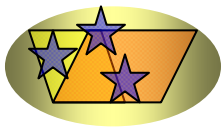
Create Designs
Level 2
Notes

Tench Computing Ltd
Pines
Glendale Road
Burgess Hill
West Sussex
RH15 0EJ

Web address: www.tenchcomputing.co.uk
Email address: jtench@globalnet.co.uk



About the author: *Jackie Tench MSc, ACIB, Cert Ed*

Jackie started her working career in branch banking with the Midland Bank (now HSBC) and was transferred to their Computing Department after achieving 100% in their ability test for programmers. She then worked for more than a decade in this department and was one of the first women to achieve a junior management grade at the age of 21. She attended a significant number of IBM programming training courses during her time there.

Jackie was the first woman to pass the ACIB (Associate Chartered Institute of Bankers) examinations in the Midland Bank (HSBC) and the youngest person at 21 years of age.

Jackie then left to raise a family but still found time to teach part-time at a college in Sheffield and to obtain a MSc in Computing and a Cert Ed in teaching.

When her children were old enough Jackie returned to work full-time and was a Senior Lecturer in Software Engineering and Computer Studies at a college in Brighton for nearly 10 years teaching all levels up to and including HND.

Therefore, Jackie has considerable business knowledge and qualifications plus wide experience in practical computing and training – covering areas such as structured design, analysis, coding, testing and implementing software applications plus training students to fulfil an important role in the computer industry.

Jackie has worked as a consultant for several blue chip companies and examination boards using her software engineering and educational training skills and is now one of the foremost experts in computing with an extensive knowledge of programming languages and applications.

Contents Page

High level programming languages

A computer program consists of a set of instructions that tell a computer what to do. The computer only understands instructions in binary. Binary consists of two values a 0 and a 1. It would be very difficult for a programmer to write a program in binary as it would take a long time and be very prone to error, for instance a 0 instead of a 1 in the wrong place and the program would not work correctly. Programs were originally written in binary and also octal, based on the digits 0-7 and hexadecimal based on the digits 0-9 and the letters A-F, which are shorthand versions of binary.

Because of the difficulty of writing programs in binary, octal and hexadecimal, high-level programming languages were developed. High-level programming languages are more readable and therefore faster to develop. High-level programming languages include COBOL, Pascal, Java, C, C++ and Visual Basic. The most readable high level programming language is COBOL which uses English words such as ADD and SUBTRACT as part of its language. The only standardised language is COBOL all the other programming languages e.g. C, have different variations depending on the compiler software supplier.

Syntax

Each programming language has a formal definition that consists of syntax and semantics. Syntax is the set of rules that define how keywords, symbols, expressions and statements can be structured and combined. Semantics is the set of rules that defines what happens when the statements are executed.

Programs written in a high-level programming language are normally compiled into machine code. Machine code is the language that a computer understands and each high-level program instruction is normally translated into several machine code instructions. Every different type of computer has a different machine code set of instructions. A compiler is used to compile the programs and the compiler will signal errors if the program contains errors in the syntax of the programming language. A program cannot be executed until it is free of syntax errors. A program that is executed may still not run correctly because it may contain logical errors. Logical errors are errors made by a programmer in the logic of the programming instructions.

Keywords

Every programming language has a set of keywords that are used in that particular language. Keywords are also called reserved words. Keywords are words like print, write, read and open that are used by a programmer when they write the program instructions. These keywords are what make a programming language high-level. Behind each keyword is a pre-written routine that a programmer is using. This means that a programmer can write one high-level program instruction instead of multiple lines of machine code instructions.

Comments

Programming languages allow comments to be inserted into the program code. Comments are ignored by a compiler and are only used as readable information for a programmer. They are used to help understand what the program instructions mean. A special character or characters are used to denote a comment line in a program. For instance Visual Basic uses the character ' to indicate a comment line whereas C uses the characters // to indicate a single comment line and /* at the start of multiple comment lines and */ at the end. These special characters are recognised by the compiler, which ignores the comment lines.

Indentation

Indentation is used to make program code more understandable by indenting code that forms a block. A compiler ignores indentation.

