Principles of Programming Languages

BLOCK -1

Programming Languages-1

Block 1 Programming Languages-1

The first block of this course deals with programming language-1. As we know programming language is the most important part of the computer science you should learn basics about the programming languages. For this reason, we create this block in to the four following units.

In the first unit we discussed about the programming languages. We also focused the importance of programming languages in computer science. As we know the importance of the programming languages, we should also know about the history of it. From the starting era to present how many languages has been introduced and what features they have.

In the second unit we focused on the language translator. In this unit we introduce the attributes of good programming language. We also focused on the simplicity of the programming language, in this unit we told what parameters makes a language simple and robust. We also discussed about the language translator. With the help of language translator you learn about how source code translate into executable code.

In the third unit, it has one of the most useful concepts for characterizing various features of programming languages called binding and binding time. This is also a part of "recurring themes" of computing, because this concepts also play important roles in operating systems, databases, and software engineering. For this reason we focused on language definitions, language implementation, program translation etc. For provide the depth in programming language we introduce the concept of data objects, types of data objects, variable, constants, .We also told about the binding and binding time, Elementary and structured data types and their specifications.

In this last unit we introduce the representations and Implementation of numbers. As we all know the numbers representation is very important in computer science. For this concept we discussed how the numbers like integers and floating points represent in computer and how they implement.

This block has examples to easily understand the topics. These topics are divided in to sections and sub-sections. This block also provide figures and tables for your convenient.

Hope you like this study material and we wish you best of luck.

UNIT-I Programming Languages Fundamental

Structure

1.0 Introduction

This is the first unit of this block of this course. This unit focused on programming languages. There are six sections in this unit. In the section Sec. 1.1 we introduce about the programming language introduction. In Sec. 1.2 you will learn about the importance of programming languages. Next section Sec. 1.3 told you about the brief history of programming languages. In Sec. 1.4 you will know about the features of programming languages. In Sec. 1.5 and 1.6 you will find summary and review questions respectively.

This unit covers very important topics related to programming languages. As you know programming languages play a very important role in computer science this unit covers a lot of important topics about programming languages.

Objective

After studying this unit you should be able to:

- Define the introduction of Programming Language.
- Define the Importance of programming languages.
- Explain history of programming languages.
- Express important features of programming languages.

1.1 Programming language Introduction

A programming language is a set of commands, instructions, and other syntax use to create a software program. Languages that programmers use to write code are called "high-level languages." This code can be compiled into a "low-level language," which is recognized directly by the computer hardware.

High-level languages are designed to be easy to read and understand. This allows programmers to write source code in a natural fashion, using logical words and symbols. For example, reserved words like function, while, if, and else are used in most major programming languages. Symbols like <, >, ==, and != are common operators. Many high-level languages are similar enough that programmers can easily understand source code written in multiple languages.

Examples of high-level languages include C++, Java, Perl, and PHP. Languages like C++ and Java are called "compiled languages" since the source code must first be compiled in order to run. Languages like Perl and PHP are called "interpreted languages" since the source code can be run through an interpreter without being compiled. Generally, compiled languages are used to create software applications, while interpreted languages are used for running scripts, such as those used to generate content for dynamic websites.

Low-level languages include assembly and machine languages. An assembly language contains a list of basic instructions and is much more difficult to read than a high-level language. In rare cases, a programmer may decide to code a basic program in an assembly language to ensure it operates as efficiently as possible. An assembler can be used to translate the assembly code into machine code. The machine code, or machine language, contains a series of binary codes that are understood directly by a computer's CPU. Needless to say, machine language is not designed to be human readable.

1.2 Importance of programming languages

As technology evolves, as a driving force in business, in the past, companies were managing by people and processes. These two factors that have led business performance and to determine whether the organization will be successful in achieving important goals. Thus, the need and importance of programming was arisen.

The programming language is artificial language designed to express computations can be performed on the machine, including a computer. Programming languages created several programs that control the behavior of the machine, to express algorithms precisely, or as any form of human communication are assigned.

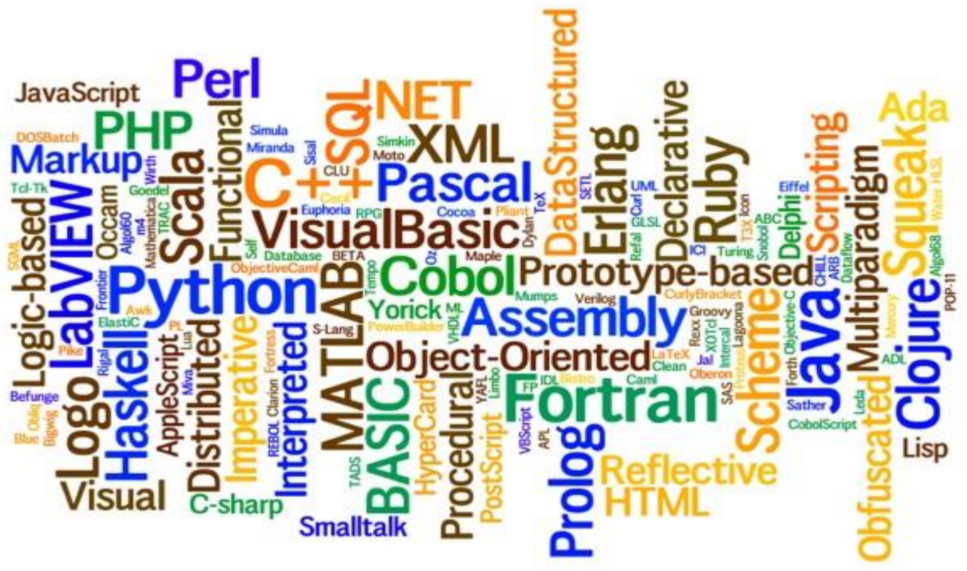


Figure 1.0

The theory of programming languages is a branch of computer science deals with the design, implementation, analysis, characterization and classification of programming languages and their individual characteristics. The programming language is a discipline of computer science, both within and affecting mathematics, software engineering and linguistics. This is a well-known branch of computer science, and the area of active research, with results published in several magazines devoted to PLT, and computer science in general and technical publications. Most undergraduate programs require computer courses in the subject line.

The role of the programming language was dropped in favor of the methodology and software tools, but it is completely null and void if it is claimed that a well-designed system can also be used in any language. But programming languages are not only a tool, but also provides the raw material of software that we see on our screens at the same day.

Programming languages exist only to fill the gap of abstraction between the hardware and the real world. It is inevitable tension between abstraction above that it is easier to understand and safer to use, and to a lesser degree of abstraction that are more flexible and can be more effective. For the development or selection of the programming language to choose the appropriate level of abstraction, and it is not surprising that the developer prefers to different levels, or that one language may not be suitable for the project, but not for another. In particular, the language, the programmer needs to understand in depth the consequences of the safety and efficacy of each structure in the language.

There are a variety of programming languages, such as .NET, Assembly, Basic, C, C#, C++, Delphi, Java, JavaScript, Pascal, Perl, PHP, Python, Ruby, VB-Script, Visual Basic and other languages and technologies. So start learning programming and be a successful programmer.

1.3 Brief history of Programming Languages

Programming languages are often spoken of in terms of their level of abstraction. To this end there is a somewhat official classification system. In said system, each generation in the hierarchy represents another level of abstraction away from the machine hardware.

First-generation programming language (1GL) – Binary

It makes sense Watson would say this seeing as how the earliest computers were programmed entirely in binary. These computers were programmed with no abstraction at all. I, for one, do not envy our forefathers in regard to this task. While the programs were small, all operations, data and memory had to be managed by hand in binary.

- Introduced in the 1940s
- Instructions/Data entered directly in binary
- Memory must be manually moved around
- Very difficult to edit/debug
- Simple programs only

Examples:

Architecture specific binary delivered on Switches, Patch Panels and/or Tape.

Second-generation programming language (2GL) – Assembly

Assembly languages were introduced to mitigate the error prone and excessively difficult nature of binary programming. While still used today for embedded systems and optimization, they have mostly been supplanted by 3GL languages due to the difficulties in controlling program flow.

- Introduced in the 1950s
- Written by a programmer in an intermediate instruction language which is later compiled into binary instructions
- Specific to platform architecture
- Designed to support logical structure, debugging
- Defined by three language elements: Opcodes (CPU Instructions), Data Sections (Variable Definitions) and Directive (Macros)

Examples:

Almost every CPU architecture has a companion assembly language. Most commonly in use today are RISC, CISC and x86 as that is what our embedded systems and desktop computers use.

Third-generation programming language (3GL) – Modern

Third generation languages are the primary languages used in general purpose programming today. They each vary quite widely in terms of their particular abstractions and syntax. However, they all share great enhancements in logical structure over assembly language.

- Introduced in the 1950s
- Designed around ease of use for the programmer
- Driven by desire for reduction in bugs, increases in code reuse
- Based on natural language
- Often designed with structured programming in mind

Examples:

Most Modern General Purpose Languages such as C, C++, C#, Java, Basic, COBOL, Lisp and ML

Fourth-generation programming language (4GL) – Application Specific

"A programming language is low level when its programs require attention to the irrelevant."

-Alan J. Perlis

A fourth generation language is designed with making problems in a specific domain simple to implement. This has the advantage of greatly reducing development time cost. At the same time there is the disadvantage of increasing developer learning cost.

- Introduced in the 1970s, Term coined by Jim Martin
- Driven by the need to enhance developer productivity
- Further from the machine
- Closer to the domain

Some examples: SQL, SAS, R, MATLAB's GUIDE, ColdFusion, CSS

Fifth-generation programming language (5GL) – Constraint Oriented

It has been argued that there is no such thing as a 5GL language. This seems to me ridiculous as working with domain specific syntax is hardly an abstraction dead end. This cynicism is likely a result of the many false claims of 5GL for the sake of marketing.

Many researchers speak of 5GL languages as constraint systems. The programmer inputs a set of logical constraints, with no specified algorithm, and the AI-based compiler builds the program based on these constraints.

- Introduced in the 1990s
- Constraint-based instead of algorithmic
- Used for AI Research, Proof solving, Logical Inference
- Not in common use

Some examples: Prolog, Mercury

Check your progress

Q1. What do you understand by programming language?

Q2. Define importance of programming languages.

1.4 Features of Programming Languages

Comparing programming languages is a lot like comparing cars. There are a lot of different features you can measure and a lot of disagreement about what factors are important, but for the most part, the features that matter can't be measured. Here is my list in no particular order:

Responsiveness

You know how, in Excel, when you change a number in a cell, all the other cells are updated immediately. That's responsiveness. The more steps between changing code and seeing the result of the change, the harder it becomes to stay in the programming flow.

The big winner in this area is Smalltalk with its live image. But other languages with powerful REPLs like Lisp and Scheme come in close second.

OCaml, Haskell, Ruby, and Python all have REPLs, but any Lisper will tell you that they are just not the same. Not that they aren't useful because you could be using any of the other languages where the best case scenario is fast compile times so you can get through your edit-compile-run cycle.

The reason a good programming language needs to be responsive is so that you can be productive. And it's not just a matter of counting the wasted time waiting for compiles (although it can be significant), it's the destruction of your flow. When your changes are instant, you keep making changes and your mind settles into the contours of the problem. When you have even 1 minute breaks between changes, you lose track of what you were doing, and you lose focus, eventually spending the afternoon reading reddit.

Safety

Have you ever heard an OCaml or Haskell programmer claim that "once it compiles, it just works"? It's a bit of an exaggeration, but not as much as you might think. Often your compile errors are pointing out real issues you need to address for the correctness of your code, and sometimes they point out issues in your whole approach that will cause you to rewrite sections of your code *without ever having run the broken version*.

That is safety. It’s the feeling that your programming language is watching your back. Haskell is a clear winner on the safety front. Its type system is powerful enough to specify some surprisingly sophisticated constraints. OCaml is a close second, and it falls off dramatically after that.

I should also point out that dynamic typing has a significant disadvantage when it comes to safety because you need to run the program to find the error, but since they usually do a pretty good job of trapping the error at runtime in ways that are easy to debug, they have an advantage over languages without any runtime support such as C or C++.

Speed

Speed appears to be the only criterion on some peoples list of features. But just because it’s over-valued doesn’t mean it’s not important. The claims of which language is fastest is hotly contested, I will just say that some languages, like C, C++, Scheme, and OCaml pay close attention to runtime performance, and some (like Ruby) do not.

Speed is not required for every application, and indeed premature optimization is the root of all evil, but sometimes it has to be fast Period.

Most languages let you call C code as the escape valve for performance, but in today’s computers speed comes from optimized memory layouts and cache efficiency, so to keep that C code fast, the data set needs to be already in memory in an efficient structure, which means that you will end up writing more and more C code to expose that data to your higher level language, and in the process importing most of the C resource management headaches as well. Languages that give you explicit control over memory layout or have seamless integration with C (like C++ or objective-C) have a clear performance advantage.

There is also a class of languages like OCaml and many Scheme and Lisp implementations that have very good performance most of the time, so a trip to C is rare.

And the final note on speed is that often the best way to speed up code is to improve the algorithm. The advantage of some of the slower high-level languages is that implementing the sophisticated algorithms is much easier.

Completeness

We all need libraries. Do you want to start your next project writing XML parsers and DB integrations, or do you just want to find the right library and get started? That’s what I thought.

We try out a lot of languages, and this is where most of the otherwise great languages fall flat. This is a chicken and egg problem because how many libraries there are for your language is mostly a function of how many users you have, and you don’t attract a lot of programmers when they need to write everything themselves from scratch.

The best way to have instant access to lots of libraries is to have seamless C integration, since almost anything you will ever need was written in (or for) C. Or you could just grow a giant user-base like Perl or Python.

Expressiveness

Expressiveness is the ability to reshape the language until you can express your program naturally. Some languages like Lisp and Scheme let you implement a new internal language for describing your program concisely. Smalltalk, Ruby and Haskell have basic building blocks and lightweight syntax that makes it easy to define new language constructs for your particular problem.

But it’s not all about making embedded mini-languages, that’s just one very effective version of expressiveness. It’s also how well the language supports you with features that let you remove boilerplate code and just write the code that is needed to do the job.

It’s a hard quality to pin down, but in many ways it’s the most important because code is meant to be read by humans first and the computer second. Each line of code is a liability; it will need to be maintained. If it’s not doing anything for you but telling the compiler things it could have figured out for itself, then it’s a line wasted.

More than anything, this is the quality that attracts programmers and rewards them for their effort in learning the language. That’s the reason that Ruby meta-programming took off after Rails hit it big: when people saw what could be done, how you could write working Ruby that read like a pseudo-code description of the program, they became drunk on the possibilities it opened up. Writing code in a highly expressive language is *fun*.

Summing it up

I don’t think that programs can be lined up on a scale from best to worst, but they can be judged on specific criteria, and the five qualities of responsiveness, safety, speed, completeness, and expressiveness cover all the important aspects of a programming language.

It may seem like a bit of an academic question, but without saying what you are looking for, you can’t possibly improve. And like most programmers I’m always looking to the future.

1.5 Summary

In this unit you learnt about basics of programming language, importance of programming languages, history of programming language and features of the programming language.

- A programming language is a set of commands, instructions, and other syntax use to create a software program.
- C++ and Java are called "compiled languages" since the source code must first be compiled in order to run.
- Compiled languages are used to create software applications, while interpreted languages are used for running scripts
- The programming language is artificial language designed to express computations can be performed on the machine, including a computer.
- Assembly languages were introduced to mitigate the error prone and excessively difficult nature of binary programming.

1.6 Review Questions

Q1. What are programming languages? Why we use programming languages? Define with example?

Q2. What is the importance of programming languages? Explain in detail with example.

Q3. Write brief history of programming languages.

Q4. What are the features of programming languages? Explain in detail with example.

Q5. List at least 10 programming language with description.

UNIT-II Language Translator

Structure

2.0 Introduction

This is the second unit of this course containing the complete description about language translator. As you know about language translator that a translator is a program that completes the translation of a program written in any given programming language into a functionally corresponding program in a different computer language, without losing the functional or logical structure of the original code (the "essence" of each program). In this unit there are four sections. In Sec. 2.1 we define attributes of good programming languages. In Sec 2.2 you will learn about the introduction of language translator. In Sec. 2.3 and 2.4 you will find summary and review questions respectively.

Objectives

After studying this unit you should be able to:

- Define attributes of good programming language
- Describe the introduction to language translator
- Define assembler, compiler, and interpreter.

2.1 Attributes of Programming Languages

Till now there are many high level languages which are very popular, and there are others, which could not become so popular in-spite of being very powerful. There might be many reasons for the success of a language, but one obvious reason is the characteristics of the language. Several characteristics believed to be important with respect to making a programming language good are briefly discussed below.

Simplicity

A good programming language must be simple and easy to learn and use. For example, BASIC is liked by many programmers only because of its simplicity. Thus, a good programming language should provide a programmer with a clear, simple and unified set of concepts which can be easily grasped. It is also easy to develop and implement a compiler or an interpreter for a programming language that is simple. However, the power needed for the language should not be sacrificed for simplicity. The overall simplicity of a programming language strongly affects the readability of the programs written in that language, and programs that are easier to read and understand are also easier to maintain.

Naturalness

A good language should be natural for the application area it has been designed. That is, it should provide appropriate operators, data structures, control structures, and a natural syntax in order to facilitate the users to code their problem easily and efficiently. FORTRAN and COBOL are good examples of scientific and business languages respectively that possess high degree of naturalness.

Abstraction

Abstraction means the ability to define and then use complicated structures or operations in ways that allow many of the details to be ignored. The degree of abstraction allowed by a programming language directly affects its variability. For example, object-oriented languages support high degree of abstraction. Hence writing programs in object-oriented languages is much easier. Object-oriented languages also support reusability of program segments due to its feature.

Efficiency

The program written in good programming language are efficiently translated into machine code, are efficiently executed, and acquires as little space in the memory as possible. That is, a good programming language is supported with a good language translator (a compiler or an interpreter} that gives due consideration to space and time efficiency.

Structured ness

Structured ness means that the language should have necessary features to allow its users to write their programs based on the concepts of structured programming. This property of a language greatly affects the ease with which a program may be written, tested, and maintained. Moreover, it forces a programmer to look at a problem in a logical way so that fewer errors are created while writing program for the problem.

Compactness

In a good programming language, programmers should be able to express intended operation concisely. A verbose language can tax the programmer's sheer writing stamina and thus reduce its usefulness. COBOL is generally not liked by many programmers because it is verbose in nature and compactness.

Locality

A good programming language should be such that while writing a program, a programmer need not jump around visually as the text of the program is prepared. This allows the programmer to concentrate almost solely on the part of the program around the statements currently being worked with. COBOL lacks locality because data definitions are separated from processing statements, perhaps by many pages of code.

Extensibility

A good programming language should allow extension through simple, natural, and elegant mechanisms. Almost all languages provide subprogram definition mechanisms for this purpose, but there are some languages that are rather weak in this aspect.

Suitability to its Environment

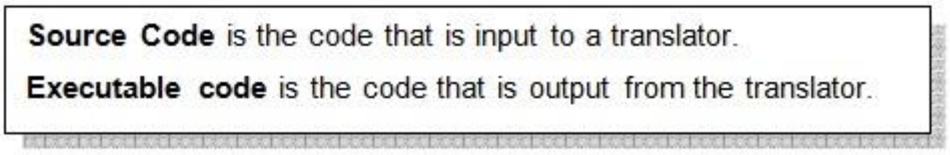
Depending upon the type of application for which a programming language has been designed, the language must also be made suitable to its environment. For example, a language designed for real time applications must be interactive in nature. On the other hand, languages used for data processing jobs like pay-roll, stores accounting, etc., may be designed to be operative in batch mode.

2.2 Introduction to language translator

Language translators convert programming source code into language that the computer processor understands. Programming source code has various structures and commands, but computer processors only understand machine language. Different types of translations must occur to turn programming source code into machine language, which is made up of bits of binary data. The three major types of language translators are assemblers, compilers and interpreters.

There are 3 types of system software used for translating the code that a programmer writes into a form that the computer can execute (i.e. machine code). These are:

1. Assemblers
2. Compilers
3. Interpreters


Source Code is the code that is input to a translator.

Executable code is the code that is output from the translator.

Figure 2-1

Assembler

An Assembler converts an assembly program into machine code.

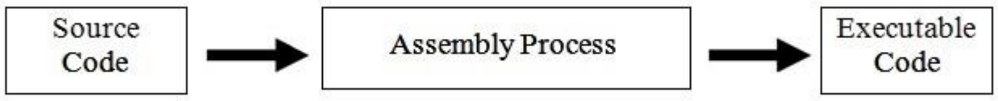


Figure 2-2

Compiler

A Compiler is a program that translates a high level language into machine code. The Turbo Pascal compiler, for example, translates a program written in Pascal into machine code that can be run on a PC.

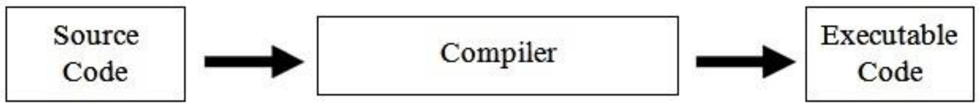


Figure 2-3

Advantages of a Compiler

1. Fast in execution
2. The object/executable code produced by a compiler can be distributed or executed without having to have the compiler present.
3. The object program can be used whenever required without the need to of re-compilation.

Disadvantages of a Compiler

1. Debugging a program is much harder. Therefore not so good at finding errors
2. When an error is found, the whole program has to be re-compiled

Interpreter

An Interpreter is also a program that translates high-level source code into executable code. However the difference between a compiler and an interpreter is that **an interpreter translates one line at a time and then executes it**: no object code is produced, and so the program has to be interpreted each time it is to be run. If the program performs a section code 1000 times, then the section is translated into machine code 1000 times since each line is interpreted and then executed.

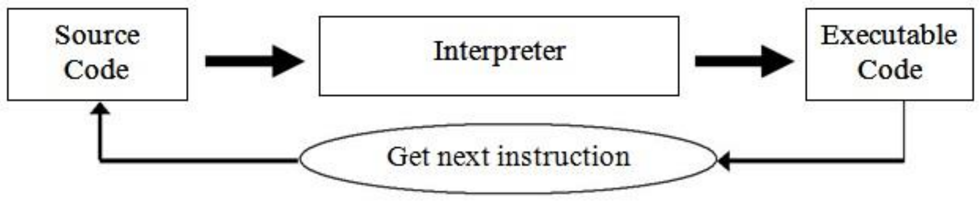


Figure 2-4

Advantages of an Interpreter

1. Good at locating errors in programs
2. Debugging is easier since the interpreter stops when it encounters an error.
3. If an error is deducted there is no need to retranslate the whole program

Disadvantages of an Interpreter

1. Rather slow
2. No object code is produced, so a translation has to be done every time the program is running.
3. For the program to run, the Interpreter must be present

Check your progress

Q1. Explain attributes of programming languages.

Q2. Explain the following

(i) Assembler

(ii) Compiler

(iii)Interpreter

2.3 Summary

In this unit you learnt about different attribute of programming language and a brief introduction about the language translator.

- A good programming language must be simple and easy to learn and use.
- A good programming language should provide a programmer with a clear, simple and unified set of concepts which can be easily grasped.
- A good language should be natural for the application area it has been designed.
- Language translators convert programming source code into language that the computer processor understands.
- A Compiler is a program that translates a high level language into machine code.

2.4 Review Questions

Q1. Define the attribute of the programming languages in details.

Q2. What is language translator? Define its concept in detail.

Q3. What do you mean by linker and loader?

Q4. What is the working of interpreter?

Q5. Write short note on Assembler, Compiler and Interpreter.

UNIT-III Data Types (Elementary and Structured)

Structure

3.0 Introduction

This unit focused on the topic of elementary and structured data types. There are five sections in this unit. In the Sec. 3.1 you will learn about binding and binding time. In the next section Sec 3.2 you will know about elementary and structured data types. As you know elementary data types and data objects in a local program are exclusively defined and declared by reference to known data types and data objects. Elementary local data types in a program make your programs easier to read and understand. If you have used such a type to define several data objects, you can change the type of all of those objects in one place, just be changing the definition in the TYPES statement. In the same manner structured data types largely known as Record, struct or Structure in most programming languages, it is the GeneXus object which allows defining complex data structures. An SDT represents data whose structure is made up of several elements like a Customer struct. The SDT makes it easy to transfer parameters (more specifically, they allow providing/using structured information when using web services), it simplifies XML automatic reading and writing and makes it possible to manage variable-length lists of elements. In Sec. 3.3 you will know about the specifications. In Sec. 3.4 and 3.5 you will find summary and review questions respectively.

Objectives

After studying this unit you should be able to:

- Define Binding and binding time.
- Elementary and structured data types
- Specifications of data types.

3.1 Binding and Binding Times

One of the most useful concepts for characterizing various features of programming languages is the notion of binding and binding time. This is also one of the "recurring themes" of computing, because these concepts also play important roles in operating systems, databases, and software engineering.

A *binding* in a program is an association of an attribute with a program component such as an identifier or a symbol. For example the data type of the value of a variable is an attribute that is associated with the variable name. The *binding time* for an attribute is the time at which the binding occurs. For example, in C the binding time for a variable type is when the program is compiled (because the type cannot be changed without changing and recompiling the program), but the value of the variable is not bound until the program executes (that is, the value of the variable can change during execution).

Some additional examples of attributes are:

- the meaning of a keyword such as **if**
- the operation associated with a symbol such as +
- the entity (variable, keyword, etc.) represented by an identifier
- the memory location for the value of an identifier

The most common binding times for attributes are (in chronological order):

1. Language definition
2. Language implementation
3. Program translation (compile time)
4. Link edit
5. Load
6. Program execution (run time)

Some attributes and their binding times for C are shown in the table below:

Attribute	**Binding time**
Type of a variable identifier	Translation
Value of a variable identifier	Execution
General meaning of +	Language definition
Specific meaning of +	Translation

Meaning of literal (constant) 23	Language definition
Internal representation of literal 23	Language implementation
Specific computation performed by an external function	Link edit
Value of a "global" data identifier	Execution

Some explanations of these are:

> **Meaning of +:** The general meaning of + was defined (as numerical addition) when the language was defined. The specific addition operation (float, double, int, etc.), however, cannot be determined until program translation time, when it becomes bound to the + symbol used in the operation.
>
> **Literal 23:** The meaning of literal 23 as the int value representing the number 23 is fixed at language definition time. However, the actual representation (sequence of bits) for the int value 23 can vary on different platforms, and it is not bound until the language implementation (compiler) is performed.
>
> Computation performed by an external function: A function in another program file (i.e., an external function) can come from a library or from another compilation. The specific function that is linked to the function call is not determined until the program modules are combined by the linkage editor, and in fact different computations could be done by using different external modules in different link edits.

The fundamental idea behind the notion of binding and binding times for programming languages is this: if you look at an identifier or symbol in a program, at what time can you say exactly what each of its attributes is? If you see the identifier "if" or the literal 23 in a C program, for example, you know that "if" is a keyword that begins an alternation statement, and 23 represents the int number 23, because these things were determined when the language was defined and cannot be changed. But if you see the identifier "sum" in a program, you have to have more context (i.e., a translation must be done) to determine whether sum is a variable or a function name, and what the type is for sum.

Note that we already have seen one important difference between C and ML that can be expressed in terms of binding time: the value of a nonlocal and non-parameter data identifier ("global"). In C this value can vary at execution time if the value associated with the global identifier changes, but in ML the value associated with the global identifier becomes fixed when the function is defined (translation time) (which provides referential transparency).

It is tempting to say that the type of an identifier is bound at translation time in C but at execution time in ML, but this is not really the case. One problem is that the distinction between translation and execution becomes blurred in ML. Another is that the tendency is to think of a val function in ML as analogous to an assignment operation in C, but it actually is more like a C declaration than an assignment operation. The type of the data value associated with an identifier becomes fixed in ML when the appropriate Val expression is translated (and executed), so there really is not much difference between C and ML here. Smalltalk is an example of a language in which type binding does not occur until execution time.

One additional note: we are sometimes tempted to say that a certain binding comes when a program is written. However, there really is no reason to make this further refinement, because there are no significant differences between program-writing time and compile time that are of interest to us.

Note that there is a tradeoff, as usual, when we consider early binding versus late binding in terms of advantages and disadvantages. Early binding facilitates efficiency of translation and program execution because fewer decisions have to be made at translation and execution time. However, the later the binding the more flexibility that there is for the programmer and the executing program, although more flexibility doesn't necessarily lead to more efficient programming, it can often help if it is not misused.

3.2 Elementary and structure data types

A ***type*** is a category of entities. Programming languages support some elementary data types. These data types are close to the real world entities and hence make it easy for the programmer to focus on the transformation to be done to produce the desires set of output. The compiler takes care of the conversion of the data type into machine recognizable form and vice-versa.

A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.

Types are useful in:

1. Error detection

2. Safety

3. Design

4. Abstraction

5. Verification

6. Software evolution

7. Documentation

All programs specify a set of operations that are to be applied to certain data in a certain sequence.

Data Objects

The storage areas of an actual computer, such as the memory, registers, and external media, usually have a relatively simple structure as sequences of bits grouped into bytes or words. However, data storage of the virtual computer for a programming language tends to have a more complex organization, with arrays, stacks, numbers, character strings, and other forms of data existing at different points during execution of a program. It is convenient to use the term ***data object*** to refer to a run time grouping of one or more pieces of data in a virtual computer. During execution of program different data objects of different data types exist.

Types of data objects

1. Programmer defined

2. System defined

A ***data object*** represents a container for ***data values*** – a place where data values may be stored and later retrieved. A data object is characterized by a set of attributes, the most important of which is its data type. The attributes determine the number and type of values that the data object may contain and also determine the logical organization of those values.

For eg: a) **A**: $\boxed{\qquad\qquad\qquad}$

Data object: A location in memory

With name A

b) **10001**

Data value: A bit pattern used by translator whenever the number 17 is used in the program

c) **A**: $\boxed{\mathbf{00000010001}}$

Bound variable: Data object bound to data value 17

A data object participates in various bindings during its lifetime. The most important attributes and bindings are as follows:

1) Type

2) Location

3) Value

4) Name

5) Component

Variables and Constants

In computer programming, a **variable** is a symbolic name given to some known or unknown quantity or information, for the purpose of allowing the name to be used independently of the information it represents. A variable name in computer source code is usually associated with a data storage location and thus also its contents and these may change during the course of program execution.

Variables in programming may not directly correspond to the concept of variables in mathematics. The value of a computing variable is not necessarily part of an equation or formula as in mathematics. In computing, a variable may be employed in a repetitive process: assigned a value in one place, then used elsewhere, then reassigned a new value and used again in the same way.

Variables in computer programming are frequently given long names to make them relatively descriptive of their use, whereas variables in mathematics often have terse, one- or two-character names for brevity in transcription and manipulation.

A variable has three essential attributes: a symbolic name (also known as an identifier), a data location (generally in storage or memory, identified by address and length), and a value, represented by the data contents of that location. These attributes are often assigned at separate times during the program execution. In some programming languages, non-identical identifiers can simultaneously refer to the same variable, location and value.

Unlike their mathematical counterparts, programming variables and constants commonly take multiple-character names, e.g. COST or total. Single-character names are most commonly used only for auxiliary variables; for instance, i, j, k for array index variables.

For eg. in C language , the declaration is,

int N; it declares a simple data object N of type integer.

& N= 23 may be used to assign the data value23 to N.

Constants

In computer programming, a **constant** is a special kind of variable whose value cannot typically be altered by the program during its execution.

Although a constant's value is specified only once, a constant may be referenced many times in a program. Using a constant instead of specifying a value multiple times in the program can not only simplify code maintenance, but it can also supply a meaningful name for it and consolidate such constant bindings to a standard code location (for example, at the beginning).

For eg in C language

const float pi2 = 3.1416;

Naming conventions for constant variables vary. Some simply name them as they would any other variable. Others use capitals and underscores for constants in a way similar to their traditional use for symbolic macros, such as SOME_CONSTANT.

Check your progress

Q1. Define Binding and Binding Times.

Q2. Explain Data Object. Also define types of data object.

3.3 Specification of elementary data types

The basic elements of a specification of a data type are:

1) The attributes

2) The values

3) The operations

Attributes: Basic attributes of any data object, such as data type and name, are usually invariant during its life time. Some of the attributes may be stored in a descriptor as part of the data object during program execution.

Values: The type of data object determines the set of possible values that it may contain. For example, the integer data type determines a set of integer values that may serve as the values for data objects of this type. C defines 4 classes of integer types: int, short, long and char.

Operations: The set of operations defined for a data type determine how data objects of that type may be manipulated. The operations may be primitive operations, which mean they are specified as part of the language definition, or they may be programmer-defined operations, in the form of subprograms or method declarations as part of class definitions.

Structured data objects

A data object that is constructed as an aggregate of other data objects, called components, is termed a structured data object or data structure. A component may be elementary or it may be another data structure.

Structured data

Homogeneous: arrays, lists, sets

Non-homogeneous: records

Structured data types

A data structure is a data object that contains other data objects as its elements or components.

Specifications

· Number of components

Fixed size - Arrays

Variable size – stacks, lists. Pointer is used to link components.

- Type of each component

Homogeneous – all components are the same type

Heterogeneous – components are of different types

- Selection mechanism to identify components – index, pointer

Two-step process:

Referencing the structure

Selection of a particular component

- Maximum number of components
- Organization of the components: simple linear sequence
- Simple linear sequence
    - Multidimensional structures:
    - Separate types (FORTRAN)
    - Vector of vectors (C++)

Operations on data structures

- Component selection operations

    - Sequential
    - Random
- Insertion/deletion of components
- Whole-data structure operations
    - Creation/destruction of data structures

3.4 Summary

In this unit you learnt about Binding and binding time, Elementary and structured data types, and their Specifications.

- Binding and binding time, Elementary and structured data types, Specifications
- A *binding* in a program is an association of an attribute with a program component such as an identifier or a symbol.
- The *binding time* for an attribute is the time at which the binding occurs.
- A ***type*** is a category of entities.
- The term ***data object*** to refer to a run time grouping of one or more pieces of data in a virtual computer.

3.5 Review Questions

Q1. What do you understand by binding and binding time? Explain with example.

Q2. What is a *type*? Define in detail with example.

Q3. Define the term data object in detail.

Q4. What are structured data object and structured data type?

Q5. Define specification of elementary data types in depth with example.

UNIT-IV Representations and Implementation of numbers

Structure

4.0 Introduction

In this unit you will learn about Representations and Implementation of numbers. As you know number representation is very important in computer. In this unit there are four sections. In Sec. 4.1 you will learn about how we can represent a number in computer and how these numbers implements in computers. This section has detailed description about numbers. These numbers are integers, floating point's numbers. As you know the set {. . . , –3, –2, –1, 0, 1, 2, 3 . . .} of integers is also referred to as signed or directed (whole) numbers. The most straightforward representation of integers consists of attaching a sign bit to any desired representation of natural numbers, leading to signed magnitude representation. The standard convention is to use 0 for positive and 1 for negative and attach the sign bit to the left end of the magnitude. In this unit we covered detailed description of integer numbers. This unit also focused on floating point numbers. A fixed-point number consists of a whole or integral part and a fractional part, with the two parts separated by a radix point (decimal point in radix 10, binary point in radix 2, and so on). The position of the radix point is almost always implied and thus the point is not explicitly shown. In Sec. 4.2 and 4.3 you will find summary and review questions respectively.

Objectives

After studying this unit you should be able to:

- Define how to represent numbers in computers.
- Express how computer implements numbers.

4.1 Representations and Implementation of numbers.

Computer uses *a fixed number of bits* to represent a piece of data, which could be a number, a character, or others. A *n*-bit storage location can represent up to 2^*n* distinct entities. For example, a 3-bit memory location can hold one of these eight binary patterns: 000, 001, 010, 011, 100, 101, 110, or 111. Hence, it can represent at most 8 distinct entities. You could use them to represent numbers 0 to 7, numbers 8881 to 8888, characters 'A' to 'H', or up to 8 kinds of fruits like apple, orange, banana; or up to 8 kinds of animals like lion, tiger, etc.

Integers, for example, can be represented in 8-bit, 16-bit, 32-bit or 64-bit. You, as the programmer, choose an appropriate bit-length for your integers. Your choice will impose constraint on the range of integers that can be represented. Besides the bit-length, an integer can be represented in various *representation* schemes, e.g., unsigned vs. signed integers. An 8-bit unsigned integer has a range of 0 to 255, while an 8-bit signed integer has a range of -128 to 127 - both representing 256 distinct numbers.

