAQA: 14910**  **APPLY THE PRINCIPLES OF COMPUTER PROGRAMMING** |

**Learner Information:**

|  |  |
| --- | --- |
| **Details** | **Please Complete this Section** |
| Name & Surname: |  |
| Organisation: |  |
| Unit/Dept: |  |
| Facilitator Name: |  |
| Date Started: |  |
| Date of Completion: |  |

**Copyright**

All rights reserved. The copyright of this document, its previous editions and any annexures thereto, is protected and expressly reserved. No part of this document may be reproduced, stored in a retrievable system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise without the prior permission.

# Key to Icons

The following icons may be used in this Learner Guide to indicate specific functions:

|  |  |
| --- | --- |
| **http://www.duluth.lib.mn.us/Images/BookStack.gif**  **Books** | **This icon means that other books are available for further information on a particular topic/subject.** |
| http://www.rpsrelocation.com/_borders/checklist.jpg  **References** | **This icon refers to any examples, handouts, checklists, etc…** |
| http://www.school-portal.co.uk/GroupDownloadAttachment.asp?GroupId=21353&AttachmentID=1300079**Important** | **This icon represents important information related to a specific topic or section of the guide.** |
| **http://cloud.graphicleftovers.com/11976/item34004/Cartoon-exercise-book.jpgActivities** | **This icon helps you to be prepared for the learning to follow or assist you to demonstrate understanding of module content. Shows transference of knowledge and skill.** |
| http://3.bp.blogspot.com/_0EodaYtqevU/TMun5XOj03I/AAAAAAAAAIU/lzrnWelQjgc/s1600/group-discussion.jpg**Exercises** | **This icon represents any exercise to be completed on a specific topic at home by you or in a group.** |
| **http://edtech.kennesaw.edu/intech/images/rubric.gif**  **Tasks/Projects** | **An important aspect of the assessment process is proof of competence. This can be achieved by observation or a portfolio of evidence should be submitted in this regard.** |
| **http://tell.fll.purdue.edu/JapanProj/FLClipart/Adjectives/busy.gifWorkplace Activities** | **An important aspect of learning is through workplace experience. Activities with this icon can only be completed once a learner is in the workplace** |
| http://blog.mindjet.com/wp-content/uploads/2010/01/helpful_tips_image.jpg**Tips** | **This icon indicates practical tips you can adopt in the future.** |
| http://school.discoveryeducation.com/clipart/images/read.gif**Notes** | **This icon represents important notes you must remember as part of the learning process.** |

# Learner Guide Introduction

|  |  |
| --- | --- |
| **About the Learner Guide…** | This Learner Guide provides a comprehensive overview of the **Apply the principles of Computer Programming,**and forms part of a series of Learner Guides that have been developed for **FURTHER EDUCATION AND TRAINING CERTIFICATE: INFORMATION TECHNOLOGY: SYSTEMS DEVELOPMENT ID 78965 LEVEL 4 – CREDITS 165**The series of Learner Guides are conceptualized in modular’s format and developed **FURTHER EDUCATION AND TRAINING CERTIFICATE: INFORMATION TECHNOLOGY: SYSTEMS DEVELOPMENT ID 78965 LEVEL 4 – CREDITS 165** They are designed to improve the skills and knowledge of learners, and thus enabling them to effectively and efficiently complete specific tasks. Learners are required to attend training workshops as a group or as specified by their organization. These workshops are presented in modules, and conducted by a qualified facilitator. |
| **Purpose** | The purpose of this Unit Standard is to **Apply the principles of Computer Programming** |
| **Outcomes** | **Apply the principles of Computer Programming** |
| **Assessment Criteria** | The only way to establish whether a learner is competent and has accomplished the specific outcomes is through an assessment process. Assessment involves collecting and interpreting evidence about the learner’s ability to perform a task. This guide may include assessments in the form of activities, assignments, tasks or projects, as well as workplace practical tasks. Learners are required to perform tasks on the job to collect enough and appropriate evidence for their portfolio of evidence, proof signed by their supervisor that the tasks were performed successfully. |
| **To qualify** | To qualify and receive credits towards the learning programme, a registered assessor will conduct an evaluation and assessment of the learner’s portfolio of evidence and competency |
| **Range of Learning** | This describes the situation and circumstance in which competence must be demonstrated and the parameters in which learners operate |
| **Responsibility** | The responsibility of learning rest with the learner, so:   * Be proactive and ask questions, * Seek assistance and help from your facilitators, if required. |

Learning Unit1

**UNIT STANDARD NUMBER :** 14910

**Apply the principles of Computer Programming**

**LEVEL ON THE NQF :** 4

**CREDITS :** 8

**FIELD :** Physical, Mathematical, Computer and Life Sciences

**SUB FIELD :**  Construction Information Technology and Computer Sciences

|  |  |
| --- | --- |
| **PURPOSE:** | This unit standard is intended:  to demonstrate the application of knowledge of the areas covered  for those working in, or entering the workplace in the area of systems development.  People credited with this unit standard are able to:  apply fundamental principles of problem analysis  demonstrate an understanding of different data representations used in computer programs  demonstrate an understanding of fundamental programming principles  demonstrate an understanding of high level programming language concepts  The performance of all elements is to a standard that allows for further learning in this area. |
| **LEARNING ASSUMED TO BE IN PLACE:** | |
| Open.  The credit value of this unit is based on a person having the prior knowledge and skills to:  demonstrate an understanding of fundamental mathematics and English (at least NQF level 2)  demonstrate PC competency skills (End User Computing unit standards)  describe the principles of Computer Programming. | |

|  |
| --- |
| **SESSION 1.**  **Operate computer programming development tools.** |
| **Learning Outcomes** |
| * 1. The operation demonstrates the use of the editor of the development tools to produce program source code. * 2. The operation includes the use of the syntax checker of the tools to check for syntax errors. * 3. The operation uses the tool to compile the program source code produced. |