Variables

Variables are used to hold data while a program is running. Each variable in a program is given a name so that it can be referred to by the program instructions. The data in the variable can change as the program runs. For example, if you were writing a program to read in ten numbers and output a total, a variable named total could be declared. Each time a number was input its value would be added to the variable total until all ten numbers had been read and the value in the total would be output.

Constants

Constants hold values that are needed by a program, but their value never changes while a program is running. Each constant in a program is given a name so that it can be referred to by the program instructions. All values that do not change while a program is running should be defined as constants. For instance if the number for pi, 3.14 was to be used by a program it should be declared as a constant e.g. PI and the program instructions should then refer to it using the name PI. The number 3.14 should not be coded directly into the program instructions. Programmers normally use uppercase characters for constants names to distinguish them from variables whose names are normally in lowercase characters.

Variable and constant names

Meaningful names should be used for variables and constants. Each programming language defines the syntax that can be used for user-defined variable and constant names. There are normally a maximum number of characters that can be used and certain characters cannot be used. Names must not be the same as the keywords that are part of the language. The first character of a name cannot be a digit but there may be other restrictions as well. If a language is case sensitive then the names indx, Indx, INDX are all seen as different. Consistent naming conventions should be used throughout a program. Some companies have house standards for naming constants and variables.

Basic data types

When a variable is defined in a program its data type must also be defined. A data type specifies the type of data that can be held in the variable. The basic data types are integer, floating point, character, string and boolean.

Integer

The integer data type is used for whole numbers. The numbers could be positive or negative. The maximum number (positive or negative), which can be held in an integer data type, depends on the programming language being used.

Floating-point

The floating-point data type is used for numbers, which contain decimal places. Again the maximum number (positive or negative), which can be held in a floating-point data type, depends on the programming language being used.

Character

The character data type is used to store one character.

Strings

Strings are made up of several characters and string literals are enclosed in special characters. Some programming languages use single quotes (') to enclose a string others use double quotes (").
For example

"John Melsham"
'John Melsham'

are different ways of using a string. Most programming languages provide prewritten functions to enable strings to be manipulated. For instance, in the example string above you might need to split the first name John from the surname Melsham. This would be done by using a string function, to find the space between the first name and the surname and then splitting off the first name and surname.

Boolean

The boolean data type can have one of two values, TRUE or FALSE. Not all languages support a boolean data type.

Literals

A literal is a value that is used directly in program code and its value does not change. Literals should be avoided in code and set up as constants instead.

Examples of literals:

Literal	**Type**
"Enter a number"	String literal
250	Numeric literal
23.86	Numeric literal
-304.56	Numeric literal (negative value)
'Y'	Character literal
""	An empty string

Arithmetic operators

Arithmetic operators are used to perform calculations on data within software. The main arithmetic operators used in software are for addition, subtraction, multiplication and division. Programming languages use the same symbols for these operators as shown below:

Operator	Symbol
Addition	+
Subtraction	-
Multiplication	*
Division	/

6 + 2 gives the result 8
10 + 15 – 20 gives the result 5
5 * 4 gives the result 20
32 / 4 gives the result 8

The modulus operator is used to divide one value by another and the result is the remainder after the division. This operator does not have a common symbol between programming languages.

15 modulus 4 gives the result 3
22 modulus 10 gives the result 2

Arithmetic operator precedence

Multiply and divide are evaluated before add and subtract. If there are two operators of the same precedence then evaluation is done from left to right.

Example 1
For an expression with addition and multiplication such as

6 + 5 * 3

The 5 * 3 is performed first

6 + 15

to give the result 21.

Example 2

3 * 6 + 4 * 3 / 2

The 3 * 6 will be calculated first

18 + 4 * 3 / 2

then the 4 * 3

18 + 12 / 2

then the 12 / 2

18 + 6

then the 18 + 6

24

Example 3
The order of precedence can be changed by using brackets.
If it was required to have the 6 + 4 done first then brackets can be used as follows:

3 * (6 + 4) * 3 / 2

The 6 + 4 will be calculated first

3 * 10 * 3 / 2

then the 3 * 10

30 * 3 / 2

then the 30 * 3

90 / 2

then the 90 / 2

45

Example 4
If more than one set of brackets is used the inner brackets are calculated first.

