PAPER NO. CT 33

SECTION 3

CERTIFIED

INFORMATION COMMUNICATION TECHNOLOGISTS

(CICT)

STRUCTURED PROGRAMMING

STUDY TEXT

KASNEB SYLLABUS

STRUCTURED PROGRAMMING

GENERAL OBJECTIVE
This paper is intended to equip the candidate with the knowledge, skills and attitude that will enable him/her to apply the structured programming approach to develop programs

LEARNING OUTCOMES

A candidate who passes this paper should be able to;

- Analyse a problem and design an appropriate solution
- Write codes using C programming language
- Test and debug a structured program code
- Produce documentation, both user and technical, to support programs.

CONTENT

1. Introduction to structured programming
- Introduction to programming languages
- Types of programming languages
- Generations of programming languages
- Programming approaches
- Language translators
- Basic concepts of structured programming
- Problem definition, structure and design
- Integrated development environment (IDE)

2. Programming basics
- Variables and data types
- Input/output statements
- Assignments
- Namespaces
- Comments
- Pre-processor directives
- Expressions and operators
- Control structures
- Writing and running a simple program

3. Functions/sub-programs
- Functions verses procedures
- Parameter passing
- Recursion
- Calling procedures
- Argument naming
- Event procedures
- Testing and debugging errors
- Writing and running a program using functions and procedures

4. Data structures
- Arrays
- Pointers
- Linked lists
- Unions
- Writing a program using data structures

5. File handling (Input/output)
- Opening files
- Writing to files
- Closing files

6. Application development
- Mobile application development
- Collaborative application development

7. Documentation
- User manuals
- Technical manuals

8. Emerging issues and trends

CONTENT **PAGE**

CHAPTER 1

INTRODUCTION TO STRUCTURED PROGRAMMING

INTRODUCTION TO PROGRAMMING LANGUAGES

Structured programing is a programming paradigm aimed on improving the clarity, quality, and development time of a computer program by making extensive use of subroutines, block structures and while loops—in contrast to using simple tests and jumps such as the *goto* statement which could lead to "spaghetti code" which is both difficult to follow and to maintain.

It emerged in the 1960s—particularly from work by Böhm and Jacopini, and a famous letter, *Go To Statement Considered Harmful*, from Edsger Dijkstra in 1968—and was bolstered theoretically by the structured program theorem, and practically by the emergence of languages such as ALGOL with suitably rich control structures.

TYPES OF PROGRAMMING LANGUAGE

There is no exact system for the classification of programming languages. Usually, classification is determined by programming paradigm. Another mode of classification is by the intended domain of use.

A language is a medium for communication. The languages we speak are called natural languages. A programming language is a subset of the set of natural languages. It contains all the symbols, characters, and usage rules that permit a human being to communicate with computers. A variety of programming languages have been invented over the years of computer history. However, every programming language must accept certain types of written instructions that enable a computer system to perform a number of familiar operations. In other words, every programming language must have instructions that fall under the following categories:

- Input/output Instructions: A program needs input data from the external world (or sometimes given implicitly) with which it performs operations on the input data, and generates output (compare with algorithm). Input/output instructions.Provide details on the type of input or output operations to be performed, and the storage locations to be used during the operations, hence they are provided with purpose.
- Arithmetic Instructions: A program might be required to perform arithmetic operations on the data in the program. Arithmetic instructions are provided for the requirement. These perform the arithmetic operations of addition, subtraction, multiplication, division, etc.
- Logical/Comparison Instructions: They are used to compare two values to check whether the values satisfy a given condition or state.

- Storage/Retrieval and Movement Instructions: These are used to store, retrieve, and move data during processing. Data may be copied from one storage location to another and retrieved as required.
- Control Instructions: These are selection and loop constructs which aid in out-of-sequence program flow.

Although all programming languages have an instruction set that permits these familiar operations to be performed, a marked difference is found between the symbols and syntax used in machine languages, assembly languages, and high-level languages.

1.8.1. Assembly Language

Some applications are developed by coding in assembly language – a language closest to the machine language. This type of application software is most efficient for processing data. However, since a particular flavor of the assembly language is designed for a particular architecture, the type of assembly language understood by a computer depends upon the underlying architecture of the microprocessor used. For example, if a Zilog® microprocessor is used, then the machine language understood is Z-80. On the other hand, a flavour of the Intel® assembly language is used for a Pentium® microprocessor which differs from the Z-80. Thus, assembly languages **are architecture-dependent.**

1.8.2. High-level Languages

Given the inability of assembly language to adapt across different architectural platforms and given the varied nature of real-world problems, a variety of computer languages, called high-level languages, were developed, each suited best to a model in a particular class of problems. High-level language programs are **architecture-independent, easier to write, and provide easier maintenance and readability than their assembly-language counterparts.** However, they require longer execution time compared to assembly-language programs.

Different high-level programming languages were introduced in the 1950s to reduce the problems that arose in writing code in assembly and machine language. When the first high-level languages were developed, the longer translation and execution time were considered as serious limitations of the technique. Nonetheless, the following factors contributed to the popularity of high-level languages for application development:

- Savings in Programming Time and Training: High-level languages were easier to learn and understand. Thus, it took less time and effort to write an error-free program or to make corrections and code-revisions.
- Increased Speed and Capacity of Hardware: The third and fourth generations of computer hardware brought about a revolution in access time, memory and disk capacities. This caused the time overhead incurred by the use of high-level languages to be tolerable.
- Increasing Complexities of Software: With time, software systems grew more complex. This necessitated use of complex constructs each of which could represent several lines of assembly code and was provided by high-level languages.

Some popular high-level languages include FORTRAN (FORmula TRANslation), COBOL (COmmon Business Oriented Language), Pascal (after Blaise Pascal), C,and Ada (after Ada Byron, Countess of Lovelace – the world's first computer programmer).

Additional note

Computer programming language can be classified into two major categories:

•Low Level
•High Level

Low Level Languages

The languages which use only primitive operations of the computer are known as low language. In these languages, programs are written by means of the memory and registers available on the computer. As we all know that the architecture of computer differs from one machine to another, so far each type of computer there is a separate low level programming language. In the other words, Programs written in one low level language of one, architectural can't be ported on any other machine dependent languages. Examples are Machine Language and Assembly Language.