**The operation demonstrates the use of the editor of the development tools to produce program source code.**

A **programming tool or software development tool** is a program or application that software developers use to create, debug, maintain, or otherwise support other programs and applications. The term usually refers to relatively simple programs, that can be combined together to accomplish a task, much as one might use multiple hand tools to fix a physical object. Sometimes called **text editor**, a program that enables you to create and edit text files. There are many different types of editors, but they all fall into two general categories: line editors: A primitive form of editor that requires you to specify a specific line of text before you can make changes to it. screen -oriented editors: Also called full-screen editors, these editors enable you to modify any text that appears on the display screen by moving the cursor to the desired location.

**EDITOR COMMANDS**

|  |  |
| --- | --- |
| **Command** | **Description** |
| Ctrl-a [Home] | Moves the cursor to the beginning of the current line. |
| Ctrl-b [Left Arrow] | Moves the cursor backwards one character. |
| Ctrl-c | Copys highlighted text (the current selection) to a temporary holding area. |
| Ctrl-d ( [Delete] on Windows ) | Deletes the character to the right of the cursor. |
| Ctrl-e [End] | Moves the cursor to the end of the current line. |
| Ctrl-f | Find a sequence of characters. A prompt bar pops up for entering the desired sequence of characters. An [Esc] aborts the find operation. |
| Ctrl-g | Find the next occurance of a sequence of characters, specified by last FIND or SEARCH. |

**The operation includes the use of the syntax checker of the tools to check for syntax errors.**

In computer science, **a syntax error** refers to an error in the syntax of a sequence of characters or tokens that is intended to be written in a particular programming language. For compiled languages syntax errors occur strictly at compile-time. A program will not compile until all syntax errors are corrected. For interpreted languages, however, not all syntax errors can be reliably detected until run-time, and it is not necessarily simple to differentiate a syntax error from a semantic error; many don't try at all. In 8-bit home computers that used BASIC interpreter as their primary user interface, the SYNTAX ERROR message became somewhat notorious, as this was the response to any command or user input the interpreter couldn't parse. A syntax error may also occur when an invalid equation is entered into a calculator. This can be caused, for instance, by opening brackets without closing them, or less commonly, entering several decimal points in one number.

**Syntax Checking Error Messages**

These messages are output when the compiler is checking your COBOL program for syntax and consistency. The descriptions for each message lists the text of each message, and where necessary explain the error or problem that causes the message and gives advice on how to prevent it. The severity is not listed, as the same message can be output with a different severity depending on the setting of directives.

**Format of Syntax Checking Error Messages**

Syntax checking error messages have the following format:

*Line-of-COBOL-code*

*nnnn-s* *code*\*\*\*\* (*mmmm*)\*\*

*message*

where the variables are:

|  |  |
| --- | --- |
| *nnnn* | The message number. |
| *mmmm* | The page where the previous error occurred. |
| *s* | One of the following severity codes:   |  |  | | --- | --- | | U | Unrecoverable. An unrecoverable error stops the COBOL system. These messages are produced by the run-time system. | | S | Severe. You must correct the syntax error or inconsistency in your program. Otherwise the compiler cannot generate code. | | E | Error. The compiler will make an assumption about what you meant. You might want to correct your program in case the compiler's assumption is not what you intended. | | W | Warning. This means there might be an error, although the program is syntactically correct. | | I | Information. This draws your attention to something in the source code that you might need to be aware of. It does not mean there is an error. | |

You can disable reporting of errors of E-level, W-level, and I-level, using the WARNING directive. When the Compiler has finished, the total number of errors in each category is also output. You can disregard some levels of errors and continue working. You can:

* Debug programs that have S-level, E-level, W-level, and I-level errors regardless of the setting of the E run-time switch.
* Produce object code from intermediate code that has E-level, W-level, and I-level errors, but not S-level errors.
* Run programs that have E-level, W-level, and I-level errors. If the E-level run-time switch is on, which overrides the default setting, you can also run programs with S-level errors.

The error messages can contain variable information. This information is indicated as an item in italics. For example:

User-name *data-name* not unique

will have the name of the item that is not unique in place of the text *data-name*.

**List of Syntax Checking Error Messages**

**0001 Undefined error. Inform Technical Support**

Your program contains an error which the COBOL system has failed to recognize.

**Resolution:**  
Send Technical Support a copy of your source code to enable them to find the cause of the error.

**0002 Unexpected SQL error. Inform Technical Support**

Your program contains an SQL error which the COBOL system has failed to recognize.

