CPT 121

Introduction to Programming

CODeL
FEDERAL UNIVERSITY OF TECHNOLOGY, MINNA
CENTRE FOR OPEN DISTANCE AND e-LEARNING

FEDERAL UNIVERSITY OF TECHNOLOGY MINNA
NIGER STATE, NIGERIA

CENTRE FOR OPEN DISTANCE AND e-LEARNING (CODeL)

B.TECH. COMPUTER SCIENCE PROGRAMME

COURSE TITLE

INTRODUCTION TO PROGRAMMING

COURSE CODE

CPT 121

COURSE CODE

CPT 121

COURSE UNIT

2

Course Coordinator
Bashir Mohammad (Ph.D.)
Department of Computer Science
Federal University of Technology (FUT) Minna
Minna, Niger State, Nigeria.

Course Development Team

CPT 121: INTRODUCTION TO PROGRAMMING

Subject Matter Experts	Opeyemi ABISOYE (PhD) Department of Computer Science FUT Minna, Nigeria.
Course Coordinator	Bashir Mohammad (Ph.D.) Department of Computer Science FUT Minna, Nigeria.
ODL Experts	Amosa Isiaka GAMBARI (Ph.D.) Nicholas Ehikioya ESEZOBOR
Instructional System Designers	Oluwole Caleb FALODE (Ph.D.) Bushrah Temitope OJOYE (Mrs.)
Language Editors	Chinenye Priscilla UZOCHUKWU (Mrs.) Mubarak Jamiu ALABEDE
Centre Director	Abiodun Musa AIBINU (Ph.D.) Centre for Open Distance & e-Learning FUT Minna, Nigeria.

CPT 121: Study Guide

Introduction

CPT 121: Introduction to programming is a 2-credit unit course for students studying towards acquiring a Bachelor of Technology in Information Technology and other related disciplines. The course is divided into 7 modules and 17 study units. It will first take a brief review of the concepts of programming languages and fundamental programming constructions. This course is a prerequisite for Object Oriented Programming (Part 1, 2) and concept of syntax and semantics of a higher-level language (C++). The course concluded by discussing some concepts like encapsulation, fundamental data structures, software development methodology and recursion.

The course guide therefore gives you an overview of what the course; CPT 121 is all about, the textbooks and other materials to be referenced, what you expect to know in each unit, and how to work through the course material.

What you will learn in this Course

The overall aim of this course, CPT 121 is to introduce you to the basic concepts of programming language. This helps the students to understand the development of C++, a high level programming language. This course highlights different programming languages concepts and fundamental programming constructions. This course will introduce you to the practical using of high level programming language.

Course Aim

The aim of this course is to introduce students to the basics and concepts of programming languages. It is believed the knowledge will enable the student to understand the functionalities and capabilities of programming language, fundamental programming constructions.

Course Objectives

It is important to note that each unit has specific objectives. Students should study them carefully before proceeding to subsequent units. Therefore, it may be useful to refer to these objectives in the course of your study of the unit to assess your progress. You should always look at the unit objectives after completing a unit. In this way, you can be sure that you have done what is required of you by the end of the unit. However, below are overall objectives of this course. On completing this course, you should be able to:

i. Know about history and basic concept of programming languages
ii. Develop fundamental programming constructs
iii. Develop algorithms and problem solving.
iv. Gain Knowledge about fundamental data structures C++
v. Understand Machine levels organization
vi. Explore Software Development Methodology
vii. Recursion and simple procedure of recursion

Working through this Course

To complete this course, you are required to study all the units, the recommended textbooks, and other relevant materials. Each unit contains some self-assessment exercises and tutor marked assignments, and at some point, in this course, you are required to submit the tutor marked assignments. There is also a final examination at the end of this course. Stated below are the components of this course and what you have to do.

Course Materials

The major components of the course are:

i. Course Guide
ii. Study Units
iii. Text Books
iv. Assignment File
v. Presentation Schedule Study Units

Study Units

There are 17 study units and 7 modules in this course. They are:

Module 1: Programming Languages

Unit 1. Brief survey of programming paradigms

Unit 2. Overview of programming languages and compilation process

Module 2: Fundamental programming constructs

Unit 1. Syntax and semantics of programming language C/C++

Unit 2. Structures of simple programs

Unit 3. Function

Module 3: Algorithms and problem solving

Unit 1: Problem solving

Unit 2: Basic concept of an algorithm

Unit 3. Search and sorting algorithms

Module 4: Fundamental data structures

Unit1. Primitive data types

Unit2. Strings

Unit3. Help allocation

Module 5: Machine organization

Unit 1. Machine levels organization

Unit 2. Assembly language programming

Module 6: Software Development Methodology

Unit 1: Fundamental design and concept principles

Unit 2: Testing and debugging strategies

Module 7: Recursion
Unit 1: Concept of recursion

Unit 2: Simple recursive procedures

Recommended Texts

These texts and especially the internet resource links will be of enormous benefit to you in learning this course:

1. http://en.wikipedia. org/
2. http://www.mycplus.com/featured-articles/
3. http://www.cprogramming.com

Assignment File

The assignment file will be given to you in due course. In this file, you will find all the details of the work you must submit to your tutor for marking. The marks you obtain for these assignments will count towards the final mark for the course. Altogether, there are tutor marked assignments for this course.

Presentation Schedule

The presentation schedule included in this course guide provides you with important dates for completion of each tutor marked assignment. You should therefore endeavor to meet the deadlines.

Assessment

There are two aspects to the assessment of this course. First, there are tutor marked assignments; and second, the written examination. Therefore, you are expected to take note of the facts, information and problem solving gathered during the course. The tutor marked assignments must be submitted to your tutor for formal assessment, in accordance to the deadline given. The work submitted will count for 40% of your total course mark.

At the end of the course, you will need to sit for a final written examination. This examination will account for 60% of your total score.

Tutor Marked Assignments (TMAs)

There are TMAs in this course. You need to submit all the TMAs. The best 10 will therefore be counted. When you have completed each assignment, send them to your tutor as soon as possible and make certain that it gets to your tutor on or before the stipulated deadline. If for any reason you cannot complete your assignment on time,

contact your tutor before the assignment is due to discuss the possibility of extension. Extension will not be granted after the deadline, unless on extraordinary cases.

Final Examination and Grading

The final examination for CPT 121 will be of last for a period of 2 hours and have a value of 60% of the total course grade. The examination will consist of questions which reflect the Self-Assessment Exercise(s) and tutor marked assignments that you have previously encountered. Furthermore, all areas of the course will be examined. It would be better to use the time between finishing the last unit and sitting for the examination, to revise the entire course. You might find it useful to review your TMAs and comment on them before the examination. The final examination covers information from all parts of the course.

The following are practical strategies for working through this course

1. Read the course guide thoroughly Organize a study schedule. Refer to the course overview for more details.
2. Note the time you are expected to spend on each unit and how the assignment relates to the units. Important details, e.g. details of your tutorials and the date of the first day of the semester are available. You need to gather together all this information in one place such as a diary, a wall chart calendar or an organizer. Whatever method you choose, you should decide on and write in your own dates for working on each unit.
3. Once you have created your own study schedule, do everything you can to stick to it. The major reason that students fail is that they get behind with their course works. If you get into difficulties with your schedule, please let your tutor know before it is too late for help.
4. Turn to Unit 1 and read the introduction and the learning outcomes for the unit.
5. Assemble the study materials. Information about what you need for a unit is given in the table of content at the beginning of each unit. You will almost always need both the study unit you are working on and one of the materials recommended for further readings, on your desk at the same time.
6. Work through the unit, the content of the unit itself has been arranged to provide a sequence for you to follow. As you work through the unit, you will be encouraged to read from your set books.
7. Keep in mind that you will learn a lot by doing all your assignments carefully. They have been designed to help you meet the objectives of the course and will help you pass the examination.
8. Review the objectives of each study unit to confirm that you have achieved them.

If you are not certain about any of the objectives, review the study material and consult your tutor.

9. When you are confident that you have achieved a unit's objectives, you can start on the next unit. Proceed unit by unit through the course and try to pace your study so that you can keep yourself on schedule.
10. When you have submitted an assignment to your tutor for marking, do not wait for its return before starting on the next unit. Keep to your schedule. When the assignment is returned, pay particular attention to your tutor's comments, both on the tutor marked assignment form and also written on the assignment. Consult you tutor as soon as possible if you have any questions or problems.
11. After completing the last unit, review the course and prepare yourself for the final examination. Check that you have achieved the unit objectives (listed at the beginning of each unit) and the course objectives (listed in this course guide).

Tutors and Tutorials

There are 8 hours of tutorial provided in support of this course. You will be notified of the dates, time and location together with the name and phone number of your tutor as soon as you are allocated a tutorial group. Your tutor will mark and comment on your assignments, keep a close watch on your progress and on any difficulties, you might encounter and provide assistance to you during the course. You must mail your tutor marked assignment to your tutor well before the due date. At least two working days are required for this purpose. They will be marked by your tutor and returned to you as soon as possible.

Do not hesitate to contact your tutor by telephone, e-mail or discussion board if you need help. The following might be circumstances in which you would find help necessary: contact your tutor if:

1. You do not understand any part of the study units or the assigned readings.
2. You have difficulty with the self-test or exercise.
3. You have questions or problems with an assignment, with your tutor's comments on an assignment or with the grading of an assignment.

You should endeavor to attend the tutorials. This is the only opportunity to have face to face contact with your tutor and ask questions which are answered instantly. You can raise any problem encountered in the course of your study. To gain the maximum benefit from the course tutorials, have some questions handy before attending them. You will learn a lot from participating actively in discussions.

Table of Contents

Module 1

Programming Languages

Unit 1. Brief survey of programming paradigms

Unit 2. Overview of programming languages and compilation process

Unit 1

Brief Survey of Programming Paradigms

Contents

1.0 Introduction

This unit gives introduction to history of programming languages and you will learn the basic structure of algorithm, program and paradigms of programming.

Also, you are going to learn how changes paradigms of programming languages. You will know more about object oriented programming languages, structure of program high level languages.

2.0 Learning Outcomes

At the end of this unit, you should be able to:

1. What is a computer program, programming and programming language?
2. What are data type and data structure?
3. Instruction and control flow, design philosophy
4. Compilation and interpretation
5. Program paradigms

3.0 Learning Contents

3.1 History of Programming Languages

A Computer Program –A computer program is a collection of instructions that performs a specific task when executed by a computer *(Knuth, Donald E. (1997))*.

A computer requires programs to function and typically executes the program's instructions in a central processing unit.

ii**.** An organized list of instructions that, when executed, causes the computer to behave in a predetermined manner. Without programs, computers are useless.

iii. A description structure of data and algorithms solving task in a programming language, automatically interpreter in special code using translator.

iv. A computer program is very similar to a cooking recipe which can be defined as a set of instructions for preparing a particular dish, including a list of the ingredients required.

Programming – Computer programming (often shortened to programming) is a process that leads from an original formulation of a computing problem to executable computer programs. Or simply put the art of writing Computer programs

ii. A process of translating mathematical task or applied task into computer language.
.

Programming language – A programming language is a formal language that specifies a set of instructions that can be used to produce various kinds of output.

ii. A vocabulary and set of grammatical rules for instructing a computer to perform specific tasks

iii. A formal notation for specifying computations, independent of a specific machine. Set of imperative commands used to direct computer to do something useful.

There are over 1000 major (i.e. publically available and used for substantial applications) programming languages. Less than half are still "alive" by most standards (some dead languages are still in use).

Languages evolved approximately thus:

1. Machine code, assembler - instruction sets vary enormously in size, complexity, and capabilities
2. FORTRAN, COBOL - earliest high-level languages. flowcharts. chaos
3. Lisp, SNOBOL, APL, BASIC - very high level, with "warts". user-friendly
4. Algol, C, Pascal, PL/1 - "structured" languages
5. Ada, Modula-2, C++ - "modular" systems programming languages
6. Smalltalk, Prolog, Icon, Perl -"very high level" and scripting languages
7. Visual Basic, Python, Java, Ruby, PHP- interface-oriented and web languages

Each programming language can be thought of as a set of formal specifications concerning syntax, vocabulary and meaning.

These specifications usually include:

1. Data types
2. Data structures
3. Instruction and control flow
4. Design philosophy
5. Compilation and interpretation

Self-Assessment Exercise(s) 1

1. A process that leads from an original formulation of a computing problem to executable computer programs is called…
2. Description structure of data and algorithms solving task in a programming language is called…
3. Formal notation for specifying computations, independent of a specific machine is called…

3.2 Data Types and Data Structures

Those languages that are widely used – or have been used for a considerable period of time – have standardization bodies that meet regularly to create and publish formal definitions of the language and discuss the extension of existing definitions.

Data is information in a form the computer can use, for example, numbers and letters. **Information** is any knowledge that can be communicated, including abstract ideas and concepts such as “the Earth is round.”

All data in modern digital computers are stored simply as ***0 or 1 (binary).***

Data comes in many different forms: letters, words, integer numbers, real numbers, dates, times, coordinates on a map, and so on. Virtually any kind of information can be represented as data, or as a combination of data and operations on it. Each kind of data in the computer is said to have a specific data type. For example, if we say that two data items are of type Integer, we know now they are represented in memory and that we can apply arithmetic operations to them.

*A **data type*** is an attribute that tells what kind of value a data can have. It is simply a specification of how information is represented in the computer and the set of operations that can be applied to it

Data type typically represents information in the real world such as names, bank accounts and measurements, so the low-level binary data are organized by programming languages into these high-level concepts.

Data types include the storage classifications like integers, floating point values, strings, characters etc.

Once data has been specified, the machine must be instructed how to perform operations on the data. Elementary statements may be specified using keywords or may be indicated using some well-defined grammatical structure.

Each language takes units of these well-behaved statements and combines them using some ordering system. Depending on the language, differing methods of grouping these elementary statements exist. This allows one to write programs that are able to cover a variety of input, instead of being limited to a small number of cases.

Furthermore, beyond the data manipulation instructions, other typical instructions in a language are those used for control flow (branches, definitions by cases, loops, backtracking, and functional composition).

The particular system by which data are organized in a program is the type system of the programming language; the design and study of type systems is known as type theory.

A **data structure** is a collection of data types, storage, organization and operations being performed on them.

Data structure languages have simple data and combined types (using arrays, lists, stacks, files).

Object oriented languages allow the programmer to define data-types called "Objects" which have their own intrinsic functions and variables (called methods and attributes respectively).

A program containing objects allows the objects to operate as independent but interacting sub-programs: this interaction can be designed at coding time to model or simulate real-life interacting objects. This is a very useful, and intuitive, functionality. Languages such as Python and Ruby have developed as OO (Object oriented) languages.

Self-Assessment Exercise(s) 2

1. An attribute that tells what kind of value a data can have is called …
2. What is a data structure?
3. Differentiate between data type and data structures with specific examples?
4. How is data being stored in modern digital computers?

3.3 Instruction and Control Flow, Design Philosophy

Once data has been specified, the machine must be instructed how to perform operations on the data. Elementary statements may be specified using keywords or may be indicated using some well-defined grammatical structure.

Each language takes units of these well-behaved statements and combines them using some ordering system. Depending on the language, differing methods of grouping these elementary statements exist. This allows one to write programs that are able to cover a variety of input, instead of being limited to a small number of cases.

Instructions are programs that prompt a computer to execute a function. These instructions are stored on the hard disk drive or another storage device and are not executed until they are run by the user.

A Control flow is the order in which instructions, statements and function calls being executed or evaluated when a program is running.

Furthermore, beyond the data manipulation instructions, other typical instructions in a language are those used for ***control flow (***branches, definitions by cases, loops, backtracking, and functional composition).

Each language has been developed using a special design or philosophy. Some aspect or another is particularly stressed by the way the language uses data structures, or by which its special notation encourages certain ways of solving problems or expressing their structure.

Self-Assessment Exercise(s) 3

1. What is a control flow?

3.4. Compilation and Interpretation

The process of transforming program text to machine code is called ***translation***. These approaches are known as compilation, done by a program known as a compiler;

and interpretation, done by an interpreter. Some programming language implementations support both interpretation and compilation.

An interpreter parses a computer program and executes it directly. One can imagine this as following the instructions of the program line-by-line. In contrast, a compiler translates the program into machine code – the native instructions understood by the computer's processor. The compiled program can then be run by itself.

Compiled programs usually run faster than interpreted ones, because the overhead of understanding and translating the programming language syntax has already been done.

The most general term for a software code converting tool is **"translator**." A translator, in software programming terms, is a generic term that could refer to a compiler, assembler, or interpreter; anything that converts higher level code into another high-level code (e.g., Basic, C++, Fortran, Java) or lower-level (i.e., a language that the processor can understand), such as assembly language or machine code.

Compilers convert high-level language code to machine (object) code in one session. Compilers can take a while, because they have to translate high-level code to lower-level machine language all at once and then save the executable object code to memory.

An assembler translates a program written in assembly language into machine language and is effectively a compiler for the assembly language, but can also be used interactively like an interpreter. Assembly language is a low-level programming language.

Interpreters

Another way to get code to run on your processor is to use an interpreter, which is not the same as a compiler. An interpreter translates code like a compiler but reads the code and immediately executes on that code, and therefore is initially faster than a compiler. Thus, interpreters are often used in software development tools as debugging tools, as they can execute a single in of code at a time. Compilers translate code all at once and the processor then executes upon the machine language that the compiler produced

Self-Assessment Exercise(s) 4

1. The process of transforming program text to machine code is called……
2. Differentiate among Interpreter, Compiler and Assembler?

3.5 Program Paradigms

Principal paradigms are:

1. Imperative / Procedural
2. Functional / Applicative
3. Object-Oriented
4. Declarative

The dominant ***imperative*** paradigm has been gradually refined over time. It basically states that to program a computer, you give it instructions in terms it understands. "Procedural" paradigm: a program is a set of procedures/functions. You write new "instructions" by defining procedures. Since the underlying machine works this way, this is the default paradigm and the one that all other paradigms reduce themselves to in order to execute.

Imperative (procedural) programs consists of actions to effect state change, principally through assignment operations or side effects. Example: Fortran, Pascal, Basic, C.

Functional and ***object-oriented*** paradigms are arguably special cases of imperative programming. In functional programming you give the computer instructions in clean, mathematical formulas that it understands. Example: LISP/Scheme, ML, Haskell. In object-oriented programming, you give the computer instructions by defining new data types and instructions that operate on those types. Example: Smalltalk, C++, Java, CLOS

Declarative programming is a polar opposite of imperative programming, introduced in many different application contexts. In declarative programming, you specify what computation is required, without specifying how the computer is to perform that computation.

The ***logic programming*** paradigm is arguably a special case of declarative programming. Logic programming is based on predicate logic. Example: Prolog, SQL, Microsoft Excel.

Concurrent programming cuts across imperative, object-oriented, and functional paradigms. Example: Erlang

Scripting programming is a very “high” level of programming. Rapid development; glue together different programs. Often dynamically typed, with only int, float, string, and array as the data types; no user-defined types. Very popular in Web development. Especially scripting active, web pagestomated reasoning, database applications. Recent trend: declarative programming. Example: Java#.

Self-Assessment Exercise(s) 5

1. Which program paradigms do you know?

4.0 Conclusion

This unit took you through the basic concept of programming languages and the overview of main paradigms of programming. Relationships or interaction among data type and data structures were also stressed. In the subsequent units, you are going to learn how make compilation process.

5.0 Summary

1. Each programming language can be thought of as a set of formal specifications concerning syntax, vocabulary and meaning.
2. These specifications usually include: data types, data structures, instruction and control flow, design philosophy, compilation and interpretation.
3. Several paradigms, or "schools of thought", have been promulgated regarding how best to program computers.

6.0 Tutor-Marked Assignment

1. Process of translation mathematical task or applied task on computer language is called...
2. Description structure of data and algorithms solving task in the programming language is called...
3. Formal notation for specifying computations, independent of a specific machine is called...
4. Typically represent information in the real world such as names, bank accounts and measurements is called...
5. What is data structure?
6. How is data being stored in modern digital computers?
7. The process of transformation of program text to machine code is named as .....
8. Program, carrying out translating of texts from one language to other is called...
9. Software product, executing the created program by a simultaneous analysis and realization of the prescribed actions is called.....
10. Which program paradigms do you know?

7.0 References/Further Reading

Design Concepts in Programming Languages. Franklyn Turbak, David Gifford and Mark A. Sheldon, 2008.

Concepts of Programming Languages (10th Edition) Robert W. Sebesta, 2012

Introduction to the Theory of Computation, Michael Sipser, 2012

Unit 2

Overview of Programming Languages and Compilation Process

Contents

1.0 Introduction

This unit introduces programming modules, allocation of program, processes compilation, translation and interpretation. Also, you are going to learn how to found errors and types of errors.

You will know more about object-oriented programming languages, structure of program's codes high level languages.

2.0 Learning Outcomes

At the end of this unit, you should be able to:

1. Understand how to allocate program in memory of computer?
2. Learn programmatic modules in object-oriented language
3. Understand types errors in programming languages?

3.0 Learning Contents

3.1 Allocate Program in Memory and Compiling

This is a formal notation for specifying computations, independent of a specific machine. Set of imperative commands used to direct computer to do something useful. Compilation process in figure 1.1.

Source code – program in programming language.

Object code – result of compilation text of program.

Executable code – program in operating memory.

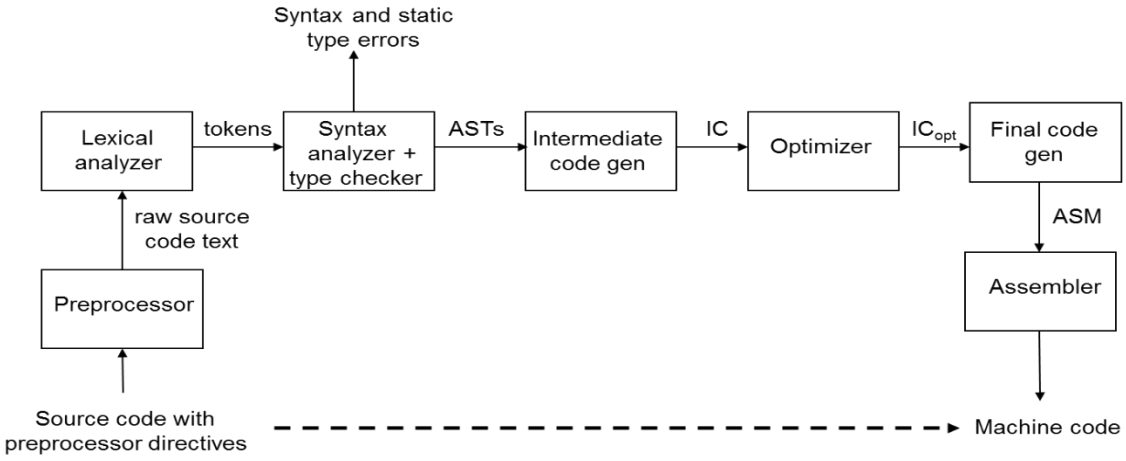


Fig. 1.1 Compilation process

Self-Assessment Exercise(s) 1

1. What is source code?
2. Result of compilation text of program is .....
3. Program in operating memory is....?

3.2 Programmatic Modules

Programmers writes the program in high-level language, i.e. most comfortable for the record of algorithm of decision of certain class of tasks. Source code of the program, entered by means of keyboard in memory of computer - initial module (source code, in a language C++ has ***expansion *.cpp***).

A translator is the program, carrying out translating of texts from one language to other, i.e. a translator translates the program from the input language of the system of programming into the absolute language of computer, on which this system functions or the developed program will function; or into intermediate language of programming, already realized or subject to realization. One of varieties of translator is a compiler, providing translating of the programs from a high-level language (close to the man) into the language of more low level or computer's language.

An interpreter is a software product, executing the created program by a simultaneous analysis and realization of the prescribed actions. At the use of interpreter dividing absents into two stages - translation and implementation.

Most translators of language of C++ which we will work with are compilers. Result of treatment of the initial module a compiler - objective module (object code, in a language C++ has ***expansion *.obj***). It cannot be executed, i.e. it is the uncompleted variant of the machine program, as, and for example, to him the modules of standard libraries must be added. Here a compiler is a type of translator, presenting program - translator source module in the language of computer instructions.

The executable (absolute, loading) module is created by the second special program - "arrange". It is yet named a link (Linker) editor. It creates the module, suitable for implementation on the basis of one or a few object modules (Figure 1.2).

Load module, ***expansion *.exe*** is the programmatic module, presented in a form, suitable for loading in memory and running.

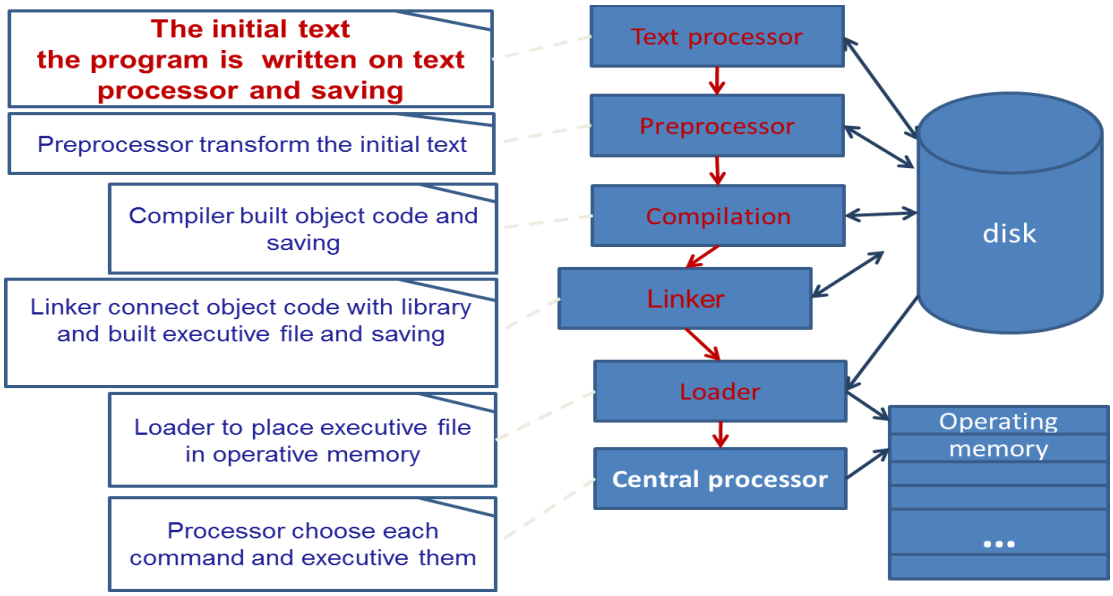


Fig. 1.2 Structure realization of compilation

Self-Assessment Exercise(s) 2

1. Which expansion is source code in a language C++?
2. Which expansion is object code in a language C++?
3. Which expansion is load module?

3.3 Errors

Errors, assumed at writing of the programs, divide on syntactic and logical. ***Syntactical errors*** are violation of formal rules of writing of the program in concrete language, revealed on the stage of translation and can be easily corrected (Figure 1.3).

Logical errors are divided by the errors of ***algorithm*** and ***semantic errors*** - can be found and corrected only by a program developer.

Reason of error of algorithm is disparity of the built algorithm to motion of receipt of end-point of the formulated task.

Reason of semantic error is the wrong understanding of sense (semantics) of operators of language.

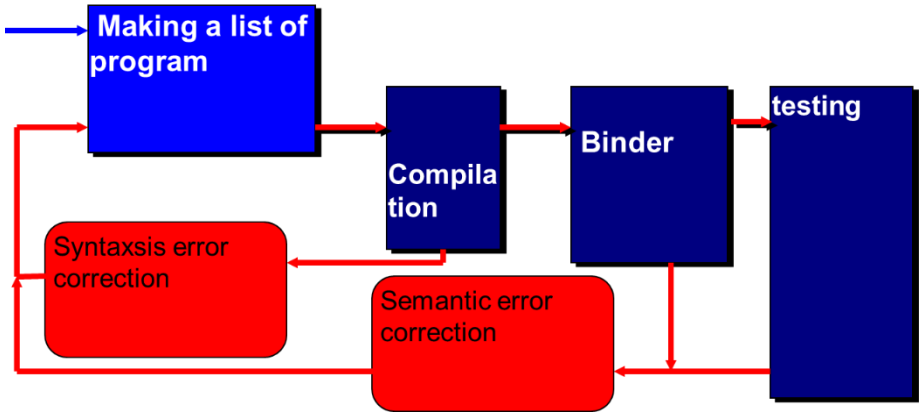


Fig. 1.3 Error corrections

Self-Assessment Exercise(s) 3

1. Errors are divided on …?
2. Logical errors are divided on…?

3.3 Functional and module to the decoupling

For most tasks algorithms of their decision are enough large and bulky. At programming it is needed to try to get the program easy-to-read, high-efficiency and easily modified. For this purpose, make decoupling of difficult put problem algorithm, i.e. his laying out on separate more simple subtask, after decouplig of subtask. For this purpose, use the receptions of the procedure-oriented programming.