3 * (((3 + 5) * 3) - 2)

The 3 + 5 will be calculated first

3 * ((8 * 3) - 2)

then the 8 * 3

3 * (24 - 2)

then the 24 - 2

3 * 22

then the 3 * 22

66

Program constructs

All software is made up of three fundamental program constructs: sequence, selection and iteration (repetition).

Sequence

A sequence is represented by statements, which are executed one after the other in sequence.

statement1
statement2
statement3
…
…
statementn

Execution starts at statement1, moves on to statement2, then statement3 through all the other statements until the last statement, statementn is executed.

Selection

Selection provides alternative paths for execution depending on the result of a conditional expression. All programming languages implement selection using IF statements.

IF *conditional expression* THEN
    statement(s)
ENDIF

The statements enclosed within the IF…ENDIF block are only executed if the *conditional expression* evaluates to TRUE. Execution continues with the next statement after the IF block.

IF *conditional expression* THEN
    statement(s)1
ELSE
    statement(s)2
ENDIF

The IF…ELSE…ENDIF format provides two alternative paths. If the *conditional expression* evaluates to TRUE statement(s)1 are executed. If the *conditional expression* evaluates to FALSE, statement(s)2 are executed. Execution continues with the next statement after the IF block.

IF statements can be nested inside an IF statement.

IF *conditional expression1* THEN
    statement(s)1
ELSE IF *conditional expression2*
    statement(s)2
    ELSE
        statement(s)3
    ENDIF
ENDIF

If *conditional expression1* evaluates to TRUE statement(s)1 are executed. If *conditional expression1* evaluates to FALSE, *conditional expression2* is evaluated. If *conditional expression2* is TRUE statement(s)2 are executed, if *conditional expression2* is FALSE statement(s)3 are executed. Execution continues with the next statement after the IF blocks.

Multiple-choice selection can be achieved by using nested IF statements. But, if too many IF statements are nested inside one another it becomes difficult to work out the logic. Some programming languages provide a simpler construct to achieve multiple-choice selection in the form of a case statement.

Iteration (repetition)

An iteration construct is used to repeat the execution of a block of statements.

Entry condition loop

In this construct the *conditional expression* is tested before the loop is executed. If the *conditional expression* evaluates to FALSE when first tested the loop statement(s) will not be executed and execution will continue with the statement after the END LOOP. If the *conditional expression* evaluates to TRUE, then the loop statement(s) will be executed once and the *conditional expression* retested. The loop will be repeated until the *conditional expression* evaluates to FALSE.

LOOP
WHILE *conditional expression*
    statement(s)
END LOOP

Exit condition loop

In this construct, the loop statements are executed once and then the *conditional expression* is tested. If the *conditional expression* is FALSE the loop statement(s) are not repeated and the statement following the END LOOP would be executed next. As long as the *conditional expression* evaluates to TRUE the loop statement(s) are repeated.

LOOP
    statement(s)
WHILE *conditional expression*
END LOOP

> An exit condition loop is always executed at least once.
>
> Care must be taken when writing loops. The conditional expression must evaluate to FALSE at some point during execution otherwise the loop will continue executing forever.

Relational operators

Relational operators are used in conditional expressions. Most programming languages use the same operators for less than, greater than, less than or equal to and greater than or equal to.

Operator	**Symbols**
Less than	<
Greater than	>
Less than or equal to	<=
Greater than or equal to	>=
The most common symbols for	
Equal to	=
Not equal to	NOT =

The operators for equal to and not equal to are different depending on which programming language is used. In most programming languages equal to is represented by the equal sign =, but in Java, C and C++ it is represented by the use of double equal signs ==. Not equal to can be represented by !=, <> or NOT = depending on the language used.

Relational operators are used in conditional expressions and the result evaluates to TRUE or FALSE.