**Resolution:**  
Send Technical Support a copy of your source code to enable them to find the cause of the error.

**0003 Illegal format: Literal**

The sequence of characters forming a literal in your source code does not conform to the rules governing the construction of such names. A literal can be either nonnumeric or numeric. If numeric it can be up to 18 digits in length, but it must not contain more than one sign character or more than one decimal point. A nonnumeric literal can consist of any allowable character in the computer's character set up to a maximum of 160 characters in the Procedure Division, or 2048 characters in the Data Division. A nonnumeric literal must be enclosed in quotation marks. If you have used a figurative constant as the literal make sure that it is referenced by an allowable reserved word (such as ZERO) which you have spelled correctly. A figurative constant and a numeric literal must not be enclosed in quotation marks. You might also have used the wrong class of literal for the context of the sentence. Alternatively, if you have used the figurative constant ALL in your code, you have not coded it in accordance with the rules governing the use of this constant. ALL must be followed by a nonnumeric literal and not by a numeric one.

**Resolution:**  
Revise your code to comply with the above rules.

**0004 Illegal character**

Your program contains a character that is not part of the COBOL language set.

**Resolution:**  
Replace the illegal character with a valid one.

**0005 User-name *user-name* not unique**

You have given the same user-name without qualification to more than one data item or procedure-name in your source code.

**Resolution:**  
You must rename or qualify the duplicated data items or procedure-names to ensure that uniqueness of reference is achieved.

**0007 specified in column 7 of otherwise blank line**

The indicator area, column 7, contains an illegal character.

**Resolution:**  
Legal characters are \*, D, -, / or space.

**0008 Unknown COPY file *filename* specified**

A file with the name *filename*, specified in conjunction with a COPY statement, cannot be found.

**0009 '.' missing**

Your code does not contain a period in a place where one is expected by the rules of COBOL syntax.

**Resolution:**  
Insert one at the relevant place.

**0010 Word starts or is continued in wrong area of source line**

The word starts either in area A when it should have started in area B, or in area B when it should have started in area A.

**0011 Reserved word missing or incorrectly used**

You have either used a reserved word in a place where a user defined word is expected or you have failed to use a reserved word where one is needed.

**Resolution:**  
Alter the reserved word into a user defined one or insert a reserved word according to the context of this message.

**The operation uses the tool to compile the program source code produced.**

User written code, standard functions, library functions.

**Library Functions:**

Q-Basic provides a number of functions. These inbuilt functions of Q-basic are called library functions. These are divided into string and numeric functions. LEFT$, LEN, MID$, LCASE$ etc are the examples of string functions and ABS, SQR, INT, VAL etc are the examples of numeric variables.

**User Defined Functions:**

While standard functions are pre-defined and provided for by QBasic, user-defined functions are completely defined and customized by the programmer. User-defined functions return a single value and are generally used to perform an operation that will be needed numerous times in a program. In QBasic, user-defined functions are referred to as procedures; similar to SUB procedures except function procedures return one value. Arguments may be sent into a function procedure for use in the function operation, but the value returned by the function will not be included in the parameter list. The value is returned in the function itself. Each user=defined function starts with BEGIN FUNCTION FunctName (x,y,z) and ends with END FUNCTION. The code between these two lines is executed whenever the function is invoked from main program, from another function, or SUB, or from itself. FunctName is a name for your function (choose a descriptive one). Arguments (x,y,z) are the variables passed to the function.The form of a function procedure is as follows:

**FUNCTION name( parameter list )**

**REM**

**REM body of function**

**REM**

**END FUNCTION**

**Subroutines and functions:**

A subroutine (also called a "module") is a "mini-program" inside your program. In other words, it is a collection of commands--and can be executed anywhere in your program. To create a subroutine: Go to the "Edit" menu Select "New Sub" Enter a name for the subroutine Type a list of commands between SUB and END SUB. (Topic 1.1 SUB ….. END SUB statement: will provide detail information)

**Functions:**

Function is the same as a subroutine, except it returns a value. Also, you must leave out the CALL command. To return a value, set a variable with the same name as the function.

**Local and Global variable:**

When a variable is declared within a main module or procedure without using SHARED attribute, only code within that main module or procedure can access or change the value of that variable. This type of variable is called as LOCAL variable. When a variable is declared with SHARED attribute in a main module, it can be used in a procedure without passing it as parameter. Any SUB or FUNCTION procedure within the module can use this type of variable. This type of variable which is available to all SUB and FUNCTION procedure with the module is known as GLOBAL variable.

|  |
| --- |
| **SESSION 2.**  **Demonstrate an understanding of different data representations used in computer programs.** |
| **Learning Outcomes** |
| * 1. The demonstration applies different number conversion techniques between data types (at least 2). * 2. The demonstration compares different logical data types (at least 3) in a language of choice (incl. pseudo code). * 3. The demonstration differentiate between different internal representations of data types (in ASCII). * 4. The demonstration distinguishes between different logical operators (at least 2). |

**The demonstration applies different number conversion techniques between data types (at least 2).**

There are infinite ways to represent a number. The four commonly associated with modern computers and digital electronics are: decimal, binary, octal, and hexadecimal.