It is important to note that a computer memory location merely *stores a binary pattern*. It is entirely up to you, as the programmer, to decide on how these patterns are to be *interpreted*. For example, the 8-bit binary pattern "0100 0001B" can be interpreted as an unsigned integer 65, or an ASCII character 'A', or some secret information known only to you. In other words, you have to first decide how to represent a piece of data in a binary pattern before the binary patterns make sense. The interpretation of binary pattern is called *data representation* or *encoding*. Furthermore, it is important that the data representation schemes are agreed-upon by all the parties, i.e., industrial standards need to be formulated and straightly followed.

Once you decided on the data representation scheme, certain constraints, in particular, the precision and range will be imposed. Hence, it is important to understand *data representation* to write *correct* and *high-performance* programs.

Rosette Stone and the Decipherment of Egyptian Hieroglyphs

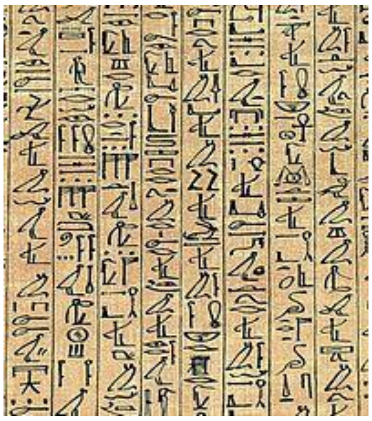


Figure 4-1

Egyptian hieroglyphs (next-to-left) were used by the ancient Egyptians since 4000BC. Unfortunately, since 500AD, no one could longer read the ancient Egyptian hieroglyphs, until the re-discovery of the Rosette Stone in 1799 by Napoleon's troop (during Napoleon's Egyptian invasion) near the town of Rashid (Rosetta) in the Nile Delta.

The Rosetta stone (left) is inscribed with a decree in 196BC on behalf of King Ptolemy V. The decree appears in *three* scripts: the upper text is *Ancient Egyptian hieroglyphs*, the middle portion Demotic script, and the lowest *Ancient Greek*. Because it presents essentially the same text in all three scripts, and Ancient Greek could still be understood, it provided the key to the decipherment of the Egyptian hieroglyphs.

The moral of the story is unless you know the encoding scheme; there is no way that you can decode the data.

Integer Representation

Integers are *whole numbers* or *fixed-point numbers* with the radix point *fixed* after the least-significant bit. They are contrast to *real numbers* or *floating-point numbers*, where the position of the radix point varies. It is important to take note that integers and floating-point numbers are treated differently in computers. They have different representation and are processed differently (e.g., floating-point numbers are processed in a so-called floating-point processor). Floating-point numbers will be discussed later.

Computers use *a fixed number of bits* to represent an integer. The commonly-used bit-lengths for integers are 8-bit, 16-bit, 32-bit or 64-bit. Besides bit-lengths, there are two representation schemes for integers:

1. *Unsigned Integers*: can represent zero and positive integers.
2. *Signed Integers*: can represent zero, positive and negative integers. Three representation schemes had been proposed for signed integers:
    a. Sign-Magnitude representation
    b. 1's Complement representation
    c. 2's Complement representation

You, as the programmer, need to decide on the bit-length and representation scheme for your integers, depending on your application's requirements. Suppose that you need a counter for counting a small quantity from 0 up to 200, you might choose the 8-bit unsigned integer scheme as there is no negative numbers involved.

n-bit Unsigned Integers

Unsigned integers can represent zero and positive integers, but not negative integers. The value of an unsigned integer is interpreted as "*the magnitude of its underlying binary pattern*".

Example 1: Suppose that *n*=8 and the binary pattern is 0100 0001B, the value of this unsigned integer is 1×2^0 + 1×2^6 = 65D.

Example 2: Suppose that *n*=16 and the binary pattern is 0001 0000 0000 1000B, the value of this unsigned integer is 1×2^3 + 1×2^12 = 4104D.

Example 3: Suppose that *n*=16 and the binary pattern is 0000 0000 0000 0000B, the value of this unsigned integer is 0.

An *n*-bit pattern can represent 2^*n* distinct integers. An *n*-bit unsigned integer can represent integers from 0 to (2^*n*)-1, as tabulated below:

N	Minimum	Maximum
8	0	(2^8)-1 (=255)
16	0	(2^16)-1 (=65,535)
32	0	(2^32)-1 (=4,294,967,295) (9+ digits)
64	0	(2^64)-1 (=18,446,744,073,709,551,615) (19+ digits)

Table 4-1

Signed Integers

Signed integers can represent zero, positive integers, as well as negative integers. Three representation schemes are available for signed integers:

1. Sign-Magnitude representation
2. 1's Complement representation
3. 2's Complement representation

In all the above three schemes, the *most-significant bit* (msb) is called the *sign bit*. The sign bit is used to represent the *sign* of the integer - with 0 for positive integers and 1 for negative integers. The *magnitude* of the integer, however, is interpreted differently in different schemes.

n-bit Sign Integers in Sign-Magnitude Representation

In sign-magnitude representation:

- The most-significant bit (msb) is the *sign bit*, with value of 0 representing positive integer and 1 representing negative integer.
- The remaining *n*-1 bits represents the magnitude (absolute value) of the integer. The absolute value of the integer is interpreted as "the magnitude of the (*n*-1)-bit binary pattern".

Example 1: Suppose that *n*=8 and the binary representation is 0 100 0001B.
Sign bit is 0 ⇒ positive
Absolute value is 100 0001B = 65D
Hence, the integer is +65D

Example 2: Suppose that *n*=8 and the binary representation is 1 000 0001B.
Sign bit is 1 ⇒ negative
Absolute value is 000 0001B = 1D
Hence, the integer is -1D

Example 3: Suppose that *n*=8 and the binary representation is 0 000 0000B.
Sign bit is 0 ⇒ positive
Absolute value is 000 0000B = 0D
Hence, the integer is +0D

Example 4: Suppose that *n*=8 and the binary representation is 1 000 0000B.
Sign bit is 1 ⇒ negative
Absolute value is 000 0000B = 0D
Hence, the integer is -0D

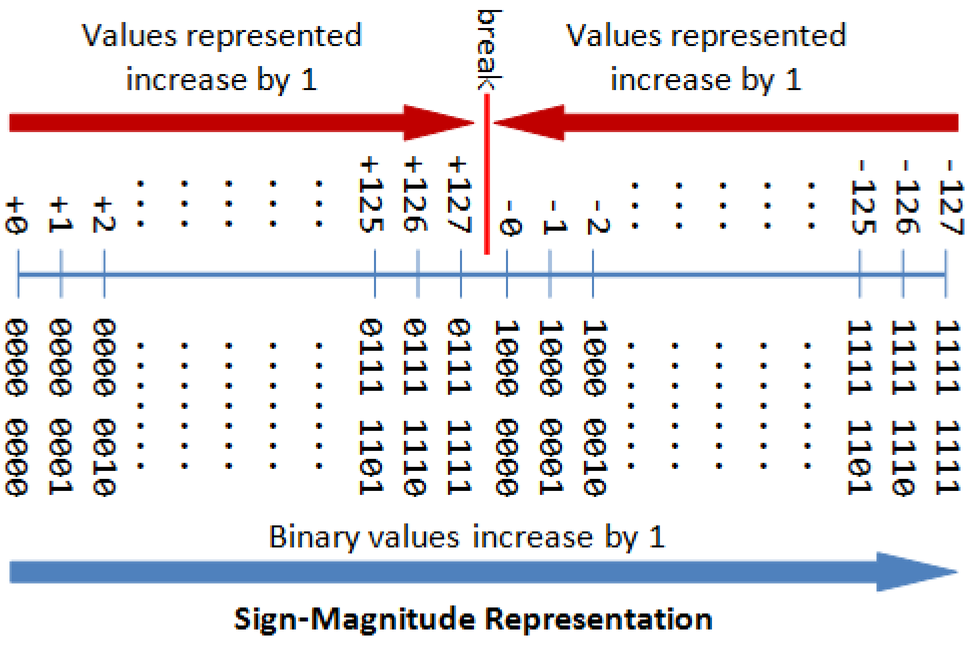


Figure 4-2

The drawbacks of sign-magnitude representation are:

1. There are two representations (0000 0000B and 1000 0000B) for the number zero, which could lead to inefficiency and confusion.
2. Positive and negative integers need to be processed separately.

n-bit Sign Integers in 1's Complement Representation

In 1's complement representation:

- Again, the most significant bit (msb) is the *sign bit*, with value of 0 representing positive integers and 1 representing negative integers.
- The remaining *n*-1 bits represents the magnitude of the integer, as follows:

- for positive integers, the absolute value of the integer is equal to "the magnitude of the (*n*-1)-bit binary pattern".
- for negative integers, the absolute value of the integer is equal to "the magnitude of the *complement* (*inverse*) of the (*n*-1)-bit binary pattern" (hence called 1's complement).

Example 1: Suppose that *n*=8 and the binary representation 0 100 0001B.
Sign bit is 0 ⇒ positive
Absolute value is 100 0001B = 65D
Hence, the integer is +65D

Example 2: Suppose that *n*=8 and the binary representation 1 000 0001B.
Sign bit is 1 ⇒ negative
Absolute value is the complement of 000 0001B, i.e., 111 1110B = 126D
Hence, the integer is -126D

Example 3: Suppose that *n*=8 and the binary representation 0 000 0000B.
Sign bit is 0 ⇒ positive
Absolute value is 000 0000B = 0D
Hence, the integer is +0D

Example 4: Suppose that *n*=8 and the binary representation 1 111 1111B.
Sign bit is 1 ⇒ negative
Absolute value is the complement of 111 1111B, i.e., 000 0000B = 0D
Hence, the integer is -0D

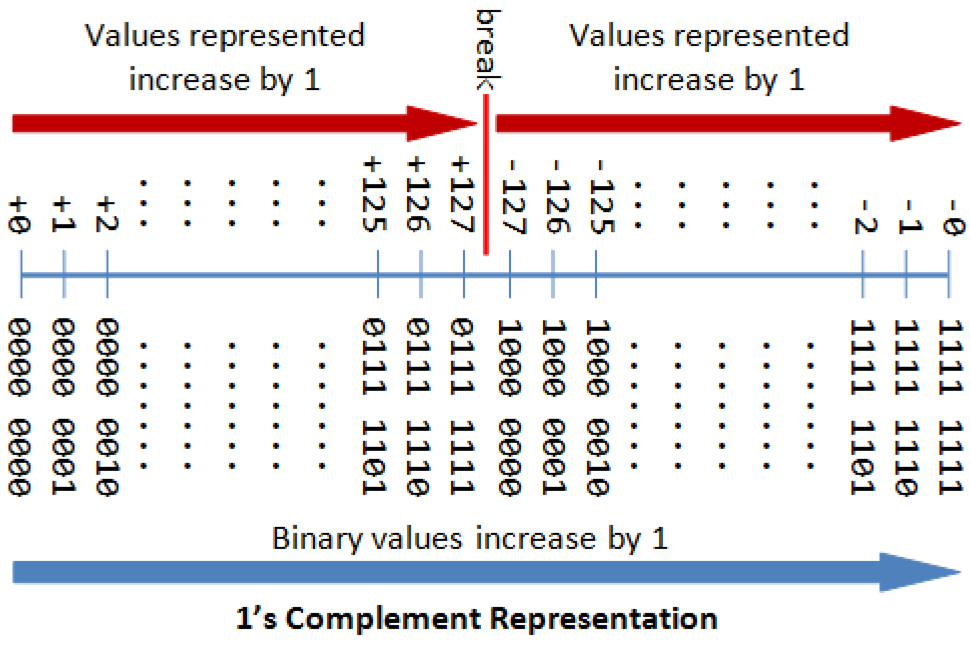


Figure 4-3

Again, the drawbacks are:

1. There are two representations (0000 0000B and 1111 1111B) for zero.

2. The positive integers and negative integers need to be processed separately.

n-bit Sign Integers in 2's Complement Representation

In 2's complement representation:

- Again, the most significant bit (msb) is the *sign bit*, with value of 0 representing positive integers and 1 representing negative integers.
- The remaining *n*-1 bits represents the magnitude of the integer, as follows:
  - For positive integers, the absolute value of the integer is equal to "the magnitude of the (*n*-1)-bit binary pattern".
  - For negative integers, the absolute value of the integer is equal to "the magnitude of the *complement* of the (*n*-1)-bit binary pattern *plus one*" (hence called 2's complement).

Example 1: Suppose that *n*=8 and the binary representation 0 100 0001B.
Sign bit is 0 ⇒ positive
Absolute value is 100 0001B = 65D
Hence, the integer is +65D

Example 2: Suppose that *n*=8 and the binary representation 1 000 0001B.
Sign bit is 1 ⇒ negative
Absolute value is the complement of 000 0001B plus 1, i.e., 111 1110B + 1B = 127D
Hence, the integer is -127D

Example 3: Suppose that *n*=8 and the binary representation 0 000 0000B.
Sign bit is 0 ⇒ positive
Absolute value is 000 0000B = 0D
Hence, the integer is +0D

Example 4: Suppose that *n*=8 and the binary representation 1 111 1111B.
Sign bit is 1 ⇒ negative
Absolute value is the complement of 111 1111B plus 1, i.e., 000 0000B + 1B = 1D
Hence, the integer is -1D

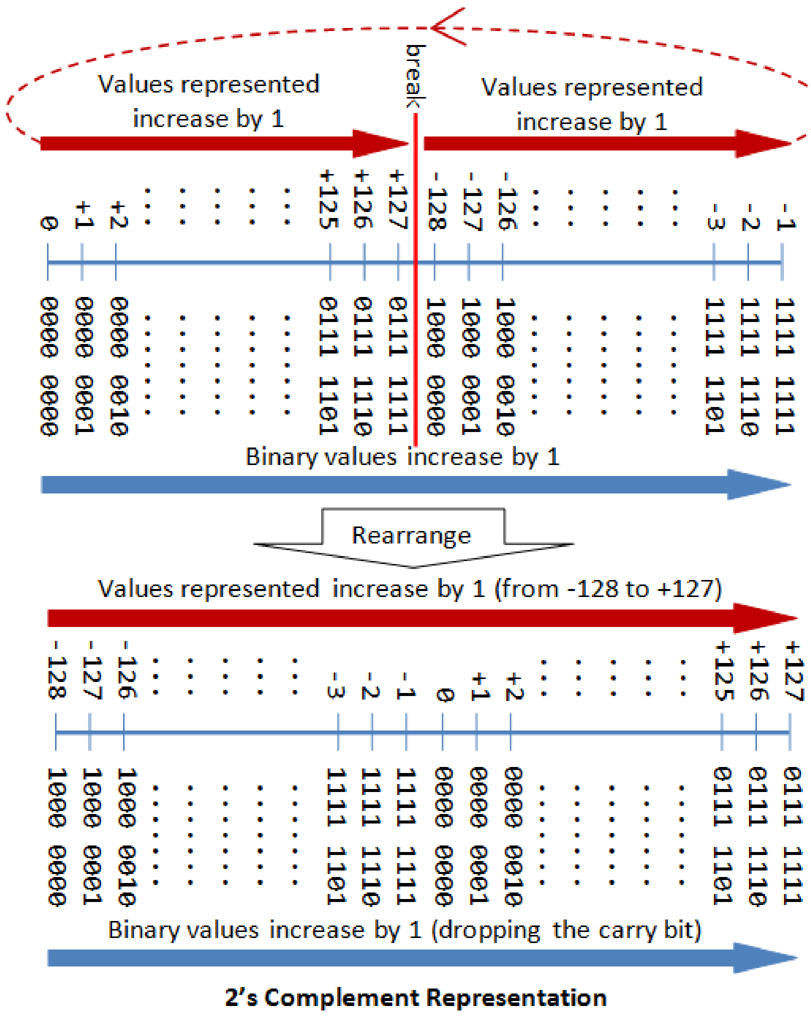


Figure 4-4

Computers use 2's Complement Representation for Signed Integers

We have discussed three representations for signed integers: signed-magnitude, 1's complement and 2's complement. Computers use 2's complement in representing signed integers. This is because:

1. There is only one representation for the number zero in 2's complement, instead of two representations in sign-magnitude and 1's complement.

2. Positive and negative integers can be treated together in addition and subtraction. Subtraction can be carried out using the "addition logic".

Example 1: Addition of Two Positive Integers: Suppose that n=8, 65D + 5D = 70D

65D → 0100 0001B

5D → 0000 0101B(+

0100 0110B → 70D (OK)

Example 2: Subtraction is treated as Addition of a Positive and a Negative Integers: Suppose that n=8, 5D - 5D = 65D + (-5D) = 60D

65D → 0100 0001B

-5D → 1111 1011B(+

0011 1100B → 60D (discard carry - OK)

Example 3: Addition of Two Negative Integers: Suppose that n=8, -65D - 5D = (-65D) + (-5D) = -70D

-65D → 1011 1111B

-5D → 1111 1011B(+

1011 1010B → -70D (discard carry - OK)

Because of the *fixed precision* (i.e., *fixed number of bits*), an *n*-bit 2's complement signed integer has a certain range. For example, for *n*=8, the range of 2's complement signed integers is -128 to +127. During addition (and subtraction), it is important to check whether the result exceeds this range, in other words, whether *overflow* or *underflow* has occurred.

Example 4: Overflow: Suppose that n=8, 127D + 2D = 129D (overflow - beyond the range)

127D → 0111 1111B

2D → 0000 0010B(+

1000 0001B → -127D (wrong)

Example 5: Underflow: Suppose that n=8, -125D - 5D = -130D (underflow - below the range)

-125D → 1000 0011B

-5D → 1111 1011B(+

0111 1110B → +126D (wrong)

The following diagram explains how the 2's complement works. By re-arranging the number line, values from -128 to +127 are represented contiguously by ignoring the carry bit.

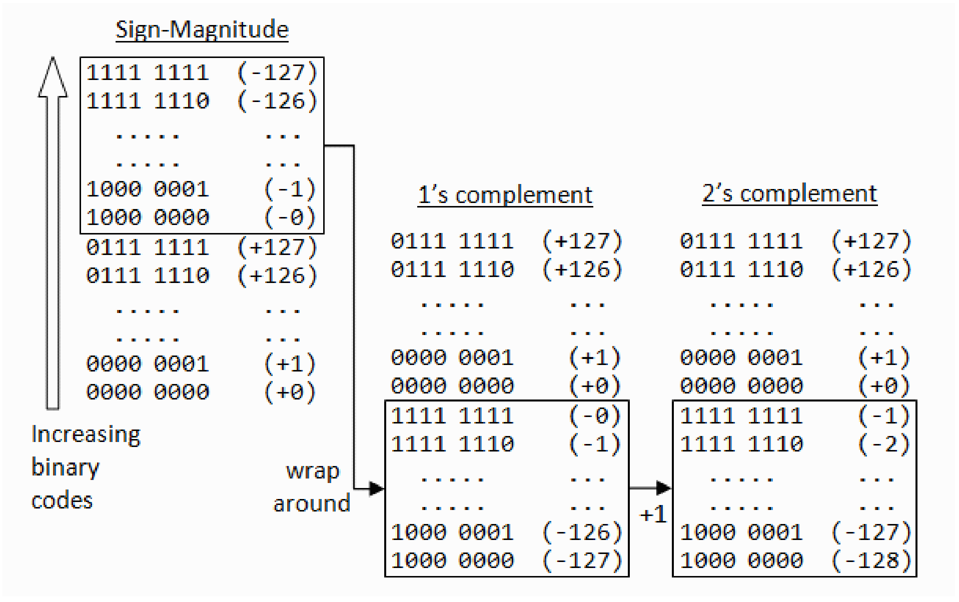


Figure 4-5

Range of n-bit 2's Complement Signed Integers

An *n*-bit 2's complement signed integer can represent integers from -2^(*n*-1) to +2^(*n*-1)-1, as tabulated. Take note that the scheme can represent all the integers within the range, without any gap. In other words, there is no missing integers within the supported range.

n	minimum	maximum
8	-(2^7) (=-128)	+(2^7)-1 (=+127)
16	-(2^15) (=-32,768)	+(2^15)-1 (=+32,767)
32	-(2^31) (=-2,147,483,648)	+(2^31)-1 (=+2,147,483,647)(9+ digits)
64	-(2^63) (=-9,223,372,036,854,775,808)	+(2^63)-1 (=+9,223,372,036,854,775,807)(18+ digits)

Table 4-2

Decoding 2's Complement Numbers

1. Check the *sign bit* (denoted as S).
2. If S=0, the number is positive and its absolute value is the binary value of the remaining *n*-1 bits.
3. If S=1, the number is negative. You could "invert the *n*-1 bits and plus 1" to get the absolute value of negative number. Alternatively, you could scan the remaining *n*-1 bits from the right

(least-significant bit). Look for the first occurrence of 1. Flip all the bits to the left of that first occurrence of 1. The flipped pattern gives the absolute value. For example,

4. n = 8, bit pattern = 1 100 0100B
5. S = 1 → negative
6. Scanning from the right and flip all the bits to the left of the first occurrence of 1 ⇒ 011 1100B = 60D

Hence, the value is -60D

Big Endian vs. Little Endian

Modern computers store one byte of data in each memory address or location, i.e., byte addressable memory. An 32-bit integer is, therefore, stored in 4 memory addresses.

The term"Endian" refers to the *order* of storing bytes in computer memory. In "Big Endian" scheme, the most significant byte is stored first in the lowest memory address (or big in first), while "Little Endian" stores the least significant bytes in the lowest memory address.

For example, the 32-bit integer 12345678H (2215053170_{10}) is stored as 12H 34H 56H 78H in big endian; and 78H 56H 34H 12H in little endian. An 16-bit integer 00H 01H is interpreted as 0001H in big endian, and 0100H as little endian.

Floating-Point Number Representation

A floating-point number (or real number) can represent a very large (1.23×10^88) or a very small (1.23×10^-88) value. It could also represent very large negative number (-1.23×10^88) and very small negative number (-1.23×10^88), as well as zero, as illustrated:

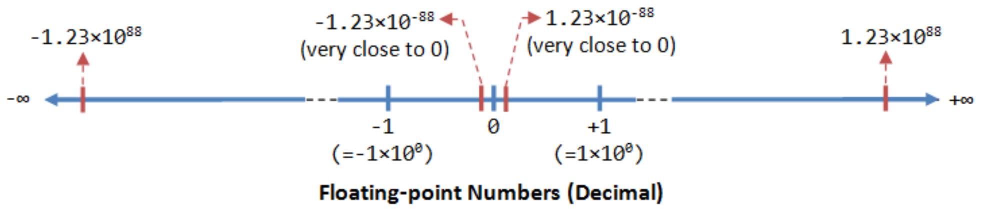


Figure 4-6

A floating-point number is typically expressed in the scientific notation, with a *fraction* (F), and an *exponent* (E) of a certain *radix* (r), in the form of F×r^E. Decimal numbers use radix of 10 (F×10^E); while binary numbers use radix of 2 (F×2^E).

Representation of floating point number is not unique. For example, the number 55.66 can be represented as 5.566×10^1, 0.5566×10^2, 0.05566×10^3, and so on. The fractional part can be *normalized*. In the normalized form, there is only a single non-zero digit before the radix point. For example, decimal number 123.4567 can be normalized as 1.234567×10^2; binary number 1010.1011B can be normalized as 1.011011B×2^3.

It is important to note that floating-point numbers suffer from *loss of precision* when represented with a fixed number of bits (e.g., 32-bit or 64-bit). This is because there are *infinite* number of real numbers (even within a small range of says 0.0 to 0.1). On the other hand, a *n*-bit binary pattern can represent a *finite* 2^*n* distinct numbers. Hence, not all the real numbers can be represented. The nearest approximation will be used instead, resulted in loss of accuracy.

It is also important to note that floating number arithmetic is very much less efficient than integer arithmetic. It could be speed up with a so-called dedicated *floating-point co-processor*. Hence, use integers if your application does not require floating-point numbers.

In computers, floating-point numbers are represented in scientific notation of *fraction* (F) and *exponent* (E) with a *radix* of 2, in the form of F×2^E. Both E and F can be positive as well as negative. Modern computers adopt IEEE 754 standard for representing floating-point numbers. There are two representation schemes: 32-bit single-precision and 64-bit double-precision.

IEEE-754 32-bit Single-Precision Floating-Point Numbers

In 32-bit single-precision floating-point representation:

- The most significant bit is the *sign bit* (S), with 0 for negative numbers and 1 for positive numbers.
- The following 8 bits represent *exponent* (E).
- The remaining 23 bits represents *fraction* (F).

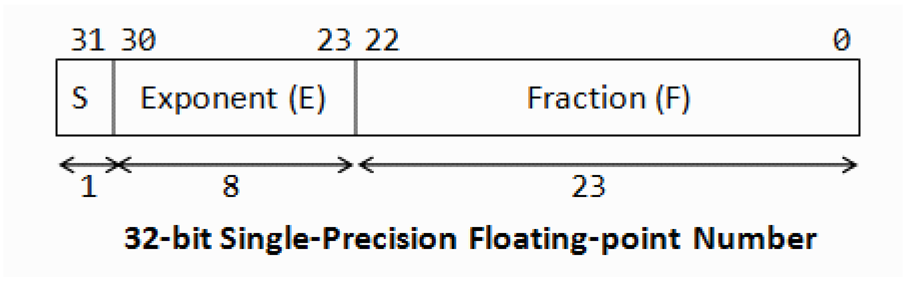


Figure 4-7

Normalized Form

Let's illustrate with an example, suppose that the 32-bit pattern is <u>1</u> <u>1000 0001</u> <u>011 0000 0000 0000 0000 0000</u>, with:

- S = 1
- E = 1000 0001
- F = 011 0000 0000 0000 0000 0000

In the *normalized form*, the actual fraction is normalized with an implicit leading 1 in the form of 1.F. In this example, the actual fraction is 1.011 0000 0000 0000 0000 0000 = 1 + 1×2^-2 + 1×2^-3 = 1.375D.

The sign bit represents the sign of the number, with S=0 for positive and S=1 for negative number. In this example with S=1, this is a negative number, i.e., -1.375D.

In normalized form, the actual exponent is E-127 (so-called excess-127 or bias-127). This is because we need to represent both positive and negative exponent. With an 8-bit E, ranging from 0 to 255, the excess-127 scheme could provide actual exponent of -127 to 128. In this example, E-127=129-127=2D.

Hence, the number represented is -1.375×2^2=-5.5D.

De-Normalized Form

Normalized form has a serious problem, with an implicit leading 1 for the fraction, it cannot represent the number zero! Convince yourself on this!

De-normalized form was devised to represent zero and other numbers.

For E=0, the numbers are in the de-normalized form. An implicit leading 0 (instead of 1) is used for the fraction; and the actual exponent is always -126. Hence, the number zero can be represented with E=0and F=0 (because 0.0×2^-126=0).

We can also represent very small positive and negative numbers in de-normalized form with E=0. For example, if S=1, E=0, and F=011 0000 0000 0000 0000 0000. The actual fraction is0.011=1×2^-2+1×2^-3=0.375D. Since S=1, it is a negative number. With E=0, the actual exponent is -126. Hence the number is -0.375×2^-126 = -4.4×10^-39, which is an extremely small negative number (close to zero).

Example 1: Suppose that IEEE-754 32-bit floating-point representation pattern is <u>0</u> <u>10000000</u> <u>110 0000 0000 0000 0000 0000</u>.

Sign bit S = 0 ⇒ positive number

E = 1000 0000B = 128D (in normalized form)

Fraction is 1.11B (with an implicit leading 1) = 1 + 1×2^-1 + 1×2^-2 = 1.75D

The number is +1.75 × 2^(128-127) = +3.5D

Example 2: Suppose that IEEE-754 32-bit floating-point representation pattern is <u>1</u> <u>01111110</u> <u>100 0000 0000 0000 0000 0000</u>.

Sign bit S = 1 ⇒ negative number

E = 0111 1110B = 126D (in normalized form)

Fraction is 1.1B (with an implicit leading 1) = 1 + 2^-1 = 1.5D

The number is -1.5 × 2^ (126-127) = -0.75D

Example 3: Suppose that IEEE-754 32-bit floating-point representation pattern is 1 01111110 000 0000 0000 0000 0000 0001.

Sign bit S = 1 ⇒ negative number

E = 0111 1110B = 126D (in normalized form)

Fraction is 1.000 0000 0000 0000 0000 0001B (with an implicit leading 1) = 1 + 2^-23

The number is - (1 + 2^-23) × 2^ (126-127) = -0.500000059604644775390625 (may not be exact in decimal!)

Example 4 (De-Normalized Form): Suppose that IEEE-754 32-bit floating-point representation pattern is 1 00000000 000 0000 0000 0000 0000 0001.

Sign bit S = 1 ⇒ negative number

E = 0 (in de-normalized form)

Fraction is 0.000 0000 0000 0000 0000 0001B (with an implicit leading 0) = 1×2^-23

The number is -2^-23 × 2^ (-126) = -2× (-149) ≈ -1.4×10^-45

IEEE-754 64-bit Double-Precision Floating-Point Numbers

The representation scheme for 64-bit double-precision is similar to the 32-bit single-precision:

- The most significant bit is the *sign bit* (S), with 0 for negative numbers and 1 for positive numbers.
- The following 11 bits represent *exponent* (E).
- The remaining 52 bits represents *fraction* (F).

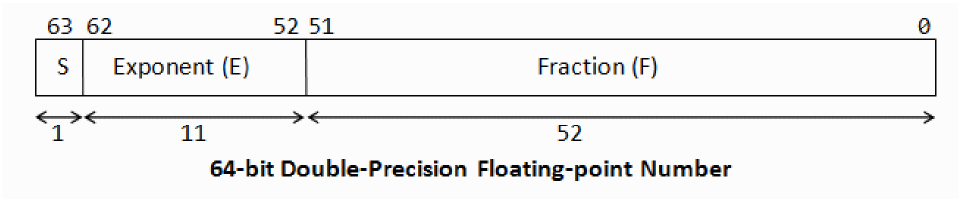


Figure 4-8

The value (N) is calculated as follows:

- Normalized form: For 1 ≤ E ≤ 2046, N = (-1) ^S × 1.F × 2^ (E-1023).

- Denormalized form: For E = 0, N = (-1) ^S × 0.F × 2^ (-1022). These are in the denormalized form.
- For E = 2047, N represents special values, such as ±INF (infinity), NaN (not a number).

More on Floating-Point Representation

There are three parts in the floating-point representation:

- The *sign bit* (S) is self-explanatory (0 for positive numbers and 1 for negative numbers).
- For the *exponent* (E), a so-called *bias* (or *excess*) is applied so as to represent both positive and negative exponent. The bias is set at half of the range. For single precision with an 8-bit exponent, the bias is 127 (or excess-127). For double precision with a 11-bit exponent, the bias is 1023 (or excess-1023).
- The *fraction* (F) (also called the *mantissa* or *significand*) is composed of an implicit leading bit (before the radix point) and the fractional bits (after the radix point). The leading bit for normalized numbers is 1; while the leading bit for denormalized numbers is 0.

Normalized Floating-Point Numbers

In normalized form, the radix point is placed after the first non-zero digit, e,g., 9.8765D×10^-23D, 1.001011B×2^11B. For binary number, the leading bit is always 1, and need not be represented explicitly - this saves 1 bit of storage.

In IEEE 754's normalized form:

- For single-precision, 1 ≤ E ≤ 254 with excess of 127. Hence, the actual exponent is from -126 to +127. Negative exponents are used to represent small numbers (< 1.0); while positive exponents are used to represent large numbers (> 1.0).

N = (-1) ^S × 1.F × 2^ (E-127)

- For double-precision, 1 ≤ E ≤ 2046 with excess of 1023. The actual exponent is from -1022 to +1023, and
  N = (-1)^S × 1.F × 2^(E-1023)

Take note that n-bit pattern has a *finite* number of combinations (=2^n), which could represent *finite* distinct numbers. It is not possible to represent the *infinite* numbers in the real axis (even a small range says 0.0 to 1.0 has infinite numbers). That is, not all floating-point numbers can be accurately represented. Instead, the closest approximation is used, which leads to *loss of accuracy*.

The *minimum* and *maximum* normalized floating-point numbers are:

Precision	Normalized N(min)	Normalized N(max)
Single	0080 0000H 0 00000001 00000000000000000000000B E = 1, F = 0 N(min) = 1.0B × 2^-126 (≈1.17549435 × 10^-38)	7F7F FFFFH 0 11111110 00000000000000000000000B E = 254, F = 0 N(max) = 1.1...1B × 2^127 = (2 - 2^-23) × 2^127 (≈3.4028235 × 10^38)
Double	0010 0000 0000 0000H N(min) = 1.0B × 2^-1022 (≈2.2250738585072014 × 10^-308)	7FEF FFFF FFFF FFFFH N(max) = 1.1...1B × 2^1023 = (2 - 2^-52) × 2^1023 (≈1.7976931348623157 × 10^308)

Table 4-3

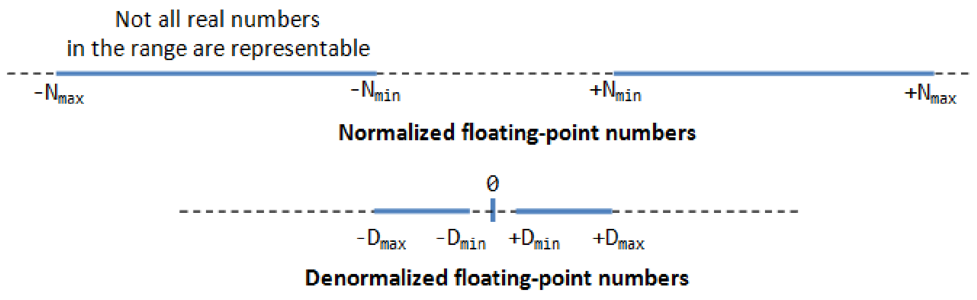

Not all real numbers
in the range are representable
-Nmax
-Nmin
+Nmin
+Nmax
Normalized floating-point numbers
0
-Dmax
-Dmin
+Dmin
+Dmax
Denormalized floating-point numbers

Figure 4-9

Denormalized Floating-Point Numbers

If E = 0, but the fraction is non-zero, then the value is in denormalized form, and a leading bit of 0 is assumed, as follows:

- For single-precision, E = 0,
  N = (-1)^S × 0.F × 2^(-126)
- For double-precision, E = 0,
  N = (-1)^S × 0.F × 2^(-1022)

Denormalized form can represent very small numbers closed to zero, and zero, which cannot be represented in normalized form, as shown in the above figure.

Precision	Denormalized D(min)	Denormalized D(max)
Single	0000 0001H 0 00000000 00000000000000000000001B E = 0, F = 00000000000000000000001B D(min) = 0.0...1 × 2^-126 = 1 × 2^-23 × 2^-126 = 2^-149 (≈1.4 × 10^-45)	007F FFFFH 0 00000000 11111111111111111111111B E = 0, F = 11111111111111111111111B D(max) = 0.1...1 × 2^-126 = (1-2^-23)×2^-126 (≈1.1754942 × 10^-38)
Double	0000 0000 0000 0001H D(min) = 0.0...1 × 2^-1022 = 1 × 2^-52 × 2^-1022 = 2^-1074 (≈4.9 × 10^-324)	001F FFFF FFFF FFFFH D(max) = 0.1...1 × 2^-1022 = (1-2^-52)×2^-1022 (≈4.4501477170144023 × 10^-308)

Table 4-4

The minimum and maximum of *denormalized floating-point numbers* are:

Special Values

Zero: Zero cannot be represented in the normalized form, and must be represented in denormalized form with E=0 and F=0. There are two representations for zero: +0 with S=0 and -0 with S=1.

Infinity: The value of +infinity (e.g., 1/0) and -infinity (e.g., -1/0) are represented with an exponent of all 1's (E = 255 for single-precision and E = 2047 for double-precision), F=0, and S=0 (for +INF) andS=1 (for -INF).

Not a Number (NaN): NaN denotes a value that cannot be represented as real number (e.g. 0/0). NaN is represented with Exponent of all 1's (E = 255 for single-precision and E = 2047 for double-precision) and any non-zero fraction.

Check your progress

Q1. How numbers represent in computer?

Q2. Explain 1’s and 2’s complements.

Q3. How fractional numbers represents in computer?

4.2 Summary

In this unit you learnt about how number represent and implement in computers. Both concepts are very important in computer Science.

