Procedural and Object-Oriented Programming Principles, Fundamental Data Structures

Draft

José M. Garrido

Department of Computer Science
College of Computing and Software Engineering

Kennesaw State University

PUBLISHED BY J GARRIDO

BOOK-WEBSITE.COM

Contents

1. Problem Solving and Computing

1.1 Introduction

Computer problem solving attempts to derive a computer solution to a real-world problem, and a computer *program* is the implementation of the solution to the problem. Computer problem solving is applied in various areas of science, engineering, and business to solve small to large-scale and complex problems.

A computer program consists of data definitions and a sequence of instructions. The instructions allow the computer to manipulate the input data to carry out computations and produce desired results when the program executes. A program is normally written in an appropriate programming language.

This chapter discusses problem solving principles and presents elementary concepts and principles of problem solving and programs.

1.2 Computer Problem Solving

Problem solving is the process of developing a computer solution to a given real-world problem. The most challenging aspect of this process is discovering or formulating the method to solve the problem. This method of solution is described by an algorithm. A general process of problem solving involves the following tasks:

1. Understanding the problem
2. Describing the problem in a clear, complete, and unambiguous form
3. Designing a solution to the problem (algorithm)
4. Developing a computer solution to the problem.

An *algorithm* is a clear, detailed, precise, and complete description of the sequence of steps to be performed in order to produce the desired results. It describes (informally) the operations on the given data and involves a sequence of instructions that are to be performed on the input data in order to produce the desired results (output data).

A program is a computer implementation of an algorithm and consists of a set of data definitions and sequences of instructions. The program is written in an appropriate programming language and it indicates the computer how to transform the given data into correct results by performing a sequence of computations on the data. An algorithm is described in a semi-formal notation such as *pseudo-code* and *flowcharts*.

1.3 Elementary Concepts

An essential approach in problem solving is to use mathematical entities such as numbers, functions, and sets to describe properties and their relationships to problems and real-world systems. Computer problem solving consists of:

- Applying a formal software development process
- Applying *Computer Science* concepts, principles and methods, such as:
  - Abstraction and decomposition
  - Programming principles
  - Data structures
  - Algorithm structures
  - Concurrency and synchronization
  - Modeling and simulation
  - Multi-threading, parallel, and distributed computing

Abstraction is a very important principle in problem solving. This is extremely useful in dealing with large and complex problems or systems. Abstraction is the hiding of the details and leaving visible only the essential features of a particular system.

One of the critical tasks in problem solving is representing the various aspects of a system at different levels of abstraction. A good abstraction captures the essential elements of a system, and purposely leaves out the rest. Reasoning about a problem and formulating a computer solution involves the following elements:

- Reasoning about computer problem solving
- The ability to describe the requirements of a problem and, if possible, design a mathematical solution that can be implemented in a computer
- The solution usually requires *multi-disciplinary* and *inter-disciplinary* approaches to problem solving
- The solution normally leads to the implementation, and verification of a *computer program*

One of the goals of the general approach to problem solving is modeling the problem at hand, building or implementing the resulting solution using appropriate tools, development environments, some appropriate programming language, such as SPSCL and C.

1.4 A Process for Program Development

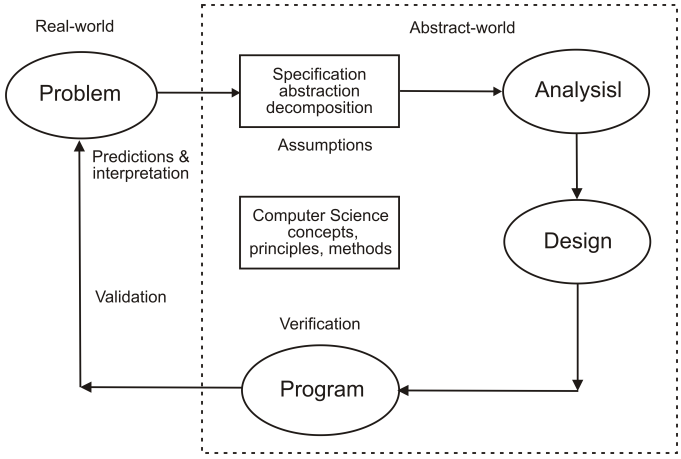

Figure 1.1: Program Development

The process of developing programs consists of a sequence of activities or stages that starts with the definition of the problem goals and terminates with a validated program. The process is carried out in a possibly iterative manner. Figure 1.1 illustrates a simplified process for developing computer programs. This process involves the following general steps:

1. Definition of the *problem statement*. This statement must provide the description of the purpose of the problem, the questions it must help to answer, and the type of expected results relevant to these questions.
2. Definition of the *problem specification* to help define the conceptual model of the problem to be solved. This is a description of what is to be accomplished with the solution to be constructed; and the assumptions (constraints), and domain laws to be followed. Ideally, the specification describes what is the problem and it should be clear, precise, complete, concise, and understandable.
3. Definition of the *design of the solution*. This stage involves deriving a representation of the problem solution using mathematical entities and expressions and the details of the algorithms, which describes how to solve the problem.
4. *Program implementation*. This stage is carried out with appropriate software tools, in a general-purpose high-level programming language, such as SPSCL, C, Ada, C++, or Java. The main tasks in this phase are coding/editing, compiling, and preliminary testing of the program.
5. *Verification*. This phase consists of applying methods that attempt to document and prove the correctness of the implementation.
6. *Validation* of the program. This stage compares the outputs of the verified program with data and properties about the real system. Validation attempts to evaluate the extent to which the implementation promotes understanding of the problem.

General Process of Software Development

For large software systems, a general software development process involves carrying out a sequence of well-defined phases or activities. The process is also known as the *software life cycle*.

The simplest approach for using the software life cycle is the *waterfall model*. This model represents the sequence of phases or activities needed to develop the software system through installation and maintenance of the software. In this model, the activity in a given phase cannot be started until the activity of the previous phase has been completed.

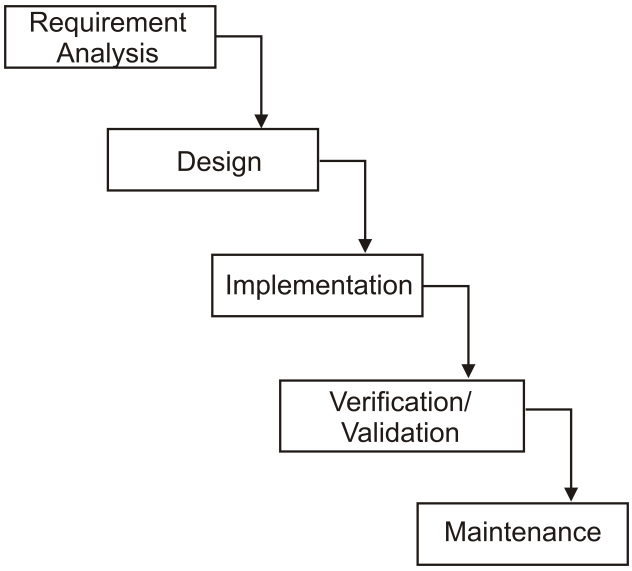

Figure 1.2: The waterfall model.

Figure 1.2 illustrates the sequence of phases that are performed in the waterfall software life cycle. The various phases of the software life cycle are the following:

1. *Analysis*, which results in documenting the problem description and what the problem solution is supposed to accomplish.
2. *Design*, which involves describing and documenting the detailed structure and behavior of the system model.
3. *Implementation* of the software using a programming language.
4. *Testing* and verification of the programs.
5. Installation that results in delivery, installation of the programs.
6. *Maintenance*.

There are some variations of the waterfall model of the life cycle. These include returning to the previous phase when necessary. More recent trends in software development have emphasized an iterative approach, in which previous stages can be revised and enhanced.

A more complete model of software life cycle is the *spiral model* that incorporates the construction of *prototypes* in the early stages. A prototype is an early version of the application that does not have all the final characteristics. Other development approaches involve prototyping and rapid application development (RAD).

1.5 Modules

To design and implement large and complex problems, two principles are essential decomposition and abstraction. A problem may be too large and complex to solve as a single unit. An important strategy in problem solving is *divide and conquer*. It consists of partitioning a large problem into *sub-problems* that are smaller, easier to solve, and easier to manage. Each sub-problem can be solved individually. These sub-problems or modules need to be well organized in order to achieve the overall solution to the problem.

The technique used in modular design is *top-down design*. A more technical term for this technique is *decomposition*. The top module is the main module and includes the high-level solution, or the *big picture* design. The modules at the next lower level include more detailed design, and the modules at the bottom level include the maximum detail of design.

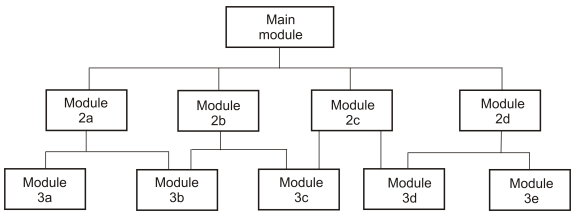

Figure 1.3: Modular design.

The *abstraction* principle is applied in this technique, the top level module includes the design at the highest level of abstraction, and is the easiest to understand and describe because it includes no details of the design. The lowest level modules contain the all the necessary detail, at the lowest level of abstraction. Applying top-down design results in a hierarchical solution to a problem and includes multiple levels of abstraction. Figure 1.3 shows the module hierarchy used in top-down design.

1.6 A Simple Problem: Temperature Conversion

The process of developing a computational model is illustrated in this section with an extremely simple problem: the temperature conversion problem. A basic sequence of steps is discussed for solving this problem and for developing a computational model.

1.6.1 Initial Problem Statement

American tourists visiting Europe do not usually understand the units of temperature used in weather reports. The problem is to devise some mechanism for indicating the temperature in Fahrenheit from a known temperature in Celsius.

1.6.2 Analysis and Conceptual Model

A brief analysis of the problem involves:

1. Understanding the problem. The main goal of the problem is to develop a temperature conversion facility from Celsius to Fahrenheit.
2. Finding the mathematical representation or formulas for the conversion of temperature from Celsius to Fahrenheit. Without this knowledge, we cannot derive a solution to this problem.

The conversion formula is the mathematical model of the problem.

3. Knowledge of how to implement the mathematical model in a computer. We need to express the model in a particular computer tool or a programming language. The computer implementation must closely represent the model in order for it to be correct and useful.
4. Knowledge of how to test the program for correctness.

1.6.3 Design

The mathematical representation of the solution to the problem are the formulas expressing a temperature measurement F in Fahrenheit in terms of the temperature measurement C in Celsius, which is:

$$F = \frac{9}{5}C + 32 \tag{1.1}$$

In this formula, C is a variable that represents the given temperature in degrees Celsius, and F is a derived variable, whose value depends on C.

A formal definition of a function is beyond the scope of this chapter. Informally, a *function* is a computation on elements in a set called the *domain* of the function, producing results that constitute a set called the *range* of the function. The elements in the domain are sometimes known as the input parameters. The elements in the range are called the output results.

Basically, a function defines a relationship between two (or more variables), x and y. This relation is expressed as $y = f(x)$, so y is a function of x. Normally, for every value of x, there is a corresponding value of y. Variable x is the independent variable and y is the dependent variable.

The mathematical model is represented by the mathematical expression for the conversion of a temperature measurement in Celsius to the corresponding value in Fahrenheit. The mathematical formula expressing the conversion assigns a value to the desired temperature in the variable F, the dependent variable. The values of the variable C can change arbitrarily because it is the independent variable.

1.7 Area and Circumference of a Circle

In this section, another simple problem is formulated: the mathematical model(s) and the algorithm. This problem requires computing the area and circumference of a circle, given its radius. The high-level algorithm description in informal pseudo-code notation is:

1. Read the value of the radius of a circle, from the input device.
2. Compute the area of the circle.
3. Compute the circumference of the circle.
4. Print or display the value of the area of the circle to the output device.
5. Print or display the value of the circumference of the circle to the output device.

For the design, the mathematical formulation is:

$$cir = 2\pi r$$
$$area = \pi r^2$$

A more detailed algorithm description follows:

1. Read the value of the radius r of a circle, from the input device.
2. Define the constant π with value 3.14159.
3. Compute the area of the circle.
4. Compute the circumference of the circle.
5. Print or display the value of *area* of the circle to the output device.
6. Print or display the value of *cir* of the circle to the output device.

The previous algorithm now can be implemented by a program that calculates the circumference and the area of a circle.

1.8 Programming Languages

A programming language is used by programmers to write programs that indicate to the computer what computations to perform on the given data. This language contains a defined set of syntax and semantic rules. The syntax rules describe how to write well-defined statements. The semantic rules describe the meaning of the statements.

The design of a solution to a problem is implemented in a program written with an appropriate programming language. This program is known as the *source program* and is written in a high-level programming language.

Once a source program is written, it is translated or converted into an equivalent program in *machine code*, which is the only programming language that the computer can process. The computer can only execute instructions that are in machine code. The program executing in the computer usually reads input data from the input device, performs some computations, and writes results to the output device(s).

1.8.1 High-Level Programming Languages

A high-level programming language is a formal notation that is used to write *instructions* to the computer in the form of a program. A programming language helps programmers in the writing of programs for a large family of problems.

High-level programming languages are hardware independent and are problem-oriented (for a given family of problems). These languages allow more readable programs, and are easy to write and maintain. Examples of conventional programming languages are: C, Fortran, Algol, Ada, Pascal, Smalltalk, C++, Eiffel, and Java.

Programming languages can require considerable effort to learn and master. Several newer and experimental programming languages have been developed that can facilitate the learning and use of programming principles. One such high-level programming language is SPSCL.

There are several integrated development environments (IDE) that are used in program development. Examples of these are: Eclipse, Netbeans, CodeBlocks, and Codelite. Other more complete computing environments are designed for numerical and scientific problem solving that have their own programming language. Some of these computational tools are: MATLAB, Octave, Mathematica, Scilab, Stella, and Maple.

1.8.2 Interpreters

An interpreter is a special program that performs syntax checking of a command in a user program written in a high-level programming language and immediately executes the command. It repeats this processing for every command in the program. Examples of interpreters are the ones used for the following languages: MATLAB, Octave, Python, PHP and PERL.

1.8.3 Compilers

A compiler is a special program that translates a source program written in a high-level programming language into an equivalent program in binary or machine code, which is the only language that the computer can process.

In addition to *compilation*, an additional step known as *linking* is required before a program can be executed. Examples of programming languages that require compilation and linking are: SPSCL, C, C++, Eiffel, Ada, and Fortran. Other programming languages such as Java, require compilation and interpretation.

1.8.4 Compiling and Executing C Programs

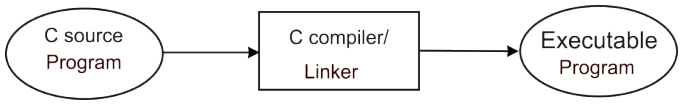

Figure 1.4: Compiling a C program.

Programs written in the C programming language are compiled, linked, and loaded into memory before executing. An executable program file is produced as a result of linking. The libraries are a collection of additional code modules needed by the program. Figure 1.4 illustrates the compilation of a C program. Figure 1.5 illustrate the linkage of the program. The executable program is the final form of the program that is produced. Before a program starts to execute in the computer, it must be loaded into the memory of the computer.