One of basic receptions is laying out of algorithm on separate functions and/or modules, using functional and/or module to the decouplig accordingly.

A ***functional decoupling*** is a method of laying out of the large program on separate functions, i.e. general algorithm - on separate steps which then and design as separate functions.

The algorithm of decoupling can be presented as follows:

1. To do the program as a sequence of more shallow executions;
2. Every working out in detail in detail to describe;
3. To present every working out in detail as an abstract operator which must simply determine a necessary action, and in the end these abstract actions will be substituted by the groups of operators of the chosen programming language. It is thus necessary to remember that every working out in detail - it one of variants of decision, and it is necessary to check, that:
4. Decision of private tasks leads to the decision of general task;
5. Chosen sequence of executions is reasonable;
6. Built decoupling allows to get commands, programming easily realized in chosen language.

Unit of compiling in the language of C++ is a ***separate file (module)***. A module decoupling is breaking up of the program on a few separate files, each of which decides a separate concrete task and, as a rule, facilitates the process of the work. In addition, a program code, divided into separate files, allows part of this code to use in other programs.

Self-Assessment Exercise(s) 4

1. How do we call method of laying out of the large program on separate functions?
2. Unit of compiling in the language of C++ is…

4.0 Conclusion

This unit took you through allocation program in memory and compiling, programmatic modules and errors. This purpose of make decoupling of difficult put problem algorithm, i.e. his laying out on separate more simple subtask, after decoupling of subtask. For this purpose use the receptions of the procedure-oriented programming. In the subsequent units, you are going to learn fundamental programming constructs language C/C++.

5.0 Summary

1. This unit about compilation process and decoupling of difficult put problem algorithm, i.e. Separate more simple subtask, after decoupling of subtask.
2. For this purpose, use the receptions of the procedure-oriented programming. Unit of compiling in the language of C++ is file (module).

3. A module decoupling is breaking up of the program on a few separate files a separate, each of which decides a separate concrete task and, as a rule, facilitates the process of the work.
4. In addition, a program code, divided into separate files, allows part of this code to use in other programs.

6.0 Tutor-Marked Assignment

1. What is source code?
2. Result of compilation text of program is .....
3. Program in operating memory is....?
4. Which expansion is source code in a language C++?
5. Which expansion is object code in a language C++?
6. Which expansion is load module?
7. Errors are divided on ...?
8. Logical errors are divided on...?
9. Method of laying out of the large program on separate functions is called.....?
10. Unit of compiling in the language of C++ is.....?

7.0 References/Further Reading

Practical Guide to Computer Simulations. A.K. Hartmann, 2009.

A history of modern computer. Paul E. Ceruzzi. 2003

Objective-C. Jiva DeVoe. John Wiley & Sons. 2011

Introduction to the Theory of Computation by Michael Sipser, 2012

Module 2

Fundamental Programming Constructs

Unit 1. Syntax and semantics of programming language C/C++

Unit 2. Structures of simple programs

Unit 3. Function

Unit 1

Syntax and Semantics of Programming Language C/C++

Contents

1.0 Introduction

This unit introduces simple operators of programming language C/C++. Also, you will learn fundamental programming construction. Also, you are going to learn how to create simple program in high level language.

You will know more about object-oriented programming languages C/C++, operators for input and output, structure of programs and develop simple program.

2.0 Learning Outcomes

At the end of this unit, you should be able to:

1. Explain roles of syntax and semantic of programming language C/C++?
2. Describe basic types of data, constants?
3. Know how to use main operators, expressions in program?
4. To develop simple input /output of data
5. Use standard library to calculate mathematical expression.

3.0 Learning Contents

3.1 Syntax and Semantic of Programming Language C/C++

In the language of C/C++ a fundamental concept is instruction (operation, operator, function) which is description of certain set of actions. Thus, the program, written in language of C/C++, consists of sequence of instructions. The alphabet of language C/C++ includes:

1. capital and string letters;
2. Arabic numerals from 0 to 9;
3. special characters: +(plus) =(equal)>(anymore) <(less than) ;(semicolon) &; [ ](square brackets) { }(figured brackets) ()(parentheses(blank) .(point) ,(comma) :(colon) ?(question-mark) !(exclamation mark) \(reverse).

From the symbols of alphabet, ***the lexemes of language*** are formed are minimum meaningful units of text in the program:

1. identifiers;
2. reserved keywords;
3. signs of operations;
4. constants;

Delimiters (brackets, point, comma, blank symbols).

The borders of lexemes are determined by other lexemes, such, as delimiters or signs of operations, and also comments.

Identifier (in future, for short - ID) ***is a programmatic entity*** (constant, variable, mark, type, function, module, field in a structure) identifier. The Latin letters, numbers and sign of underlining, can be used in an identifier; the first symbol of ID can be a letter or sign of underlining, but not number; blanks ID is shut out inwardly.

Length of identifier is determined by realization (by a version) of translator C/C++ and link (arrange) editor. A modern tendency is lifting restrictions of length of identifier.

At naming of objects it is necessary to adhere to the generally accepted agreements:

1. ID of variable usually written with lowercases, for example index (for comparison:
2. Index - it ID of type or function, and INDEX is a constant);
3. an identifier must carry some sense, explaining setting of object in the program, for example ***birth_date*** (birthday) or sum (sum);
4. If ID consists of a few words, as, for example ***birth_date***, then it is accepted either to divide words an underscore (***birth_date***) or write every next word from a capital letter (***birthDate***).

Delimiters of identifiers of objects: blanks; symbols of tabulation, line and page feed; comments.

The presence of delimiters does not influence to work of the program. In C/C++ capital and string letters are different symbols. Identifiers of Name, NAME, and name are different objects.

In C++ comments are limited to the pair of symbols /* and */, and in C++ the variant of comment, which begins symbols // and ends with a word-wrapping symbol, was entered.

An example of use of these operators:

int c, d, t = 3;

c = (++ t); //c will receive value 4, t - 4

d = (t ++); //d will receive value 4, and t – 5

Self-Assessment Exercise(s) 1

1. What are lexemes in C/C++ language
2. What is an identifier?

3.2 Basic Types of Data, Constants

Basic types of base data: standard whole (***int***), material with single exactness (***float***) and symbol (***char***). In turn, data of integer type can be short (***short***), long (***long***), and without by a sign (***unsigned***), and material - with the doubled exactness (***double***). Difficult types are arrays, structures (***struct***), associations or mixtures

(***union***), enumeration (***enum***). Data whole and material types are in certain numerical ranges because occupy the different volume of main memory table 2.1:

Table 2.1: Allocation in memory different types of data

Type of data	**Memory (byte)**	**Range of values**
Char	1	-128 ...127
Int	2	-32768...32767
Short	2(1)	-32768...32767(-128...127)
Long	4	-2147483648...2147483647
unsigned int	4	0...65535
unsigned long	4	0...4294967295
Float	4	$3,14*10^{-38}$...$3,14*10^{38}$
Double	8	$1,7*10^{-308}$ $1,7*10^{308}$

All objects necessary to declare in the beginning of the program. Two forms of declaration are thus possible:

1. Description, not resulting in allocation of memory;
2. Determination at which under an object the volume of main memory will be distinguished, in accordance with his type; an object can be at once initialized in this case, i.e. to set initial value.

Except for constants, which can be set in a source code, all objects of the program must be obviously declared on a next format:

<attributes> <list of ID of objects>;

Elements of list are divided by commas, and attributes - by delimiters.

For example: int i, j, k; float a, b;

Program objects in general case next attributes have:

<storage class> it is description of method of placing of objects in memory (static, dynamic), a visibility scope and time of life of variable (by default - auto) determines, these attributes will be considered later;

<type> it is description of mechanism of interpretation of data, i.e. it is an aggregate of information about that, how many an object needs to distinguish to memory, what kind knows and what actions information above her it admits (by default - int).

A storage class and type is attributes optional and can absent, then their values will be set by default. Examples of declaration of simple objects:

int i, j, k;

char r;

double gfd;

Data of integer type (int)

A type of int is an integer, usually corresponding to the natural size whole in the used computer. The qualificators of short and long, which it is possible use with the type of int, specify on the different sizes of whole, i.e. determine size, distinguished under variables (table 2.1) memory. Examples:

short int x;

long int x;

unsigned int x = 8; (declaration with simultaneous initialization number 8).

The attribute of int in such situations can be momentous. The attributes of signed and unsigned show, as a most significant bit of number, as sign, is interpreted or as part of number (figure 2.1). If the attribute of int is indicated only, it means - short signed int.

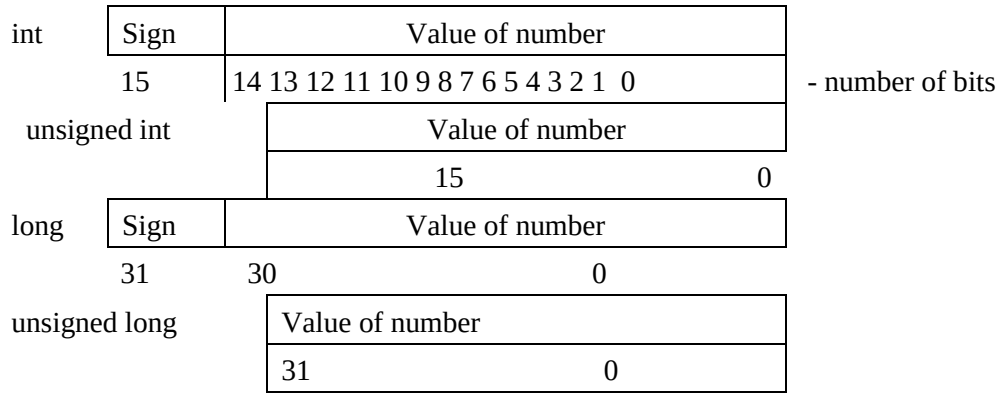


Fig.2.1. Allocation in memory integer data

Data of character type (char)

A character variable occupies in memory a 1 byte and by representative like code from 0 to 255. Fixing of concrete symbols after codes is produced by code tables. For the personal computers ASCII (American Standard Code for Information Interchange) is most widespread table of codes. Data of type of char are examined by a compiler as "whole", the use of signed char (by default) is therefore possible are symbols with codes from - 128 to +127 and unsigned char are symbols with codes from 0 to 255.

Examples: char res, simv1, simv2;

char let = 's'; (declaration with the simultaneous initializing the symbol of s).

Data of material type (float, double)

Data of material type occupy in memory, accordingly, float are 4 bytes; double - 8 byte; long double (enhanceable exactness) - 10 byte. For placing of data of type of float

usually 8 a bit is distinguished for presentation of order and sign and 24 bit under exactness (table 2.2).

Table 2.2 Allocation material types

Type	**Exactness**	**Order**
float	7 numbers after point	± 38
double	15	± 308
Long double	19	± 4932

Constants

Constants are objects, not subject to the use in the left part of operator of appropriating, as a constant - is a non-addressable size and, although is kept in memory of computer, there is no method to know this address. In the language C/C++ constants it is been:

1. Constitute oneself arithmetic, symbol and string data;
2. Identifiers of arrays and functions;
3. Elements of enumerations.
4. Arithmetic constants can be whole or material types.

Integer's constants

General format: ±n (+ not put usually).

Decimal constants are a sequence of numbers 0..9, first from which must not be 0. For example, 22 and 273 are ordinary whole constants, if it is needed to enter a long whole constant, then the sign of L(l) is specified - 273L (273l). For such constant it will be taken are 4 bytes. Ordinary whole constant which is too long for the type of ***int*** examined as long.

Constants of material type

These constants take place in memory on the format of double, and in external representation can have two forms:

1) With the fixed decimal point, format of record: ± n.m, where n, m - whole and shot to part of number;
2) With a floating decimal point (exponential form): ±n.mE ±p, where n, m - whole and shot to part of number, p is an order; ±0.xxxE±p is the normalized kind, for example:

$1{,}25 \cdot 10^{-8}$ = 0.125E-8 = 0.125E-8.

Examples of constants with fixed and floating points:

1.0 - 3.125 100e- 10 0.12537e+13

Character constants

A character constant is a symbol, put in single brackets: 'A ', 'x' (occupies a 1 byte). A type of char is unit of int. The special sequences of symbols are similarly used, it is escape sequences, basic them:

\n new line;
\t horizontal tabulation;
\0 zero symbol (emptily).

At appropriating of character variable these sequences must be celled in apostrophes. Character constant '\of 0 ', representing a symbol 0 (zero - emptily), often written down instead of whole constant 0, to underline symbol nature of some expres. Text symbols are directly entered from a keyboard, and the special and managing appear in a source code the pair of text symbols. Examples of presentation of the special characters of language C/C++ :

\\ Back slash;
' Apostrophe;
" Quotation marks.
'A', '9 ', '$', '\n ', '\72'.

String constants

A string constant is a sequence of symbols of code of ASCII, prisoner in quotation marks (") . In an internal form to the sign constants a zero symbol '\of 0 ', is added the yet called zero-terminator, marking an end-of-raw. Quotation marks are not part of line, and serve only for her limitation. A line is an array, consisting of symbols. Internal form of constant: "01234\0ABCDEF".

'0 ',' 1 ',' 2 ',' 3 ',' 4 ','\0 ',' A ',' B ',' C ',' D ',' E ',' F ','\0'

Examples of string constants:

"System", "\n\t Argument \of n", "State \of "WAIT\""

In the end of string constant, a compiler places 0-simbol automatically. 0 - simbol is a not number 0, he on printing does not hatch and in the table of code of ASCII has a code 0.

For example, a line " " is null string.

Self-Assessment Exercise(s) 2

1. What are the basic types of data in language C/C++?
2. How many bytes in memory of computer take integer data?
3. How many bytes in memory of computer take double data?

3.3 Review of Operations, Expressions

The operations of language C/C++ are intended for the management of data and necessary to know: syntax; priorities (15 levels); an order of implementation.

Arithmetic operations

Arithmetic operations are binary. List of arithmetic operations and their denotations:

+ Addition;
- Deduction (operation is a change of sign);
/ Division (for int operands - with the casting-out of remain);
* multiply
% remain from partaking of integers operands with the sign of the first operand (division on the module).

The operands of traditional arithmetic operations (+ - * /) can be constants, variables, identifiers of functions, elements of arrays, pointers, any arithmetic expressions.

Order of implementation of operations:

Expressions in parentheses;
Functions (standard mathematical, functions of user);
- * / executed from left to right;
- + ? From left to right.

The order of implementation of operations can be determined by parentheses, then expression in brackets is executed first (from left to right) of all.

Single operations + and - possess the highest priority, certain only for whole and material operands, "+" carries informative character only, "-" changes the sign of value of operand on opposite (not address operation)

Thus, because operations *, /, % possess top priority above operations +, -, at the record of difficult expressions it is needed to use the generally accepted. Example mathematical rules: *x+y*z - a/b ? x+(y*z) -(a/b)*

Operations of appropriating

Format of operation of appropriating:

< ID > = <expression>;

Value assignment is in the language C/C++, unlike traditional interpretation examined as expression, mattering the left operand after appropriating. Thus, appropriating can include a few operations of appropriate. For example:

```
int i, j, k;
float x, y, z;
    ...
i = j = k = 0;                  =    k = 0; j = k; i = j;
x = i+(y = 3) - (z = 0);    =    z = 0; y = 3; x = i + y - z;
```

Attention, the left operand of operation of appropriating can be only named or by implication addressed by a pointer variable. Examples of impermissible expressions:

a) Appropriating to the constant: 2 = x+y;

b) Appropriating of function: getch() = i;

c) Appropriating to the result of operation: (i+1) = 2+y;

Two varieties of reductions of record of operation of appropriating are assumed:

a) Instead of record: v = v @ e;

Where @ is an arithmetic operation or operation above bit presentation of operands, it is recommended to use the

Record *of v @ = e;*

For example, *i = i + 2; ? i + = 2;*

b) instead of record (autoincrease) *: x = x # 1;*where # is a symbol + or -, designating the operation of increment , x is a integer variable, variable-pointer), it is recommended to use a record *prefix: ##x; postfix: x##;*

There can be operands of different types at implementation of operations. In this case they will be transformed to the general type in accordance with the small set of rules.

The types of operands will be transformed in order of increase of their "size of memory", i.e. volume of memory, necessary for storage of their values. It is therefore possible to talk that non-obvious transformations always go from "less" objects to "large" one. Chart of implementation of transformations of operands of arithmetic operations:

short, char → int → unsigned → long → double float → double

In any expression type conversion can be carried out obviously. For this purpose, it is enough before any expression to put the identifier of corresponding type in brackets.

Type of record of operation: (*type) expression*;

The result is a value of expression, regenerate to the set type of presentation.

The operation of bringing a type over forces a compiler to execute the indicated transformation, but responsibility for consequences laid on a programmer. It is recommended to use this operation in exceptional cases. For example:

float x;

int n=6, k=4;

1) x=(n+k)/3; - fractional part will be throw-away

2) x(n+k)/3; - the use of operation of bringing a type over here allows to avoid rounding off of result of division of integer operands.

Operations of comparison

== - equal or equivalently;

!= - not equal;

< - less than;

<= - less than or equal;

> - anymore;

>= - anymore or equal.

Dividing the pair of symbols of corresponding operations is impossible. General view of operations of relations:

<expression1> <sign_of operation> <expression2>

General rules:

operands can be any base (scalar) types;

the values of operands after a calculation before comparison will be transformed to one type;

a result of relational operator is an integer value 1, if a relation is true, or 0 otherwise.

Boolean operations

List of boolean operations in order of decrease of relative priority and their denotation:

! denial (logical it not);

&& (logical AND);

|| (logical OR).

General view of operation of denial: *<expression>*

For example:

y>0 && x=7 → truth, if 1 and 2 expressions are true;

e>0 || x=7 → truth, if even one expression is true.

An unzero value of operand is interpreted as "truth", and zero is a "lie".

For example:

!0 →*1*
!5 →*0*
x=10;
!((x=y)>0) →*0*

Operation, (comma)

This operation is used for organization of the strictly assured sequence of calculation of expressions. Form of record: *expression1, …., expressionN* is calculated assuredly consistently and the value of expression of N becomes the result of operation.

Example:

m=(i=1, j=i++, k=6, n=i+j+k); the same operation *i =1; j = i; i++; k = 6; n = i+j+k; m = n;*

Self-Assessment Exercise(s) 3

1. What are the results of these operations: *x* = y; i+ = 2; x/ = y+15*;
2. State the result of this fragment in a program:

```
int n,a,b,c,d;
n = 2; a = b = c = 0;
a = ++n;
```

3.4 Simple Input/Output of Data

Any program uses data of certain types. For placed data on memory of compute use operators of input/output of data. Input data can be with:

1. Keyboard
2. From file of data

Output data can be on:

1. Screen of computer
2. File
3. Paper or other material medium.

Method for input data from keyboard is called ***input***. Method for output data – result of work program is called ***output***.

The identifier to represent output operator on style C++ is - ***cout***. This is fetched from the include library ***iostream.h***

```
// Example C ++ program
#include <iostream.h>
main ()
{ cout<<"C++ language. Example\n"; }
```

The very first property which possesses C ++ in comparison with usual C is console input/output. In language C it is realised by of standard functions ***printf ()*** and ***scanf ()***, and file input/output as ***fprintf ()*** and ***fscanf ().*** It see as macros. C ++ console input/ output is realised also as well as C on the basis standard streams of input/output (stdin/stdout in C) **cout** and **cin** and by means of the overloaded operators of digit-by-digit shift << and >>.

Let's consider a simple example of the program on C ++ which carries out console input and output.

```
//translate of inches to sm
#include <iostream.h>
int main()
{
 int inches=0;
 cout<<"Inches=";
 cin>>inches;
```

```
 cout<<inches<<" inches="<<inches*2.54<<" cm\n";
 return 0;
}
```
Inches=10
10 inches=25.4 cm
Use ***cout*** for output of numbers:
```
#include <iostream.h>
void main(void)
{
  cout << 1001;
   cout << 0.12345;
// you must out integer and real number as constant and as value of variable
}
```
Output of several values at one time. As you already know, the double sign is << (this operation inserts symbols into a target stream).
```
#include <iostream.h>
void main(void)
{
  cout << 1 << 0 << 0 << 1;
  cout << " Results == " << 1001;
  cout << "If you have  " << 100 << " friends " << " that you is rich man" << endl;
}
```
If it is necessary to move the cursor to beginning of next line, it is possible to place symbol of a new line (**\n**) in a target stream. C ++ have two different ways of generation of a new line. First, you can place symbols **\n** in a symbolical line. And second, you must use operator **endl**

You can use the special symbols on previous table for drive output. The remark: At use of the special symbols listed in tab. 1, you should have them in unary commas. if you use the given symbols itself, for example '\n ', or in double commas if you use them in a line, for example "Goog day\nWorld!".

If you need make correct arrangement of blanks, you must use the modifier **setw** specify the minimum quantity of symbols occupied with number. To use the modifier **setw**, your program should switch on a heading file **iomanip.h**:

```
#include <iostream.h>

#include <iomanip.h>

void main (void)

{

  cout << "My lovely number is " << setw(3) << 1001 << endl;
  cout << " My lovely number is " << setw(4) << 1001 << endl;
  cout << " My lovely number is " << setw(5) << 1001 << endl;
  cout << " My lovely number is " << setw(6) << 1001 << endl;
```

```
}
```

```
My lovely number is1001
My lovely number is1001
My lovely number is 1001
My lovely number is   1001
```

With using setw output formatted the multiplay table

```
#include <iostream.h>
#include <iomanip.h>
void main (void)
{
  cout << "My table " << endl;
for (int i=0; i<=30; i++) {
 cout << i << setw (5) << i<<setw (3)<<“*”<< i<<setw(3)<<“=”<< i*i  << setw(3)<<
endl;
  }
```

Input from the keyboard can be followings ways for input data: from keyboard and from file. Example:

```
#include <iostream.h>
void main(void)
{
  int number;   II Number
  cout << "Input number and press  Enter: ";
  cin >> number;
  cout << "Your number is  " << number << endl;
}
```

The following program requests of you two numbers. The program appropriates numbers to variables first and second. Then the program output numbers, using ***cout***:

```
#include <iostream.h>
void main(void)
{
  int first, second; // Numbers
  cout << "Input two numbers and press  Enter: ";
  cin >> first >> second;
  cout << "You input following numbers " << first << " and " << second << endl;
}
```

If necessary input two or large variable with use ***cin*** that each variable divide symbols >>

```
cin>> first>> second;
cin>> first>> second>> third;
```

This program calculate speed of body

```
#include <iostream.h>
void main(void)
{
```

```
    const float g=9.8;
    float h;
    cout << "Please, enter the value of height (m): ";  cin >>h;
    float v=sqrt(2.0 * g * h);
    cout << "Calculated value of velocity (m/s) is  " << v << endl;
    _getch();
    return 0;
}
```

Self-Assessment Exercise(s) 4

1. How can input and output data be represented in C/C++ language?
2. What do you see in the screen after run this program?

```
#include <iostream.h>
void main(void)
{
cout << "f";
cout << 1 << '\n' << 0 << '\n' << 0 << '\n' << 1;
cout << "Now... the end " << endl    << "teach  C++";
}
```

3.5 Standard Library

In any program except for operators and operations facilities of libraries, entering in programming which facilitate creation of the programs are used. Part of libraries - standardized and is supplied with a compiler.

Functions, macros, global constants, are included in a standard library. It is files with expansion *.h, kept in the folder of include.

The mathematical functions of algorithmic language are declared in the files <***math.h***> and <***stdlib.h***>. In subsequent records the arguments of x and y have a type of double; the parameter of n has a type of int. The arguments of trigonometric functions must be set in radians (2 π radian = 3600). Most mathematical functions (resulted here) are returned value (result) of type of double (table 2.3).

Table 2.3 Mathematical function

MATHEMATICAL FUNCTIONS	**FUNCTION IN C/C++**
$\sqrt{x}$	sqrt(x)
|x|	fabs(x)
e^x	exp(x)
x^y	pow(x,y)
ln(x)	log(x)
$lg_{10}(x)$	log10(x)

sin(x)	sin(x)
cos(x)	cos(x)
tg(x)	tan(x)
arcsin(x)	asin(x)
arccos(x)	acos(x)
arctg(x)	atan(x)
arctg(x / y)	atan2(x)
sh(x)=0.5 (e^x-e^{-x})	sinh(x)
ch(x)=0.5 (e^x+e^{-x})	cosh(x)
tgh(x)	tanh(x)
remain from the division of x on y	fmod(x,y)
the least unit >=x	ceil(x)
most unit	floor(x)

Self-Assessment Exercise(s) 5

1. Develop the program for calculation of expression:

$$v = \left(a * \sqrt{\left|\sin\left(b * c + a\right)\right|} - e^{-a*c}\right) / \sqrt{\left|2 * b + d\right|}$$

4.0 Conclusion

This unit took you through fundamental programming construction of programming language C/C++. In the language C/C++ a fundamental concept is instruction (operation, operator, function) which is description of certain set of actions. Thus, the program, written in language C/C++, consists of sequence of instructions. In the subsequent units, you are going to learn main decomposition algorithms programming language C/C++.

5.0 Summary

1. Program in language C/C++use any value for calculation.
2. This value must have some different types of data.
3. We learn for input and output data.
4. For correctly work operators of input/output necessary include standard library.

6.0 Tutor-Marked Assignment

1. What does lexemes of language C/C++ includes?
2. What is an identifier?
3. Which basic types of base data in language C/C++?
4. How many bytes are in memory of computer take integer data?
5. How many bytes are in memory of computer take double data?
6. Which constants do you know in language C/C++?

7. How to write this operator: x* = y; i+ = 2; x/ = y+15;
8. What is the result of fragment this program:

```
int n,a,b,c,d;
n = 2; a = b = c = 0;
a = ++n;
```

9. How can input and output data in language C/C++ be represented?
10. Develop the program for calculation of expression:

$$v = \left(a * \sqrt{\left|\sin\left(b * c + a\right)\right|} - e^{-a*c}\right) / \sqrt{\left|2 * b + d\right|}$$

7.0 References/Further Reading

C++ for Programmers. Kalnay - Smashwords, 2012

C++ Reference Guide. Danny Kalev - Informit, 2008

Data Structures and Algorithm Analysis in C++. Clifford A. Shaffer - Dover Publications, 2012

C++ Annotations . Frank B. Brokken - University of Groningen, 2008

Mastering C++. K. R. Venugopal.2008

Object Oriented Programming with C++, Balagurusamy.2006

A Complete Guide to Programming in C++. Ulla Kirch-Prinz, Peter Prinz, 2009 8. Object-Oriented Programming: Using C++ for Engineering and Technology. Goran Svenk. 2008

Unit 2

Structures of Simple Programs

Contents

1.0 Introduction

This unit introduces the simple operators of programming language C/C++. Also, you will learn fundamental programming construction – condition statements and cycle construction. Also, you are going to learn how to create simple program in high level language.

You will know more about object oriented programming languages C/C++, structure of programs and develop simple program.

2.0 Learning Outcomes

At the end of this unit, you should be able to:

i. State which operator to use in conditional construction of programming language C/C++?
ii. Define cycle construction?
iii. Differentiate between a pointer and a reference?
iv. To develop program with iteration process?

3.0 Learning Contents

3.1 Conditional Statements

There are two varieties of conditional statements: simple and complete. Syntax of simple statement of conditional implementation: *if (expression) operator1;*

If expression in brackets not zero, i.e. truly, then оператор1 is executed, it is ignored otherwise. Operator1 - simple or component (block).

Examples of record:

```
if (x>0) x=0;
if (i!=1)  j++,  s=1;
      if (i!=1) { j++;  s=1; }  -;
            if (getch()!=27)
                  k=0;
                  }
            if (i) exit(1);  ↔     if (i!=0) exit(1);
            if (i>0)
if (i<n) k++;          ↔     if ((i>0)&&(i<n)) k++;
            if (1) i=0;      ↔     i=0;
```

For example, using if:

int main() // determines is the entered number greater than 100

{

```
   int x;
   cout << "\nEnter a number: ";
   cin >> x;
   if( x > 100 )
     cout << "That number is greater than 100\n"<<;
   return 0;
}
```

Syntax of complete operator of conditional implementation:

if (expression) operator1; else operator2;

If expression in brackets not zero (truth), then оператор1 is executed, otherwise - оператор2. Operators 1 and 2 can be simple or component.