- Computer uses *a fixed number of bits* to represent a piece of data, which could be a number, a character, or others.
- Integers, for example, can be represented in 8-bit, 16-bit, 32-bit or 64-bit
- A computer memory location merely *stores a binary pattern.*
- The interpretation of binary pattern is called *data representation* or *encoding*.
- Signed integers can represent zero, positive integers, as well as negative integers.

4.3 Review Questions

Q1. What is data representation? How data represent in computers? Explain with example.

Q2. Define the concept of implementation of data in computer memory. Elaborate your answer.

Q3. Differentiate between signed and unsigned integer with example.

Q4. What are 1’s and 2’s complement? Explain in detail with example.

Q5. What is normalized float-point numbers? Explain with example.

BIBLIOGRAPHY

Friedman, Daniel P., and Mitchell Wand. 2008. *Essentials of Programming Languages*. 3rd ed. Cambridge, MA: The MIT Press.

One of the better introductory programming language texts. EOPL walks through a series of interpreters written in Scheme.

Pierce, Benjamin C. 2002. *Types and Programming Languages*. Cambridge, Mass: The MIT Press.

Pierce, Benjamin C., ed. 2004. *Advanced Topics in Types and Programming Languages*. Cambridge, Mass: The MIT Press.

B. C. Pierce, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjoberg, and B. Yorgey. 2015. *Software Foundations*.

C. A. R. Hoare. An axiomatic basis for computer programming. Communications of the ACM, 12(10):576-580 and 583, October 1969.

Peter J. Landin. The next 700 programming languages. Communications of the ACM, 9(3):157-166, March 1966.

Robin Milner. A theory of type polymorphism in programming. Journal of Computer and System Sciences, 17:348-375, August 1978.

Gordon Plotkin. Call-by-name, call-by-value, and the λ-calculus. Theoretical Computer Science, 1:125-159, 1975.

John C. Reynolds. Towards a theory of type structure. In Colloque sur la Programmation, Paris, France, volume 19 of Lecture Notes in Computer Science, pages 408-425. Springer-Verlag, 1974.

Luis Damas and Robin Milner. Principal type schemes for functional programs. In *ACM Symposium on Principles of Programming Languages (POPL), Albuquerque, New Mexico*, pages 207-212, 1982.

Edsger W. Dijkstra. Recursive programming. In Saul Rosen, editor, *Programming Systems and Languages*, chapter 3C, pages 221-227. McGraw-Hill, New York, 1960.

Edsger W. Dijkstra. Go to statement considered harmful. 11(3):147-148, March 1968.

William A. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479-490. Academic Press, 1980. Reprint of 1969 article.

Robert Kowalski. Predicate logic as programming language. In *IFIP Congress*, pages 569-574, 1974. Reprinted in Computers for Artificial Intelligence Applications, (eds. Wah, B. and Li, G.-J.), IEEE Computer Society Press, Los Angeles, 1986, pp. 68-73.

Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308-320, January 1964.

BLOCK -2

Programming Languages-2

BLOCK 2 Programming Languages-2

This is the second block on programming languages-2. In this block we divided the programming language part two in to the four following units.

In the first unit we discussed about the vectors, accessing Elements of a Vector and Array Dimensions. We also described records, operations of Records, assignment and comparison of Records. We also discussed about representation in memory and how we use character strings as Arrays, variable size data structure. We also focused in structured data types, Specification of Data Structure Types, Sets and its Implementation, Bit-string representation of sets, Hash-coded representation of sets.

In the second unit we introduce the concept of files and how input files works. We also introduce how a file can be create, open, read or write and close. This unit also deals with the most important concept of object oriented programming's encapsulation and information hiding. As we know the importance of programming language we focused on the concept of sub programs, its basic definition, parameters, procedures and functions, design issues for sub programs, local referencing environment. We also define the concept of parameter passing methods, generic subprograms, and design issue for functions.

In the third unit we focused on co-routines, type definition and abstract data types. In this unit we also focused on handling problems of programming languages. We also described properties of abstract data types and its importance, generic abstract data types & notation. Implicit and explicit sequence control in programming languages.

In the last unit we told about subprogram sequence control, Recursive sub-programs. We also told about exception and exception handlers. We introduce the concept of co-routines and scheduled subprograms.

This block has some important examples with figures and tables.

Hope you will like it and we wish you all the best.

UNIT-I Variable Size data structure

Structure

1.0 Introduction

This is the first unit of this block. This unit focused on the variable size data structure. In this unit there are eight sections. Each section play a very important role. In Sec. 1.1 you will learn about vectors. As you have studied earlier about vectors and have some knowledge about vectors. In this unit vector is correlated to arrays which is defined in Sec. 1.2. Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type. In the Sec. 1.3 you will learn about records. In the next Sec.1.4 you will know about character string. In Sec. 1.5 we defined variable size data structure and in Sec. 1.6 we explain sets. In Sec. 1.7 and 1.8 you will find summary and review questions respectively.

Objectives

After studying this unit you should be able to:

- Define Vectors, Arrays, Records
- Express character string, Variable size data structure and Sets

1.1 Vectors

Vector is a container class from the C++ standard library. As is true for an array, it can hold objects of various types. Vector will also resize, shrink or grow, as elements are added. The standard library provides access to vectors via iterators, or subscripting. Iterators are classes that are abstractions of pointers. They provide access to vectors using pointer-like syntax but have other useful methods as well. The use of vectors and iterators is preferred to arrays and pointers. Common bugs involving accessing past the bounds of an array are avoided. Additionally, the C++ standard library includes generic algorithms that can be applied to vectors and to other container classes.

In C++, vectors are one type of dynamically-allocated container from the standard template library (STL). They are capable of storing multiple elements of any defined data type, and do so in a contiguous block of memory. Unlike raw arrays, vectors are access-safe as long as you use the built-in access methods associated with them. Thus, overrunning the end of a vector will throw an exception rather than crashing the program. For quickly writing access-safe code that requires very little in the way of programmer intervention for memory management, they are hard to beat. There are reasons to use other containers from the standard template library (such as lists or double-ended queues), but vectors are generally the first choice of most programmers for their versatility and speed.

Accessing Elements of a Vector

There are a number of ways to access the elements of a vector. For the moment, I will focus on two of them, one safe and one unsafe. And as a reminder, C++ vectors (and other STL containers), like raw C/C++ arrays, are accessed with indices starting at zero. This means that the first element is at position 0 in the vector, and the last element is at position (number of elements)-1.

1.2 Arrays

Arrays are a data structure that is used to store a group of objects of the same type sequentially in memory. They are a built-in part of the C++ language. All the elements of an array must be the same data type, for example float, char, int, pointer to float, pointer to int, a class, structure or function. Functions provide a way to define a new operation. They are used to calculate a result or update parameters. The elements of an array are stored sequentially in memory. This allows convenient and powerful manipulation of array elements using pointers. The use of arrays is common in C and C++ coding and is important to understand well.

Arrays in C++ are zero based. Suppose an array named myExample contains N elements. This array is indexed from 0 to (N-1). The first element of myExample is at index 0 and is accessed as myExample [0]. The second element is at index 1 and is accessed as myExample [1]. The last element is at index (N-1) and is accessed as myExample [N-1]. As a concrete example, suppose N equals 5 and that myExample will store integer data.

Array Dimensions

The C++ language performs no error checking on array bounds, which is one reason to use vectors rather than arrays. If you define an array with 50 elements and you attempt to access index 50 (the 51st element), or any out of bounds index, the compiler issues no warnings. It is the programmer's task alone to check that all attempts to access or write to arrays are done only at valid array indexes. Writing or reading past the end of arrays is a common programming bug and can be hard to isolate.

1.3 Records

In the computer science and engineering, records are the simplest form of data structure. Sometimes it is called as tuples, structs, or compound data. We can say that a record is a value that contains other values, usually in fixed number and sequence and usually indexed by names. The elements of records are usually called fields or members.

For instance, a date can be stored as a record, containing a numeric year field, a month field represented as a string, and a numeric day-of-month field. As another example, a student record might contain a name, an id, and a marks. As yet another example, we can say a Circle record might contain a center and a radius. In this example, the center itself might be represented as a Point record containing x and y coordinates.

Records are different from array by the fact that their number of fields is typically fixed, each field has a name, and that each field may have a different type.

A record type is a data type that describes such values and variables. Most modern computer languages allow the programmer to describe new record types. The definition includes specifying the data type of each field and an identifier (called name or label) by which it can be accessed. In type theory, product types (without field names) are generally preferred due to their simplicity, but proper record types are studied in languages such as System F-Sub. Since type-theoretical records may contain first-class function-typed fields in addition to data, they can express many features of OOPs (Object Oriented Programming Language).

Records can occur in any storage medium, including main memory and mass storage devices such as magnetic tapes or hard disks. Records are a fundamental element of most data structures, especially linked data structures. Many computer records are organized as arrays of logical records, often grouped into larger physical records or chunks for efficiency.

The parameters of a function or procedure or module can often be viewed as the fields of a record variable; and the arguments passed to that function can be seen as a record value that gets allocated to that variable at the time of the call. Also, in the call stack that is often used to implement procedure calls or function calls, each entry is an activation record or call frame, containing the procedure parameters and local variables, the return address, and other internal fields.

An object in object-oriented programming language is essentially a record that contains modules specialized to handle that record; and object types are an elaboration of record types. Indeed, in most object-oriented languages, records are just special cases of objects, and are known as plain old data structures (PODSs), to contrast with objects that use Object Oriented features.

A record can be seen as the computer analog of a mathematical tuple. In the same manner, a record type can be viewed as the computer language analog of the Cartesian product of two or more mathematical sets, or the implementation of an abstract product type in a specific language.

Operations of Records

A programming language that does supports record types usually provides some or all of the following operations:

- Declaration of a new record type, including the location, type, and (if possibly) name of each field;
- Declaration of variables and values as requiring a given record type;
- Construction of a new record value from given field values with given field names;
- Selection of a field of record with an explicit name;
- Assignment of a record value to a record variable name;
- Comparison of two records for equality;
- Computation of a standard hash value for the record.

Some languages may provide services that enumerate all fields of a record, or at least the fields that are references. This facility is required to implement certain services such as debuggers, garbage collectors, and serialization. It needs some degree of type polymorphism.

The selection of a field from a record value yields a value.

Assignment and comparison of Records

Most languages allow assignment between records that have accurately the same record type (including same field types and same field's names, in the same order). Depending on the language, however, two record data types defined separately may be considered as distinct types even if they have exactly the same fields.

Some languages may also allow assignment between records whose fields have different names, matching each field value with the corresponding field variable by their locations within the record; so that, for example, a complex number with fields named *real* and *imag* can be allocated to a 2D point record variable with fields X and Y. In this substitute, the two operands are mandatory to have the same sequence of field types. Some languages may also need that corresponding types have the same size and encoding as well, so that the entire record can be assigned as an uninterpreted bit string. Other languages may be more flexible in this respect, and need only that each value field can be validly assigned to the corresponding variable field; so that, for example, a short integer field can be assigned to a long integer field, or vice-versa.

Other languages (such as COBOL) can be match fields and values by their names, rather than locations.

These same prospects apply to the comparison of two record values for equality. Some languages may also allow order comparisons ('<'and '>'), using the lexicographic order based on the comparison of individual fields

PL/I permits both of the previous types of assignment, and also allows structure expressions, such as a = a+1; where "a" is a record, or structure in PL/I terminology.

Representation in memory

The representation of records in computer memory varies depending on the programming languages. Typically the fields are kept in consecutive positions in memory, in the same order as they are declared in the record type. This may result in two or more fields kept into the same word of memory; definitely, this feature is often used in systems programming to access particular bits of a word. On the other hand, most compilers will add padding fields, mostly imperceptible to the programmer, in order to fulfil with arrangement constraints executed by the machine—say, that a floating point field must occupy a single word.

Some languages may implement a record as an array of addresses pointing to the fields (and, possibly, to their names and/or types). Objects in object-oriented languages are often implemented in rather complicated ways, especially in languages that allow multiple class inheritance.

Examples

The following shows the examples of record definitions:

- PL/I:

```
declare 1 date,
        2 year    picture   ‘9999’,
        2 month picture   ‘99’,
        2 day     month     ‘99’;
```

- C:

```
struct time
{
        int hour;
        int minute;
        int second;
}
```

Check your progress

Q1. What do you understand by vectors?

Q2. Explain Arrays? Also define dimensions of arrays.

Q3. Define records. Also explain operations of records.

1.4 Character Strings as Arrays

In this section, our job is to collect and print non-numeric text data (i.e. an arrangement of characters which are called strings). A string is a list (or string) of characters stored contiguously with an indicator to indicate the end of the string. Let us consider the task:

STRING0: Read and store a string of characters and print it out.

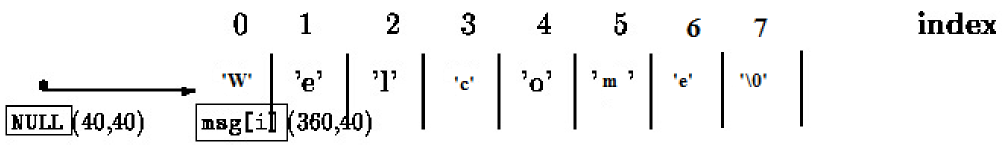
Since the characters of a string are kept contiguously, we can easily implement a string by using an array of characters if we retain the track of the number of elements kept in the array. However, common operations on strings include breaking them up into parts (known as substrings), joining them together (known as concatenation) to create new strings, replacing parts of them with other strings, etc. There must be some way of detecting the size of a current valid string stored in an array of characters.

In C language, a string of characters is stored in consecutive elements of a character array and terminated by the NULL character. For example, the string "Welcome" is stored in a character array, msg[], as follows:

```
char msg[SIZE];
    msg[0] = 'W';
    msg[1] = 'e';
    msg[2] = 'l';
    msg[3] = 'c';
    msg[4] = 'o';
```

```
msg[5] = 'm';
msg[6] = 'e';
msg[7] = '\0';
```

The NULL character is transcribed using the escape sequence '\0'. The ASCII value of NULL is 0, and NULL is defined as a macro to be 0 in ‘stdio.h’ header file; so programs can use the symbol, NULL, in expressions if the header file is included in to the program. The remaining elements in the array after the NULL may have any garbage values. When the string is regained, it will be retrieved starting at index 0 and following characters are obtained by incrementing the index until the first NULL character is reached signaling the end of the string. Figure 2.1 shows a string as it is stored in memory. Note, string constants, such as "Welcome" are automatically terminated by NULL by the compiler.



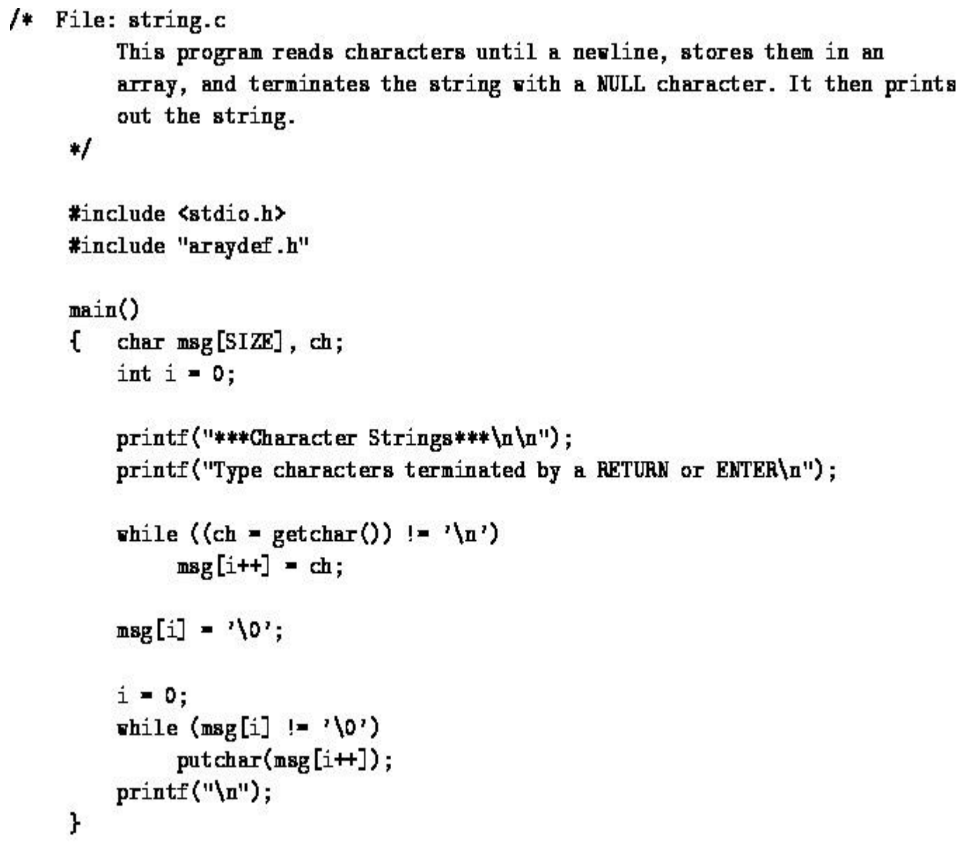
Given this execution of strings in C, the algorithm to implement our job is now easily written. We will assume that a string input is a sequence of characters terminated by a newline character. (The newline character is not part of the string). Here is the algorithm:

```
initialize index to 0
      while not a newline character entered
            read and store a character in the array at the next index
      increment the value of index
      dismiss the string of characters in the array with a NULL char.
Again initialize index to 0
navigate the array until a NULL character is reached
print the character  of array at index
increment the value of index
```

The program implementation has:

- a loop to read characters of string until a newline is reached;
- a statement to terminate the string with a NULL character;
- A loop to print the string.

The code is shown in Figure a sample session form the program is shown below.



```
/*  File: string.c
        This program reads characters until a newline, stores them in an
        array, and terminates the string with a NULL character. It then prints
        out the string.
    */

    #include <stdio.h>
    #include "araydef.h"

    main()
    {   char msg[SIZE], ch;
        int i = 0;

        printf("***Character Strings***\n\n");
        printf("Type characters terminated by a RETURN or ENTER\n");

        while ((ch = getchar()) != '\n')
            msg[i++] = ch;

        msg[i] = '\0';

        i = 0;
        while (msg[i] != '\0')
            putchar(msg[i++]);
        printf("\n");
    }
```

Sample Session:

- ***Character Strings***
- Type characters terminated by a RETURN or ENTER
- *Welcome*
- Welcome

The first while loop reads a character by character into ch and checks if it is a newline, which rejected and the loop terminated. Otherwise, the character is kept in msg[i]and the array index, i, incremented. When the loop ends, a NULL character is added to the string of characters. In this program, we have assumed that the size of msg[] is large enough to store the string. Since a line on a terminal is 80 characters wide and since we have also defined SIZE to be 100, this seems a harmless statement.

The next while loop in the program navigates the string and prints each character by character until a NULL character is got. Note, we do not want to save a count of the number of characters stored in the array in this program since the first NULL character encountered indicates the end of the string. In our program, when the first NULL is got we dismiss the string output with a newline.

The assignment expression in the above program:

```
msg[i] = '\0';
```

can also be written as:

```
msg[i] = NULL;
```

or:

```
msg[i] = 0;
```

In the first case, the character whose ASCII value is 0 is assigned to; where in the other cases, a zero value is assigned to msg[i]. The above assignment expressions are same. The first expression makes it clear that a null character (\0) is assigned to msg[i], but the second uses a symbolic constant which is easier to recognize and understand.

To accommodate the ending NULL character, the size of an array that houses a string must be at least one greater than the expected maximum size of string. Since different strings may be kept in an array at different times, the first NULL character in the array demarcates a valid string. The importance of the NULL character to signal the end of a valid string is obvious. If there were no NULL character inserted after the valid string, the loop traversal would continue to print values interpreted as characters, maybe beyond the array boundary until it fortuitously found a (0) character.

The second while loop may also be written:

```
while (msg[i] != NULL)
        putchar(msg[i++]);
```

and the while condition further simplified as:

```
while (msg[i])
        putchar(msg[i++]);
```

If msg[i] is any character with a non-zero ASCII value, the while expression evaluates to True. If msg[i] is the NULL character, its value is zero and thus False. The last form of the while condition is the more common usage. While we have used the increment operator in the putchar() argument, it may also be used separately for clarity:

```
while (msg[i])
{
        putchar(msg[i]);
        i++;
    }
```

It is likely to be possible for a string to empty; that is, a string may have no characters in it. An empty string is a character array with the NULL character in the zeroth index position,i.e. on msg[0].

1.5 Variable size data structure

Structured Data Types

A data structure is a data object that holds other data objects as its elements or modules.

Structured Data Objects and Data Types: -

A data object that is built as a collection of other data objects called components is termed a structured data object or date, structure. A component may be fundamental or it may be another data structure (e.g.. a component of an array may be a number or it may be a record, character string, or another types of an array). Many of the issues and ideas surrounding data structures in programming languages are the same as for fundamental data objects and have been treated. As with fundamental data objects some are defined by the programmer and others are defined by the system during the execution of the program. The bindings of data structures to values, names, and locations are essential and somewhat more complex in this setting. With structured data, the requirements, specification and implementation of structural information become a central problem: how to indicate the component data objects of a data structure and their relationships in such a way that selection of a component from the structure is straightforward. Second many operations or procedures on data structures bring up storage management issues that are not present for elementary data objects.

Specification of Data Structure Types

The main features for requiring data structures include the following:

1. Number of components or modules: - A data structure may be of fixed size if the number of components or module is invariant during its lifetime or of variable size if the number of components or module changes vigorously. Variable-size data structure types usually define operations that addition and remove components from structures. Arrays and records are some common instances of fixed-size data structure types; stacks, lists, set, table and files are examples of variable-size types. Variable' size data objects repeatedly use a pointer data type that allows fixed-size data objects to be linked together explicitly by the programmer.

2. Component type: - A data structure is uniform if all its components are of the same type or homogeneous. It is heterogeneous if its components are of different types. Arrays, sets, and files are usually known as homogeneous whereas records and lists are usually known as heterogeneous.

3. Names to be used for selecting components or module: - A data structure type required a selection mechanism for recognizing separate components of the data structure. For an array, the name of an individual component may be an integer subscript or sequence of subscripts: for a table the name may be a programmer-defined identifier: for a record, the name is usually a programmer defined identifier. Some data structure types such as stacks and files permit access to only a specific component (e.g. the top or existing component) at any time but operations are delivered to change the component that is currently accessible.

4. Maximum number of components: -For a variable-size data structure such as a stack, a maximum size for the structure in terms of number of components may be specified.

5. Organization of the components: -The most common organization is a simple linear sequence of components. Vectors (one-dimensional arrays), records, stacks, lists, and files are data structures with this organization. Array record, and list types, however are usually comprehensive to multidimensional forms: multidimensional arrays, records whose components arc records, and lists whose components are lists. These comprehensive forms may be preserved as separate types or simply as the basic sequential type in which the components are data structures of similar type. For example, a two-dimensional array (matrix) may be considered as a separate type (as in FORTRAN's A (i, j)) or as a vector of vectors (as in C's A vector in which the components (the rows or columns) are vectors. Records also may include variants: alternative sets of components of which only one is included in each data object of that type.

Operations on Data Structures

Specification of the domain and range of operations on data structure types may be given in much the same manner as for elementary types. Some new classes of operations are of particular importance:

1. **Component selection operations:** Processing of data structures often continues by retrieving each component of the structure. Two types of selection operations access components of a data structure and make them available for processing by other operations: *random selection*, in which an arbitrary component of the data structure is accessed, and *sequential Selection* in which components are selected in a predetermined order. For example. In processing a vector, the subscripting operation selects a component at random (e.g.. V[4]). and subscripting combined with a for or while loop is used to select a sequence of components—as in

   for I := 1 to 10 do …………. V[I] …………………..

2. **Whole-data structure operations:** Operations may yield entire data structures as arguments and produce new data structures as results. Most languages deliver a limited set of such whole-data structure operations (e.g.. addition of two arrays. assignment of one record to another, or a union operation on sets). Languages such as *APL* and *SNOBOL4* offer rich sets of whole-data structure operations however so that the programmer need seldom select individual modules of data structures for processing.

3. **Insertion/deletion of components:** Operations that change the number of components in a data structure have a major influence on storage representations and storage management for data structures.

4. **Creation/destruction of data structures:** Operations that produce and destroy data structures also have a major impact on storage management for data structures

   Accessing or selection of a component or data value in a data object should be eminent from the related operation of referencing. Ordinarily a data object is given a name (e.g.. the vector previously named V). When we write V [2] in a program to select the second component of V. we actually invoke a two-step sequence composed of a referencing operation Allowed by a selection operation. The referencing operation regulates the current location of the name V (i.e.. its 1-value), returning as its result a pointer to the location of the entire vector data object designated by the name V. The selection operation takes the pointer to the vector, together with the subscript 4 of the designated component of the vector, and returns a pointer to the location of that particular component within the vector.

1.6 Sets

A set is a data object containing an unordered collection of distinct values. In contrast, a list is an ordered collection of values, some of which may be repeated. The following are the basic operations on sets:

1. Membership: - Is data value X a member of set S (i.e.. is $X \in S$)?

2. Insertion and deletion of single values: -Insert data value X in set S provided it is not already a member of S. Delete data value X from S if a member.

3. Union, intersection and difference of sets: - Given two sets. S1 and S2, create set S3 that contains all members of both S1 and S2. With duplicates deleted (union operation), create S3 to contain only values that are members of both SI and S2. (Intersection operation), or create S3 contain only values that are in SI but not in S2 (difference operation).

Note that accessing of components of a set by subscript or relative position plays no part in set processing.

Implementation of Set

In any programming languages, the term set is smeared to a data structure representing an ordered set. An ordered set is actually a list with matching values removed; it requires no special attention. The unordered set however declares two specialized storage representations that merit attention.

Bit-string representation of sets

The storage of bit-string representation is suitable, where the size of the essential universe of values (the values that may appear in set data objects) is known to be lesser or small. Suppose that there are N elements in the universe. We can order these elements arbitrarily as u, u_1, u_2 ... u_N. A set of elements chosen from this universe may then be signified by a hit string of length N. where the i^{th} bit in the string is a 1 if u1 is in the set and 0 if not. The bit string represents the characteristic function of the set. With this representation insertion of an element into a set consists of setting the appropriate bit to 1, deletion consists of setting the appropriate bit to 0 and membership is determined simply by interrogating the appropriate bit. The union, intersection, and difference operations on whole sets max' be signified by the Boolean operations on bit strings that are usually delivered by the hardware: The Boolean or of two bit-strings signifies union and signifies intersection, and the end of the first string and the complement of the second represents the difference operation.

The hardware support for bit-string operations, if provided, makes manipulation of the bit-string representation of sets efficient. However, the hardware operations usually apply only to bit strings up to a certain fixed length (e.g., the word length of the central memory). For strings longer than this maximum, software simulation must be used to break up the string into lesser units that can be handled by the hardware.

Hash-coded representation of sets

Another representation for a set is based on the technique of hash calling or scatter storage. When the underlying universe of possible values is large (e.g. when the set covers numbers or character strings) than this method may be used. It permits the membership test, insertion, and (with some methods) deletion of values to be achieved efficiently. However, union, intersection and difference operations must be implemented as a sequence of membership tests, insertions, and deletions of individual elements so they are inefficient. To be effective and efficient, hash-coding methods also require a substantial allocation of storage. Most commonly, a language does not provide this representation for a set data type available to the user, but the language implementation uses this representation for some of the system-defined data required during translation or execution. For instance, most LISP executions use this storage representation for the set called the *object list*, which consists of the names of all the atomic data objects in use by a LISP program during its implementation. Almost every compiler uses hash coding to look up names in its symbol table. In a vector, every subscript defines a unique component that is easily addressable with a simple accessing function. The goal with hash coding is to duplicate this property so that efficient accessing of a component is maintained. However, the problem is that the potential set of valid names is gigantic compared with the available memory. If we allocate memory, called the hash table, at least twice as large as what we expect to use, then hash coding can be very efficient (e.g.. if our set will have 1,000 members, then allocate space for 2000 members). Rather than storing elements of the set in sequential locations within this block, however, the elements are dispersed randomly through the block. The trick is to store each new element in such a way that its presence or absence can later be directly determined without a search of the block.

Consider how this may be done. Suppose that we wish to add a new element x represented by bit siring B_x to the set S, represented by the block of storage M_s, First, we must determine whether x is already a member of S and, if not, add it to the set. We determine a position for B the block M_s by the application of a hashing function to the bit string $B_{x.}$ The hashing function hashes (chops up into little pieces and mixes together) the bit string B, and then extracts a hash address I_x from the results. This hash address is used as an index pointing to a position in the block M. We look at that position in the block, if A' is already in the set, then it must be stored at that position. If not, then we store the bit string BA at the location designated by I_x. Any later attempt to find whether x is a member of S will be answered by hashing the new bit string B_x representing x obtaining I_x, retrieving the block M_s at that position and finding the previously stored string B_x. No search of the table is ever desirable.

Exactly how the hashing function works is not critical as long as it is relatively fast and generates hash addresses that are fairly randomly distributed. An example illustrates the idea more directly. Suppose that we allocate a block of 1024 words (a block length equal to a power of 2 for a binary computer is most suitable) for the block M_s. Suppose also that the data items to be stored are character strings represented by double-word bit strings. We may represent a set of up to 511 distinct elements within this block. Suppose that the starting address of the block in memory is α. An appropriate hash address for such a table would be a string I_x, of 9 bits because the formula $\alpha + 2 X I_x$ would always generate an address within the block. We might compute I, from a given two-word-long bit string B_x, by the following algorithm (assuming that B_x, is stored in words a and b)

1. Multiply a & b, giving c (two-word product).

2. Add together the two words of c, giving one word value d.

3. Square d, giving e.

4. Extract the center nine bits of e. giving I_x

Even the best hashing function cannot assurance that dissimilar data items will generate dissimilar hash addresses when hashed. Although it is needed that the hashing function spread the generated hash addresses throughout the block as much as possible, almost inevitably two data items may be hashed to the same hash address, leading to a collision. A collision occurs when we have a data item to be added to the set, go to the block at the designated hash address and find the block entry at that point filled with a data item different from the one to be stored (but that just happened to hash to the same hash address). (That is two different data items x and y produce the same hash value I_x.) Many methods or techniques for handling collisions are known:

1. Rehashing: - We might change the original bit string B_x, (e.g. by multiplication by a constant) and then rehash the result, producing a new hash address. If another collision occurs, we rehash again until either B_x is found or an empty block location is encountered.

2. Sequential scan: - From the original point of the collision in the block, we might begin a sequential (end-around) search until either B_x, is got or an empty block location is encountered.

3. Bucketing: - In place of direct storage in the block we might temporary pointers to linked bucket lists of the elements having the same hash addresses. After hashing B_x, and retrieving the pointer to the appropriate bucket list, we search the list for B_x, and, if not found, add it to the end of the list.

Dissimilar hashing functions are appropriate depending on the properties of the bit-string representations of the data to be stored. With a good hashing function and the table at most half full, collisions hardly occur. Collision resolution is why the 512-entry table in the prior example can only he used to store 511 entries. Eventually collision resolution must find an empty entry or else the resolution process may never terminate.

Check your progress

Q1. How character string works? Elaborate your answer.

Q2. Define structured data object and its types.

Q3. Explain sets. Also define Operations of sets.

1.7 Summary

In this unit you learnt about vectors, arrays, records, character strings, variable size data structure and sets. All these are very important in computer science.

- Vectors can hold objects of various types.
- Vector will also resize, shrink or grow, as elements are added.
- Arrays are a data structure that is used to store a group of objects of the same type sequentially in memory.
- Records are a generalization of tuples.
- Subtyping is a key feature of object-oriented languages. It was first introduced in the SIMULA languages by the Norwegian researchers Dahl and Nygaard.

1.8 Review Questions

Q1. What do you mean by vectors? Define all types of vectors with example.

Q2. What is an array? How many types of an array available?

Q3. What is the difference between an arrays and vectors? Define with example.

Q4. What is record? Define in details with example.

Q5. Write a short note on Sets?

UNIT-II Encapsulation

Structure

2.0 Introduction

In this unit we defines the concept of Encapsulation. This is the second unit of this block. In this unit there are six sections. In section Sec 2.1 we defined input files. Next section Sec 2.2 explain about the concept of encapsulation. In this section you will learn about the object oriented programming's important concept called encapsulation. Encapsulation binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of data hiding. Sec. 2.3 describe about the data hiding. It is a software development technique specifically used in object-oriented programming (OOP) to hide internal object details (data members). Data hiding ensures exclusive data access to class members and protects object integrity by preventing unintended or intended changes. Sec. 2.4 define sub program and its working in programming language. In Sec. 2.5 and 2.6 you will find summary and review questions respectively.

Objectives

After studying this unit you should be able to:

- Define encapsulation
- Express data hiding
- Define sub programs

2.1 Input files

In C programming, file is a place on disk where a group of related data is stored.

Why files are needed?

When the program is terminated, the entire data is lost in C programming. If you want to keep large volume of data, it is time consuming to enter the entire data. But, if file is created, these information can be accessed using few commands.

There are large numbers of functions to handle file I/O in C language. In this tutorial, you will learn to handle standard I/O (High level file I/O functions) in C.

High level file I/O functions can be categorized as:

1. Text file
2. Binary file

File Operations

1. Creating a new file
2. Opening an existing file
3. Reading from and writing information to a file
4. Closing a file

Working with file

While working with file, you need to declare a pointer of type file. This declaration is needed for communication between file and program.

FILE *ptr;

Opening a file

Opening a file is performed using library function fopen(). The syntax for opening a file in standard I/O is:

ptr=fopen("fileopen","mode")

For Example:

fopen("E:\\cprogram\program.txt","w");

/* -- */

E:\\cprogram\program.txt is the location to create file.

"w" represents the mode for writing.

/* -- */

Here, the program.txt file is opened for writing mode.

Opening Modes in Standard I/O

File Mode	Meaning Of Mode	During Inexistence Of File
r	Open for reading.	If the file does not exist, fopen() returns NULL.
w	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a	Open for append. i.e., Data is added to end of file.	If the file does not exists, it will be created.
r+	Open for both reading and writing.	If the file does not exist, fopen() returns NULL.
w+	Open for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a+	Open for both reading and appending.	If the file does not exists, it will be created.

Closing a File

The file should be closed after reading/writing of a file. Closing a file is performed using library function fclose().

fclose(ptr); //ptr is the file pointer associated with file to be closed.

The Functions fprintf() and fscanf() functions.

The functions fprintf() and fscanf() are the file version of printf() and fscanf(). The only difference while using fprintf() and fscanf() is that, the first argument is a pointer to the structure FILE

Binary Files

Depending upon the way file is opened for processing; a file is classified into text file and binary file. If a large amount of numerical data it to be stored, text mode will be insufficient. In such case binary file is used.

Working of binary files is similar to text files with few differences in opening modes, reading from file and writing to file.

Opening modes of binary files

Opening modes of binary files are rb, rb+, wb, wb+,ab and ab+. The only difference between opening modes of text and binary files is that, b is appended to indicate that, it is binary file.

Reading and writing of a binary file

Functions fread() and fwrite() are used for reading from and writing to a file on the disk respectively in case of binary files.

Function fwrite() takes four arguments, address of data to be written in disk, size of data to be written in disk, number of such type of data and pointer to the file where you want to write.

fwrite(address_data,size_data,numbers_data,pointer_to_file);

Function fread() also take 4 arguments similar to fwrite() function as above.

2.2 Encapsulation

The idea of encapsulation comes from (i) the need to cleanly distinguish between the specification and the implementation of an operation and (ii) the need for modularity. Modularity is necessary to structure complex applications designed and implemented by a team of programmers. It is also necessary as a tool for protection and authorization.

There are two views of encapsulation: the programming language view (which is the original view since the concept originated there) and the database adaptation of that view.

The idea of encapsulation in programming languages comes from abstract data types. In this view, an object has an interface part and an implementation part. The interface part is the specification of the set of operations that can be performed on the object. It is the only visible part of the object. The implementation part has a data part and a procedural part. The data part is the representation or state of the object and the procedure part describes, in some programming language, the implementation of each operation.

The database translation of the principle is that an object encapsulates both program and data. In the database world, it is not clear whether the structural part of the type is or is not part of the interface (this depends on the system), while in the programming language world, the data structure is clearly part of the implementation and not of the interface.