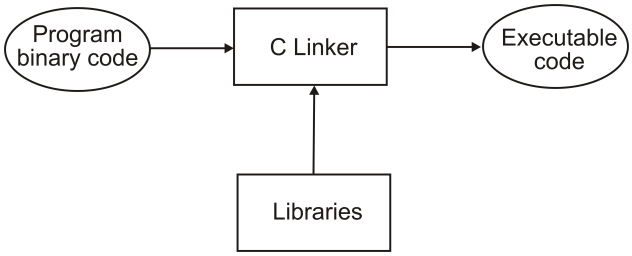

Figure 1.5: Linking a C program.

1.9 The SPSCL Language

The syntax of the C language is too low level, and it is relatively easy to make mistakes in coding an scientific application. These are mainly involved in memory-management issues and the like. Programming large applications in C requires very much detail knowledge of the language.

A higher level of abstraction in design and programming improve significantly the effectiveness of software development. The SPSCL language supports the conceptual frame-work of the scientific style of computation and can facilitate the teaching, learning, and application of modeling and developing computational models of large and complex systems.

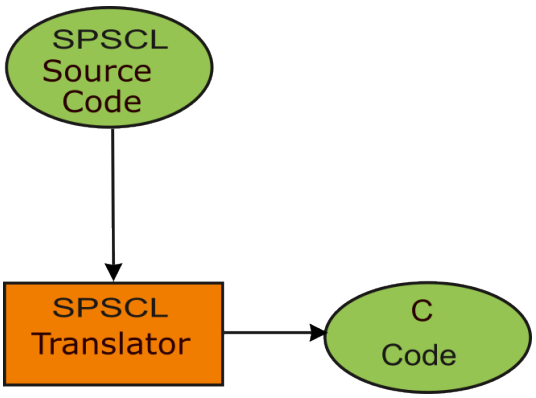

Figure 1.6: Compiling/translating an SPSCL program.

The language syntax is defined to be at a higher level of abstraction than C and C++ (and Java) and it is an enhancement to the widely-used informal pseudo-code syntax used in program and algorithm design. It includes language statements for improving program readability, debugging, maintenance, and correctness.

The SPSCL compiler is a language translator that converts (translates) an SPSCL program into a C program. Figure 1.6 illustrates the compilation of an SPSCL program. The generated C code can be integrated conveniently with any C library.

1.9.1 Compiling and Executing C/C++ Programs

The C/C++ compiler and linker take as input a source program in C or C++ and generates an executable program. The following figure illustrates the compilation/linkage of a C source program.

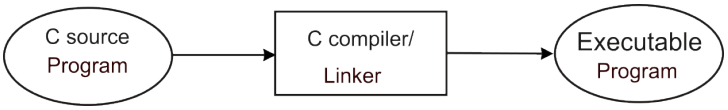

Figure 1.7: Compiling/linking a C program.

1.9.2 Compiling and Execution of Java Programs

To compile and execute programs written in the Java programming language, two special programs are required, the compiler and the interpreter. The Java compiler checks for syntax errors in the

source program and translates it into *bytecode*, which is the program in an intermediate form. The Java bytecode is not dependent on any particular platform or computer system. This makes the bytecode very portable from one machine to another. To execute this bytecode, the Java Virtual Machine (JVM), which carries out the interpretation of the bytecode.

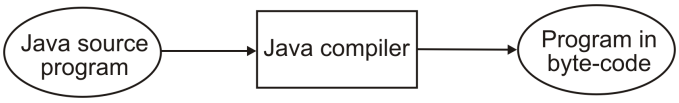

Figure 1.8: Compiling a Java source program.

Figure 1.8 shows what is involved in compilation of a source program in Java. The Java compiler checks for syntax errors in the source program and then translates it into a program in byte-code, which is the program in an intermediate form. Figure 1.9 shows the Java virtual machine (JVM) that executes a program in byte-code.

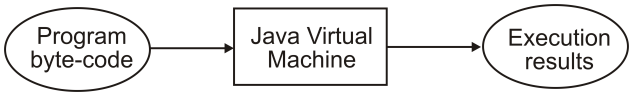

Figure 1.9: Executing a Java program.

1.10 Object-Oriented Programming Languages

Simula was the first object-oriented language developed in the mid-sixties and used to write simulation models. The original language was an extension of Algol. In a similar manner, C++ was developed as an extension to C in the early eighties.

Java was developed by Sun Microsystems in the early nineties and has several very significant features as an object-oriented programming language, unfortunately, it has a syntax similar to C++ and the programs are not very efficient compared to other OO languages.

Languages like C++ and Java can require considerable effort to learn and master. Several newer and experimental, higher-level, object-oriented programming languages have been developed. Each one has a particular goal. One such language is *OOSCL* (Object Oriented System Computing Language); its main purpose is to help the implement large programs, make it easier to learn object-oriented programming principles, and help students transition to Java and/or to C++.

Programming languages like C++ in addition to *compilation*, another step known as *linking* is required before a program can be executed. Programming languages like Java, require compilation and interpretation. This last step is necessary to run the compiled program.

Two special program tools are needed for C++, the C++ compiler and the linker. An executable file is generared. For Java, the two tools required are the Java compiler and the JVM interpreter. The Java compiler checks for syntax errors in the source program and translates it into *bytecode*, which is the program in an intermediate form. The Java bytecode is not dependent on any particular platform or computer system. The Java Virtual Machine (JVM) carries out the interpretation of the bytecode, therefore it runs the bytecode.

1.10.1 Compiling OOSCL Programs

Programs written in OOSCL (an object-oriented language that is higher level than Java and C++), are translated/compiled with the OOSCL compiler. There are two similar translators: OOSCL translates to C++ source code, and JOOSCL that translates to Java source code. The translator that checks for syntax errors in the language, and translates the source program from OOSCL to Java or to C++. These translators/compilers, are available as executable files from the following Web page:

```
http://ksuweb.kennesaw.edu/~jgarrido/scl
```

The OOSCL compilation is illustrated in Figure 1.10. Appendix A explains in further detail how to use the OOSCL translators.

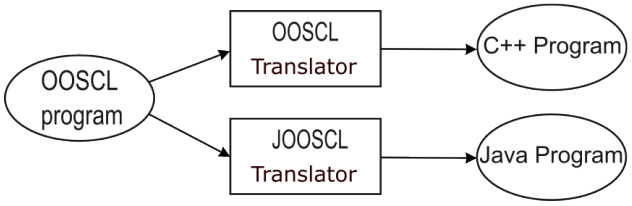

Figure 1.10: OOSCL Translation/Compilation

Summary

Computer problem solving is used to solve large and complex problems in science, engineering, business, and other disciplines. Solution to these problems are implemented by computer programs coded in a particular programming language. There are several programming languages available, such as SPSCL, C, C++, Eiffel, Ada, Java. Compilation is the task of translating a program from its source language to an equivalent program in machine code. Other languages in such as Python, Perl, MATLAB and Octave are interpreted. Computations are carried out on input data by executing individual commands or complete programs. Application programs are programs that the user interacts with to solve particular problems.

Key Terms

computer problem solving	mathematical model	abstraction
algorithm	conceptual model	program development
compilers	linkers	interpreters
programs	commands	instructions
programming language	Java	C
SPSCL	C++	Eiffel
Ada	bytecode	JVM
data definition	Source code	high-level language
program execution	keywords	identifiers

Exercises

Exercise 1.1 Explain the differences between a program and a design.

Exercise 1.2 Explain the reason why the concept of abstraction is important in developing programs.

Exercise 1.3 Investigate and write a short report on the programming languages used to implement programs.

Exercise 1.4 What is a programming language? Why are they needed?

Exercise 1.5 Explain why there are many programming languages.

Exercise 1.6 What are the differences between compilation and interpretation in high-level programming languages?

Exercise 1.7 Explain the purpose of compilation. How many compilers are necessary for a given application? What is the difference between program compilation and program execution? Explain.

Exercise 1.8 What is the real purpose of developing a program? Can we just use a spreadsheet program to solve numerical problems? Explain.

Exercise 1.9 Find and explain the differences in compiling and linking SCL, C, Java, and C++ programs.

Exercise 1.10 For developing small programs, is it still necessary to use a software development process? Explain. What are the main advantages in using a process for program development? What are the disadvantages?

2. Programs

2.1 Introduction

This chapter presents an overview of the structure of a computer program, which include data definitions and basic instructions using the SPSCL and C programming languages. Because functions are the building blocks and the fundamental components of SPSCL and C programs, the concepts of function definitions and function invocations are gradually explained, and complete programs are introduced that illustrate further the role of functions.

2.2 Programs

A *program* consists of data definitions and instructions that manipulate the data. A program is normally written in an appropriate *programming language*. It is considered part of the *software* components in a computer system. The general structure of a program consists of:

- *Data definitions*, which are declarations of all the data to be manipulated by the instructions.
- A *sequence of instructions*, which perform the computations on the data in order to produce the desired results.

2.3 Data Definitions

The data in a program consists of one or more data items. These are manipulated or transformed by the computations (computer operations). Each data definition is specified by declaring a data item with the following:

- The *type* of the data item

- A unique *name* to identify the data item
- An optional initial *value*

The name of a data item is an *identifier* and is defined by the programmer; it must be different from any *keyword* in the programming language. The type of data defines:

- The set of possible values that the data item can take
- The set of possible operations or computations that can be applied to the data item

2.3.1 Names of Data Items

The names of the data items are used in a program for uniquely identifying the data items and are known as *identifiers*. The special text words or symbols that indicate essential parts of a programming language are known as *keywords*. These are reserved words and cannot be used for any other purpose; they cannot be used as identifiers.

For example, the problem for calculating the area of a triangle uses four identifiers, *a, b, c*, and *area*. A valid identifier starts with a letter of the alphabet (upper or lower case) and followed by one or more instances of a letter, a number, or the underscore symbol (_). The data items usually change their values when they are manipulated by the various operations. For example, the following sequence of instructions in SPSCL first assigns the value of 34.5 to variable *y* then adds the value of *x* and the value of *y*; the results are assigned to variable *z*. The SPSCL keyword **set** is used in assignment statements.

```
set y = 34.5
set z = y + x
```

In the C programming language, the two assignment statements are:

```
y = 34.5;
z = y + x;
```

In this example, the data items named *x* and *y* are known as *variables* because their values change when computations are applied on them. Those data items that do not change their values are known as *constants*. For example, *MAX_NUM*, *PI*, etc. These data items are given an initial value that will never change during the program execution.

When a program executes, all the data items used by the various computations are stored in the computer memory, each data item occupying a different memory location.

2.3.2 Data Types

Primitive data types are classified into the three categories:

- Numeric
- Text
- Boolean

The numeric types are further divided into three basic types, *integer*, *float*, and *double*. The non-integer types are also known as fractional, which means that the numerical values have a fractional part.

Values of *integer* type are those that are countable to a finite value, for example, age, number of parts, number of students enrolled in a course, and so on. Values of type *float* and *double* have a decimal point; for example, cost of a part, the height of a tower, current temperature in a boiler, a time interval. These values cannot be expressed as integers. Values of type *double* provide more precision than type *float*.

Text data items are of the basic types: *char* and *string*. Data items of type *string* consist of a sequence of characters, so it is declared as an array of type *char*. The values for these two types of data items are text values. The string type is the most common, such as the text value: 'Welcome!'.

The third data type is used for variables of type *bool* or *boolean*, whose values can take any of two truth-values (*true* or *false*).

2.3.3 Data Declarations in SPSCL

The data declarations are the data definitions in a program and include the name of every variable or constant with its type. The initial values, if any, for the data items are also included in the data declaration.

In the SPSCL programming language, data is declared followed by the keyword **define** and in three different subsections indicated by the corresponding keywords: **constants**, **variables**, and **structures**. The statement for the declaration of a data item has the following basic syntactic structure:

define ⟨ variable_name ⟩ **of type** ⟨ *type* ⟩

The following lines of code in the SPSCL programming language are examples of statements that declare two constants and three variables of type *integer*, *float*, and *boolean*. Only two subsections of declarations are needed: **constants** and **variables**.

```
constants
   define NNP = 5.1416 of type float
   MAX_NUM = 100 of type integer
variables
   define count of type integer
   define weight = 57.85 of type float
   define busy of type boolean
```

2.3.4 Data Declarations in C

The data declarations are the data definitions and include the name of every variable or constant with its type. The initial values, if any, for the data items are also included in the data declaration.

In the C programming language (also in C++ and Java), a statement for the declaration of data items has the following basic syntactic structure:

⟨ *type* ⟩ ⟨ variable_name ⟩;

The following lines of code in the C programming language are examples of statements that declare two constants and three variables of type *int*, *float*, and *bool*.

```
const float  NNP = 5.1416;
const int MAX_NUM = 100;
int count;
float weight = 57.85;
bool busy;
```

2.4 Numerical Types

Numerical values are integers or floating point values. This section briefly describes the number representation of numerical values in a computer system and consequently, and the range and precision of values.

2.4.1 Integer Numbers

In programming languages such as the SPSCL and C, the types for integers are used for whole numbers and are: *integer*, *short integer*, and *long integer*. The type *integer* is the normal type for integers and the values of this type each are stored in a 4-byte (32-bit) memory block on most computer systems. This allows for integer values in the range of $-2,147,488,647$ to $2,147,483,647$.

Values of type *short integer* take less memory storage, only 2 bytes (16 bits) on most computer systems and allows for a range of values from $-32,768$ to $32,767$. Values of type *long integer* may take 8 bytes or more, depending on the system. This allows for integer values in a much larger range. The type *long long* is used for very long integer values and is 128 bits long. The default mode for integer values is *signed*; the mode *unsigned* can be used for types of integer already listed.

2.4.2 Numbers with Decimal Points

The type for numbers with a decimal point are known as floating point types. In SPSCL and C, the types for floating-point values are: *float*, *double*, and *long double*.

Scientific Notation

Scientific notation is used to display very large and very small floating-point values. For example:

```
set y = 5.77262e+12.0
```

The mathematical equivalent for this value is 5.77262×10^{12} and is usually an approximate value. Scientific notation can also be used in mathematical expressions with assignments. For example:

```
set x = 5.4e8 + y
set y = x * 126.5e10
```

Floating-point Representation

In the scientific notation number representation, there are two parts in a floating point numeric value: the *significand* (also known as the *mantissa*) and the *exponent*. The base of the exponent is 10.

Therefore, the value of the mantissa is scaled by a power of 10. The IEEE standard for floating-point numbers specifies how a computer should store these two parts of a floating-point number. This influences not only the *range* of values of the type used, but also the *precision*.

In a 32-bit floating-point representation, the mantissa is stored in the 24-bit slot and the exponent in the remaining 8-bit slot of the complete 32-bit block allocated to the floating-point value. Each of the two slots includes a bit for the sign. This gives a precision 7 digits, and a range that depends on the value of exponent from -38 to 38. The type *float* is implemented this way or in a very similar internal representation.

The type *double* is stored in an 8-byte (64 bits) block of memory. The significand part of the floating-point value is stored in a 53-bit slot, and the rest (11-bit slot) for the exponent value. This gives a precision of about 16 decimal digits and a much wider range of values.