**Decimal** (base 10) is the way most human beings represent numbers. Decimal is sometimes abbreviated as dec.

Decimal counting goes:  
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, and so on.

**Binary** (base 2) is the natural way most digital circuits represent and manipulate numbers. (Common misspellings are “bianary”, “bienary”, or “binery”.) Binary numbers are sometimes represented by preceding the value with '0b', as in 0b1011. Binary is sometimes abbreviated as bin.

Binary counting goes:  
0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111, 10000, 10001, and so on.

**Octal** (base 8) was previously a popular choice for representing digital circuit numbers in a form that is more compact than binary. Octal is sometimes abbreviated as oct.

Octal counting goes:  
0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 20, 21, and so on.

**Hexadecimal** (base 16) is currently the most popular choice for representing digital circuit numbers in a form that is more compact than binary. (Common misspellings are “hexdecimal”, “hexidecimal”, “hexedecimal”, or “hexodecimal”.) Hexadecimal numbers are sometimes represented by preceding the value with '0x', as in 0x1B84. Hexadecimal is sometimes abbreviated as hex.

Hexadecimal counting goes:  
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, and so on.

All four number systems are equally capable of representing any number. Furthermore, a number can be perfectly converted between the various number systems without any loss of numeric value.

|  |  |  |  |
| --- | --- | --- | --- |
| Dec | Hex | Oct | Bin |
| 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 | 0 1 2 3 4 5 6 7 8 9 A B C D E F | 000 001 002 003 004 005 006 007 010 011 012 013 014 015 016 017 | 00000000 00000001 00000010 00000011 00000100 00000101 00000110 00000111 00001000 00001001 00001010 00001011 00001100 00001101 00001110 00001111 |

**The demonstration differentiates between different internal representations of data types (in ASCII).**

Data Representation refers to the methods used internally to represent information stored in a computer. Computers store lots of different types of information:

* numbers
* text
* graphics of many varieties (stills, video, animation)
* sound

At least, these all seem different to us. However, ALL types of information stored in a computer are stored internally in the same simple format: a sequence of 0's and 1's. *How can a sequence of 0's and 1's represent things as diverse as your photograph, your favorite song, a recent movie, and your term paper?* It all depends on how we *interpret* the information. Computers use numeric codes to represent all the information they store. These codes are similar to those you may have used as a child to encrypt secret notes: let 1 stand for A, 2 stand for B, etc. With this code, any written message can be represented numerically. The codes used by computers are a bit more sophisticated, and they are based on the binary number system (base two) instead of the more familiar (for the moment, at least!) decimal system. Computers use a variety of different codes. Some are used for numbers, others for text, and still others for sound and graphics.

**Memory Structure in Computer**

* Memory consists of bits (0 or 1)
  + a single bit can represent two pieces of information
* bytes (=8 bits)
  + a single byte can represent 256 = 2x2x2x2x2x2x2x2 = 28 pieces of information
* words (=2,4, or 8 bytes)
  + a 2 byte word can represent 2562 pieces of information (approximately 65 thousand).
* Byte addressable - each byte has its own address.

**Binary Numbers**

Normally we write numbers using digits 0 to 9. This is called base 10. However, any positive integer (whole number) can be easily represented by a sequence of 0's and 1's. Numbers in this form are said to be in base 2 and they are called binary numbers. Base 10 numbers use a positional system based on powers of 10 to indicate their value. The number 123 is really 1 hundred + 2 tens + 3 ones. The value of each position is determined by ever-higher powers of 10, read from left to right. Base 2 works the same way, just with different powers. The number 101 in base 2 is really 1 four + 0 twos + 1 one (which equals 5 in base 10).

**Text**

Text can be represented easily by assigning a unique numeric value for each symbol used in the text. For example, the widely used ASCII code (American Standard Code for Information Interchange) defines 128 different symbols (all the characters found on a standard keyboard, plus a few extra), and assigns to each a unique numeric code between 0 and 127. In ASCII, an "A" is 65," B" is 66, "a" is 97, "b" is 98, and so forth. When you save a file as "plain text", it is stored using ASCII. ASCII format uses 1 byte per character 1 byte gives only 256 (128 standard and 128 non-standard) possible characters The code value for any character can be converted to base 2, so any written message made up of ASCII characters can be converted to a string of 0's and 1's.

**Graphics**

Graphics that are displayed on a computer screen consist of pixels: the tiny "dots" of color that collectively "paint" a graphic image on a computer screen. The pixels are organized into many rows on the screen. In one common configuration, each row is 640 pixels long, and there are 480 such rows. Another configuration (and the one used on the screens in the lab) is 800 pixels per row with 600 rows, which is referred to as a "resolution of 800x600." Each pixel has two properties: its location on the screen and its color. A graphic image can be represented by a list of pixels. Imagine all the rows of pixels on the screen laid out end to end in one long row. This gives the pixel list, and a pixel's location in the list corresponds to its position on the screen. A pixel's color is represented by a binary code, and consists of a certain number of bits. In a monochrome (black and white) image, only 1 bit is needed per pixel: 0 for black, 1 for white, for example. A 16 color image requires 4 bits per pixel. Modern display hardware allows for 24 bits per pixel, which provides an astounding array of 16.7 million possible colors for each pixel!