Consider, for instance, an Employee, in a relational system, an employee is represented by some tuple. It is queried using a relational language and, later, an application programmer writes programs to update this record such as to raise an Employee's salary or to fire an Employee. These are generally either written in an imperative programming language with embedded DML statements or in a fourth generation language and are stored in a traditional file system and not in the database. Thus, in this approach, there is a sharp distinction between program and data, and between the query language (for *ad hoc* queries) and the programming language (for application programs).

In an object-oriented system, we define the Employee as an object that has a data part (probably very similar to the record that was defined for the relational system) and an operation part, which consists of the *raise* and *fire* operations and other operations to access the Employee data. When storing a set of Employees, both the data and the operations are stored in the database.

Thus, there is a single model for data and operations, and information can be hidden. No operations, outside those specified in the interface, can be performed. This restriction holds for both update and retrieval operations.

Encapsulation provides a form of ``logical data independence'': we can change the implementation of a type without changing any of the programs using that type. Thus, the application programs are protected from implementation changes in the lower layers of the system.

We believe that proper encapsulation is obtained when only the operations are visible and the data and the implementation of the operations are hidden in the objects.

However, there are cases where encapsulation is not needed, and the use of the system can be significantly simplified if the system allows encapsulation to be be violated under certain conditions. For example, with ad-hoc queries the need for encapsulation is reduced since issues such as maintainability are not important. Thus, an encapsulation mechanism must be provided by an OODBS, but there appear to be cases where its enforcement is not appropriate.

2.3 Information Hiding

In computer science, **information hiding** is the principle of segregation of the *design decisions* in a computer program that are most likely to change, thus protecting other parts of the program from extensive modification if the design decision is changed. The protection involves providing a stable interface which protects the remainder of the program from the implementation (the details that are most likely to change).

Written another way, information hiding is the ability to prevent certain aspects of a class or software component from being accessible to its clients, using either programming language features (like private variables) or an explicit exporting policy.

Overview

The term *encapsulation* is often used interchangeably with information hiding. Not all agree on the distinctions between the two though; one may think of information hiding as being the principle and encapsulation being the technique. A software module hides information by encapsulating the information into a module or other construct which presents an interface.

A common use of information hiding is to hide the physical storage layout for data so that if it is changed, the change is restricted to a small subset of the total program. For example, if a three-dimensional point (*x*,*y*,*z*) is represented in a program with three floating point scalar variables and later, the representation is changed to a single array variable of size three, a module designed with information hiding in mind would protect the remainder of the program from such a change.

In object-oriented programming, information hiding (by way of nesting of types) reduces software development risk by shifting the code's dependency on an uncertain implementation (design decision) onto a well-defined interface. Clients of the interface perform operations purely through it so if the implementation changes, the clients do not have to change.

Example of information hiding

Information hiding serves as an effective criterion for dividing any piece of equipment, software or hardware, into modules of functionality. For instance a car is a complex piece of equipment. In order to make the design, manufacturing, and maintenance of a car reasonable, the complex piece of equipment is divided into modules with particular interfaces hiding design decisions. By designing a car in this fashion, a car manufacturer can also offer various options while still having a vehicle which is economical to manufacture.

For instance, a car manufacturer may have a luxury version of the car as well as a standard version. The luxury version comes with a more powerful engine than the standard version. The engineers designing the two different car engines, one for the luxury version and one for the standard version, provide the same interface for both engines. Both engines fit into the engine bay of the car which is the same between both versions. Both engines fit the same transmission, the same engine mounts, and the same controls. The differences in the engines are that the more powerful luxury version has a larger displacement with a fuel injection system that is programmed to provide the fuel air mixture that the larger displacement engine requires.

In addition to the more powerful engine, the luxury version may also offer other options such as a better radio with CD player, more comfortable seats, a better suspension system with wider tires, and different paint colors. With all of these changes, most of the car is the same between the standard version and the luxury version. The radio with CD player is a module which replaces the standard radio, also a module, in the luxury model. The more comfortable seats are installed into the same seat mounts as the standard types of seats. Whether the seats are leather or plastic, or offer lumbar support or not, doesn't matter.

The engineers design the car by dividing the task up into pieces of work which are assigned to teams. Each team then designs their component to a particular standard or interface which allows the sub-team flexibility in the design of the component while at the same time ensuring that all of the components will fit together.

Motor vehicle manufacturers frequently use the same core structure for several different models, in part as a cost-control measure. Such a "platform" also provides an example of information hiding, since the floor pan can be built without knowing whether it is to be used in a sedan or a hatchback.

As can be seen by this example, information hiding provides flexibility. This flexibility allows a programmer to modify functionality of a computer program during normal evolution as the computer program is changed to better fit the needs of users. When a computer program is well designed decomposing the source code solution into modules using the principle of information hiding, evolutionary changes are much easier because the changes typically are local rather than global changes.

Cars provide another example of this in how they interface with drivers. They present a standard interface (pedals, wheel, shifter, signals, gauges, etc.) on which people are trained and licensed. Thus, people only have to learn to drive a car; they don't need to learn a completely different way of driving every time they drive a new model. (Granted, there are manual and automatic transmissions and other such differences, but on the whole cars maintain a unified interface.)

Check your progress

Q1. Why files are needed? Explain.

Q2. Define operations of files.

Q3. Differentiate between encapsulation and data hiding.

2.4 Subprograms

Fundamentals of Subprograms

General Subprogram Characteristics

a. A subprogram has a single entry point.
b. The caller is suspended during execution of the called subprogram. "Only one subprogram in execution at any given time."
c. Control always returns to the caller when the called subprogram's execution terminates

Basic Definitions

- A **subprogram definition** is a description of the actions of the subprogram abstraction.
- A **subprogram call** is an explicit request that the called subprogram be executed.
- A subprogram is said to be **active** if, after having been called, it has begun execution but has not yet completed that execution.
- The two fundamental types of the subprograms are:

    - Procedures
    - Functions
- A **subprogram header** is the first line of the definition, serves several definitions:
    - It specifies that the following syntactic unit is a subprogram definition of some particular kind.
    - The header provides a name for the subprogram.
    - May optionally specify a list of parameters.
- Consider the following examples:
- Fortran

  Subroutine Adder(parameters)
- Ada

  procedure Adder(parameters)
- C

  void Adder(parameters)
- No special word appears in the header of a C subprogram to specify its kind.
- The **parameter profile** of a subprogram is the number, order, and types of its formal parameters.
- The **protocol** of a subprogram is its parameter profile plus, if it is a function, its return type.
- A subprogram declaration provides the protocol, but not the body, of the subprogram.
- A formal parameter is a dummy variable listed in the subprogram header and used in the subprogram.
- An actual parameter represents a value or address used in the subprogram call statement.
- Function declarations are common in C and C++ programs, where they are called **prototypes**.

Parameters

- Subprograms typically describe computations. There are two ways that a non-local method program can gain access to the data that it is to process:
    1. Through direct access to non-local variables.
        - The only way the computation can proceed on different data is to assign new values to those non-local variables between calls to the subprograms.

    - Extensive access to non-locals can reduce reliability.

  2. Through parameter passing “more flexible”.
     - A subprogram with parameter access to the data it is to process is a parameterized computation.
     - It can perform its computation on whatever data it receives through its parameters.

- A **formal parameter** is a dummy variable listed in the subprogram header and used in the subprogram.
- Subprograms call statements must include the name of the subprogram and a list of parameters to be bound to the formal parameters of the subprogram.
- An **actual parameter** represents a value or address used in the subprogram call statement.
- Actual/Formal Parameter Correspondence:
  1. **Positional**: The first actual parameter is bound to the first formal parameter and so forth. “Practical if list is short.”
  2. **Keyword**: the name of the formal parameter is to be bound with the actual parameter. “Can appear in any order in the actual parameter list.”

     SORT(LIST => A, LENGTH => N);

     - **Advantage**: order is irrelevant
     - **Disadvantage**: user must know the formal parameter’s names.

- Default Values:

  ```
  procedure SORT(LIST : LIST_TYPE;
                  LENGTH : INTEGER := 100);
  ...
  SORT(LIST => A);
  ```

- In C++, which has no keyword parameters, the rules for default parameters are necessarily different.
- The default parameters must appear last, for parameters are positionally associated.
- Once a default parameter is omitted in a call, all remaining formal parameters must have default values.

  float compute_pay (float income, float tax_rate, int exemptions = 1)

- An example call to the C++ compute_pay function is:

pay = compute_pay (20000.0, 0.15);

Procedures and Functions

- **Procedures**: provide user-defined parameterized computation statements.
- The computations are enacted by single call statements.
- Procedures can produce results in the calling program unit by two methods:
    - If there are variables that are not formal parameters but are still visible in both the procedure and the calling program unit, the procedure can change them.
    - If the subprogram has formal parameters that allow the transfer of data to the caller, those parameters can be changed.
- **Functions** provide user-defined operators which are semantically modeled on mathematical functions.
    - If a function is a faithful model, it produces no side effects.
    - It modifies neither its parameters nor any variables defined outside the function.

Design Issues for Subprograms

1. What parameter passing methods are provided?
2. Are parameter types checked?
3. Are local variables static or dynamic?
4. Can subprogram definitions appear in other subprogram definitions?
5. What is the referencing environment of a passed subprogram?
6. Can subprograms be overloaded?
7. Are subprograms allowed to be generic?

Local Referencing Environments

- Vars that are defined inside subprograms are called **local vars**.
- Local vars can be either static or stack dynamic "bound to storage when the program begins execution and are unbound when execution terminates."
- <u>Advantages of using stack dynamic</u>:

  a. Support for recursion.

  b. Storage for locals is shared among some subprograms.
- <u>Disadvantages</u>:

  a. Allocation/deallocation time.

b. Indirect addressing “only determined during execution.”

c. Subprograms cannot be history sensitive “can’t retain data values of local vars between calls.”

- Advantages of using static vars:

  a. Static local vars can be accessed faster because there is no indirection.

  b. No run-time overhead for allocation and deallocation.

  c. Allow subprograms to be history sensitive.

- Disadvantages:

  a. Inability to support recursion.

  b. Their storage can’t be shared with the local vars of other inactive subprograms.

- In C, and C++ functions, locals are stack-dynamic unless specifically declared to be **static**.

- Ex:

```
int adder(int list[ ], int listlen) {
    static int sum = 0;
    int count;                    //count is stack-dynamic
    for (count = 0; count < listlen; count++)
      sum += list[count];
    return sum;
}
```

Parameter Passing Methods

Semantic Models of Parameter Passing

- Formal parameters are characterized by one of three distinct semantic models:
  - **in mode**: They can receive data from corresponding actual parameters.
  - **out mode**: They can transmit data to the actual parameter.
  - **inout mode**: They can do both.

- There are two conceptual models of how data transfers take places in parameter transmission:

  - Either an actual value is copied (to the caller, to the callee, or both ways), or
  - An access path is transmitted.
- Most commonly, the access path is a simple pointer or reference.
- Figure 2-1 below illustrates the three semantics of parameter passing when values are copied.

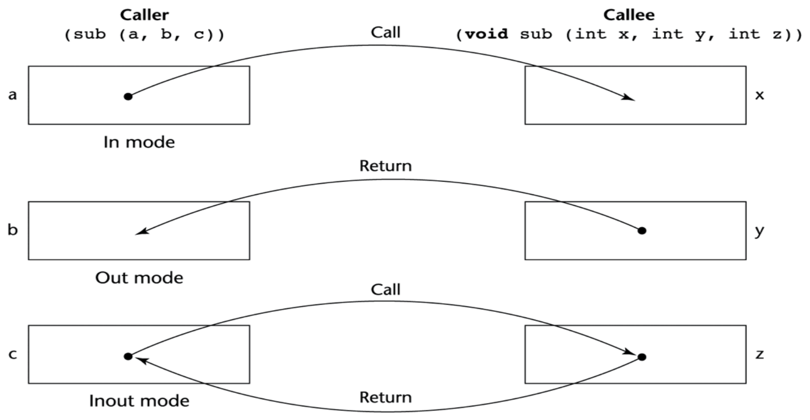


Figure 2-1

Implementation Models of Parameter Passing

1. **Pass-by-Value**

   When a parameter is passed by value, the value of the actual parameter is used to initialize the corresponding formal parameter, which then acts as a local var in the subprogram, thus implementing in-mode semantics.

   <u>Disadvantages</u>:

   Additional storage is required for the formal parameter, either in the called subprogram or in some area outside both the caller and the called subprogram.

   The actual parameter must be copied to the storage area for the corresponding formal parameter. "If the parameter is large such as an array, it would be costly.

2. **Pass-by-Result**

   Pass-by-Result is an implementation model for out-mode parameters.

   When a parameter is passed by result, no value is transmitted to the subprogram.

   The corresponding formal parameter acts as a local var, but just before control is transferred back to the caller, its value is transmitted back to the caller's actual parameter, which must be a var.

One problem with the pass-by-result model is that there can be an actual parameter collision, such as the one created with the call.

sub(p1, p1)

In sub, assuming that the two formal parameters have different names, the two can obviously be assigned different values.

Then whichever of the two is copied to their corresponding actual parameter last becomes the value of p1.

3. **Pass-by-Value-Result**

   It is an implementation model for in-out mode parameters in which actual values are copied.

   It is a combination of pass-by-value and pass-by-result.

   The value of the actual parameter is used to initialize the corresponding formal parameter, which then acts as a local var.

   At subprogram termination, the value of the formal parameter is transmitted back to the actual parameter.

   It is sometimes called **pass-by-copy** because the actual parameter is copied to the formal parameter at subprogram entry and then copied back at subprogram termination.

4. **Pass-by-reference**

   This method transmits an access path to the called subprogram. This provides the access path to the cell storing the actual parameter.

   The actual parameter is shared with the called subprogram.

   <u>Advantages</u>:

   The passing process is efficient in terms of time and space.

   <u>Disadvantages</u>:

   Access to the formal parameters will be slower than pass-by-value, because of additional level of indirect addressing that is required.

   Inadvertent and erroneous changes may be made to the actual parameter.

   Aliases can be created as in C++.

   void fun(**int** &first, **int** &second)

   If the call to fun happens to pass the same var twice, as in

   fun(total, total)

   Then first and second in fun will be aliases.

5. **Pass-by-Name**

The method is an inout-mode parameter transmission that doesn't correspond to a single implementation model.

When parameters are passed by name, the actual parameter is, in effect, textually substituted for the corresponding formal parameter in all its occurrences in the subprogram.

A formal parameter is bound to an access method at the time of the subprogram call, but the actual binding to a value or an address is delayed until the formal parameter is assigned or referenced.

Type-Checking Parameters

- It is now widely accepted that software reliability demands that the types of actual parameters be checked for consistency with the types of the corresponding formal parameters.
- Ex:

```
result = sub1(1)
```

- The actual parameter is an integer constant. If the formal parameter of sub1 is a floating-point type, no error will be detected without parameter type checking.
- Early languages, such as Fortran 77 and the original version of C, did not require parameter type checking.
- Perl, PHP, and JavaScript do not.

Implementing Parameter-Passing Methods

- In most contemporary languages, parameter communication takes place through the run-time stack.
- The run-time stack is initialized and maintained by the run-time system, which is a system program that manages the execution of programs.
- The run-time stack is used extensively for subprogram control linkage and parameter passing.
- **Pass-by-value** parameters have their values copied into stack locations.
- The stack location then serves as storage for the corresponding formal parameters.
- **Pass-by-result** parameters are implemented as the opposite of pass-by-value.
- The values assigned to the pass-by-result actual parameters are placed in the stack, where they can be retrieved by the calling program unit upon termination of the called subprogram.
- **Pass-by-value-result** parameters can be implemented directly from their semantics as a combination pf **pass-by-value** and **pass-by-result**.

- The stack location for the parameters is initialized by the call and it then used like a local var in the called subprogram.
- **Pass-by-reference** parameters are the simplest to implement.
- Only its address must be placed in the stack.
- Access to the formal parameters in the called subprogram is by indirect addressing from the stack location of the address.
- Figure 2-2 below illustrates the previous parameters' passing methods.

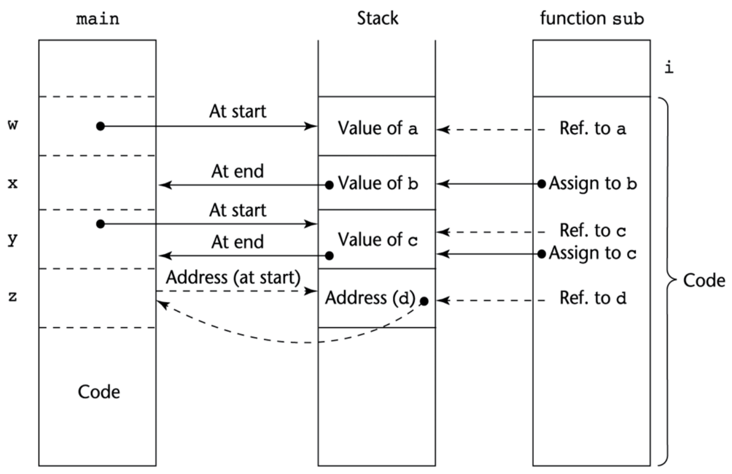


Figure 2-2

- The subprogram sub is called from main with the call sub (w, x, y, z), where w is passed by result, y is passed by value-result, and z is passed by reference.
- A subtle but fatal error can occur with pass-by-reference and pass-by-value-result parameters if care is not take in their implementation.
- Suppose a program contains two references to the constant **10**, the first as an actual parameter in a call to a subprogram.
- Further suppose that the subprogram mistakenly changes the formal parameter that corresponds to the **10** to the value **5**.
- The compiler may have built a single location for the value **10** during compilation, as compilers often do, and use that location for all references to the constant **10** in the program.
- But after the return from the subprogram, all subsequent occurrences of **10** will actually be references to the value **5**.
- If this is allowed, it creates a programming problem that is hard to diagnose.
- This happened in many implementations of Fortran IV.

Parameters that are Subprogram Names

- In languages that allow nested subprograms, such as JavaScript, there is another issue related to subprogram names that are passed as parameters.
- The question is what referencing environment for executing the passed subprogram should be used.
- The three choices are:
  1. It is the environment of the call statement that enacts the passed subprogram "**Shallow binding**."
  2. It is the environment of the definition of the passed subprogram "Deep binding."
  3. It is the environment of the call statement that passed the subprogram as an actual parameter "**Ad hoc binding**; has never been used"
- Ex: "written in the syntax of Java"

```
function sub1( )
{
    var x;
    function sub2( ) {
      alert(x);              // Creates a dialog box with the value of x
    };

    function sub3( )
    {
      var x;
      x = 3;
      sub4(sub2);
    };

    function sub4(subx ) {
      var x;
      x = 4;
      subx( );
    };
x = 1;
sub3( );

};
```

- Consider the execution of sub2 when it is called in sub4.
- **Shallow Binding**: the referencing environment of that execution is that of sub4, so the reference to x in sub2 is bound to the local x in sub4, and the output of the program is 4.
- **Deep Binding**: the referencing environment of sub2's execution is that of sub1, so the reference so the reference to x in sub2 is bound to the local x in sub1 and the output is 1.
- **Ad hoc**: the binding is to the local x in sub3, and the output is 3.

Overloaded Subprograms

- An overloaded operator is one that has multiple meanings. The types of its operands determine the meaning of a particular instance of an overloaded operator.
- For example, if the * operator has two floating-point operands in a Java program, it specifies floating-point multiplication.
- But if the same operator has two integer operands, it specifies integer multiplication.
- An **overloaded subprogram** is a subprogram that has the same name as another subprogram in the same referencing environment.
- Every version of an overloaded subprogram must have a unique protocol; that is, it must be different from the others in the number, order, or types of its parameters, or in its return if it is a function.
- The meaning of a call to an overloaded subprogram is determined by the actual parameter list.
- Users are also allowed to write multiple versions of subprograms with the same name in Ada, Java, C++, and C#.
- Overloaded subprograms that have default parameters can lead to ambiguous subprogram calls.

Generic Subprograms

- A programmer should not need to write four different sort subprograms to sort four arrays that differ only in element type.
- A generic or polymorphic subprogram takes parameters of different types on different activations.
- Overloaded subprograms provide a particular kind of polymorphism called **ad hoc polymorphism**.

- **Parametric polymorphism** is provided by a subprogram that takes a generic parameter that is used in a type expression that describes the types of the parameters of the subprogram.

Generic Functions in C++

- Generic functions in C++ have the descriptive name of template functions.
- The following is the C++ version of the generic sort subprogram.

```
template <class type>
void generic_sort (Type list [ ], int len) {
  int top, bottom;
  Type temp;
  for (top = 0, top < len –2; top ++)
      for (bottom = top + 1; bottom <  len – 1; bottom++)
         if (list [top] > list [bottom]) {
            temp = list [top];
            list[top] = list[bottom];
         } // end for bottom
  } // end for generic
```

- The instantiation of this template function is:

```
float flt_list [100];
…
generic_sort (flt_list, 100);
```

Design Issues for Functions

- Are side effects allowed?
- What types of values can be returned?

Functional Side Effects

- Because of the problems of side effects of functions that are called in expressions, parameters to functions should always be in-mode parameters.
- Ada functions can have only in-mode formal parameters.

- This effectively prevents a function from causing side effects through its parameters or through aliasing of parameters and global.
- In most languages, however, functions can have either pass-by-value or pass-by-reference parameters, thus allowing functions that cause side effects and aliasing.

Types of Returned Values

- C allows any type to be returned by its functions except arrays and functions.
- C++ is like C but also allows user-defined types, or classes, to be returned from its functions.
- JavaScript functions can be passed as parameters and returned from functions.

User-Defined Overloaded Operators

- Nearly all programming languages have overloaded operators.
- Users can further overload operators in C++ and Ada (Not carried over into Java)
- How much operator overloading is good, or can you have too much?

Co-routines

- A co-routine is a subprogram that has multiple entries and controls them itself.
- It is also called symmetric control.
- It also has the means to maintain their status between activation.
- This means that co-routines must be history sensitive and thus have static local vars.
- Secondary executions of a co-routine often begin at points other than its beginning.
- A co-routine call is named a resume.
- The first resume of a co-routine is to its beginning, but subsequent calls enter at the point just after the last executed statement in the co-routine.

Check your progress

Q1. What do you understand by subprogram?

Q2. Differentiate between Actual and Formal parameters.

2.5 Summary

In this unit you learnt about input files, encapsulation, information hiding, and sub program.

- In C programming, file is a place on disk where a group of related data is stored.
- If a large amount of numerical data it to be stored, text mode will be insufficient. In such case binary file is used.
- **Information hiding** is the principle of segregation of the *design decisions*
- The term *encapsulation* is often used interchangeably with information hiding.
- Information hiding serves as an effective criterion for dividing any piece of equipment, software or hardware, into modules of functionality.

2.6 Review Questions

Q1. What do you mean by input files? Explain their working in detail.

Q2. What is encapsulation? Explain with example.

Q3. What is the difference between encapsulation and information hiding?

Q4. What are sub-programs?

Q5. Write short note on Parameter passing method.

UNIT-III Data Types & Sequence Control

Structure

3.0 Introduction

This is the third unit of this block. It focused on data types and sequence control. There are six sections in this unit. In the Sec 3.1 we defined type definition. As you know the vast majority of the programming languages deal with typed values, i.e., integers, Booleans, real numbers, people, vehicles, etc. There are however, programming languages that have no types at all. These programming languages tend to be very simple. Good examples in this category are the core lambda calculus, and Brain Fuc. You will get detailed description about type definition in the first section. In Sec. 3.2 you will learn about data types. In Sec. 3.3 you will learn about abstract data types. In computer science, an abstract data type (ADT) is a mathematical model for data types where a data type is defined by its behavior (semantics) from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations. In Sec. 3.4 you will know about sequence control in programming language. In Sec. 3.5 and 3.6 you will find summary and review questions respectively.

Objectives

After studying this unit you should be able to:

- Define type definition, data types and abstract data types.
- Describe sequence control and Explicit and Implicit Sequence Control

3.1 Type Definition

The type definition facility allows us to define a new name for an existing data type. The general syntax is

```
typedef  data_type  new_name;
```

Here typedef is a keyword, data_type is any existing data type that may be a standard data type or a user defined type, new_name is an identifier, which is a new name for this data type. Note that we are not creating any new data type but we are only creating a new name for the existing data type. For example we can define a new name for int type by writing-

```
typedef int marks;
```

Now marks is a synonym for int and we can use marks of int anywhere in the program, for example-

```
marks sub1, sub2;
```

Here sub1, sub2 are actually int variable and are similar to any declared variable using int keyword. The above declaration is equivalent to- int sub1, sub2;

Some more examples are-

```
typedef usigned long int ulint;
```

```
typdef float real;
```

Here ulint is another name for type unsigned long int, and real is another name for float. The typedef declaration can be written wherever other declarations are allowed. We can give more than one name to a single data type using only one typedef statement, for example-

```
typedef int age, marks, units;
```

In the above typedef statement, we have defined three names for the data type int.

Since typdef is syntactically considered as a storage class, so we can’t include a storage class in typedef statement.for example statements of these types is invalid-

```
typedef static char schar;
```

```
typedef extern int marks;
```

3.2 Data Types

The vast majority of the programming languages deal with typed values, i.e., integers, Booleans, real numbers, people, vehicles, etc. There are however, programming languages that have no types at all. These programming languages tend to be very simple. Good examples in this category are the core lambda calculus, and Brain Fuc. There exist programming languages that have some very primitive typing systems. For instance, the x86 assemblies allow to store floating point numbers, integers and addresses into the same registers. In this case, the particular instruction used to process the register determines which data type is being taken into consideration. For instance, the x86 assembly has a subl instruction to perform integer subtraction, and another instruction, fsubl, to subtract floating point values. As another example, BCPL has only one data type, a word. Different operations treat each word as a different type. The most important question that we should answer now is "what is a data type". We can describe a data type by combining two notions:

- Values: a type is, in essence, a set of values. For instance, the boolean data type, seen in many programming languages, is a set with two elements: true and false. Some of these sets have a finite number of elements. Others are infinite. In Java, the integer data type is a set with 2^{32} elements; however, the string data type is a set with an infinite number of elements.

- Operations: not every operation can be applied on every data type. For instance, we can sum up two numeric types; however, in most of the programming languages, it does not make sense to sum up two booleans. In the x86 assembly, and in BCPL, the operations distinguish the type of a memory location from the type of others.

Types exist so that developers can represent entities from the real world in their programs. However, types are not the entities that they represent. For instance, the integer type, in Java, represents numbers ranging from -2^{31} to $2^{31} - 1$. Larger numbers cannot be represented. If we try to assign, say, 2^{31} to an integer in Java, then we get back -2^{31}. This happens because Java only allows us to represent the 31 least bits of any binary integer.

Types are useful in many different ways. Testimony of this importance is the fact that today virtually every programming language uses types, be it statically, be it at runtime. Among the many facts that contribute to make types so important, we mention:

- Efficiency: because different types can be represented in different ways, the runtime environment can choose the most efficient alternative for each representation.

- Correctness: types prevent the program from entering into undefined states. For instance, if the result of adding an integer and a floating point number is undefined, then the runtime environment can trigger an exception whenever this operation might happen.

- Documentation: types are a form of documentation. For instance, if a programmer knows that a given variable is an integer, then he or she knows a lot about it. The programmer knows, for example, that this variable can be the target of arithmetic operations. The programmer also knows much memory is necessary to allocate that variable. Furthermore, contrary to simple comments, that mean nothing to the compiler, types are a form of documentation that the compiler can check.

Types are a fascinating subject, because they classify programming languages along many different dimensions. Three of the most important dimensions are:

- Statically vs dynamically typed.
- Strongly vs weakly typed.
- Structurally vs nominally typed.

In any programming language there are two main categories of types: primitive and constructed. Primitive types are atomic, i.e., they are not formed by the combination of other types. Constructed, or composite types, as the name already says, are made of other types, either primitive or also composite.

3.3 Abstract Data Types

Some authors describe object-oriented programming as programming *abstract data types* and their relationships. Within this section we introduce abstract data types as a basic concept for object-orientation and we explore concepts used in the list example of the last section in more detail.

3.3.1 Handling Problems

The first thing with which one is confronted when writing programs is the *problem*. Typically you are confronted with ``real-life" problems and you want to make life easier by providing a program for the problem. However, real-life problems are nebulous and the first thing you have to do is to try to understand the problem to separate necessary from unnecessary details: You try to obtain your own abstract view, or *model*, of the problem. This process of modeling is called *abstraction* and is illustrated in Figure 3.1.

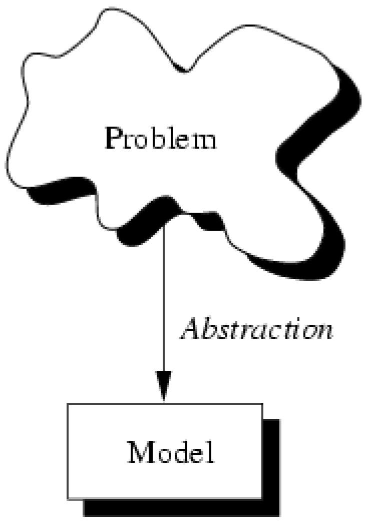


Figure 3.1: Create a model from a problem with abstraction.

The model defines an abstract view to the problem. This implies that the model focusses only on problem related stuff and that you try to define *properties* of the problem. These properties include

- the *data* which are affected and
- the *operations* which are identified

by the problem.

As an example consider the administration of employees in an institution. The head of the administration comes to you and ask you to create a program which allows administering the employees. Well, this is not very specific. For example, what employee information is needed by the administration? What tasks should be allowed? Employees are real persons who can be characterized with many properties; very few are:

- name,
- size,
- date of birth,
- shape,
- social number,
- room number,
- hair color,
- Hobbies.

Certainly not all of these properties are necessary to solve the administration problem. Only some of them are *problem specific*. Consequently you create a model of an employee for the problem. This model only implies properties which are needed to fulfill the requirements of the administration, for instance name, date of birth and social number. These properties are called the *data* of the (employee) model. Now you have described real persons with help of an abstract employee.

Of course, the pure description is not enough. There must be some operations defined with which the administration is able to handle the abstract employees. For example, there must be an operation which allows you to create a new employee once a new person enters the institution. Consequently, you have to identify the operations which should be able to be performed on an abstract employee. You also decide to allow access to the employees' data only with associated operations. This allows you to ensure that data elements are always in a proper state. For example you are able to check if a provided date is valid.

To sum up, abstraction is the structuring of a nebulous problem into well-defined entities by defining their data and operations. Consequently, these entities *combine* data and operations. They are **not** decoupled from each other.

3.3.2 Properties of Abstract Data Types

The example of the previous section shows, that with abstraction you create a well-defined entity which can be properly handled. These entities define the *data structure* of a set of items. For example, each administered employee has a name, date of birth and social number.

The data structure can only be accessed with defined *operations*. This set of operations is called *interface* and is *exported* by the entity. An entity with the properties just described is called an *abstract data type* (ADT).

Figure 3.2 shows an ADT which consists of an abstract data structure and operations. Only the operations are viewable from the outside and define the interface.

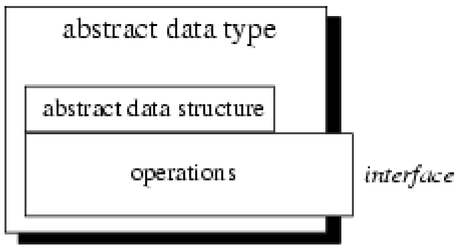


Figure 3.2: **An abstract data type (ADT).**

Once a new employee is ``created'' the data structure is filled with actual values: You now have an *instance* of an abstract employee. You can create as many instances of an abstract employee as needed to describe every real employed person.

Let's try to put the characteristics of an ADT in a more formal way:

Definition (Abstract Data Type) *An* ***abstract data type*** *(ADT) is characterized by the following properties:*

1. It exports a ***type****.*

2. It exports a ***set of operations****. This set is called* ***interface****.*

3. Operations of the interface are the one and only access mechanism to the type's data structure.

4. Axioms and preconditions define the application domain of the type.

With the first property it is possible to create more than one instance of an ADT as exemplified with the employee example. You might also remember the list example of unit 2. In the first version we have implemented a list as a module and were only able to use one list at a time. The second version introduces the ``handle'' as a reference to a ``list object''. From what we have learned now, the handle in conjunction with the operations defined in the list module defines an ADT *List*:

1. When we use the handle we define the corresponding variable to be of type *List*.

2. The interface to instances of type *List* is defined by the interface definition file.

3. Since the interface definition file does not include the actual representation of the handle, it cannot be modified directly.

4. The application domain is defined by the semantically meaning of provided operations. Axioms and preconditions include statements such as

- ``An empty list is a list."
- ``Let l=(d1, d2, d3, ..., dN) be a list. Then l.append (dM) results in l=(d1, d2, d3, ..., dN, dM)."
- ``The first element of a list can only be deleted if the list is not empty."

However, all of these properties are only valid due to our understanding of and our discipline in using the list module. It is in our responsibility to use instances of *List* according to these rules.

Importance of Data Structure Encapsulation

The principle of hiding the used data structure and to only provide a well-defined interface is known as *encapsulation*. Why is it so important to encapsulate the data structure?

To answer this question considers the following mathematical example where we want to define an ADT for complex numbers. For the following it is enough to know that complex numbers consists of two parts: *real part* and *imaginary part*. Both parts are represented by real numbers. Complex numbers define several operations: addition, subtraction, multiplication or division to name a few. Axioms and preconditions are valid as defined by the mathematical definition of complex numbers. For example, it exists a neutral element for addition.

To represent a complex number it is necessary to define the data structure to be used by its ADT. One can think of at least two possibilities to do this:

- Both parts are stored in a two-valued array where the first value indicates the real part and the second value the imaginary part of the complex number. If *x* denotes the real part and *y* the imaginary part, you could think of accessing them via array subscription: *x*=*c*[0] and *y*=*c*[1].
- Both parts are stored in a two-valued record. If the element name of the real part is *r* and that of the imaginary part is *i*, *x* and *y* can be obtained with: *x*=*c*.r and *y*=*c*.i.

Point 3 of the ADT definition says that for each access to the data structure there must be an operation defined. The above access examples seem to contradict this requirement. Is this really true?

Let's look again at the two possibilities for representing imaginary numbers. Let's stick to the real part. In the first version, *x* equals *c*[0]. In the second version, *x* equals *c*.r. In both cases *x* equals ``something". It is this ``something" which differs from the actual data structure used. But in both cases the performed operation ``equal" has the same meaning to declare *x* to be equal to the real part of the complex number *c*: both cases achieve the same semantics.

If you think of more complex operations the impact of decoupling data structures from operations becomes even clearer. For example the addition of two complex numbers requires you to perform an addition for each part. Consequently, you must access the value of each part which is different for each version. By providing an operation ``add" you can *encapsulate* these details from its actual use. In an application context you simply ``add two complex numbers" regardless of how this functionality is actually achieved.

Once you have created an ADT for complex numbers, say *Complex*, you can use it in the same way like well-known data types such as integers.

Let's summarize this: The separation of data structures and operations and the constraint to only access the data structure via a well-defined interface allows you to choose data structures appropriate for the application environment.

Generic Abstract Data Types

ADTs are used to define a new type from which instances can be created. As shown in the list example, sometimes these instances should operate on other data types as well. For instance, one can think of lists of apples, cars or even lists. The semantical definition of a list is always the same. Only the type of the data elements change according to what type the list should operate on.

This additional information could be specified by a *generic parameter* which is specified at instance creation time. Thus an instance of a *generic ADT* is actually an instance of a particular variant of the ADT. A list of apples can therefore be declared as follows:

```
List<Apple> listOfApples;
```

The angle brackets now enclose the data type for which a variant of the generic ADT *List* should be created. *ListOfApples* offers the same interface as any other list, but operates on instances of type *Apple*.

Notation

As ADTs provide an abstract view to describe properties of sets of entities, their use is independent from a particular programming language. Each ADT description consists of two parts:

- **Data**: This part describes the structure of the data used in the ADT in an informal way.
- **Operations**: This part describes valid operations for this ADT, hence, it describes its interface. We use the special operation **constructor** to describe the actions which are to be performed once an entity of this

ADT is created and **destructor** to describe the actions which are to be performed once an entity is destroyed. For each operation the provided *arguments* as well as *preconditions* and *post-conditions* are given.

As an example the description of the ADT *Integer* is presented. Let *k* be an integer expression:

ADT *Integer* is

Data

A sequence of digits optionally prefixed by a plus or minus sign. We refer to this signed whole number as *N*.