Newer computer systems include type *long double*, which is stored in a 10-byte (80 bits) block of memory. The significand part of the floating-point value is stored in a 65-bit slot, and the rest (15 bits) for the exponent value. This gives a precision of about 19 decimal digits and a much wider range of values.

2.5 Program Modules

In software development, an important task is to convert solution design into a program using an appropriate programming language. In the program, the solution to each subproblem is implemented as a *program module*. A complete program is typically divided into one or program modules. In the SPSCL and C programming languages, three categories of modules are used:

1. Libraries
2. Program files
3. Functions

Functions are the fundamental modules in an SPSCL and C program. A typical program consists of one or more functions. The simplest program includes only a single function, *main*. A very large program has several functions, is partitioned into several program files, and use libraries.

Program files include functions and are part of the main program or module. Libraries are sets of pre-developed and related functions available to use by a program. Most often, libraries are external modules that are reused by many programs. One example of a library is the *math* library that includes most of the common functions that compute mathematical operations.

2.6 Structure of an SPSCL Program

An SPSCL program follows a similar structure of a C program, however more discipline is imposed as the SPSCL program consists of a sequence of *sections*. Figure 2.1 illustrates the general structure of an SPSCL program. The sections are:

1. Imports and Symbols, which may contain data and/or function declarations.
2. Specifications, which include structure specifications.
3. Forward Declarations, which include function declarations that are called before fully defined.

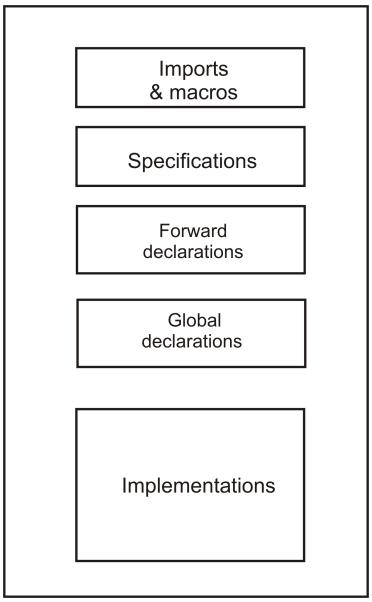

Figure 2.1: General structure of an SPSCL program.

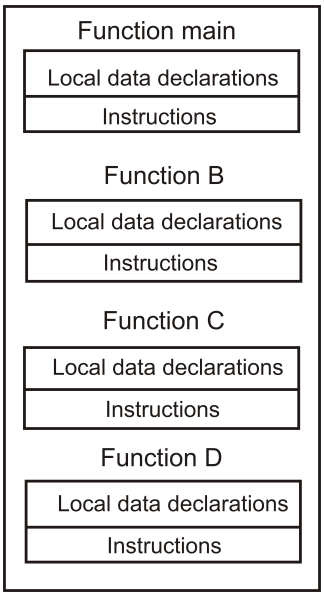

Figure 2.2: The *implementations* section of an SPSCL program.

4. Global Declarations, which are data definitions that can be used in all functions of the program.
5. Implementations, which are the complete functions of the program. This includes function *main*, the only required function in the program.

The sections of a PSPSCL program are all optional, however at least one section must appear in a program. The order of the sections in a program is important. The general structure of the implementations part of an SPSCL program is shown in figure 2.2.

A function can be called or invoked if it has been defined. A function starts executing when it is called by another function. Before a function can be called in a program, a function declaration is required. This declaration is also known as a function *prototype*. This is one of the reasons to have forward and global declarations before the *implementations* section that completely defines all functions of the program.

2.7 Structure of a C Program

A typical program in the C language is structured as shown in Figure 2.3. It consists of several parts in the following sequence:

1. Compiler directives, which indicate to the compiler to append header files to the program that contain data and/or function declarations also known as function *prototypes*, and additional directives. A compiler directive starts with the pound symbol (#). A function prototype is only the header of a function and consists of the type of the function, the function name, and any parameters listed.

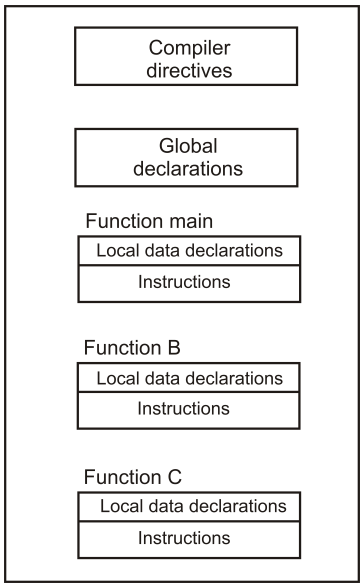

Figure 2.3: Structure of a C program.

2. Global declarations, which may consist of data declarations and/or function declarations. These are global because they can be used by all functions in a program.
3. Definition of function `main`. This is the only required function in a C program; all other functions may be absent. In other words, a C program contains at lease function `main`. Execution of the program starts in this function, and normally ends in this function.
4. Definition of additional functions in the program.

Function definitions include programmer-defined functions that are invoked or called in the program. The other functions that can be called are the built-in functions provided by standard C *libraries* or by external libraries. In C, a library is a collection of related function definitions that may also include data declarations.

Once a function is defined, it can be called or invoked by any other function in the program. A function starts executing when it is called by another function. In a simple C program, a few functions are called in function `main`.

Before a function can be called in a C program, a function declaration is required. This declaration is also known as a function *prototype*. This is one of the reasons to have compiler directives and global declarations before the definition of function `main` and the other function definitions.

2.8 Language Statements

An instruction performs a specific computation or manipulation on the data. In the source program, a computation is written in one or more appropriate language statements. The *assignment statement* is the most fundamental statement (high-level instruction); its general form in SPSCL is:

set ⟨ *variable_name* ⟩ = ⟨ expression ⟩

The assignment statement starts with the keyword **set** and the assignment *operator* is denoted by the = symbol. On the left side of this operator a variable name must always be written. On the right side of the assignment operator, an expression is written.

In the following SPSCL source code example, the first line is a simple assignment statement that will move the value 34.5 into variable x. The second line is a slightly more complex assignment that will perform an addition of the value of variable x and the constant 11.38. The result of the addition is assigned to variable y.

```
set x = 34.5          // assign 34.5 to variable x
set y = x + 11.38
```

The equivalent statements in the C programming language are:

```
x = 34.5;           /* assign 34.5 to variable x */
y = x + 11.38;
```

In the previous example, the second line of source code is an assignment statement and the expression on the right side of the assignment operator is $x + 11.38$. The expression on the right side of an assignment statement may be a simple arithmetic expression or a much more complex expression.

2.9 Simple Functions

As mentioned previously, a typical program program consists of several functions. A function carries out a specific task in a program and is an internal decomposition unit.

Data declared within a function is known only to that function—the scope of the data is *local* to the function. The local data in a function has a limited lifetime; it only exists during execution of the function.

2.9.1 Function Definitions

SPSCL and C programs normally consist of (or *decomposed* of one or more program files, and some of these are header files with data definitions and/or function prototypes, other files include complete function definitions.

Figure 2.4 illustrates the general form of a function in the SPSCL and C languages. The general form of a function definition in the SPSCL programming language is as follows:

```
description
    .  .  .
  */
function ⟨ funct_name ⟩ return type ⟨ type ⟩
   parameters ⟨ param_list ⟩ is
     .  .  .  local declarations
   begin
     .  .  .  statements
endfun ⟨ function_name ⟩
```

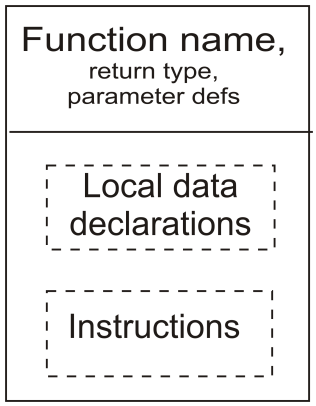

Figure 2.4: Structure of a function.

The relevant documentation of the function definition is included after the keyword *description* and can have one or more lines of comments, which end with a star-slash (*/). The general form of a function definition in the C programming language is:

```
/*
    .  .  .  comments
  */
⟨ return type ⟩ ⟨ funct_name ⟩ ⟨ param_list ⟩ {
    .  .  .  local declarations
    .  .  .  statements
}
```

The relevant documentation of the function definition in C is described in one or more lines of comments, which begins with the characters slash-star (/*) and ends with a star-slash (*/).

The declarations that define local data in the function are optional. The instructions implement the body of the function. The following SPSCL source code shows an example of a simple function for displaying a text message on the screen.

```
description
   This function displays a message
   on the screen. */
function show_message is
begin
     display "Computing data"
endfun show_message
```

The same function in the C programming language is:

```
/*
   This function displays a message
   on the screen. */
void show_message () {
```

```
    printf("Computing data");
}
```

This is a very simple function and its only purpose is to display a text message on the screen. This function does not declare parameters and the type of this function is *void* to indicate that this function does not return a value.

2.9.2 Function Calls

The name of the function is used when calling or invoking the function by some other function. The function that calls another function is known as the calling function; the second function is known as the called function. When a function calls or invokes another function, the flow of control is altered and the second function starts execution immediately.

When the called function completes execution, the flow of control is transferred back (returned) to the calling function and it continues execution from the point after it called the second function. Figure 2.5 illustrates function *A* calling function *B*. After completing its execution, function B returns the flow of control to function A.

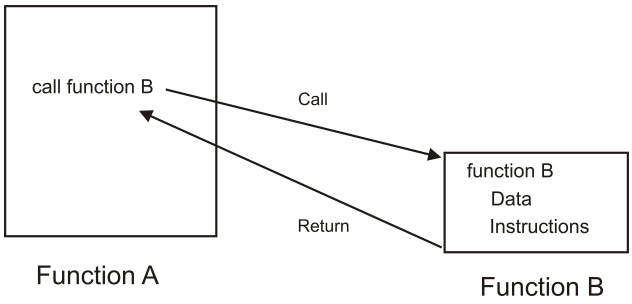

Figure 2.5: A function calling another function

In SPSCL , the statement that calls a simple function uses the keyword **call** followed by the function name. For example, the call to function *show_message* is written as:

```
call show_message
```

In C, the statement that calls a simple function uses the the function name and an empty parentheses pair. The call to function *show_message* is written as:

```
show_message();
```

After the function completes, and depending on the function type, the called function may or may not return a value to the calling function. In the previous example, function *show_message* returns without a value.

2.10 A Simple Program

Listing 2.1a shows a very simple but complete SPSCL program. Lines 1–6 include comments that help in documenting the program. Line 7 is an import command that indicates the inclusion of

the header file `scl.h`. This file contains the function prototypes for the system functions (built-in) needed for input output (I/O) of data. Line 9 starts the definition of function `main`, it indicates name of the function being defined. This simple program defines only a single function: `main`. Line 11 is a local data declaration; it declares variable *x* of type `double`. Line number 13 includes a `display` statement that prints on the screen a string literal. Line 14 has an assignment statement. Line 15 prints a string literal and the value of variable *x*. Line 16 includes a terminate statement using the `exit` keyword.

Listing 2.1a: A simple program in SPSCL .

```
 1 /*
 2  Program     : welcome.scl
 3  Author      : J M Garrido, November 20, 2021.
 4  Description : Display welcome message on the screen
 5     and the value of variable x.
 6  */
 7 import "scl.h"
 8 implementations
 9 function  main is
10 variables
11     define x of type double   // a variable declaration
12 begin
13     display "Welcome to the world of SPSCL "
14     set x = 45.95             // assigns a value to variable x
15     display "Value of x: ", x
16     exit                    // execution terminates OK
17 endfun main
```

2.10.1 Translating an SPSCL Program and Compiling the C Code

Using Codeblocks

Using an IDE such as Codeblocks makes it very easy to edit, translate, compile the C program, link, and execute a program. To translate the SPSCL program, activate the Build menu on the top bar and select the Build option. This basically translates the SPSCL program to C, compiles the C program, links, and creates an executable file of the program. See Appendix A for detailed information on setting up and using Codeblocks for implementing SPSCL (and C) programs.

Using the Command Line Interface

To translate an SPSCL program `welcome.scl` shown in Listing 2.1a on a command line interface, use the SPSCL translator executable file `spscl.exe` on Windows or `spscl.out` on Linux.

The next few command lines translate the file `welcome.scl` in a command window, also known as the command prompt, or the command line interface on Windows. [1]

```
C:\comp_progs\scl_progs> C:\SPSCL\spscl welcome.scl
SPSCL  v 1.1 File: welcome.scl Sun Jan 26 08:43:49 2022
File: welcome.scl no syntax errors, lines processed: 17
```

[1] On Windows, click the Start button located on the bottom left part of the desktop and type `cmd` on the small dialog box that appears. More details are included in Appendix A.

Assume that the source files (SPSCL programs) are stored in folder `comp_progs/scl_progs` and that the SPSCL translator is stored in folder `SPSCL` On Linux, open a *terminal* window and type the commands.

```
$ cd comp_progs/scl_progs
$ ~/SPSCL /scl welcome.scl
SPSCL  v 1.1 File: welcome.scl Wed Jan 26 08:43:49 2022
File: welcome.scl no syntax errors, lines processed: 17
```

The C program produced `welcome.c` by the SPSCL translator is shown in the following listing.

```
 1 // SPSCL  v 1.0 File: welcome.c, Wed Jan 26 09:33:50 2022
 2 /*
 3  Program     : welcome.scl
 4  Author      : J M Garrido, November 20, 2021.
 5  Description : Display welcome message on the screen
 6     and the value of variable x.
 7  */
 8 #include "scl.h"
 9 int main() {
10   double  x;   // a variable declaration
11   printf("Welcome to the world of SPSCL  \n");
12   x = 45.95;  // assigns a value to variable x
13   printf("Value of x:   %lf\n",x);
14   return 0;   // execution terminates OK
15 }  // end main
```

Listing 2.1b shows the same program written manually using the C programming language. Lines 1–6 include comments that help in documenting the program. Line 7 is a compiler directive that indicates the inclusion of the header file `stdio.h`. This contains the function prototypes for the system functions (built-in) needed for input output (I/O) of data. Line 9 starts the definition of function `main`, it indicates the type and name of the function being defined. This simple program defines only a single function: `main`. Line 10 is a local data declaration; it declares variable *x* of type *double*.

Line 11 calls the system function `printf` to display the text string within quotes, on the screen. Line 12 is a simple assignment statement; it assigns value 45.95 to variable *x*. Line 13 invokes system function `printf` to display a text string and the value of variable *x*. The escape character `\n` is a line feed control character for output; it indicates a change to a new line. Line 14 is a termination statement by returning a value zero to the operating system.