Conditional expression	**Meaning**	**Result**
6 > 5	6 greater than 5	TRUE
4 <= 4	4 less than or equal to	TRUE
15 > 21	15 greater than 21	FALSE
35 >= 49	35 greater than or equal to 49	FALSE
17 = 19	17 equal to 19	FALSE
12 <> 12	12 not equal to 12	FALSE
25 >= 15	25 greater than or equal to 15	TRUE

Logical operators

Logical operators are used in conditional expressions to combine relational expressions. The logical operators are AND, OR and NOT. The symbols used for these operators are different in different programming languages but the meaning is always the same.

AND

The result from a conditional expression that uses AND to join expressions is TRUE only if ALL the individual expressions evaluate to TRUE.

Conditional expression	Meaning	Result
5 > 6 AND 8 < 12	5 greater than 6 AND 8 less than 12	The result is FALSE because 5 is not greater than 6 so both relational expressions are not TRUE.
12 >= 4 AND 16 < 20	12 greater than or equal to 4 AND 16 less than 20	The result is TRUE because both the relational expressions are TRUE.
36 > 35 AND 6 < 10 AND 41 >= 30	36 greater than 35 AND 6 less than 10 AND 41 greater than or equal to 30	The result is TRUE because all the expressions evaluate to TRUE.
26 > 50 AND 8 > 4 AND 19 <= 20	26 greater than 50 AND 8 greater than 4 AND 19 less than or equal to 20	The result is FALSE as the first expression evaluates to FALSE.

OR

The result from a conditional expression that uses OR to join expressions is TRUE if at least one of the individual expressions evaluates to TRUE. The result is FALSE only when all the expressions evaluate to FALSE.

Conditional expression	Meaning	Result
5 > 6 OR 8 < 12	5 greater than 6 OR 8 less than 12	The result is TRUE as one of the expressions 8 < 12 is TRUE.
12 >= 4 OR 16 < 20	12 greater than or equal to 4 OR 16 less than 20	The result is TRUE because both the relational expressions are TRUE.
36 > 35 OR 6 < 10 OR 41 >= 45	36 greater than 35 OR 6 less than 10 OR 41 greater than or equal to 45	The result is TRUE because two of the expressions evaluate to TRUE.
26 > 50 OR 2 > 4 OR 19 <= 11	26 greater than 50 OR 2 greater than 4 OR 19 less than or equal to 11	The result is FALSE as all the expressions evaluate to FALSE.

NOT

A relational expression that uses NOT has the result reversed so that TRUE becomes FALSE and FALSE become TRUE.

Conditional expression	Meaning	Result
NOT 8 > 4	NOT 8 greater than 4	8 > 4 is TRUE so the result is reversed by the NOT to FALSE.
NOT (12 >= 4) OR (24 < 20)	NOT 12 greater than or equal to 4 OR 24 less than 20	12 >= 4 is TRUE so the result is reversed by the NOT to FALSE. 24 < 20 is FALSE. Both expressions evaluate to FALSE so the result is FALSE because they are joined by an OR.
NOT (19 < 14 AND 30 > 26)	NOT (19 less than 14 AND 30 greater than 26)	Because the brackets are around the whole expression it is the result for the whole expression that is reversed. 19 less than 14 is FALSE. 30 greater than 26 is TRUE. Because they are joined by an AND the result is FALSE but applying the NOT the result is reversed to TRUE.

Assignment operator

The assignment operator is used to assignment a value to a variable. Most programming languages use the equal sign, **=** as the assignment operator symbol.

Pascal uses a colon followed by an equal sign, **:=**. Java, C and C++ have extra assignment operators which are used as shorthand to do an assignment operation plus an arithmetic operation e.g. *= which is used to do a multiply operation as well as an assignment.

```
total = 12 + 4;
```

This statement would add the values 12 and 4 and then assign the result to the variable total.

Processes in creation of software

The art of computer programming is not just to produce a program that will work. A program should be carefully thought out and designed before being coded into a computer programming language. It should be documented so that anyone reading the documentation can understand how to make alterations to the program and also how to run the program. Design of software is even more important where a team of programmers is involved in the creation of the software. Unless the software is designed as a whole, then when each individual programmer's code is joined together to form the whole software, the program will not work because each programmer has worked independently instead of from one design document.