Operations

Constructor

Creates a new integer.

add(k)

Creates a new integer which is the sum of *N* and *k*.

Consequently, the *post-condition* of this operation is *sum* = *N*+*k*. Don't confuse this with assign statements as used in programming languages! It is rather a mathematical equation which yields ``true" for each value *sum*, *N* and *k* after *add* has been performed.

sub(k)

Similar to *add*, this operation creates a new integer of the difference of both integer values. Therefore the post-condition for this operation is *sum* = *N*-*k*.

set(k)

Set *N* to *k*. The post-condition for this operation is *N* = *k*.

...

end

The description above is a *specification* for the ADT *Integer*. Please notice, that we use words for names of operations such as ``add". We could use the more intuitive ``+" sign instead, but this may lead to some confusion: You must distinguish the operation ``+" from the mathematical use of ``+" in the post-condition. The name of the operation is just *syntax* whereas the *semantics* is described by the associated pre- and post-conditions. However, it is always a good idea to combine both to make reading of ADT specifications easier.

Real programming languages are free to choose an arbitrary *implementation* for an ADT. For example, they might implement the operation *add* with the infix operator ``+" leading to a more intuitive look for addition of integers.

Abstract Data Types and Object-Orientation

ADTs allow the creation of instances with well-defined properties and behavior. In object-orientation ADTs are referred to as *classes*. Therefore a class defines properties of *objects* which are the instances in an object-oriented environment.

ADTs define functionality by putting main emphasis on the involved data, their structure, operations as well as axioms and preconditions. Consequently, object-oriented programming is ``programming with ADTs'': combining functionality of different ADTs to solve a problem. Therefore instances (objects) of ADTs (classes) are dynamically created, destroyed and used.

Check your progress

Q1. Define type definition.

Q2. Differentiate between data types and abstract data types.

3.4 Sequence Control

Control structures in a programming language provide the basic framework within which operations and data are combined into programs and sets of programs. To this point, we have been concerned with data and operations in isolation. Now we must consider their organization into complete executable programs. This involves two aspects: control of the order of execution of the operations, both primitive and user defined which we term sequence control.

3.4.1 Implicit and Explicit Sequence Control

Sequence-control structures may be conveniently categorized into four groups:

1. Expressions form the basic building blocks for statements and express how data are manipulated and changed by a program. Properties such as precedence rules and parentheses determine how expressions become evaluated.

2. Statements or groups of statements, such as conditional and iteration statements, determine how control flows from one segment of a program to another.

3. Declarative programming is an execution model that does not depend on statements, but nevertheless causes execution to proceed through a program. The logic programming model of Prolog is an example of this.

4. Subprograms: - such as subprogram calls and coroutines, form a way to transfer control from one segment of a program to another. This division is

necessarily somewhat imprecise. For example, some languages such as LISP and APL have no statements, only expressions, yet versions of the usual statement sequence-control mechanisms are used. Sequence-control structures may be either implicit or explicit. Implicit (or default) sequence-control structures are those defined by the language to be in effect unless modified by the programmer through some explicit structure. For example, most languages define the physical sequence of statements in a program as controlling the sequence in which statements are executed, unless modified by an explicit sequence-control statement. Within expressions there is also commonly a language-defined hierarchy of operations that controls the order of execution of the operations in the expression when parentheses are absent. Explicit sequence-control structures are those that the programmer may optionally use to modify the implicit sequence of operations defined by the language (e.g., by using parentheses within expressions or goto statements and statement labels).

3.5 Summary

In this unit you learnt about Type definition and abstract data types, Implicit and explicit sequence control. All these are very important concept in programming languages.

- Types exist so that developers can represent entities from the real world in their programs
- Process of modeling is called *abstraction*
- To represent a complex number it is necessary to define the data structure to be used by its ADT.
- ADTs allow the creation of instances with well-defined properties and behavior.

3.6 Review Questions

Q1. What is type def.? Explain in details.

Q2. What are data types? How many data types available in programming languages?

Q3. What are abstract data types? How differ these from data type?

Q4. What is a sequence? Is it necessary to use a sequence in our program? Elaborate your answer.

Q5. What is the difference between implicit and explicit sequence?

UNIT-IV Exception Handlers & Co-routines

Structure

4.0 Introduction

This is the last unit of this block. This unit focused on Exception Handlers & Co-routines. There are seven sections in this unit. In Sec. 4.1 you will learn about Subprogram sequence control. In the subprogram sequence control we can define it as a program is composed of a single main program, which during execution may call various subprograms, variables procedures, which in turn may each call other sub-sub-programs and so forth to any depth. When a subprogram is start execution, the execution of its parent (calling) program stops temporarily and will start its execution after completion of its subprogram from the same point it stops execution. This means a subprogram will work like a copy statement. Instead of pasting the statement we pass the execution to other place. In Sec. 4.2 you will know about recursive sub programs. A *recursive* program is one that is defined in terms of itself. There are many mathematical functions and common data processing algorithms that are more elegant and easier to understand when defined recursively. In Sec. 4.3 you will learn about exception and its handlers. In Sec. 4.4 we define co-routines. In the Sec. 4.5 we explain scheduled sub program. In Sec. 4.6 and 4.7 you will find summary and review questions respectively.

Objectives

After studying this unit you should be able to:

- Define Subprogram sequence control, Recursive sub-programs
- Define Exception and exception handlers, Co-routines and Scheduled subprograms

4.1 Subprogram Control

Subprogram control: interaction among subprograms and how subprograms manage to pass data among themselves in a structured and efficient manner.

Terminology:

Function call – subprograms that return values directly
Subroutine call – subprograms that operate only through side effects on shared data.

A. **Subprogram Sequence Control**

Simple subprogram call return

Copy rule view of subprograms: the effect of a call statement is the same as if the subprogram were copied and inserted into the main program.

Implicit assumptions present in this view :

- Subprograms cannot be recursive
- Explicit call statements are required
- Subprograms must execute completely at each call
- Immediate transfer of control at point of call
- Single execution sequence

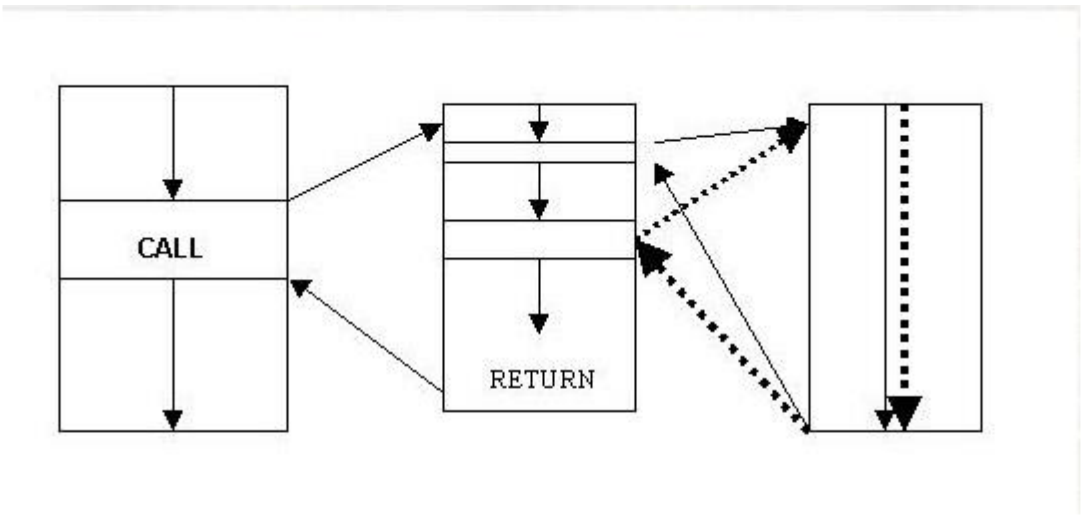


Figure 4-0

1. **Simple call-return subprograms**

Execution of subprograms
Outline of the method:

1. Subprogram definition and subprogram activation.

   The definition is translated into a template, used to **create an activation** each time a subprogram is called.

2. Subprogram activation: consists of

   - a code segment (the invariant part) - executable code and constants
   - an activation record (the dynamic part) - local data, parameters

   Created anew each time the subprogram is called, destroyed when the subprogram returns.

Execution is implemented by the use of two system-defined pointers:

- **Current-instruction pointer** – CIP-address of the next statement to be executed
- **Current-environment pointer** – CEP- pointer to the activation record.

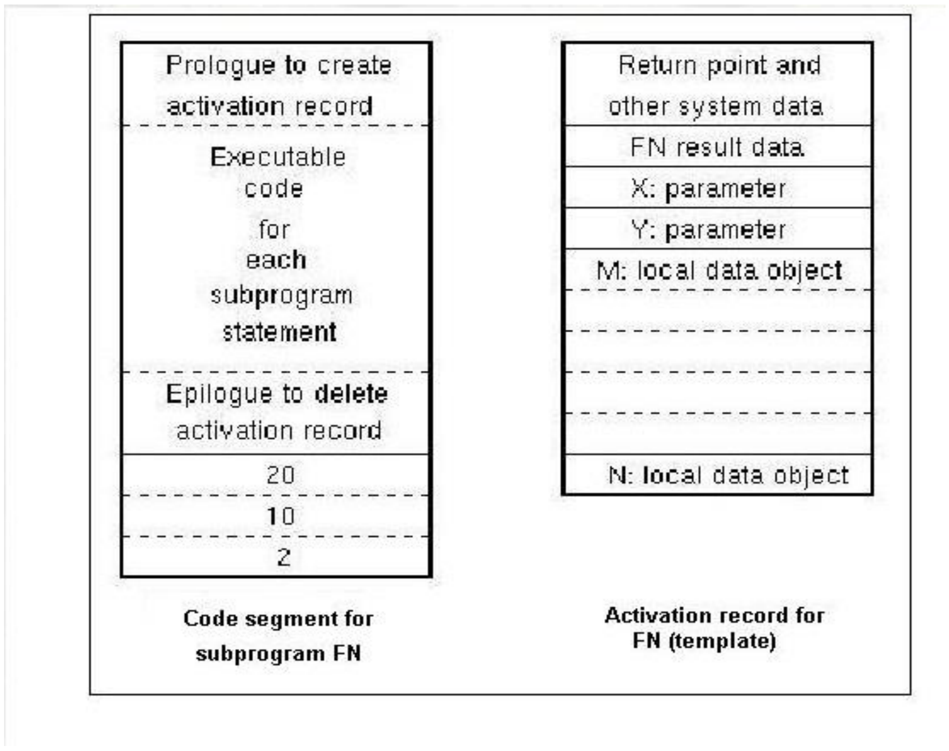

Prologue to create activation record
Executable code for each subprogram statement
Epilogue to delete activation record
20
10
2
Code segment for subprogram FN
Return point and other system data
FN result data
X: parameter
Y: parameter
M: local data object
N: local data object
Activation record for FN (template)

Figure 4-2

On **call** instruction:

- An activation record is created.
- Current CIP and CEP are saved in the created activation record as *return point*
- CEP is assigned the address of the activation record.
- CIP gets the address of the first instruction in the code segment
- The execution continues from the address in CIP

On **return**

- The old values of CIP and CEP are retrieved.
- The execution continues from the address in CIP

Restrictions of the model: **at most one activation** of any subprogram

The simplest implementation: to allocate storage for each activation as an extension of the code segment. Used in FORTRAN and COBOL. The activation record is not destroyed - only reinitialized for each subprogram execution.

Hardware support - CIP is the program counter, CEP is not used, simple jump executed on return.

Stack-based implementation - the simplest run-time storage management technique

call statements : push CIP and CEP
return statements : pop CIP and CEP off of the stack.

Used in most C implementations
LISP: uses the stack as an environment.

4.2 Recursive subprograms

Specification

Syntactically - no difference
semantically - multiple activations of the same subprogram exist simultaneously at some point in the execution.

Implementation

Stack-based - CIP and CEP are stored in stack, forming a dynamic chain of links.

- A new activation record is created for each call and destroyed at return.
- The lifetimes of the activation records cannot overlap - they are nested.

Some language compilers (C, Pascal) always assume recursive structure of subprograms,
while in others non-recursive subprograms are implemented in the simple way.

B. **Attributes of data control**

Data control features: determine the accessibility of data at different points during program execution.

Central problem: the meaning of variable names,
i.e. the correspondence between names and memory locations.

- **Names and referencing environments**

  - Two ways to make a data object available as an operand for an operation.
  - **Direct transmission** – A data object computed at one point as the result of
    an operation may be directly transmitted to another operation as an operand
    - **Example:** x = y + 2*z;
      The result of multiplication is transmitted directly as an operand of the addition operation.
  - **Referencing through a named data object** –
    A data object may be given a name when it is created, and the name may then
    be used to designate it as an operand of an operation.

- **Program elements that may be named**
    - Variables
    - Formal parameters
    - Subprograms
    - Defined types
    - Defined constants
    - Labels
    - Exception names
    - Primitive operations
    - Literal constants

Names from 4 thru 9 - resolved at translation time.
Names 1 thru 3 - discussed below.

Simple names: identifiers, e.g. *var1*.
Composite names: names for data structure components, e.g. *student[4].last_name*.

- **Associations and Referencing Environments**

  Association: binding identifiers to particular data objects and subprograms
  Referencing environment: the set of identifier associations for a given subprogram.
  Referencing operations during program execution: determine the particular data object
  or subprogram associated with an identifier.

Local referencing environment:

The set of associations created on entry to a subprogram that represents formal parameters, local variables, and subprograms defined only within that subprogram

Nonlocal referencing environment:

The set of associations for identifiers that may be used within a subprogram
but that are not created on entry to it can be global or predefined.

Global referencing environment: associations created at the start of execution
of the main program, available to be used in a subprogram,

Predefined referencing environments: predefined association in the language definition.

Visibility of associations

Associations are visible if they are part of the referencing environment. Otherwise associations are hidden

Dynamic scope of associations

The set of subprogram activations within which the association is visible

- **Aliases for data objects**: Multiple names of a data object
    - separate environments - no problem
    - in a single referencing environment - called aliases.
    - Problems with aliasing
    - Can make code difficult to understand for the programmer.
    - Implementation difficulties at the optimization step - difficult to spot interdependent statements - not to reorder them
- **Static and dynamic scope**

The **dynamic scope of an association** for an identifier is that set of subprogram activations in which the association is visible during execution.

Dynamic scope rules

Relate references with associations for names during program execution.

The **static scope of a declaration** is that part of the program text where a use of the identifier is a reference to that particular declaration of the identifier.

Static scope rules

Relate references with declarations of names in the program text.

Importance of static scope rules - recording information about a variable during translation.

- **Block structure**

  Block-structured languages (Pascal):

  - Each program or subprogram is organized as a set of nested blocks.
  - The chief characteristic of a block is that it introduces a new local referencing environment.

  Static scope rules for block-structured programs

  - **Local data and local referencing environments**

    Local environment of a subprogram: various identifiers declared in the subprogram -
    variable names, parameters, subprogram names.

Static scope rules: implemented by means of a table of the local declarations
Dynamic scope rules: two methods:

- **Retention** - associations and the bound values are retained after execution.
- **Deletion** - associations are deleted.
  (For further explanation and example see Figure 9.9 on p. 369)

Implementation of dynamic scope rules in local referencing environments: by means of a local environment table to associate names, types and values.

Retention: the table is kept as part of the code segment
Deletion: the table is kept as part of the activation record, destroyed after each execution.

4.3 Exception and Exception Handling

Exception

An arithmetic exception arises when an attempted atomic arithmetic operation has no result that would be acceptable universally. The meanings of atomic and acceptable vary with time and place.

For example, an exception arises when a program attempts to take the square root of a negative number. (This example is one case of an *invalid operation* exception.) When such an exception occurs, the system responds in one of two ways:

- If the exception's trap is disabled (the default case), the system records the fact that the exception occurred and continues executing the program using the default result specified by IEEE 754 for the accepting operation.

- If the exception's trap is enabled, the system generates a SIGFPE signal. If the program has installed a SIGFPE signal handler, the system transfers control to that handler; otherwise, the program aborts.

IEEE 754 defines five basic types of floating point exceptions: *invalid operation, division by zero, overflow, underflow* and *inexact*. The first three (invalid, division, and overflow) are sometimes collectively called *common exceptions*. These exceptions can seldom be ignored when they occur. ieee_handler (3m) gives an easy way to trap on common exceptions only. The other two exceptions (underflow and inexact) are seen more often--in fact, most floating point operations incur the inexact exception--and they can usually, though not always, be safely ignored.

TABLE 4-1 condenses information found in IEEE Standard 754. It describes the five floating point exceptions and the default response of an IEEE arithmetic environment when these exceptions are raised.

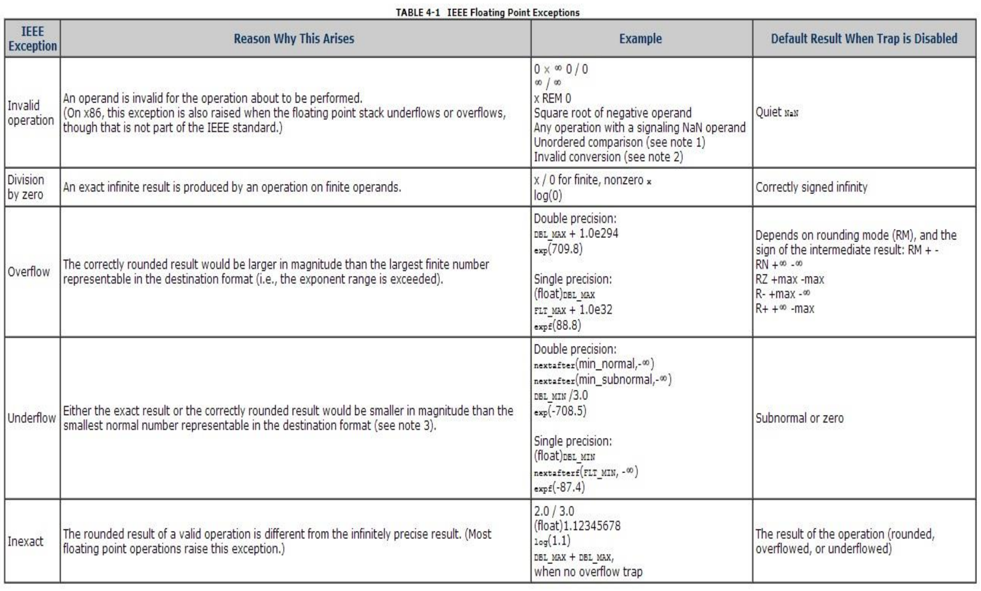


TABLE 4-1 IEEE Floating Point Exceptions

IEEE Exception	Reason Why This Arises	Example	Default Result When Trap is Disabled
Invalid operation	An operand is invalid for the operation about to be performed. (On x86, this exception is also raised when the floating point stack underflows or overflows, though that is not part of the IEEE standard.)	0 × ∞ 0 / 0 ∞ / ∞ x REM 0 Square root of negative operand Any operation with a signaling NaN operand Unordered comparison (see note 1) Invalid conversion (see note 2)	Quiet NaN
Division by zero	An exact infinite result is produced by an operation on finite operands.	x / 0 for finite, nonzero x log(0)	Correctly signed infinity
Overflow	The correctly rounded result would be larger in magnitude than the largest finite number representable in the destination format (i.e., the exponent range is exceeded).	Double precision: DBL_MAX + 1.0e294 exp(709.8) Single precision: (float)DBL_MAX FLT_MAX + 1.0e32 expf(88.8)	Depends on rounding mode (RM), and the sign of the intermediate result: RM + - RN +∞ -∞ RZ +max -max R- +max -∞ R+ +∞ -max

Underflow	Either the exact result or the correctly rounded result would be smaller in magnitude than the smallest normal number representable in the destination format (see note 3).	Double precision:
		nextafter(min_normal,-∞)
		nextafter(min_subnormal,-∞)
		DBL_MIN /3.0
		exp(-708.5)
		
		Single precision:
		(float)DBL_MIN
		nextafterf(FLT_MIN, -∞)
		expf(-87.4)
			Subnormal or zero

Inexact	The rounded result of a valid operation is different from the infinitely precise result. (Most floating point operations raise this exception.)	2.0 / 3.0 (float)1.12345678 log(1.1) DBL_MAX + DBL_MAX, when no overflow trap	The result of the operation (rounded, overflowed, or underflowed)

Exception Handling

Exception handling is a means of taking the handling of interruptions from the operating system to the level of the program.

Exceptions are events that cause interruptions. Exception handling is an activity performed by the program if an exception occurs. The exception handler is the part of the program that runs after the occurrence of a given exception as a reaction to the event.

Exception handling makes event control possible at the level of programs. Similarly to the way in which certain interruptions may be masked at the operating system level, it is also possible to mask exceptions at the language level by making the monitoring of certain exceptions disabled or enabled. Disabling the monitoring of an exception is the simplest form of exception handling: an event causes an interruption, which propagates to the level of the program raising an exception, of which the program takes no notice and continues running. The effect of the unhandled exception on the further functioning of the program is completely unknown, it is possible that the program cannot recover from the exception at all, or its operation will be corrupted.

Exceptions usually have a name (which usually plays the role of the message describing the event) and a code (an integer).

Exception handling was introduced first in PL/I and later in Ada, too. However, the two languages have vastly different principles concerning the details of the exception handling mechanism. According to PL/I, the reason why an exception was raised is that the algorithm implemented by the program has not been prepared to handle an exceptional situation, therefore it is the cause of the exceptional event that needs to be managed. The exception handler extinguishes the extraordinary situation, and returns to the normal functioning of the program. The program continues from the point where the exception was thrown.

As opposed to this, Ada claims that exceptional situations cannot be “cured”; the original activity should be abandoned. Exception handlers in Ada perform activities which are adequate substitutes to the original event, and do not return to the point where the exception was thrown.

Languages differ in the following aspects of exception handling:

1. What kind of built-in exceptions are in the language?

2. Can the programmer define custom exceptions?

3. What kind of scoping rules does the exception handler have?

4. Is it possible to bind exception handlers to programming elements (expressions, statements, or program units)?

5. How does the program continue after exception handling?

6. What happens if an exception occurs in the exception handler?

7. Are there built-in exception handlers in the language?

8. Is it possible to write a general exception handler that can handle all kinds of exceptions?

9. Can the exception handler be parameterized?

There are no parameterized and built-in exception handlers in PL/I and Ada.

Exception Handling in PL/I

PL/I provide the following built-in exceptions:

CONVERSION	**conversion error**
FIXEDOVERFLOW	**fixed point overflow**
OVERFLOW	**floating-point overflow**
UNDERFLOW	**floating-point underflow**
ZERODIVIDE	**division by zero**
SIZE	**size error**
SUBSCRIPTRANGE	**index out of bounds**
STRINGRANGE	
STRINGSIZE	
CHECK[(identifier)]	**trace**
AREA	**addressing error**
ATTENTION	**external interrupt**
FINISH	**regular ending of the program**
ENDFILE(file_name)	**end of the file**
ENDPAGE(file_name)	**end of the page**
KEY(file_name)	**key error**
NAME(file_name)	
RECORD(file_name)	
TRANSMIT(file_name)	
UNDEFINEDFILE(file_name)	

PENDING(file_name)	
ERROR	**general exception**

The programmer may declare custom exceptions of the form CONDITION(name).

The first five built-in exceptions are monitored by default but can be disabled if required. The following five exception are disabled by default but can be enabled. The rest of the built-in exceptions and all programmer-defined exceptions are always enabled and cannot be disabled.

Every statement can be preceded by a rule that specifies whether the monitoring of exceptions possibly thrown by the statement should be disabled or enabled. Any number of exceptions separated by commas can be enclosed in round brackets

(exception_name [, exception_name...]):statement

to indicate that the given exceptions should be monitored.

To disable the monitoring of an exception, the keyword NO must precede the name of the exception, as in the following example:

(NOZERODIVIDE, SIZE):IF ...

If the monitoring rule is placed before the first statement of a block or a subprogram its scope spans the entire program unit including the contained program units as well. It is possible to override the rule by individual statements of the program unit. If the rule precedes a statement that contains an expression, the rule applies to the expression alone rather than the entire statement. If the statement contains no expressions, the rule applies to the entire statement.

The statement SIGNAL exception_name; is used to throw an exception explicitly. This is the only way of throwing programmer-defined exceptions.

Exception handlers in PL/I have the following syntax:

ON exception_name executable_statement;

where blocks may stand in for executable_statement.

Exception handlers can be placed anywhere in the source code.

The scope of an exception handler lasts from the moment the control reaches it until:

- control reaches another exception handler intended to handle the same exception, which overrides the effect of the previous handler;

- control reaches the statement REVERT exception_name; which undoes the effect of the last ON statement;
- or the termination of the program unit

Including every program unit called from inside the scope of the exception handler.

It follows that exception handlers have dynamic scoping.

If an exception is raised in a programming unit, the run-time system examines whether the monitoring of the given exception is enabled or not. If monitoring is disabled, the execution of the program unit continues. If monitoring is enabled, the run-time system examines the availability of an exception handler which contains the name of the given exception. If such an exception handler is found, the exception handler is executed. If the exception handler contains a GOTO statement, the program continues on the statement with the given label. If there is no GOTO statement, control returns either to the statement in which the exception occurred or to the following statement, depending on the nature of the exception. For example, in the case of a CONVERSION exception, the same statement which caused the error will be executed again; in the case of arithmetical errors and custom exceptions, the program continues with the statement immediately following the one which threw the exception.

If a proper named exception handler is not available in the given program unit, control steps back on the call chain and continues looking for appropriate handlers in the caller. If none of the calling units contain an appropriate handler, the exception ERROR is raised. Now it is the ERROR which needs to be handled. The ON ERROR exception handler is used to manage all sorts of unnamed exceptions; it is the general exception handler of PL/I. If no general exception handler is specified, control returns to the operating system. The program failed to handle the given exception.

To assist the programmer in handling exceptions, PL/I provides special built-in functions which can be called only from exception handlers. The ON functions have no parameters, and are used to specify the event, place or cause of exceptions.

The following are examples of exception handler functions:

- ONCODE: Returns the unique code of the error; useful for handling built-in exceptions (e.g. KEY) which name event groups, in which case the individual events can only be identified via their code.
- ONCHAR: If a conversion error is caught, the function returns the character which caused the error during data transfer. Since the function behaves as a pseudo variable (i.e. it can be assigned a value), it is possible to replace the troublesome character; as a result, the I/O operation may be repeated.
- ONKEY: Returns the primary key of the record which caused an I/O error.

- ONLOCK: Returns the name of the subprogram in which the exception was raised.

The dynamic scoping of exception handlers sometimes cause problems, While names have static scoping, exception handlers support dynamic scoping, this may lead to conflicts. The called program unit inherits the effects of the exception handler activated in the caller program unit. This is considered dangerous, because it may cause non-local jumps. If the exception is not handled by the same program unit in which it occurs, another exception handler may be activated which is located considerably earlier in the call chain, and this conduct may be entirely wrong.

Although programmer-defined exceptions are of a great benefit for testing purposes, they are not considered effective during run-time.

Example:

In PL/I, sequential files may be processed in the following way:

```
DECLARE F FILE;

1 S,

2 ID PICTURE ’’9999,

3 OTHERS CHARACTER(91),

EOF BIT(1) INIT’’(0B);

ON ENDFILE(F) EOF’’=1B;

OPEN FILE(F);

READ FILE(F) INTO(S);

DO WHILE:(EOF);

...

READ FILE(F) INTO(S);

END;
```

Exception Handling in Ada

The built-in exceptions of Ada generally name groups of events. These are the following:

- CONSTRAINT_ERROR: The event group that occurs if a declaration constraint is violated, for example, an index bound is exceeded.
- NUMERIC_ERROR: Arithmetic errors, including underflow and overflow errors, division by zero, etc.
- STORAGE_ERROR: Memory errors (including all allocation-related problems): the memory region referred to is not available.
- TASKING_ERROR: Rendezvous is not possible with the given task.

- SELECT_ERROR: SELECT statement error.

Every exception is monitored by default, but the monitoring of certain events (especially checks) can be disabled with the pragma

SUPPRESS:

SUPPRESS(name [,ON => { object_name | type }])

where name is the name of the event to be disabled; identifies a single event, and does not correspond to built-in exception names.

A name can take the following values:

- ACCESS_CHECK: checking address;

- DISCRIMINANT_CHECK: checking the discriminant of a record;

- INDEX_CHECK: index checking;

- LENGTH_CHECK: length checking;

- RANGE_CHECK: range checking;

- DIVISION_CHECK: checking division by zero;

- OVERFLOW_CHECK: overflow checking;

- STORAGE_CHECK: checking the availability of storage.

The object_name identifies a programming object (e.g. a variable). If the optional part is not specified, monitoring the event is disabled in the entire program. If, however, the optional part is present, monitoring is disabled only on objects of the type or of the object_name specified. Custom exceptions can be declared with the EXCEPTION attribute. Exception handlers can be placed after the body of every program unit, right before the closing END. Exception handlers have the following syntax:

EXCEPTION

WHEN exception_name [,exception_name]... => statements

[WHEN exception_name [,exception_name...] => statements]...

[WHEN OTHERS => statements]

The statements part comprises any number and kind of executable statements. An exception handler may include any number of WHEN branches, but at least one is mandatory. The WHEN OTHERS branch may occur only once, and it must be the last branch. The WHEN OTHERS branch is used to handle unnamed exceptions (i.e. it is the general exception handler of Ada).

The exception handler is accessible to the entire program unit and those program units which have been called from it, unless they specify their own exception handlers. In Ada, exception handlers have dynamic scoping, which is inherited via the call chain.

The statement RAISE exception_name; is used to throw exceptions of any type. Programmer-defined exceptions can only be raised in this way.

If an exception is raised in a program unit, the run-time system first examines whether the monitoring of the given exception is disabled or not. If monitoring is disabled, the execution of the program continues; otherwise, the program unit terminates. Then the run-time system examines the availability of exception handlers within the program unit. If an exception handler is found, the run-time system examines if the handler contains a WHEN branch which names the given exception. If one of the branches satisfies this condition, the run-time system executes the statements contained therein. If the statements include a GOTO statement the program continues on the statement with the given label; otherwise the program continues as if the program unit terminated regularly. If none of the branches match the name of the exception the run-time system examines the availability of a WHEN OTHERS branch. If the WHEN OTHERS branch is specified, the statements contained therein are executed subject to the same rules as in the previous case. If none of branches match and no WHEN OTHERS branch is specified or the unit does not contain an exception handler at all, the program unit propagates the exception. This means that the exception is raised by the program unit call, and the process of looking for an appropriate handler is repeated. The run-time system keeps searching for a proper exception handler up the call chain. If control reaches the beginning of the call chain without having found a proper exception handler the program does not handle the exception and control is transferred to the operating system.

Exceptions thrown in the exception handler are propagated immediately.

The RAISE; statement can be used to propagate the exception only in exception handlers.

Exceptions thrown in declaration statements are propagated immediately.

Exceptions thrown in nested packages are propagated to the containing unit, while exceptions thrown in compilation-level packages abort the main program.

The dynamic scopes of exception handlers raise the same problems as in PL/I, the only difference being that the program units on the call chain terminate regularly. The Ada compiler cannot check the operation of exception handlers.

Custom exceptions have an essential role in Ada programs as they provide a means of communication between program units via event control.

Example:

```
function FACT(N : natural) return float is

begin

if N=1 then return 1.0;

else return float(N)*FACT(N-1);
```

```
end if;
exception
when NUMERIC_ERROR => return FLOAT_MAX;
end;
```

If the function is called with a value too large to calculate its factorial, an overflow error occurs, which is caught by the NUMERIC_ERROR exception handler such that the function returns the largest floating-point value.

Check your progress

Q1. Define recursive sub program.

Q2. Differentiate between local and non-local referencing environment.

Q3. Explain exception. How we can handle an exception? List all built-in exceptions.

4.4 Co-routines

In computer science, coroutines are program components that generalize subroutines to allow multiple entry points and suspending and resuming of execution at certain locations. Coroutines are well-suited for implementing more familiar program components such as cooperative tasks, iterators, infinite lists and pipes.

The term "coroutine" was originated by Melvin Conway in his seminal 1963 paper.

Comparison with subroutines

Coroutines are more generic than subroutines. The lifespan of subroutines is dictated by last in, first out (the last subroutine called is the first to return); in contrast, the lifespan of coroutines is dictated entirely by their use and need.

The start of a subroutine is the only point of entry. Subroutines can return only once; in contrast, coroutines can return (yield) several times. The start of a coroutine is the first point of entry and subsequent points of entry are following yield commands. Practically, yielding returns the result to the calling coroutine and gives it back control, like a usual subroutine. However, the next time the coroutine is called, the execution does not start at the beginning of the coroutine but just after the yield call.

Here is a simple example of how coroutines can be useful. Suppose you have a consumer-producer relationship where one routine creates items and adds them to a queue and another removes items from the queue and uses them. For reasons of efficiency, you want to add and remove several items at once. The code might look like this:

var q := new queue coroutine produce loop while q is not full create some new items add the items to q yield to consume coroutine consume loop while q is not empty remove some items from q use the items yield to produce

The queue is then completely filled or emptied before yielding control to the other coroutine using the yield command. The further coroutines calls are starting right after the yield, in the inner coroutine loop.

Although this example is often used to introduce multithreading, it is not necessary to have two threads to achieve this: the yield statement can be implemented by a branch directly from one routine into the other.

Detailed comparison

Since coroutines can have more points of entry and exit than subroutines, it is possible to implement any subroutine as a coroutine. "Subroutines are special cases of ... coroutines." —Donald Knuth

Each time a subroutine is called (invoked), execution starts at the beginning of the invoked subroutine. Likewise, the first time a coroutine is invoked, execution starts at the beginning of the coroutine; however, each subsequent time a coroutine is invoked, execution resumes following the place where the coroutine last returned (yielded).

A subroutine returns only once. In contrast, a coroutine can return multiple times, making it possible to return additional values upon subsequent calls to the coroutine. Coroutines in which subsequent calls yield additional results are often known as generators.

Subroutines only require a single stack that can be preallocated at the beginning of program execution. In contrast, coroutines, able to call on other coroutines as peers, are best implemented using continuations. Continuations may require allocation of additional stacks and therefore are more commonly implemented in garbage-collected high-level languages. Coroutine creation can be done cheaply by preallocating stacks or caching previously allocated stacks.

Coroutines and generators

Generators are also a generalization of subroutines, but with at first sight less expressive power than coroutines; since generators are primarily used to simplify the writing of iterators, the yield statement in a generator does not specify a coroutine to jump to, but rather passes a value back to a parent routine. However, it is still possible to implement coroutines on top of a generator facility, with the aid of a top-level dispatcher routine that passes control explicitly to child generators identified by tokens passed back from the generators:

var q := new queue generator produce loop while q is not full create some new items add the items to q yield consume generator consume loop while q is not empty remove some items from q use the items yield produce subroutine dispatcher var d := new dictionary〈generator → iterator〉 d[produce] := start produce d[consume] := start consume var current := produce loop current := next d[current]

A number of implementations of coroutines for languages with generator support but no native coroutines (e.g. Python prior to 2.5) use this or a similar model.

Common uses of co routines

Coroutines are useful to implement the following:

- State machines within a single subroutine, where the state is determined by the current entry/exit point of the procedure; this can result in more readable code.
- Actor model of concurrency, for instance in computer games. Each actor has its own procedures (this again logically separates the code), but they voluntarily give up control to central scheduler, which executes them sequentially (this is a form of cooperative multitasking).
- Generators and these are useful for input/output and for generic traversal of data structures.

Programming languages supporting coroutines

- Icon
- High Level Assembly
- ChucK
- Io
- Limbo
- Lua
- µC++
- Modula-2
- Simula-67
- Python (since 2.5)
- Scheme
- Stackless Python
- Squirrel
- SuperCollider

- MiniD
- Aikido
- Ruby (since 1.9, using Fibers)
- Dynamic C
- BETA

Since continuations can be used to implement coroutines, programming languages that support them can also quite easily support coroutines.

Coroutine alternatives and implementations

Coroutines originated as an assembly-language technique, but are supported in some high-level languages, Simula and Modula-2 being two early examples.

As of 2003, many of the most popular programming languages, including C and its derivatives, do not have direct support for coroutines within the language or their standard libraries. (This is, in large part, due to the limitations of stack-based subroutine implementation).

In situations where a coroutine would be the natural implementation of a mechanism, but is not available, the typical response is to create a subroutine that uses an ad-hoc assemblage of boolean flags and other state variables to maintain an internal state between calls. Conditionals within the code result in the execution of different code paths on successive calls, based on the values of the state variables. Another typical response is to implement an explicit state machine in the form of a large and complex switch statement. Such implementations are difficult to understand and maintain.