Listing 2.1b: A simple program in C.

```
1 /*
2  Program     : welcome.c
3  Author      : J M Garrido, Nov 2021
4  Description : Display welcome on the screen
5     and the value of a variable.
6  */
7 #include <stdio.h>
```

```
 8
 9 int main() {
10     double x;     // variable declaration
11     printf("Welcome to the world of C\n");
12     x = 45.95;    // assigns a value to variable x
13     printf("Value of x: %lf \n", x);
14     return 0;     // execution terminates OK
15 }
```

2.10.2 Compiling and Executing the C Program

Using Codeblocks

After building the SPSCL program as a project, the executable of the program has been created in Codeblocks. To execute the program activate the Build menu on the top bar, then select Run. The program starts executing in a new window, as shown in Figure 2.6. See Appendix A for detailed information on setting up Codeblocks for implementing SPSCL programs.

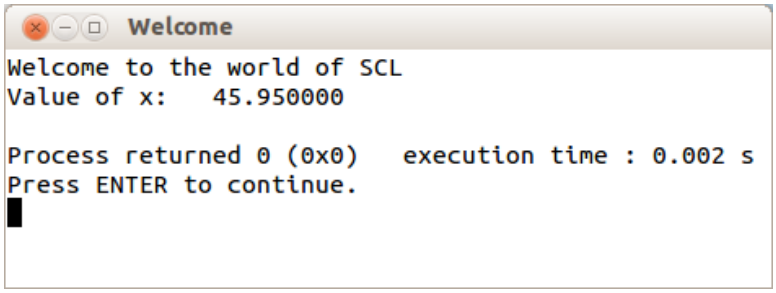

Figure 2.6: Executing the program.

Using the Command Line Interface

The use of the GNU C compiler tool (*gcc*) to compile and link a C program (`welcome.c`) on a Linux Terminal is exactly the same as using this compiler on the Windows command prompt. Note that by default, the executable file is named `a.out`. On Windows, the default name of the executable file is `a.exe`. When the executable program is run, it produces the results shown; note that there are two lines printed by the program.

A recommended way to use the *gcc* compiler tool with a more appropriate and name of the executable program file is:

```
gcc -Wall -o welcome.out welcome.c
```

The compiler flag `-Wall` should always be included so the compiler will display any warning messages, in addition to the error messages. The flag `-o` is used to indicate the name `welcome.out` of the executable file generated by the compiler.

Summary

The structure of an SPSCL computer program and similarly a C program include data definitions and basic instructions that manipulate the data. Functions are the building blocks and the funda-

mental components of these programs. Functions are defined and called in the various parts of a program. These basic programming constructs are used in developing programs by implementing the corresponding design.

Key Terms

programs	functions	function invocation
function call	assignment statement	assignment operator
executable file	console	local declaration
terminal	main	variables
constants	compiler directive	global declaration
function prototype	header file	function definition

Exercises

Exercise 2.1 Why is a function a decomposition unit? Explain.

Exercise 2.2 What is the purpose of function *main*?

Exercise 2.3 Explain the various categories of function definitions.

Exercise 2.4 Develop a program that computes the area of a right triangle given values of the altitude and the base.

Exercise 2.5 Develop a program that computes the temperature in Celsius, given the values of the temperature in Fahrenheit.

Exercise 2.6 Develop a program that computes the circumference and area of a square, given the values of its sides.

Exercise 2.7 Develop a program that computes the slope of a line between two points in a plane: P_1 with coordinates (x_1, y_1), and P_2 with coordinates (x_2, y_2). Use the coordinate values: $(0, -3/2)$ and $(2, 0)$.

3. Assignments and Basic Input/Output

3.1 Introduction

An expression combines one or more operations and is evaluated to produce a single value.This chapter presents the notion of mathematical expressions in assignments and basic input and output statements using the SPSCL and C programming languages. Complete program are introduced that illustrate the use of these statements.

3.2 Assignment and Mathematical Expressions

Expressions may include constants, variables, and functions. As explained previously, the *assignment* statement is used to assign a value to a variable. This variable must be written on the left-hand side of the assignment operator. When this statement executes, the value that results after evaluating an expression is assigned to the variable o the left-hand side of the assignment operator.

The assignment operator is written with the '=' symbol, which is the standard symbol for the equal sign in arithmetic and algebra.

In the following example, the first statement assigns the constant value 21.5 to variable y. The second statement assigns the value 4.5 to variable x. In the third assignment statement, the expression $y + 1.5 \times x^3$ is evaluated and the resulting value is assigned to variable z. In SPSCL, these assignment statements are written as follows:

```
set y = 21.5
set x = 4.5
set z = y + 1.5 x^3
```

3.2.1 Simple Numeric Computations Using SPSCL

Because an assignment statement causes an expression to be evaluated and the result assigned to a variable, the assignment statement changes the value of the variable on the left-hand side of the assignment ('=') operator. For example, add 15 to the value of variable x, subtract the value of variable y from variable z. In SPSCL, the *add* and the *subtract* operations are written in simple arithmetic expressions in an assignment statement with "+" and "−" operator symbols. The two statements are written in SPSCL as follows:

```
set x = x + 15
set z = z - y
```

In the first statement, the new value of x is computed by adding 15 to the previous value of x. In the second assignment, the new value of variable z is assigned the value that results after subtracting the value of y from the previous value of z.

Integer operations include addition, subtraction, multiplication, and division. Integer division might not produce an integer result. This operation will normally produce a quotient and a remainder. For example 7 divided by 2 results in 3 and a remainder of 1. In SPSCL, the **mod** operation produces the remainder when the first operand is greater than zero. For example, the expression `7 mod 2` produces 1. For example, a more general expression with the **mod** operator is:

```
set yy = x mod 3
```

To add or subtract 1 in SPSCL, the **increment** and the **decrement** statements can be used. For example, the first of the following statements adds the constant 1 to the value of variable j. In a similar manner, the second statement statement subtracts the constant value 1 from the value of variable k.

```
increment j
decrement k
```

Priority of Integer Operations

The order in which the operations are performed in an integer expression is also known as the *precedence rule*. The order of these operations is:

1. The expression within a parenthesis pair are evaluated.
2. The operations multiplication, **mod**, and division are performed from left to right.
3. The operations of addition and subtraction are performed from left to right.

3.2.2 More Advanced Expressions in SPSCL

In the previous examples, simple arithmetic expressions were used in the assignment statements. These are addition, subtraction, multiplication, division, and exponentiation. More complex calculations use various numerical functions, such as square root and trigonometric functions. For example, the value of the expression $\cos p + q$ assigned to variable y and the value of $\sqrt{x-y}$ assigned to variable q. These expressions apply the mathematical functions *cos* and *sqrt* and the corresponding statements in SPSCL are:

```
set y = cos(p) + q
set q = sqrt(x - y)
```

In the following example, the value of the mathematical expression $x^n \times y \times sin^{2m} x$ is assigned to variable z and the statement in SPSCL is:

```
set z= x^n * y * sin(x)^(2*m)
```

3.3 Simple Input/Output

Input and output statements are used to read (input) data values from the input device (e.g., the keyboard) and write (output) data values to an output device (the computer screen).

Output Statement

In SPSCL, the *display* statement that can be used for the output of a list of variables and literals; it is written with the keyword **display**. This output statement writes the value of one or more variables to the output device. The variables do not change their values. The general form of the output statement in SPSCL is:

display ⟨ *data_list* ⟩

For example, the following output displays the string literal "value of x= ", followed by the value of variable x, then the string literal "value of y= ", followed by the value of variable y.

```
display "value of x= ", x, "value of y = ", y
```

Input Statements

The *input* statement reads a value of a variable from the input device (e.g., the keyboard). This statement is written with the keyword **input**, for input of a single data value and assign to a variable. The following two lines of pseudo-code include the general form of the input statement and an example that uses the *read* statement to read a value of variable y.

input ⟨ string_lit⟩ ⟨ *var_name* ⟩

For example, the following input statement reads the value of variable y after it displays the string literal "Enter value of y: ".

```
input "Enter value of y: ", y
```

3.4 Assignment and Numeric Computations Using C

As explained previously, the *assignment* statement is used to assign a value to a variable. This variable must be written on the left-hand side of the assignment operator. The value assigned to the variable is the value that results after evaluating the expression.

Simple Numerical Computations

In the following example, the first statement assigns the constant value 21.5 to variable y. The second statement assigns the value 4.5 to variable x. In the the third assignment statement in the example, variable z is assigned the result of evaluating the expression, $y + 1.5 \times x^3$. In C, the statements are:

```
y = 21.5;
x = 4.5;
z = y + 10.5 * pow(x, 3);
```

The assignment statement changes the value of the variable on the left-hand side of the assignment operator. For example, add 15 to the value of variable x, subtract the value of variable y from variable z. In C these are implemented as:

```
x = x + 15;
z = z - y;
```

To add 1 to a variable applies the increment operator. To subtract the constant value 1 from the value of of a variable applies the decrement operator. For example, to add the constant 1 to the value of variable j and to decrement the value of variable k in C, these are written as follows:

```
j++;
k--;
```

More Advanced Expressions in C

More complex calculations use various numerical functions, such as square root and trigonometric functions. For example, the value of the expression $\cos p + q$ assigned to variable y and the value of $\sqrt{x-y}$ assigned to variable q. These expressions apply the mathematical functions *cos* and *sqrt* and the corresponding statements in C are:

```
y = cos(p) + q;
q = sqrt(x - y);
```

In the following example, the value of the mathematical expression $x^n \times y \times sin^{2m} x$ is assigned to variable z and the statement in C is:

```
z = pow(x, n) * y * pow(sin(x),(2*m));
```

3.5 Basic Input/Output Using C

Input and output statements are used to read (input) data values from the input device (e.g., the keyboard) and write (output) data values to an output device (the computer screen).

Output in C

The C programming languages provide several functions for console input/output. Function *printf* is used to display or print string literals, and for every variable, the format specifier used to print the value of a variable, and value of the variable. The general form for this function call in C is:

```
printf (" [string], [format specifier]", [variable list] );
```

The following output displays the string literal "value of x= ", followed by the value of variable *x*, then the string literal "value of y= ", followed by the value of variable *y*. The statement in C is:.

```
printf ("values of x= %lf value of y= %lf\n", x, y);
```

Format Specifiers

There are several format specifiers defined in C. The general notation for specifiers is `%fs` that indicates that a format specifier immediate follows the % symbol. The The following table provides a list of the most common format specifiers.

Format specifier	Data type	Description
%c	char	single character
%d (%i)	int	signed integer
%e (%E)	float or double	exponential format
%LE	long double	exponential format
%f	float or double	signed decimal
%lf	double	signed decimal
%Lf	long double	signed decimal
%g (%G)	float or double	general decimal
%o	int	unsigned octal value
%p	pointer type	address stored in pointer
%s	array of char	sequence of characters
%u	int unsigned	unsigned decimal
%x (%X)	int unsigned	hex value

To display variable *var* of type double and variable *j* of type integer, is performed by calling function *printf* in the following manner:

```
printf("v = %lf  j = %d", var, j);
```

Finer control of the output value can be achieved by indicating the width of the output field (number of character spaces), *w*, and the number of digits to write after the decimal point, *d*. For example, to write the value of a variable of type double with a specifier that will fill the output with a field size of 10 including 2 decimal digits, the specifier is: `10.2%lf`. If the total number of digits in the value is less than the indicated field size, then it is padded with spaces on the left of the value. This format specifier in a call to function *printf* is:

```
printf( "Variable: %10.2lf", alpha);
```

Control Characters in Output

A control characters is usually necessary in an output instruction to move the cursor to the next line after printing the values indicated. The control character

`n` represents a newline character and is known as an escape sequence. The most commonly used escape sequences are:

`\n` (newline)
`\t` (tab)
`\v` (vertical tab)
`\f` (new page)
`\b` (backspace)
`\r` (carriage return)

Input in C

The input statement implies an assignment statement for the variable y, because the value that is read from the input device is assigned to the variable specified. Function *scanf* reads the value of one or more variables from the input device. Function *scanf* is used with a format specifier following '%' operator for every input value. In the following example, the value of a numeric floating-point variable, *var* is read from input with the format `lf` that follows the '%' operator and is is shown in the following line of code. The name of the variable follows the '&' operator.

```
scanf ("%lf", &var);
```

A string message is typically included to prompt the user for input of a data value. This is implemented with the *printf* function. The previous example for reading the value of variable y is written in C as follows:

```
printf ("Enter value of y: ");
scanf ("%lf", &y);
```

3.6 Computing the Distance Between Two Points

3.6.1 Problem statement

The following problem requires computing the distance between two points in a Cartesian plane. A program is to be developed that computes this distance, given the values of the coordinates of the two points.

3.6.2 Analysis of the Problem

A Cartesian plane consists of two directed lines that perpendicularly intersect their respective zero points. The horizontal directed line is called the *x-axis* and the vertical directed line is called the *y-axis*. The point of intersection of the x-axis and the y-axis is known as the *origin* and is denoted by the letter O.

Figure 3.1 shows a Cartesian plane with two points, P_1 and P_2. Point P_1 is defined by two coordinate values (x_1, y_1) and point P_2 is defined by the coordinate values (x_2, y_2).

3.6.3 Design of the Solution

The horizontal distance between the two points, Δx, is computed by the difference $x_2 - x_1$. Similarly, the vertical distance between the two points is denoted by Δy and is computed by the difference

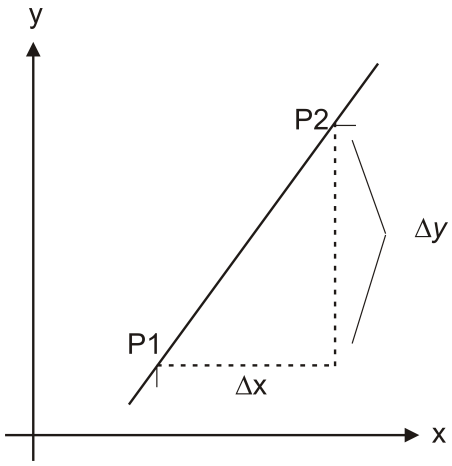

Figure 3.1: Horizontal and vertical distances between two points.

$y_2 - y_1$. The distance, d, between two points P_1 and P_2 in a Cartesian plane is calculated with the following mathematical expression:

$$d = \sqrt{\Delta x^2 + \Delta y^2}$$

A detailed design in an algorithm follows:

1. Read the values of the coordinates for point P_1 from input device (keyboard)
2. Read the values of the coordinates for point P_2 from the input device
3. Compute the horizontal distance, Δx, between the two points:

$$\Delta x = x_2 - x_1$$

4. Compute the vertical distance, Δy, between the two points:

$$\Delta y = y_2 - y_1$$

5. Compute the distance, d, between the two points:

$$d = \sqrt{\Delta x^2 + \Delta y^2}$$

6. Display the value of the distance between the two points, on the output device (video screen)

3.6.4 Implementation

This phase implements the design by coding a program in SPSCL , compiling/linking, and testing the program. Listing 3.1a shows the SPSCL source program. This program is stored in file `distpoints.scl`.