**The demonstration compares different logical data types (at least 3) in a language of choice (incl. pseudo code).**

**Numeric Data**

Numeric data simply means **numbers**. But, just to complicate things for you, numbers come in a variety of different **types**...

**Integers**

An integer is a **whole number** - it has **no decimal or fractional parts**. Integers can be either **positive** or **negative**. *Examples*

* 12
* 45
* 1274
* 1000000
* -3
* -5735



**Real Numbers**

Any number that you could place on a number line is a real number. Real numbers include **whole numbers**(integers) and **numbers with decimal/fractional parts**. Real numbers can be **positive** or **negative**.

*Examples*

* 1
* 1.4534
* 946.5
* -0.0003
* 3.142



*Some computer software used strange names for real data.   
  
You might see this data type referred to as '****single****', '****double****' or '****float****'.*

**Currency**

Currency refers to **real** numbers that are **formatted** in a specific way. Usually currency is shown with a **currency symbol** and (usually) **two decimal places**.

*Examples*

* £12.45
* -£0.01
* €999.00
* $5500



**Percentage**

Percentage refers to **fractional real** numbers that are formatted in a specific way - **out of 100**, with a **percent symbol**. So, the real value **0.5** would be shown as **50%**, the value **0.01** would be shown as**1%** and the number **1.25** would be shown as **125%**

*Examples*

* 100%
* 25%
* 1200%
* -5%



*Inside the computer the 50% is stored as a****real****number: 0.5, But when it is displayed it is shown****formatted****as a percentage*

**Alphanumeric (Text) Data**

Alphanumeric (often simply called 'text') data refers to data made up of **letters** (alphabet) and **numbers** (numeric). Usually **symbols** ($%^+@, etc.) and spaces are also allowed.

*Examples*

* DOG
* “A little mouse”
* ABC123



*Text data is often input to a computer with****speech marks****(". . .") around it:*

*"MONKEY"*

*These tell the computer that this is text****data****and not some special command.*

**Date and Time Data**

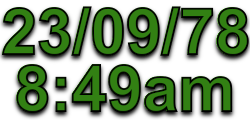
Date (and time) data is usually **formatted** in a specific way. The format depends upon the **setup** of the computer, the software in use and the user’s **preferences**.

*Date Examples*

* 25/10/2007
* 12 Mar 2008
* 10-06-08

*Time Examples*

* 11am
* 15:00
* 3:00pm
* 17:05:45



*With inputting dates particular care has to be taken if the data contains* ***American****style dates and the computer is setup to expect* ***international****style dates (or vice-versa)...  
The date 06/09/08 refers to 6th September 2008 in the international system, but would be 9th June 2008 in America! Check your computer’s settings.*

**Boolean (Logical) Data**

Boolean data is sometimes called 'logical' data (or in some software, 'yes/no' data). Boolean data can only have two values: **TRUE** or **FALSE**

Examples

* TRUE
* FALSE
* ON
* OFF
* YES
* NO



*Note that TRUE and FALSE can also be shown as****YES****/****NO****,****ON****/****OFF****, or even graphically as****tick boxes****(ticked / unticked)*

**Selecting Data Types**

When we are presented with data to be input into a computer system, we must analyse it and select **appropriate data types** for each value... e.g. For the following data, we might use the date types shown:

**Data Name**

* Name
* Height
* Date of Birth
* Phone No.
* Pay Rate
* Tax Rate

**Data Type**

* **Text**
* **Real**
* **Date**
* **Alphanumeric**
* **Currency**
* **Percentage**

**The demonstration distinguishes between different logical operators (at least 2).**

And, Or, Not.

Logical operators are typically used with Boolean (logical) values; when they are, they return a Boolean value. However, the && and ||operators actually return the value of one of the specified operands, so if these operators are used with non-Boolean values, they may return a non-Boolean value.

**Logical Operators**

Common Lisp provides three operators on Boolean values: and, or, and not. Of these, and or are also control structures because their arguments are evaluated conditionally. The function not necessarily examines its single argument, and so is a simple function. The logical operators are described in the following table:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Usage** | **Description** |
| Logical AND (&&) | *expr1* &&*expr2* | Returns expr1 if it can be converted to false; otherwise, returns expr2. Thus, when used with Boolean values, && returns true if both operands are true; otherwise, returns false. |
| Logical OR (||) | *expr1* ||*expr2* | Returns expr1 if it can be converted to true; otherwise, returns expr2. Thus, when used with Boolean values, || returns true if either operand is true; if both are false, returns false. |
| Logical NOT (!) | !*expr* | Returns false if its single operand can be converted to true; otherwise, returns true. |

Fortran has five **LOGICAL** operators that can only be used with expressions whose results are logical values (*i.e.*, **.TRUE.** or **.FALSE.**). All **LOGICAL** operators have priorities lower than *arithmetic* and *relational* operators. Therefore, if an expression involving arithmetic, relational and logical operators, the arithmetic operators are evaluated first, followed by the relational operators, followed by the logical operators.