Threads (and to a lesser extent fibers) are an alternative to coroutines in mainstream programming environments today. Threads provide facilities for managing the real-time cooperative interaction of "simultaneously" executing pieces of code. Threads are widely available in environments that support C (and are supported natively in many other modern languages), are familiar to many programmers, and are usually well-implemented, well-documented and well-supported. However, as they solve a large and difficult problem they include many powerful and complex facilities and have a correspondingly difficult learning curve. As such, when a coroutine is all that is needed, using a thread can be overkill.

One important difference between threads and coroutines is that threads are typically preemptively scheduled while coroutines are not. Because threads can be rescheduled at any instant and can execute concurrently, programs using threads must be careful about locking. In contrast, because coroutines can only be rescheduled at specific points in the program and do not execute concurrently, programs using coroutines can often avoid locking entirely. (This property is also cited as a benefit of event-driven or asynchronous programming.)

Since fibers are cooperatively scheduled, they provide an ideal base for implementing coroutines above. However, system support for fibers is often lacking compared to that for threads.

Implementation in the .NET Framework as fibers

During the development of the .NET Framework 2.0, Microsoft extended the design of the CLR hosting APIs to handle fiber-based scheduling with an eye towards its used in fiber-mode for SQL server. Prior to release, support for the task switching hook ICLRTask::SwitchOut was removed due to time constraints. Consequently the use of the fiber API to switch tasks is currently not a viable option in the .NET framework.

Implementations for C

Several attempts have been made, with varying degrees of success, to implement coroutines in C with combinations of subroutines and macros. Simon Tatham's contribution is a good example of the genre, and his own comments provide a good evaluation of the limitations of this approach. The use of such a device truly can improve the writ ability, readability and maintainability of a piece of code, but is likely to prove controversial. If your employer fires you for using this trick, tell them that repeatedly as the security staff drags you out of the building."

A more reliable approach to implementing coroutines in C is to give up on absolute portability and write processor-family-specific implementations, in assembly, of functions to save and restore a coroutine context. The standard C library includes functions named setjmp and longjmp which can be used to implement a form of coroutine. Unfortunately, as Harbison and Steele note, "the setjmp and longjmp functions are notoriously difficult to implement, and the programmer would do well to make minimal assumptions about them." What this means is if Harbison and Steele's many cautions and caveats are not carefully heeded, uses of setjmp and longjmp that appear to work in one environment may not work in another. Worse yet, faulty implementations of these routines are not rare. Indeed, setjmp/longjmp, because it only countenances a single stack, cannot be used to implement natural coroutines, as variables located on the stack will be overwritten as another coroutine uses the same stack space.

Thus for stack-based coroutines in C, functions are needed to create and jump between alternate stacks. A third function, which can usually be written in machine-specific C, is needed to create the context for a new coroutine. C libraries complying with POSIX or the Single UNIX Specification provide such routines as getcontext, setcontext, makecontext and swapcontext. The setcontext family of functions is thus considerably more powerful than setjmp/longjmp, but conforming implementations are as rare if not rarer. The main shortcoming of this approach is that the coroutine's stack is a fixed size and cannot be grown during execution. Thus, programs tend to allocate much more stack than they actually need in order to avoid the potential for stack overflow.

Due to the limitations of standard libraries, some authors have written their own libraries for coroutines. Russ Cox's libtask library is a good example of this genre. It uses the context functions if they are provided by the native C library; otherwise it provides its own implementations for ARM, PowerPC, Sparc, and x86. Other notable implementations include libpcl, coro, libCoroutine and libcoro.

Implementations for Python

- PEP 342 - better support for coroutine-like functionality, based on extended generators (implemented in Python 2.5)
- Greenlets
- kiwi tasklets
- multitask
- chiral
- cogen

Implementations for Ruby

- Coroutines in Ruby (with commentary in Japanese language)
- An implementation by Marc De Scheemaecker

Implementations for Perl

- Coro

Coroutines will also be a part of Perl 6.

Implementations for Smalltalk

Since in most Smalltalk environments the execution stack is a first-class citizen, Coroutine can be implemented without additional library or VM support.

Implementations for Delphi (programming language)

- Cool little Coroutines function Coroutine implementation by Bart van der Werf
- C# Yield implementation in Delphi by Sergey Antonov

Implementations in assembly languages

Machine-dependent assembly languages often provide direct methods for coroutine execution. For example, in MACRO-11, the assembly language of the PDP-11 minicomputer, the coroutine switch is the instruction "JSR PC,@(SP)++" which jumps to the address popped from the stack and pushes the current instruction address onto the stack.

4.5 Scheduled subprograms

The concept of subprogram scheduling results from relaxation of the assumption that execution of a subprogram should always be initiated immediately on its call. One may think of an ordinary subprogram call statement as specifying that the called subprogram is to be scheduled for execution immediately, without completing execution of the calling program. Completion of execution of the calling program is rescheduled to occur immediately on termination of the subprogram. The exception-handling control structure may also he viewed as a means of subprogram scheduling. The exception handler executes whenever a particular exception is raised. Generalizing further, other subprogram scheduling techniques are possible:

1. Schedule subprograms to be executed before or after other subprograms, as, for example: call B after A, which would schedule execution of Subprogram B after execution of Subprogram A is completed.

2. Schedule subprograms to be executed when an arbitrary Boolean expression becomes true, as, for example,

call 13 when X = 5 and Z > 0

Such scheduling provides a sort of generalized exception-handling feature: B is called whenever the values of Z and X are changed to %misty the given conditions.

3. Schedule subprograms on the basis of a simulated time scale, as, for example,

call B at time = 25 or call B at time = CurrentTime + 10

Such scheduling allows a general interleaving of subprogram calls scheduled from different sources.

4. Schedule subprograms according to a priority designation, as, for example,

call B with priority 7

which would activate B when no other subprogram with higher priority has been scheduled. Generalized subprogram scheduling is a feature of programming languages designed for discrete system simulation, such as GPSS, SIMSCRIPT, and SIMULA, although the concepts have wide applicability. Each of the prior scheduling techniques appears in at least one of the simulation languages mentioned. The most important technique in system simulation is the third in the list: scheduling based on a simulated time scale. We emphasize this technique in our discussion.

When we speak of subprogram scheduling, we mean scheduling of subprogram activations because this scheduling is a run-time activity in which the same subprogram may be scheduled to be activated at many different points during execution. In generalized subprogram scheduling, the programmer no longer writes a main program. Instead, the main program is a system-defined scheduler program that typically maintains a list of currently scheduled subprogram activations ordered in the sequence in which they are to be executed. Statements are provided in the language through which subprogram activations may be inserted into this list during execution. The scheduler operates by calling each subprogram on the list in the indicated sequence. When execution of one subprogram terminates, execution of the next subprogram on the list is initiated. Usually provision is also made for ordinary subprogram calls, sometimes simply by allowing a subprogram to suspend its own execution and schedule immediate execution of another subprogram.

In simulation languages, the most common approach to subprogram scheduling is based on a type of generalized coroutine. Execution of a single subprogram activation proceeds in a series of active and passive phases. During an active phase, the subprogram has control and is being executed; in a passive phase, the subprogram has transferred control elsewhere and is awaiting a resume call. However, rather than each coroutine directly transferring control to another coroutine when it switches from active to passive, control is returned to the scheduler, which then transfers control to the next subprogram on its list of scheduled activations. This transfer of control may take the form of a resume call if the subprogram is already partially executed or an entirely new activation of the subprogram may be initiated.

The coroutine scheduling concept is particularly direct using a simulated time scale. Assume that each active phase of execution of a subprogram may be scheduled to occur at any point on an integer time scale beginning at time T = 0. T is a simple integer variable that always contains the value of the current time on the simulated scale. Execution of an active phase of a subprogram always occurs instantaneously on this simulated time scale (i.e., the value of T does not change during execution of an active phase of a subprogram). When a subprogram completes an active phase and returns control to the scheduler, the scheduler updates the value of T to that at which the next subprogram on the list of scheduled subprograms is to be activated and transfers control to that subprogram. The newly activated routine partially executes and returns control to the scheduler, which again updates T and activates the next routine on the list.

Check your progress

Q1. Compare co-routines and sub-routines.

Q2. Which programming language support co-routines?

Q3. Explain schedule sub program.

4.6 Summary

In this unit you learnt about the core part of programming languages like Subprogram sequence control, Recursive sub-programs, Exception and exception handlers, Co-routines, Scheduled subprograms

- Subprograms cannot be recursive
- The set of associations created on entry to a subprogram that represents formal parameters, local variables, and subprograms defined only within that subprogram
- The **dynamic scope of an association** for an identifier is that set of subprogram activations in which the association is visible during execution.
- An arithmetic exception arises when an attempted atomic arithmetic operation has no result that would be acceptable universally. The meanings of atomic and acceptable vary with time and place.
- Coroutines are program components that generalize subroutines to allow multiple entry points and suspending and resuming of execution at certain locations.

4.7 Review Questions

Q1. What do you mean by sub program sequence control? Explain in details.

Q2. Explain recursive sub program in details.

Q3. What are exceptions? How can we handle them?

Q4. What is co-routines? Explain in details.

Q5. What do you mean by scheduled sub program?

BIBLOGRAPHY

John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184-195, April 1960.

Eugenio Moggi. Computational lambda-calculus and monads. In *IEEE Symposium on Logic in Computer Science (LICS), Asilomar, California*, pages 14-23, June 1989. Full version, titled *Notions of Computation and Monads*, in Information and Computation, 93(1), pp. 55-92, 1991.

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System-F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527-568, May 1999.

George C. Necula. Proof-carrying code. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Paris, France*, pages 106-119, January 1997.

Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223-255, 1977.

Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

Guy Lewis Steele, Jr. RABBIT: A compiler for SCHEME. Technical Report AITR-474, MIT Artificial Intelligence Laboratory, May 6 1978.

Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17(1-3):35-75, December 1991.

Andrzej Filinski. Representing layered monads. In *ACM Symposium on Principles of Programming Languages (POPL), San Antonio, Texas*, pages 175-188, New York, NY, 1999.

Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19-32, Providence, Rhode Island, 1967. American Mathematical Society.

Michael J. Gordon, Robin Milner, F. Lockwood Morris, Malcolm Newey, and Christopher P. Wadsworth. A meta language for interactive proof in LCF. Internal Report CSR-16-77, University of Edinburgh, Edinburgh, Scotland, September 1977.

C. A. R. Hoare. Proof of a program: FIND. *Communications of the ACM*, 14(1):39-45, January 1971.

C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666-677, August 1978. Reprinted in ``Distributed Computing: Concepts and Implementations" edited by McEntire, O'Reilly and Larson, IEEE, 1984.

BLOCK -3

Programming Languages-3

Block 3 Programming Languages-3

This is the third block of your course and it deals with the programming languages-3. As you know about the basics of programming language, this block describe detail knowledge about the programming language. This block has been divided in to four following units.

In the first unit we introduce the task and concurrency exception, the exception during task activation, exceptions raised during communication between tasks. We also explained how abnormal situations arrived in an accept statement, how an exception is raised within an accept statement. We also focused on concurrency and exceptions, exception handling in the concurrency runtime, referencing environments.

In the second unit we discussed about Static and Dynamic structures, comparison between static and dynamic structures. We also described block structure and its type like monolithic block structure, flat Block Structure, nested Block Structure.

In the third unit we focused on Local data and local referencing environments. We also focused on its Implementation. In this unit we described the concept of retention, deletion and its advantages and disadvantages.

In the last unit we introduce the concept of scope, its type. We also focused on static scope, dynamic scope and some important example of static and dynamic scope.

As you study the material you will come across abbreviations in the text, e.g. Sec. 1.1, Eq.(l .l) etc. The abbreviation Sec. stands for section, and Eq. for equation. Figure, a. b refers to the b^{th} figure of Unit a, i.e. Figure. 1.1 is the first figure in Unit 1. Similarly, Sec. 1.1 is the first section in Unit 1 and Eq.($.8) is the eighth equation in Unit 4. Similarly Table x. y refers to the y^{th} table of Unit x, i.e. Table. 1.1 is the first table in Unit 1.

In your study you will also find that the units are not of equal-length and your study time for each unit will vary.

We hope you enjoy studying the material and wish you success.

UNIT-I Task and Exception

Structure

1.0 Introduction

In this unit we explain the detailed description of Task and Exception. In this unit there are five sections. In Sec. 1.1 and Sec. 1.2 you will learn about Task and concurrency exception. In the Concurrency Runtime, a *task* is a unit of work that performs a specific job and typically runs in parallel with other tasks. A task can be decomposed into additional, more fine-grained tasks that are organized into a *task group*. You use tasks when you write asynchronous code and want some operation to occur after the asynchronous operation completes. For example, you could use a task to asynchronously read from a file and then use another task known as *continuation task*. Conversely, you can use tasks groups to decompose parallel work into smaller pieces. For example, suppose you have a recursive algorithm that divides the remaining work into two partitions. You can use task groups to run these partitions concurrently, and then wait for the divided work to complete. In Sec. 1.3 we define reference environment. In Sec. 1.4 and 1.5 you will find summary and review questions respectively.

Objectives

After studying this unit you should be able to:

- Express task and concurrency exception
- Define name & reference environments

1.1 Tasks and Exceptions

The program execution that does not hold a task is defined in terms of a sequential execution of its actions, according to the rules. These actions can be deliberated to be executed by a single logical processor.

Tasks are entities whose implementations proceed in parallel in the following sense. Each task can be considered to be executed by a logical processor of its own. Different tasks (we can say different logical processors) continue independently, except at points where they synchronize or coordinate. Some tasks have entries. An entry of a task can be called by other tasks. A task accepts a call of one of its entries by executing an accept statement for the entry. Synchronization is achieved by rendezvous between a tasks issuing an entry call end a task accepting the call. Some entries have parameters; entry calls and accept statements for such entries are the principal means of communicating or interactive values between tasks.

Each tasks properties are defined by a corresponding task unit or task components which contains a task specification and a task body. Task units or components are one of the four forms of program unit of which programs can be composed. The other forms are subprograms, packages and generic units. The properties of task units, tasks, and entries, and the statements that affect the interaction between tasks (that is, entry call statements, accept statements, delay statements, select statements, and abort statements) are also important in any programming language.

If we talk out exceptions than the question arises that what is exception? An exception is an event, which occurs during the execution of a program that interrupts the normal flow of the program's instructions.

When an error occurs within a routine, the routine generates an object and hands it off to the runtime system. The object, called an *exception object*, holds information about the error, including its type and the state of the program when the error arose. Creating or generating an exception object and handing it to the runtime system is called *throwing an exception*.

After a method or routine throws an exception, the runtime system tries to find something to handle it. The set of possible "something" to handle the exception is the ordered list of methods or routine that had been called to get to the method where the error occurred. The list of methods is known as the *call stack*

We will now consider two other interactions between tasking and exception handling:

- Exceptions raised during the activation of a task
- Exceptions raised during communication between tasks

Exceptions during Task Activation

Consider a procedure that declares three local tasks, A, B, and C. The actions performed by the procedure are partially subcontracted to these local tasks.

```
procedure PERFORM is
  task A;
  task B;
  task C;
  ...
begin
  -- activation of A, B, C, in parallel
  -- (1)
  ...
exception
  when TASKING_ERROR =>
  ...         --  (2)
end;
```

The semantics of Ada specifies that the three local tasks are activated, in parallel, after begin statement but before the first statement of the procedure. This means that at (1) we can rely on the fact that all three tasks are activated.

Consider now what happens if the activation of one (or more) of these tasks is not started as a consequence of the raising of an exception. It would not make much sense to execute the statements of the procedure, once it is known that one of the basic preconditions for its proper operation is not satisfied. For this reason, the execution of statements of the procedure is not started, and the predefined exception TASKING_ERROR is propagated at (1) to be handled by the exception handler at (2).

(By analogy, if A, B, and C were array declarations, the statements of the procedure would not be executed if the elaboration of any of these declarations raised an exception. The analogy stops the execution. However, since in the case of task activation the exception is raised in the statements, and can be handled locally, whereas in the case of arrays the exception is raised in the declarative part and propagated to the caller to be dealt with. Activation of a task behaves like an implicit initialization statement - placed after begin.)

Exceptions Raised During Communication between Tasks

When two tasks are attempting to communicate with each other, or two task are engaged in communication with each other, an abnormal situation arises in one of them. It may have an effect on the other.

As a basis for discussing the various cases that may arise, consider a task SERVER that provides the entry UPDATE:

```
task SERVER is
  entry UPDATE(THIS :  in out ITEM);
end;

task body SERVER is
  ...
begin
  ...
  accept UPDATE(THIS :  in out ITEM) do
    -- statements for servicing the request
  end UPDATE;
  ...
end SERVER;
```

and another task called USER, having no entry at all:

```
task USER;
task body USER is
  THING :  ITEM;
begin
  ...
  SERVER.UPDATE(THIS =>  THING);
  ...
end USER;
```

In this situation consider that what happens if the USER calls an entry of the SERVER

```
    SERVER.UPDATE(THIS =>  THING);
```

At a time when this called task has already completed its execution. Clearly the called task will never accept the entry call, and hence the caller USER wait forever. Consequently the exception TASKING_ERROR is propagated to the caller at the point of call: the caller is thereby informed that the call cannot be accepted. For similar reasons, this exception is also raised if the called task

(SERVER) has not already completed its execution, but proceeds to do so without encountering an accept statement for the entry call.

Abnormal Situations in an Accept Statement

Consider the other interactions correspond to error situations that may arise while two tasks are engaged in a rendezvous. We distinguish three possible error situations:

1. An exception is raised by the execution of the accept statement
2. The called task (SERVER) is disrupted while executing the accept statement
3. The caller (USER) is disrupted while the accept statement is being executed

The first situation corresponds to the usual error situation. The second and third cases are only possible if another task has issued an abort statement. For example:

```
abort SERVER;   -- in the second case

abort USER;     -- in the third case
```

Although such statement to be used only in extreme circumstances and it will eventually cause completion of the aborted task. This may be occur in any case, especially when the aborted task reaches a synchronization point: a point where it causes the activation of another task; an entry call; the start or the end of an accept statement; a select statement; a delay statement; an exception handler; or an abort statement.

We next analyze the consequences of each of these possible abnormal situations, both with respect to the task issuing the entry call (USER) and with respect to the task containing the accept statement (SERVER).

An exception is raised within an accept statement

Consider a situation in which an exception, say error, is raised within the accept statement of SERVER:

```
task body USER is                task body SERVER is

  ...

  SERVER.UPDATE(                   ...

    THIS =>  SOME_ITEM);

  ...

                                    accept UPDATE(THIS :  out ITEM) do
```

```
 ...
                                      error
 ...
                                     end;
end USER;                             ...
                                   end SERVER;
```

From the point of view of the caller, the accept statement is analogous to a procedure body that is executed when the corresponding entry is called. Hence if an exception is raised (and not handled within the accept statement itself) it should be propagated at the point of call of SERVER.UPDATE.

However, from the point of view of the task that contains the accept statement, this statement is a normal statement of its body. Hence if an exception is raised within SERVER, it should be handled by a handler provided within that same task (outside the accept statement).

To summarize, an exception raised within an accept statement (and not handled there) is propagated both in the calling and in the called tasks. Both tasks may provide handlers for the exception.

The called task is disrupted

A different treatment must be employed if the called task (SERVER) is aborted by a third task. In this case the caller (USER) must be informed that the entry call will never be completed. For this reason, the exception TASKING_ERROR is propagated at the point of the entry call.

The calling task is disrupted

In this case, the called task (SERVER) completes the rendezvous normally: the called task is unaffected.

There are good reasons for this dissymmetry of treatment. First, we can expect servers to be programmed in a more robust manner than user tasks. Moreover it is important to ensure continuity of service, and this would not be the case if it were possible for a single unsound user to affect the service to all user tasks. In terms of the implementation, this means that the storage of an aborted task cannot be reclaimed before the end of the rendezvous: this is important if the entry has in out or out parameters that are implemented by copy, or parameters of any mode that are implemented by reference.

1.2 Concurrency and Exceptions

Computer users take it for granted that their systems can do more than one thing at a time. They assume that they can continue to work in a word processor, while other applications download files, manage the print queue, and stream audio. Even a single application is often expected to do more than one thing at a time. For example, that streaming audio application must simultaneously read the digital audio off the network, decompress it, manage playback, and update its display. Even the word processor should always be ready to respond to keyboard and mouse events. No matter how busy it is, it reformatting the text or updating the display. Software that can do such things is known as *concurrent* software.

Exception Handling in the Concurrency Runtime

The Concurrency Runtime uses C++ exception handling to communicate many kinds of errors. These errors include invalid use of the runtime, runtime errors such as failure to acquire a resource, and errors that occur in work functions that you provide to tasks and task groups. When a task or task group throws an exception, the runtime holds that exception and marshals it to the context that waits for the task or task group to finish. For components such as lightweight tasks and agents, the runtime does not manage exceptions for you. In these cases, you must implement your own exception-handling mechanism. This topic describes how the runtime handles exceptions that are thrown by tasks, task groups, lightweight tasks, and asynchronous agents, and how to respond to exceptions in your applications.

Key Points

- When a task or task group throws an exception, the runtime holds that exception and marshals it to the context that waits for the task or task group to finish.
- When possible, surround every call to concurrency::task::get and concurrency::task::wait with a try/catch block to handle errors that you can recover from. The runtime terminates the application if a task throws an exception and that exception is not caught by the task, one of its continuations, or the main application.
- A task-based continuation always runs; it does not matter whether the antecedent task completed successfully, threw an exception, or was canceled. A value-based continuation does not run if the antecedent task throws or cancels.
- Because task-based continuations always run, consider whether to add a task-based continuation at the end of your continuation chain or not. This can help guarantee that your code observes all exceptions.
- The runtime throws concurrency::task_canceled when you call concurrency::task::get and that task is canceled.
- The runtime does not manage exceptions for lightweight tasks and agents.

Check your progress

Q1. What is the role of exceptions during task activations?

Q2. Define exception handling in the concurrency run time.

1.3 Referencing Environments

Referencing environment of a statement:

- Collection of all names that are visible in the statement.
- In a static scoped language, it corresponds to the variables declared in the local scope of the statement plus the collection of all variables of its ancestor scopes that are visible.
- It is needed while the statement is being compiled, so code and data structures can be created to allow references to variables from other scopes during runtime.

e.g, Pascal

```
program example;
var a, b : integer;
…
procedure sub1;
var x, y : integer;
begin { sub1 }
… (1)
end; { sub1 }
procedure sub2;
var x : integer;
…
procedure sub3;
var x : integer;
begin { sub3 }
… (2)
end; { sub3 }
begin { sub2 }
```

```
… (3)
end; { sub2 }
begin { example }
… (4)
end. { example }
```

Point Referencing Environment

1. x and y of sub1, a and b of example
2. x of sub3, (x of sub2 is hidden), a and b of example
3. x of sub2, a and b of example
4. a and b of example

A subprogram is active if its execution has begun but has not yet terminated.

The referencing environment of a statement in a dynamically scoped language is the locally declared variables, plus the variables of all other subprograms that are currently active.

Note that recent subprogram activations can have declarations for variables that hide variables with the same names in previous subprogram activations.

e.g., C

```
void sub1 (void) {
int a, b;
… (1)
} /* end of sub1 */
void sub2 (void) {
int b, c;
… (2)
sub1 ( );
} /* end of sub2 */
void main ( ) {
int c, d;
… (3)
sub2 ( );
} /* end of main */
```

Point Referencing Environment

1. d of main, c of sub2, a and b of sub1 (c of main and b of sub2 are hidden)
2. d of main, b and c of sub2 (c of main is hidden)
3. c and d of main

1.3 Named *Constants*

Variable bound to a value only at the time it is bound to storage; its value cannot be changed by assignment or by an input statement.

Advantage:

- readability
- control limits definition

Pascal named constant declarations require simple values on the RHS of the = operator.

Modula-2, FORTRAN 90 allow constant expressions to be used

Named constants in languages that use static binding of values are sometimes called manifest

Ada allows dynamic typing of values to named constants.

e.g.,

MAC : constant integer := 2 * WIDTH + 1;

Original C did not include named constants. Similar capability provided by the C preprocessor (macro processor).

e.g., #define LISTLEN 100

LISTLEN is called a named literal.

ANSI C/C++ have named constants, these cannot be used in ANSI C to define the length of arrays.

Variable Initialization

Binding of a variable to a value at the time it is bound to storage.

If the variable is statically bound to storage, binding and initialization occur before runtime.

If the storage binding is dynamic, initialization is also dynamic.

e.g., FORTRAN (compile time initialization)

REAL PI

INTEGER SUM

DATA SUM /0/, PI /3.14159/

e.g., Ada

```
SUM : INTEGER := 0;
LIMIT : constant INTEGER := OLD_LIMIT + 1;
COUNT : INTEGER := OLD_COUNT + 1;
```

e.g., ALGOL 68

```
int first := 10; (variable initialization)
int second = 10; (named constant initialization)
```

e.g., Pascal, Modula-2

No way to initialize variables other than at run time with assignment statements

Initialization occurs only once for static variables, but occurs with every allocation for dynamically allocated variables (e.g., local variables in an Ada procedure).

Check your progress

Q1. Define referencing environment.

Q2. Explain named constant.

1.4 Summary

In this unit you learnt about Task and concurrency exception, Name and reference environments.

- When two tasks are attempting to communicate with each other, or are engaged in communication, an abnormal situation arises in one of them and it may have an effect on the other.
- Computer users take it for granted that their systems can do more than one thing at a time.
- The Concurrency Runtime uses C++ exception handling to communicate many kinds of errors.
- A task-based continuation always runs; it does not matter whether the antecedent task completed successfully, threw an exception, or was canceled.
- ANSI C/C++ have named constants, these cannot be used in ANSI C to define the length of arrays.

1.5 Review Questions

Q1. What is a task exception? Define with example.

Q2. What do you mean by concurrency? In which case concurrency exception occur? Elaborate your answer.

Q3. What do you mean by naming? Define with example.

Q4. Define reference environment in detail.

Q5. What is variable initialization? How a variable initialize?

UNIT-II Structures (Static, Dynamic & Block)

Structure

2.0 Introduction

In this unit we define structures. These structures may be static, dynamic and block. In this unit there are four sections. In Sec. 2.1 you will know about static and dynamic structures. As you know Arrays allow to define type of variables that can hold several data items of the same kind. Similarly structure is another user defined data type that allows to combine data items of different kinds. Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book –Title, Author, Subject, Book ID. In Sec. 2.2 you will learn about block structure. ALGOL had recursive subprograms—procedures that could invoke themselves to solve a problem by reducing it to a smaller problem of the same kind. ALGOL introduced block structure, in which a program is composed of blocks that might contain both data and instructions and have the same structure as an entire program. Block structure became a powerful tool in programming. In Sec. 2.3 and 2.4 you will find summary and review questions respectively.

Objectives

After studying this unit you should be able to:

- Define static and dynamic structures.
- Express block structure.

2.1 Static and Dynamic structures

A data structure is a collection of data items, in addition a number of operations are provided by the software to manipulate the data structure.

There are two approaches to creating a data structure.

2.1.1 STATIC DATA STRUCTURE

With a static data structure, the size of the structure is fixed. Static data structures are very good for storing a well-defined number of data items.

For example a programmer might be coding an 'Undo' function where the last 10 user actions are kept in case they want to undo their actions. In this case the maximum allowed is 10 steps and so he decides to form a 10 item data structure.

2.1.2 DYNAMIC DATA STRUCTURE

There are many situations where the number of items to be stored is not known beforehand.

In this case the programmer will consider using a dynamic data structure. This means the data structure is allowed to grow and shrink as the demand for storage arises. The programmer should also set a maximum size to help avoid memory collisions.

For example a programmer coding a print spooler will have to maintain a data structure to store print jobs, but he cannot know beforehand how many jobs there will be.

The table below compared the two approaches

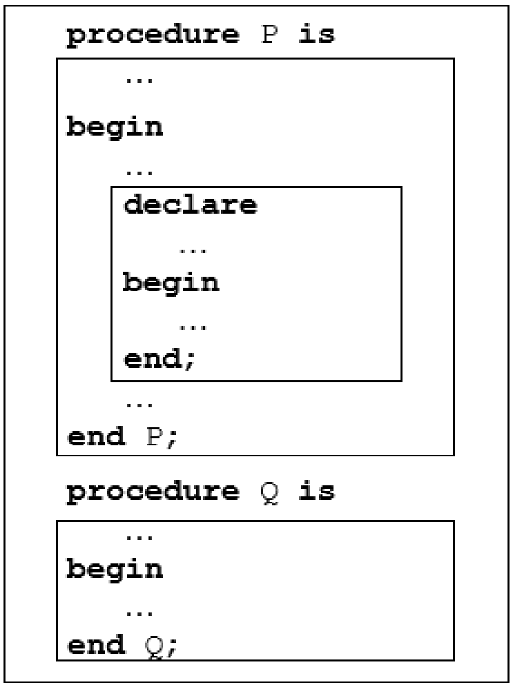
Dynamic and Static data structures	
DYNAMIC	**STATIC**
Memory is allocated to the data structure dynamically at run time i.e. as the program executes.	Memory is allocated at compile time. Fixed size.
Disadvantage: Because the memory allocation is dynamic, it is possible for the structure to 'overflow' should it exceed its allowed limit. It can also 'underflow' should it become empty.	Advantage: The memory allocation is fixed and so there will be no problem with adding and removing data items.
Advantage: Makes the most	Disadvantage: Can be very

efficient use of memory as the data structure only uses as much memory as it needs	inefficient as the memory for the data structure has been set aside regardless of whether it is needed or not whilst the program is executing.
Disadvantage: Harder to program as the software needs to keep track of its size and data item locations at all times	Advantage: Easier to program as there is no need to check on data structure size at any point.

Table-1

2.2 Block structure

A *block* is a construct that delimits the scope of any definitions that it may contain. It provides a local environment i.e., an opportunity for local definitions. The *block structure* (the textual relationship between blocks) of a programming language has a great deal of influence over program structure and modularity. In the discussion that follows, we will refer to the block structures found in Figure 2.0



```
procedure P is
    ...
begin
    ...
    declare
        ...
    begin
        ...
    end;
    ...
end P;
procedure Q is
    ...
begin
    ...
end Q;
```

Figure 2.0 Block Structure of ADA

Figure 2.0 presents two styles of blocks, the first requires the definitions to proceed the body and the second requires definitions to follow the body.

There are three basic block structures--monolithic, flat and nested.

Monolithic Block Structure

A program has a *monolithic* block structure if it consists of just one block. This structure is typical of BASIC and early versions of COBOL. The monolithic structure is suitable only for small programs. The scope of every definition is the entire program. Typically all definitions are grouped in one place even if they are used in different parts of the program.

Global Data

Return Address$_1$

...

Return Address$_n$

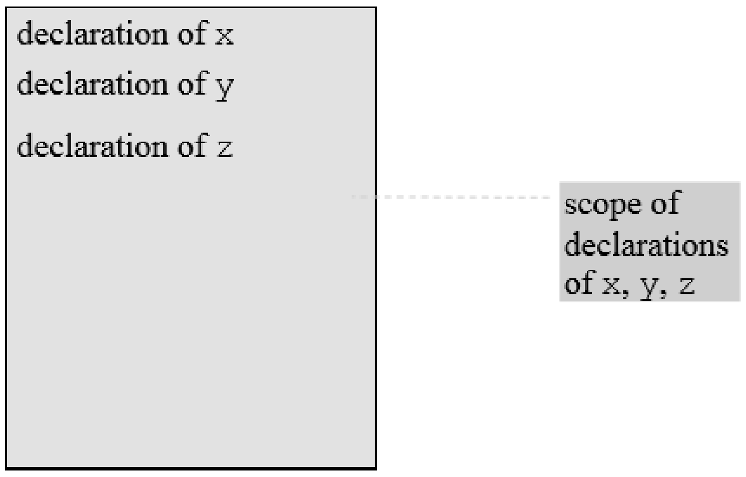


Figure 2.1 Monolithic Block Structure

As shown in the figure, the whole program is a single block. The scope of every declaration is the whole program.

Flat Block Structure

A program has a *flat* block structure if it is partitioned into distinct blocks, an outer block may contain one or more inner blocks i.e., the body may contain additional blocks but the inner blocks may not contain blocks. This structure is typical of FORTRAN and C. In these languages, all subprograms (procedures and functions) are separate, and each acts as a block. Variables can be declared inside a subprogram are then local to that subprogram. Subprogram names are part of the outer block and thus their scope is the entire program along with global variables. All subprogram names and global variables must be unique. If a variable cannot be local to a subprogram then it must be global and accessible by all subprograms even though it is used in only a couple of subprograms.

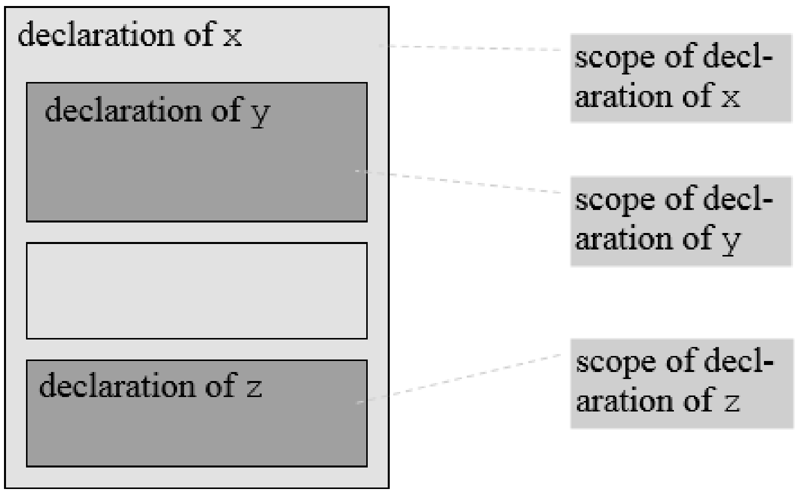

declaration of x
declaration of y
declaration of z
scope of decl-aration of x
scope of decl-aration of y
scope of decl-aration of z

Figure-2.2

Nested Block Structure

A program has *nested* block structure if blocks may be nested inside other blocks i.e., there is no restriction on the nesting of blocks within the body. This is typical of the block structure of the Algol-like languages. A block can be located close to the point of use. In blocks visibility is controlled by nesting. All names are visible (implicitly exported) to internally nested blocks. No names are visible (exported) to enclosing blocks. In a block, the only names visible are those that are declared in all enclosing blocks or are declared in the block, but not those declared in nested blocks.

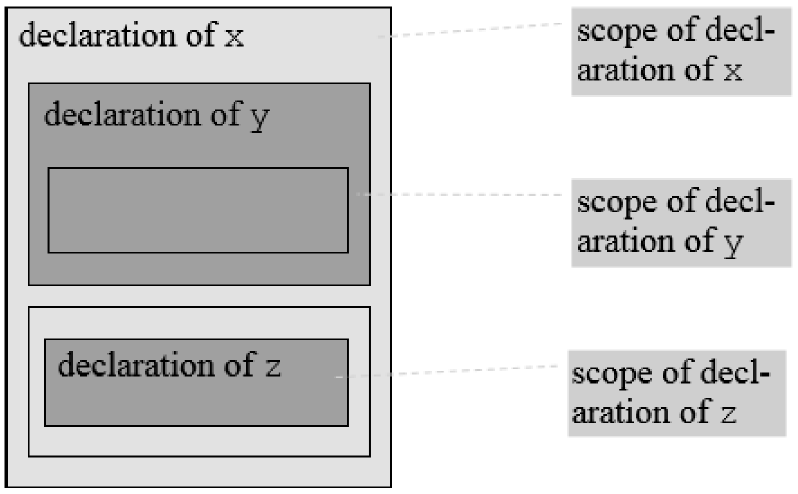

declaration of x
declaration of y
declaration of z
scope of decl-aration of x
scope of decl-aration of y
scope of decl-aration of z

Figure-2-3

A *local name* is one that is declared within a block for use only within that block.

A *global name* is a name that when referenced within a block refers to a name declared outside the block.

An *activation* of a block is a time interval during which that block is being executed.

The three basic block structures are sufficient for what is called *programming in the small* (PITS). These are programs which are comprehensible in their entirety by an individual programmer. However, they are not general enough for very large programs. Large programs which are written by many individuals and which must consist of modules that can be developed and tested independently of other modules. Such programming is called *programming in the large* (PITL*)*.