Examples of record:

```
if (x>0) j=k+10;
                    else m=i+10;
```

If there is the inlaid sequence of operators of if - else, then else contacts with the nearest previous if, not containing else. For example:

```
if (n>0)
                if(a>b)  z=a;
                    else   z=b;
or
if(n>0)
{ if (a>b)  z=a;  }
                else   z=b;
or
   if (expression1)  operator1;
                else if (expression 2)  operator2;
                    else if (expression 3)  operator 3;
                            else   operator 4;
```

Example:

```
int main() // more complicated program to guess a magic number
{
...
 if (guess == magic)
 {
    cout << endl << "** Right ** magic " ;
```

```
    cout << magic << "==" << guess << " (guess}";
 }
 else
 {
    cout << endl << "** Wrong ** ";
    if (guess > magic) cout << " too high " ;
    else cout << " too low " ;
 }
...
}
```

If some expression appears true, then the operator related to him is executed and this all chainlet ends with. Every operator can be either a separate operator or group of operators in the figured brackets. The last part with else deals with a case, when none of the checked up terms is executed. Sometimes here it is not needed to undertake no acts overt, in this case else оператор4; can be tomentous, or he can be used for control, to locate an "impossible" condition (economy on verification of terms).

Example:

```
if( n < 0 ) cout << "n negative\n" <<;
                    else if ( n==0 ) cout<<  "n =zero\n;
                        else cout<< "n positive\n;
```

Conditional operation "? of:"

The format of writing of conditional operation following:

expression a 1 ? expression 2: expression 3;

if expression 1 differently from a zero (truly), then the result of operation is expression 2, otherwise - the result of operation is expressions 3. Each time calculated only one of expressions 2 or 3.

We will write down operator of if, calculating maximum from a and b and appropriating his value of z.

```
if (a > b)  z=a;
else  z=b;
```

Using a conditional operation, this example can be written down : z = (a>b)? a: b;

A conditional operation can be used as well as any other expression. If expressions 2 and 3 have different types, then the type of result is determined on the rules of transformation.

Example:

```
int main()  // Calculates min value of a, b, c using conditional operator
{
```

```
float a=0, b=0,c=0, min=0;
    cout << "Enter  a, b, c divided by blanks:";
    cin >> a >> b >> c;
      min = a<=b ? (a<=c ? a : c) : (b<=c ? b : c);
    cout << endl << "The minimum value is " << min;
    getch ();
    return 0;
```

Self-Assessment Exercise(s) 1

1. Which value receive variable s or s1 after executing fragment of this program?

   If (d>=f)s=a; else s=c; cout <<s;?

   int d=13; int f=0xD; int g=66; int v=012; char c='A', a='D'; int s; char s1;

2. Develop the program for calculation of these expressions

$$d = \begin{cases} \ln(a*b)^2 & \text{если } a*b<0 \\ \ln(a*b) & \text{если } a*b>0 \\ 0 & \text{если } a*b=0 \end{cases}$$

3.2 Cycle Construction

Cycle is one of fundamental concepts of programming. ***Cycle*** organized reiteration of some sequence of operators. For organization of cycles the special operators are used.

Operators which are executed several times, presets, modifications of cycle and verification of condition of continuation of implementation of cycle index.

One passage-way of cycle is named ***an iteration***. Verification of condition is executed on every iteration either to the code of cycle (with a pre-condition) or after the code of cycle (with a post-condition).

List of varieties of iteration statements:

iteration statement with a pre-condition;

iteration statement with a post-condition;

iteration statement with a pre-condition and correction.

Operator with the pre-condition while

General view: *while (expression) code_of cycle;*

If expression in brackets is truth (0 is not equal), then the code_of cycle is executed. It recurs until expression will not take on a value 0 (lie). An operator, following while, is executed in this case. If expression is in brackets - falsely (0 is equal), then a cycle will be executed never.

The ***code_of cycle*** can include any amount of managing operators, related to the construction of while, taken in the figured brackets (block), if them more than one. Among these operators can be continue - passing to the next iteration of cycle and break is a loop exit.

For example, it is necessary to count the amount of symbols in a line. It is assumed that an input stream is adjusted on beginning of line. Then the count of symbols is executed as follows:

```
char ch;
    int count=0;
    while (( ch=getchar())!='\n')   count++;
```

For a loop exit while at truth of expression, as well as for an exit from other cycles it is possible to use the operator of break.

Example 1:

```
    while (1) {                          // no end
                ...
        if (kbhit()&&(getch()==27)) break;
                ...
        }
```

Example 2:

```
        ...
        while (!kbhit());
```

Iteration statement with the post-condition do - while

General view of record: *do code_of cycle of while (expression);*

The code_of cycle will be executed until "expression" is true. All, that talked higher justly and here, except for that, this cycle is always executed even once, checked up whereupon, whether he is necessary to execute once again.

Iteration statement with a pre-condition and correction of for

General view of operator:

for (expression1; expression2; expression3) code_of cycle;

The cycle of for is equivalent to the sequence of instructions:

```
    expression1;
    while (expression2)
        {
code_of cycle ..
```

expression3;

}

Expression1 is initiation of meter (initial value), Expressions 2 – condition continuations of account, Expression3 is an increase of meter. Expressions 1,2 and 3 can absent (empty expressions), but dropping symbols ";" is impossible.

For example, for adding up of first N of natural numbers it is possible to write:

sum = 0;

for (i=1; i<=N; i++) sum+=i;

Operation a "comma" is mostly used in the operator of for. She allows to plug a few initializing expressions in his specification. A previous example can be written in a kind:

for (sum=0, i=1; i<=N; sum+= i, i++);

Operators control transfer

Formally to the operators of control transfer behave:

operator of unconditional branch of goto;

switch control statement to the next step (iterations) of cycle of continue;

loop exit, or operator of switch - break;

return statement from the function of return.

Operator unconditional branch goto

The operator ***goto*** although in most cases it is possible to do without him. General view of operator:

goto < mark >;

It is intended for a control transfer on operator, noted by a mark. A mark is an identifier, executed in due form authentications of variables with a symbol "colon" after him, for example, dummy noted statement:

m1: ;

A purview of mark is a function, where this mark is certain. It is in the case of necessity possible to use a block.

The most characteristic case of the use of operator of ***goto*** is implementation of breaking (output) in the inlaid structure in case of occurring of flagrant incorrigible errors in datains. And it is necessary in this case, to go out from two (or more) cycles, where it is impossible to use directly operator of break, as he interrupts only the most inner loop:

for (...)

for (...)

```
{ ...
        if ( error )  goto error;
}
        ...
error:  - operations for missing error;
```

If an error routine is non-trivial and errors can arise up in a few places, then such organization appears comfortable.

Operator continue

This operator can be used in all types of cycles, but not in the operators of switch of switch. The presence of operator of continue causes admission of "remaining" part of iteration and passing to beginning following, i.e. pre-schedule completion of current step and passing to the next step.

In the cycles of while and do it means the direct passing to verification part. In the loop a for management is passed on the step of correction, i.e. modifications of expression 3.

Fragment of the program of treatment only of positive array of a cells, negative values are skipped:

```
for ( i = 0; i<n; i++)
{  if( a[i]<0)  continue;
  ...                    //
}
```

Appears too difficult, so that consideration of condition, reverse to checked up, results in a too high program nesting level.

Operator break

The operator ***break*** produces an urgent exit from the most inner loop or operator-switch of switch, which he belongs to, and transfers control to the first operator, to the following by a current operator.

Operator return

Operator ***return***; produces a pre-schedule exit from a current function. Similarly returns the value of result of function*: return <expression>;*

In functions, not returning a result, he is unobviously present after the last operator. The value of expression if necessary will be transformed to the type of the value returned by a function.

Example of 1:

```
  float estim(float *x, int n) {
int i;
```

```
float y;
if ((!x)||(!n) {
        error(x,n);
        return 0;
        }
for (y=i=0; i<n; i++) y+=x[i];
   return y/n;
}
```

Self-Assessment Exercise(s) 2

1. What is mean cycle?
2. What is one passage-way of cycle?
3. The operator .... produces an urgent exit from the most inner loop.
4. Operator ... produces a pre-schedule exit from a current function
5. Which operator need mark?
6. Develop the program for calculation and a conclusion to a seal values of function at X = 5, 6, 7, ..., 35; A = 10.2.

3.3 Pointers

A pointer is a variable which can contain the address of some object. A pointer is declared as follows:

*<type> *< ID of variable-pointer>;*

For example: *int *a; double *f; char *w;*

& is a unary operator - returns the memory address of its operand.

For example,

m = &count;

Places into m the memory address of the variable count. This address is the computer's internal location of the variable. It has nothing to do with the value of count an operation & means to "take an operand address". An operation * makes sense - "value, located to indicated address".

Address to the objects of any type can the operands of operations be conducted:

1. on the name (to the identifier - ID);

2. on a pointer (operation of the deferred addressing): pointer = &of ID_of object;

Example 1:

```
int x,                    // variable of type int
```

```
*y;                    // pointers to data  int
y=&x;                  // y – address of variable x
*y=1;                  // deferred addressing of pointers  x, т.е.
                       // write to address 1 → x=1;
```

Example 2:

```
int i, j=8,k=5, *y;
 y=&i;
 *y=2;                 // i=2
 y=&j;
 *y+=i;                // j+=i  →  j=j+i  →  j=j+2=10
 y=&k;
 k+=*y;                // k+=k  →  k=k+k = 10
 (*y)++;               // k++   →  k=k+1 = 10+1 = 11
```

It is necessary to take into account at the calculation of addresses of objects, that identifiers of arrays and functions are constant pointers. Such constant can be appropriated to the variable of type pointer, but it is impossible to expose to transformations, for example:

```
int x[100], *y; // Correctly is appropriating of constant to the variable of x = y;
y = x;          // Error: in the left part is a pointer-constant
```

Pointer - variable can appropriate the value of other pointer, or expressions of type pointer with the use, if necessary, operations of bringing a type over. Bringing a type over is optional, if one of pointers has a type of "void *".

```
int i,*x;
 char *y;
 x=&i;              // x  →  field of object int
 y=(char *)x;       // y  f → field of object char
 y=(char *)&i;      // y  →  field of object char
```

Example:

```
int  a;          // simple variable
int *b ;         // pointer on int type
cout << a;       //10
cout << b;       // 1004
```

*cout << *b; // 1000*

Value b interpreted as address of memory. This operator means – output value placed on address = value b (table 2.4).

Table 2.4 Memory addresses and value of variable

Memory address	Value of variable	Symbolic name of variable
1000	10	A
1002	1004	B
1004	1000	

Pointers expression

As with any variable, you may use a pointer on the right-hand side of an assignment statement to assign its value to another pointer. For example:

```
int x;
int *p1, *p2;
p1 = &x;
p2 = p1;
cout<< p2; /* print the address of x, not x's value! */
```

There are only two arithmetic operations that you may use on pointers: addition and subtraction. To understand what occurs in pointer arithmetic, let p1 be an integer pointer with a current value of 2000. Also, assume integers are 2 bytes long.

After the expression

p1++;

p1 contains 2002, not 2001. The reason for this is that each time p1 is incremented, it will point to the next integer. The same is true of decrements. For example, assuming that p1 has the value 2000, the expression p1--; causes p1 to have the value 1998.

Generalizing from the preceding example, the following rules govern pointer arithmetic.

Each time a pointer is incremented, it points to the memory location of the next element of its base type. Each time it is decremented, it points to the location of the previous element.

When applied to character pointers, this will appear as "normal" arithmetic because characters are always 1 byte long. All other pointers will increase or decrease by the length of the data type they point to. This approach ensures that a pointer is always pointing to an appropriate element of its base type.

```
char  *ch = 1000;
int *I = 1000;
```

Reference

Reference is a not type of data, and constant pointer, i.e. it is an object which specifies on position of other variable.

Reference is a constant pointer, which differs from a variable pointer that for reference the special operation of noname is not required. Above pointers arithmetic operations are possible. Above reference arithmetic operations are forbidden, as reference is declared as follows: type &ID = initializer;

Example:

```
int  a = 8;
int  &r = a;
```

Reference got the pseudonym of object indicated as an initializer. In the set pattern, identical will be next actions:

```
a++;  r++;
int a = 10;
 int b = 20;
 int &c = a;
 c = b;
```

Self-Assessment Exercise(s) 3

1. Differentiate between a pointer and a reference?

4.0 Conclusion

This unit took you through fundamental programming construction of programming language C/C++. In the language C/C++ a fundamental concept - comparison and iteration, which is description of certain set of actions. Thus, the program, written in language C/C++, consists of practical works. In the subsequent units, you are going to learn main functions and parameter passing; structured decomposition.

5.0 Summary

1. Program in language C/C++use any value for calculation. This value must have some different types of data.
2. Conditional statements, cycle construction and pointers.
3. For different operators IF, FOR. Last unit consist practical work for each students.

6.0 Tutor-Marked Assignment

1. Which value receive variable s or s1 after execute fragment of this program?

If (d>=f)s=a; else s=c; cout <<s;?

int d=13; int f=0xD; int g=66; int v=012; char c='A', a='D'; int s; char s1;

2. Develop the program for calculating this expression

$$d = \begin{cases} \ln(a*b)^2 & если\ \ a*b<0 \\ \ln(a*b) & если\ \ a*b>0 \\ 0 & если\ \ a*b=0 \end{cases}$$

3. What is mean cycle?
4. What is one passage-way of cycle?
5. The operator .... produces an urgent exit from the most inner loop.
6. Operator ... produces a pre-schedule exit from a current function
7. Which operator need mark?
8. Develop the program for calculation and a conclusion to a seal of values of function at X = 5, 6, 7, ..., 35; A = 10.2.
9. What is a pointer?
10. Differentiate between a pointer and a reference?

7.0 References/Further Reading

1. C++ for Programmers. Kalnay - Smashwords , 2012
2. C++ Reference Guide. Danny Kalev - Informit , 2008
3. Data Structures and Algorithm Analysis in C++. Clifford A. Shaffer - Dover Publications, 2012
4. C++ Annotations . Frank B. Brokken - University of Groningen , 2008
5. Mastering C++. K. R. Venugopal.2008
6. Object Oriented Programming with C++, Balagurusamy.2006
7. A Complete Guide to Programming in C++. Ulla Kirch-Prinz, Peter Prinz, 2009
8. Object-Oriented Programming: Using C++ for Engineering and Technology. Goran Svenk. 2008

Unit 3

Functions

Contents

1.0 Introduction

This unit introduces function and parameter passing of programming language C/C++. Also, you will learn how it is using in programs. Also, you are going to learn what is structured decomposition, structure of programs and develop simple program.

2.0 Learning Outcomes

At the end of this unit, you should be able to:

i. State what is a function in programming language C/C++?
ii. Declare function?
iii. State parameter of a function?
iv. State the need of structured decomposition?

3.0 Learning Contents

3.1 Declaration of Function

A prototype for a function which takes a function parameter looks like the following:

```
void func ( void (*f)(int) );
```

This states that the parameter f will be a pointer to a function which has a void return type and which takes a single ***int*** parameter. The following function (print) is an example of a function which could be passed to ***func*** as a parameter because it is the proper type:

```
void print ( int x ) {
  cout << x << endl;
}
```

When calling a function with a function parameter, the value passed must be a pointer to a function. Use the function's name (without parens) for this:

```
func(print);
```

would call ***func***, passing the ***print*** function to it.

As with any parameter, ***func*** can now use the parameter's name in the function body to access the value of the parameter. Let's say that ***func*** will apply the function it is passed to the numbers 0-4. Consider, first, what the loop would look like to call ***print*** directly:

```
for ( int ctr = 0 ; ctr < 5 ; ctr++ ) {
  print(x);
}
```

Since ***func's*** parameter declaration says that *f* is the name for a pointer to the desired function, we recall first that if f is a pointer then **f* is the thing that *f* points to (i.e. the function print in this case). As a result, just replace every occurrence of print in the loop above with **f*:

```
void func ( void (*f)(int) ) {
        for ( int ctr = 0 ; ctr < 5 ; ctr++ ) {
           (*f)(x);
         }
        }
```

The extra set of parens around *f is necessary because of the precedence of *: *f(x) would mean the thing that the value returned by f(x) points to.

Description of function consists in a coersion in the beginning of programmatic file of her prototype. The prototype of function reveals to the compiler that further in text of the program her complete determination:

type_of result of ID_of function (type of variable1, ., variable of N type);

We will notice that identifiers of variables in the parentheses of prototype to specify not necessarily, because the compiler of language them does not process.

Description of prototype enables to the compiler to check up accordance of types and amount of parameters at the actual call of this function.

Example of description of function of fun with the list of parameters:

float fun(int, float, int, int);

Complete determination of function has a next kind:

type_of result of ID_of function(list of parameters)

The type of result determines the type of expression, the value of which goes back into a her invocation point through an operator

return <expression>.

If the type of function is not indicated, then the type of **int** is assumed by default.

The list of parameters consists of list of types and identifiers of parameters, divided by commas. A function cannot have parameters, but parentheses are needed in any case. If a function returns no value, she must be described as a function of type of void (empty). In this case operator of return it is possible not to put. There can be a few operators of return in a function, but there can be none. In such cases a return in the calling program takes place after implementation of the last operator in a function. Example of function, qualificatory the least value from two integer variables:

```
int  min (int x, int y)
                {
```

```
    return (x<y)? x : y;
}
```

All functions, returning a value, must be used in right part of expressions and re-entry result will be lost otherwise. If the list of parameters absents at a function, then during declaration of such function desirably in parentheses also to specify the keyword of void. Example, void main(void).

Self-Assessment Exercise(s) 1

1. Write example the function with parameters?
2. How mane variables return this function?

```
int  min (int x, int y)
                    {
return (x<y)? x : y;
}
```

3.2 Call of Function

The call of function has a next format: *ID _of function (list of arguments)* whereas arguments it is possible to use ***constants, variables, expressions*** (their values before the call of function will be certain a compiler).

The arguments of list of call must fully coincide with the ***list of parameters*** of the caused function on an amount, in order of the following and on the types of corresponding by him parameters.

Connection between functions is carried out through arguments and values returned by functions. Function can be carried out also through external, global variables.

Functions can be situated in an initial file in any order. And the initial program can take place in a few files.

In a language C arguments at the standard call of function are passed by value, i.e. in a stack a place is distinguished for the formal parameters of function and in this distinguished place at her call the values of actual arguments are brought. Then a function uses and can change these values in a stack.

On leaving from a function the changed values are lost. The caused function cannot change the values of variable, indicated as actual arguments at an address to this function.

In the case of necessity, a function can be used for the change of transferrable by her arguments. In this case as an argument it is necessary in the caused function to pass a variable not value, and her address. And for an address to the value of argument-original.

```
void  zam (int *x, int *y)    {
```

```
int t;
t = *x;
        *x = *y;
*y = t;
}
```

Program with this function:

```
void zam (int*, int*);
void main (void)  {
int a=2, b=3;
...
printf(" a = %d , b = %d\n", a, b);
zam (&a, &b);
printf(" a = %d , b = %d\n", a, b);
...
}
```

Result of program:

a = 2 , b=3

a = 3 , b=2

To any type of data, both standard and certain an user, it is possible to set the new name by means of operation of typedef: *typedef <type> <new_name>;*

The new type entered thus is used like standard types, for example, entering user types: ***typedef*** unsigned int UINT;

typedef char M_s[100];

declarations of identifiers of the entered types look like:

UINT i, j; → two variables of type of unsigned int

M_s str[10]; ? → array from 10 lines for 100 symbols

Self-Assessment Exercise(s) 2

1. Which arguments is possible to use when call function?
2. The arguments of list of call must fully coincide with ....

3.3 Pointers on a Function

Identifier of function is a constant pointer to beginning of function in-memory and can not be a variable value. But there is possibility to declare pointers to the functions with

which it is possible to apply as with variables (for example, it is possible to create an array the elements of which will be pointers on a function).

We will consider the methods of work with pointers on a function.

1. As well as any object of language C/C++, ***pointer it is necessary to declare on a function***. The format of announcement of pointer on a function following:

type(list of parameters);

i.e. a pointer, which can be set on functions, returning the result of the indicated type and which have the indicated list of parameters, is declared. The presence of the first parentheses is obligatory, because without them is declaration of function which returns a pointer to the job of the indicated type performance and has the indicated list of parameters.

For example, announcement of kind: float (char, float); talks about that is declared pointer of p_f, which can be set on a function, return material result and having two parameters: first - character type, and second - material type.

2***. Identifier of function is a constant pointer***, therefore in an order to set a variable-pointer to the concrete function, sufficiently to appropriate her identifier her:

a variable is a pointer = of ID_of function;

For example, there is a function with a prototype: float f1(char, float); then operation of p_f = f1; the pointer of p_1 will set to this function.

3. Call functions after setting on pointer looks so:

*(*variable-pointer)(List of arguments);* or

variable-pointer (list of arguments);

After such actions except for a standard function call:

ID_of function(list of arguments);

Two methods of call of function appear:

*(*variable-pointer)(list of arguments);*or

variable-pointer (list of arguments);

Last justly, because p_f also is the address of beginning of function in-memory. For our example to the function of f1 it is possible to appeal next methods:

f1('z ', 1.5); // Function call on ID

(p_f)('z ', 1.5);* // Function call on the pointer of

p_f('z ', 1.5); // Function call on ID of pointer

4. Let there is the second function with a prototype: float f2(char, float); then reinstalling the pointer of p_f to this function: p_f = f2; we have again three methods

of her call : f2('z ', 1.5); // on ID

of function('z ', 1.5); // on a function pointer

p_f('z ', 1.5); // on ID of function pointer

The basic setting of pointers on a function is providing of possibility of transmission of identifiers of functions as parameters in a function which will realize some calculable process, using the formal name of the caused function.

It is sometimes impossible to transfer types and number of all possible parameters of function. In these cases the list of parameters appears suspension points (..) which disconnects the mechanism of verification of types. The presence of suspension points tells a compiler, that there can be an arbitrary number of parameters of unknown beforehand types at a function. Suspension points are used in two formats:

void varParFun(param_list, ..);

void varParFun(..);

The first format gives announcements for part of parameters. In this case verification of types for the declared parameters is produced, and for remaining actual arguments - no. A comma after announcement of the known parameters is optional.

Forced use of suspension points the function printf***()*** of standard library of C. exemplifies Her first parameter is C - by a line: *int printf (const char* ..);*

It guarantees that at any call ***printf()*** her the first argument of type will be passed to const char*. Maintenance of such line, called pick-up, determines whether additional arguments are needed at a call. At presence of in the line of format of symbols, begun with a symbol %, a function waits the presence of these arguments.

Most functions with suspension points in announcement get information about types and amount of actual arguments by value the obviously declared parameter. Consequently, the first format of suspension points is used more frequent.

Next announcements are unequivalent:

void f();

void f(..);

In first case f() is declared as a function without parameters, in the second - as having a zero or more than parameters. Calls: f (someValue);

f (cnt, a, b, c);

correct only for the second announcement. Call of f();

we will apply to any of two functions.

Self-Assessment Exercise(s) 3

1. Is it necessary to declare pointer on a function?
2. Identifier of function is ...

3.4 Parameters Passing

Consider a pair of C++ functions defined in Program. The function ***One*** calls the function ***Two***. In general, every function call includes a (possibly empty) list of arguments. The arguments specified in a function call are called ***actual parameters***. In this case, there is only one actual parameter - y. Example of ***Pass-By-Value Parameter Passing***.

```
void Two (int  x)
{
   x=2;
   cout<<x<<endl;
}
void One ()
{
   int  y=1;
   Two (y);
   cout<<x<<endl;
}
```

The method by which the parameter is passed to a function is determined by the function definition. In this case, the function ***Two*** is defined as accepting a single argument of type ***int*** called *x*. The arguments which appear in a function definition are called ***formal parameters***. If the type of a formal parameter is *not* a reference, then the parameter passing method is ***pass-by-value***.

The semantics of pass-by-value work like this: The effect of the formal parameter definition is to create a local variable of the specified type in the given function. E.g., the function ***Two*** has a local variable of type *int* called *x*. When the function is called, the *values* (*r*-values) of the ***actual parameters*** are used to initialize the ***formal parameters*** before the body of the function is executed.

Since the formal parameters give rise to local variables, if a new value is assigned to a formal parameter, that value has no effect on the actual parameters. Therefore, the output Pass By Reference.

The only difference between this code and the code given in program is the definition of the formal parameter of the function ***Two***: In this case, the parameter *x* is declared to be a ***reference*** to an ***int***. In general, if the type of a formal parameter is a reference, then the parameter passing method is ***pass-by-reference***. Example of ***Pass-By-Reference Parameter Passing***

```
void Two (int & x)
{
```

```
    x=2;
    cout<<x<<endl;
}
void One ()
{
    int  y=1;
    Two (y);
    cout<<x<<endl;
}
```

A reference formal parameter is not a variable. When a function is called that has a reference formal parameter, the effect of the call is to associate the reference with the corresponding actual parameter. I.e., the reference becomes an alternative name for the corresponding actual parameter. Consequently, this means that the actual parameter passed by reference must be variable.

A reference formal parameter can be used in the called function everywhere that a variable can be used. In particular, if the reference formal parameter is used where a *r*-value is required, it is the *r*-value of actual parameter that is obtained. Similarly, if the reference parameter is used where an *l*-value is required, it is the *l*-value of actual parameter that is obtained.

Constant Parameters

In general, pass-by-reference parameter passing allows the called function to modify the actual parameters. However, sometimes it is the case that the programmer does not want the actual parameters to be modified. Nevertheless, pass-by-reference may be the preferred method for performance reasons.

In C++ we can use the const keyword to achieve the performance of pass-by-reference while at the same time ensuring that the actual parameter cannot be modified. Consider the following definition of the function ***Two***:

```
void Two (int const& x)
{
   x = 2; // Not allowed.
   cout << x << endl; // This is ok.
}
```

The ***const*** keyword modifies the type ***int***. It says that the ***int*** to which *x* refers is a constant. The value of *x* can be used without impunity. However, *x* cannot be used as the target of an assignment statement. In fact, the variable x cannot be used in any context where it might be modified. Any attempt to do so is an error that is detected by the compiler.

Self-Assessment Exercise(s) 4

1. What is formal parameters?
2. What is actual parameters?
3. Write example of Pass-By-Reference Parameter Passing.
4. Why the use of constant parameters?

4.0 Conclusion

This unit took you through function of programming language C/C++. In the language C/C++ you can use function with parameters and without parameters. Thus, the program, written in language C/C++, consist examples using parameters.

5.0 Summary

1. Program in language C/C++use any functions.
2. This value must have some different types of functions.
3. Function with parameters and without parameters.
4. Consider a pair of C++ functions defined in program.

6.0 Tutor-Marked Assignment

1. Write examples of the function with parameters?
2. How mane variables return this function?

```
int  min (int x, int y)
                {
    return (x<y)? x : y;
```

3. What argument is possible to use when call function?
4. The arguments of list of call must fully coincide with ….
5. Is it necessary to declare pointer on a function?
6. Identifier of function is …
7. What is formal parameters?
8. What is actual parameters?
9. Write example of Pass-By-Reference Parameter Passing.
10. Why the use of constant parameters?

7.0 References/Further Reading

1. C++ for Programmers. Kalnay - Smashwords , 2012
2. C++ Reference Guide. Danny Kalev - Informit, 2008
3. Data Structures and Algorithm Analysis in C++. Clifford A. Shaffer - Dover Publications, 2012
4. C++ Annotations. Frank B. Brokken - University of Groningen, 2008
5. Mastering C++. K. R. Venugopal.2008
6. Object Oriented Programming with C++, Balagurusamy.2006
7. A Complete Guide to Programming in C++. Ulla Kirch-Prinz, Peter Prinz, 2009
8. Object-Oriented Programming: Using C++ for Engineering and Technology. Goran Svenk. 2008

Module 3

Algorithms and Problem Solving

Unit 1: Problem solving

Unit 2: Basic concept of an algorithm

Unit 3. Search and sorting algorithms

Unit 1

Problem Solving

Contents

1.0 Introduction

This unit introduces you to problem solving: programming, problem solving techniques, algorithm. Also, you will learn problem solving strategies. Also, you are going to learn how divide problem solving. You will know more about design steps to solve problem.

2.0 Learning Outcomes

At the end of this unit, you should be able to:

1. State steps of problem solving process.
2. Understand Problem solving strategies.

3.0 Learning Contents

3.1 Problem Solving

There are next problem of solving:

1. Programming is a process of problem solving
2. Problem solving techniques: analyze the problem; outline the problem requirements; design steps (algorithm) to solve the problem
3. Algorithm: step-by-step problem-solving process; solution achieved in finite amount of time.

Problem solving process consist 3 steps:

Step 1 - ***Analyze the problem***

- Outline the problem and its requirements
- Design steps (algorithm) to solve the problem

Step 2 - ***Implement the algorithm***

- Implement the algorithm in code
- Verify that the algorithm works

Step 3 - ***Maintenance***

- Use and modify the program if the problem domain changes

First of all, thoroughly understand the problem; understand problem requirements. Does program require user interaction? Does program manipulate data? What is the output?

If the problem is complex, divide it into subproblems then analyze each subproblem as above. The idea behind the computer program. Stays the same independent of: which kind of hardware it is running on; which programming language it is written in; solves a well-specified problem in a general way;

Is specified by: Describing the set of instances (input) it must work on; describing the desired properties of the output. Before a computer can perform a task, it must have an algorithm that tells it what to do.

Informally: "An algorithm is a set of steps that define how a task is performed."