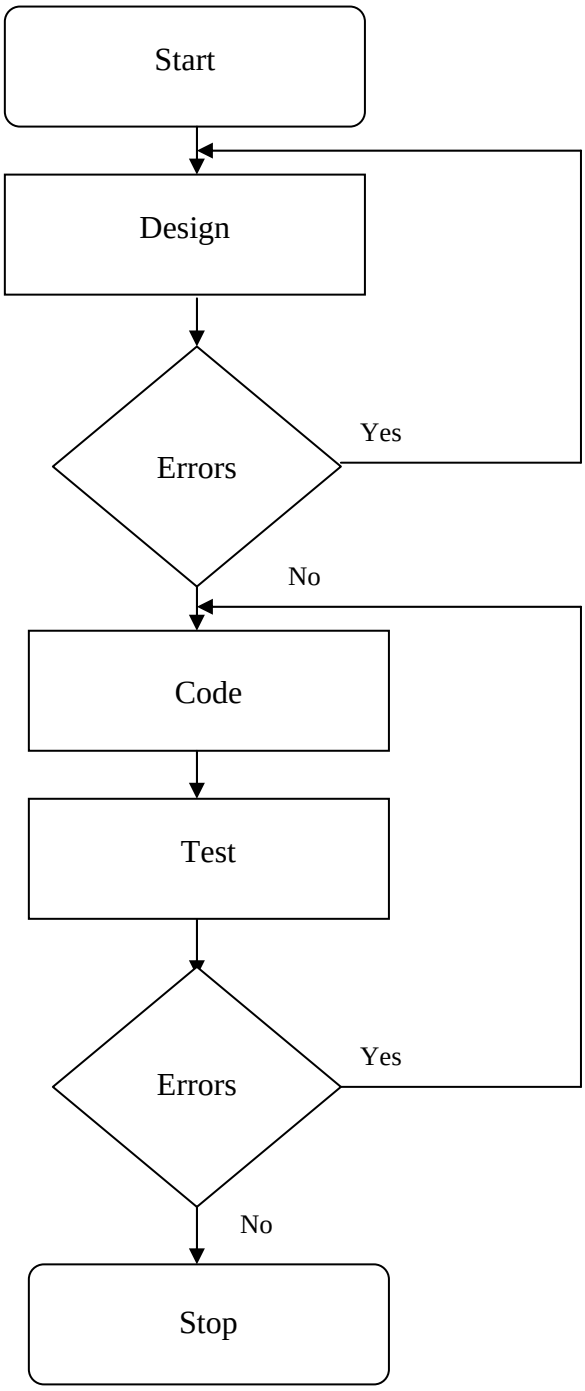
The 4 stages involved in producing a computer program are as follows: -

Stage		Time factor
1	Design	50%
2	Code	10%
3	Test	35%
4	Documentation	5%

Stage 1 Design the program

This stage seems to be omitted by a lot of developers. It is in fact the most important part of producing a program. If a program is designed correctly then it will speed up the processes of coding and testing of a program and produce more accurate programs. The program is designed and the design must be correct before the next stage of coding the program is started. Once the program has been coded then it must be tested. If any errors are found when the program is tested then the errors must be corrected and the testing restarted. If the program has been badly designed then this could mean that the design stage has to be re-entered so that the design can be altered. This should be avoided, since it is very bad practice to redesign a program once it has progressed as far as the testing stage. In other words it is far better to spend more time on the design and get it correct first time than to have to go back and redesign the program at a later stage.

The processes involved in producing a program are shown in the following flowchart diagram :-



A program should be designed as a series of self-contained modules. This will aid the later testing stage since the modules can be tested independently. It also means that if an error is found in a module the amendment to correct the error should not affect the operation of the other modules. Most programs require amending at some stage in their life cycle. It is far easier to amend a program that is written in a modular form. This is because an amendment to one module should not affect the other modules in the program. If a program is designed as a series of self-contained modules then several programmers can be involved in the creation of the code.

When designing the program the developer should bear in mind that at a future date someone else may have to alter the program. Therefore, variables should be given meaningful names; comment lines should be inserted to explain the purpose of routines and programs should be written so that they are easy to amend. Actual literal values should not be used in the program if a value is liable to change at a later date. A good example of this is a payroll program. The rates for tax and national insurance are set by the government and are changed fairly regularly. Therefore, these rates should be held as variables, not as actual values within program statements. Only one alteration is required to alter a value assigned to a variable. If actual literal values are used in the program statements these statements have to be found in order to alter the values contained within them.