Listing 3.1a: SPSCL program for computing the distance between two points.

```
 1 /*
 2  Program     : distpoints.scl
 3  Author      : Jose M Garrido, January, 20, 2022.
 4  Description : This program computes the distance between two
 5                    points in a Cartesian plane.
 6  */
 7 import "scl.h"
 8 implementations
 9 function main is
10 variables
11     define x1 of type double  // x-coord of P1
12     define y1 of type double  // y-coord of P1
13     define x2 of type double  // x-coord of P2
14     define y2 of type double  // y-coord of P2
15     define dx of type double  // horizontal distance
16     define dy of type double  // vertical distance
17     define d of type double   // resulting distance
18 begin
19     input "Enter value of x-coordinate of P1: ", x1
20     input "Enter value of y-coordinate of P1: ", y1
21     display "Coordinates of P1: ", x1, y1
22     input "Enter value of x-coordinate of P2: ", x2
23     input "Enter value of y-coordinate of P2: ", y2
24     display "Coordinates of P2: ", x2, y2
25     // compute horizontal distance between points
26     set dx = x2 - x1
27     // compute vertical distance between points
28     set dy = y2 - y1
29     display "Horizontal and vertical distances: ", dx, dy
30     // compute the distance between the points
31     set d = sqrt( dx^2 + dy^2 )
32     // display result
33     display "Distance between P1 and P2: ", d
34     exit      // execution terminates ok
35 endfun main
```

Listing 3.1b shows the C source program produced by the SPSCL translator. This program is stored in file `distpoints.c`.

Listing 3.1b: C program for computing the distance between two points.

```
 1 // SPSCL  v 1.0 File: distpoints.c, Wed Jan 22 12:32:47 2022
 2 /*
 3  Program     : distpoints.scl
 4  Author      : Jose M Garrido, January, 20, 2022.
 5  Description : This program computes the distance between two
 6                  points in a Cartesian plane.
 7  */
 8 #include "scl.h"
 9 int main() {
10   double  x1;   // x-coord of P1
11   double  y1;   // y-coord of P1
```

```
 12   double  x2;   // x-coord of P2
 13   double  y2;   // y-coord of P2
 14   double  dx;   // horizontal distance
 15   double  dy;   // vertical distance
 16   double  d;    // resulting distance
 17   printf("Enter value of x-coordinate of P1: ");
 18   scanf(" %lf",  &x1);
 19   printf("Enter value of y-coordinate of P1: ");
 20   scanf(" %lf",  &y1);
 21   printf("Coordinates of P1:   %lf %lf\n",x1, y1);
 22   printf("Enter value of x-coordinate of P2: ");
 23   scanf("   %lf",  &x2);
 24   printf("Enter value of y-coordinate of P2: ");
 25   scanf("   %lf",  &y2);
 26   printf("Coordinates of P2:   %lf %lf\n",x2, y2);
 27   // compute horizontal distance between points
 28   dx = (x2) - (x1);
 29   // compute vertical distance between points
 30   dy = (y2) - (y1);
 31   printf("Horizontal and vertical distances: %lf %lf\n"
             ,dx, dy);
 32   // compute the distance between the points
 33   d = sqrt(( pow(dx, 2) ) + ( pow(dy, 2) ));
 34   // display result
 35   printf("Distance between P1 and P2:   %lf\n",d);
 36   return 0;    // execution terminates ok
 37 }  // end main
```

The following listing shows the SPSCL translation, compiling and linking the C program, and running the executable program created (`dispoints.exe`) with the input values shown. This operational process was performed on Windows. The name of the SPSCL translator is `spscl.exe` and the files are located in folder `C:\SPSCL`.

```
C:\SPSCL >spscl distpoints.scl
SPSCL  v 1.1 File: distpoints.scl Wed Jan 29 13:07:37 2022
File: distpoints.scl no syntax errors, lines processed: 35

C:\SPSCL >gcc distpoints.c -o distpoints.exe -lm

C:\SPSCL >distpoints
Enter value of x-coordinate of P1: 2.25
Enter value of y-coordinate of P1: 1.5
Coordinates of P1:   2.250000 1.500000
Enter value of x-coordinate of P2: 1.3
Enter value of y-coordinate of P2: 0.45
Coordinates of P2:   1.300000 0.450000
Values of horizontal and vertical distances:   -0.950000 -1.050000
Distance between P1 and P2:   1.415980
```

3.7 Computing Area and Circumference

In the following example, a program is developed that computes the area and circumference of a circle. The input value of the radius is read from the keyboard and the results (the area and circumference) are written to the screen.

3.7.1 Specification

The specification of the problem can be described as a high-level description: given the value of the radius of a circle, compute the area and circumference of the circle and show the results.

3.7.2 Design

A detailed description of the design is described in form of an algorithm as follows:

1. Read the value of the radius r of a circle, from the input device.
2. Define the constant π with value 3.14159, or a predefined value.
3. Compute the area of the circle, $area = \pi r^2$.
4. Compute the circumference of the circle $cir = 2\pi r$.
5. Print or display the value of *area* of the circle to the output device.
6. Print or display the value of *cir* of the circle to the output device.

3.7.3 Program Implementation

SPSCL Program

The program is implemented in the SPSCL programming language and is stored in file *areacir.scl*, which is shown in Listing 3.2a.

Listing 3.2a: SPSCL program for computing the area and circumference.

```
1 import "scl.h"
2 /*
3  Program     : areacir.scl
4  Description : Read value of the radius of a circle from input
5     console, compute the area and circumference, display value of
6     of these on the output console.
7  Author      : Jose M Garrido, Nov 20 2021.
8  */
9 symbol PI M_PI
10 implementations
11 function main is
12 variables
13     define r of type double    // radius of circle
14     define area of type double
15     define cir of type double // circumference of circle
16 begin
17     input "Enter value of radius: ", r
18     display "Value of radius: ", r
19     set area = PI * r^2
20     set cir = 2.0 * PI * r
21     display "Value of area: ", area
22     display "Value of circumference: ", cir
```

```
 23     exit      // execution terminates ok
 24 endfun main
```

The following listing shows the SPSCL translation, compiling and linking the C program, and running the executable program created (`areacir.exe`) with the input values shown. This operational process was performed on Windows. The name of the SPSCL translator is `spscl.exe`.

```
C:\SPSCL >spscl areacir.scl
SPSCL  v 1.1 File: areacir.scl Thu Jan 30 12:34:56 2022
File: areacir.scl no syntax errors, lines processed: 24

C:\SPSCL >gcc areacir.c -o areacir.exe -lm

C:\SPSCL >areacir
Enter value of radius: 1.25
Value of radius:   1.250000
Value of area:   4.908739
Value of circumference:   7.853982
C:\SPSCL >
```

C Program

The C program that results from translating the SPSCL program follows and is stored in file *areacir.c*, which is shown in Listing 3.2b.

Listing 3.2b: C program for computing the area and circumference.

```
 1 // SPSCL  v 1.0 File: areacir.c, Thu Jan 30 12:34:56 2022
 2 /*
 3  Program     : areacir.scl
 4  Description : Read value of the radius of a circle from input
 5     console, compute the area and circumference, display value of
 6     of these on the output console.
 7  Author      : Jose M Garrido, Nov 20 2021.
 8  */
 9 #include "scl.h"
10 #define PI M_PI
11 int main() {
12   double  r;    // radius of circle
13   double  area;
14   double  cir;  // circumference of circle
15   printf("Enter value of radius: ");
16   scanf(" %lf",  &r);
17   printf("Value of radius:   %lf\n",r);
18   area = (PI) * ( pow(r, 2) );
19   cir = ((2.0) * (PI)) * (r);
20   printf("Value of area:   %lf\n",area);
21   printf("Value of circumference:   %lf\n",cir);
22   return 0;    // execution terminates ok
23 }  // end main
```

The following listing shows the shell Linux commands that compile and link the file `areacir.c` and execute the file `areacir.out`.

```
$ gcc -Wall areacir.c -o areacir.out
$ ./areacir.out
Enter value of radius: 3.15

Value of radius: 3.150000
Value of area: 31.172453
Value of circumference: 19.792034
```

3.8 Temperature Conversion Problem

This section describes the development of a program that solves the temperature conversion problem. The solution and implementation is derived by following a basic sequence of steps

3.8.1 Specification of the Problem

Given the value of the temperature in degrees Celsius, compute the corresponding value in degrees Fahrenheit and show this result.

3.8.2 Design

The solution design for this problem is based on a mathematical representation, the formula expressing a temperature measurement F in Fahrenheit in terms of the temperature measurement C in Celsius is:

$$F = \frac{9}{5}C + 32$$

The solution to the problem is the mathematical expression for the conversion of a temperature measurement in Celsius to the corresponding value in Fahrenheit. The mathematical formula expressing the conversion assigns a value to the desired temperature in the variable F, the dependent variable. The values of the variable C can change arbitrarily because it is the independent variable. The mathematical model uses real numbers to represent the temperature readings.

3.8.3 Implementing the Program

The program is derived by implementing the mathematical model in a program using the SPSCL programming language. This program is developed using the command line interface of Linux and using the *gedit* editor program. On Windows, the *Notepad++* editor is a good choice. In a similar manner to the previous example, the program is developed by writing a SPSCL program, translating it to a C program, then compiling, linking, and executing the executable program generated by the C compiler/linker.

The SPSCL Program

The SPSCL program in Listing 3.3a shows the complete source program. In a similar manner to the previous examples, these simple programs have one a single function, *main*.

Listing 3.3a: Temperature conversion program in SPSCL .

```
 1 /*
 2  Program     : tempctof.scl
 3  Author      : Jose M Garrido
 4  Description : Read value of the temperature Celsius from input console,
 5                convert to degrees Fahrenheit, and display value of this
 6                new temperature value on the output console
 7  */
 8 import "scl.h"
 9 implementations
10 function main is
11 variables
12     define tempc of type double  // temperature in Celsius
13     define tempf of type double   // temperature in Fahrenheit
14 begin
15     input "Enter value of temp in Celsius: ", tempc
16     display "Value of temp in Celsius: ", tempc
17     set tempf = tempc * (9.0/5.0) + 32.0
18     display "Value of temperature in Fahrenheit: ", tempf
19     exit      // execution terminates ok
20 endfun main
```

The following listing shows the SPSCL translation, compiling and linking the C program, and running the executable program created (`tempctof.exe`) with the input values shown.

```
C:\SPSCL spscl tempctof.scl
SPSCL  v 1.1 File: tempctof.scl Thu Jan 30 15:58:08 2022

File: tempctof.scl no syntax errors, lines processed: 20

C:\SPSCL >gcc tempctof.c -o tempctof.exe -lm
C:\SPSCL >tempctof
Enter value of temp in Celsius: 23.5
Value of temp in Celsius:   23.500000
Value of temperature in Fahrenheit:   74.300000
```

The C Program

The C program in Listing 3.3b shows the complete source program produced by translating the SPSCL program.

Listing 3.3b: Temperature conversion program in C.

```
 1 // SPSCL  v 1.0 File: tempctof.c, Thu Jan 30 15:58:08 2022
 2 /*
 3  Program     : tempctof.scl
 4  Author      : Jose M Garrido
 5  Description : Read value of the temperature Celsius from input console,
 6                convert to degrees Fahrenheit, and display value of this
 7                new temperature value on the output console
 8  */
 9 #include "scl.h"
10 int main() {
```

```
11   double  tempc;   // temperature in Celsius
12   double  tempf;   // temperature in Fahrenheit
13   printf("Enter value of temp in Celsius: ");
14   scanf(" %lf",  &tempc);
15   printf("Value of temp in Celsius:   %lf\n",tempc);
16   tempf = ((tempc) * (( (9.0)/(5.0)) )) + (32.0);
17   printf("Value of temperature in Fahrenheit:   %lf\n",tempf);
18   return 0;     // execution terminates ok
19 }  // end main
```

Compiling and linking the C program generates an executable file using the GNU C compiler/linker. Figure 3.2 shows a terminal window on Linux and the corresponding commands that compiles/links, and then executes the program.

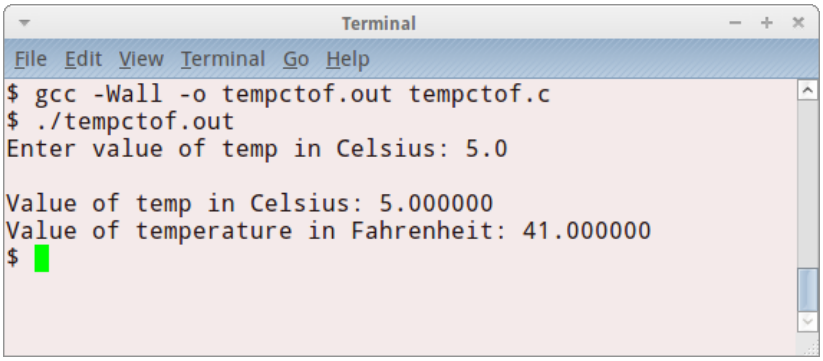

Figure 3.2: Compiling, linking, and executing a program.

Executing (running) the program computes the temperature in Fahrenheit given the value 5.0 for temperature in Celsius. The result is 41 degrees Fahrenheit.

The program can be executed several times to compute the Fahrenheit temperature starting with a given value of 10.0 for the temperature in Celsius and then repeating in increments of 5.0 degrees Celsius. The last computation is for a given value of 45.0 degrees Celsius.

Table 3.1 shows all the values of temperature in Celsius used to compute the corresponding temperature in Fahrenheit. This is a short set of results of the original problem.

Table 3.1: Celsius and Fahrenheit temperatures.

Celsius	5	10	15	20	25	30	35	40	45
Fahrenheit	41	50	59	68	77	86	95	104	113

Summary

The assignment statement is the most fundamental statement in a program. In SPSCL an assignment statement starts with the **set** keyword. It assigns a value to the variable on the left-hand side of the assignment operator, '='. The right-hand side can be any value specified, a simple expression, or a

mathematical expression that is evaluated. An input statement is used to read the value of a variable from an input device, such as the keyboard. An output statement displays the value of one or more variables and strings. A format specifier is used in C programs, one or more control characters or escape sequences may also be used with input and output statements.

Key Terms

assignment statement	expressions	assignment operator
output statement	display	output device
format specifier	control character	escape sequence
input statement	mod	SPSCL translation

Exercises

Exercise 3.1 Develop a program that computes the conversion of values from kilometers to miles.

Exercise 3.2 Develop a program that computes the conversion of values from miles to kilometers.

Exercise 3.3 Develop a program that computes the conversion of values from kilograms to pounds.

Exercise 3.4 Develop a program that computes the conversion of values from pounds to kilograms.

Exercise 3.5 Develop a program that computes the conversion of values from liters to gallons.

Exercise 3.6 Develop a program that computes the conversion of values from gallons to liters.

Exercise 3.7 Develop a program that computes the area of a trapezoid given values of the altitude, base1 and base2.

Exercise 3.8 Develop a program that computes the perimeter of a trapezoid given values of the altitude, base1 and base2.

Exercise 3.9 Develop a program that computes the area of an ellipse, given the values of its semiaxes.

Exercise 3.10 Develop a program that computes the area of the surface of a sphere, given the value of its radius.

Exercise 3.11 Develop a program that computes the volume of a sphere, given the value of its radius.

Exercise 3.12 Develop a program that computes the circumference and area of a square, given the values of its sides.

Exercise 3.13 Develop a program that computes the circumference and area of a sector of a circle, given the values of the radius and the angle that defines the sector .

Exercise 3.14 Develop a program that computes the slope of a line between two points in a plane: P_1 with coordinates (x_1, y_1), and P_2 with coordinates (x_2, y_2). Use the coordinate values: $(0, -3/2)$ and $(2, 0)$.