Formally: "An algorithm is an ordered set of unambiguous executable steps, defining a terminating process."

Ordered set of steps: structure!

Executable steps: doable!

Unambiguous steps: follow the directions!

Terminating: must have an end!

Problem solving techniques are not unique to Computer Science. The CS field has joined with other fields to try to solve problems better. Ideally, there should be an algorithm to find/develop algorithms. However, this is not the case as some problems do not have algorithmic solutions. Problem solving remains an art!

Self-Assessment Exercise(s) 1

1. List the steps in problem solving process?
2. State the process in analyzing the problem?
3. What does "implement the algorithm" means?
4. What is maintenance?

3.2 Problem solving strategies

i. Working backwards: reverse-engineer; once you know it can be done, it is much easier to do; what are some examples?

ii. Look for a related problem that has been solved before: Java design patterns; sort a particular list such as: David, Alice, Carol and Bob to find a general sorting algorithm

iii. Stepwise Refinement: break the problem into several sub-problems; solve each sub problem separately; produces a modular structure

Stepwise refinement is a top-down methodology in that it progresses from the general to the specific. Bottom-up methodologies progress from the specific to the general. These approaches complement each other. Solutions produced by stepwise refinement posses a natural modular structure - hence its popularity in algorithmic design.

Four stages to the decomposition process

i. Brainstorming
ii. Filtering
iii. Scenarios
iv. Responsibility algorithms

Brainstorming - a group problem-solving technique that involves the spontaneous contribution of ideas from all members of the group. All ideas are potential good ideas. Think fast and furiously first, and ponder later. A little humor can be a powerful force. Brainstorming is designed to produce a list of candidate classes

Filtering determine which are the core classes in the problem solution. There may be two classes in the list that have many common attributes and behaviors. There may be classes that really don't belong in the problem solution.

Scenarios- assign responsibilities to each class. There are two types of responsibilities:

What a class must know about itself (knowledge)

What a class must be able to do (behavior)

Encapsulation is the bundling of data and actions in such a way that the logical properties of the data and actions are separated from the implementation details.

The algorithms must be written for the ***responsibilities***: knowledge responsibilities usually just return the contents of one of an object's variables; action responsibilities are a little more complicated, often involving calculations

Self-Assessment Exercise(s) 2

1. The spontaneous contribution of ideas from all members of the group is called…
2. … determine the core classes in the problem solution.
3. … assign responsibilities to each class
4. What do you mean by the algorithms must be written for the *responsibilities*?

4.0 Conclusion

This unit took you through problem solving: programming, problem solving techniques, and algorithm. Also, you have learned problem solving strategies. You know more about design steps to solve problem.

5.0 Summary

1. Stepwise refinement is a top-down methodology in that it progresses from the general to the specific.
2. Bottom-up methodologies progress from the specific to the general. These approaches complement each other.
3. Solutions produced by stepwise refinement possess a natural modular structure - hence its popularity in algorithmic design.

6.0 Tutor-Marked Assignment

1. State the steps involve in problem solving process?
2. State the processes analyzing the problem?

3. What is “implement the algorithm”
4. What is maintenance?
5. The spontaneous contribution of ideas from all members of the group is called…
6. . … determine the core classes in the problem solution.
7. … assign responsibilities to each class
8. What do you mean by the algorithms must be written for the *responsibilities*?

7.0 References/Further Reading

1. Data Structures and Algorithm Analysis in C++. Clifford A. Shaffer - Dover Publications, 2012
2. Introduction to Algorithms. T Cormenr, C Leiserson, R Rivest, C Stein, 2009
3. Machine Learning Algorithms for Problem Solving in Computational Applications. Siddhivinayak Kulkarni. 2012

Unit 2

Basic Concept of an Algorithm

Contents

1.0 Introduction

This unit introduce you to algorithm and characteristics of algorithms. The characteristics of algorithms are presented in this chapter, so are the two forms of representation for algorithms, namely, pseudo-codes and flowcharts. The three control structures: sequence, branch, and loop, which provide the flow of control in an algorithm, are introduced.

2.0 Learning outcomes

At the end of this unit, you should be able to:

1. What is an algorithm?
2. Characteristics of an algorithm.
3. Using pseudo-codes and flowcharts to represent algorithms.

3.0 Learning Contents

3.1 What is an Algorithm?

An algorithm is a well-defined computational procedure consisting of a set of instructions, that takes some value or set of values, as input, and produces some value or set of values, as output. In other word, an algorithm is a procedure that accepts data, manipulate them following the prescribed steps, so as to eventually fill the required unknown with the desired value(s) (Figure 3.1).

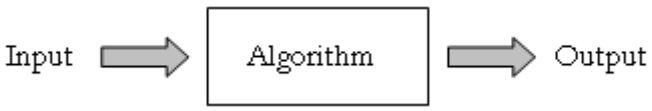


Fig.3.1 what is an algorithm?

Put tersely, an algorithm, a jargon of computer specialists, is simply a procedure. People of different professions have their own form of procedure in their line of work, and they call it different names. A cook, for instance, follows a procedure commonly known as a recipe that converts the ingredients (input) into some culinary dish (output), after a certain number of steps.

Algorithms and their alter ego, programs, are the software. The machine that runs the programs is the hardware. Referring to the cook in our analogy again, his view can be depicted as follows:

The first documented algorithm is the famous Euclidean algorithm written by the Greek mathematician Euclid in 300 B.C. in his Book VII of the Elements. It is rumoured that King Ptolemy, having looked through the Elements, hopefully asked Euclid if there were not a shorter way to geometry, to which Euclid severely answered: “In geometry there is no royal road!”

The modern Euclidean algorithm to compute gcd (greatest common divisor) of two integers, is often presented as followed.

1. Let A and B be integers with $A > B \geq 0$.
2. If $B = 0$, then the gcd is A and the algorithm ends.
3. Otherwise, find q and r such that

   $A = qB + r$ where $0 \leq r < B$

Note that we have $0 \leq r < B < A$ and $\gcd(A,B) = \gcd(B,r)$. Replace A by B, B by r. Go to step 2.

Walk through this algorithm with some sets of values.

In algorithmic problem solving, we deal with objects. Objects are data manipulated by the algorithm. To a cook, the objects are the various types of vegetables, meat and sauce. In algorithms, the data are numbers, words, lists, files, and so on. In solving a geometry problem, the data can be the length of a rectangle, the area of a circle, etc. Algorithm provides the logic; data provide the values. They go hand in hand. Hence, we have this great truth:

Program = Algorithm + Data Structures

Data structures refer to the types of data used and how the data are organized in the program. Data come in different forms and types. Most programming languages provides simple data types such as integers, real numbers and characters, and more complex data structures such as arrays, records and files which are collections of data.

Because algorithm manipulates data, we need to store the data objects into variables, and give these variables names for reference. For example, in mathematics, we call the area of a circle A, and express A in terms of the radius r. (In programming, we would use more telling variable names such as area and radius instead of A and r in general, for the sake of readability.) When the program is run, each variable occupies some memory location(s), whose size depends on the data type of the variable, to hold its value.

Self-Assessment Exercise(s) 1

1. What is an algorithm?
2. Algorithm + Data Structures=....

3.2 Characteristics of an Algorithm

What makes an algorithm an algorithm? There are four essential properties of an algorithm.

1. ***Each step of an algorithm must be exact***.

This goes without saying. An algorithm must be precisely and unambiguously described, so that there remains no uncertainty. An instruction that says “shuffle the deck of card” may make sense to some of us, but the machine will not have a clue on how to execute it, unless the detail steps are described. An instruction that says “lift the restriction” will cause much puzzlement even to the human readers.

2. ***An algorithm must terminate***.

The ultimate purpose of an algorithm is to solve a problem. If the program does not stop when executed, we will not be able to get any result from it. Therefore, an algorithm must contain a finite number of steps in its execution. Note that an algorithm that merely contains a finite number of steps may not terminate during execution, due to the presence of 'infinite loop'.

3. ***An algorithm must be effective***.

Again, this goes without saying. An algorithm must provide the correct answer to the problem.

4. ***An algorithm must be general.***

This means that it must solve every instance of the problem. For example, a program that computes the area of a rectangle should work on all possible dimensions of the rectangle, within the limits of the programming language and the machine.

An algorithm should also emphasize on the whats, and not the hows, leaving the details for the program version. However, this point is more apparent in more complicated algorithms at advanced level, which we are unlikely to encounter yet.

Self-Assessment Exercise(s) 2

1. Each step of an algorithm must be….
2. What do you mean by an algorithm must terminate?
3. What is an effective algorithm?
4. What do you mean by an algorithm must be general?

3.3 Pseudo-Codes and Flowcharts

We usually present algorithms in the form of some ***pseudo-code***, which is normally a mixture of English statements, some mathematical notations, and selected keywords from a programming language. There is no standard convention for writing pseudo-code; each author may have his own style, as long as clarity is ensured. Below are two versions of the same algorithm, one is written mainly in English, and the other in pseudo-code. The problem concerned is to find the minimum, maximum, and average of a list of numbers. Make a comparison.

Algorithm version 1:

First, you initialize *sum* to zero, *min* to a very big number, and *max* to a very small number. Then, you enter the numbers, one by one.

For each number that you have entered, assign it to *num* and add it to the *sum*.

At the same time, you compare *num* with *min*, if *num* is smaller than *min*, let *min* be *num* instead.

Similarly, you compare *num* with *max*, if *num* is larger than *max*, let *max* be *num* instead.

After all the numbers have been entered, you divide *sum* by the numbers of items entered, and let *ave* be this result.

End of algorithm.

Algorithm version 2:

sum ← *count* ← *0* *{ sum = sum of numbers;*

count = how many numbers are entered? }

min ← *?* *{ min to hold the smallest value eventually }*

max ← *?* *{ max to hold the largest value eventually }*

for each num entered,

increment count

sum ← *sum + num*

if num < min then min ← *num*

if num > max then max ← *num*

ave ← *sum/count*

Note the use of indentation and symbols in the second version. What should *min* and *max* be initialized with? Algorithms may also be represented by diagrams. One popular diagrammatic method is the flowchart, which consists of terminator boxes, process boxes, and decision boxes, with flows of logic indicated by arrows.

The pseudo-code and flowchart in the previous section illustrate the three types of control structures. They are: Sequence; Branching; Loop.

These three control structures are sufficient for all purposes. ***The sequence*** is exemplified by sequence of statements place one after the other – the one above or before another gets executed first. In flowcharts, sequence of statements is usually contained in the rectangular process box.

The ***branch*** refers to a binary decision based on some condition. If the condition is true, one of the two branches is explored; if the condition is false, the other alternative is taken. This is usually represented by the 'if-then' construct in pseudo-codes and programs. In flowcharts, this is represented by the diamond-shaped decision box.

The ***loop*** allows a statement or a sequence of statements to be repeatedly executed based on some loop condition. It is represented by the 'while' and 'for' constructs in most programming languages, for unbounded loops and bounded loops respectively. In the flowcharts, a back arrow hints the presence of a loop. A trip around the loop is known as iteration. You must ensure that the condition for the termination of the looping must be satisfied after some finite number of iterations, otherwise it ends up as an infinite loop, a common mistake made by inexperienced programmers.

Combining the use of these control structures, for example, a loop within a loop (nested loops) is not uncommon. Complex algorithms may have more complicated logic structure and deep level of nesting, in which case it is best to demarcate parts of the algorithm as separate smaller modules. Beginners must train themselves to be proficient in using and combining control structures appropriately, and go through the trouble of tracing through the algorithm before they convert it into code.

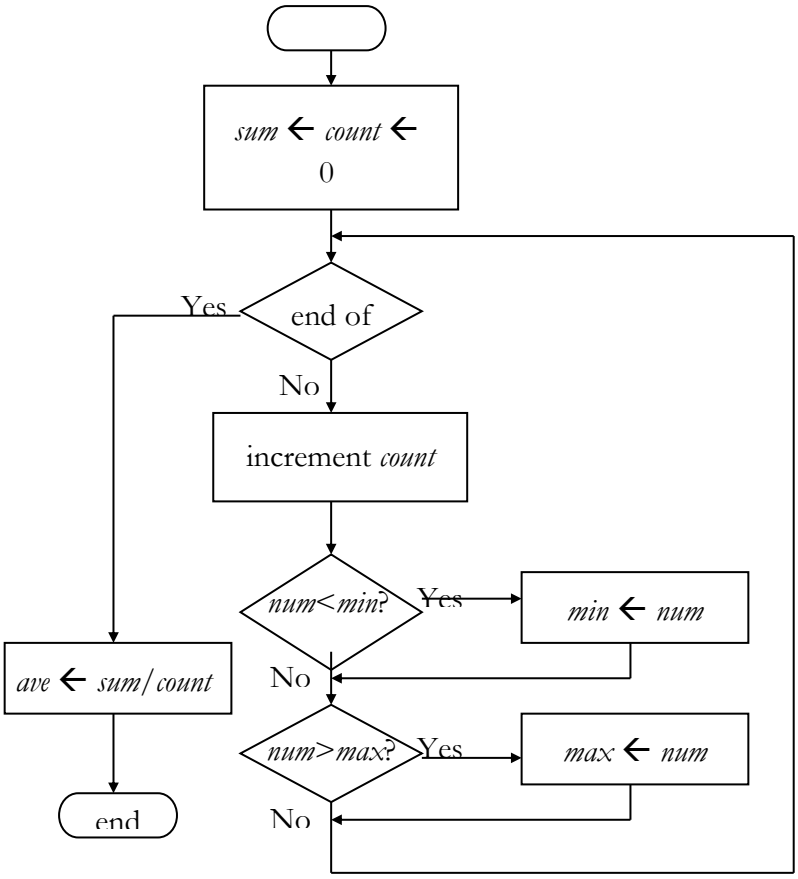


Fig. 3.1 Flowcharts

Self-Assessment Exercise(s) 3

1. What is pseudo-code?
2. What do you mean sequence structure?

4.0 Conclusion

1. This unit took you to algorithm and characteristics of algorithms. There are four essential properties of an algorithm.
2. The pseudo-code and flowchart in the previous section illustrate the three types of control structures: sequence; branching; loop.

5.0 Summary

1. Development of program may be have any ways- development of algorithm and it's realisation on programming languages. It's called – structured programming. Program is definitions of data and some sequence of function above this data.
2. The characteristics of algorithms are presented in this chapter, so are the two forms of representation for algorithms, namely, pseudo-codes and flowcharts.
3. The three control structures: sequence, branch, and loop, which provide the flow of control in an algorithm, are introduced.
4. Search and sorting algorithms.

6.0 Tutor-Marked Assignment

1. What is an algorithm?
2. Algorithm + Data Structures=....
3. State the steps in problem solving process?
4. State the process of analyzing a problem?
5. What do you mean by implement the algorithm?
6. What is maintenance?
7. What is pseudo-code?
8. What do you mean by sequence structure?

7.0 References/Further Reading

1. Analysis and design of algorithms. : A.A. Puntambekar. 2008
2. Algorithms. Robert Sedgewick, Kevin Wayne. 2011
3. Computer and Machine Vision: Theory, Algorithms, Practicalities . E.R. Davies. 2012
4. Data structures and algorithms using C++. Akepogu Anand Rao. 2010
5. Foundation of algorithms. Richard E. Neopolitan, Kumarss Naimipour. 2010

Unit 3

Search and Sorting Algorithms

Contents

1.0 Introduction

This unit introduces search and sorting algorithms. Also, you will learn fundamental programming construction: sequential, sorting by insert, binary search. Also, you are going to learn how to make program using sorting and search algorithms.

2.0 Learning Outcomes

At the end of this unit, you should be able to understand:

1. What is sorting algorithm?
2. What is search algorithm?
3. Types of sorting and search algorithms.

3.0 Learning Contents

3.1 Sequential search

The person constantly should face problems of search of the demanded information. Algorithms of search, thus, are the basic algorithms of data processing at the decision both system, and applied problems.

As typical examples with that or other directory, a telephone directory, a card file work can serve in library etc.

Let the set from n elements is set in the form of a file a [n].

It is possible to carry following operations to the simplest problems of search.

1. To find at least one element equal to preset value X as a result. It is necessary to receive i - an index (serial number) of an element of a file for which equality a [i] =X is carried out.
2. To find all elements equal to a preset value X. As a result, it is necessary to receive quantity of such elements and their serial numbers.

Sometimes search will be organized not on coincidence to value of an element X, and on performance of some set conditions. For example, search for an interval of values i.e. when the elements are found, satisfying to inequalities $X1 \le a[i] \le X2$ where values for X1 and X2 are set will be organized.

Let's notice that search operations can be carried out with the data which are in different conditions. These conditions are defined by degree of orderliness of the data.

For a file of not ordered data the unique way for search of the set element consists in comparison of each element of a file with set and at coincidence of some element of a file with set its position in a file is fixed.

Thus, if what or the additional information on the searched data is not set, the following approach - simple consecutive viewing of data file is obvious. Such method name linear (sequential) search.

Condition of the termination of search is one of following two situations:

1. The demanded element is found, т. e. condition a [i] = X it is carried out.
2. All file is seen, but coincidence it was revealed not, i.e. the demanded element is absent.

One of possible algorithms of the decision of the given problem can be:

1. We enter a variable flag logic type which to matter "true" (1) if the demanded element is found in a file also "lie" (0) otherwise.
2. ix - a variable for number of a required element.

flag = "lie";

ix=0;

The cycle beginning (for i from 1 to N)

if a [i] == X

then

flag = "true";

ix=i; all

The cycle end if flag == 1 then:

Conclusion: «the element is found, it number=ix»;

Otherwise: a conclusion: «the element is not found».

If after algorithm performance flag = "true" an element have found, and if in a file some such elements in a variable ix it will be stored.

If the variable flag has not changed the value, means an element is not present, and the variable remained in zero.

In practice consecutive search demands the greatest time expenses. Therefore speed of work of the programs realizing such search, is in direct dependence on the sizes of data sets.

The considered algorithm demands viewing of all data file, even in the event that the required element is in a file on the first place.

For reduction of an operating time of the program it is possible to stop its work right after how the element will be found. In this case all file will be seen only when a required element or last or it in general is not present.

And in this case having received the following algorithm:

flag = "lie";

ix=0; i=1;

The cycle beginning: while (i≤N and a [i]! =X)

if a [i] == X

then

flag = "true";

ix=i; all

The cycle end

If flag == 1 then:

Conclusion: «the element is found, it number=ix»; otherwise: a conclusion: «the element is not found».

The cycle termination is guaranteed, as on its each step value of parameter cycle i increases, and it for final number of steps will reach numbers N or as soon as the required element a cycle ahead of schedule will be found will end.

From a cycle condition follows that if the element is found, it is the first of such elements.

In this algorithm, thus on each step it is required to increase an index and to calculate logic expression.

Self-Assessment Exercise(s) 1

2. How do we organize search in sequential algorithm?

3.2 Sorting by Inserts

Earlier the general statement of a problem of sorting has been formulated and methods of sorting are considered by a choice. Now we will consider algorithms of sorting by means of an insert.

Method 1. A linear (simple) insert

The given method is based on a consecutive insert of elements in the ordered working file. The linear insert is used more often then, dynamically make changes to a file which all elements are known also which all time should be ordered.

Algorithm of a linear insert the following. The first element of an initial file is located in the first position of a working file. The following element of an initial file is compared to the first. If this element is more, it is located in the second position of a working file. If this element is less, the first element moves on the second position, and the new element is located on the first. Further all elements chosen from an initial file are consistently compared to elements of a working file, since the first until there will be an element more than the added. Then this element and all elements of a working file subsequent to it are displaced on one position, releasing a place for a new element.

Thus, each new element a [i] we will insert on a suitable place into already ordered set a [1], a [2], ... a [i-1]. And this place is defined by consecutive comparisons of an element a [i] with the ordered elements a [1]..., a [i-1].

Such method of sorting also named sorting by simple inserts. He, as well as a choice demands an order (N2-N)/2 comparison operations.

Method 2. The aligned and binary inserts

The central element of this file often name a median which breaks a file on two branches - descending (left) and ascending (right).

The algorithm of the aligned insert can be formulated so.

In a position located in the middle of a working file, the first element (he and will be a median) will go mad. Descending and ascending branches have indexes which show on the nearest to the beginning and the end the taken positions. After loading of the first element in the central position both indexes coincide and show on it. The following element of an initial file is compared to a median. If the new element is less, it takes places on a descending branch, otherwise - on an ascending branch. Besides, the corresponding trailer index moves ahead on unit downwards (a descending branch) or on unit upwards (an ascending branch).

Each subsequent element of an initial file is compared in the beginning to a median, and then with elements on a corresponding branch until will take of the necessary position.

Method 3. A binary (binary) insert

Here for search of a place for an element insert a [i] the algorithm of binary (binary) search is used. It is compared, in the beginning with an element [i/2]. Then, if it is less compared to an element [i/4] and if it is more: - with [i/2] + [i/4 | etc. until for it there will be no place. All elements of a working file, since a position of an insert and more low, move on one position, releasing a place for i th element.

The quantity of operations of comparison will be order N*log2 (N).

Variant of algorithm of a binary insert

The cycle beginning (for i from 2 to N) R=i; L=1

The cycle beginning: while L <R; k = (L+R)/2; if a [k]> a [i] then

L=k+1; otherwise

R=k;

all

The cycle end k=R; x=a [i]; the cycle beginning (for m from i to k+1 a step-1)

a [m] = a [m-1]; the cycle end a [k] =x;

all

the cycle end

Cycle «while L <R» - a cycle of search of a place of an insert. It is based on a method of binary search.

The cycle «for m from i to k+1 a step-1» shifts elements for clearing of a place of an insert.

Example. Let there is a sequence of numbers {0, 1, 9, 2, 4, 3, 6, 5}. We will sort by decrease.

Initially working file is empty. We place in it 1. For simplicity we will write down all set, and an element which is inserted, it will be underlined. The existing file is separated by a vertical line. For an element insert in a proper place all elements, standing up for it, we shift to a position which was occupied with an element inserted on a given step.

1)1,|9,2,4,3 6,5,0

2)9,1,|2,4,3, 6,5,0

3)9,2,1,|4,3, 6,5,0

4)9,4,2,1,|3, 6,,0
5)9,4,3,2,1,|
6,5,0
6)9,6,4,3,2,1,|5,
0
7)9,6,5,4,3,2,1,|
0

Last step:

8)9,6,5,4,3,2,1,0|

Self-Assessment Exercise(s) 2

1. How many comparisons can we make in a linear insert?
2. How many comparisons can we make in a binary insert?
3. ..... of an initial file is compared in the beginning to a median, and then with elements on a corresponding branch until will take of the necessary position

3.3 Binary Search

Let's give an example from a life. There was a following situation. In airport police station the message that somebody tries to carry by a bomb in the plane has arrived. It is known that it while is in an airport building. A task in view: as soon as possible to define, who from passengers has a bomb. Most simple, but also the most enduring decision - it consecutive check of all passengers which are in a building.

In such situation it is better to arrive so. We place all passengers in two rooms. For example, the guard dog reacting to an explosive can specify, in what of rooms there is

a bomb. As a result of such action the quantity of "suspects" will decrease twice. With "suspects" we will arrive similarly: we will dissolve them on two rooms and again the quantity of "suspects" will be cut by half. We continue to arrive so, yet we will not find a bomb.

Such method of search is called as binary search or: binary search, logarithmic search, a halving method, a dichotomy.

In programming such search apply to a finding of an element X in the sorted file a [1. N].

The basic idea consists in the following.

Let the file is sorted in decreasing order:

1. the average element of a file a [m], where m = (n+1)/2 is defined;
2. it is compared it with X, and if X=a [m] search on it comes to an end;
3. if a [m] <X all elements from m to n reject;
4. if a [m]> X reject elements from 1 to m;
5. thus, each time is necessary for recalculating left (L) or right (R) borders, i.e. it is primary L=1, R=n (first step); on the second step or L=m Or R=m i.e. on the second step either left, or the right border will change the Value etc.;
6. search proceeds until the element will not be found: a [m] =X, or while left and the rights of border of search will not coincide: (L=R), i.e. object X in a file is absent.

Let K - quantity of operations of comparison which are necessary for an element finding in the ordered file by means of algorithm of binary search. The number K is defined from a following inequality: N≤2K, and K - the minimum number from all possible.

Usually in the mathematician for number definition to use function log. Then the number K will be calculated under the following formula ***K = [log2N] +1*** where square brackets designate the whole part of the number which are in brackets.

Examples

4. In a file a to find an element x=1

a = 9, 6, 5,4, 3,2, 1,0 (N=8)

flag = «lie»

L=1; R=8; m1 = 9 % 2 = 4 (% - operation of integer division)

m=4; a[4]=4; 4>1 L=5; R=8; flag = «lie» result: 3, 2, 1 ,0

5 6 7 8

m_2= 5% 2 = 2 m=6; a[6]=2; 2>1 flag = «lie»

Result:
L=7; R=8; m=7; a[7]=1; 1=1 1=1 flag = «true»

1. Now array a=9, 8, 7, 6, 5, 4, 3, 2, 0
   In a file a to find an element x=1

   x=1,

   flag = «lie»

   L=I; R=9; m=5; a[5]=5; flag = «lie»

   L=6; R=9; m=7; a[7]=3; flag = «lie»

   L=8; R=9; m=8; a[8]=2; flag = «lie»

   L=9; R=9; m=9; a[9]=0; flag = «lie»

   L=9; R=8; L>R; all. flag = «lie»

   L=8; R=9; m=8; a[8]=2; flag = «lie»

   L=9; R=9; m=9; a[9]=0; flag = «lie»

   L=9; R=8; L>R; all. flag = «lie»

   Element is upset in file.

   Next algorithm:

   flag = «lie»;

   L=1;

   R=N

   Begin cycle: while (L≤R) u (flag = =«lie»);

   m=(R+L)/2;

   if a[m] = = X]

   then

   flag = «true»;

   else a[m]> X 1;

   else

   R=m+1;

   all

   all

   end cycle if (flag= = true) then print: «element X found, number - m»,

   else print: «element X not found, number - m»,

It is possible to simplify some algorithm if, as well as in case of linear search, to add an element X in an initial file.

L=1;

R=N+1

A[N+1]=X

Begin cycle: while (L≤R);

m=(R+L)/2;

if a[m] = = X]

then

flag = «true»;

else

if a[m]> X

then

L=m+1; else

R=m+1;

end cycle

After performance of a cycle number of the element equal. X is stored in variable R. If R=N-1, means coincidence is not present.

Self-Assessment Exercise(s) 3

1. Which mathematician function is being use in binary search?
2. Do you know any of the basic sorting techniques?

4.0 Conclusion

This unit took you through search and sorting algorithms. Also, you have learnt fundamental programming construction: sequential, sorting by insert, binary search. Also, you are going to learn how to make program using sorting and search algorithms.

5.0 Summary

1. In practice consecutive search demands the greatest time expenses.
2. Therefore, speed of work of the programs realizing such search, is in direct dependence on the sizes of data sets.
3. The program in language C/C++ use search and sorting algorithms in this unit.

6.0 Tutor-Marked Assignment

1. How do we organize search in a sequential algorithm?
2. How many comparisons can you make in a linear insert?
3. ..... of an initial file is compared in the beginning to a median, and then with elements on a corresponding branch until will take of the necessary position.
4. How many comparisons can you make in a binary insert?

5. Which mathematician function is being use in binary search?
6. What are the basic sorting techniques?

7.0 References/Further Reading

1. Analysis and design of algorithms. : A.A. Puntambekar, 2008
2. Algorithms. Robert Sedgewick, Kevin Wayne, 2011
3. Computer and Machine Vision: Theory, Algorithms, Practicalities . E.R. Davies, 2012
4. Data structures and algorithms using C++. Akepogu Anand Rao, 2010
5. Foundation of algorithms. Richard E. Neopolitan, Kumarss Naimipour, 2010

Module 4

Fundamental Data Structures

Unit 1. Primitive data types

Unit 2. Strings

Unit 3. Help allocation

Unit 1

Primitive Data Types

Contents

1.0 Introduction

This unit introduces the principles of Encapsulation and fundamental data structures of programming language C/C++. Also, you will learn using fundamental data construction. Also, you are going to learn how to using array, application of pointers, structures, stack, files.

You will know more about fundamental data structures of object oriented programming languages C/C++.