During the design stage exceptions must be looked for and incorporated in the design. It is the exceptions that will cause the program to crash if they have not been identified.

Example
If a program is doing a division then the divisor should be checked, to ensure that it is not zero, and action taken if it is. Division by zero causes a program to crash. It is not good enough to say that the divisor will never contain zero, if incorrect data is present then it is possible and must be allowed for.

In designing a program certain questions have to be asked

- What is the purpose of the program?
- What computer hardware is used for input and output?
- What input data is required and how should it be input?
- What processing is to be done?
- What output is required?
- What files are required?
- How is the data to be stored in the files?
- How much space is required for the files?

A successful developer will question and question again until everything in the design specification has been clearly defined.

Stage 2 Code the program

If the design specification has been done well this part of the process is the simplest. It is merely a case of translating the design specification into a particular programming language.

Stage 3 Test the program

This is another important part of the process of writing a program. The purpose of testing is to find any errors in the program. It is no good just entering data that you know the program will accept.

Invalid data must be entered to ensure that the program will reject it. If numbers to be entered can only be in the range 50 - 500 then numbers below 50 and above 500 should be entered to ensure that the program would reject them. If the program does not reject them then it will be working on incorrect data. All paths through the program should be tested and all the output checked thoroughly.
A test plan should be written out and a record kept of all the tests made and the results obtained. If major faults are found in the program it may be necessary to start the testing again from the beginning. Even the simplest amendment to a program can cause errors to occur which were not present before the amendment was made.

Stage 4 Documentation

Although this is called stage 4 the documentation should be started during stage 1.
A design specification should be produced in stage 1. This will include a structure diagram for the program, any decision tables required, the layout of input and output files, screen and printer layouts.

During stage 2 the programmer should be writing the program description and documentation. This documentation should be a clear explanation of what the program does and how it does it. It may also include instructions for running the program, that is, the operating instructions. If there are any limitations these should also be included, for example, if the program can process a maximum of 500 records, then this must be stated.

Features of well-designed software

A developer should ensure that software is

- Modular
  The software should be broken down into modules where each module carries out one function.

- Easy to modify
  Software should be designed so that it can be easily modified at a later date. Consistent naming conventions should be used for variables, files and functions/procedures/subroutines.

- Efficient in performance
  Software must meet its specification whilst still maintaining and meeting performance criteria.

- Reliable
  Software must be tested to eliminate errors; this is an easier task if the software is well designed and modular.

- Documented
  Software must be documented and the documentation kept up to date so that people other than the authors can modify and maintain it.

Advantages of a rigorous and standardised method of design

The benefits of applying a structured approach to design are listed below:

1. Large-scale software projects can be divided between members of a development team.

2. Overall development time can be reduced.

3. Software can be developed and tested incrementally when it has been designed in modules.

4. Software is more reliable which lowers the maintenance costs.

5. Maintenance and modification are easier to perform.

Questions 1

1. What is the result from each of the following arithmetic expressions?

   a. 4 + 10 * 2 + 6 / 2
   b. 5 * 4 / 2 + 16
   c. 20 – 12 * 3 + 6 / 3
   d. (20 – 12) * (3 + 6) / 3
   e. 15 + 2 * ((5 – 3) + 7)

2. What is the result (TRUE or FALSE) from each of the following conditional expressions?

   a. 29 >= 28
   b. 45 < 45
   c. 34 <= 34
   d. 16 > 8 AND 9 < 3
   e. 3 < 10 AND 12 >= 9 AND 18 < 19
   f. 25 <= 25 AND 62 > 63 AND 22 <> 22
   g. 6 > 4 OR 14 <= 12
   h. 26 >= 26 OR 13 < 9 OR 15 <> 15
   i. 39 > 49 OR 55 >= 70 OR 46 <= 42
   j. NOT 5 > 6
   k. NOT (12 < 13) AND (55 > 31)
   l. NOT (23 >= 23 OR 33 < 22)