4. Modules and Functions

4.1 Introduction

A program is usually partitioned into modules. A function is the most fundamental module or decomposition unit in SPSCL and C programs. When called (invoked), a function executes and carries out a specific task in a program and can receive input data from another function. These input data values are known as arguments. The function can also return output data when it completes execution.

This chapter provides more details on function definitions, invocation, and decomposition. It also discusses the basic mechanisms for data transfer between two functions; several examples are included. Scientific and mathematical built-in functions are used in the case studies.

4.2 Modular Decomposition

As mentioned previously, a problem is often too large and complex to deal with as a single unit. In problem solving and algorithmic design, the problem is partitioned into smaller problems that are easier to solve. The final solution consists of an assembly of these smaller solutions. The partitioning of a problem into smaller parts is known as *decomposition*. These small parts are known as *modules*, which are much easier to develop and manage.

With modular design, the solution to a problem consists of several smaller solutions corresponding to each of the subproblems. A problem is divided into smaller problems (or subproblems), and a solution is designed for each subproblem. These modules are considered building blocks for constructing larger and more complex algorithms.

4.3 Defining Functions

SPSCL and C programs are normally decomposed into source files, and these are divided into functions. A function is the smallest decomposition unit or module in a program. A function performs a specific sequence of operations to accomplish a particular task. A previously defined function can be invoked or called.

A *function definition* is a complete implementation of the function in the programming language. This definition includes the function header and the body of the function, which consists of local data definitions and instructions.

The function header or function prototype describes or specifies the type of data returned by the function, the name of the function, and the parameter list. This list includes the type and name of each parameter. The general form of a function in SPSCL is:

```
description
    .  .  .    */
function ⟨ funct_name ⟩ return type ⟨ type ⟩  parameters ⟨
param_list ⟩ is
    .  .  .  local declarations
  begin
    .  .  .  statements
endfun ⟨ function_name ⟩
```

The type specified in a function definition must be a primitive type, such as *int*, *float*, *double*, *char*, an aggregate type, or a programmer-defined type. The parameter list declares all the parameters of the function

In C, the parameter list must be written within parentheses. The general form of a function definition in the C programming language is:

```
⟨ type ⟩ ⟨ function_name ⟩ ⟨ (parameter_list) ⟩
{
    .  .  .  local data declarations
    .  .  .  statements
}
```

As mentioned previously, local data declared within a function is known only to that function—the scope of the data is *local* to the function. The local data in a function has a limited lifetime; it only exists during execution of the function. The following sample SPSCL code defines a function with name *disp_message*, with no return type and no parameters. This functions does not include local data declarations.

```
description
This function displays a message on the screen. */
function disp_message is
begin
     display "Computing data"
endfun disp_message
```

The following sample C code defines a function with name *disp_message*, which uses *void* as its type has no parameters and no local data declarations.

```
/*
This function displays a message
 on the screen. */
void disp_message() {
     printf("Computing data");
}
```

4.4 Calling Functions

A function is called by another function and starts executing immediately. Upon completion, the called function returns control to the calling function and this first function continues execution.

In SPSCL, the name of the function is included after the keyword **call** when it is called or invoked by some other function. This SPSCL statement is used for a calling a function that was defined without a return type. The arguments are included after the keyword **using**. The SPSCL statement for a calling a function that was defined without a return type is:

call ⟨ *function_name* ⟩ **using** ⟨ *argument_list* ⟩

Suppose function *main* calls function *disp_message*. After completing execution function *disp_message* returns control to function *main* without returning a value. Function *main* continues execution from the point after the call. The function call to function *disp_message* is shown in following SPSCL statement.

```
call disp_message
```

In C, the name of the function is used when it is called or invoked by some other function. After the name of the function, a parenthesis pair is required, even if there are no arguments. The function call to function *disp_message* is shown in following C statement.

```
disp_message();
```

4.5 Classification of Functions

Data transfer is possible between the calling function and the called function. This data transfer may occur from the calling function to the called function, from the called function to the calling function, or in both directions. With respect to data transfer, there are four categories of functions:

1. Simple functions that do not return any value when they are called. The previous example, function *show_message*, is a simple (or void) function because it does not return any value to the function that invoked it.
2. Functions that return a single value after completion.
3. Functions that declare one or more parameters, which are data items used in the function.
4. Functions that have data transfers combined as in (2) and (3). These functions declare one or more parameters and return a value to the calling function.

4.5.1 Simple Function Calls

Simple functions do not return a value to the calling function. An example of this kind of function is *disp_message*, discussed previously. There is no data transfer to or from the function.

Figure 4.1 shows a function *A* calling a second function *B*. After completing its execution, the called function *B* returns the flow of control to function *A*.

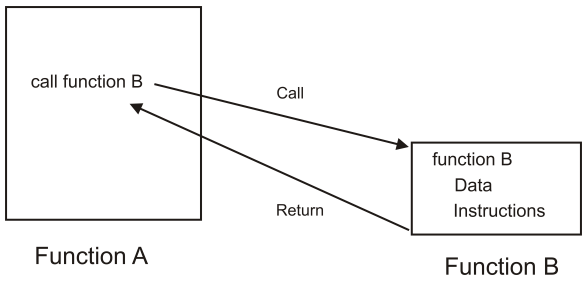

Figure 4.1: A function calling another function

4.5.2 Calling Functions that Return Data

Value-returning functions transfer data from the called function to the calling function. In the called function, a single value is computed and is returned to the calling function. The value-returning function can be called and the value returned can be used in several ways:

- Call the function in a simple assignment statement
- Call the function in an assignment statement with an arithmetic expression

The function is defined with a type, which is the type of the return value. The *return type* may be a simple type or a programmer-defined type.

This function definition must include at least one return statement, which is written with the keyword **return** followed by an expression. The value in the return statement can be any valid expression, following the **return** keyword. The expression can include constants, variables, or a combination of these.

The following example defines a function, *square*, that returns the value of the square of variable *x*. The computed value is returned to the calling function. In the header of the function, the type of the value returned is indicated as **float**. The SPSCL code for this function definition is:

```
description
    This function returns the square of variable x.  */
function square return type float is
    variables
         define x of type float
         define sqx of type float
    begin
        set x = 3.15
        set sqx = x^2
        return sqx
endfun square
```

Calling a function that returns a value in SPSCL is written in an assignment statement. After the execution of the called function completes, control is returned to the calling function with a single value. The following example calls function *square* that returns a value of type *float*. This example consists of an assignment statement that calls function *square* and the returned value is assigned to variable *y*, also of type *float*. Note that the name of the function is followed by a parenthesis pair.

```
set y = square()
```

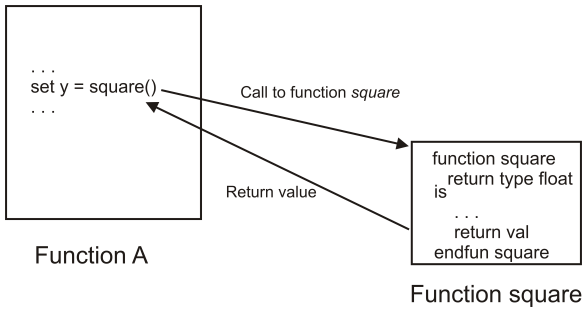

Figure 4.2: Function *A* calling function *square*

Figure 4.2 shows a function with name *A* that calls function *square*. The return value from function *square* is used directly by function *A* in an assignment statement that assigns the value to variable *y*.

With the assignment statement, the function call can occur in a more complex expression. In the following example, the final value of the expression is assigned to variable *fres*.

```
set fres = y + w * square() + 23.45
```

Using the C Language

The general form of the C language statement that defines the form of a value-returning function is:

```
⟨ return_type ⟩ ⟨ function_name ⟩ (⟨ parameter_list ⟩)
{
      . . .
      return ⟨ return_value ⟩
}
```

The following example in C, defines a function, *square*, that returns the value of the square of variable x. The computed value is returned to the calling function. In the header of the function, the type of the value returned is indicated as **float**.

```
 /*
    This function returns the square of variable x.
 */
float square () {
      float x;
      float sqx;
      x = 3.15;
      sqx = x * x;
      return sqx;
}
```

The value returned by the called function is used by the calling function by assigning this returned value to another variable. The following example shows a function calling function *square*. The calling function assigns the value returned to variable *res*.

```
res = square();
fres = y + w * square() + 23.45;
```

The value returned is used in an assignment with an arithmetic expression after calling a function. In the example shown, after calling function *square*, the value returned is assigned to variable *res*. In the second line of C code, the function call occurs in an arithmetic expression and the final value is assigned to variable *fres*.

4.5.3 Calling Functions with Arguments

A function that is called with arguments must be defined with one or more parameters, which are data declarations in the *parameter* list, using the **parameters** keyword. These parameter declarations are similar to local data declarations and have local scope and persistence.

The parameter list consists of one or more parameter declarations that follow the **parameters** keyword and are separated by commas. Each parameter declaration includes the type and name of the parameter. A function with parameters is very useful because it allows data transfer into the function when called. The data values used when calling a function are known as *arguments*. The parameters are also known as *formal parameters*, and the arguments are also known as *actual parameters*.

Every argument in the function call must be consistent with the corresponding parameter declaration in the function definition. The following example shows the definition of function *squared* that includes one parameter declaration, *p*, and returns the value of local variable, *res*.

Figure 4.3 illustrates an example in which one parameter is declared in the function definition of *squared*. The function call requires an argument and a call to the function computes the square of the value in the argument; the called function returns the value computed.

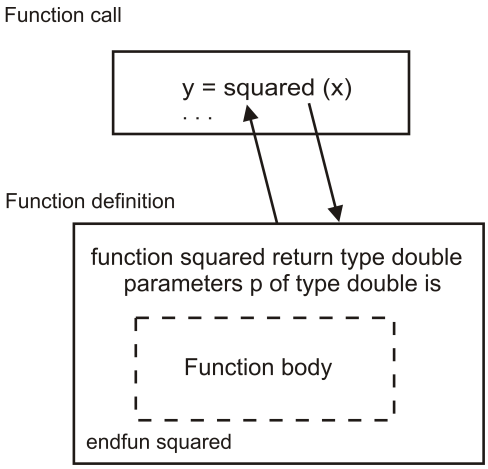

Figure 4.3: Calling function *squared* with an argument.

```
function squared return type double
     parameters  p  of type double
is
   variables
        define res of type double
begin
    set res = p^2
    return res
endfun squared
```

Listing 4.1a shows the complete program in SPSCL . Line 8 includes the function prototype for function *squared* and the function is called in line 18 with argument *x*. The function definition of this function has only one parameter declaration, so the function is called with one argument. It computes the square of the argument value, which is the value of variable *x* and is used as the input value in the function call. The value returned by the function call is assigned to variable *y*.

Listing 4.1a: The SPSCL program that computes the square of a value.

```
 1 /*
 2  Program     : squareprog.scl
 3  Author      : Jose M Garrido
 4  Description : Compute the square of the value of a variable.
 5  */
 6 import "scl.h"
 7 forward declarations
 8 function squared return type double
 9      parameters s of type double   // prototype
10 implementations
```

```
11 function main is
12 variables
13     define x of type double       // variable declaration
14     define y of type double
15 begin
16     display "Square of a value "
17     input "Enter value: ", x      // read a value of variable x
18     set y = squared(x)            // compute square of x
19     display "The square of ", x, " is: ", y
20     return 0                      // execution terminates OK
21 endfun main
22 //
23 function squared return type double
24      parameters p of type double is
25   variables
26     define res of type double
27   begin
28     set res = p ^ 2
29     return res
30 endfun squared
```

The following listing includes the commands on Windows that translates the SPSCL program, compiles and links the C program, and executes the program using an input value of 3.5.

```
C:\SPSCL >spscl squareprog.scl
SPSCL  v 1.1 File: squareprog.scl Sat Feb 01 10:09:36 2022
File: squareprog.scl no syntax errors, lines processed: 30

C:\SPSCL >gcc squareprog.c -o squareprog.exe -lm
C:\SPSCL >squareprog
Square of a value
Enter value: 3.5
The square of 3.500000  is: 12.250000
```

Using the C Language

The parameter list is enclosed with parenthesis and consists of one or more parameter declarations separated by commas. Each parameter declaration includes the type and name of the parameter. A function with parameters is very useful because it allows data transfer into a function when called. The data values used when calling a function are known as *arguments*. The parameters are also known as *formal parameters*, and the arguments are also known as *actual parameters*.

Every argument in the function call must be consistent with the corresponding parameter declaration in the function definition. The following example shows the definition of function *squared* that includes one parameter declaration, *p*, and returns the value of local variable, *res*.

```
double squared (double p)  {
    double res;
    res = pow(p,2);
    return res;
}
```

Listing 4.1b shows a short but complete program in C. Line 9 is the function prototype for function *squared* and the function is called in line 16 with argument *x*. The function definition of this function has only one parameter declaration, so the function is called with one argument. It computes the square of the argument value, which is the value of variable *x* and is used as the input value in the function call. The value returned by the function call is assigned to variable *y*.

Listing 4.1b: C program to compute the square of a value.

```
 1 // SPSCL  v 1.0 File: squareprog.c, Sat Feb 01 10:09:36 2022
 2 /*
 3  Program     : squareprog.scl
 4  Author      : Jose M Garrido
 5  Description : Compute the square of the value of a variable.
 6  */
 7 #include "scl.h"
 8 // forward declarations
 9 double  squared( double  s );   // prototype
10 int main() {
11   double  x;   // variable declaration
12   double  y;
13   printf("Square of a value  \n");
14   printf("Enter value: ");
15   scanf("   %lf",  &x);  // read a value of variable x
16   y = squared(x);        // compute square of x
17   printf("The square of   %lf  is:   %lf\n",x, y);
18   return 0;      // execution terminates OK
19 }  // end main
20 //
21 double  squared( double  p ) {
22    double  res;
23    res = (p) * (p);
24    return res;
25 }  // end squared
```

The following listing shows the Linux shell commands that translate, compile, link, and execute the program.

```
$ ./spscl squareprog.scl
SPSCL  v 1.1 File: squareprog.scl Sat Feb 01 10:09:36 2022
File: squareprog.scl no syntax errors, lines processed: 30
$ gcc squareprog.c squareprog.out -lm
$ ./squareprog.out
Square of a value
Enter value: 45.95
The square of 45.950000 is: 2111.402500
```

4.6 Built-in Mathematical Functions

SPSCL programs can call all functions available in the C programming language. This includes a wide variety of functions organized and stored in various libraries and provide the SPSCL program

with many pre-defined functions. On such library is the mathematical library. In SPSCL programs the access to this library is included by the `import "scl.h"` command at the top of an SPSCL program.