2.0 Learning Outcomes

At the end of this unit, you should be able to:

1. State the principles of encapsulation.
2. State how to declare arrays?
3. Understand the application of pointers?

3.0 Learning Contents

3.1 Principles of Encapsulation

All structural programming languages provide the simple scheme of interaction of an executed code and data. Data are grouped with use of accessible possibilities of the programming language, as in about files, structures, the associations, separate variables of various types (Figure 4.1)

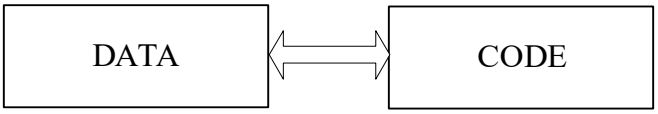


Fig. 4.1 Simple scheme of interaction of an executed code and data

Functions are applied to processing and interaction of a code with data (in). For interaction with various types and structures of data it is necessary to use various functions, that in turn complicates a writing of the big projects in which are used set of structures of data. In object-oriented technology of programming key concept of interaction of a code and data is the object. As object we will understand a certain structure which includes data and a code which co-operates with these data. Structurally it is possible to present it as follows (Figure 4.2):

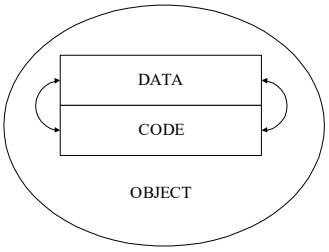


Fig. 4.2 Concept of interaction of a code and data is the object

```
class <a class name>
{
  [The description of given]
  [The code description]
};
```

Unlike structures and mixes classes contain not only data, but also a code operating these data. After definition of similar structure it is possible to define concrete copies of classes which represent objects:

<Name of class> <object 1>, <object 2>..., <object n>;

First concept object-oriented programming is the packing of data differently named ***encapsulation*** is a mechanism of the programming language which allows to unite data and a code co-operating with these data. The described class represents new type of data. The copy (element) of new type represents object. The part of data or a code can be protected from influence and use out of the description of object (class).

Encapsulation is such mechanism of language which unites data and code manipulating these data, and also protects both from external intervention or misuse.

Classes can be considered from the several points of view. The class is convenient for considering as abstract type of the data concerning operation of the announcement. Thus the program code realizing types of the data, is broken into two parts.

The interface part accessible to the user represents set of operations which define ***behavior*** of abstraction.

The second part - a part of ***realization***, is visible or accessible to the programmer, who is carrying out realization of this type of the data. Regarding realization values of variables which are necessary for maintenance of inwardness of object are visible. As example we will consider ATD "stack". The user sees the description of admissible operations: push, pop, top etc.

So, that concrete details are hidden (encapsulation) inside more abstract object. The term "object" is used for designation of representative of class. The term "an object variable" or "attribute" will be applied to internal variable of object. Each object has own set of attributes. Usually they do not change clients directly, but only by means of methods of classes specially intended for this purpose.

Object encapsulation in itself a status and behavior. The ***status*** is described by attributes of object, and the behavior is characterized by methods. Outside clients can learn only about behavior of objects. From within the full information on how methods provide necessary behavior is accessible, change a status and co-operate other objects.

Self-Assessment Exercise(s) 1

1. .....is a mechanism of language which unites data and code manipulating these data, and also protects both from external intervention or misuse.
2. Object encapsulation has...
3. .....is described by attributes of object, and the behavior is characterized by methods?

3.2 Concept of array

Array - the numbered sequence of data of one type which are stored in continuous area of memory one after another. Members of sequence of data are called as elements of array.

Access to elements of array is made by instructions of name array and element number.

Numbering of elements can be carried out by one or several sequences of integers - index sequences.

If numbering is carried out by one sequence say, that the array is one-dimensional, otherwise - multidimensional.

Numbering of elements of array always begins with 0, and number of each following member is more than number previous on 1.

In mathematics for comfort of record of different operations often use index variables: vectors, matrices, tensors. So, a vector appears the set of numbers (c1 c2...cn), called his component, thus every component has the number which it is accepted to designate as an ***index***. ***Matrix*** is the table of numbers (a_{ij}, i=1,.., m; j=1,.., n), i is a line number, j is a number of column. Operations above matrices and vectors usually have a short record, which designates certain, at times difficult actions above their index components. For example, works of two vectors write:

$$\vec{c} \cdot \vec{b} = \sum_{i=1}^{n} c_i \, b_i$$

Work of matrix on a vector

$$\vec{b} = A \cdot \vec{c}, \quad b_i = \sum_{j=1}^{n} a_{ij} \cdot c_j$$

Work of two matrices

$$D = A \cdot G, \quad d_{ij} = \sum_{k=1}^{n} a_{ik} \cdot g_{kj}$$

Introduction of index variables in programming languages also allows considerably to facilitate realization of many difficult algorithms, related to treatment of arrays of the same type data.

In the language C/C++ for this purpose there is a difficult type of variables - array, being a well-organized eventual aggregate of elements of one type. The number of array cells is named measuring. Every array cell is determined by an array identifier and sequence number - by an index. An index is an integer on which access is produced to the array cell. Indexes can be a few. In this case an array is named multidimensional, and an amount of indexes of one array cell is his dimension.

Indexes at unidimensional arrays in the language C/C++ begin from 0, and in the program an unidimensional array is declared as follows:

<type> < ID_of array>[size]={list of initial values};

where: a type is a base type of elements (whole, material, symbol); a size is an amount

The size of array can be set by a constant or constant expression. It is impossible to set the array of variable size. For this purpose there is a separate mechanism - dynamic allocation of memory.

Example of announcement of array of integer type: int a[5];

In the array of "a" there is the first element: and [0], second - and [1], fifth - and [4].

An address to the array cell in the program in language of Si is carried out in traditional for many other languages style - record of operation of appeal on an index, for example:

a[0]=1;

a[i]++;

a[3]=a[i]+a[i+1];

Example of announcement of array of integer type with initialising of values:

int a[5]={2, 4, 6, 8, 10};

If in a group {.} the list of values is shorter, then to the remaining elements of =0.

The mechanism of check of ranges of change of indexes of arrays absents with the purpose of increase of the speed of program. At a necessity such mechanism must be programed obviously.

b[0][0], b[0][1], b[1][0], b[1][1], b[2][0], b[2][1].

A next example illustrates determination of array of integer type, consisting of three lines and four columns, with a simultaneous appropriation to his elements (initialising) of initial values: int a[3][4] = {{1,2,0,0},{9, - 2,4,1},{- 7,0,0,0}};

If in some group { } the list of values is shorter, then remaining elements is appropriated 0. As marked already, except for unidimensional arrays work is possible with multidimensional arrays. Announcement of multidimensional array:

<type> < ID >[size1][size2]….[sizeN]={{list of initial values},

{list of initial values},.};

The last index of array cells changes most quickly, as multidimensional arrays take place in memory of computer in the sequence of columns.

For example, two-dimensional array of b cells [3][2] take place in memory of computer in the next order:

sizeof(int) → size of memory – 2 byte,

int b[5];

sizeof(b) → size of memory – 10 byte.

char s[256];

*int kol = sizeof(s)/sizeof(*s);* // number of elements of massive s

Example: Set array from 10 integer number. Get this summa of 10 numbers

```
#include <iostream.h>
int main()
{
   const int n=4;
   int a[n];
   int sum=0;
   cout << "Enter elements..." << endl;
      {
         cout << endl << "a[" << i << "]: ";
         cin >> a[i];
            }
      for (int i=0; i<n; i++) sum=sum+a[i];
cout<< "summa =" << sum<<endl;
return 0;
}
```

Self-Assessment Exercise(s) 2

1. Indexes at unidimensional arrays begin from…?
2. Set array from 10 double numbers. Develop the program which finds sum of positive elements and product of negative elements.
3. What is a two-dimension array?

3.3 Application of pointers

An array identifier is an address of memory, which he is located since, i.e. address of his first element. Work with arrays is closely constrained with the use of pointers.

Let the array of a from 5 integer elements and pointer of q is declared on integer variables: *int a[5], *q;*

ID of array of an is a constant pointer to his beginning (Figure 4.3).

a

a[0]	a[1]	a[2]	a[3]	a[4]

4000

Figure 4.3 Array of a from 5 integer elements and pointer of q

A symbolic image over of main memory, distinguished by a compiler for the declared integer array and is here brought [5]. Pointer and contains the address of his beginning in-memory, i.e. "symbolic address"=4000 (and=4000).

If an operation is executed: q=and; it is appropriating of constant to the variable, i.e. q=4000 (analogue: q=&a[0]), then taking into account address arithmetic of expression and[i] and *(q+i) result in identical results - appeal to i to the th array cell.

Identifiers and and q are pointers, obviously, that expressions and[i] and *(and+i) equivalent. It is necessary from here, that the operation of address to the array cell on an index is applicable and at his naming a variable-pointer. Thus, for any pointers it is possible to use two equivalent forms of expressions for access to the array cells: q[i] and *(q+i). The first form is more comfortable for read of text, second - more effective on the fast-acting of the program.

1) address massive in memory:

&a[0] ↔ &(*a) ↔ a

2) ask first element of massive:

*a ↔ a[0]

Number elements of massive:

```
type x[100];  // number will constant
    ...
int n = sizeof(x) / sizeof(*x);
```

Pointers, as well as variables of any other type, can unite in arrays.

Announcement of array of pointers to the integers looks like: int *a[10], y;

Now each of array cells it is possible to appropriate integer variable of y address, for example: a[1]=&y;

In the language C/C++ it is possible to describe the variable of type "pointer to the pointer". It is a main memory cell, in which will be kept pointer address to what or variable. A sign of such type of data is a reiteration of symbol "*" before a variable identifier. The amount of symbols "*" determines a nesting of pointers level in each other. At announcement of pointers to the pointers their simultaneous initializing is possible. For example:

int a=5;

*int *p1=&a;*

*int **pp1=&p1;*

*int ***ppp1=&pp1;*

Now we will appropriate to the integer variable and new value, for example 10. The identical appropriating will be produced by next operations: a=10;

**p1=10;* ***pp1=10;* ****ppp1=10;*

For access to the area ОП, taken under a variable and it is possible to use indexes. Next analogues are just, if we work with multidimensional arrays:

**p1 <-> p1[0]* ***pp1 <-> pp1[0][0]* ****ppp1 <-> ppp1[0][0][0]*

We will mark, that a two-dimensional array identifier is a pointer to the array of pointers (variable of type pointer to the pointer: int **m;), therefore expression and[i][j] equivalently to expression *(*(m+i)+j).

For example, array of m[3][4]; a compiler examines as an array of four pointers, each of which specifies on beginning of array with values measuring for three elements each (Figure 4.4).

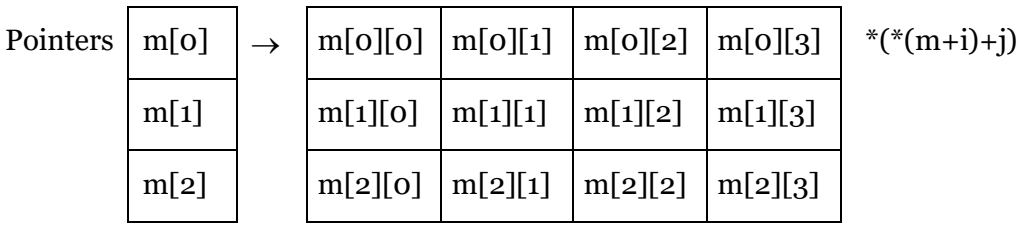


Fig. 4.4 Array of m[3][4];

Example: Create next two dimension array: 1222

0122

0012

0001

int main() // working with two-dimensional arrays

{

```
  const int n=4;
  int a[n][n] = {0};            // set all elements of array 0

  for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
    {
    if (i == j) a[i][j]=1;        // elements of main diagonal is 1
    else if (i < j) a[i][j]=2;   // elements of top triangle is 2
    }
  for (int i=0; i<n; i++)
    {
    cout << endl;
    for (int j=0; j<n; j++) cout << a[i][j];
     }
return 0;
}
```

Self-Assessment Exercise(s) 3

1. Variables of any other type are called...?
2. What is an array of pointers?
3. Describe the variable of type "pointer to the pointer"?

4.0 Conclusion

This unit introduces the principles of Encapsulation and fundamental data structures of programming language C/C++. Also, you have learnt using fundamental data construction. Also, learn how to using array, application of pointers.

You know more about fundamental data structures of object oriented programming languages C/C++.

5.0 Summary

1. This unit introduced principles of Encapsulation.
2. Presentation of one and two dimension arrays.
3. Application of pointers in the program.

6.0 Tutor-Marked Assignment

1. Object encapsulation has…
2. …..is a mechanism of language which unites data and code manipulating these data, and also protects both from external intervention or misuse.
3. Object encapsulation has…
4. ……..is described by attributes of object, and the behavior is characterized by methods? Indexes at uni-dimensional arrays begin from…?
5. The number of array cells is named…?
6. What is a two-dimension array?
7. Variables of any other type are called…?
8. What is an array of pointers?
9. Describe the variable of type "pointer to the pointer"?

7.0 References/Further Reading

1. C++ for Programmers. Kalnay - Smashwords , 2012
2. C++ Reference Guide. Danny Kalev - Informit , 2008
3. Data Structures and Algorithm Analysis in C++. Clifford A. Shaffer - Dover Publications, 2012
4. C++ Annotations . Frank B. Brokken - University of Groningen , 2008
5. Mastering C++. K. R. Venugopal.2008
6. Object Oriented Programming with C++, Balagurusamy.2006
7. A Complete Guide to Programming in C++. Ulla Kirch-Prinz, Peter Prinz, 2009
8. Object-Oriented Programming: Using C++ for Engineering and Technology. Goran Svenk. 2008

Unit 2

Strings

Contents

1.0 Introduction

This unit introduces you to strings of programming language C/C++. Also, you will learn function of string. Also, you are going to learn how to make manipulation with string. You will know more about fundamental data structures of object oriented programming languages C/C++.

2.0 Learning Outcomes

At the end of this unit, you should be able to understand:

1. Manipulation with string?
2. Function of string.
3. How to work with string?

3.0 Learning Contents

3.1 Manipulations with string

In language C++, strings are not a built-in data type, but rather a Standard Library facility. Thus, whenever we want to use strings or string manipulation tools, we must provide the appropriate **#include** directive, as shown below:

```
#include <string>
using namespace std;    // Or using std::string;
```

We now use string in a similar way as built-in data types, as shown in the example below, declaring a variable name:

```
string name;
```

Unlike built-in data types (int, double, etc.), when we declare a string variable without initialization (as in the example above), we *do have* the guarantee that the variable will be initialized to an empty string — a string containing zero characters.

C++ strings allow you to directly initialize, assign, compare, and reassign with the intuitive operators, as well as printing and reading (e.g., from the user), as shown in the example below:

```
string name;
cout << "Enter your name: " << flush;
cin >> name;
   // read string until the next separator
   // (space, newline, tab)
   // Or, alternatively:
getline (cin, name);
   // read a whole line into the string name
if (name == "")
{
```

```
    cout << "You entered an empty string, "
        << "assigning default\n";
    name = "John";
  }
  else
  {
    cout << "Thank you, " << name
        << "for running this simple program!"
        << endl;
  }
```

C++ strings also provide many string manipulation facilities. The simplest string manipulation that we commonly use is concatenation, or addition of strings. In C++, we can use the + operator to concatenate (or “add”) two strings, as shown below:

```
string result;
string s1 = "hello ";
string s2 = "world";
result = s1 + s2;
  // result now contains "hello world"
```

Notice that both **s1** and **s2** remain unchanged! The operation reads the values and produces a result corresponding to the concatenated strings, but doesn't modify any of the two original strings.

The += operator can also be used. In that case, one string is appended to another one:

```
string result;
string s1 = "hello";
   // without the extra space at the end
string s2 = "world";
result = s1;
result += ' ';
   // append a space at the end
result += s2;
```

After execution of the above fragment, result contains the string "hello world".

You can also use two or more + operators to concatenate several (more than 2) strings. The example below shows how to create a string that contains the full name from first name and last name (e.g.,firstname = "John", lastname = "Smith", fullname = "Smith, John").

```
string firstname, lastname, fullname;
cout << "First name: ";
getline (cin, firstname);
cout << "Last name: ";
getline (cin, lastname);
```

```
fullname = lastname + ", " + firstname;
cout << "Fullname: " << fullname << endl;
```

Of course, we didn't need to do that; we could have printed it with several **<<** operators to concatenate to the output. The example intends to illustrate the use of strings concatenation in situations where you need to store the result, as opposed to simply print it.

Now, let's review this example to have the full name in format **"**SMITH, John**"**. Since we can only convert characters to upper case, and not strings, we have to handle the string one character at a time. To do that, we use the square brackets, as if we were dealing with an array of characters, or a vector of characters.

For example, we could convert the first character of a string to upper case with the following code:

```
str[0] = toupper (str[0]);
```

The function ***toupper*** is a Standard Library facility related to character processing; this means that when using it, we have to include the **<*cctype*>** library header:

```
#include <cctype>
```

If we want to change all of them, we would need to know the length of the string. To this end, strings have a method ***length***, that tells us the length of the string (how many characters the string has).

Thus, we could use that method to control a loop that allows us to convert all the characters to upper case:

```
 for (string::size_type i = 0; i < str.length(); i++)
{
   str[i] = toupper (str[i]);
}
```

Notice that the subscripts for the individual characters of a string start at zero, and go from **0** to **length−1**.

Notice also the data type for the subscript, ***string::size_type***; it is recommended that you always use this data type, provided by the **string** class, and adapted to the particular platform. All string facilities use this data type to represent positions and lengths when dealing with strings.

The example of the full name is slightly different from the one shown above, since we only want to change the first portion, corresponding to the last name, and we don't want to change the string that holds the last name — only the portion of the full name corresponding to the last name. Thus, we could do the following:

```
fullname = lastname + ", " + firstname;
for (string::size_type i = 0; i < lastname.length(); i++)
{
```

```
    fullname[i] = toupper (fullname[i]);
}
```

Self-Assessment Exercise(s) 1

1. How many characters do a string not initialized contains?
2. What the result of fragment this program?

```
string result;
string s1 = "programming language ";
string s2 = "C++";
result = s1 + s2;
```

3. Which method helps to know how many characters the string has?

3.2 Search Facilities

Another useful tool when working with strings is the ***find*** method. This can be used to find the position of a character in a string, or the position of a substring. For example, we could find the position of the first space in a string as follows:

```
position = str.find (' ');
```

If the string does not contain any space characters, the result of the find method will be the value ***string::npos***. The example below illustrates the use of ***string::npos*** combined with the ***find*** method:

```
if (str.find (' ') != string::npos)
{
   cout  << "Contains at least one space!" << endl;
}
else
{
   cout  << "Does not contain any spaces!" << endl;
}
```

The ***find*** methods returns the position of the *first* occurrence of the given character (or ***string::npos***). We also have the related ***rfind*** method — the **r** stands for *reverse* search; in other words, **rfind** returns the position of the last occurrence of the given character, or ***string::npos***. You could also look at it as the first occurrence while starting the search at the end of the string and moving backwards.
The ***find*** and ***rfind*** methods can also be used to find a substring; the following fragment of code can be used to determine if the word "the" is contained in a given string:

```
string text;
getline (cin, text);

if (text.find ("the") != string::npos)
```

```
{
   // ...
```

For both cases (searching for a single character or searching for a substring), you can specify a starting position for the search; in that case, the ***find*** method will tell the position of the first occurrence of the search string or character after the position indicated; the ***rfind*** method will return the position of the last occurrence before the position indicated — you could look at it as the first occurrence while moving backwards starting at the specified position. The requirement for this optional parameter is that it must indicate a valid position within the string, which means that the value must be between 0.

The following fragment of code shows how to test if a string contains at least two spaces. It performs one search for a space, and then it does a second search, starting at the position where the first one was found:

```
string text;
getline (cin, text);
 string::size_type position = text.find (' ');
if (position != string::npos)
{
   if  (text.find (' ', position+1) != string::npos)
   {
      cout << "Contains at least two spaces!" << endl;
   }
   else
   {
      cout << "Contains less than two spaces!" << endl;
   }
}
else
{
   cout << "Contains no spaces!" << endl;
}
```

There are several other string facilities that are related to ***find***. Two of them are ***find_first_of***, and ***find_first_not_of***. Instead of finding the first occurrence of an exact string (as ***find*** does), ***find_first_of*** finds the first occurrence of any of the characters included in a specified string, and ***find_first_not_of*** finds the first occurrence of a character that is not any of the characters included in the specified string. An example of use is shown below:

```
string text;
getline (cin, text);
if (text.find_first_of ("aeiouAEIOU") == string::npos)
{
   cout << "The text entered does not contain vowels!"
```

```
        << endl;
    }
```

In this case the expression text.find_first_of ("aeiouAEIOU") returns the position of the first occurrence of any of the characters included in the string passed as parameter — a string containing all the vowels. ***find_first_of***, like ***find***, returns the position of the found character, *or* ***string::npos*** if no character matching the specified requirement was found. In this example, if ***find_first_of*** returns ***string::npos***, that means that the string did not contain any vowel (lowercase or capital).

find_first_not_of works in a similar way, except that it finds the first character that is not one of the characters specified in the string, as shown in the example shown below:

```
string card_number;
cout << "Enter Credit Card Number: ";
getline (cin, card_number);
 if (card_number.find_first_not_of ("1234567890– ") != string::npos)
{
   cout << "The card number entered contains invalid characters"
       << endl;
}
```

Like ***find***, both ***find_first_of*** and ***find_first_not_of*** accept also an extra parameter indicating the position where to start the search (naturally, if the parameter is omitted, the search starts at position **0**).

Self-Assessment Exercise(s) 2

1. Method which found position of a character in a string is ...
2. Function returns the position of the last occurrence of the given character is ...

3.3 Strings function and processing

You can check if a string is empty with the ***empty()*** function. There is also a function ***clear()*** which empties a string,

This demos these string functions ***find()***, ***insert()***, ***append()*** and ***replace()*** plus ***swap()*** for swapping this string with another, and ***substr()*** for getting part of a string.

substr() works with 0, 1 or 2 parameters as both parameters have defaults. The equivalent of the Basic string functions ***Left()***, ***Mid()*** and ***Right()*** are

Left(string,len) - substr(0,len)

Mid(string,startpos,len) - substr(startpos,len)

Right(string,len) - substr(size()-startpos+1, len)

The ***max_size()*** function returns the longest string possible, just less than the static constvalue ***string::npos*** that is returned if a call to ***find()*** fails. ***npos*** is the unsigned equivalent of -1. This is a magic number that's always returned for failed searches. It will always be bigger than ***max_size()***.

These are other variations of the find function.

- ***find_first_of()*** - Find first char in string (public member function)
- ***finc_first_not_of()*** - Find first char not in the string
- ***find_last_of()*** - Find first char in string from the end
- ***find_last_not_of()*** - Find first character not in string from the end

Each of these has overloaded versions for searching for string, char and char *.

The design means that when the constructor has finished, the file will have been processed and the list of url will be complete. There are two main processes used.

1. Read in the file into a string.
2. Process the string into a list. [/uol]

This uses the ***ifstream*** class to open the file. It then calls the ***seekg()*** function to move a file pointer to end of the file, calls ***tellg()*** to report its position i.e. indicate how big the file is. This is passed into **reserve ()** so that the string starts as big as it needs to be. A simple loop to read the file contents into the text string line by line. If you are searching a file, line breaks can be a real pain to deal with. Just in case any slipped through, the ***removecrlf()*** method converts them to spaces.

The method ***extractlinks()*** does the bulk of the work. It finds the start and end of a link through calls to ***find()***, then calls the ***push_back()*** method. This adds an element on the end of the list. This loop uses the variable **pos** to limit the search. When ***extractlinks()*** finishes, the links list is complete.

The last part of the ***textfile*** constructor calls the ***sort()*** function. This is an STL algorithm which works on the list and sorts the words alphabetically. It just requires two iterations ***links.begin()*** and ***links.end()***.

Self-Assessment Exercise(s) 3

1. Which function show that a string is empty?
2. Which function return the longest string?

4.0 Conclusion

This unit introduces the function of string programming language C/C++. Also, you have learnt how to use functions and methods of strings. Also, you have learnt functions longest string and empty string.

You know more about strings and string process of object oriented programming languages C/C++.

5.0 Summary

1. This unit introduced strings and processing of strings.
2. Functions and methods of strings.
3. Simple functions longest string, empty string.

6.0 Tutor-Marked Assignment

1. How many characters do a string not initialized contains?
2. What the result of fragment this program?

```
string result;
string s1 = "programming language ";
string s2 = "C++";
result = s1 + s2;
```

3. Which method helps to know how many characters the string has?
4. Method which found position of a character in a string is ...
5. Function returns the position of the last occurrence of the given character is ...
6. Which function show that a string is empty?
7. Which function returns the longest string?

7.0 References/Further Reading

1. Object Oriented Programming with C++, Balagurusamy.2006
2. A Complete Guide to Programming in C++. Ulla Kirch-Prinz, Peter Prinz, 2009
3. Object-Oriented Programming: Using C++ for Engineering and Technology. Goran Svenk. 2008

Unit 3

Help Allocation

Contents

1.0 Introduction

This unit introduces the new data structures of programming language C/C++ like structure, stack, files. Also, you will learn using fundamental data construction. Also, you are going to learn how to work runtime storage management.

You will know more about fundamental data structures of object oriented programming languages C/C++.

2.0 Learning Outcomes

At the end of this unit, you should be able to:

1. How to declare structures?
2. What is a stack?
3. How to work on runtime storage management?

3.0 Learning Contents

3.1 Structures and stack

Structure it is a component object of language C/C++, being an aggregate of the logically constrained data of different type, incorporated in a group under one identifier (ID). Given, included in this group name the fields.

Term a "structure" in the language C/C++ corresponds to two different on sense concepts:

i. structure is denotation of area of main memory, where the concrete values of data are situated. In future is a structural variable the fields of which are situated in contiguous areas ОП.
ii. structure is rules formings of structural variable, which are followed by a compiler during a selection to her places in ОП and organizations of access to her fields.

Determination of objects of type of structure is produced for two steps:

i. Declaration of structural type of data, not resulting in the selection of area to memory;
ii. Determination of structural variables with a selection for them to memory.

The structural type of data is set as a template, the general format of description of which following:

struct ID of structural type {

description of the fields

};

Attribute of "ID of structural type", i.e. her identifier is optional and can absent.

Description of the fields is produced in general a way: types and identifiers are specified.

Example of determination of structural type. It is necessary to create a template, describing information about a student: number of group, name, ID number. One of possible variants:

```
struct Stud_type {
        char Number[10];
    char Fio[40];
    double S_b;
    };
struct Stud_type {
```

Fields of one type at description it is possible to unite in one group:

```
        char Number[10], Fio[40];

    double S_b;
    };
```

Object type of Stud_type:

Number	Fio	S_b
10	40	8

Bytes

The structural type of data comfortably to apply for the batch-processing of the logically constrained objects. The parameters of such operations are an address and size of structure.

Examples of group operations:

capture and liberation of memory for an object;

record and reading of given, kept on external transmitters as physical and/or logical records with the known structure (during work with files).

As one of parameters of batch-processing of structural objects is size, it is not recommended to declare the field of structure a pointer to the object of variable dimension, as in this case many operations with mi is structural data will be not

correct, for example `struct Stud_type {  char *Number, *fio;`

```
                                    double S_b;
                                    };
```

in this case, entering the lines of Number and fio of different length, the sizes of objects will be also different. It is now necessary to create the necessary amount of variables with the brought structure over and doing it is possible two methods.

Method 1. In any place of the program for declaration of structural variables, arrays, functions et cetera, the structural type declared in a template is used, for example: struct Stud_type student;

struct Stud_type student; // structural variable;

Stud_type Stud[100]; //array of structures

*Stud_type *p_stud;* // pointer to the structure

Stud_type Fun(Stud_type);* //prototype of function with a parameter structural *type*, returning a pointer to the object of structural type.

Method 2: in the template of structure between the closed figured bracket and symbol ";" specify the identifiers of structural data through commas.

For our example, using, it is possible to write:

```
struct Stud_type {
        char Number[10], Fio[40];

    double S_b;
    } student, Stud[100], *p_stud;
```

During declaration of structural variables their simultaneous initializing is possible

For example:

```
struct Stud_type
 {
        char Number[10], Name[40];

    double S_b;
    } student = {"123456", "Odoko Dan.", 6.53 };
```

or:

```
        Stud_Type stud1 = {"123456", "Odoko Dan." };
```

Example: Structure of students contain surname, year of born and three exam. Calculate average of exams each students:

```
#include <iostream.h>
# include <iomanip.h>
# include <fstream.h>
const int n=2;
```

```
typedef struct STUD{
        char surn[20];
        int year_born;
        int exam[3];
        double aver;};
int main()
{   STUD st[n];
    int i;
     // input data
 for(i=0;i<n;i++){
  cout << "input data for " << i+1 << " - students"<<endl;
  cout<<"surname-->";
  cin >> st[i].surn;
  cout <<"year of born-->" ;
  cin>>st[i].year_born;
   cout <<"input result of exam"<<endl;
  for (int j=0;j<3;j++)cin >>st[i].exam[j];
                }
for(i=0;i<n;i++){
         double av=0;
       for(int j=0;j<3;j++) av+=st[i].exam[j];
        st[i].aver=av/3; cout << st[i].aver;
         }
   // output result as table
    cout << endl<<"table of exam result"<<endl;
    cout <<"-----------------------------------"<<endl;

    for(i=0;i<n;i++)
    cout <<setw(15)<<st[i].surn <<" ! "<<setprecision(2)<<st[i].aver<<endl;
      char c;
      cin >>c;
     //cin.getc();
```

```
}
```

Stack

Stack is a linear structure of data, in which addition and removal of elements probably only from one end (stack top)**.**

Stack = a bale, a heap, a pile

LIFO = Last In - First Out

«Who the last has entered, that the first left».