Check your progress

Q1. Differentiate between static and dynamic data structure.

Q2. Explain

(i) Monolithic block structure

(ii) Flat block structure

(iii)Nested block structure.

2.3 Summary

In this unit you learnt about different structures of programming languages like static structure, dynamic structure, and block structure.

- *A data structure is a collection of data items, in addition a number of operations are provided by the software to manipulate the data structure.*
- Static data structure, the size of the structure is fixed
- Memory is allocated to the data structure dynamically i.e. as the program executes.
- A *block* is a construct that delimits the scope of any definitions that it may contain.
- A program has a *flat* block structure if it is partitioned into distinct blocks, an outer all inclosing block one or more inner blocks i.e., the

body may contain additional blocks but the inner blocks may not contain blocks.

2.4 Review Questions

Q1. Define the structure in the programming languages with example.

Q2. What do you mean by static structure with example?

Q3. What do you mean by dynamic structure with example?

Q4. Define block structure in details.

Q5. What are data models? How many types of data models available in DBMS?

UNIT-III Local Referencing Environments

Structure

3.0 Introduction

In this unit you will learn about local referencing environments. In Sec. 3.1 we define local data & local referencing environments. As you have learnt subprograms are generally allowed to declare their own variables, thereby defining local referencing environments. Variables that are declared inside subprograms are called local variables because access to them is usually restricted to the subprogram in which they are declared. The main disadvantages of stack-dynamic local variables are the following: first there is the cost of the time required to allocate, initialize (when necessary), and deallocate such variables for each call. Second, references to stack-dynamic local variables must be indirect, whereas references to static variables can be direct. On most computers, indirect references are slower than direct references. Finally, with stack-dynamic local variables, subprograms cannot be history sensitive subprograms. A common example of a need for a history sensitive subprogram is one whose task is to generate pseudorandom number. Each call to such a subprogram computes one pseudorandom number, using the last one it computed. It must, therefore, store the last one in a static local variable. Co-routines and the subprogram used in iterator loop constructs are other examples of the need to be history sensitive.

The primary advantage of static local variables is that they are efficient they usually can be accessed faster because there is no indirection and no run-time overhead of allocation and de-allocation. And, of course, they allow subprograms to be history sensitive. The greatest disadvantage is the inability to support recursion.

In Sec. 2.7 and 2.8 you will find summary and review questions respectively.

Objectives

After studying this unit you should be able to:

- Define local data and local referencing environments
- Express its advantages and disadvantages.

3.1 Local data and local referencing environments

We now begin to look at the various data-control structures used in programming languages. Firstly we have to discuss about Local data. The term local data used to define the data elements that are to be used exclusively by a single Natural module in an application. These elements or fields can be defined within the statement itself; or they can be defined outside the program in a separate local data area (LDA) or a parameter data area (PDA), with the statement referencing that data area.

The general syntax of local data is-

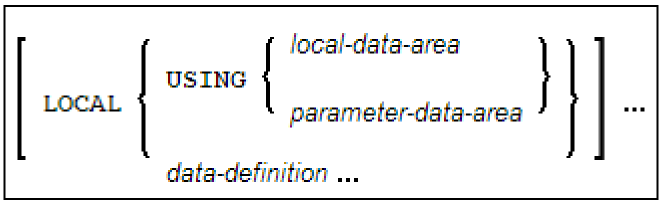


Figure 3.0: Syntax Local data

Now we have to discuss about Local referencing environments, which form the simplest structure, are treated in this section. The local environment of a subprogram Q consists of the various identifiers declared in the head of Subprogram Q. The subprogram names of interest are the names of subprograms that are defined locally within Q (i.e., subprograms whose definitions are nested within Q).

For local environments, static and dynamic scope rules are easily made consistent. The static scope rule specifies that a reference to an identifier X in the body of Subprogram Q is related to the local declaration for X in the head of Subprogram Q (assuming one exists). The dynamic scope rule specifies that a reference to X during execution of Q refers to the association for X in the current activation of Q (note that in general, there may be several activations of Q, but only one will be currently in execution).To implement the static scope rule, the compiler simply maintains a table of the local declarations for identifiers that appear in the head of Q. and. while compiling the body of Q. it refers to this table first whenever the declaration of an identifier is required. Implementation of the dynamic scope rule may be done in two ways, and each gives a different semantics to local references. Consider Subprograms P, Q and R with a local variable X declared in Q in Figure. Subprogram P calls Q, which in turn calls R. which later returns control to Q, which completes its execution and returns control to P. Let us follow variable A' during this execution sequence:

1. When P is in execution. X is not visible in P because X is local to Q.

2. When P calls Q, X becomes visible as the name of an integer data object with initial value, 30, As Q executes, the first statement references X and prints its current value 30.
3. When Q calls R the association for X becomes hidden, but it is retained while R executes.
4. When R returns control to Q, the association for X becomes visible again. X still names the same data object, and that data object still has the value 30.
5. Q resumes its execution, X is referenced and incremented, and then its new value, 31, is printed.
6. When Q returns control to P, the association for X again becomes hidden, but two different meanings might be provided for this association:

```
procedure R;
    . . .
end;
procedure Q;
var X: integer := 30;   – initial value of X is 30
begin
    write (X);          – print value of X
    R;                  – call subprogram R
    X := X + 1;         – increment value of X
    write (X)           – print value of X again
end;
procedure P;
    . . .
    Q;                  – call subprogram Q
    . . .
end;
```

Figure 3-1 local referencing environment: retention or deletion?

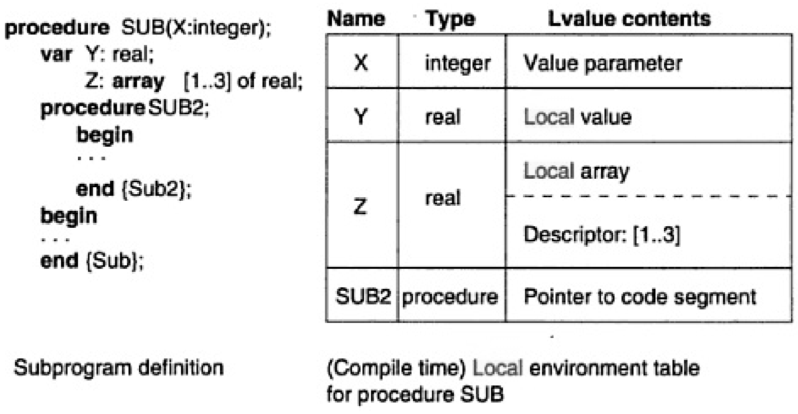
Retention: - The association for X might be retained until Q is called again, just as it was while Q called R. If the association is retained, then when Q is called the second time, Q is still associated with the same data object, which still has its old value, 31. Thus, the first statement executed will reference X and print the value 31, if the entire cycle repeats and Q is called a third time. X will have the value 32 and so on.

Deletion: - Alternatively, the association for X might he deleted (i.e. the association binding X to the data object might be broken, and the data object destroyed and its storage reallocated for some other use). When Q is called a second time, a new data object is created and assigned the initial value 30 and the association with X is recreated as well. The first statement in Q then prints the value 30 every time Q is executed.

Retention and deletion are two different approaches to the semantics of local environments and are concerned with the lifetime of the environment. (Java, Pascal, Ada, LISP, APL and SNOBOL4 use the deletion approach: Local variables do not retain their old values between successive calls of a subprogram. COBOL and many versions of FORTRAN use the retention approach: Variables do retain their old values between calls. PL/I and ALGOL provide both options; each individual variable may be treated differently.

Implementation

In discussing the implementation of referencing environments, it is convenient to represent the local environment of a subprogram as a local environment table consisting of pairs, each containing an identifier and the associated data object as shown in Figure. As discussed previously, the storage for each object is represented as a type and its location in memory as an I-value. Drawing a local environment table this way does not imply that the actual identifiers (e.g., "X") are stored during program execution. Usually they are not. The name is only used so that later references to that variable will be able to determine where that variable will reside in memory during execution. Using local environment tables, implementation of the retention and deletion approaches to local environments is straightforward.



```
procedure SUB(X:integer);
    var Y: real;
        Z: array [1..3] of real;
    procedure SUB2;
        begin
        . . .
        end {Sub2};
    begin
    . . .
    end {Sub};
```

Name	Type	Lvalue contents
X	integer	Value parameter
Y	real	Local value
Z	real	Local array / Descriptor: [1..3]
SUB2	procedure	Pointer to code segment

Subprogram definition

(Compile time) Local environment table for procedure SUB

Figure 3-2 Pascal local environment table

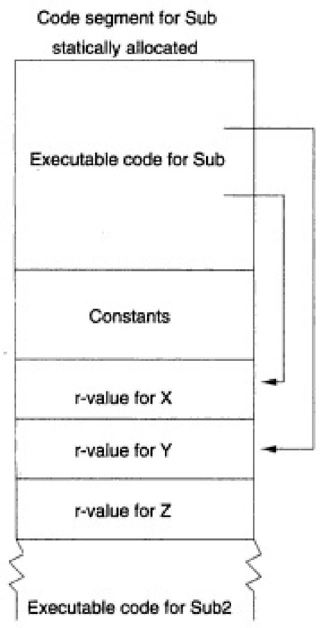


Figure 3-3 Allocation and referencing of retained local variable

Retention: - if the local environment of subprogram Sub of Figure 3-2 is to be retained between calls, then a single local environment table containing the retained variables is allocated as part of the code segment of Sub, as shown in Figure 3-3. Because the code segment is allocated storage statically and remains in existence throughout execution, any variables in the local environment part of the code segment are also retained. If a variable has an initial value, such as the Value 30 for Y, then the initial value may be stored in the data object when the storage is allocated (just as the value for a constant in the code segment would be stored). Assuming each retained local variable is declared at the start of the definition of Sub; the compiler can determine the size of each variable in the local environment table and compute the offset of the start of the data object from the start (the base address) of the code segment. When a statement within the code references a variable Y during execution, the offset of Y is added to the base address of the code segment to find the location of the data object associated with Y .The identifier "Y" is not needed during execution and is not stored at all. With this implementation of retention for the local environment, no special action is needed to retain the values of the data objects: the values stored at the end of one call of Sub will still be there when the next call begins. Also no special action is needed to change from one local environment to another as one subprogram calls another. Because the code and local data for each subprogram are part of the same code segment, a transfer of control to the code for another subprogram automatically results in a transfer to the local environment for that subprogram as well.

Deletion: - if the local environment of SUB is to be deleted between calls and recreated a new on each entry then the local environment table containing the deleted variables is allocated storage as part of the activation record for *Sub.* Assuming the activation record is created on a central stack on entry to *Sub* and deleted from the stack on exit, as discussed in previous Section, deletion of the local environment follows automatically. Assuming each deleted local variable is declared at the start of the definition of *Sub,* the compiler again can *determine* the number of variables and the size of each in the local environment table and may compute the oil set of the start of each data object from the start of the activation record (the base address). Recall that the CEP pointer (current-environment pointer) is maintained during execution so that it points to the base address of the activation record in the stack for the subprogram that is currently executing at any point. If *Sub* is executing and references Variable Y, then the location of the data object associated with Y is found by adding the offset for Y to the contents of the CEP Again the identifier “Y” need not be stored in the activation record at all only the data objects are needed. Figure 3-4 shows this implementation. The dotted arrow shows the offset computed for a reference to Y.

Both the retention and deletion approaches are easy to implement in our general model of sub program implementation as given in previous section. Several additional points deserve notice:

1. In the implementation of the *simple* call-return structure described earlier, retention and deletion in the absence of recursion lead to essentially, the same implementation because there is never more than one activation record that may be allocated statically. However if initial values are provided for variables, two different meanings for initializing results.

2. Individual variables may readily be given either treatment, with those whose values are to be retained allocated storage in the code segment and those w hose values are to be deleted allocated storage in the activation record. This is the approach used in PL/I where a variable declared STATIC is retained and a variable declared AUTOMATIC is deleted.

3. A subprogram name, which is associated with a declaration for the subprogram in the local environment may always be treated as retained, the association of name and definition may be represented as a pointer data object in the code segment, where the pointer points to the code segment representing the subprogram.

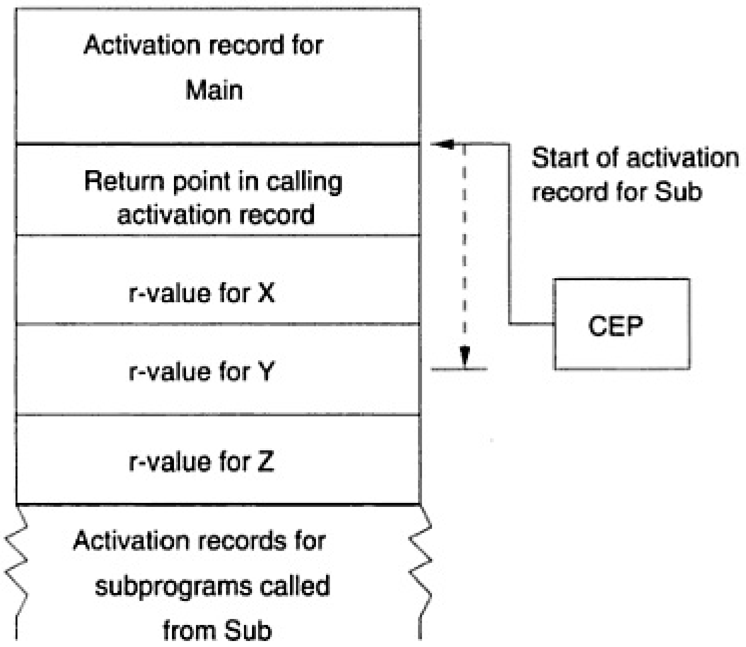

Activation record for Main
Return point in calling activation record
r-value for X
r-value for Y
r-value for Z
Activation records for subprograms called from Sub
Start of activation record for Sub
CEP

Figure 3-4 Allocation & referencing of deleted local variables.

4. A formal-parameter name represents a data object that is reinitialized with a new value on each call of the subprogram, as described in previous section. This re-initialization precludes retaining an old value for a formal parameter between calls. Thus, formal parameters are appropriately treated as deleted associations.

5. If recursive subprogram calls are allowed, and then multiple activations of a subprogram Sub may exist as separate activation records in the central stack at the same time. If a variable Y is treated as deleted, then each activation record will contain a separate data object named Y, and as each activation executes, it will reference its own local copy of Y. Ordinarily separate copies of Y in each activation are what is desired, and thus for languages with recursion (or other subprogram control structures that generate multiple simultaneous activations of a subprogram) the deletion of local environments is ordinarily used. However, retention of sonic local variables is often of value. In ALGOL 60, for example, a local variable declared as own is treated as a retained variable, although the subprogram containing the declaration may be recursive. If multiple activations of a subprogram Sub reference the retained Variable Y, then there is only one data object Y that is used by every activation of Sub and its value persists from one activation to another.

Advantages and disadvantages: -Both retention and deletion are used in a substantial number of important languages. The retention approach allows the programmer to write subprograms that are history sensitive in that their results on each call are partially determined by their inputs and partially by the local data values computed during previous activations. The deletion approach does not allow any local data to be carried over from one call to the next, so a variable that must be retained between calls must be declared as nonlocal to the subprogram. For recursive subprograms, however, deletion is the more natural strategy. Deletion also provides a savings in storage space in that local environment tables exist only for those subprograms that are in execution or suspended execution. Local environment tables for all subprograms exist throughout execution using retention.

Check your progress

Q1. How retention and deletion works in local referencing environment?

3.2 Summary

In this unit you learnt about Local data, local referencing environments. These concepts are very important in programming languages.

- Local referencing environments, which form the simplest structure

- The local environment of a subprogram Q consists of the various identifiers declared in the head of Subprogram Q.
- Retention and deletion are two different approaches to the semantics of local environments and are concerned with the lifetime of the environment.
- Variables do retain their old values between calls
- Retention and deletion are used in a substantial number of important languages.

3.3 Review Questions

Q1. Define the term, Local data in details.

Q2. What is local referencing environment? Explain in detail.

Q3. Define retention with example.

Q4. Define detention in detail.

Q5. Write the procedure of implementing the retention and detention with example.

UNIT-IV Scope of Shared Data

Structure

4.0 Introduction

In this unit we introduced the concept of scope of shared data. In this unit there are four sections. In Sec. 4.1 you will learn about dynamic and static scope of shared data. As you know in computer programming, the scope of a name binding – an association of a name to an entity, such as a variable – is the part of a computer program where the binding is valid: where the name can be used to refer to the entity. In other parts of the program the name may refer to a different entity (it may have a different binding), or to nothing at all (it may be unbound). The scope of a binding is also known as the visibility of an entity, particularly in older or more technical literature – this is from the perspective of the referenced entity, not the referencing name. A scope is a part of a program that is or can be the scope for a set of bindings – a precise definition is tricky, but in casual use and in practice largely corresponds to a block, a function, or a file, depending on language and type of entity. The term "scope" is also used to refer to the set of all entities that are visible or names that are valid within a portion of the program or at a given point in a program, which is more correctly referred to as context or environment. In Sec. 4.2 you will know about types of scope. In Sec. 4.3 and 4.4 you will find summary and review questions respectively.

Objectives

After studying this unit you should be able to:

- Define dynamic and static scope of shared data.
- Express types of scopes.

4.1 Dynamic and static scope of shared data

Typically, scope is used to manage the visibility and accessibility of variables from different parts of a program.

Several encapsulating abstractions (such as *namespaces*, *modules*, etc.) related with a scope, are invented to provide a better modularity of a system and to avoid *naming variables conflicts*. In the same respect, there are *local variables* of *functions* and local variables of *blocks*. Such techniques help to *increase the abstraction* and encapsulate the internal data (not bothering a user of this abstraction with details of the implementation and exact internal variable names).

Concept of a scope helps us to use in one program the same name variables but with possibly different meanings and values. From this viewpoint:

Scope is an enclosing context in which a variable is associated with a value.

We may also say that this is a *logical boundary* in which a *variable* (or even an *expression*) has its *meaning*. For example, a *global variable*, a *local variable*, etc., which generally reflects a logical range of a variable *lifetime* (or *extent*)?

Block and function concepts lead us to one of the major scope properties — *to nest* other scopes or *to be nested*. Thus, as we'll see, not all implementations allow nested functions; the same as not all implementations (and in particular the current version of ECMA Script) provide block-level scope.

Consider the following C example:

```
// global "x"

int x = 100;


void foo() {


  // local "x" of "foo" function

  int x = 200;


  if (true) {

    // local "x" of if-block

    int x = 300;

    printf("%d", x); // 300

  }
  printf("%d", x); // 200

}

foo();

printf("%d", x); // 100
```

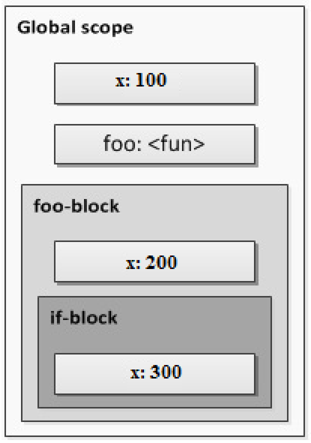


Figure 4-1: - Nested Scope

4.2 Types of Scope

Basically there are two types of scope available in the programming language, i.e. static scope and dynamic scope (As shown in the figure 4-2)

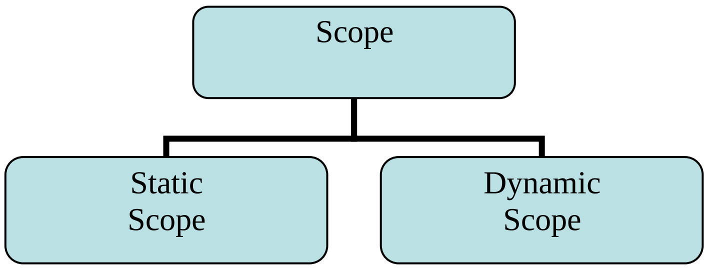


Figure 4-2: - Scope Type

- **Static Scope**

Definition: -

i. The declarations local to a block define the local environment of that block. The local declarations of a block include only those present in the block (usually at the start of the block itself) and not those possibly present in blocks nested inside the block in question.

ii. If a name is used inside a block, the valid association for this name is the one present in the environment local to the block, if it exists. If no association for the name exists in the environment local to the block, the associations existing in the environment local to the block immediately containing the starting block are considered. If the association is found in this block, it is the valid one, otherwise the search continues with the blocks containing the one with which we started, from the nearest to the furthest. If, during this search, the outermost block is reached and it contains no association for the name, then this association must be looked up in the language's predefined environment. If no association exists here, there is an error.

iii. A block can be assigned a name, in which case the name is part of the local environment of the block which immediately includes the block to which the name has been assigned. This is the case also for blocks associated with procedures.

In static scoping, an identifier refers to its nearest lexical environment. The word "lexical" in this case relates to a property of a program text. I.e. where lexically in the source text a variable appears (that is, the exact place in the code) — in that scope it will be resolved later at runtime on referencing this variable. The word "environment" implies again something that lexically surrounds the definition.
The word "static" relates to ability to determine the scope of an identifier during the parsing stage of a program. That is, if we (by looking on the code) can say before the starting the program, in which scope a variable will be resolved — we deal with a static scope.

For example

```
{
   int x = 0;
   void fie(int n){
      x = n+1;
```

```
    }
    fie(3);
    write(x);
    {
        int x = 0;
        fie(3);
        write(x);
    }
    write(x);
}
```

This is a case of static scope, the first write(x) output the 4, because the fie(3) modify the x=n+1 and n=3 here. So the output 1 is 4. The second write(x) output 0, because fie(3) still use the x define the out-most block according the static scope, but not the nest-est definition of x, so the inner-est x is not be modified, still 0, so the second output is 0. And the last one is output 4, it is obviously.

- **Dynamic Scope**

Based on the flow of execution

```
{
    const x = 0;
    void fie(){
        write(x);
    }
    void foo(){
        const x =1;
        fie();
    }
    foo();
}
```

This is a case of dynamic scope, the dynamic scope is based on the flow of execution, so the father block of a block is the one which call the block, but not where is the block in the file. So the foo() call the fie(), and the father block of fie block is foo, so the output x is the one which is defined in the foo, whose value is 1, the output is 1.

In contrast with the static scope, dynamic scope assumes that we cannot determine at parsing stage, to which value (in which environment) a variable will be resolved. Usually it means that the variable is resolved not in the lexical environment, but rather in the dynamically formed global stack of variables. Every met variable declaration just puts the name of the variable onto the stack. After the scope (lifetime) of the variable is finished, the variable is removed (popped) from the stack.

That means that for a single function we may have infinite resolution ways of the same variable name— depending on the context from which the function is called.

Some Example of Static and Dynamic Scope

- Consider the following program fragment written in a pseudo-language which uses static scope and where the primitive read(Y) allows the reading of the variable Y from standard input.

```
int X = 0;
int Y;
void fie()
{
   X++;
}
void foo()
{
   X++;
   fie();
}
read(Y);
if Y > 0
{
   int X = 5;  // line 12
   foo();
```

```
    }
else
    foo();
write(X);
```

This is a case of static scope, the definition of static scope the father block is the one out of this block. In this case, the difference is where we call the foo(), actually the definition x=5 in line 12 doesn't influence the execution of foo(), so whatever the input of Y, the output is always 2 (add 1 in the foo and then add 1 in the fie).

- Consider the following program fragment written in a pseudo-language that uses dynamic scope.

```
int X;      // line 1
X = 1;
int Y;
void fie() {
    foo();
    X = 0;
}
void foo(){
    int X;
    X = 5;
}
read(Y);
if Y > 0{
    int X;  // line 14
    X = 4;
    fie();
}
else
    fie();
write(X);
```

State which is (or are) the printed values.

When Y greater than 0, the

```
    int X;
  X = 4;
  fie();
```

will be executed, cause this is a case of dynamic scope, the father block is according the running environment, so the X in the father block of fie() here is defined in line 14, that means the fie execute and modify the value of X, but it's modify the inner-est X, but not the one defined in line 1, and the write(X) output the X defined in line 1, so the output is 1.

When Y less or equal than 0, the fie() will be executed, this case is simple, the output is 0.

- Consider the following code fragment in which there are gaps indicated by (*) and (**). Provide code to insert in these positions in such a way that:

a) If the language being used employs static scope, the two calls to the procedure foo assign the same value to x.

b) If the language being used employs dynamic scope, the two calls to the procedure foo assign different values to x.

The function foo must be appropriately declared at (*).

```
{
  int i;
  (*)
  for (i=0; i<=1; i++){
    int x;
    (**)
    x= foo();
  }
}
```

There is no a standard answer for this question, the point here is the definition of function foo, which data will be return from the foo, it is obviously the foo will use a object inherent from father block, possibly modify it, and return a result according the object.

example:

```
{
    int i;
    i = 0;
    foo {
        return i
    }
    for (i=0; i<=1; i++){
        int x;
        (blank)
        x= foo();
    }
}
```

In this example, the return of the foo depend on the value of i, in static scope case, the i is always 0, but in the dynamic case, the value of i is depend on the loop.

Check your progress

Q1. Explain static and dynamic scope.

4.3 Summary

In this unit you learnt Dynamic scope of shared data, Static scope of shared data and types of scope

- Several encapsulating abstractions (such as *namespaces*, *modules*, etc.) related with a scope, are invented to provide a better modularity of a system and to avoid *naming variables conflicts*.
- Concept of a scope helps us to use in one program the same name variables but with possibly different meanings and values.
- ***Scope is an enclosing context in which a variable is associated with a value.***
- Block and function concepts lead us to one of the major scope properties — *to nest* other scopes or *to be nested*.

- The local declarations of a block include only those present in the block and not those possibly present in blocks nested inside the block in question.

4.4 Review Questions

Q1. What do you mean by shared data? Explain with example.

Q2. What do you mean by scope? Why a programming language need scope? Justify your answer.

Q3. How many types of scope available in programming language?

Q4. Define static scope in detail.

Q5. Define dynamic scope in detail.

BIBLIOGRAPHY

Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. In *Functional programming languages and computer architecture. Proc. of a conference (Nancy, France, Sept. 1985)*, pages 190-203, New York, NY, USA, 1985. Springer-Verlag Inc.

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, Hemel Hempstead, Hertfordshire, UK, 1993.

G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471-475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 247 of *Lecture Notes in Computer Science*, pages 22-39. Springer-Verlag, 1987.

Donald E. Knuth. Structured programming with go to statements. *Computing Surveys*, 6(4):261-301, 1974.

Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

Peter Naur et al. Revised report on the algorithmic language ALGOL 60. *Communications of the ACM*, 6(1):1-17, January 1963.

John C. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83, Paris, France*, pages 513-523. Elsevier, 1983.

John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363-397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972), with a foreword.

Dana Scott. Continuous lattices. In F. W. Lawvere, editor, *Toposes, Algebraic Geometry, and Logic*, number 274 in Lecture Notes in Mathematics, pages 97-136. Springer-Verlag, 1972.

Dana Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. Programming Research Group Technical Monograph PRG-6, Oxford Univ. Computing Lab., 1971.

Brian Cantwell Smith. Reflection and semantics in lisp. In *ACM Symposium on Principles of Programming Languages (POPL), Salt Lake City, Utah*, pages 23-39, January 1984.

Christopher Strachey. Towards a formal semantics. In *Formal Language Description Languages for Computer Programming*, pages 198-220. North Holland, 1966.

BLOCK -4

Programming Languages-4

BLOCK 4 Programming Languages-4

This is the fourth block of this course on programming languages-4. This block concentrate on programming language and focused on important parts of it. This block has been divided in to four following units.

In this first unit we introduce the block structure and its parameters and transmission. This block also handle the concept of different parameter transmission methods like call by value, call by result, call by value-result, call by reference and call by name. In this unit some example of parameter transmission methods has also been provided.

In the second unit we discussed the task and shard data storage requirement for major runtime elements in the programming language. We also introduced mutual exclusion, critical region conditional critical regions, its implementation, its limitations, monitors, message passing, storage management, elements requiring storage.

In the third unit we discuss about the program and system controlled storage management. We also focused on storage management phases, initial allocation, recovery, compaction and reuse.

In this last unit we focused on static and stack based storage management. We also focused on fixed size and variable size heap and storage management. We deals with First-fit method, Best-fit method, Worst-fit method and Problem of fragmentation arise which can be solved by: Partial compaction, Full compaction.

As you study the material you will come across abbreviations in the text, e.g. Sec. 1.1, Eq.(l .l) etc. The abbreviation Sec. stands for section, and Eq. for equation. Figure, a. b refers to the bth figure of Unit a, i.e. Figure. 1.1 is the first figure in Unit 1. Similarly, Sec. 1.1 is the first section in Unit 1 and Eq.($.8) is the eighth equation in Unit 4. Similarly Table x. y refers to the yth table of Unit x, i.e. Table. 1.1 is the first table in Unit 1.

In your study you will also find that the units are not of equal-length and your study time for each unit will vary.

We hope you enjoy studying the material and wish you success.

UNIT-I Parameter & their transmission

Structure

1.0 Introduction

In this unit we define parameters and their transmission. In this unit there are four sections. In Sec. 1.1 you will learn about block structure. In Sec. 1.2 you will know about parameter and their transmission. As you know Parameter transmission Subprograms need mechanisms to exchange data. Arguments - data objects sent to a subprogram to be processed Obtained through parameters and non-local references. In this reference the Results will be data object or values delivered by the subprogram it can Returned through parameters, assignments to non-local variables and explicit function values. We will now look at some of the different relationships between formal and actual parameters. That is, we will look at some of the different ways that the actual parameter is transmitted to a subprogram and the implications of the transmission methods in terms of their effects on the results of subprogram execution. The formal names of the parameter transmissions are "call by" followed by the name of the mechanism, for example "call by value". However, the terminology "pass by" followed by the name of the mechanism, for example "pass by value", is used at least as often as the formal name. In Sec. 1.3 and 1.4 you will find summary and review questions respectively.

Objectives

After studying this unit you should be able to:

- Define block structure
- Express parameters and their transmission

1.0 Block structure

A *block* is a concept that defines the scope of any definitions that it may contain. It delivers a local environment i.e., an opportunity for local definitions. The *block structure* (the textual relationship between blocks) of a programming language has a great deal of inspiration over program structure and modularity. There are three basic block structures--monolithic, flat and nested.

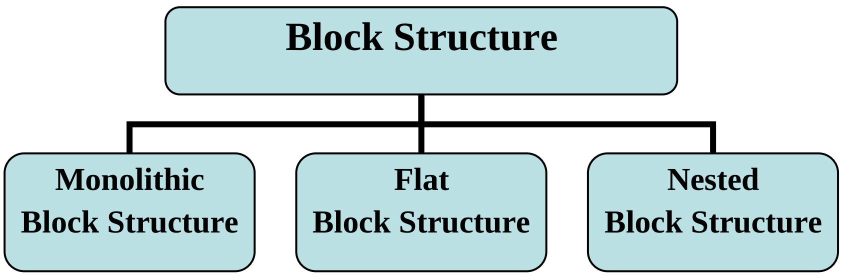


Figure 4-1: Types of block structure

Monolithic Block Structure

A program has a *monolithic* block structure if it consists of just one block. This structure is typical of BASIC and early versions of COBOL. The monolithic structure is suitable only for small programs. The scope of every definition is the entire program. Typically all definitions are grouped in one place even if they are used in different parts of the program.

Global Data
Return Address$_1$
...
Return Address$_n$

Flat Block Structure

A program has a *flat* block structure if it is partitioned into distinct blocks, an outer all inclosing block one or more inner blocks i.e., the body may contain additional blocks but the inner blocks may not contain blocks. This structure is typical of FORTRAN and C. In these languages, all subprograms (procedures and functions) are separate, and each acts as a block. Variables can be declared inside a subprogram are then local to that subprogram. Subprogram names are part of the outer block and thus their scope is the entire program along with global variables. All subprogram names and global variables must be unique. If a variable cannot be local to a subprogram then it must be global and accessible by all subprograms even though it is used in only a couple of subprograms.

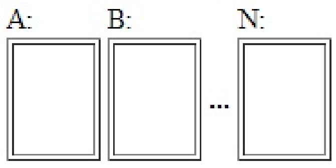
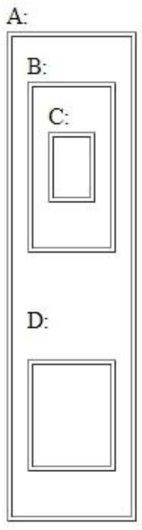


Figure-4-2 Flat Block Structure

Nested Block Structure

A program has *nested* block structure if blocks may be nested inside other blocks i.e., there is no restriction on the nesting of blocks within the body. This is typical of the block structure of the Algol-like languages. A block can be located close to the point of use. In blocks visibility is controlled by nesting. All names are visible (implicitly exported) to internally nested blocks. No names are visible (exported) to enclosing blocks. In a block, the only names visible are those that are declared in all enclosing blocks or are declared in the block, but not those declared in nested blocks.



A:

- Can see all names in A including the names A and B.
- Cannot see names in B, C, or D.

B:

- Can see all names in A and B including the names A, B, and C.
- Cannot see names in C or D.

C:

- Can see all names in A, B, and C including the names A, B, and C.
- Cannot see names in D..

D:

- Can see all names in A and D including the names A, B, and D.
- Cannot see names in B or C.

Figure-4-2 Nested Block Structure

A *local name* is one that is declared within a block for use only within that block.

A *global name* is a name that when referenced within a block refers to a name declared outside the block.

An *activation* of a block is a time interval during which that block is being executed.

The three basic block structures are sufficient for what is called *programming in the small* (PITS). These are programs which are comprehensible in their entirety by an individual programmer. However, they are not general enough for very large programs. Large programs which are written by many individuals and which must consist of modules that can be developed and tested independently of other modules. Such programming is called *programming in the large* (PITL*)*.

1.2 Parameters and their transmission

Terminology

In the definition (declaration) of a subprogram such as (in C)

```
int f (int x, float y) { ... }
```

x and y are called the *formal parameters* of function f, and we often just call them "parameters", rather than "formal parameters". (Actually, it is the value of x and the value of y that are the parameters to f, but we rarely use the complete precise terminology because it becomes too cumbersome.) In an invocation of f such as

```
k = f(n, z);
```

The values of n and z are called the *actual parameters* or *arguments* of f. If the distinction is not crucial, the term "parameter" is sometimes used to mean either a formal parameter or an actual parameter.

Parameter transmission methods

The following techniques are used to pass arguments in traditional imperative languages:

- call by value
- call by result
- call by value-result
- call by reference
- call by name

1. **call by value:** copy going into the procedure
2. **call by result:** copy going out of the procedure
3. **call by value result:** copy going in, and again going out
4. **call by reference:** pass a pointer to the actual parameter, and indirect through the pointer. If the actual parameter is a variable or an array element (*not* an expression), then the procedure can assign to the formal parameter and as a result assign into the actual parameter as well. (The more general programming language term for "variable or array element" or the like is l-value, in other words, something that can be on the left hand side of an assignment statement.)
5. **call by name:** re-evaluate the actual parameter on every use. For actual parameters that are simple variables, this is the same as call by reference. For actual parameters that are expressions, the expression is re-evaluated on each access. It should be a runtime error to assign into a formal parameter passed by name, if the actual parameter is an expression. One possible implementation: use anonymous function ("thunk") for call by name expressions. (This is traditionally used to implement call by name in languages such as Algol and Simula.)

Call by name is not supported in recent languages, but is still an important concept in programming language theory. (In newer languages, if you want the effect of call by name, pass a procedure as a parameter.)

For a pure functional language, lazy evaluation has exactly the same semantics as call by name, but will generally be more efficient. With lazy evaluation, the actual parameter is not evaluated until the formal parameter's value is needed. At that time, the actual parameter is evaluated, and the value is cached. Then, if the value of the formal parameter is needed again, the cached value will be used. (This is safe in a pure functional language, since the value of an expression will always be the same each time it is evaluated.) Thus, with lazy evaluation, an actual parameter is evaluated either 0 or 1 times.

Call by value is particularly efficient for small pieces of data (such integers), since they are trivial to copy, and since access to the formal parameter can be done efficiently. Java uses call by value for primitive data types.

Call by reference is particularly efficient for large pieces of data (such as large arrays), since they don't need to be copied.

Fortran uses call by reference
Insecurity in early FORTRANs -- passing a constant allowed the procedure to change the constant!