An example is the library of trigonometric functions. The following language statement includes an expression that calls function *sin* applied to variable *x*, which is the *argument* written in parentheses and its value is assumed in radians. The value returned by the function is the sine of *x*.

```
set y = 2.16 + sin(x)
set j = 0.335
set z = x * sin(j * 0.32)
```

To compute the square root of the value of a variable, z expressed mathematically as: $\sqrt{z}$, the mathematical library provides the *sqrt()* function. For example, to compute the square root of z and assign the value to variable q, the SPSCL language statement is:

```
set q = sqrt(z)
```

Given the following mathematical expression:

$$var = \sqrt{\sin^2 x + \cos^2 y}$$

The assignment statement to compute the value of variable *var* uses three functions: *sqrt()*, *sin()*, and *cos()* and is coded as:

```
set var = sqrt( sin(x)^ 2 + cos(y)^ 2 )
```

The exponential function *exp()* computes e raised to the given power. The following statement computes $q = y + x e^k$.

```
set q = y + x * exp(k)
```

To compute the logarithm base e of x, denoted mathematically as $\ln x$ or $\log_e x$, function *log()* is called. For example:

```
set t = log((q-y)/x)
```

The following table lists the basic mathematical functions in the mathematical library.

Function	Description
`fabs(x)`	Returns the absolute value of x
`sqrt(x)`	Returns the square root of x, $x \geq 0$
`pow(x, y)`	Returns x to the power of y
`ceil(x)`	Returns the nearest integer larger than x
`floor(x)`	Returns the nearest integer less than x
`exp(x)`	Returns the value e^x, e is the base for natural logarithms
`log(x)`	Returns the natural logarithm of x (base e), $x > 0$
`log10(x)`	Returns the logarithm base 10 of x, $x > 0$

The following table lists the trigonometric and hyperbolic functions of the library.

Function	Description
`sin(x)`	Returns the sine of x, where x is in radians
`cos(x)`	Returns the cosine of x, where x is in radians
`tan(x)`	Returns the tangent of x, where x is in radians
`asin(x)`	Returns the arcsine of x, where $-1 < x < 1$
`acos(x)`	Retirns the arccosine of x, where $-1 < x < 1$
`atan(x)`	Returns the arctangent of x
`atan2(y, x)`	Returns the arctangent of the value y/x
`sinh(x)`	Returns the hyperbolic sine of x
`cosh(x)`	Returns the hyperbolic cosine of x
`tanh(x)`	Returns the hyperbolic tangent of x

Using the C Language

The C programming language include libraries of built-in functions that compute the most common mathematical operations. To use the C standard mathematical library, the program must include at the top of the source program the following compiler directive:

```
#include <math.h>
```

The following language statement includes an expression that calls function *sin* applied to variable x, which is the *argument* written in parentheses and its value is assumed in radians. The value returned by the function is the sine of x.

```
y = 2.16 + sin(x);
j = 0.335;
z = x * sin(j * 0.32);
```

To compute the value of a variable to a given power, function *pow* is used for exponentiation. The function computes x^n and needs two arguments the variable and the exponent. The following simple assignment statement computes x^n, which is the value of variable x to the power of n and assigns the value computed to variable y.

```
y = pow(x, n);
```

To compute the square root of the value of a variable, z expressed mathematically as: $\sqrt{z}$, the C programming language provides the *sqrt()* function. For example, to compute the square root of z and assign the value to variable q, the C language statement is:

```
q = sqrt(z);
```

Given the following mathematical expression:

$$var = \sqrt{\sin^2 x + \cos^2 y}$$

The C assignment statement to compute the value of variable *var* uses four functions: *sqrt()*, *pow*, *sin()*, and *cos()* and is coded as:

```
var = sqrt( pow(sin(x), 2) + pow(cos(y), 2) );
```

The exponential function *exp()* computes e raised to the given power. The following C statement computes $q = y + x e^k$.

```
q = y + x * exp(k);
```

To compute the logarithm base e of x, denoted mathematically as $\ln x$ or $\log_e x$, C provides function *log()*. For example:

```
t = log((q-y)/x);
```

The following example describes a program that computes the area and perimeter of a triangle, given its the three sides. Listing 4.2 shows the complete SPSCL program. The source files are stored in files `ctriangle.scl` and `ctriangle.c`

Listing 4.2 Computing the area and perimeter of a triangle.

```
/*
      This program computes the area and perimeter of a triangle, given
       its the three sides.
      Program: ctriangle.scl
      Revised January 2022.  J. Garrido
*/
import "scl.h"
implementations
  description
    This function computes the perimeter of a
    triangle. */
  function cperimeter return type double
  parameters
       x of type double,  // first side
       y of type double,  // second side
```

```
       z of type double   // third side
  is
  variables
       define lperim of type double
  begin
       set lperim = x + y + z
       return lperim
endfun cperimeter
//
description
     This function computes the area of a
     triangle.     */
function carea return type double
  parameters
       x of type double,  // first side
       y of type double,  // second side
       z of type double   // third side
  is
  variables
       define s of type double    // intermediate result
       define r of type double
       define larea of type double
  begin
       set s = 0.5 * (x + y + z)
       set r = s * (s - x)*(s - y)*(s - z)
       set larea = call sqrt using r
       return larea
endfun carea
//
description
      This function gets the area and perimeter of a triangle.
      */
function main is
  variables                    // data declarations
       define x of type double    // first side of triangle
       define y of type double   // second side
       define z of type double   // third side
       define area of type double
       define perim of type double
  begin
      // body of function starts here
      display "This program computes the area of a triangle"
      input "enter value of first side: ", x
      input "enter value of second side: ", y
      input "enter value of third side: ", z
      set area = call carea using x, y, z
      // display "Area of triangle is: ", area
      set perim = call cperimeter using x, y, z
      // display "Perimeter of triangle is: ", perim
      display "Area: ", area, " Perim: ", perim
      exit
```

```
endfun main
```

The following shell commands translate SPSCL to C, compile, link, and execute the program that computes the area and perimeter of the triangle.

```
$ spscl ctriangle.scl
$ gcc -Wall ctriangle.c -o ctriangle.out basic_lib.o -lm
$ ./ctriangle.out
```

Summary

Functions are fundamental modular units in a program. A function has to be defined in order to be called and calling functions involves several mechanisms for data transfer. Calling simple functions does not involve data transfer between the calling function and the called function. Value-returning functions return a value to the calling function. Calling functions that define one or more parameters involve values sent by the calling function and used as input in the called function.

Key Terms

modules	functions	function definition
function call	local declaration	arguments
return value	assignment	parameters
pre-defined functions	built-in functions	libraries

Exercises

Exercise 4.1 Why is a function a decomposition unit? Explain.

Exercise 4.2 What is the purpose of function *main*?

Exercise 4.3 Explain variations of data transfer among functions.

Exercise 4.4 Write the SPSCL code of a function defined with more than two parameters.

Exercise 4.5 Write the SPSCL code that calls the function that was defined with more than two parameters.

Exercise 4.6 Develop an SPSCL program that defines two functions, one to compute the area of a triangle, the other function to compute the circumference of a triangle. The program must include the function prototypes and call these functions from *main*, which must input the corresponding values.

Exercise 4.7 Develop an SPSCL program that defines two functions, one to compute the area of a circle, the other function to compute the circumference of a circle. The program must include the function prototypes and call these functions from *main*, which must input the corresponding values.

Exercise 4.8 Develop an SPSCL program that defines a function that computes the volume of a cylinder. The program must include the function prototype and call this function from *main*, which must input the corresponding values.

Exercise 4.9 Develop an SPSCL program that defines a function that computes the volume of a sphere. The program must include the function prototype and call this function from *main*, which must input the corresponding values.

Exercise 4.10 Develop an SPSCL program that defines two functions, one to compute the area of an ellipse, the other function to compute the circumference of an ellipse. The program must include the function prototypes and call these functions from *main*, which must input the corresponding values.

Exercise 4.11 Develop an SPSCL program that computes the slope of a line between two points in a plane: P_1 with coordinates (x_1, y_1), and P_2 with coordinates (x_2, y_2). The program should include two functions: *main* and *slopef*. The parameters of the second function (*slopef*) are the coordinates of the points in a plane. The coordinate values must be read in function *main*.

5. Algorithms and Control Design Structures

5.1 Introduction

The first part of this chapter presents general concepts and discussions of design control structures, which are used in designing and implementing an algorithm. In implementation, these control structures help to structure and organize the program. A complete case study is developed.

The second part of the chapter explains in more detail the selection control structure and the corresponding statements in pseudo-code, SPSCL, and C statements for implementing programs.

Conditions are expressions that evaluate to a truth-value (true or false). Conditions are used in the selection statements. Simple conditions are formed with relational operators for comparing two data items. Compound conditions are formed by joining two or more simple conditions with *logical* operators.

The solution to a quadratic equation is discussed as an example of applying the selection control statements.

As mentioned previously, the purpose of computer problem solving is to design a solution to a problem; an algorithm describes precisely this design and is implemented into a computer program. Analyzing the problem includes understanding the problem, identifying the given (input) data and the required results. Developing a program involves implementing a computer solution to solve some real-world problem. Design of a solution to the problem requires finding some method to solve the problem.

Designing a solution to the problem consists of defining the necessary computations to be carried out in an appropriate sequence to the given data to produce the final required results.

The design of the solution to a problem is described by an *algorithm*, which is a complete and precise sequence of steps that need to be carried out to achieve the solution to a problem. After the algorithm has been formulated and written, its implementation and the corresponding data definitions is carried out with a with a programming language such as SPSCL and C.

5.2 Algorithms

An algorithm is a clear, detailed, precise, and complete description of the sequence of steps to be performed in order to produce the desired results. An algorithm can be considered the transformation on the given data and involves a sequence of commands or operations that are to be carried out on the data in order to produce the desired results.

The algorithm is usually broken down into smaller tasks; the overall algorithm for a problem solution is decomposed into smaller algorithms, each defined to solve a subtask.

A computer implementation of an algorithm consists of a group of data definitions and one or more sequences of instructions to the computer for producing correct results when given appropriate input data. The implementation of an algorithm is in the form of a program, which is written in an a programming language and it indicates to the computer how to transform the given data into correct results. An algorithm is often described in a semi-formal notation such as pseudo-code and flowcharts.

5.3 Implementing Algorithms

As discussed in previous chapters, the implementation of an algorithm is carried out by writing the code in a programming language, such as SPSCL , C, C++, Ada, Fortran, Eiffel, or Java .

Programming languages have well-defined syntax and semantic rules. The syntax is defined by a set of grammar rules and a vocabulary (a set of words). The legal sentences are constructed using sentences in the form of *statements*. There are two groups of words that are used to write the statements: *reserved words* also known as *keywords* and *identifiers*.

Reserved words are the keywords of the language and have a predefined purpose. These are used in most statements. Examples are: **for**, **endfor**, **function**, **while**, and **if**. Identifiers are names for variables, constants, and functions that the programmer chooses, for example, *height*, *temperature*, *pressure*, *number_units*, and so on.

5.4 Algorithm Description

Designing a solution to a problem consists of designing and defining an algorithm, which will be as general as possible in order to solve a family or group of similar problems. An algorithm can be described at several levels of abstraction. Starting from a very high level and general level of description of a preliminary design, to a much lower level that has more detailed description of the design.

Several notations are used to describe an algorithm. An algorithmic notation is a set of general and informal rules used to describe an algorithm. Two widely used notations are:

- Flowcharts
- Pseudo-code

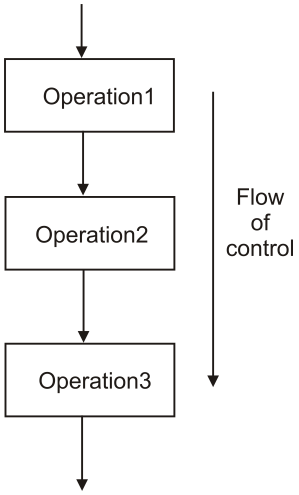

Figure 5.1: Flow of control.

5.4.1 Flowcharts

The flow of control in a program is the order in which the operations will be executed. The most basic flow of control is sequential—the operations are executed in sequence, as seen in Figure 5.1.

A flowchart is a visual representation of the flow of the data and the operations on this data. A flowchart consists of a set of symbolic blocks connected by arrows. The arrows that connect the blocks show the order for describing a sequence of design or action steps. The arrows also show the flow of data.

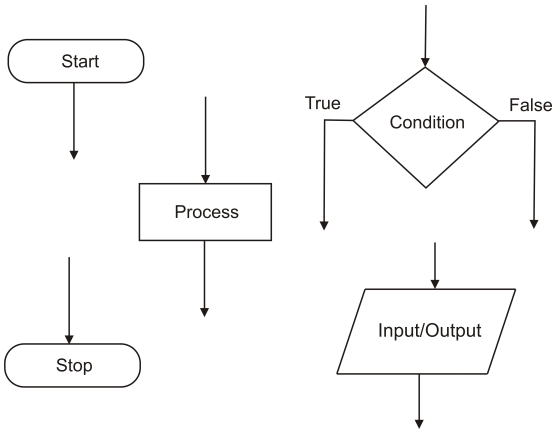

Figure 5.2: Basic flowchart symbols.

Several basic flowchart blocks are shown in Figure 5.2. Every flowchart block has a specific symbol. A flowchart always begins with a *start* symbol, which has an arrow pointing from it. A flowchart ends with a *stop* symbol, which has one arrow pointing to it.

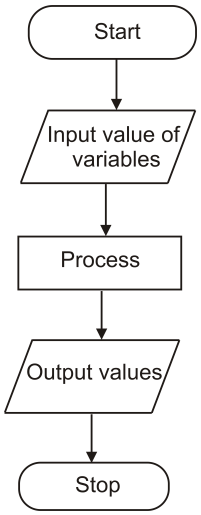

Figure 5.3: A simple flowchart example.

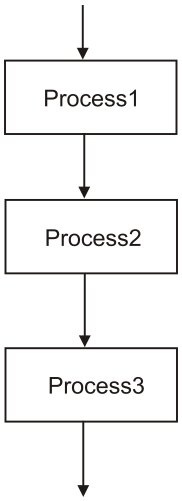

Figure 5.4: A flowchart with a sequence.

The *process* or *transformation* symbol is the most common and general symbol, shown as a rectangular box. A process block represents one or more operations of computation. This symbol is used to represent any computation or sequence of computations carried out on some data. There is one arrow pointing to it and one arrow pointing out from it.

The *selection* flowchart symbol has the shape of a vertical diamond and represents a *selection* of alternate paths in the sequence of design steps. It is shown in Figure 5.2 with a condition that is evaluated to *True* or *False*. This symbol is also known as a *decision block* because the sequence of instructions can take one of two directions in the flowchart, based on the evaluation of a condition.

The *input-output* flowchart symbol is used for a data input or output operation. There is one arrow pointing into the block and one arrow pointing out from the block.

An example of a simple flowchart with several basic symbols in shown in Figure 5.3. For larger or more complex algorithms, flowcharts are used mainly for the high-level description of the algorithms and pseudo-code is used for describing the details.

5.4.2 Pseudo-Code

Pseudo-code is an informal notation that uses a few simple rules and English for describing the algorithm that defines a problem solution. It can be used to describe relatively large and complex algorithms. It is relatively easy to convert the pseudo-code description of an algorithm to a computer implementation in a high-level programming language.