Machine Language

In machine language program, the computation is based on binary numbers. All the instructions including operations, registers, data and memory locations are given in there binary equivalent.

The machine directly understands this language by virtue of its circuitry design so these programs are directly executable on the computer without any translations. This makes the program execution very fast. Machine languages are also known as first generation languages.

A typical low level instruction consists essentially of two parts:

•An Operation Part:
Specifies operation to be performed by the computer, also known as Opcode.

•An Address Part:
Specifies location of the data on which operation is to be performed.

Advantages

Machine language makes most efficient use of computer system resources like storage, registers, etc. the instruction of a machine language program are directly executable so there is no need of translators. Machine language instruction can be used to manipulate the individual bits in a computer system with high execution speed due to direct manipulation of memory and registers.

Drawbacks

Machine languages are machine dependent and, therefore, programs are not portable from one computer to other. Programming in machine language usually results in poor programmer productivity. Machine languages require programmers to control the use of each register in the computer‘s Arithmetic Logic Unit and computer storage locations must be addressed directly, not symbolically. Machine language requires a high level of programming skill which increases programmer training costs. Programs written in machine language are more error prone and difficult to debug because it is very difficult to remember all binary equivalent of register, opcode, memory location, etc. program size is comparatively very big due to non-use of reusable codes and use of very basic operations to do a complex computation.

Assembly Language

Assembly language are also known as second generation languages. These languages substitutes alphabetic or numeric symbols for the binary codes of machine language. That is, we can use mnemonics for all opcodes, registers and for the memory locations which provide us with a facility to write reusable code in the form of macros. Has two parts, one is macro name and the other is macro body which contains the line of instructions. A macro can be called at any point of the program by its name to use the instruction. A macro can be called at any point of the program by its name to use the instructions given in the macro repetitively.

These language require a translator known as ―Assembler‖ for translating the program code written in assembly language to machine language. Because computer can interpret only the machine code instruction, once the translation is completed the program can be executed.

Advantages

Assembly languages provide optimal use of computer resources like registers and memory because of direct use of these resources within the programs. Assembly language is easier to use than machine language because there is no need to remember or calculate the binary equivalents for opcode and registers. An assembler is useful for detecting programming errors. Assembly language encourages modular programming which provides the facility of reusable code, using macro.

Drawbacks

Assembly language programs are not directly executable due to the need of translation. Also, these languages are machine dependent and, therefore, not portable from one machine to another. Programming in assembly language requires a high level of programming skills and knowledge of computer architecture of the particular machine.

High Level Languages (HLL)

All high level language are procedure-oriented language and are intended to be machine independent. Programs are written in statements akin to English language, a great advantage over mnemonics of assembly languages require languages use mnemonics of assembly language. That is, the high level languages use natural language like structures. These languages require translators (compilers and interpreters) for execution. The programs written in a high level language can be ported on any computer that is why they are known as machine independent. The early high-levellanguage comes in third generation of languages, COBOL, BASIC, APL, etc. These languages enable the programmer to write instruction using English words and familiar mathematical symbols which makes it easier than technical details of the computer. It makes the programs more readable too.

Procedures

Procedures are the reusable code which can be called at any point of the program. Each procedure is defined by a name and set of instructions accomplishing a particular task. The procedure can be called by its name with the list of required parameters which should pass to that procedure.

Advantages of High Level Languages

These are the third generation languages. These are procedure-oriented languages and are machine independent. Programs are written in English like statements. As high level languages are not directly executable, translators (compilers and interpreters) are used to convert them in machine language equivalent.

Advantages

1) These are easier to learn than assembly language.
2) Less time is required to write programs.
3) Theseprovide better documentation.
4) These are easier to maintain.
5) These have an extensive vocabulary.

Limitation of Programming language

1) A long sequence statements is to be written for every program.
2) Additional memory space is required for storing compiler or interpreter.
3) Execution time is very high as the HLL programs are not directly executable.

A language is a system of communication. A programming language consists of all the symbols, characters, and usage rules that permit people to communicate with computers. There are at least several hundred, and possibly several thousand different programming languages. Some of these are created to serve a special purpose (controlling a robot), while others are more flexible general-purpose tools that are suitable for many types of applications.

Definition of Programming Language

"A programming language is a set of written symbols that instructs the computer hardware to perform specific tasks. Typically, a programming language consists of a vocabulary and a set of rules (called syntax) that the programmer must learn".

1st generation of programming languages

Machine language is the only programming language that the computer can understand directly without translation. It is a language made up of entirely 1s and 0s. There is not, however, one universal machine language because the language must be written in accordance with the special characteristics of a given processor. Each type or family of processor requires its own machine language. For this reason, machine language is said to be machine-dependent (also called hardware-dependent).

In the computer's first generation, programmers had to use machine language because no other option was available. Machine language programs have the advantage of very fast execution speeds and efficient use of primary memory. Use of machine language is very tedious, difficult and time consuming method of programming. Machine language is low-level language. Since the programmer must specify every detail of an operation, a low-level language requires that the programmer have detailed knowledge of how the computer works. Programmers had to know a great deal aobut the computer's design and how it functioned. As a result, programmers were few in numbers and lacked complexity. To make programming simpler, other easier-to-use programming languages have been developed. These languages, however must ultimately be translated into machine language before the computer can understand and use them.

2nd Generation of programming languages

The first step in making software development easier and more efficient was the creation of Assembly languages. They are also classified as **low-level languages** because detailed knowledge of hardware is still required. They were developed in 1950s. Assembly languages use mnemonic operation codes and symbolic addresses in place of 1s and 0s to represent the operation codes. A mnemonic is an alphabetical abbreviation used as memory aid. This means a programmer can use abbreviation instead of having to remember lengthy binary instruction codes. For example, it is much easier to remember L for Load, A for Add, B for Branch, and C for Compare than the binary equivalents i-e different combinations of 0s and 1s.

Assembly language uses symbolic addressing capabilities that simplify the programming process because the programmer does not need to know or remember the exact storage locations of instructions or data. Symbolic addressing is the ability to express an address in terms of symbols chosen by the programmer rather than in terms of the absolute numerical location. Therefore, it is not necessary to assign and remember a number that identifies the address of a piece of data.