Algol 60 has call by name, call by value

Ada uses different designations: IN, OUT, IN OUT:

- For scalar data types (such as integers), IN is the same as call by value, OUT is the same as call by result, and IN OUT is the same as call by value result. In addition, IN parameters are local constants -- you can't assign into them.
- For compound data types (such as arrays), these can be implemented as above, or using call by reference. (You can write a test program to determine which method your compiler is using -- however, programs that rely on one implementation choice or the other are "erroneous".)

For objects (not primitive types), Java uses call-by-value with pointer semantics. In other words, Java passes a copy of the reference to the object to the called method (but doesn't copy the object itself). Smalltalk and Scheme work in exactly the same way. Unless you assign into the formal parameter, this gives the same answers as call-by-reference. However, if you assign into the formal parameter, the link with the actual parameter is broken. (Actually, Smalltalk won't let you assign into the formal parameter, but Java and Scheme will.)

An important related concept: **aliasing**. Two variables are aliased if they refer to the same storage location. A common way that aliasing can arise is using call by reference. A formal parameter can be aliased to a nonlocal variable, or two formal parameters can be aliased.

Example of call by value versus value-result versus reference

The following examples are in an Algol-like language.

```
  begin
  integer n;
  procedure p(k: integer);
     begin
     n := n+1;
     k := k+4;
     print(n);
     end;
  n := 0;
  p(n);
  print(n);
  end;
```

Note that when using call by reference, n and k are aliased.

Output:

```
call by value:        1 1
call by value-result: 1 4
call by reference:    5 5
```

Example of call by value versus call by name

```
  begin
  integer n;
  procedure p(k: integer);
     begin
     print(k);
     n := n+10;
     print(k);
     n := n+5;
     print(k);
     end;
  n := 0;
  p(n+1);
```

```
   end;
Output:
call by value:     1  1  1
call by name:      1 11 16
```

Example of call by reference versus call by name

This example illustrates assigning into a parameter that is passed by reference or by name

```
   begin
   array a[1..10] of integer;
   integer n;
   procedure p(b: integer);
      begin
      print(b);
      n := n+1;
      print(b);
      b := b+5;
      end;
   a[1] := 10;
   a[2] := 20;
   a[3] := 30;
   a[4] := 40;
   n := 1;
   p(a[n+2]);
   new_line;
   print(a);
   end;
```

Output:

```
call by reference:  30 30
               10 20 35 40

call by name:       30 40
               10 20 30 45
```

Check your progress

Q1. Differentiate between Monolithic block structure and flat block structure.

Q2. Compare call by reference and call by name

1.3 Summary

In this unit you learnt about block structure and Parameters and their transmission.

- A block is a construct that delimits the scope of any definitions that it may contain. It provides a local environment i.e., an opportunity for local definitions.
- The block structure (the textual relationship between blocks) of a programming language has a great deal of influence over program structure and modularity.
- A program has a monolithic block structure if it consists of just one block
- A program has a flat block structure if it is partitioned into distinct blocks.
- All subprogram names and global variables must be unique

1.4 Review Questions

Q1. What is block structure? Define its use in programming language.

Q2. How many types of block structure available in programming language?

Q3. What do you mean by parameters?

Q4. How many types of parameters available in programming language?

Q5. Write a method of parameter transmission.

UNIT-II Task and Shared Data Storage

Structure

2.0 Introduction

In this unit we define task and shared data storage. In this unit there are three sections. In Sec. 2.1 you will know about task and shared data. Task-shared storage, also known as the task-shared pool, is shared between all CICS tasks. As such, all synchronization to these areas is the responsibility of the applications that want to use them. You can Allocate an area of shared storage, Release the allocated shared area, Load a map set or data table, release a map set or data table. In Sec. 2.2 and 2.3 you will find summary and review questions respectively.

Objectives

After studying this unit you should be able to:

- Define Task and shard data storage
- Express mutual exclusion
- Define critical region, semaphore, and monitor.
- Express message passing, storage management.

2.1 Task and shard data storage

Task, co-routines and exception handlers all have a problem with mutual exclusion. Operation on a set of shared data.

Mutual Exclusion: - A mutual exclusion (mutex) is a program object that stops concurrent access to a shared resource. This concept is used in concurrent programming with a critical section, a piece of code in which processes or threads access a shared resource. Only one thread owns the mutex at a time, thus a mutex with a unique name is created when a program starts. When a thread holds a resource, it has to lock the mutex from other threads to prevent concurrent access of the resource. Upon releasing the resource, the thread unlocks the mutex.

Potential Solutions

Critical Region

- A critical region is a section of code that is always executed under mutual exclusion.
- Critical regions shift the responsibility for enforcing mutual exclusion from the pro-grammar (where it resides when semaphores are used) to the compiler.
- They consist of two parts:
    - Variables that must be accessed under mutual exclusion.
    - A new language statement that identifies a critical region in which the variables are accessed.

Example: *This is only pseudo-Pascal-FC — Pascal-FC doesn't support critical regions*

```
var
v : shared T;
...
region v do begin
...
end;
```

All critical regions that are 'tagged' with the same variable have compiler-enforced mutual exclusion so that only one of them can be executed at a time:

Process A:

region V1 do

```
begin
{ Do some stuff. } end;
region V2 do begin
{ Do more stuff. } end;

Process B:
region V1 do
begin
{ Do other stuff. } end;
```

Here process A can be executing inside its V2 region while process B is executing inside its V1 region, but if they both want to execute inside their respective V1 regions only one will be permitted to proceed.
Each shared variable (V1 and V2 above) has a queue associated with it. Once one process is executing code inside a region tagged with a shared variable, any other processes that attempt to enter a region tagged with the same variable are blocked and put in the queue.

Semaphore: - A semaphore, in operating systems terms, is very similar to a gate in real life. A semaphore protects access to a resource that is shared between two or more processes, or threads, but cannot be written to by multiple processes at the same time, e.g., a memory location. A semaphore typically has test, lock, test and lock, and unlock operations.

To access the resource protected by the semaphore, the thread first must acquire the lock, via the "lock" or "test and lock" operation. Once the semaphore is acquired (locked) the thread uses the resource. While the semaphore is locked, any other thread attempting to acquire the semaphore is told to wait. When the thread holding the semaphore has completed its operations, the semaphore is unlocked (released). Any thread waiting for the semaphore then try to acquire (lock) the semaphore so the resource may be accessed again.

Conditional Critical Regions

Critical regions aren't equivalent to semaphores. As described so far, they lack condition synchronization. We can use semaphores to put a process to sleep until some condition is met (e.g. see the bounded-buffer Producer-Consumer problem), but we can't do this with critical regions.

Conditional critical regions provide condition synchronization for critical regions:

```
region v when B do begin

    ...

    end;
```

Where B is a Boolean expression (usually B will refer to v). Conditional critical regions work as follows:

1. A process wanting to enter a region for v must obtain the mutex lock. If it cannot, then it is queued.

2. Once the lock is obtained the boolean expression B is tested. If B evaluates to true then the process proceeds, otherwise it releases the lock and is queued. When it next gets the lock it must retest B.

Implementation

Each shared variable now has two queues associated with it. The main queue is for processes that want to enter a critical region but find it locked. The event queue is for the processes that have blocked because they found the condition to be false. When a process leaves the conditional critical region the processes on the event queue join those in the main queue. Because these processes must retest their condition they are doing something akin to busy-waiting, although the frequency with which they will retest the condition is much less. Note also that the condition is only retested when there is reason to believe that it may have changed (another process has finished accessing the shared variable, potentially altering the condition). Though this is more controlled than busy-waiting, it may still be sufficiently close to it to be unattractive.

Limitations

- Conditional critical regions are still distributed among the program code.
- There is no control over the manipulation of the protected variables — no information hiding or encapsulation. Once a process is executing inside a critical region it can do whatever it likes to the variables it has exclusive access to.
- Conditional critical regions are more difficult to implement efficiently than semaphores.

Monitors

- Consist of private data and operations on that data.
- Can contain types, constants, variables and procedures.
- Only the procedures explicitly marked can be seen outside the monitor.
- The monitor body allows the private data to be initialized.
- The compiler enforces mutual exclusion on a particular monitor.
- Each monitor has a boundary queue, and processes wanting to call a monitor routine join this queue if the monitor is already in use.
- Monitors are an improvement over conditional critical regions because all the code that accesses the shared data is localized.

A monitor has ***four*** components as shown below: initialization, private data, monitor procedures, and monitor entry queue. The ***initialization*** component contains the code that is used exactly once when the monitor is created, the ***private data*** section contains all private data, including private procedures that can only be used *within* the monitor. Thus, these private items are not visible from outside of the monitor. The ***monitor procedures*** are procedures that can be called from outside of the monitor. The ***monitor entry queue*** contains all threads that called monitor procedures but have not been granted permissions.

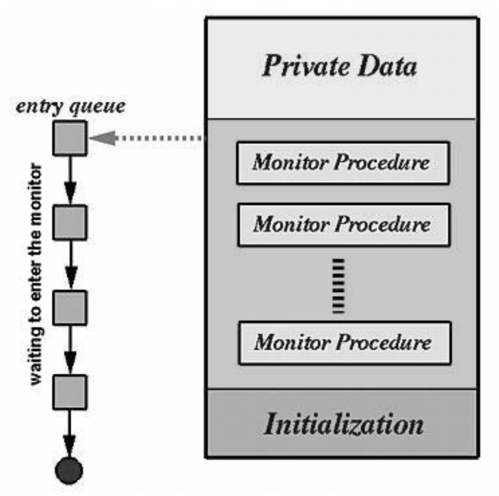


Figure 2-1 The monitor

Therefore, a monitor looks like a class with the initialization, private data and monitor procedures corresponding to constructors, private data and methods of that class. The only major difference is that classes do not have entry queues.

Message Passing

The message passing model is one of several computational models for conceptualizing program operations. The message passing model is defined as:

- set of processes having only local memory
- processes communicate by sending and receiving messages
- the transfer of data between processes requires cooperative operations to be performed by each process (a send operation must have a matching receive)

Other models include:

- data parallelism
  data partitioning determines parallelism
- shared memory
  multiple processes sharing common memory space
- remote memory operation
  set of processes in which a process can access the memory of another process without its participation
- threads
  a single process having multiple (concurrent) execution paths
- combined models composed of two or more of the above

Note: these models are machine/architecture independent; any of the models can be implemented on any hardware given appropriate operating system support. An effective implementation is one which closely matches its target hardware.

The message passing model has gained wide use in the field of parallel computing due to advantages that include:

- Hardware match - The message passing model fits well on parallel supercomputers and clusters of workstations which are composed of separate processors connected by a communications network.
- Functionality - Message passing offers a full set of functions for expressing parallel algorithms, providing the control not found in data-parallel and compiler-based models.
- Performance - Effective use of modern CPUs requires management of their memory hierarchy, especially their caches. Message passing achieves this by giving programmer explicit control of data locality.

The principle drawback of message passing is the responsibility it places on the programmer. The programmer must explicitly implement a data distribution scheme and all inter-process communication and synchronization.

In so doing, it is the programmer's responsibility to resolve data dependencies and avoid deadlock and race conditions.

Storage Management

Most languages also have the ability to dynamically allocate and free the storage for data objects in a somewhat arbitrary manner. These objects stores in the heap. In many implementations the stack begins at one end of memory in a computer and the heap is at the other end. If the two structures—the stack and the help—ever meet, then the program is out of memory and halts. In this module, we discuss various methods for managing the heap storage requirements for a programming language. Storage management for data is one of the central concerns of the programmer, language implementer, and language designer. In this section the various problems and techniques in storage management arc considered.

Typically languages contain many features or restrictions that may be explained only by a desire on the part of the designers to allow one or another storage-management technique to be used. Take, for example, the restriction in FORTRAN to non-recursive subprogram calls Recursive calls could he allowed in FORTRAN without change in the syntax, but their implementation would require a run-time stack of return points—a structure necessitating dynamic storage management during execution. Without recursive calls, FORTRAN may be implemented with only static storage management. It is a dramatic break with the past (hat. in the latest FORTRAN 90 standard. limited dynamic storage is permitted. Pascal is carefully designed to allow stack-based storage management. LISP to allow garbage collection and so on.

Although each language design ordinarily permits the use of certain storage-management techniques, the details of the mechanisms and their representation in hardware and software are the task of the implementer, For example, although the LISP design may imply a free-space list and garbage collection as the appropriate basis for storage management, there are several different garbage-collection techniques from which the implementer must choose based on available hardware and software.

Although the programmer is also deeply concerned with storage management and must design programs that use storage efficiently, the programmer is likely to have little direct control over storage. A program affects storage only indirectly through the use or lack of use of different language features. This is made more difficult by the tendency of both language designers and implementors to treat storage management as a machine-dependent topic that should not be directly discussed in language manuals. Thus, it is often difficult for a programmer to discover what techniques are actually used.

ELEMENTS REQUIRING STORAGE

The programmer tends to view storage management largely in terms of storage of data and translated programs. However, run-time storage management encompasses many other areas. Some, such as return points for subprograms, have been touched on previously. Let us look at the major program and data elements requiring storage during program execution.

Code segments for translated user programs: - A major block of storage in any system must be allocated to store the code segments representing the translated form of user programs, regardless of whether programs are hardware - or software - interpreted. In the former case, programs are blocks of executable machine code: in the latter case, blocks are in some intermediate form.

System run-time programs: -Another substantial block of storage during execution must be allocated to system programs that support the execution of the user programs. These may range from simple library mutates, such as Sine, Cosine, or print-string functions, to software interpreters or translators present during execution. Also included here are the routines that control run-time storage management.

User-defined data structures and constants: - Space for user data must be allocated for all data structures declared in or created by user programs including constants. Subprogram return points Subprograms may he invoked from different points in a program. Therefore, internally generated sequence-control information, such as subprogram return points, coroutine resume points, or event notices for scheduled subprograms, must be allocated storage.

Referencing environments: - Storage of referencing environments (identifier associations) during execution may require substantial space, as, for example, the LISP A-list.

Temporaries in expression evaluation: - Expression evaluation requires the use of system-defined temporary storage for the intermediate results of evaluation, For example in evaluation of the expression (x + y) x (u + v), the result of the first addition may have to be stored in a temporary while the second addition is performed. When expressions involve recursive function calls a potentially unlimited number of temporaries may be required to store partial results at each level of recursion.

Temporaries in parameter transmission: - When a subprogram is called, a list of actual parameters must be evaluated and the resulting values stored in temporary storage until evaluation of the entire list is complete. Where evaluation of one parameter may require evaluation of recursive function calls, a potentially unlimited amount of storage may be required, as in expression evaluation.

Input-output buffers: - Ordinarily input and output operations work through buffers, which serve as temporary storage areas where data are stored between the time of the actual physical transfer of the data to or from external storage and the program-initiated input and output operations.

Miscellaneous system data: - In almost every language implementation storage is required for various system data: tables, status information for input-output, and various miscellaneous pieces of state information (e.g. reference counts or garbage-collection bits).

Besides the data and program elements requiring storage, various operations may require storage to be allocated or freed. The following are the major operations:

Subprogram calls and return operations: - "The allocation of storage for a subprogram activation record, the local referencing environment, and other data on call of a subprogram is often the major operation requiring storage allocation. The execution of a subprogram return operation usually requires freeing of the storage allocated during the call. Data structure creation and destruction operations. If the language provides operations that allow new data structures to be created at arbitrary points during program execution (rather than only on subprogram entry-e.g., new in Java), then these operations ordinarily require storage allocation that is separate from that allocated on subprogram entry. The language may also provide an explicit destruction operation, such as the Pascal dispose and the C free function, which may require that storage be freed. Java has no explicit destruction operation: garbage collection is employed.

Component insertion and deletion operations: - If the language provides operations that insert and delete components of data structures, storage allocation and freeing may be required to implement these operations (e.g. the Perl push function adds an element to an array).

Although these operations require explicit storage manage neat, many other operations require some hidden storage management to take place. Much of this storage management activity involves the allocation and freeing of temporary storage for housekeeping purposes (e.g., during expression evaluation and parameter transmission).

Check your progress

Q1. Define mutual exclusion and critical region.

Q2. Explain monitor in detail.

2.2 Summary

In this unit you learnt about Task and shard data storage requirement for major runtime elements

- Task, co-routines and exception handlers all have a problem with mutual exclusion.
- A critical region is a section of code that is always executed under mutual exclusion.
- Critical regions aren’t equivalent to semaphores.
- Each shared variable now has two queues associated with it.
- A monitor has ***four*** components: initialization, private data, monitor procedures, and monitor entry queue.

2.3 Review Questions

Q1. Define the term task in programming language.

Q2. How shared data stored in computer’s memory? Justify your answer.

Q3. What is a critical region? Define with example.

Q4. What is a monitor? Explain in detail.

Q5. Write a short note on Storage Management.

UNIT-III System Controlled Storage Management

Structure

3.0 Introduction

In this unit we defined System Controlled Storage Management. There are four sections in this unit. In the first section i.e. Sec. 3.1 you will learn about program and system controlled storage management. Storage management also known as Run-time storage it includes Code segment-translated, user program, System programs, User defined data structures, Sub-program return points, Referencing environments, Temporaries for expression evaluation, I/O buffers and System data. In Sec. 3.2 you will learn about storage management phase. In Sec. 3.3 and 3.4 you will find summary and review questions respectively.

Objectives

After studying this unit you should be able to:

- Express storage management.
- Define stages of storage management.
- Describe user program, system programs etc.

3.1 Program and system controlled storage management.

To what extent should the programmer be allowed to directly control storage management? On the one hand, C has become very popular because it allows extensive programmer control over storage via malloc and free, which allocate and free storage for programmer-defined data structures On the other hand, many high-level languages allow the programmer no direct control: storage management is affected only implicitly through the use of various language features.

The difficulty with programmer-controlled storage management is twofold: It may place a large and often undesirable burden on the programmer and it may also interfere with the necessary system-controlled storage management. No high-level language can allow the programmer to shoulder the entire storage-management burden, For example, the programmer can hardly he expected to be concerned with storage for temporaries, subprogram return points, or other sys-tem data. At best a programmer might control storage management for local data (and perhaps programs). Yet even simple allocation and freeing of storage for data structures, as in C, are likely to permit generation of garbage and dangling references. Thus, programmer-controlled storage management is dangerous to the programmer because it may lead to subtle errors or loss of access to available storage. Programmer-controlled storage management also may interfere with system-controlled storage management, in that special storage areas and storage-management routines may be required for programmer-controlled storage allowing less efficient use of storage overall.

The advantage of allowing programmer control of storage management is that it is often extremely difficult for the system to determine when storage may be most effectively allocated and freed. The programmer often knows quite precisely when a particular data structure is needed or when it is no longer needed and may be freed.

This dilemma is by no means trivial and is (then at the heart of what language to use on a given project. Does one provide protection for the programmer by using a language with strong typing and effective storage-management features with a corresponding decrease in performance? Or does one need the performance characteristics (e.g., storage management and execution speed) with an increase in risk that the program may contain errors and fail during execution? This is one of the fundamental debates in the software engineering community. This book does not solve this debate but a major goal is to provide the reader with the appropriate details to make an intelligent choice in these matters.

3.2 Storage Management Phases

It is convenient to identify three basic aspects of storage management:

Initial allocation: At the start of execution, each piece of storage may either be allocated for some use or free. If free initially, it is available for allocation dynamically as execution proceeds, Any storage-management system requires sonic technique for keeping track of free storage as well as mechanisms for allocation of free storage as the need arises during execution.

Recovery: Storage that has been allocated and used, and that subsequently becomes available, must be recovered by the storage manager for reuse. Recovery may be very simple, as in the repositioning of a stack pointer, or very complex, as in garbage collection.

Compaction and reuse: Storage recovered may be immediately ready for reuse or compaction may be necessary to construct large blocks of free storage from small pieces. Reuse of storage ordinarily involves the same mechanisms as initial allocation.

Many different storage-management techniques are known and are in use in language implementations. It is impossible to survey them all, but a relative handful suffices to represent the basic approaches. Most techniques are variants of one of these basic methods.

Check your progress

Q1. Define Program and system controlled storage management.

Q2. Storage Management Phases

3.3 Summary

In this unit you learnt Program and system controlled storage management. You also learn about how program works and how system controlled storage management works.

- C has become very popular because it allows extensive programmer control over storage via malloc and free.
- Many high-level languages allow the programmer no direct control.
- No high-level language can allow the programmer to shoulder the entire storage-management burden.
- Programmer-controlled storage management also may interfere with system-controlled storage management.
- The advantage of allowing programmer control of storage management is that it is often extremely difficult for the system to determine when storage may be most effectively allocated and freed.

3.4 Review Questions

Q1. What do you mean by program controlled storage management?

Q2. How system controlled storage management works? Define.

Q3. What are storage management phases?

UNIT-IV Storage Management

Structure

4.0 Introduction

This is fourth unit of this block. This unit focused on storage management. In this unit there are six sections. As you have studied earlier, the storage management is an executing program uses memory (storage) for many different purposes, such as for the machine instructions that represent the executable part of the program, the values of data objects, and the return location for a function invocation.

In general, storage is managed in three phases: 1) allocation, in which needed storage is found from available (unused) storage and assigned to the program; 2) recovery, in which storage that is no longer needed is made available for reuse; and 3) compaction, in which blocks of storage that are in use but are separated by blocks of unused storage are moved together in order to provide larger blocks of available storage. (Compaction is desirable, but usually it is difficult or impossible to do in practice so it is not often done.) These phases may be repeated many times during the execution of a program.

In Sec. 4.5 and 4.6 you will find summary and review questions respectively.

Objectives

After studying this unit you should be able to:

- Express static based storage management, Stack based storage management
- Define fixed size heap storage management.
- Describe variable size heap storage management

4.1 STATIC STORAGE MANAGEMENT

The simplest form of allocation is static allocation—allocation during translation that remains fixed throughout execution. Ordinarily storage for the code segments of user and system programs is allocated statically, as is storage for I/O buffers and various miscellaneous system data. Static allocation requires no run-time storage-management software, and. of course; there is no concern for recovery and reuse.

In the usual FORTRAN implementation, all storage is allocated statically. Each subprogram is compiled separately; with the compiler setting up the code segment (including an activation record) containing the compiled program its data areas, temporaries, return-point location, and miscellaneous items of system data, the loader allocates space in memory for these compiled blocks at load time, as well as space for system run-time routines. During program execution, no storage management needs to take place.

Static storage allocation is efficient because no time or space is expended for storage management during execution. The translator can generate the direct I-value addresses for all data items. However, it is incompatible with recursive subprogram calls, with data structures whose size is dependent on computed or input data, and with many other desirable language features. In the next few subsections of this unit we discuss various techniques for dynamic (run-time) storage management. However, the reader should not lose sight of the importance of static allocation. For many programs, static allocation is quite satisfactory. Two of the most widely used programming languages. FORTRAN and COBOL are designed for static storage allocation (although as stated previously. FORTRAN 90 now permits dynamic arrays and recursive procedures). Languages like C. which have dynamic storage also permit static data to be created for efficient execution.

4.2 Stack based storage management

Free storage at the start of execution is set up as a sequential block in memory. As storage is allocated it is taken from sequential locations in this static block beginning at one end. Storage must be freed in the reverse order of allocation so that a block of storage being freed is always at the top of stack. A stack pointer is all that is needed to control storage management. The stack pointer always points to the top of the stack, the next available word of free storage in the stack block. All storage in use lies in the stack below the location pointed to be the stack pointer. All free storage lies above the pointer. When a block of K locations is to be allocated the pointer is simply moved to point K locations farther up the stack area. When a block of K locations is freed the pointer is moved back K locations. Compaction occurs automatically as part of freeing storage & makes it available for reuse. When a subprogram is called a new activation record is created on the top of the stack termination causes is deletion from the stack.

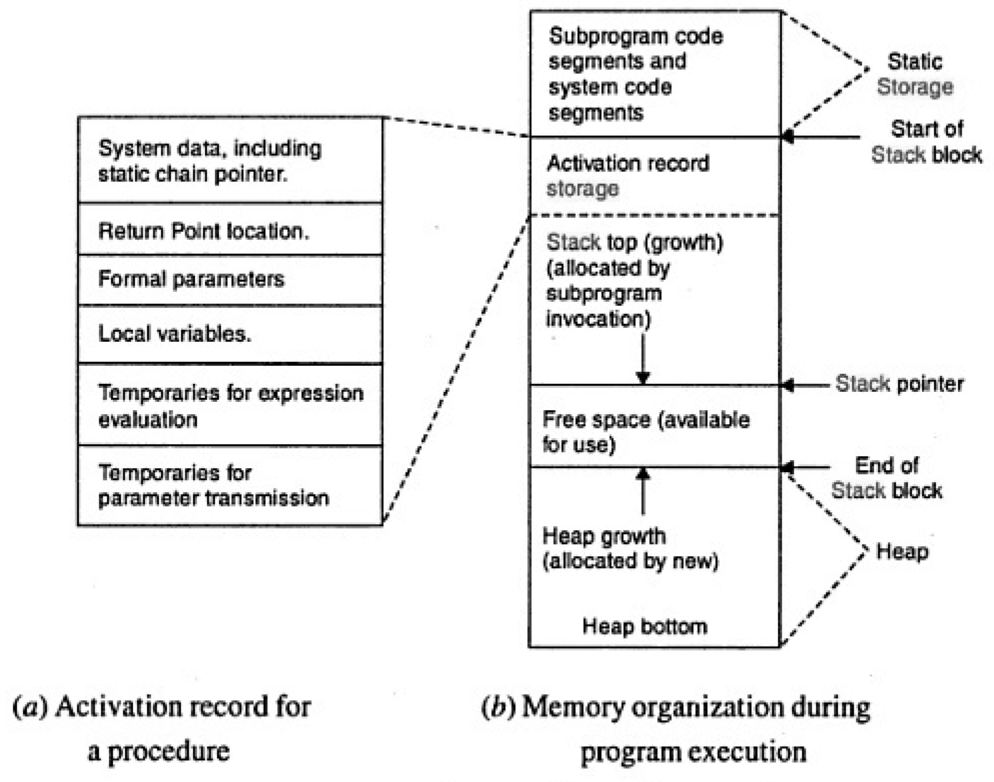

System data, including static chain pointer.
Return Point location.
Formal parameters
Local variables.
Temporaries for expression evaluation
Temporaries for parameter transmission
Subprogram code segments and system code segments
Static Storage
Start of Stack block
Activation record storage
Stack top (growth) (allocated by subprogram invocation)
Stack pointer
Free space (available for use)
End of Stack block
Heap growth (allocated by new)
Heap
Heap bottom
(a) Activation record for a procedure
(b) Memory organization during program execution

Figure 4-1 Pascal Memory Organization

Most PASCAL implementations are built around a single central stack of activation records for subprograms together with a statically allocated area, consisting of all the variable items of information associated with a given subprogram activation heap storage area used for storage allocated by new & freed by dispose. Stack based storage management is the most widely used technique for run-time storage management. The division of storage into an area allocated statically an area allocated as a stack and an area allocated as a stack seen in PASCAL and LISP. Ada 'C' and PROLOG also use this tripartite division of memory into areas managed in different ways.

4.3 Heap Storage Management (Fixed-size element)

A heap is a block of storage within which pieces are allocated and freed in some relatively unstructured manner. Here are problems of storage allocation. Recovery compaction & reuse may be severe. The need for heap storage arises when a language permits storage to be allocated and freed at arbitrary points during program execution. As when a language allows creation destruction or extension of program points, For example, in ML two lists may be concatenated to create a new list at any arbitrary point during execution. In both ML and LISP storage may also be freed at unpredictable points during execution. Assuming the heap occupies a contiguous block of memory, we conceptually divide the heap block into a sequence of K elements, and each N word long, such as KxN is the size of the heap. Whenever an element is needed one of these is allocated from the heap. Whenever an element is freed, it must be one of these original heap elements. If a structure is destroyed before all access paths to the structure have been destroyed any remaining access paths become dangling references. If the last access path to a structure is destroyed without the structure itself being destroyed and the storage recovered, then the structure becomes garbage. In context of heap storage management element that has been returned to free-space list. A garbage element is one that is available for reuse but not on the free space list and thus it has become in accessible. The explicit return of heap storage facilities creating garbage and dangling reference

```
int *iptr, *qptr;                  int *iptr, *qptr;
………………              or      ……………
iptr=malloc(sizeof (int));     iptr = malloc (sizeof(int));
iptr=qptr;                         qptr=iptr;
                                   free(iptr);
```

One alternative called reference counts requires explicit return but provides a way of creaking the no of pointers to a given element so that no dangling references are created. Second alternative, called garbage collection, and is to allow garbage to be crated but no dangling references. Later if the free space list becomes exhausted a garbage collector mechanism is invoked to identify and recover the garbage. Initially K elements are linked together to form as free space list (i.e., the first word of each item on the free list points to the first word of the next item on the free list). To allocate an element the first element on the free space list is removed from the list and a pointer to it is returned to the operation requesting the storage. When an element is freed it is simply linked back at the head of the free space list.

The three ways to recover the storage & returned to free-space list

(a) Explicit return by programmer or system

When an element that has been in use becomes available for reuse, it must be explicitly identified as "free" and returned to the free space list e.g., a call to dispose in PASCAL or explicit call of a free run-time. Explicit return is natural recovery technique for heap storage but unfortunately it is not always feasible because of following problems:

- Garbage collection
- Dangling references

(b) Reference Counts

Within each element in the heap some extra space is provided for a reference counter. The reference counter contains the reference count indicating the no of pointer to that element that exist. When an element is initially allocated from the free space list, its reference count is set to 1 each time a new pointer to the element is created, its reference count is increased by 1 each time a pointer is destroyed the reference count is decreased by 1. When the reference count of an element reaches zero, the element is free and may be returned to the free space list. Reference count allows both garbage and dangling reference to be avoided in most situations.

e.g., cdr in LISP is a primitive operation on linked list, which given a pointer to one element on a linked list, returns a pointer to the next element in the list. Free statement decrements the reference count of the structure by 1. Only if the count then is zero is the structure actually returned to the free space list. A non-zero reference count indicates that the structure is still accessible and the free command should be ignored. Reference count leads to the cost of maintaining them since testing, incrementing and decrementing occur continuously throughout execution often causing a substantial decrease in execution efficiency. In addition there is the cost of the extra storage for the reference count.

(c) Garbage Collection

Dangling reference result when storage is freed "too soon" and garbage wheat storage is not freed until "too late". Garbage generation is clearly to be preferred in order to avoid dangling references. It is better not to recover storage. At all than to recover it too soon, when the free-space list is entirely exhausted and more storage is needed, the computation is suspended temporarily and an extra ordinary procedure instituted a garbage collection which identifies garbage elements in the heap and return then to the free-space list. The original computation is then resumed and garbage again accumulates until the free space list is exhausted, at which time another garbage collection is initiated and soon. Two stages in garbage collection arc:

- **Mark:** In the first stage each element in the heap which is active. i.e., which is part to an accessible data structure must be marked. Each element must contain a garbage collection bit set initially to "on". The marking algorithm sets the garbage collection bit of each active element "off ".
- **Sweep:** Once the marking algorithm has marked active element (i.e., still in use) all those remaining whose garbage collection bit is "on" are garbage and may be returned to the free space list. A simple sequential scan of the heap is sufficient. The garbage collection bit of each element is checked as it is encountered in the scan. If "off-, element is passed over: if "on", the element is linked into the free-space list. All garbage collection bits are reset to "on" during the scan.
- Unfortunately inspection of an element can't indicate its status because there is nothing intrinsic to a garbage element to indicate that it is no longer accessible from another active element. An element is active it true is a pointer to it from outside the heap or from another active heap element. These new elements are then marked and searched for other pointers and soon.

Check your progress

Q1. Explain static storage management.

Q2. Define Stack based storage management

4.4 Heap Storage Management: Variable Size Elements

Heap storage management where variable size elements are allocated & recovered is more difficult than with fixed sized elements although. The major difficulties with variable size elements concern reuse of recovered space. We maintain a free space in blocks of as large a size as possible. Initially then we consider the heap as simply one large block of free storage. A heap pointer is appropriate for initial allocation when a block of N words is requested, the heap pointer is advanced by N and the original heap pointer value returned as a pointer to the newly allocated element. Eventually the heap pointer reaches the end of the heap block. Some of the free space back in the heap must now be reused.

Two possibilities for Reuse

- Use free space list for allocation, searching the list for an appropriate size block and returning any leftover space to the free list after the allocation.
- Compact the free space by moving all the active elements to one end of the heap. Leaving the free space as a single block at the end & resetting the heap pointer to the beginning of the block.

For reusing directly from a free space list we can have certain memory allocation methods:

- **First-fit method:** When an N-word block is needed, the free-space list is scanned for the first block of N or more words, which is then split into an N- word block, and the remainder which is returned to the free-space list.
- **Best-fit method:** When an N-word block is needed the free-space list is scanned for the block with the minimum number of words greater than or equal to N. This block is allocated as a unit, if it has exactly N words, or is split and the remainder returned to the free space list.
- **Worst-fit method:** When an N-word block is needed, the free-space list is scanned for the block with the maximum number of words greater than or equal to N. This block is allocated as a unit, if it has exactly N words, or is split and the remainder returned to the free space list.

For recovery with variable size block the reference counts may be used and also garbage collections. Here garbage collection technique also consists of marking phase followed by collecting phase while. Colleting a problem in determining the boundaries between elements arises without which garbage cannot be collected. The simplest solution is to maintain along with the garbage collection bit in the first word of each block active or not, an integer length indicator specifying the length of the block. With variable size elements a problem of fragmentation arise which can be solved by:

- **Partial compaction:** If active blocks cannot be shifted or it is too costly to do so, then only adjacent free blocks on the free-space list may be compacted.
- **Full compaction**: If active blocks can be shifted, then all active blocks may be shifted to one end of the heap, leaving all free space at the other in a contiguous block. Full compaction requires that when an active block is shifted, all pointers to that block be modified to the new location.

4.5 Summary

In this unit you learnt about Static based storage management, Stack based storage management, fixed size heap storage management, Variable size heap storage management

- The simplest form of allocation is static allocation—allocation during translation that remains fixed throughout execution.
- Static allocation requires no run-time storage-management software
- FORTRAN implementation, all storage is allocated statically.
- Static storage allocation is efficient because no time or space is expended for storage management during execution.
- Free storage at the start of execution is set up as a sequential block in memory.

4.6 Review Questions

Q1. What do you mean by static based storage management? Define with example.

Q2. What is the purpose storage management? Why we use it? Explain with example.

Q3. What do you mean by stack based storage management? Define with example.

Q4. Define fixed size heap storage management.

Q5. What is variable size heap storage management? Define with example.

BIBLIOGRAPHY

Christopher Strachey. Fundamental concepts in programming languages. Lecture Notes, International Summer School in Computer Programming, Copenhagen, August 1967. Reprinted in *Higher-Order and Symbolic Computation*, 13(1/2), pp. 1-49, 2000.

Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Programming Research Group Technical Monograph PRG-11, Oxford Univ. Computing Lab., 1974. Reprinted in *Higher-Order and Symbolic Computation*, vol. 13 (2000), pp. 135-152.

D. A. Turner. A new implementation technique for applicative languages. *Software - Practice and Experience*, 9(1):31-49, Jan 1979.

David Ungar and Randall B. Smith. Self: The power of simplicity. *Lisp and Symbolic Computation*, 4(3):187-205, 1991.

N. Wirth. The programming language Pascal (revised report). Technical report 5, Dept. Informatik, Inst. Für Computersysteme, ETH Zürich, Zürich, Switzerland, Jul 1973.

Niklaus Wirth and C. A. R. Hoare. A contribution to the development of ALGOL. *Communications of the ACM*, 9(6):413-432, June 1966.

Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38-94, November 1994.

See R. Cezzar, *A Guide to Programming Languages: Overview and Comparison* (1995),

T. W. Pratt and M. V. Zelkowitz, *Programming Languages: Design and Implementation* (3d ed. 1996);

C. Ghezzi and M. Jazayem,*Programming Language Concepts* (3d ed. 1997);

R. W. Sebasta, *Concepts of Programming Languages* (4th ed. 1998).

Burnett, Margaret M. and Marla J. Baker, A Classification System for Visual Programming Languages, Journal of Visual Languages and Computing, 287-300, September 1994.

Chang, S. K., Margaret Burnett, Stefano Levialdi, Kim Marriott, Joseph Pfeiffer, and Steven Tanimoto, The Future of Visual Languages. In 1999 IEEE Symposium on Visual Languages, Tokyo, Japan, Sept. 1999. Pages 58 to 61.