These five logical operators are

* **.NOT.** : logical **not**
* **.AND.** : logical **and**
* **.OR.** : logical **or**
* **.EQV.** : logical **equivalence**
* **.NEQV.** : logical **not equivalence**

|  |
| --- |
| **SESSION 3.**  **Demonstrate an understanding of fundamental programming principles.** |
| **Learning Outcomes** |
| * 1. The demonstration illustrates the differences between the various algorithmic structures of programming languages, using a language of choice (incl. Pseudo code). * 2. The demonstration compares good & bad program documentation principles (at least 3), using a language of choice (incl. Pseudo code) where needed. * 3. The demonstration illustrates good programming quality assurance principles. |

**The demonstration illustrates the differences between the various algorithmic structures of programming languages, using a language of choice (incl. Pseudo code).**

An **algorithm** is the steps that a programmer will write that will become a program. It is written in a form of structured language called**Pseudocode**. Once an algorithm is created using pseudocode, it is simple to translate it into real programming code. It gives an opportunity to detect any logic errors prior to actual coding, which is a lot more expensive and time consuming. It is used for planning the programming. Pseudocode consists of short, English phrases used to explain specific task within a program’s algorithm. Pseudocode should not include keywords in any specific computer languages. It should be written as a list of consecutive phrases. You should not use flowcharting symbols but you can draw arrows to show looping processes. Indentation can be used to show the logic in pseudocode as well. For example, a first-year, 9th grade Visual Basic programmer should be able to read and understand the pseudocode written by a 12th grade AP Data Structures student. In fact, the VB programmer could take the other students pseudocode and generate a VB program based on that pseudocode.

The programming process is a complicated one. You must first understand the program specifications, of course. Then you need to organise your thoughts and create the program. This is a difficult task when the program is not trivial (i.e. easy). You must break the main tasks that must be accomplished into smaller ones in order to be able to eventually write fully developed code. Writing pseudocode WILL save you time later during the construction & testing phase of a program’s development.

Keep in mind that, if you have used pseudocode to write your algorithm, the coding will become very simple.

|  |  |  |
| --- | --- | --- |
|  | |  | | --- | | **Pseudocode** Pseudocode is an outline of a program, written in a form that can easily be converted into real programming statements. | |

**The *sequence* structure**

We have been using the *sequence* structure since early in the course.  Basically we can describe the sequence structure using the pseudocode shown in Figure 1.

**Figure 1. The sequence structure in pseudocode.**

|  |
| --- |
| Enter  Perform one or more actions in sequence  Exit |

Thus, the general requirement for the sequence structure is that one or more actions may be performed in sequence after entry and before exit. There may not be any branches or loops between the entry and the exit.

All actions must be taken in sequence.

**The action elements themselves may be structures**

However, it is important to note that one or more of the action elements may themselves be sequence, selection, or loop structures. If each of the structures that make up the sequence has only one entry point and one exit point, each such structure can be viewed as a single action element in a sequence of actions. Obviously, the sequence structure is the simplest of the three.

**The *selection* structure**

The *selection* or *decision* structure can be described as shown in the pseudocode **The selection structure in pseudocode.**

|  |
| --- |
| Enter  Test a condition for true or false  On true  Take one or more actions in sequence  On false  Take none, one, or more actions in sequence  Exit |

**Test a condition for true or false**

Once again, there is only one entry point and one exit point. The first thing that happens following entry is that some condition is tested for true or false. If the condition is true, one or more actions are taken in sequence and control exits the structure. If the condition is false, **none**, one or more different actions are taken in sequence and control exits the structure.  *(Note the inclusion of the word none here.)*

**The action elements may themselves be structures**

Once again, each of the action elements in the sequence may be another sequence, selection, or loop structure. Eventually all of the actions for a chosen branch will be completed in sequence and control will exit the structure.

**Sometimes no action is required on false**

It is often the case that no action is required when the test returns false.  In that case, control simply exits the structure without performing any actions.

**The *loop* structure**

The *loop* or *iteration* structure can be described as shown in the pseudocode

**The loop structure in pseudocode.**

|  |
| --- |
| Enter  Test a condition for true or false  Exit on false  On true  Perform one or more actions in sequence.  Go back and test the condition again |

As before, there is only one entry point and one exit point.

**Perform the test and exit on false**

The first thing that happens following entry is that a condition is tested for true or false.

If the test returns false, control simply exits the structure without taking any action at all.

**Perform some actions and repeat the test on true**

If the test returns true:

* One or more actions are performed in sequence.
* The condition is tested again.

During each iteration, if the test returns false, control exits the structure.  If the test returns true, the entire cycle is repeated.

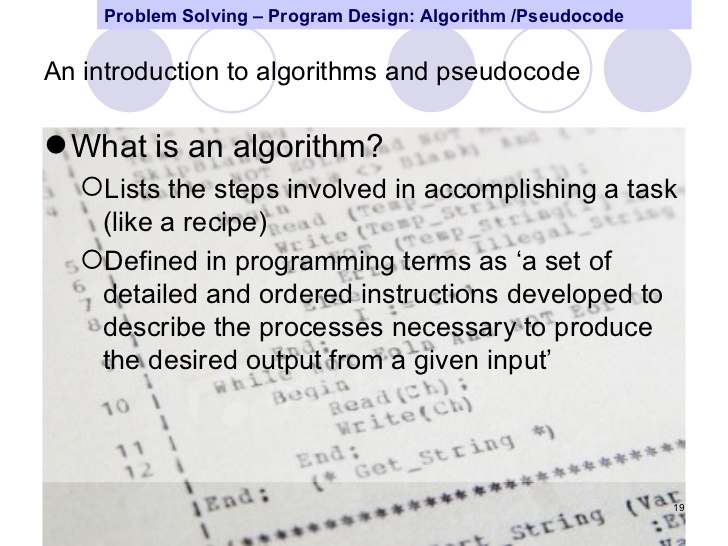
**Each action element may be another structure**

Each of the action elements may be implemented by another sequence, selection, or loop structure. Eventually all of the actions will be completed and the condition will be tested again.