Although assembly languages represented an improvement, they had obvious limitations. Only computer specialists familiar with the architecture of the computer being used can use them. And because they are also machine dependent, assembly languages are not easily converted to run on other types of computers.

Before they can be used by the computer, assembly languages must be translated into machine language. A language translator program called an assembler does this conversion. Assembly languages provide an easier and more efficient way to program than machine languages while still maintaining control over the internal functions of a computer at the most basic level. The advantages of programming with assembly languages are that they produce programs that are efficient, use less storage, and execute much faster than programs designed using high-level languages.

3rd Generation of programming languages

Third generation languages, also known as high-level languages, are very much like everyday text and mathematical formulas in appearance. They are designed to run on a number of different computers with few or no changes.

Objectives of high-level languages

- To relieve the programmer of the detailed and tedious task of writing programs in machine language and assembly languages.
- To provide programs that can be used on more than one type of machine with very few changes.
- To allow the programmer more time to focus on understanding the user‘s needs and designing the software required meeting those needs.

Most high level languages are considered to be procedure-oriented, or Procedural languages, because the program instructions comprise lists of steps, procedures, that tell the computer not only what to do but how to do it. High-level language statements generate, when translated, a comparatively greater number of assembly language instructions and even more machine language instructions. The programmer spends less time developing software with a high level language than with assembly or machine language because fewer instructions have to be created.

A language translator is required to convert a high-level language program into machine language. Two types of language translators are used with high level languages: compilers and interpreters.

4th Generation of programming languages

Fourth generation languages are also known as very high level languages. They are non-procedural languages, so named because they allow programmers and users to specify what the computer is supposed to do without having to specify how the computer is supposed to do it. Consequently, fourth generation languages need approximately one tenth the number of statements that a high level languages needs to achieve the same results. Because they are so much easier to use than third generation languages, fourth generation languages allow users, or non-computer professionals, to develop software.

Objectives of fourth generation languages

- Increasing the speed of developing programs.

CHAPTER 2

PROGRAMMING BASICS

VARIABLES AND DATA TYPES

Variables and Data Types

Variables are the nouns of a programming language: they are the entities (values, data) that act or are acted upon. The character-counting program uses two variables--count and args. The program increments count each time it reads a character from the input source and ignores args. A variable declaration always contains two components: the type of the variable and its name. Also, the location of the variable declaration, that is, where the declaration appears in relation to other code elements, determines the scope of the variable.

Variable Types

All variables in the Java language must have a data type. A variable's type determines the values that the variable can have and the operations that can be performed on it. For example, the declaration int count declares that count is an integer (int). Integers can have only whole number values (both positive and negative) and you can use the standard arithmetic operators (+, -, and so on) on integers to perform the standard arithmetic operations (addition, subtraction, and so on). There are two major categories of data types in the Java language: primitive types and reference types.

Primitive types contain **a single value and include types such as integer, floating point, character, and Boolean.** The following table lists, by keyword, all of the primitive data types supported by Java, their size and format, and a brief description of each.

Type	Size/Format	Description
(whole numbers)		
byte	8-bit two's complement	Byte-length integer
short	16-bit two's complement	Short integer
int	32-bit two's complement	Integer
long	64-bit two's complement	Long integer
(real numbers)		
float	32-bit IEEE 754	Single-precision floating point
double	64-bit IEEE 754	Double-precision floating point
(other types)		
Char	16-bit Unicode character	A single character

Boolean true or false aBoolean value (true or false)
Reference types are called such because **the value of a reference variable is a reference (a pointer in other terminology) to the actual value or set of values represented by the variable.** For example, the character-counting program declares (but never uses) one variable of reference

type, args, which is declared to be an array of String objects. When used in a statement or expression, the name args evaluates to the address of the memory location where the array lives. This is in contrast to the name of a primitive variable, the count variable, which evaluates to the variable's actual value.
Besides arrays, classes and interfaces are also reference types. Thus when you create a class or interface you are in essence defining a new data type. See Objects, Classes, and Interfaces for information about defining your own classes and interfaces.

Note to C and C++ Programmers: There are three C Data Types Not Supported By the Java Language. They are pointer, struct, and union. These data types are not necessary in Java; you use classes and objects instead.

Variable Names
A program refers to a variable's value by its name. For example, when the character-counting program wants to refer to the value of the count variable, it simply uses the name count. By convention, variable names begin with a lower case letter (class names begin with a capital letter).

In Java, a varia**ble name:**

- Must be a legal Java identifier comprised of a series of Unicode characters. Unicode is a character coding system designed to support text written in diverse human languages. Unicode allows for the codification of up to 65,536 characters (currently 34,168 have been assigned). This allows you to use characters in your Java programs from various alphabets such as Japanese, Greek, Russian, Hebrew, and so on. This is important so that programmers can write code that is meaningful in their native languages.
- must not be the same as a keyword or a Boolean literal (true or false)
- must not have the same name as another variable whose declaration appears in the same scope

Rule #3 implies that variables may have the same name as another variable whose declaration appears in a different scope. This is true. In addition, in some situations, a variable may share names with another variable which is declared in a nested scope.
By convention, variables names begin with a lower case letter. If a variable name is comprised of more than one word, such as is Visible, the words are joined together and each word after the first begins with an upper case letter.

Scope
A variable's scope is the block of code within which the variable is accessible. Also, a variable's scope determines when the variable is created and destroyed. You establish the scope of a variable when you declare it. Scope places a variable into one of these four categories:

- member variable
- local variable
- method parameter
- exception handler parameter

A member variable is a member of a class or an object and is declared within a class (but not within

any of the class's methods). The character-counting program declares no member variables. Local variables are declared within a method or within a block of code in a method. In the character-counting example, count is a local variable. The scope of count, that is, the code that can access count, extends from the declaration of count to the end of the main() method (indicated by the first right curly bracket ('}') that appears in the sample code). In general, a local variable is accessible from its declaration to the end of the code block in which it was declared.

Method parameters are formal arguments to methods and constructors and are used to pass values into methods and constructors. The discussion about writing methods on the Implementing Methods page in the next lesson talks about passing values into methods and constructors through method parameters. In the character-counting example, args is a method parameter to the main() method. The scope of a method parameter is the entire method or constructor for which it is a parameter. So, in the example, the scope of args is the entire main method.