Operations with a stack:

1. Add an element on top (*Push* = push)*;*
2. Remove an element from top *(Pop* = take off with a sound*).*

Example:

Read words and print them in reverse order. Variation using stack and string.

```
#include <iostream>
#include <stack>
#include <string>
using namespace std;
int main()
 {
   stack<string> allwords;        // stack of all words
   string word;                   // input buffer for words.
                                  // read words/tokens from input stream
   while (cin >> word) {
      allwords.push(word);
   }
   cout << "Number of words = " << allwords.size() << endl;
                                   // write out all the words in reverse order.
   while (!allwords.empty()) {
     cout << allwords.top() << endl;
     allwords.pop();               // remove top element
   }
   return 0;
}                                  //end main
```

Self-Assessment Exercise(s) 1

1. What is structure?
2. How to declare structure?
3. What is a stack?

3.2 Files

A file is a set of data, placed on an external transmitter and looking like the process of treatment and sending as single unit. In the files of given, intended for the protracted storage.

Distinguish two types of files: text and binary. ***Text files*** are a sequence of ASCII of symbols and can be look and edited by means of any text editor.

Binary files are a sequence of data, the structure of which is determined programmatic.

The internal logical name, used in future for an address to him, is appropriated every file. The logical name (file identifier) is a pointer to the file, i.e. on a storage area, where all necessary information is about a file. The format of announcement of pointer to the file following:

*FILE *pointer*

FILE is an identifier of structural type, described in the standard library of stdio.h and containing next information: *type struct { short*

Before to begin to work with a file, i.e. to get possibility of reading or record of information in a file, he needs to be opened for access. A function is usually used for this purpose

FILE fopen(char * ID_of file, char *mode);*

Physical name, i.e. the file name and way to him is set by the first parameter - line, for example, "a: Mas_dat.dat" is a file with the name of Mas_dat.dat, being on a disk "d:\\work\\Sved.cpp" is a file with the name of Sved.cpp, being on winchester, in the catalogue of work.

At the successful opening the function of fopen() returns a pointer to the file (in future is a file pointer). NULL returns at an error. This situation arises up usually, when a way is unright specified to the opened file. For example, if in the display class of our university, to specify a way, forbidden for a record (usually, settled is d:\work\).

The second parameter is a line the file access mode is set in which : w - a file is opened for a record; if a file with the set name is not present, that he will be created; if such file exists, then before opening former information is destroyed;

r - a file is opened readonly; if such file is not present, then there is an error;

a - a file is opened for adding to his end of new information;

r+ - a file is opened for editing of data - both a record and reading of information is possible;

w+ - the same, what for r+;

a+ - the same, what for a, only a record can be executed in any place of file; accessible reading of file;

t - a file is opened in the character mode; the fields of r are used, w, a, r+, w+, a+;

b - a file is opened in the binary mode.

The character mode differs from binary that at opening a file as text steam of symbols "line feed", the "return of carriage" is substituted by single character: "line feed" for all functions of record of data in a file, and for all functions of conclusion symbol a "line feed" is now substituted by two symbols: "line feed", "return of carriage".

By default a file is opened in the character mode.

Example:

*FILE *f;* - a pointer is declared to the file of f;

f = fopen ("d:\\work\\Dat_sp.cpp", "w"); - a file is opened for a record with the logical name of f, having the physical name of Dat_sp.cpp, disk-based d, in the catalogue of work or more briefly:

*FILE *f = fopen ("d:\\work\\Dat_sp.cpp", "w");*

For closing of a few files a function, declared as follows, is entered:

void fcloseall(void);

If it is required to change the file access mode, then for this purpose at first it is necessary to close this file, and then again to open him, but with other access. For this purpose use a built-in function:

FILE freopen (char *ID_of file, char *mode, FILE *file pointer_);*

This function closes a file, declared a "file (as it does function of fopen) pointer"_at first, and then opens a file with the "file name"_and rights for access "mode".

Self-Assessment Exercise(s) 2

1. What is a file?
2. What is binary file?
3. Write example of file name.
4. How do you open file?

3.3 Run-time storage management

Run-time type identification make it possible to determine the type of an object E.g. given a pointer to a base class, determine the derived class of the pointed object. The type (class) must be known at compile time. *Introspection* makes general class

information available at run-time. The type (class) does not have to be known at compile time. This is very useful in component architectures and visual programming E.g. list the attributes of an object

Run-time type information may be new to you because it is not found in nonpolymorphic languages, such as C. In no polymorphic languages there is no need for run-time type information because the type of each object is known at compile time (i.e., when the program is written). However, in polymorphic languages such as C++, there can be situations in which the type of an object is unknown at compile time because the precise nature of that object is not determined until the program is executed. Since base-class pointers may be used to point to objects of the base class or any object derived from that base, it is not always possible to know in advance what type of object will be pointed to by a base pointer at any given moment in time. This determination must be made at run time, using run-time type identification. To obtain an object's type, use typeid. You must include the header ***<typeinfo>*** in order to use typeid.

typeid(object)

Here, object is the object whose type you will be obtaining. It may be of any type, including the built-in types and class types that you create. typeid returns a reference to an object of type ***type_info*** that describes the type of object. The ***type_info*** class defines the following public members:

bool operator==(const type_info &ob);

bool operator!=(const type_info &ob);

bool before(const type_info &ob);

*const char *name( );*

The overloaded == and != provide for the comparison of types. The before( ) function returns true if the invoking object is before the object used as a parameter in collation order. (This function is mostly for internal use only. Its return value has nothing to do with inheritance or class hierarchies.) The name( ) function returns a pointer to the name of the type.

Self-Assessment Exercise(s) 3

1. What shows the type of object at any moment in time?
2. What make it possible to determine the type of an object?
3. What must you include in the header of program to know run-time information?

4.0 Conclusion

This unit introduceg you to fundamental data structures of programming language C/C++. Also, you learn using fundamental data construction. Also, you have learnt the use of structures, stack, files.

You know more about fundamental data structures of object oriented programming languages C/C++.

5.0 Summary

1. This unit introduced new types of data language C/C++.
2. Receive run-time information and help allocation.

6.0 Tutor-Marked Assignment

1. What is structure?
2. How to declare structure?
3. What is a stack?
4. What is a file?
5. What is binary file?
6. Write example of file name.
7. How do you open file?
8. What shows the type of object at any moment in time?
9. What make it possible to determine the type of an object?
10. What must you include in the header of program to know run-time information?

7.0 References/Further Reading

1. Object Oriented Programming with C++, Balagurusamy.2006
2. A Complete Guide to Programming in C++. Ulla Kirch-Prinz, Peter Prinz, 2009
3. Object-Oriented Programming: Using C++ for Engineering and Technology. Goran Svenk. 2008

Module 5

Machine Organization

Unit 1. Machine levels organization

Unit 2. Assembly language programming

Unit 1

Machine Levels Organization

Contents

1.0 Introduction

This unit introduces machine organization. Also, you will learn first machine of Von Neuman and level machine organization.

You will know more about representation of data; bits, bytes, and words; binary representation of integers; representation of character data; representation of records and arrays

2.0 Learning Outcomes

At the end of this unit, you should be able to know:

1. How to represent data in computer?
2. How to construct machine language
3. Basic organization of the Von Neumann machine

3.0 Learning Contents

3.1 Machine level Representation of Data

A ***memory cell*** is characterized by two things:
1) An address

2) Content

The basic memory cell on many computers including the VAX holds one byte (8 bits). Such machines are called byte-addressable. Other machines have larger storage units. Components of the computer are connected by Buses. A bus in the simplest form is a set of wires that used to carry information in the form of electrical signals between CPU and memory and CPU and I/O.

There are 3 buses in the system

- Data Bus
- Address Bus
- Control (signal) Bus

VAX has a 32-bit address bus and a 32-bit data bus

The architecture of a computer system is the ***user-visible interface***: The structure and operation of the system as seen by the programmer. It includes:

- The instruction set the computer can obey
- The ways in which the instructions can specify the locations of data to be processed
- The type and representation of data
- - The format in which the instructions are stored in memory

In the computer the information is stored sequence of symbols of the binary alphabet - 0 and 1, each of which is represented to one of two stages conditions of physical

object. Set of these physical objects (the condenser of dynamic operative memory, the electronic trigger of static memory, a magnetic particle of a surface of a hard disk, etc.) makes memory of the computer. Processed data and the program should be necessarily placed in operative memory. Operative memory represents linear sequence of elements, each of which can be in a condition 0 or 1. Elements (or blocks of elements) are numbered from 0 and further. Access to them is carried out under numbers.

8 bit = 1 byte; 1024 byte = 1kilobyte; 1024K=1megabyte; 1024 M = 1 gigabyte

The executed program represents the sequence of machine codes located in operative memory - the binary figures interpreted by a control mean of the COMPUTER. The program is sequence of codes in Read Only Memory consists from 0 and 1.

The executed program is stored in memory of the computer in the form of machine commands - sequences of zero and units.

Follow next step how to receive the program in the form of machine commands:

1. Directly to write down these commands
2. To write the program on the Assembler
3. To use the programming language interpreter
4. To take advantage of the programming language compiler

Memory does not know what it stores. Must use enough memory for what we want to store. One reason for lots of problems with arrays. Each "cell" stores just one byte. Can find how much memory a datatype or object needs with the 'sizeof' operator.

```
cout << sizeof(int);        // 4
int i = 0;
cout << sizeof(i);          // 4
cout << sizeof(char);       // 1
int a[10];
cout << sizeof(a);          // 40 = 4 x 10
```

There are four basic data types in C/C++:

char - single byte capable of holding one character in the local character set. Represents a single *byte* (8 *bits*) of storage. Can be *signed* or *unsigne*. Internally char is just a number. Numerical value is associated with character via a *character set*. ASCII character set used in ANSI C

int - integer of unspecified size. Represents a signed integer of typically 4 or 8 bytes (32 or 64 bits). Precise size is machine-dependent.

float - single-precision floating point

double - double-precision floating point.

Float and double represent typically 32 and/or 64 bit real numbers. How these are represented internally and their precise sizes depend on the architecture. We won't obsess over this now.

Note that other types can be constructed using the modifiers: short, long, signed, unsigned

The precise sizes of these types is machine-specific. We will not worry about them for the time being. To find out the meaning of short int, etc. on a given system, use <limits.h>

3 examples of basic data types:

- int (used to declare numeric program variables of integer type)
- char (used to declare character variable)
- double (used to declare floating point variable)

In addition, there are float, void, short, long, etc. Declaration: specifies the type of a variable.
Example*: int local_var*;
Definition: assigning a value to the declared variable.
Example: *local_var = 5*;

Self-Assessment Exercise(s) 1

1. How is memory cell characterized?
2. How many buses are in a system? State them
3. How many bits are in 1 byte?
4. How many bytes are in 1 kilobyte?

3.2 Assembler level machine organization

An ***assembler*** creates object code by translating assembly instruction mnemonics into opcodes, and by resolving names for memory locations and other entities. The use of symbolic References/Further Reading is a key feature of assemblers, saving tedious calculations and manual address updates after program modifications. Most assemblers also include macro facilities for performing textual substitution—e.g., to generate common short sequences of instructions as inline, instead of called ***subroutines***.

There are two types of assemblers based on how many passes through the source are needed to produce the executable program.

One-pass assemblers go through the source code once. Any symbol used before it is defined will require "errata" at the end of the object code (or, at least, no earlier than the point where the symbol is defined) telling the linker or the loader to "go back" and overwrite a placeholder which had been left where the as yet undefined symbol was used.

Multi-pass assemblers create a table with all symbols and their values in the first passes, then use the table in later passes to generate code.

In both cases, the assembler must be able to determine the size of each instruction on the initial passes in order to calculate the addresses of subsequent symbols. This means that if the size of an operation referring to an operand defined later depends on the type or distance of the operand, the assembler will make a pessimistic estimate when first encountering the operation, and if necessary pad it with one or more "no-operation" instructions in a later pass or the errata. In an assembler with peephole optimization addresses may be recalculated between passes to allow replacing pessimistic code with code tailored to the exact distance from the target.

The original reason for the use of one-pass assemblers was speed of assembly— often a second pass would require rewinding and rereading a tape or rereading a deck of cards. Modern computers perform multi-pass assembly without unacceptable delay. The advantage of the multi-pass assembler is that the absence of errata makes the linking process (or the program load if the assembler directly produces executable code) faster. More sophisticated high-level assemblers provide language abstractions such as:

i. Advanced control structures
ii. High-level procedure/function declarations and invocations
iii. High-level abstract data types, including structures/records, unions, classes, and sets
iv. Sophisticated macro processing (although available on ordinary assemblers since late 1950s for IBM 700 series and since 1960's for IBM/360, amongst other machines)
v. Object-oriented programming features such
classes, objects, abstraction, polymorphism, and inheritance

A program written in assembly language consists of a series of (mnemonic) processor instructions and meta-statements (known variously as directives, pseudo-instructions and pseudo-ops), comments and data. Assembly language instructions usually consist of an opcode mnemonic followed by a list of data, arguments or parameters. These are translated by an assembler into machine language instructions that can be loaded into memory and executed.

For example, the instruction that tells an x86/IA-32 processor to move an immediate 8-bit value into a register. The binary code for this instruction is 10110 followed by a 3-bit identifier for which register to use. The identifier for the *AL* register is 000, so the following machine code loads the *AL* register with the data 01100001.

10110000 01100001

This binary computer code can be made more human-readable by expressing it in hexadecimal as follows

B0 61

Here, B0 means 'Move a copy of the following value into *AL'*, and 61 is a hexadecimal representation of the value 01100001, which is 97 in decimal. Intel assembly language provides the mnemonic MOV (an abbreviation of *move*) for instructions such as this, so the machine code above can be written as follows in assembly language, complete with an explanatory comment if required, after the semicolon. This is much easier to read and to remember.

MOV AL, 61h *; Load AL with 97 decimal (61 hex)*

In some assembly languages the same mnemonic such as MOV may be used for a family of related instructions for loading, copying and moving data, whether these are immediate values, values in registers, or memory locations pointed to by values in registers. Other assemblers may use separate opcodes such as L for "move memory to register", ST for "move register to memory", LR for "move register to register", MVI for "move immediate operand to memory", etc.

The Intel opcode 10110000 (B0) copies an 8-bit value into the *AL* register, while 10110001 (B1) moves it into *CL* and 10110010 (B2) does so into *DL*. Assembly language examples for these follow.

MOV AL, 1h; Load AL with immediate value 1

MOV CL, 2h; Load CL with immediate value 2

MOV DL, 3h; Load DL with immediate value 3

The syntax of MOV can also be more complex as the following examples show.

MOV EAX, **[**EBX**]**; Move the 4 bytes in memory at the address contained in EBX into EAX

MOV [ESI+EAX**]**, CL*;* Move the contents of CL into the byte at address ESI+EAX

In each case, the MOV mnemonic is translated directly into an opcode in the ranges 88-8E, A0-A3, B0-B8, C6 or C7 by an assembler, and the programmer does not have to know or remember which.

Transforming assembly language into machine code is the job of an assembler, and the reverse can at least partially be achieved by a disassembler. Unlike high-level languages, there is usually a one-to-one correspondence between simple assembly statements and machine language instructions. However, in some cases, an assembler may provide ***pseudoinstructions*** (essentially macros) which expand into several machine language instructions to provide commonly needed functionality. For example, for a machine that lacks a "branch if greater or equal" instruction, an assembler may provide a pseudo instruction that expands to the machine's "set if less than" and "branch if zero (on the result of the set instruction)". Most full-featured assemblers also provide a rich macro language (discussed below) which is used by vendors and programmers to generate more complex code and data sequences.

Each computer architecture has its own machine language. Computers differ in the number and type of operations they support, in the different sizes and numbers of registers, and in the representations of data in storage. While most general-purpose computers are able to carry out essentially the same functionality, the ways they do so differ; the corresponding assembly languages reflect these differences.

Multiple sets of mnemonics or assembly-language syntax may exist for a single instruction set, typically instantiated in different assembler programs. In these cases, the most popular one is usually that supplied by the manufacturer and used in its documentation.

Self-Assessment Exercise(s) 2

1. What is subroutines?
2. Enumerate the types of assemblers you know?
3. Pseudo instructions is…

3.3 Machine of Von Neuman

The computer consists of four main sub-systems (Figure 5.1):

1. Memory ALU (Arithmetic/Logic Unit)
2. Control Unit
3. Input/Output System (I/O)
4. Program is stored in memory during execution.
5. Program instructions are executed sequentially

Memory consists of many memory cells (storage units) of a fixed size. Each cell has an address associated with it: 0, 1, … All accesses to memory are to a specified address. A cell is the minimum unit of access (fetch/store a complete cell). The time it takes to fetch/store a cell is the same for all cells. When the computer is running, both program and Data (variables) are stored in the memory.

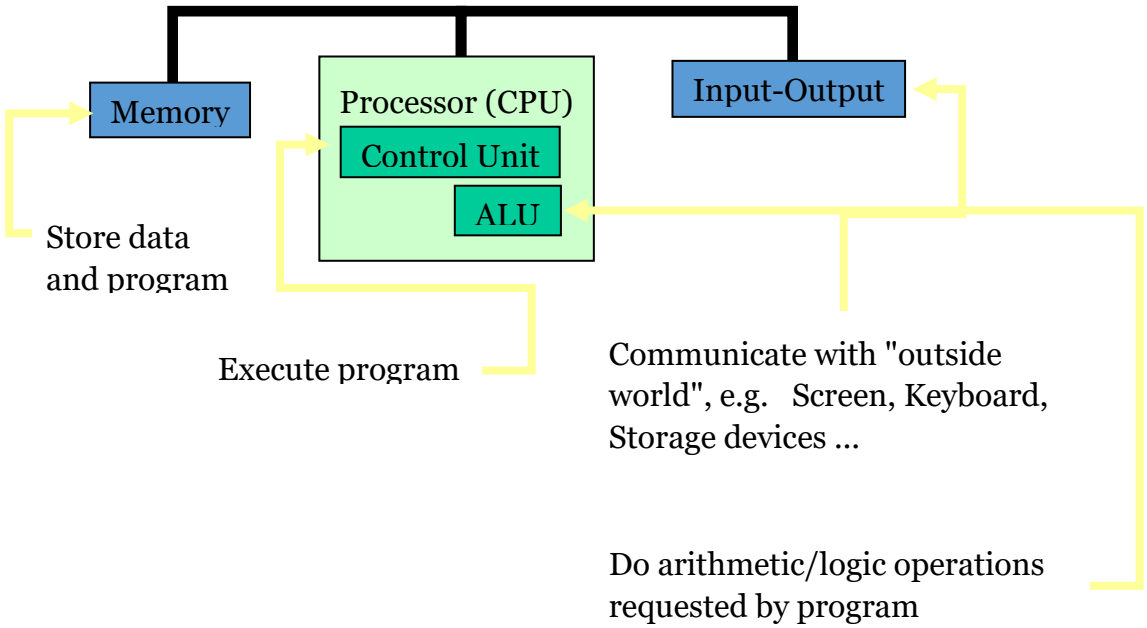


Fig. 5.1 Model for designing and building computers

The idea for the architecture was the factory/warehouse model, where the work was done in the factory on material that was stored in the warehouse.

Main Memory (RAM) is like the warehouse, storing the instructions (program) to be executed and also the data the program uses.

The computer consists of the CPU & Main Memory (RAM) and the bundle of wires (bus) connecting these modules to allow the CPU and RAM to exchange data (bit patterns).

RAM is a type of main memory constructed so that (1) the CPU can access any byte at any time without first accessing any other byte, and (2) the time it takes the CPU to access any byte of memory is the same no matter which byte it is accessing.

Dynamic RAM requires a charge to keep the capacitors (the state devices) properly charged. (Hence, when you turn off the computer, the charge disappears, causing your data, like your term paper, to disappear. Thus, save often!)

The Central Processing Unit (CPU) is like the factory, where programs are executed.

The only tasks the CPU knows how to execute are in the Arithmetic Logic Unit (ALU), consisting of some electrical circuits, each of which is responsible for doing one specific task.

CPU uses a special circuit, the Machine Cycle, to call on the ALU circuits as needed to execute programs that have been stored into RAM. (Ex. For your computer to execute MS Word, this program must first be stored into RAM, which is done when you click its icon on the desktop or start menu)

External Devices: To this computer are connected Input/Output devices (i.e., mouse, monitor, printer, microphone, and keyboard) and external storage devices (i.e., hard drive, flash drive). The built-in hard drive is really "external" to the computer (whose job is to compute, not deal in long term storage). When you "save" your work (e.g., term paper created in MS Word), you are saving to an external storage device that you selected. (Ex. C: drive is your built in hard drive. F: Removable Device might be your flash drive.)

Random Access Memory is a temporary storage medium in a computer. All data to be processed by the computer are transferred from a storage device or keyboard to RAM during data processing. Results obtained from executing any program are also stored in RAM. RAM is a volatile memory. Latest computers use RAM with a memory of more than 512 MB. There are provisions also available to increase the RAM memory in any computer.

RAM consists of many storage cells each of size 1 byte and is identified by using a number called as address or memory location. The memory address is assigned by the computer which also varies from computer to computer and time to time. The data stored in memory are identified using the memory address.

Read Only Memory is a permanent storage medium which stores start up programs. These programs which are loaded when computer is switched on. ROM stores

essentially the BIOS (Basic Input Output System) programs which are recorded by the manufacturer of the computer system. ROM is non-volatile memory.

ROM is also known as firmware. In ROM programs are burnt during manufacturing. Normally system programs and language translators are stored in ROM chips.

Both ROM and RAM are semiconductor chips. Normally size of the ROM holds 8k and more depending on the requirement.

Self-Assessment Exercise(s) 3

1. Which memory stores the instructions to be executed and also the data of a program?
2. Where do programs executed?
3. What permanent storage medium stores start up programs?

4.0 Conclusion

This unit introduced you to all levels machine organization. Also, you have learnt memory of machine Von Neuman.

You know more about organization of machine Von Neuman and higher and lower levels machine organization.

5.0 Summary

1. This unit introduced to all levels machine organization and organization of machine Von Neuman.
2. Assembler language for machine organization.

6.0 Tutor-Marked Assignment

1. How is memory cell characterized?
2. How many buses are in a system? State them
3. How many bits are in 1 byte?
4. How many bytes are in 1 kilobyte?
5. What is subroutines?
6. What types of assemblers do you know?
7. Pseudo instructions is...
8. Which memory store instructions to be executed and also the data of a program?
9. Where do programs executed?
10. What permanent storage medium stores start up programs?

7.0 References/Further Reading

1. Essentials of Computer Organization and Architecture. Linda Null and Julia Lobur, 2010.

2. Computer Systems. J. Stanley Warford, 2009
3. The Apollo Guidance Computer: Architecture and Operation (Springer Praxis Books / Space Exploration), Frank J. O'Brien, 2010.

Unit 2

Assembly Language Programming

Contents

1.0 Introduction

This unit introduces the execution of an instruction by a processor is divided in three parts. These parts are fetching, decode and execute. Also, you are going to learn machine language which dependent on varies from model to model of computers.

2.0 Learning Outcomes

At the end of this unit, you should be able to:

1. What are fetch, decode, and execution?
2. Introduction to Assembler language programming.

3.0 Learning Contents

3.1 Instruction fetch, decode, and execution

Most modern processors work on fetch-decode-execute principle. This is also called Von Neumann Architecture. When a set of instructions is to be executed, the instructions and data are loaded in main memory. The address of the first instruction is copied into the program counter. The execution of an instruction by a processor is divided in three parts. These parts are fetching, decode and execute.

Program is stored in memory as machine language instructions, in binary. The task of the is to execute programs by repeatedly:

- Fetch from memory the next instruction to be executed.
- Decode it, that is, determine what is to be done.
- Execute it by issuing the appropriate signals to the ALU, memory, and I/O subsystems.
- Continues until the HALT instruction

1. Fetch instruction

In the first step, the processor fetches the instruction from the memory. The instruction is transferred from memory to instruction register.

In the following figure, the processor is ready to fetch instruction. The instruction pointer contains the address 0100 contains the instruction MOV AX, 0.

The memory places the instruction on the data bus. The processor then copies the instruction from the data bus to the instruction register (Figure 5.2).

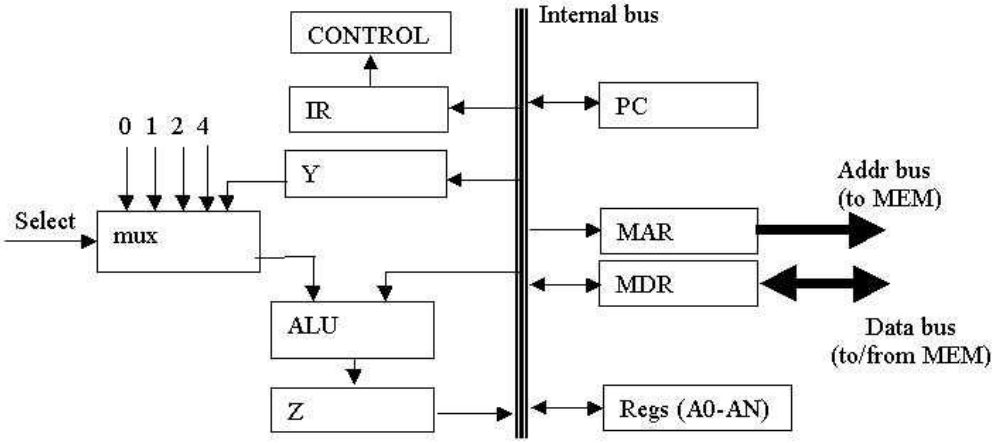


Fig. 5.2 Fetch instruction

2. Decode instruction

In this step, the instruction is decoded by the processor. The processor gets any operand if required by the instruction. For example, the instruction MOV AX, 0. Stores the value 0 in Ax register. The processor will fetch the constant value 0 from the next location in memory before executing the instruction.

In the above figure, the processor transfers the instruction from instruction register to the decode unit. The instruction tells the computer to store 0 into AX register. The decode unit now has all the details of how to do this (Figure 5.3).

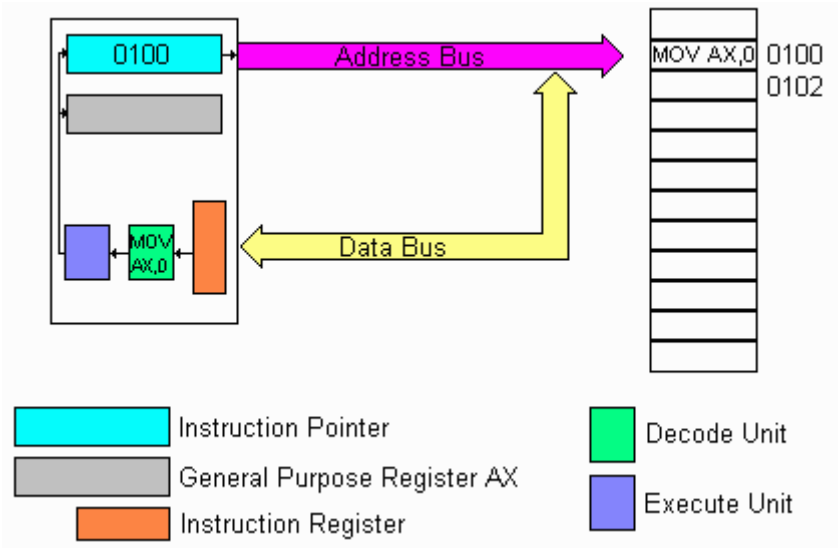


Fig. 5.3 Decode instruction

3. Execute instruction

In the last phase, the processor executes the instruction, it stores 0 in register AX. In above figure, the processor executes the instruction MOV AX, 0. Finally it adjusts the instruction pointer to point to next instruction to be executed stored at address 0102.