**Need to avoid infinite loops**

Generally speaking, unless something is done in one of the actions to cause the test to eventually return false, control will never exit the loop.

In this case, the program will be caught in what is commonly referred to as an *infinite loop.*



**The demonstration compares good & bad program documentation principles (at least 3), using a language of choice (incl. Pseudo code) where needed.**

**Documenting the Program**

Documenting is an ongoing, necessary process, although, as many programmers are, you may be eager to pursue more exciting computer-centered activities. Documentation is a written detailed description of the programming cycle and specific facts about the program. Typical program documentation materials include the origin and nature of the problem, a brief narrative description of the program, logic tools such as flowcharts and pseudocode, data-record descriptions, program listings, and testing results. Comments in the program itself are also considered an essential part of documentation. Many programmers document as they code. In a broader sense, program documentation can be part of the documentation for an entire system.

The wise programmer continues to document the program throughout its design, development, and testing. Documentation is needed to supplement human memory and to help organize program planning. Also, documentation is critical to communicate with others who have an interest in the program, especially other programmers who may be part of a programming team. And, since turnover is high in the computer industry, written documentation is needed so that those who come after you can make any necessary modifications in the program or track down any errors that you missed.

**The demonstration illustrates good programming quality assurance principles**

**Quality Information Systems: Vital to Total Quality Management**

Heightened global competition has made it imperative for organizations to deliver products - goods and services - of consistently high quality. The principles of total quality management (TQM) recognize that consistent product quality results from designing and executing business processes so as to remove error and waste.

**Product quality is the result of:**

a. Customer focus

b. Introduction and continuous improvement of business processes and product-development processes that reduce variations

c. Creation of a company-wide quality culture through motivation and training of all its members

d. Continuous measurement and analysis of the accomplished results.

**Quality information systems are vital to total quality management, because:**

1. Business processes of the firm depend on information systems and, therefore, their quality depends to a large degree on IS quality

2. Information systems enable most projects of business process design. This IS enabled streamlining of processes gives fewer opportunities for error, thus leading to higher quality of the processes' outputs.

3. Information systems are a necessary component of the feedback loop in managing an enterprise. IS are necessary to continually gauge any deviations from the expected norms in the firm's performance and thus help reduce variance in performance.

International Standards Organization (ISO) 9000 - many companies, particularly in the manufacturing sectors, comply with the ISO 9000 group of quality standards. Such compliance is mandatory for those selling any of a broad range of products to the countries of the European Union. Some other countries also require this certification. The standards aim to ensure quality of products by certifying quality assurance during business processes, such as product design, manufacturing, delivery, and service support. Extensive quality-oriented information processing is a prerequisite for a certification of compliance.

**Software Quality**

There are many attributes of software quality. These include:

1. Effectiveness

2 Usability

3. Efficiency

4. Reliability

5. Maintainability

6. Understandability

7. Modifiability

8. Testability

**Effectiveness** Refers to the satisfaction of the user and organizational requirements as established during an analysis of these requirements, possibly using prototyping.

**Usability** The ease with which the intended users can use the system, depend on the proper user-system interface.

**Efficient** operation is reflected mainly in how economically hardware resources are used to satisfy the given effectiveness requirements

**Reliability** Refers to the probability that the information system will operate correctly; that is, according to specifications over a period of time. It may also be defined as the mean time between failures. Software reliability is rooted in its freedom from defects. If a system must run on different hardware or systems software platforms, portability should be included as a desired attribute.

**Maintainability** Refers to ease of understanding, modifying, and testing.

**Understandability** Is achieved by readable and well-commented system code and by documentation, which includes the requirements specifications, system documentation, user manuals, and, sometimes special maintenance documentation.

**Modifiability** Means that it is relatively easy to identify and change any part of the system that requires maintenance without affecting its other parts.

**Testability** Is the ease with which we can demonstrate that a modification resulted in a quality system.

|  |
| --- |
| **SESSION 4.**  **Demonstrate an understanding of high level programming language concepts.** |
| **Learning Outcomes** |
| * 1. The demonstration explains what is understood by constants and variables. * 2. The demonstration illustrates the concepts of operators and expressions. * 3. The demonstration illustrates different modular programming features and variable passing. * 4. The demonstration applies different debugging techniques. |

**The demonstration explains what is understood by constants and variables.**

A symbol or name that stands for a value. For example, in the expression

x+y

x and y are variables. Variables can represent numeric values, characters, character strings, or memory addresses. Variables play an important role in computer programming because they enable programmers to write flexible programs. Rather than entering data directly into a program, a programmer can use variables to represent the data. Then, when the program is executed, the variables are replaced with real data. This makes it possible for the same program to process different sets of data. Every variable has a name, called the variable name, and a data type. A variable's data type indicates what sort of value the variable represents, such as whether it is an integer, a floating-point number, or a character. The opposite of a variable is a constant. Constants are values that never change. Because of their inflexibility, constants are used less often than variables in programming.