Exception handler parameters are similar to method parameters but are arguments to an exception handler rather than to a method or a constructor. The character-counting example does not have any exception handlers, so it doesn't have any exception handler parameters. Handling Errors using Exceptions talks about using Java exceptions to handle errors and shows you how to write an exception handler with its parameter.

Variable Initialization
Local variables and member variables can be initialized when they are declared. The character-counting program provides an initial value for count when declaring it: int count = **0**; The value assigned to the variable must match the variable's type.
Method parameters and exception handler parameters cannot be initialized in this way. The value for a parameter is set by the caller

Additional notes

At the core of any program are *variables*. Variables are where the dynamic information is stored. When you type your name into a web form and send it, your name is a variable.

Not all variables are the same though. In fact, there are many different types of variables that nearly every programming language has. Let‘s look at a small selection of them, as well as their short names if they have one:

Character (char): This is a single character, like **X**, **£**, **4**, or *****. You don‘t often create single character variables, but they are at the core of the language so you need to know what they are**String**: This is a ―string‖ of *characters* (see how they‘re at the core?) of any length. In my previous example – your name on web form – your name would be stored as a *String variable*.

Integer (int): A whole number – whole meaning there are no digits after a decimal point. So *65* would be a valid integer; *65.78* would not.

Floating-point number (float): A number that may have digits after the decimal place. *65.00* is technically a floating point number, even though it could be represented just as easily as an integer

as *65*. It takes more memory to store a float, which is why there is a distinction instead of just creating a ―number‖ data type.

Boolean (bool): A variable to represent true or false (or it could also mean 0 or 1, on or off). The simplest data type and commonly used – get used to this one!

Array: These are essentially lists of other variables. There are a variety of array types depending on the language, but basically they‘re just a collection of variables in a sequential list. For example: *1, 2,3,4,5* might be stored as an array (of length 5) containing integer variables. Each variable in the array can then be accessed using an index – but you should know the first item in the list has an index of 0 (yes, that can be confusing sometimes). By storing them as an array, we make it easy to send a collection of variables around the program and do things with them as a whole – such as counting how many things are in the array or doing the same thing to each item (which is called an iteration, and we‘ll get to that another time). You should also know that a **string** is actually just an **array** of **characters**.

Strong and Weak Typed:

Moving on, programming languages can be divided into those that are strongly-typed, and those that are weakly-typed. A strongly typed language (such as Java) requires that you explicitly declare what type of variable you are creating, and they get very upset if you start trying to do things with them that you shouldn‘t. For example, a strongly typed language would give you errors if you tried to add an **integer** and a **string** together. ―*How on earth am I supposed to mathematically add together a word and a number?*‖ it would cry – even though you as a human clearly understand a **string ―5‖** is semantically the same as an **integer with the value of 5**.

A weakly typed language on the other hand would just say ―*whatever*‖, and give it a shot without complaint – but the answer could go either way. Perhaps ―5+5‖ = 10, perhaps it‘s ―55‖ – who knows! It might seem at first like weakly-typed languages are easier to write, but they can often result in curious errors and unexpected behavior that take you a while to figure out.

Assignment and Equality:

Nothing to do with socialism…Instead, it‘s a concept that catches out many programming newbies so I wanted to address it now. There is a difference between **assigning** and **testing for equality**. Consider the following, both of which you would probably read as ―*A is equal to 5*‖:

```
A = 5;
A == 5;
```

Can you tell the difference? The first is known as assignment. It means **assignthe value of 5to variable A**. You are ―setting‖ the variable value. The second statement is one of equality. It‘s a test

– so it actually means —*is A equal to 5?*‖ – The answer given back to you would be a **Boolean value**, true or false. You'll see how this can mess up your programs in later lessons.

CHAPTER 3

FUNCTIONS/ SUB-PROGRAMS

USER-DEFINED FUNCTIONS/SUB-PROGRAMS:
In many programs, a task needs to be completed more than once. For example, you may need to calculate the factorial of a number many times in a program.
Additionally, if a program is very long it sometimes makes sense to break the program into many parts. For example, in air quality modeling there are sub-programs for handling advection, diffusion, chemistry, emissions, deposition, etc.
You are already familiar with the built-in functions such as **cos(x)**, **sqrt(x)**, and **mean(x)**. Often you will want to write your own functions to accomplish tasks that your program needs to do often. Note: Technically, it is possible to never use functions in your Matlab/Octave code. However, your code may become unnecessarily complicated and lengthy if you do not use them.
You *call* a function in the main program or another function. A list of **arguments** (input) is sent to the function and the function returns some **output**.
With **cos(x)**, the argument **x** (an array) is passed down to the function **cos( )**. The answer (an array) is returned as output
User-defined functions are created in separate M-Files. These M-Files must have the following syntax:
Function [output variables] = function name(input variables, also called arguments)
Commands
return
endfunction
(Note: If you only have 1 output variable, which often is the case, you may leave off the brackets.)
Let's create a function y(x) = x2 + 2x + 1 using techniques we have learned in the past.
In main.m:

```
x = input('Enter the value of x: ');
y = x^2 + 2*x + 1;
fprintf('The value of y(x) is: %f \n', y)
```

After execution: (assume the value of 3 is entered by the user)
Enter the value of x: 3
The value of y(x) is: 16.000000 2.
We can do the same task with a function. The function M-File name must be the same as the name of the function (**function_name**). For example,