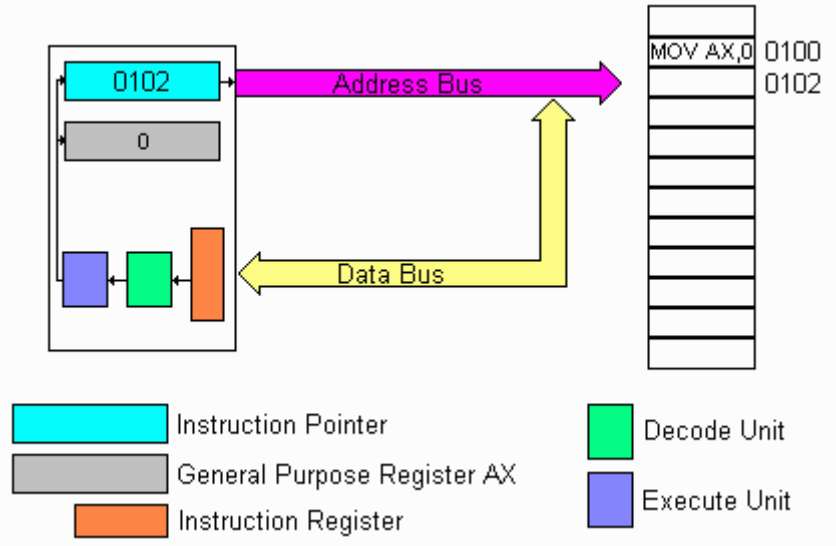


Fig. 5.4 Execute cycle, executing the instruction

Fetch (address):

- Fetch a copy of the content of memory cell with the specified address.
- Non-destructive, copies value in memory cell.

Store (address, value):

- Store the specified value into the memory cell specified by address.
- Destructive, overwrites the previous value of the memory cell.

The memory system is interfaced via:

- Memory Address Register (MAR)
- Memory Data Register (MDR)
- Fetch/Store signal

Self-Assessment Exercise(s) 1

1. The instruction being transferred from memory to instruction register is called…
2. The instruction decoded by the processor is called…
3. What does it mean to execute instruction?

3.2 Assembly language programming for a simulated machine

1. An operation code (OPCODE) that specifies the function to be performed.
2. One or more operand specifies that describe the locations of the information units on which the operation is performed.

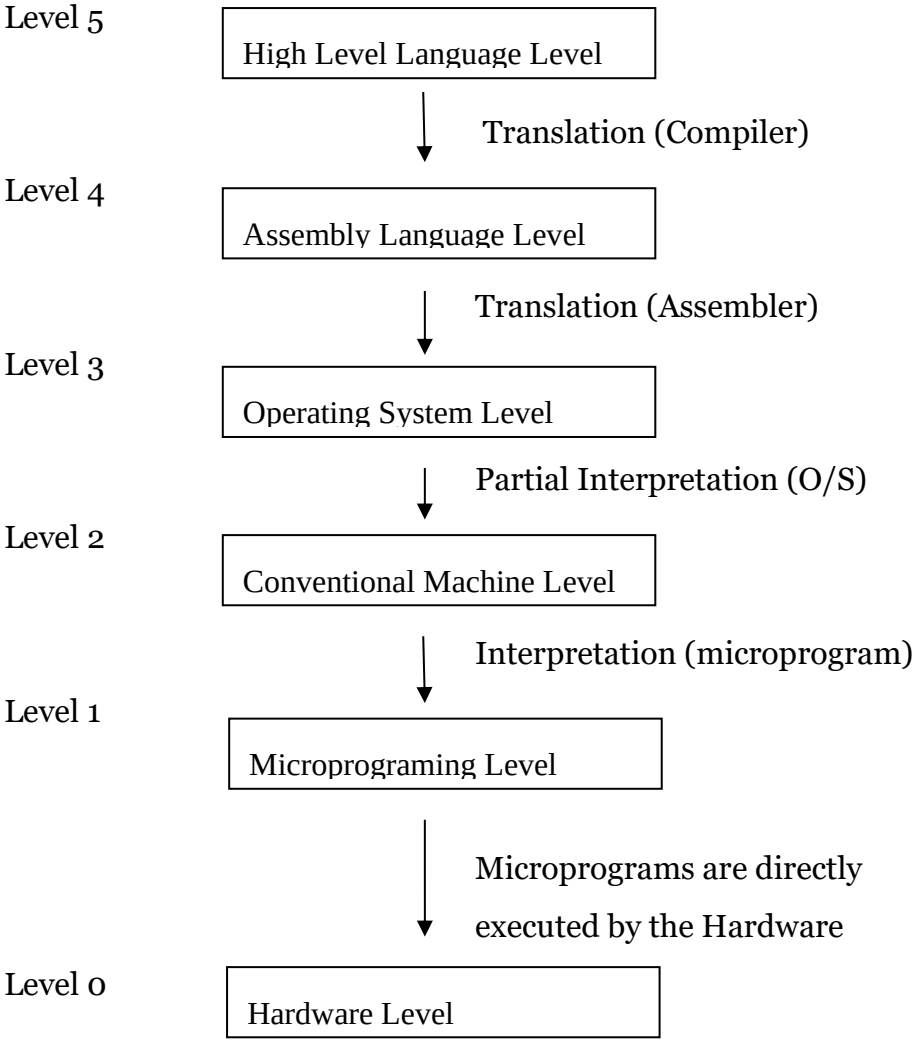
A computer by itself can't do anything. It needs software. A sequence of instructions called a program, makes the computer work.

Microprocessors by themselves only react to patterns of electrical signals. These patterns comprise what we call a ***machine language***, which consists of the binary

patterns of 0's and 1's that represent the signals the CPU understand. Machine language is machine dependent and varies from model to model of computers. Is a symbolic code that allows mnemonics for machine language instructions and symbolic names for memory locations?

It is a programming language in which instructions correspond closely to the individual machine instructions that are carried by a particular computer. So assembly language, like machine language, is machine dependent. The VAX/VMS assembly language is called MACRO.

Assembly language uses symbols to represent the operation codes while a special notation is used to represent the different addressing modes. A translator is used to translate the assembly code into machine code. This translator is called an ***assembler*** (Figure 5.5).



The name of the assembler we will use is MACRO. The most important system software is the operating system, which is an integrated system of programs that manages the system resources, and provides various support services such as the computer executing the application programs of users.

Self-Assessment Exercise(s) 2

1. What is machine language?
2. A translator that translates assembly code into machine code is called .....?
3. What is the lowest level of machine organization?
4. What is the highest level of machine organization?

4.0 Conclusion

This unit introduces all levels of machine organization. Also, you have learnt Assembler level machine organization and instruction fetch, decode, and execution. You know more about organization of machine Von Neuman and higher and lower levels machine organization.

5.0 Summary

1. This unit introduced instruction fetch, decode, and execution
2. Assembly language programming for a simulated machine

6.0 Tutor-Marked Assignment

1. The instruction is transferred from memory to instruction register called...
2. The instruction is decoded by the processor called...
3. What is execute in instruction?
4. What is machine language?
5. How called translator the assembly code into machine code?
6. What is the lowest level of machine organization?
7. What is the highest level of machine organization?

7.0 References/Further Reading

1. Assembly Language for x86 Processors (6th Edition), Kip R. Irvine, 2010.
2. Introduction to 80x86 Assembly Language and Computer Architecture. Richard C. Detmer, 2009.
3. The Art of Assembly Language, Randall Hyde, 2010.
4. Assembly Language Step-by-Step: Programming with Linux, Jeff Duntemann, 2009.
5. Essentials of 80X86 Assembly Language, Richard C. Detmer, 2011.

Module 6

Software Development Methodology

Unit 1: Fundamental design and concept principles

Unit 2: Testing and debugging strategies

Unit 1

Fundamental Design and Concept Principles

Contents

1.0 Introduction

This unit introduce software development methodology as a framework that is used to structure, plan, and control the process of developing an information system. You will know more about several software development approaches have been used since the origin of information technology.

2.0 Learning Outcomes

At the end of this unit, you should be able to:

1. Process of developing an information system
2. Waterfall: a linear framework
3. Prototyping: an iterative framework
4. Incremental: a combined linear-iterative framework
5. Spiral: a combined linear-iterative framework
6. Rapid application development (RAD): an iterative framework

3.0 Learning Contents

3.1 Process of Developing System

As a noun, a software development methodology is a framework that is used to structure, plan, and control the ***process of developing an information system*** - this includes the pre-definition of specific deliverables and artifacts that are created and completed by a project team to develop or maintain an application. The three basic approaches applied to software development methodology frameworks.

Software engineering is the practice of using selected process techniques to improve the quality of a software development effort. This is based on the assumption, subject to endless debate and supported by patient experience, that a methodical approach to software development results in fewer defects and, therefore, ultimately provides shorter delivery times and better value. The documented collection of policies, processes and procedures used by a development team or organization to practice software engineering is called its ***software development methodology*** (SDM) or ***system development life cycle*** (SDLC).

Every software development methodology approach acts as a basis for applying specific frameworks to develop and maintain software. Several software development approaches have been used since the origin of information technology. These are (Figure 6.1):

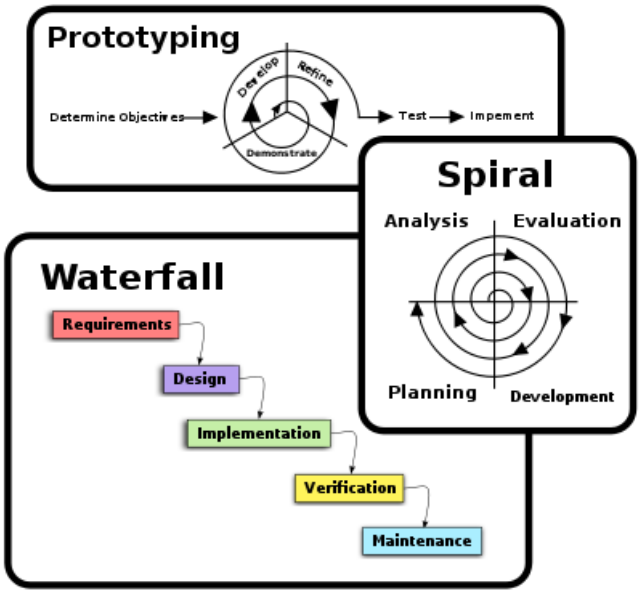


Fig.6.1 Developing an information system

i. Waterfall: a linear framework
ii. Prototyping: an iterative framework
iii. Incremental: a combined linear-iterative framework
iv. Spiral: a combined linear-iterative framework
v. Rapid application development (RAD): an iterative framework
vi. Extreme Programming

Self-Assessment Exercise(s) 1

1. State the processes involve in developing an information system?
2. Practice of software engineering is called…

3.2 Waterfall development

The ***Waterfall model*** is a sequential development approach, in which development is seen as flowing steadily downwards (like a waterfall) through the phases of requirements analysis, design, implementation, testing (validation), integration, and maintenance. The basic principles are:

- Project is divided into sequential phases, with some overlap and splashback acceptable between phases.
- Emphasis is on planning, time schedules, target dates, budgets and implementation of an entire system at one time.
- Tight control is maintained over the life of the project via extensive written documentation, formal reviews, and approval/signoff by the user and information technology management occurring at the end of most phases before beginning the next phase.

The Waterfall model is a traditional engineering approach applied to software engineering. It has been widely blamed for several large-scale government projects running over budget, over time and sometimes failing to deliver on requirements due to the ***Big Design Up Front*** approach. Except when contractually required, the Waterfall model has been largely superseded by more flexible and versatile methodologies developed specifically for software development.

Self-Assessment Exercise(s) 2

Sequential development approach is called…

Self-Assessment

3.3 Prototyping

Software prototyping, is the development approach of activities during software development, the creation of prototypes, i.e., incomplete versions of the software program being developed.
The basic principles are:

- Not a standalone, complete development methodology, but rather an approach to handle selected parts of a larger, more traditional development methodology (i.e. incremental, spiral, or rapid application development (RAD)).
- Attempts to reduce inherent project risk by breaking a project into smaller segments and providing more ease-of-change during the development process.
- User is involved throughout the development process, which increases the likelihood of user acceptance of the final implementation.
- Small-scale mock-ups of the system are developed following an iterative modification process until the prototype evolves to meet the users' requirements.
- While most prototypes are developed with the expectation that they will be discarded, it is possible in some cases to evolve from prototype to working system.
- A basic understanding of the fundamental business problem is necessary to avoid solving the wrong problem.

Self-Assessment Exercise(s) 3

1. Incomplete version of the software program being developed is called ……?

3.4 Incremental development

Various methods are acceptable for combining linear and iterative systems development methodologies, with the primary objective of each being to reduce

inherent project risk by breaking a project into smaller segments and providing more ease-of-change during the development process.

The basic principles are:

i. A series of Mini-Waterfalls are performed, where all phases of the Waterfall are completed for a small part of a system, before proceeding to the next increment, or
ii. Overall requirements are defined before proceeding to evolutionary, mini-Waterfall development of individual increments of a system, or
iii. The initial software concept, requirements analysis, and design of architecture and system core are defined via Waterfall, followed by iterative Prototyping, which culminates in installing the final prototype, a working system.

Self-Assessment Exercise(s) 4

1. Which software model combines linear and iterative systems development methodologies?

3.5 Spiral development

The spiral model is a software development process combining elements of both design and prototyping-in-stages, in an effort to combine advantages of top-down and bottom-up concepts. It is a meta-model, a model that can be used by other models. The basic principles are:

i. Focus is on risk assessment and on minimizing project risk by breaking a project into smaller segments and providing more ease-of-change during the development process, as well as providing the opportunity to evaluate risks and weigh consideration of project continuation throughout the life cycle.
ii. "Each cycle involves a progression through the same sequence of steps, for each part of the product and for each of its levels of elaboration, from an overall concept-of-operation document down to the coding of each individual program."
iii. Each trip around the spiral traverses four basic quadrants: (1) determine objectives, alternatives, and constraints of the iteration; (2) evaluate alternatives; Identify and resolve risks; (3) develop and verify deliverables from the iteration; and (4) plan the next iteration.
iv. Begin each cycle with an identification of stakeholders and their win conditions, and end each cycle with review and commitment (Figure 6.2).

Fig. 6.2 The spiral model

Self-Assessment Exercise(s) 5

i. State the model being used by other models?

3.6 Rapid application development

Rapid application development (RAD) is a software development methodology, which involves iterative development and the construction of prototypes. The basic principles are:

i. Key objective is for fast development and delivery of a high quality system at a relatively low investment cost.
ii. Attempts to reduce inherent project risk by breaking a project into smaller segments and providing more ease-of-change during the development process.
iii. Aims to produce high quality systems quickly, primarily via iterative Prototyping (at any stage of development), active user involvement, and computerized development tools. These tools may include ***Graphical User Interface*** (GUI) builders, ***Computer Aided Software Engineering*** (CASE) tools, ***Database Management Systems*** (DBMS), ***fourth-generation programming languages***, code generators, and object-oriented techniques.
iv. Key emphasis is on fulfilling the business need, while technological or engineering excellence is of lesser importance.
v. Project control involves prioritizing development and defining delivery deadlines or "timeboxes". If the project starts to slip, emphasis is on reducing requirements to fit the timebox, not in increasing the deadline.
vi. Generally includes ***joint application design*** (JAD), where users are intensely involved in ***system design***, via consensus building in either structured workshops, or electronically facilitated interaction.
vii. Active user involvement is imperative.
viii. Iteratively produces production software, as opposed to a throwaway prototype.
ix. Produces documentation necessary to facilitate future development and maintenance.

x. Standard systems analysis and design methods can be fitted into this framework.

Other practices

i. Other methodology practices include:
ii. ***Object-oriented*** development methodologies, such as Grady Booch's object-oriented design (OOD), also known as object-oriented analysis and design (OOAD). The Booch model includes six diagrams: class, object, state transition, interaction, module, and process.
iii. ***Top-down programming***: evolved in the 1970s by IBM researcher Harlan Mills (and Niklaus Wirth) in developed structured programming.
iv. ***Unified Process*** (UP) is an iterative software development methodology framework, based on Unified Modeling Language (UML). UP organizes the development of software into four phases, each consisting of one or more executable iterations of the software at that stage of development: inception, elaboration, construction, and guidelines. Many tools and products exist to facilitate UP implementation. One of the more popular versions of UP is the ***Rational Unified Process*** (RUP).
V. ***Agile software development*** refers to a group of software development methodologies based on iterative development, where requirements and solutions evolve via collaboration between self-organizing cross-functional teams.

Self-Assessment Exercise(s) 6

1. Which model involves iterative development and the construction of prototypes?

4.0 Conclusion

This unit introduced software development methodology as a framework that is used to structure, plan, and control the process of developing an information system.You know more about several software development approaches have been used since the origin of information technology.

5.0 Summary

1. This unit about software model development.
2. Waterfall- a linear framework; prototyping - an iterative framework; incremental - a combined linear - iterative framework; spiral - a combined linear-iterative framework; rapid application development - an iterative framework; extreme programming.

6.0 Tutor-Marked Assignment

1. State the processes involve in developing an information system?
2. Practice of software engineering is called…
3. Sequential development approach is called…
4. Incomplete versions of the software program being developed is called ….?
5. Which software model combines linear and iterative systems development methodologies?
6. Which model is being used by other models?
7. Which model involves iterative development and the construction of prototypest?

7.0 References/Further Reading

1. Agile Software Development: The Cooperative Game Alistair Cockburn, 2008
2. The Art of Agile Development, James Shore, Chromatic, 2007
3. The Software Project Manager's Bridge to Agility, Michele Sliger, Stacia Broderick, 2008
4. Growing Object-Oriented Software, Steve Freeman, Nat Pryce, 2009

Unit 2

Testing and Debugging Strategies

Contents

1.0 Introduction

This unit introduce to project phases and four major project phases: initiation, planning, execution and closure.
You will know more design review procedure, project risk management and testing methodology.

2.0 Learning Outcomes

At the end of this unit, you should be able to:

1. Identify four major project phases: initiation, planning, execution and closure.
2. Understand main design review procedures
3. Know what is Rick?
4. State model for testing project

3.0 Learning Contents

3.1 Project phases

Over the years, those involved in managing projects have observed that projects have special characteristics that can be exploited to manage them more effectively. One of those areas somewhat peculiar to the project environment deals with project phases:

i. Projects go through definite and describable phases;
ii. Each phase can be brought to some sense of closure as the next phase begins;
iii. Phases can be made to result in discrete products or accomplishments (e.g., test results) to provide the starting point for the next phase;
iv. The cost for each phase begins small and increase throughout the project, culminating in development, procurement, and the operations and support phases;
v. Phase transitions are ideal times to update planning baselines, to conduct high level management reviews, and to evaluate project costs and prospects.

Projects should be structured to take advantage of the natural phases that occur as work progresses. The phases should be defined in terms of schedule and also in terms of specific accomplishments. You should define how you will know when you are finished each phase and what you will have to show for it.

The Project Management Institute defines four ***major project phases***: ***initiation, planning, execution and closure***. One could make the case that almost every project goes through these four phases. Within these phase are smaller gradations. Some methodologies suggest decomposing projects into phases, stages, activities, tasks and steps.

Cost and schedule estimates, plans, requirements, specifications, and so forth, should be updated and evaluated at the end of each phase, sometimes before deciding whether to continue with the project. Large projects are usually structured to have major

program reviews at the conclusion of significant project phases. These decision-points in the life of a project are called Major Milestones.

The following illustrates how the concept of project phases is incorporated into a new product development methodology (Figure 6.3).

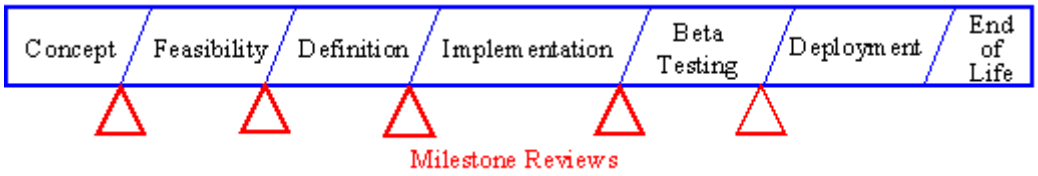


Fig. 6.3 Structures design

This illustrates the linking of major milestone review meetings with the completion of each phase. Milestone decisions are made after conducting a major program review where the project manager presents the approved statement of requirements, acquisition strategy, design progress, test results, updated cost and schedule estimates, and risk assessments, together with a request for authorization to proceed to the next phase.

The early phases will shape the direction for all further efforts on the project. They provide requirements definitions, evaluation of alternative approaches, assessment of maturity of technologies, review of cost, schedule and staffing estimates, and development of specifications.

Milestone completions can be defined in terms of "exit criteria" as well as by calendar dates. Using "event based" schedules rather than date-based schedules ties project phase completions to the successful achievement of predetermined criteria such as completion of testing, demonstration of prototypes, adequacy of technical documentation, or approval of conceptual designs and specifications.

A relatively short-term or technically straight-forward project may have only one approval event, following a proposal or feasibility study. Nevertheless, the project manager should report to customers and interested senior managers at intervals to keep them up to date on project progress and to ensure the continuing soundness of the project direction and requirements.

On small projects, if no formal agreements are written, the project manager should deal with customers and sponsors in an informal yet somewhat ***contractual*** way. This means managing expectations and making clear agreements about what will be produced and when.

If project phases take place over many months or even years, it is vital to provide interim deliverables to give the customers and sponsors a sense that work is being accomplished, to provide an opportunity for feedback, and to capture project successes in documented form.

The project planning process should be built around the project life cycle. Particular care should be given to defining the work to be accomplished in each phase. This should include definition of the deliverables to be produced, identifying testing and demonstrations to be completed, preparing updates of cost and schedule estimates, re-assessing risks, and conducting formal technical and management reviews.

If your project runs into an immovable obstacle and progress comes to a complete halt, you may want to declare victory and bring that phase to a close. This can be done by documenting the work already completed, and then writing a report describing the work successfully completed and defining the steps required should project sponsors decide to proceed.

Self-Assessment Exercise(s) 1

i. Which project management institute defines major project phases?

3.2 Design review procedures

System Requirements Review (SRR). The purpose of the SRR is to ***review the system requirements specification*** *document*, to ensure the documented requirements reflect the current knowledge of the customer and market requirements, to identify requirements that may not be consistent with product development constraints, and to put the requirements document under version control to serve as a stable baseline for continued new product development.

Preliminary Design Review (PDR). The purpose of the PDR is to review the ***conceptual design*** to ensure that the planned technical approach will meet the requirements.

Critical Design Review (CDR). The purpose of the CDR is to review the detailed design to ensure that the design implementation has met the requirements.

Test Readiness Review (TRR). The purpose of the TRR is to review ***preparations and readiness for testing of software configuration items***, including adequate version identification of software and test procedures.

Production Readiness Review (PRR). The purpose of the PRR is to ensure that the design is completely and accurately documented and ready for ***formal release to manufacturing.***

Self-Assessment Exercise(s) 2

1. What is the purpose of System Requirements Review (SRR)?
2. What is the purpose of Preliminary Design Review (PDR)?
3. What do you mean by Test Readiness Review (TRR)?
4. What is Production Readiness Review (PRR)?

3.3 Project Risk Management

Risk is a combination of the probability of a negative event and its consequences. If an event is inevitable but inconsequential, it does not represent a risk, because it has no impact. Alternatively, an improbable event with significant consequences may not be a high risk. These two factors are combined in what we experience as the possibility of loss, failure, danger, or peril.

$$\text{Project Risk} = \sum (\text{Events} * \text{Probabilities} * \text{Consequences})$$

Risk analysis consists of ***risk identification, probability assessment, and impact estimate***. Start by identifying all the risk events that can occur on your project. Then estimate the probability of each event happening. Finally, estimate the impact in hours or dollars if the event occurs. Once you have listed and quantified the project risks, then you should prepare a risk management plan for each significant risk item. The final step would be to formalize this into a risk management activity, establish metrics, and track your top ten risks week by week.

An easy way to reduce risk is to have less ambitious goals. After evaluating risks, one can choose a path of risk avoidance or risk mitigation and management. If we understand the risks on a project, we can decide which risks are acceptable and take actions to mitigate or forestall those risks. If our project risk assessment determines risks are excessive, we may want to consider restructuring the project to within acceptable levels of risk.

Risks that do not offer the potential for gain (profit?) should be avoided. Risks associated with achieving challenging and worthwhile goals should be managed. One way to reduce risk is to gather information about relevant issues to lower the level of uncertainty. Then we can look for ways to reduce probabilities of failures or to reduce their consequences.

What may look like an unacceptable risk to one person might rightly appear as an attractive opportunity to another. The difference is vision.

Items in the project plan that are important and that are uncertain of success should be considered risk areas and given special attention. Risk should be associated with areas where the scope is not well defined or is subject to change. An unproven or immature technical approach, or known technical difficulty or complexity will increase project risk. Ambitious goals always result in risk. Unfamiliarity with the process, or inexperienced personnel, constitute project risks, if for no other reason than being unknowns. Exterior interfaces cause risks because they can change and, even if they don't change, their descriptions or specifications may be inaccurate. Exterior organizational dependencies create project risks. Incomplete planning or optimistic cost or schedule goals create risk. If the customer is involved in schedule dependencies for document review and approval or for delivering process information, this creates project risks. Conversely, project risks are created if the customer is not involved in periodic review of the system design and project plans.

Any area over which the project manager does not have control can be project risks. Anything that is not well understood, anything that is not well documented, and anything that can change, these all create project risks. Things that haven't been tested are always at risk.

An organizational culture that has previously had problems executing projects will be likely to repeat the same mistakes. These problem areas should be understood and managed as significant project risks. They must be counteracted by specific bold mitigating management initiatives or repeated failures are guaranteed

The known unknowns are more likely to be project risks than the unknown unknowns. This means you should trust your instincts and pay attention to what seems risky to you. It is more likely you will have problems from known risk areas than be surprised by things completely unforeseen.

Self-Assessment Exercise(s) 3

1. What is Risk?
2. What constitute Risk analysis?

3.4 Testing methodology

he ***V Model*** is a software development and testing model, which helps to highlight the need to plan and prepare for testing early in the development process. In the V Model, each development phase is linked to a corresponding testing phase. In practice, the V Model provides a powerful tool for managing and controlling risk within the testing component of a software development project (Figure 6.4).

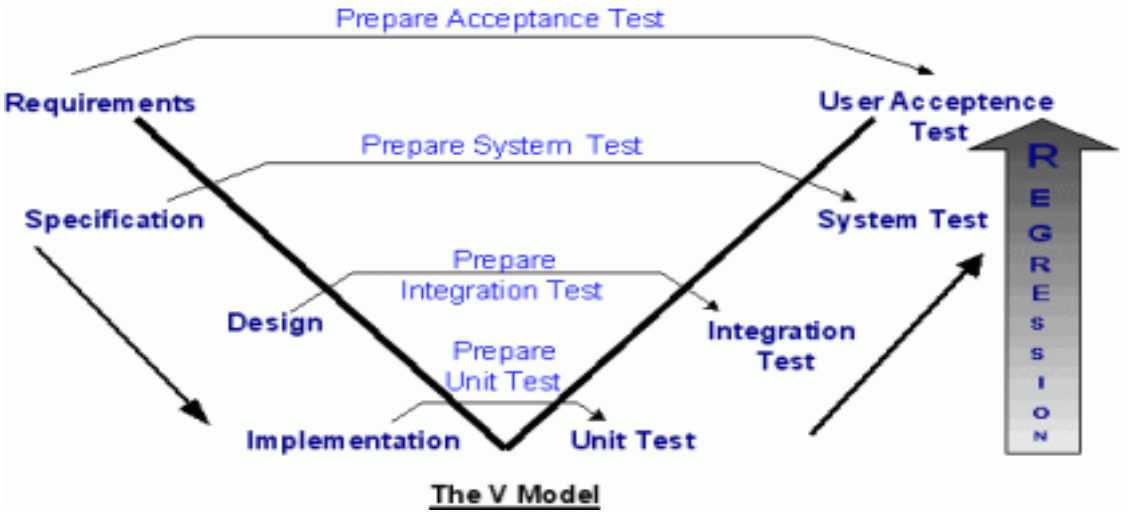


Fig. 6.4 Testing model

The international standard describing the method to select, implement and monitor the life cycle for software is ***ISO/IEC 12207***.

A decades-long goal has been to find repeatable, predictable processes that improve productivity and quality. Some try to systematize or formalize the seemingly unruly

task of writing software. Others apply project management methods to writing software. Without project management, software projects can easily be delivered late or over budget. With large numbers of software projects not meeting their expectations in terms of functionality, cost, or delivery schedule, effective project management appears to be lacking.

Self-Assessment Exercise(s) 4

1. Which model helps to plan and prepare for testing early in the development process?
2. Which international standard describing the method to select, implement and monitor the life cycle for software?

4.0 Conclusion

This unit introduced you to project and four major project phases: initiation, planning, execution and closure. You know more design review procedure, project risk management and testing methodology.

5.0 Summary

1. This unit developing software technology and testing methodology.
2. Four major project phases: initiation, planning, execution and closure.
3. Main design review procedures
4. Model for testing project

6.0 Tutor-Marked Assignment

1. Which project management institute defines major project phases?
2. What is the purpose of System Requirements Review (SRR)?
3. What is the purpose of Preliminary Design Review (PDR)?
4. What do you mean Test Readiness Review (TRR)?
5. What is Production Readiness Review (PRR)?
6. What is Risk?
7. Which consist Risk analysis?
8. Which model helps to plan and prepare for testing early in the development process?
9. Which international standard describing the method to select, implement and monitor the life cycle for software?