**The demonstration illustrates the concepts of operators and expressions.**

An ***operator***is a code element that performs an operation on one or more code elements that hold values. Value elements include variables, constants, literals, properties, returns from **Function** and **Operator** procedures, and expressions.

An *expression* is a series of value elements combined with operators, which yields a new value. The operators act on the value elements by performing calculations, comparisons, or other operations.

**Types of Operators**

Visual Basic provides the following types of operators:

* Arithmetic Operators perform familiar calculations on numeric values, including shifting their bit patterns.
* Comparison Operators compare two expressions and return a **Boolean** value representing the result of the comparison.
* Concatenation Operators join multiple strings into a single string.
* Logical and Bitwise Operators in Visual Basic combine **Boolean** or numeric values and return a result of the same data type as the values.

The value elements that are combined with an operator are called *operands* of that operator. Operators combined with value elements form expressions, except for the assignment operator, which forms a *statement*.

The end result of an expression represents a value, which is typically of a familiar data type such as **Boolean**, **String**, or a numeric type.

The following are examples of expressions.

5 + 4

' The preceding expression evaluates to 9.

15 \* System.Math.Sqrt(9) + x

' The preceding expression evaluates to 45 plus the value of x.

"Concat" & "ena" & "tion"

' The preceding expression evaluates to "Concatenation".

763 < 23

' The preceding expression evaluates to False.

Several operators can perform actions in a single expression or statement, as the following example illustrates.

VB

x = 45 + y \* z ^ 2

In the preceding example, Visual Basic performs the operations in the expression on the right side of the assignment operator (**=**), then assigns the resulting value to the variable x on the left. There is no practical limit to the number of operators that can be combined into an expression, but an understanding of Operator Precedence in Visual Basic is necessary to ensure that you get the results you expect.

**The demonstration illustrates different modular programming features and variable passing.**

**Modular Programming**

When the complete program code started from one point and ended in another point is compiled within a single program, the program become very complex and unmanaged which is called as **linear programming**. In making large programs, you may need to repeat parts of the code again and again, but not necessarily from within a loop that does only that section of code. You may also have complex mathematical formulas that use several variables to calculate a single value. If the complex and unmanaged program is broken into the simpler programs to perform different task, then the program become simpler and managed which is called as **modular programming**. SUBs (subroutines) and FUNCTIONS are the tools you can use to organize your code and create mini-programs within your one QBasic program.

**Advantages of Modular Programming:**

* The same procedure can be used more than one without rewriting, so it reduces the length of the program.
* The debugging is much easier since the programs are found in the different parts.
* The procedure can be tested and debugged separately.
* Due to small programs, the task of documentation is easier.
* The program is much more readable than the linear program.

**MODULE:**

The main program or source file of the sequential program containing the entry point and the ending point of the program and which can be separately compiled is known as module. It may contain the SUB and FUNCTION as well as codes not directly part of the SUB and FUNCTION. The statements that are the not the part of the SUB or FUNCTION are called module level code, for example DIM, TYPE etc. Every program has one special module (the main module) and may consists of many supporting modules

**The demonstration applies different debugging techniques.**

* **Debugging.** A term used extensively in programming, debugging means detecting, locating, and correcting bugs (mistakes), usually by running the program. These bugs are logic errors, such as telling a computer to repeat an operation but not telling it how to stop repeating. In this phase you run the program using test data that you devise. You must plan the test data carefully to make sure you test every part of the program.

**RTFM technique**

*RTFM* stands for *Read The Fine Manual*. Make sure you take the time to find relevant documentation for the task at hand, i.e. the documentation of the tools (not only the compiler, but also **make**, the preprocessor and the linker), libraries and algorithms you are expected to use, such as [CPP][GCC][MAKE]. Often you do not need to know everything in the documentation, but you do need to be aware what documentation is relevant and what its purpose is. You should at the very least browse through it; hopefully this will give you a feeling of *deja-vu* where needed, so you know where to look.In examining documentation, you should distinguish between tutorials and reference documentation.

**print() debugging**

*printf debugging* is our term for a debugging technique we encounter all too often. It consists of ad hoc addition of lots of printf (C) or cerr or cout (C++) statements to track the control flow and data values in the execution of a piece of code.

This technique has strong disadvantages:

* It is very ad hoc. Code is temporarily added, to be removed as soon as the bug at hand is solved; for the next bug, similar code is added etc. There are better ways of adding debugging information, as you shall see shortly.
* It clobbers the normal output of your program, and slows it down considerably.

**ANWB debugging**

`ANWB debugging' is based on a simple principle: the best way to learn things is to teach them.

In `ANWB debugging' you find a, preferably innocent and willing, bystander and explain to her how your code works  . This forces you to rethink your assumptions, and explain what is really happening; often you find the cause of your problems this way.