In the M-File y.m:

```
function [result] = y(x)
```

result = x^2 + 2*x + 1;
return
endfunction
There is one argument (input) called **x** and one output variable called **output**. You can call the function **y(x)** in any other M-File as long as function M-file and M-file that calls the function are both in the same directory. For example, in the main M-File (let‘s call it ―main.m‖),
In main.m:

x = input('Enter the value of x: ');
fprintf('The value of y(x) is: %f \n', y(x))
fprintf('The value of y(x) is: %f \n', y(1))
At command line:
>main
Enter the value of x: 3
The value of y(x) is: 16.000000
The value of y(x) is: 4.000000
IMPORTANT: Both main.m and y.m must be in the same directory!
Each time you call a function, the VALUE of the argument in the calling program (main.m, in this case) is passed down to the argument in the function. In the above example,
- In the first fprintf( ) the VALUE of x (which is 3) is passed down to the variable **x** in the function **y**. This value is used to calculate **result** (which is 16) in the function **y**, which is the output for the function. Once the function is completed, VALUE of the result is passed back to the calling program and displayed to the screen.
- In the second fprintf( ) the VALUE of the argument (which is 1) is passed down to the variable **x** in the function **y**. This value is used to calculate **result** (which is 4) in the function **y**, which is the output for the function. Once the function is completed, VALUE of the result is passed back to the calling program and displayed to the screen.You can call up multiple functions within a program.
Let‘s create another function z(x) = x-1. First, we need to create another M-file called **z.m**.
In the M-File z.m:
function [result] = z(x)
result = x-1;
return
endfunction

Now we can use this function whenever we want in the main program as long as **z.m** is in the same directory as **main.m**
In main.m:
x = input('Enter the value of x: ');
fprintf('The value of y(x) is: %f \n', y(x))
fprintf('The value of z(x) is: %f \n', z(x))
fprintf('The value of y(x) * z(x) is: %f \n', z(x)*y(x))
x = 2; % The value of x has changed
fprintf('The value of y(x) * z(x) is: %f \n', z(x)*y(x))
After execution:
Enter the value of x: 3
The value of y(x) is: 16.000000
The value of z(x) is: 2.000000
The value of y(x) * z(x) is: 32.000000
The value of y(x) * z(x) is: 9.000000
You can call functions as many times as you like, which is the entire point of creating functions in the first place.

Functions with multiple input:
Thus far, we have only examined functions with a single input (or argument). We can pass down many arguments to functions. The surface area of a cylinder is a function of both the height and radius. Let‘s make a function to calculate the surface area of a cylinder called **SAC**.
In SA_cyl.m:

```
function [SA] = SAC(h,r)
SA = 2*pi*r^2 + 2*pi*r*h;
return
endfunction 4 .
```

In main.m:

```
r1 = 2;
h1 = 1;
fprintf ('The surface area of cylinder 1 is: %7.2f \n‘,SAC (h1, r1) )
r2 = 1;
h2 = 1;
fprintf('The surface area of cylinder 2 is: %7.2e \n ' , SAC(h2,r2) )
```

At command line:

```
>main
The surface area of cylinder 1 is: 37.70
The surface area of cylinder 2 is: 1.26e+001
```

Notice that the arguments are labeled **h1**, **r1**, **h2**, and **r2**, not the names of the arguments in the function (**h** and **r**). Remember that we are only sending down the VALUE of the arguments **h1**, **r1**, **h2**, and **r2** to the function.

Functions with multiple output:
Thus far, we have only made functions with a single output. But often we want functions to calculate multiple quantities. Here is a currency converter. In main.m:

```
dollars = input('Enter number of dollars: ');
yenperdollar = 90;
eurosperdollar = 0.70;
[Yen, euros] = convert (dollars, yenperdollar,
eurosperdollar); fprintf('This is %6.1f yen \n', yen);
fprintf('This is %6.1f euros \n', euros);
```

In convert.m:

```
function [yen, euros] = convert (dollars, yenperdollar, eurosperdollar)
yen = dollars * yenperdollar;
euros = dollars * europerdollar;
return
endfunction
```

At command line:

```
>main
Enter number of dollars: 50
This is 4500.0 yen
    This is 35.0 euro
```

Functions definition

Functions are used extensively in computer languages and spreadsheets. Recall that a function takes an input , does some calculations on the input, and then gives back a result. In computer programming they are a very similar idea, with a few changes to naming and properties.

A function in a programming language is a program fragment that 'knows' how to perform a defined task. For example a function may be written that finds the average of three supplied numbers. Once written, this function may be used many times without having to rewrite it over and over.

Example - the function avg

```
function avg(a,b,c)
{ var result =
(a+b+c)/3; return result;
```

The above, written in JavaScript, performs the average function. On the first line, the name of the function is 'avg', It expects three inputs called a,b and c. In computer programming these are called parameters; they stand for the three values sent when the function is used. The function has its own private variable called result which is calculated from the parameters and then the function 'returns' the result;

Using the function

In computer programming the act of using the function is "calling the function". In the program below there are two "calls" to the function. In each case, three particular values are sent as parameters and the result will be the average of the three.

```
        /* main program*/
..
        var averageHt = avg(6, 4, 7);
        ..
        ..
        ..
        var averageAge = avg(30, 45, 21);
    ..
```

So as you can see, functions in computer programming and spreadsheets are very similar to those in math, and serve to 'package' some calculations so it can be separated and used over and over.

CHAPTER 4

DATA STRUCTURES

Introduction

In computer science, a **data structure** is a particular way of storing and organizing data in a computer so that it can be used efficiently.

Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, B-trees are particularly well-suited for implementation of databases, while compiler implementations usually use hash tables to look up identifiers.

Data structures provide a means to manage large amounts of data efficiently, such as large databases and internet indexing services. Usually, efficient data structures are a key to designing efficient algorithms. Some formal design methods and programming languages emphasize data structures, rather than algorithms, as the key organizing factor in software design. Storing and retrieving can be carried out on data stored in both main memory and in secondary memory.

Overview

- An **array** stores a number of elements in a specific order. They are accessed using an integer to specify which element is required (although the elements may be of almost any type). Typical implementations allocate contiguous memory words for the elements of arrays (but this is not always a necessity). Arrays may be fixed-length or expandable.
- **Records** (also called **tuples** or **structs**) are among the simplest data structures. A record is a value that contains other values, typically in fixed number and sequence and typically indexed by names. The elements of records are usually called *fields* or *members*.
- A **hash table** (also called a **dictionary** or **map**) is a more flexible variation on a record, in which name-value pairs can be added and deleted freely.
- A **union** type specifies which of a number of permitted primitive types may be stored in its instances, e.g. "float or long integer". Contrast with a record, which could be defined to contain a float *and* an integer; whereas, in a union, there is only one value at a time. Enough space is allocated to contain the widest member datatype.
- A **tagged union** (also called a **variant**, **variant record**, **discriminated union**, or **disjoint union**) contains an additional field indicating its current type, for enhanced type safety.
- A **set** is an abstract data structure that can store specific values, without any particular order, and with no repeated values. Values themselves are not retrieved from sets, rather one tests a value for membership to obtain a boolean "in" or "not in".