7.0 References/Further Reading

1. Inside Windows Debugging: A Practical Guide to Debugging and Tracing Strategies in Windows. Tarik Soulami, 2012.

2. Advanced Windows Debugging. Mario Hewardt and Daniel Pravat, 2007.
3. The Developer's Guide to Debugging: 2nd Edition. Thorsten Grötker, Ulrich Holtmann, Holger Keding and Markus Wloka, 2012.
4. Debugging at the Electronic System Level. Frank Rogin and Rolf Drechsler, 2010.

Module 7

Recursion

Unit 1: Concept of recursion

Unit 2: Simple recursive procedures

Unit 1

Concept of Recursion

Contents

1.0 Introduction

This unit introduces the concept of recursion, characteristics of recursion and you will learn recursive mathematical function. Also, you are going to learn how to use recursive mathematical function
You will know more about using recursion in object oriented programming languages C/C++.

2.0 Learning Outcomes

At the end of this unit, you should be able to:
1. What is recursion?

2. Base characteristics of recursion

3. Example recursive mathematical function

3.0 Learning Contents

3. 1 What is recursion?

Recursion is the process of repeating items in a self-similar way. For instance, when the surfaces of two mirrors are exactly parallel with each other the nested images that occur are a form of infinite recursion. The term has a variety of meanings specific to a variety of disciplines ranging from linguistics to logic. The most common application of recursion is in mathematics and computer science, in which it refers to a method of defining functions in which the function being defined is applied within its own definition. Specifically, this defines an infinite number of instances (function values), using a finite expression that for some instances may refer to other instances, but in such a way that no loop or infinite chain of References/Further Reading can occur. The term is also used more generally to describe a process of repeating objects in a self-similar way.
In general, ***recursion*** means self-repeating patterns. In Mathematics, it can be a function that is defined in terms of itself, such as factorial, Fibonacci etc. In computer programming, recursion means a function that is defined in terms of itself. In other words, a function that calls itself. Every recursive function has a termination condition; otherwise, it will call itself forever, and this condition can be called the base condition.

In C++, the types of recursion can be defined in more than one dimension. In one dimension, it can be categorized as runtime recursion and compile time recursion using template meta-programming.

Runtime recursion is the most common recursion technique used in C++. This can be implemented when a C++ function (or member function) calls itself.

In C++, we can also do compile time recursion with the help of template meta-programming. When you instantiate a template class (or structure) in C++, the

compiler will create the code of that class at compile time. Just like runtime recursion, we can instantiate the template class itself to perform the recursion. Just like runtime recursion, we also need the termination condition; otherwise, instantiation will go forever, at least theoretically, but, of course, limited to the resources of the computer and the compiler. In template meta-programming, we can specify the termination condition (or base condition) with the help of template specialization or partial template specialization, depending on the termination condition.

One can think that we might do the same thing with the preprocessor of C++, using macros, because they will also be replaced during compilation. In fact, technically, preprocessor replaces all macros even before compilation, so it is not performing at compile time. The preprocessor also has lots of limitations like, there is no debug symbol defined for the debugger because of simple text replacement, but the most critical limitation is it can not be recursive. The other way to look at recursion is how a recursive algorithm is implemented. Recursive algorithms can be implemented in more than one way, such as linear, tail, mutual, binary, or nested recursion. We can implement them either at compile time using template meta-programming, or at runtime using functions or member functions.

We can represent the different types of recursion using the following diagram. This diagram shows the different types of recursion based on their implementation (i.e., linear, tail, mutual etc.) and when it will be performed (Figure 7.1).

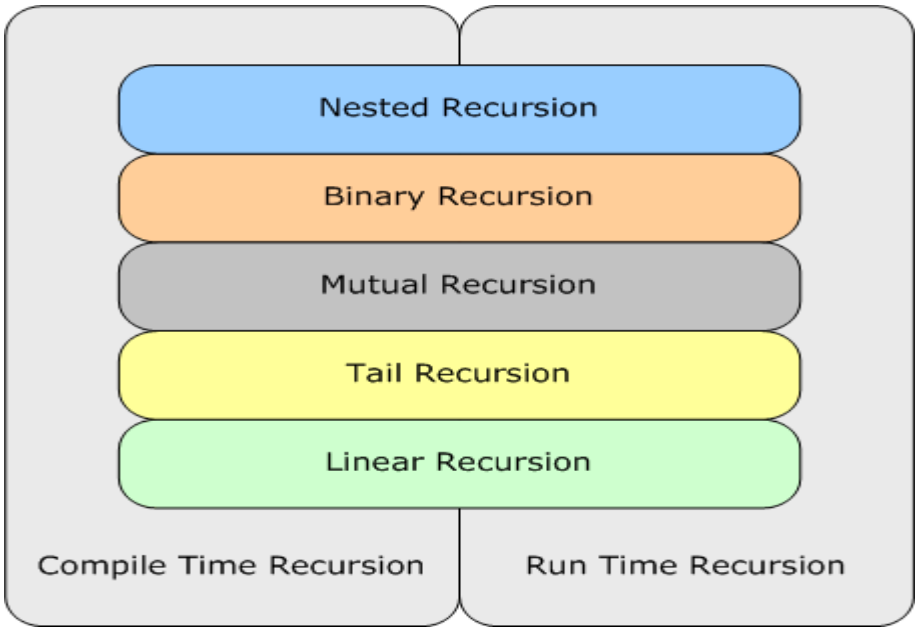


Fig. 7.1 Different types of recursion

Simply put, ***recursion*** is when a function calls itself. That is, in the course of the function definition there is a call to that very same function. At first this may seem like a never ending loop, or like a dog chasing its tail. It can never catch it. So too it seems our function will never finish. This might be true is some cases, but in practice we can check to see if a certain condition is true and in that case exit (return from) our function. The case in which we end our recursion is called a ***base case***. Additionally,

just as in a loop, we must change some value and incrementally advance closer to our base case.
A class of objects or methods exhibit recursive behavior when they can be defined by two properties:

1. A simple base case (or cases), and
2. A set of rules which reduce all other cases toward the base case.

For example, the following is a recursive definition of a person's ancestors:

1. One's parents are one's ancestors (*base case*).
2. The parents of one's ancestors are also one's ancestors (*recursion step*).
3. The Fibonacci sequence is a classic example of recursion:
4. Fib(0) is 0 [base case]
5. Fib(1) is 1 [base case]
6. For all integers n > 1: Fib(n) is (Fib(n-1) + Fib(n-2)) [recursive definition]

Consider this function.

```
void myFunction( int counter)
{
if(counter == 0)
   return;
else
    {
    cout <<counter<<endl;
    myFunction(--counter);
    return;
    }
}
```

This recursion is not infinite, assuming the function is passed a positive integer value. What will the output be?
Consider this function:

```
void myFunction( int counter)
{
if(counter == 0)
   return;
else
    {
    cout<<"hello"<<counter<<endl;
    myFunction(--counter);
    cout<<counter<<endl;
    return;
    }
}
```

If the function is called with the value 4, what will the output be? Explain. The above recursion is essentially a loop like a for loop or a while loop. When do we prefer

recursion to an iterative loop? We use recursion when we can see that our problem can be reduced to a simpler problem that can be solved after further reduction.

Self-Assessment Exercise(s) 1

1. What is recursion?
2. The case in which we end our recursion is called …

3.2 Characteristics of recursion

Every recursion should have the following characteristics.

i. A simple base case which we have a solution for and a return value. Sometimes there are more than one base cases.
ii. A way of getting our problem closer to the base case. I.e. a way to chop out part of the problem to get a somewhat simpler problem.
iii. A recursive call which passes the simpler problem back into the function.

The key to thinking recursively is to see the solution to the problem as a smaller version of the same problem. The key to solving recursive programming requirements is to imagine that your function does what its name says it does even before you have actually finish writing it. You must pretend the function does its job and then use it to solve the more complex cases. Here is how.

Identify the base case(s) and what the base case(s) do. A base case is the simplest possible problem (or case) your function could be passed. Return the correct value for the base case. Your recursive function will then be comprised of an if-else statement where the base case returns one value and the non-base case(s) recursively call(s) the same function with a smaller parameter or set of data. Thus you decompose your problem into two parts: (1) The simplest possible case which you can answer (and return for), and (2) all other more complex cases which you will solve by returning the result of a second calling of your function. This second calling of your function (recursion) will pass on the complex problem but reduced by one increment. This decomposition of the problem will actually be a complete, accurate solution for the problem for all cases other than the base case. Thus, the code of the function actually has the solution on the first recursion.

Self-Assessment Exercise(s) 2

1. What is a simple base case?
2. What call a recursive?

3.3 Recursive mathematical functions

Recursion is a lot like proof by mathematical induction. That is, you show that *if* a statement is true for any number, n, then it must also be true for n+1. Or we could show that if the statement is true for n-1 then it must be true for n. Then you show that

it is true for n=0 (or n=1). So too, in solving a a problem recursively, we write our function to work for n while we assume the function works for n-1. We then make the function work in the base case, i.e. where n=1. The key is to assume our function works when writing it. That is why we can call it from within itself. The logic is that if it works for n=1 (the base case with an explicit return), then when solving the problem for n=2 we can assume it works for n=1 because we have explicitly solved that problem. Now the solution for n=3 must also work, since we have a solution for n=2 (i.e. n = 3-1). This thinking can go on *ad infinitum* and therefore we have solved the problem for any n. Let's consider writing a function to find the factorial of an integer. For example 7! equals 7*6*5*4*3*2*1. But we are also correct if we say 7! equals 7*6!. In seeing the factorial of 7 in this second way we have gained a valuable insight. We now can see our problem in terms of a simpler version of our problem and we even know how to make our problem progressively more simple. We have also defined our problem in terms of itself. I.e. we defined 7! in terms of 6!. This is the essence of recursive problem solving. Now all we have left to do is decide what the base case is. What is the simplest factorial? 1!. 1! equals 1. Let's write the factorial function recursively.

```
int myFactorial( int integer)
{
if( integer == 1)
   return 1;
else
     {
     return (integer * (myFactorial(integer-1)));
     }
}
```

Note that the base case (the factorial of 1) is solved and the return value is given. Now let us imagine that our function actually works. If it works we can use it to give the result of more complex cases. If our number is 7 we will simply return 7 * the result of factorial of 6. So we actaully have the exact answer for all cases in the top level recursion. Our problem is getting smaller on each recursive call because each time we call the function we give it a smaller number. Try running this program in your head with the number 2. Does it give the right value? If it works for 1 then it must work for two since 2 merely returns 2 * factorial of 1. Now will it work for 3? Well, 3 must return 3 * factorial of 2. Now since we know that factorial of 2 works, factorial of 3 also works. We can prove that 4 works in the same way, and so on and so on.

Food for thought: ask yourself, could this be written iteratively? Note: make it your habit of writing the base case in the function as the first statement.

Note: Forgetting the base case leads to infinite recursion. However, in fact, your code won't run forever like an infinite loop, instead, you will eventually run out of stack space (memory) and get a run-time error or exception called a stack overflow. There are several significant problems with recursion. Mostly it is hard (especially for inexperienced programmers) to think recursively, though many AI specialists claim that in reality recursion is closer to basic human thought processes than other

programming functions (such as iteration). There also exists the problem of stack overflow when using some forms of recursion (head recursion.) The other main problem with recursion is that it can be slower to run than simple iteration. Then why use it? It seems that there is always an iterative solution to any problem that can be solved recursively. Is there a difference in computational complexity? No. Is there a difference in the efficiency of execution? Yes, in fact, the recursive version is usually less efficient because of having to push and and pop recursions on and off the run-time stack, so iteration is quicker. On the other hand, you might notice that the recursive versions use fewer or no local variables.

So why use recursion? The answer to our question is predominantly because it is easier to code a recursive solution once one is able to identify that solution. The recursive code is usually smaller, more concise, more elegant, and possibly even easier to understand, though that depends on ones thinking style. But also, there are some problems that are very difficult to solve without recursion. Those problems that require backtracking such as searching a maze for a path to an exit or tree based operations are best solved recursively. There are also some interesting sorting algorithms that use recursion.

Self-Assessment Exercise(s) 3

1. Write example of recursive mathematical functions.
2. Why use recursion?

4.0 Conclusion

This unit introduced you to recursion and characteristics of recursion, you have learnt recursive mathematical function and know recursive mathematical function.

5.0 Summary

1. The recursive code is usually smaller, more concise, more elegant, and possibly even easier to understand, though that depends on ones thinking style.
2. Characteristics of recursion
3. Recursive mathematical function

6.0 Tutor-Marked Assignment

1. What is recursion?
2. The case in which we end our recursion is called …
3. What is a simple base case?
4. What call a recursive?
5. Write example of recursive mathematical functions.
6. Why use recursion?

7.0 References/Further Reading

1. Computability Theory: An Introduction to Recursion Theory. Herbert B. Enderton, 2010.
2. Recursion. Jesse Russell and Ronald Cohn, 2012.
3. Programs, Recursion and Unbounded Choice (Cambridge Tracts in Theoretical Computer Science). Wim H. Hesselink, 2005.
4. Computability Theory: An Introduction to Recursion Theory, Students Solutions Manual (e-only). Herbert B. Enderton, 2011.

Unit 2

Simple Recursive Procedures

Contents

1.0 Introduction

This unit introduce you to the examples of simple recursion procedures of programming language C/C++. Also, you will learn implementation of recursion. Also, you are going to learn how to create simple program using recursion in high level language.

2.0 Learning Outcomes

At the end of this unit, you should be able to state:

1. Three main examples of simple recursion procedures
2. Implementation of recursion

3.0 Learning Contents

3.1 Examples of simple recursion procedures

3.1.1 Towers of Hanoi

This problem comes from history, monks in Vietnam were asked to carry 64 gold disks from one tower (stack) to another. Each disk is of a different size. There are 3 stacks, a source stack, a destination stack and an intermediate stack. A disk is placed on one of three stacks but no disk can be placed on top of a smaller disk. The source tower holds 64 disks. How will the monks solve this problem? How long will it take them?

The easiest solution is a recursive one. The key to the solution is to notice that to move any disk, we must first move the smaller disks off of it, thus a recursive definition. Another way to look at it is this, if we had a function to move the top three disks to the middle position, we could put the biggest disk in its place. All we need to do is assume we have this function and then call it.

Let's start with 1 disk (our base case): Move 1 disk from start tower to destination tower and we are done. To move 2 disks: Move smaller disk from start tower to intermediate tower, move larger disk from start tower to final tower, move smaller disk from intermediate tower to final tower and we are done. To move n disks (or think of, say, 3 disks): Solve the problem for n - 1 disks (i.e. 2 disks) using the intermediate tower instead of the final tower (i.e. get 2 disks onto the intermediate tower). Then, move the biggest disk from start tower to final tower. Then again solve the problem for n - 1 disks but use the intermediate tower instead of the start tower (i.e. get the 2 disks onto the final tower using the start tower as the intermediate tower).

3.1.2 Russian Dolls

One of the properties of recursive functions is that a given function call must wait for the following function call to complete its run before it itself can finish. A convenient metaphor would be that of Russian dolls. If you have a set of dolls within dolls and you take them apart you will first open the outermost doll, then the next, and so on till you

have opened the innermost doll. Now when you put them back together, you cannot start with the outermost doll. You first must connect the innermost doll and then put it inside the next larger doll, and then connect it. Last of all you will reconnect the outermost doll around all the inner dolls. So too with recursion. You must finish the innermost function call before you can finish the next inner function call. You should see that the first function call (the outermost one) will be the last one to complete.

3.1.3 Tail Recursion

Tail recursion is defined as occurring when the recursive call is at the end of the recursive instruction. This is not the case with my factorial solution above. It is useful to notice when ones algorithm uses tail recursion because in such a case, the algorithm can usually be rewritten to use iteration instead. In fact, the compiler will (or at least should) convert the recursive program into an iterative one. This eliminates the potential problem of stack overflow.

This is not the case with head recursion, or when the function calls itself recursively in different places like in the Towers of Hanoi solution. Of course, even in these cases we could also remove recursion by using our own stack and essentially simulating how recursion would work.

In this example of factorial above the compiler will have to call the recursive function before doing the multiplication because it has to resolve the (return) value of the function before it can complete the multiplication. So the order of execution will be "head" recursion, i.e. recursion occurs before other operations.

To convert this to tail recursion we need to get all the multiplication finished and resolved before recursively calling the function. We need to force the order of operation so that we are not waiting on multiplication before returning. If we do this the stack frame can be freed up.

The proper way to do a tail-recursive factorial is this:

```
int factorial(int number) {
   if(number == 0) {
        return 1;
      }
    return  factorial_i(number, 1);
}
int factorial_i(int currentNumber, int sum) {
   if(currentNumber == 1) {
      return sum;
   } else {
      return factorial_i(currentNumber - 1, sum*currentNumber);
   }
}
```

Self-Assessment Exercise(s) 1

1. Which examples of recursive you know?

3.2 Implementation of recursion

1. Example to print numbers counting down:

```
void print(int p)
{
if (p==0)
  return;
cout<<p;
print(p-1);
return;
}
```

2 Example to print counting up

```
void print(int p)
{
if (p==0)
  return;
print(p-1);
cout<<p;
return;
}
```

3. Fibonacci(int n)

```
{
if (n==0)
   return 0;
if (n==1)
   return 1;
return( Fibonacci(n-2) + Fibonacci(n-1) );
}
```

4. A simple example of recursion would be:

```
void recurse()
{
 recurse(); //Function calls itself
}
int main()
{
 recurse(); //Sets off the recursion
}
```

This program will not continue forever, however. The computer keeps function calls on a stack and once too many are called without ending, the program will crash. Why not write a program to see how many times the function is called before the program terminates?

```
#include <iostream>
```

```
using namespace std;
void recurse ( int count ) // Each call gets its own count
{
 cout<< count <<"\n";
 // It is not necessary to increment count since each function's
 // variables are separate (so each count will be initialized one greater)
 recurse ( count + 1 );
}
int main()
{
 recurse ( 1 ); //First function call, so it starts at one
}
```

This simple program will show the number of times the recurse function has been called by initializing each individual function call's count variable one greater than it was previous by passing in count + 1. Keep in mind, it is not a function restarting itself, it is hundreds of functions that are each unfinished with the last one calling a new recurse function.
It can be thought of like the Russian dolls that always have a smaller doll inside. Each doll calls another doll, and you can think of the size being a counter variable that is being decremented by one.

Think of a really tiny doll, the size of a few atoms. You can't get any smaller than that, so there are no more dolls. Normally, a recursive function will have a variable that performs a similar action; one that controls when the function will finally exit. The condition where the function will not call itself is termed the base case of the function. Basically, it is an if-statement that checks some variable for a condition (such as a number being less than zero, or greater than some other number) and if that condition is true, it will not allow the function to call itself again. (Or, it could check if a certain condition is true and only then allow the function to call itself).

A quick example:

```
void doll ( int size )
{
 if ( size == 0 )      // No doll can be smaller than 1 atom (10^0==1) so doesn't call itself
  return;              // Return does not have to return something, it can be used
                       //  to exit a function
 doll ( size - 1 );    // Decrements the size variable so the next doll will be smaller.
}
int main()
{
```

```
    doll ( 10 );          //Starts off with a large doll (it's a logarithmic scale)
}
```

This program ends when size equals one. This is a good base case, but if it is not properly set up, it is possible to have a base case that is always true (or always false). Once a function has called itself, it will be ready to go to the next line after the call. It can still perform operations. One function you could write could print out the numbers 123456789987654321. How can you use recursion to write a function to do this? Simply have it keep incrementing a variable passed in, and then output the variable...twice, once before the function recurses, and once after...

```
void printnum ( int begin )
{
 cout<< begin;
 if ( begin < 9 )              // The base case is when begin is greater than 9
 {                     //  for it will not recurse after the if-statement
   printnum ( begin + 1 );
 }
 cout<< begin;       // Outputs the second begin, after the program has
                     //  gone through and output
}
```

This function works because it will go through and print the numbers begin to 9, and then as each printnum function terminates it will continue printing the value of begin in each function from 9 to begin. This is just the beginning of the usefulness of recursion. Here's a little hallenge, use recursion to write a program that returns the factorial of any number greater than 0. (Factorial is number * (number - 1) * (number - 2) ... * 1).

Self-Assessment Exercise(s) 2

1. Which examples of recursive you know?
2. Which operators different between examples of programs of print numbers counting down and print counting up?
3. Write example of Fibonacci recursion?
4. Develop program which print next numbers: 123456789987654321.

4.0 Conclusion

This unit introduced to simple examples of recursion and programs in language C/C++ using mathematical recursion. You have learnt recursive mathematical function.

5.0 Summary

1. This unit introduced three main simple recursions and using of programming languages of C/C++.

2. Recursion to calculate of Fibonacci, Factorial.

6.0 Tutor-Marked Assignment

1. Which operators differentiate between examples of programs of print numbers counting down and print counting up?
2. Write example of recursion Fibonacci?
3. Develop program which print next numbers: 1234567899876543 21.

7.0 References/Further Reading

1. Computability Theory: An Introduction to Recursion Theory. Herbert B. Enderton, 2010.
2. Recursion. Jesse Russell and Ronald Cohn, 2012.
3. Programs, Recursion and Unbounded Choice (Cambridge Tracts in Theoretical Computer Science). Wim H. Hesselink, 2005.
4. Computability Theory: An Introduction to Recursion Theory, Students Solutions Manual (e-only). Herbert B. Enderton, 2011.

Answers To Self-Assessment Exercises

MODULE 1. PROGRAMMING LANGUAGES

Unit 1: Brief survey of programming paradigms

1. Programming
2. Program
3. Programming language
4. Data type
5. Simple data and combined types
6. Binary (1 and 0)
7. Particularly stressed by the way the language uses data structures.
8. Translation
9. Interpreter
10. Imperative; Functional; Object-Oriented; Declarative

Unit 2: Overview of programming languages and compilation process

1. Program in programming language
2. Object code
3. Executable code
4. *.cpp
5. *.obj
6. *.exe
7. Syntactic and logical
8. Algorithm and semantic errors
9. Functional decouplig
10. Separate file (module)

MODULE 2. FUNDAMENTAL PROGRAMMING CONSTRUCTS

Unit 1: Syntax and semantics of programming language C/C++

1. Identifiers; reserved keywords; signs of operations; constants; delimiters
2. Programmatic entity (constant, variable, mark, type, function, module, field in a structure)
3. int, float, char, struct, union, enum.
4. 2
5. 8
6. x = x*y; i = i+2; x = x/(y+15);
7. n=3, a=3
8. cout; cin
9. You can see on the screen:

   1

   0

   1 Now... the end

   teach C++

10. Example of program:

```
#include <iostream.h>
#include <stdio.h>
#include <math.h>
        void main(void)
{
double a=1.5 ,b=4.3 ,c=10 ,d=7.1 ,v;
double x=b*c+a,y=2*b+d;
double sx=sin(x);
if (sx<0) sx*=-1;
if (y<0)  y*=-1;
v=(a*sqrt(sx)-exp(-a*c))/sqrt(y);
cout << "Calculated value v is  " << v << endl;
        _getch();
        return 0;
}
```

Unit 2: Structures of simple programs

1. 68
2. Example of program:

```
#include < iostream.h>
#include <math.h>
#include <conio.h>
double a, b, d;
void main(void)
{
clrscr();
cin >>a >> b;
if (a*b < 0) d=log(pow(a*b,2));
if (a*b > 0) d=log(a*b);
if (a*b == 0) d=0;
cout << "Calculated value v is  " << d << endl;
        _getch();
        return 0;
}
```

3. Organized reiteration of some sequence of operators
4. Iteration

5. Goto

6. Break

7. Return

8. Example of program:

```
#include <stdio.h>
#include <math.h>
double y, a, x;
void main ()
{
a=10.2; x=5;
while (x<=35)
{
y=a * pow(x, 2);
cout <<Y<<X;
x++;
}
}
```

9. Variable which can contain the address of some object

10. Object which specifies on position of other variable.

Unit 3: Function

1. Function with parameters:

```
void print ( int x ) {
 cout << x << endl;
}
```

2. Two integer variables

3. Constants, variables, expressions

4. List of parameters

5. Yes

6. Constant pointer

7. Arguments which appear in a function definition

8. Arguments specified in a function

9. Example of program:

```
void Two (int & x)
{
        x=2;
        cout<<x<<endl;
}
void One ()
{
        int  y=1;
        Two (y);
        cout<<x<<endl;
}
```

10. Achieve the performance of pass-by-reference

MODULE 3. ALGORITHMS AND PROBLEM-SOLVING

Unit 1: Problem solving

1. Analyze the problem; Implement the algorithm; Maintenance
2. Outline the problem and its requirements; Design steps to solve the problem
3. Implement the algorithm in code; Verify that the algorithm works
4. Use and modify the program if the problem domain changes
5. Involves the spontaneous contribution of ideas from all members of the group
6. Brainstorming
7. Filtering
8. Usually just return the contents of one of an object's variables; action responsibilities are a little more complicated, often involving calculations

Unit 2: Basic concept of an algorithm

1. Well-defined computational procedure consisting of a set of instructions
2. Program
3. Exact
4. Algorithm must contain a finite number of steps in its execution
5. An algorithm must provide the correct answer to the problem.
6. Solve every instance of the problem
7. Normally a mixture of English statements, some mathematical notations, and selected keywords from a programming language
8. Sequence of statements place one after the other

Unit 3: Search and sorting algorithms

1. The elements are found, satisfying to inequalities $X1 \leq a[i] \leq X2$ where values for X1 and X2 are set will be organized.

2. (N2-N)/2
3. Binary insert
4. N*log2 (N)
5. K = [log2N] +1
6. Sequential, Sorting, Binary search

MODULE 4. FUNDAMENTAL DATA STRUCTURES

Unit 1: Primitive data types

1. Encapsulation
2. Status and behavior
3. Status
4. Behavior
5. 0
6. Example of program:

```
#include <iostream.h>
int main()
{
   const int n=10;
   double a[n];
   double sum=0;        double pr=1;
   cout << "Enter elements..." << endl;
        {
          cout << endl << "a[" << i << "]: ";
          cin >> a[i];
              }
        for (int i=0; i<n; i++)
    if (a[i])>=0) sum=sum+a[i];
        else pr=pr*a[i];
cout<< "summa =" << sum<<endl;
cout<< "product=" << pr<<endl;
return 0;
}
```

7. Matrix (a_{ij}, i=1,.., m; j=1,.., n), i - line number, j - number of column
8. Pointer
9. int *y[3]={x0,x1,x2}
10. int **m

Unit 2: Strings

1. Zero
2. “programming language C++”
3. length
4. find
5. rfind
6. empty()
7. max_size()

Unit 3. Help allocation

1. Component object constrained data of different type
2. Example of structure:

```
struct Stud_type {
    char Number[10];
    char Fio[40];
double S_b;
};
struct Stud_type {
```

3. Linear structure of data
4. Set of data, placed on an external transmitter
5. Sequence of data, the structure of which is determined programmatic
6. d:\\work\\Sved.cpp
7. f = fopen ("d:\\work\\Dat_sp.cpp", "w");
8. Run-time type identification
9. Run-time type information
10. typeinfo

MODULE 5. MACHINE ORGANIZATION

Unit 1. Machine levels organization

1. Address and Content
2. Data , Address, Control buses
3. 8
4. 1024
5. short sequences of instructions as inline
6. One-pass and multi-pass assemblers
7. Macros
8. Main Memory (RAM)
9. Central processor
10. Read Only Memory (ROM)

Unit 2. Assembly language programming

1. Fetch instruction
2. Decode instruction
3. Processor execute the instruction
4. Binary patterns 0 and 1
5. Assembler
6. Hardware level
7. High level programming

MODULE 6. SOFTWARE DEVELOPMENT METHODOLOGY

Unit 1: Fundamental design and concept principles

1. Pre-definition of specific deliverables
2. Software development methodology (SDM)
3. Waterfall model
4. Software prototyping
5. Incremental development
6. Spiral model
7. Rapid application development

Unit 2: Testing and debugging strategies

1. Initiation, planning, execution and closure
2. Review the system requirements specification document
3. Conceptual design
4. Detailed design
5. Readiness for testing of software
6. Release to manufacturing
7. Unit, integration, validation, system testing
8. Brute force, Backtracking, Cause elimination
9. ISO/IEC 12207.

MODULE 7. RECURSION

Unit 1: Concept of recursion

1. Process of repeating items in a self-similar way
2. Base case
3. Solution for and a return value
4. Passes the simpler problem back into the function
5. Factorial
6. Easier to code a recursive solution once one is able to identify that solution.

Unit 2: Simple recursive procedures

1. Towers of Hanoi, Russian Dolls, Tail Recursion
2. cout<<p; print(p-1);
3. Example:

```
{
if (n==0)
   return 0;
if (n==1)
   return 1;
return( Fibonacci(n-2) + Fibonacci(n-1) );
}
```

4. Example:

```
void printnum ( int begin )
{
 cout<< begin;
 if ( begin < 9 )        {
    printnum ( begin + 1 );
 }
 cout<< begin;
}
```