- **Graphs** and **trees** are linkedabstract data structures **composed** of *nodes*. **Each** node contains a value and also one or more pointers to other nodes. Graphs can be used to represent networks, while variants of trees can be used for sorting and searching, having their nodes arranged in some relative order based on their values.
- An **object** contains data fields, like a record, and also contains program code fragments for accessing or modifying those fields. Data structures not containing code, like those above, are called plain old data structures.

Many others are possible, but they tend to be further variations and compounds of the above.

Basic principles

Data structures are generally based on the ability of a computer to fetch and store data at any place in its memory, specified by an address—a bit string that can be itself stored in memory and manipulated by the program. Thus the record and array data structures are based on computing the addresses of data items with arithmetic operations; while the linked data structures are based on storing addresses of data items within the structure itself. Many data structures use both principles, sometimes combined in non-trivial ways (as in XOR linking).

The implementation of a data structure usually requires writing a set of procedures that create and manipulate instances of that structure. The efficiency of a data structure cannot be analyzed separately from those operations. This observation motivates the theoretical concept of an abstract data type, a data structure that is defined indirectly by the operations that may be performed on it, and the mathematical properties of those operations (including their space and time cost).

Language support

Most assembly languages and some low-level languages, such as BCPL (Basic Combined Programming Language), lack support for data structures. On the other hand, many high-level programming languages and some higher-level assembly languages, such as MASM, have special syntax or other built-in support for certain data structures, such as records and arrays. For example, the C and Pascal languages support structs and records, respectively, in addition to vectors (one-dimensional arrays) and multi-dimensional arrays.

Most programming languages feature some sort of library mechanism that allows data structure implementations to be reused by different programs. Modern languages usually come with standard libraries that implement the most common data structures. Examples are the C++Standard Template Library, the Java Collections Framework, and Microsoft's .NET Framework.

Modern languages also generally support modular programming, the separation between the interface of a library module and its implementation. Some provide opaque data types that allow clients to hide implementation details. Object-oriented programming languages, such as C++, Java and Smalltalk may use classes for this purpose.

Many known data structures have concurrent versions that allow multiple computing threads to access the data structure simultaneously

ARRAYS

The simplest type of data structure is a linear array. This is also called one-dimensional array. In computer science, an **array data structure** or simply an **array** is a data structure consisting of a collection of *elements* (values or variables), each identified by at least one *array index* or *key*. An array is stored so that the position of each element can be computed from its index tuple by a mathematical formula.

For example, an array of 10 integer variables, with indices 0 through 9, may be stored as 10 words at memory addresses 2000, 2004, 2008, … 2036, so that the element with index i has the address $2000 + 4 \times i$.

Because the mathematical concept of a matrix can be represented as a two-dimensional grid, two-dimensional arrays are also sometimes called matrices. In some cases the term "vector" is used in computing to refer to an array, although tuples rather than vectors are more correctly the mathematical equivalent. Arrays are often used to implement tables, especially lookup tables; the word *table* is sometimes used as a synonym of *array*.

Arrays are among the oldest and most important data structures, and are used by almost every program. They are also used to implement many other data structures, such as lists and strings. They effectively exploit the addressing logic of computers. In most modern computers and many external storage devices, the memory is a one-dimensional array of words, whose indices are their addresses. Processors, especially vector processors, are often optimized for array operations.

Arrays are useful mostly because the element indices can be computed at run time. Among other things, this feature allows a single iterative statement to process arbitrarily many elements of an array. For that reason, the elements of an array data structure are required to have the same size and should use the same data representation. The set of valid index tuples and the addresses of the elements (and hence the element addressing formula) are usually, but not always, fixed while the array is in use.

The term *array* is often used to mean array data type, a kind of data type provided by most high-level programming languages that consists of a collection of values or variables that can be selected by one or more indices computed at run-time. Array types are often implemented by array structures; however, in some languages they may be implemented by hash tables, linked lists, search trees, or other data structures.

The term is also used, especially in the description of algorithms, to mean associative array or "abstract array", a theoretical computer science model (an abstract data type or ADT) intended to capture the essential properties of arrays.

Additional notes

1. Array solves the problem of storing a large number of values and manipulating them is a data structure designed to store a fixed-size sequential collection of elements of the same type, i.e., it is a collection of variables of the same type
2. Array Declarations. creates a storage location for a Reference to an Array, i.e., creating a

Reference Variable for an Array double[ ] temperature; -- preferred notation
double temperature[ ]; -- inherited from the C programming language
3. Array Creation. Specify the Array Size, i.e., Determine the Array Length ,Allocate Memory for the Array & Assign a Reference to that Memory Location

temperature = new double[24];
which allocates sufficient memory to store 24 different temperature readings

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23

POINTERS

In computer science, a **pointer** is a programming language object whose value refers directly to (or "**points** to") another value stored elsewhere in the computer memory using its address. For high-level programming languages, pointers effectively take the place of general purpose registers in low-level languages such as assembly language or machine code, but may be in available memory. A pointer *references* a location in memory, and obtaining the value stored at that location is known as *dereferencing* the pointer. A pointer is a simple, more concrete implementation of the more abstract *reference* data type. Several languages support some type of pointer, although some have more restrictions on their use than others. As an analogy, a page number in a book's index could be considered a pointer to the corresponding page; dereferencing such a pointer would be done by flipping to the page with the given page number.

Pointers to data significantly improve performance for repetitive operations such as traversing strings, lookup tables, control tables and tree structures. In particular, it is often much cheaper in time and space to copy and dereference pointers than it is to copy and access the data to which the pointers point.

Pointers are also used to hold the addresses of entry points for called subroutines in procedural programming and for run-time linking to dynamic link libraries (DLLs). In object-oriented programming, pointers to functions are used for bindingmethods, often using what are called virtual method tables.

While "pointer" has been used to refer to references in general, it more properly applies to data structures whose interface explicitly allows the pointer to be manipulated (arithmetically via *pointer arithmetic*) as a memory address, as opposed to a magic cookie or capability where this is not possible. Because pointers allow both protected and unprotected access to memory addresses, there are risks associated with using them particularly in the latter case. Primitive pointers are often stored in a format similar to an integer; however, attempting to dereference or "look up" a pointer whose value was never a valid memory address would cause a program to crash. To alleviate this potential problem, as a matter of type safety, pointers are considered a separate type parameterized by the type of data they point to, even if the underlying representation is an integer. Other measures may also be taken.

Formal description

In computer science, a pointer is a kind of reference.

A *data primitive* (or just *primitive*) is any datum that can be read from or written to computer memory using one memory access (for instance, both a *byte* and a *word* are primitives).

A *data aggregate* (or just *aggregate*) is a group of primitives that are logically contiguous in memory and that are viewed collectively as one datum (for instance, an aggregate could be 3 logically contiguous bytes, the values of which represent the 3 coordinates of a point in space). When an aggregate is entirely composed of the same type of primitive, the aggregate may be called an *array*; in a sense, a multi-byte *word* primitive is an array of bytes, and some programs use words in this way.

In the context of these definitions, a *byte* is the smallest primitive; each memory address specifies a different byte. The memory address of the initial byte of a datum is considered the memory address (or *base memory address*) of the entire datum.

A *memory pointer* (or just *pointer*) is a primitive, the value of which is intended to be used as a memory address; the pointer is said to *point to a memory address*, or *point to a datum [in memory]* when the pointer's value is the datum's memory address.

More generally, a pointer is a kind of reference, the pointer is said to *reference a datum stored somewhere in memory*; to obtain that datum is *to dereference the pointer*. The feature that separates pointers from other kinds of reference is that a pointer's value is meant to be interpreted as a memory address, which is a rather low-level concept.

References serve as a level of indirection: A pointer's value determines which memory address (that is, which datum) is to be used in a calculation. Because indirection is a fundamental aspect of algorithms, pointers are often expressed as a fundamental data type in programming languages; in statically (or strongly) typed programming languages, the type of a pointer determines the type of the datum to which the pointer points.

Use in data structures

When setting up data structures like lists, queues and trees, it is necessary to have pointers to help manage how the structure is implemented and controlled. Typical examples of pointers are start pointers, end pointers, and stack pointers. These pointers can either be **absolute** (the actual physical address or a virtual address in virtual memory) or **relative** (an offset from an absolute start address ("base") that typically uses fewer bits than a full address, but will usually require one additional arithmetic operation to resolve).

A two-byte offset, containing a 16-bit, unsigned integer, can be used to provide relative addressing for up to 64 kilobytes of a data structure. This can easily be extended to 128K, 256K or 512K if the address pointed to is forced to be on a half-word, word or double-word boundary (but, requiring an additional "shift left" bitwise operation—by 1, 2 or 3 bits—in order to adjust the offset by a factor

CHAPTER 5

FILE HANDLING (INPUT/OUTPUT)

This chapter will explain following functions related to files:

- Opening a file
- Reading a file
- Writing a file
- Closing a file

Opening a File

The PHP **fopen()** function is used to open a file.It requires two arguments stating first the file name and then mode in which to operate.

The first parameter of this function contains the name of the file to be opened and the second parameter specifies in which mode the file should be opened:

```
<html>
<body>

<?php
$file=fopen("welcome.txt","r");
?>

</body>
</html>
```

Files modes can be specified as one of the six options in this table.

Mode	Description
r	Read only. Opens the file for reading only. Places the file pointer at the beginning of the file.
r+	Read/Write.Opens the file for reading and writing. Places the file pointer at the beginning of the file.

CHAPTER 6

APPLICATION DEVELOPMENT

Coding- In programming, code (noun) is a term used for both the statements written in a particular programming language - the source code , and a term for the source code after it has been processed by a compiler and made ready to run in the computer - the object code .

To code (verb) is to write programming statements - that is, to write the source code for a program.

MOBILE APPLICATION DEVELOPMENT

Mobile application development is the process by which application software is developed for low-power handheld devices, such as personal digital assistants, enterprise digital assistants or mobile phones. These applications can be pre-installed on phones during manufacturing, downloaded by customers from various mobile software distribution platforms, or delivered as web applications using server-side or client-side processing (e.g. JavaScript) to provide an "application-like" experience within a Web browser. Application software developers also have

to consider a lengthy array of screen sizes, hardware specifications and configurations because of intense competition in mobile software and changes within each of the platforms.Mobile app development has been steadily growing, both in terms of revenues and jobs created. A 2013 analyst report estimates there are 529,000 direct App Economy jobs within the EU 28 members, 60% of which are mobile app developers.

How To Build Your First Mobile App In 12 Steps: Part 1

So you woke up in the middle of the night and had this great idea for an amazing app — you can picture it, you know it is useful, and you can imagine that many people would like it, too.

If this is your first-ever app development attempt, here is a brief guide on how to get from A to Z and make the project a success!

Step 1: Define Your Goal

Having a great idea is the starting point into every new project. Before you go straight into detailing though, you must clearly define the purpose and mission of your app. What is it going to do? What is its core appeal? What concrete problem is it going to solve, or what part of life is it going to make better?

Defining a clear goal for the app is also going to help you get there faster.

CHAPTER 7

DOCUMENTATION:

Software documentation, also referred to as source code documentation is a text that describes computer software. It explains how software works but it can also explain how to use the software properly. Several types of software documentation exist and can be classified into:

User Documentation

Also known as software manuals, user documentation is intended for end users and aims to help them use software properly. It is usually arranged in a book-style and typically also features table of contents, index and of course, the body which can be arranged in different ways,

depending on whom the software is intended for. For example, if the software is intended for beginners, it usually uses a tutorial approach and guides the user step-by-step. Software manuals which are intended for intermediate users, on the other hand, are typically arranged thematically, while manuals for advanced users follow reference style.

Besides printed version, user documentation can also be available in an online version or PDF format. Often, it is also accompanied by additional documentation such as video tutorials, knowledge based articles, videos, etc.

Requirements Documentation

Requirements documentation, also referred to simply as requirements explains what a software does and shall be able to do. Several types of requirements exist which may or may not be included in documentation, depending on purpose and complexity of the system. For example, applications that don‘t have any safety implications and aren‘t intended to be used for a longer period of time may be accompanied by little or no requirements documentation at all. Those that can affect human safety or/and are created to be used over a longer period of time, on the other hand, come with an exhausting documentation.

Architecture Documentation

Also referred to as software architecture description, architecture documentation either analyses software architectures or communicates the results of the latter (work product). It mainly deals with technical issues including online marketing and seo services but it also covers non-technical issues in order to provide guidance to system developers, maintenance technicians and others involved in the development or use of architecture including end users. Architecture documentation is usually arranged into architectural models which in turn may be organized into different views, each of which deals with specific issues.

Comparison document is closely related to architecture documentation. It addresses current situation and proposes alternative solutions with an aim to identify the best possible outcome. In order to be able to do that, it requires an extensive research.

CHPATER 8

EMERGING ISSUES AND TRENDS

Programming languages have always been heavily influenced by programming paradigms, which in turn have been characterized by general computing trends. The cumbersomeness of low-level machine code yielded imperative programming languages. These languages take advantage of compilers or interpreters in order to generate low-level machine code based on easier to handle higher-level languages. As a result of increasing complexity and scale of programs, there was a need for finer granularity and encapsulation of code, which led to new modularization concepts. This has been later complemented and extended by the object-oriented paradigm, which introduced new concepts like polymorphism and inheritance. The object-oriented paradigm promotes the design and implementation of large, but manageable, software systems and thus addresses the requirements of large-scale applications. However, the prevalence of multi-core architectures and the pervasion of distributed systems and applications in everyday life represent other trends affecting upcoming programming languages. Some even believe that "the concurrency revolution is likely to be more disruptive than the OO revolution" [Sut05]. Although this is a controversial statement, it is remarkable that most of the new programming languages take concurrency seriously into account and provide advanced concurrency concepts aside from basic threading support [Gho11].

Apart from the object-oriented paradigm, there are several less common paradigms such as declarative or functional programming that focus on high expressiveness. Programming languages following these paradigms have been considered as esoteric and academic languages by the industry for a long time. Interestingly, there is an increasing popularity of these alternative concepts, especially in functional programming and even for web application programming [Vin09]; not least because these languages provide inherently different concurrency implications. As functional languages favor immutability and side-effect free programming, they are by design easier to execute concurrently. They also adapt other techniques for handling mutable state and explicit concurrency.

The gap between imperative, object-oriented languages and purely functional languages has been bridged by another tendency: multi-paradigm programming languages. By incorporating multiple paradigms, programming languages allow the developer to pick the right concepts and techniques for their problems, without committing themselves to a single paradigm. Multi-paradigm languages often provide support for objects, inheritance and imperative code, but incorporate higher-order functions, closures and restricted mutability at the same time. Although these languages are not pure in terms of original paradigms, they propose a pragmatic toolkit for different problems with high expressiveness and manageable complexity at the same time.

The higher expressiveness is often a basic prerequisite for the design of so-called domain-specific languages [Gho10], another trending topic in software development. Domain-specific languages allow to specify and express domain objects and idioms as part of a higher-level language for programming. By providing a higher level of abstraction, domain-specific languages allow to focus on the application or business domain while concealing details of the programming language

or platform. Several frameworks for web development can be considered as domain-specific languages for the domain of web applications.

New Programming Languages for Web Programming

Thanks to the web, JavaScript has not just become the lingua franca of the web, but also the most widespread programming language in general. Every browser, even mobile ones, act as an execution environment for JavaScript applications, thus JavaScript is available on virtually all computing devices. But JavaScript is also increasingly popular outside the web browser, thanks to projects like node.js. Microsoft uses JavaScript as the main programming language for Metro applications in the upcoming Windows 8 release. JavaScript is a multi-paradigm language that incorporates prototypal object inheritance combined with many functional aspects of Scheme. It has been designed as a general-purpose programming language, but reached attention, and sometimes also faced hatred, not until it became popular through the web. However, JavaScript is notoriously known for some of its "bad parts" [Cro08]. Deficiencies include odd scoping, global variables, automatic syntax correction (i.e. semicolon insertion) with misleading results, and problems with the type system and with value comparisons.

As we are somehow locked-in to JavaScript concerning browsers, there are several approaches to circumvent these drawbacks, as long as they are not fixed in the language itself by upcoming specifications. Crock ford suggests a subset of the languages that makes only use of the "good parts" of the language [Cro08]. Others attempt to transcompile (i.e. executing source-to-source compilation) different languages to JavaScript. Popular examples therefore are ClojureScript [McG11] and CoffeeScript. ClojureScript translates Clojure code into JavaScript, though some of the Clojure features are missing. For instance, JavaScript is single-threaded, so the usage of concurrency concepts of Clojure is limited. Coffee Script takes a different route. It is a language that exclusively Trans compiles to JavaScript and has been designed as a syntactic replacement for JavaScript. Coffee Script not just adds syntactic sugar, but also provides some advanced features like array comprehension and pattern matching.

When Google was dissatisfied with the progress on new JavaScript specifications and was reasoning about the future of web programming, they identified the need for a general-purpose web programming language for both clients and servers, independent of JavaScript. This need was shortly after addressed by Google's new Dart [Tea12] programming language. Dart is derived from JavaScript, but incorporates several concepts from Java and other languages. It is class-based, like Java, and supports interfaces, abstract classes and generics. Dart is dynamically typed, but annotations can be used to enforce static typing. A core library provides common data structures and operations, and a DOM library supports HTML5 DOM. For server applications, Dart includes an I/O library with an asynchronous, non-blocking programming model and an event loop. Dart also ships with an HTTP library as a foundation for web servers. Concerning concurrency in Dart, the specification disallows shared-state concurrency. However, Dart proposes actor-like structures, so-called isolates. Each isolate represents an independent flow of control and it can thus be assumed to be single-threaded for the developers. Multiple isolates can communicate via message passing. There are different ways to execute Dart applications. Google's Chrome browser already supports Dart. Time will tell if other browser vendors will eventually support Dart as well. As an interim solution, there is a Dart-to-JavaScript trans compiler that generates pure JavaScript code out of Dart sources. For usage outside of the browser, Google