5.5 Control Design Structures

An algorithm can be completely defined with only four fundamental control design structures. These control structures can be specified using flowcharts and with pseudo-code notations. The design structures are:

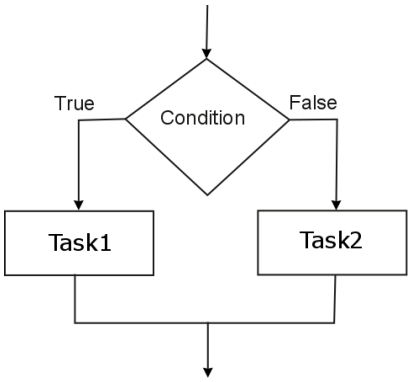

Figure 5.5: Selection structure.

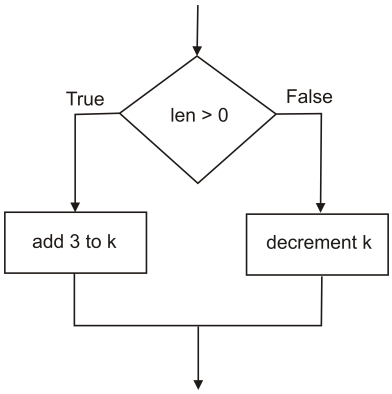

Figure 5.6: An example of the selection structure.

1. *Sequence*, describes a sequence of operations.
2. *Selection*, this part of the algorithm takes a decision and selects one of several alternate paths of flow of actions. This structure is also known as alternation or conditional branch.
3. *Repetition*, this part of the algorithm has a block or sequence of steps that are to be executed zero, one, or more times.
4. *Input-output*, the values of variables are read from an input device (such as the keyboard) or the values of the variables (results) are written to an output device (such as the screen)

5.5.1 Sequence

A sequence control structure consists of a group of operations that are to be executed one after the other, in the specified order, as shown in Figure 5.1. The symbol for a sequence can be directly represented by two or more *process* blocks connected by arrows in a flowchart. Figure 5.4 illustrates the sequence structure with several blocks. The sequence structure is the most common and basic structure used in algorithmic design.

5.5.2 Selection

With the selection control structure, one of several alternate paths of the algorithm will be chosen based on the evaluation of a condition. Figure 5.5 illustrates the selection structure in flowchart form. In the figure, the actions or instructions in *Task1* are executed when the condition is true. The instructions in *Task2* are executed when the condition is false.

A concrete and simple flowchart example of the selection structure is shown in Figure 5.6. The condition of the selection structure is $len > 0$ and when this condition evaluates to true, the block with the action `add 3 to k` will execute. Otherwise, the block with the action `decrement k` will execute.

5.5.3 Repetition

The repetition control structure indicates that a set of action steps are to be repeated several times. Figure 5.7 shows this structure. The execution of the actions in the *Process* block are repeated while the condition is true. This structure is also known as the *while loop*.

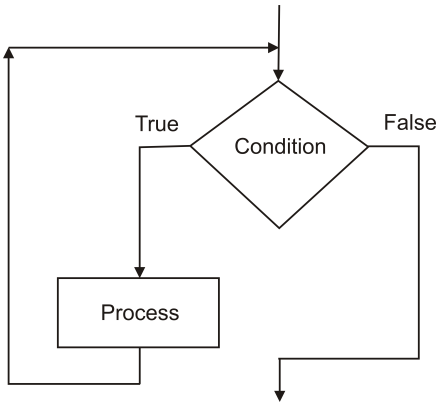

Figure 5.7: While loop of the repetition control structure.

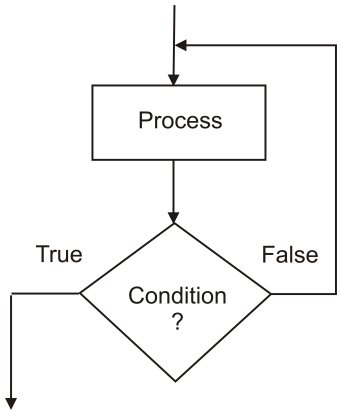

Figure 5.8: Repeat-until loop of the repetition control structure.

A variation of the repetition control structure is shown in Figure 5.8. The actions in the *Process* block are repeated until the condition becomes true. This structure is also known as the *repeat-until* loop.

5.5.4 Simple Input/Output

Input and output statements are used to read (input) data values from the input device (e.g., the keyboard) and write (output) data values to an output device (mainly to the computer screen). The flowchart symbol is shown in Figure 5.9.

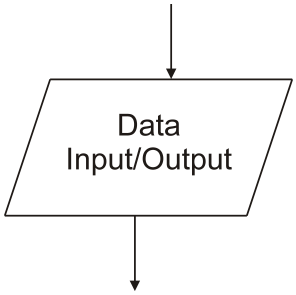

Figure 5.9: Flowchart data input/output symbol.

Output

In SPSCL, the output statement that can be used for the output of a list of variables and literals; it is written with the keyword **display**. The output statement writes the value of one or more variables to the output device. The variables do not change their values. The general form of the output statement is:

display ⟨ *data_list* ⟩

```
display "value of x= ", x, "value of y = ", y
```

In C, function *printf* can display a single string and data value, so it can be used individually to display every variable and its value. The general form for this function call in C is:

```
printf (" [string], [format operators]", [variable list] );
```

The previous example outputs the value of two variables, *x* and *y*, as shown next:

```
printf ("values of x: %lf and y:  %lf  \n", x, y);
```

Input

The input statement using SPSCL reads a value of one or more variables from the input device (e.g., the keyboard). This statement is written with the keywords **read**.

The following two lines include the general form of the input statement and an example that uses the *read* statement to read a value of variable *y*.

read ⟨ *var_list* ⟩

read y

The read statement implies an assignment statement for the variable *y*, because the variable changes its value to the new value that is read from the input device. A string message is typically included to prompt the user for input of a data value. The **input** statement is also useful to read the value of a single variable, as seen in previous chapters. For example:

input "Enter value of y: ", y

The C programming language provides functions for console input. The function for input reads the value of a variable from the input device. The name of the function is *scanf*, and is used with a format operator. The general form for input in C for reading the value of a numeric floating-point variable, *var*, is shown in the following line of code.

```
scanf (" %lf", &var);
```

A string message is typically included to prompt the user for input of a data value. This is implemented with the *printf* function. The previous example for reading the value of variable *y* is written in C as follows:

```
printf ("Enter value of y: ");
scanf ("%lf", &y);
```

5.6 Conditional Expressions

A Conditional expression, also known as a Boolean expression, consists of an expression that evaluates to a truth-value, *true* or *false*.

5.6.1 Relational Operators

A conditional expression can be constructed by comparing the values of two data items and using a relational operator. The following list of relational operators in arithmetic notation can be used in a condition:

Arithmetic Operator	Description
$>$	Greater than
$<$	Less than
$=$	Equal to
$\leq$	Less or equal to
$\geq$	Greater or equal to
$\neq$	Not equal to

These relational operators are used to construct conditions as in the following examples in algebra notation:

$\texttt{y} \leq 20.15$
$p \geq q$
$a = b$

With programming languages such as SPSCL, C, C++, and others, the relational operators use the following notation:

Relational Operator	Description	Arithmetic notation
>	Greater than	$>$
<	Less than	$<$
==	Equal to	$=$
>=	greater than or equal to	$\geq$
<=	Less than or equal to	$\leq$
!=	Not equal to	$\neq$

In SPSCL, these examples of conditional expressions can be written as follows:

```
y <= 20.15
p >= q
a == b
```

The SPSCL syntax also provides keywords that can be used for the relational operators. For example, the previous conditional expressions can also be written as follows:

```
y less or equal 20.15
p greater or equal q
a equal b
```

Arithmetic expressions can be used as part of conditional expression. For example:

```
y >= (x + 45.6)
```

When this conditional expression is evaluated, the arithmetic expression $(x+45.6)$ is evaluated first, then the relational operators are applied. The order in which these operators are evaluated is specified in the following table:

Operator	Order of evaluation
( )	1
*, /, **mod**	2
+, -	3
=, <, >, <=, >=, !=	4

A conditional expression can be assigned to a variable of type **bool**. For example, the following statement assigns the value of the previous conditional expression to variable *y_flag*, which was declared of type **bool**:

```
set y_flag = y >= (x + 45.6)
```

5.6.2 Logical Operators

A compound conditional expression consists of one or more simple conditional expressions. The logical operators **and**, **or**, and **not** are used to construct compound conditional expressions from simpler conditions.

The **not** logical operator is applied with one simple condition, *cond*. If a compound condition is defined as **not** *cond*, the truth value of the compound condition is simply the negation of the truth value of the simple condition. For example, when the simple condition *cond* has truth value **false**, the resulting compound condition has a truth value of **true**. The opposite also applies, when the truth value of *cond* is **true**, the truth value of the compound condition is **false**. These rules are summarized in the following table.

cond	**not** cond
true	false
false	true

The **and** and **or** logical operators are used with two simple conditions *cond1* and *cond2*. The rules that apply for the truth value of the resulting compound condition are summarized in the following table.

cond1	cond2	cond1 **and** cond2	cond1 **or** cond2
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

The table shows that when the **and** logical operator is applied to two simple conditions, the truth value of the resulting compound condition is **true** when the truth values of both simple conditions are **true**. When the **or** logical operator is applied, the truth value of the compound condition is **true** when either or both simple conditions have value **true**.

The general forms of a compound conditions from the simple conditions, *cond1* and *cond2* in SPSCL are:

```
cond1 and cond2
cond1 or cond2
not cond1
```

The following examples include the **or** and the **and** logical operators:

```
(y > 12) or (x < 23)
(p==q) and (z <= r)
```

The previous conditions can also be written in the following manner:

```
(y greater than 12) or (x less than 23 )
(p equal to q) and (z less or equal to r)
```

The following example includes the **not** operator:

```
not (y > 12)
```

C Language

The C programming language provides several logical operators and functions for constructing complex conditions from simpler ones. The basic logical operators for scalars (individual numbers) in C are: the `&&` (and) operator, the `||` (or) operator, and the `!` (not) operator.

C Operator	Description
&&	And
||	Or
!	Not

In the following example, two simple conditions are combined with the `&&` operator (and). The result condition evaluated to *true*.

```
x = 5.25;
(2.5 < x) && (x < 21.75)
```

In C, the boolean type is available since C99 compliant compilers, as type *bool*. The header file `addtypes.h` defines type `bool`.

5.7 Selection Structure

The selection design structure is also known as *alternation*, because alternate paths are considered, based on the evaluation of a condition. This section covers describing the selection structure with flowcharts, the concepts associated with conditional expressions, and implementation in the SPSCL and C programming languages.

5.7.1 Selection Structure with Flowcharts

The selection structure is used for decision-making in the logic of a program. Figure 6.1 shows the selection design structure using a flowchart. Two possible paths for the execution flow are shown. The condition is evaluated, and one of the paths is selected. If the condition is true, then the left path is selected and *Task1* is performed. If the condition is false, the other path is selected and *Task2* is performed.

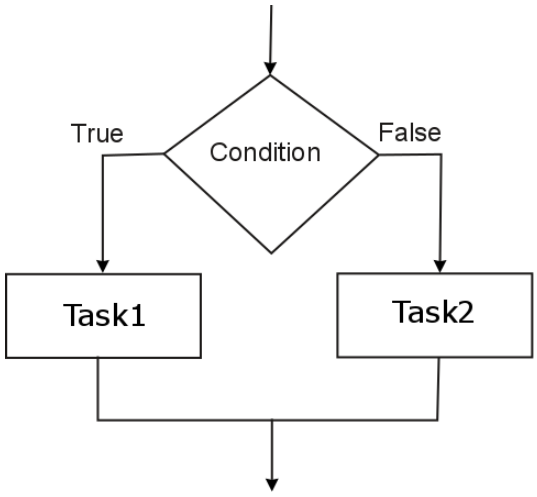

Figure 5.10: Flowchart of the selection structure.

5.7.2 Selection with SPSCL

With the SPSCL language, the selection structure is written with an **if** statement, also known as an if-then-else statement. This statement includes three sections: the *condition*, the *then-section*, and the *else-section*. The else-section is optional. Several keywords are used in this statement: **if**, **then**, **else**, and **endif**. The general form of the **if** statement is:

```
if ⟨ condition ⟩
   then
       ⟨ statements in Process1 ⟩
   else
       ⟨ statements in Process2 ⟩
endif
```

When the condition is evaluated, only one of the two alternatives will be carried out: the one with the statements in *Process1* if the condition is *true*, or the one with the statements in *Process2* if the condition is *false*.

5.7.3 Implementing Selection with the C Language

Similarly to pseudo-code, the C language includes the **if** statement that allows checking for a specified condition and executing statements if the condition is met. Note that C does not have the **then** keyword, but uses braces ({) and (}) instead of delimit blocks of code. The general form of the **if** statement in is:

```
if (⟨ condition ⟩ )
        { ⟨ statements in Process1 ⟩ }
else
        { ⟨ statements in Process2 ⟩ }
```

The first line specifies the condition to check. If the condition evaluates to **true**, the program executes the body of the first indented block of statements, otherwise the second indented block of statements after the **else** keyword is executed.

SPSCL and C do not require you to indent the body of an **if** statement, but it makes the code more readable and it is good programming practice.

5.7.4 Example with Selection

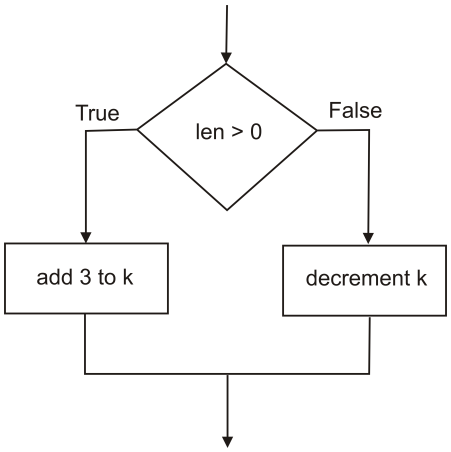

Figure 5.11: Example of selection structure.

The following example evaluates the condition $len > 0$, to select which operation is to be performed on variable k. Figure 6.2 shows the flowchart for part of the algorithm that includes this selection structure. In SPSCL , this example is written as:

```
if len greater than 0
then
    add 3 to k
else
    decrement k
endif
```

The previous example can also be written in the following manner:

```
if len > 0
   then
      set k = k + 3
   else
      set k = k - 1
endif
```

In C, the previous example is written as follows:

```
if (len > 0)
    k= k + 3;
else
    k=k - 1;
```

The following is a selection statement in SPSCL that includes a compound condition:

```
if a < b  or x ≥ y
then
    ⟨ block1 ⟩
else
    ⟨ block2 ⟩
endif
```

5.8 An Example Problem with Selection

The following problem involves developing a program that includes solving a quadratic equation, which is a simple mathematical model of a second-degree equation. The solution to the quadratic equation involves complex numbers.

5.8.1 Analysis

The goal of the solution to the problem is to compute the two roots of the equation. The mathematical model is defined in the general form of the quadratic equation (second-degree equation):

$$ax^2 + bx + c = 0$$

The given data for this problem are the values of the coefficients of the quadratic equation: a, b, and c. Because this mathematical model is a second degree equation, the solution consists of the value of two roots: x_1 and x_2.

5.8.2 Algorithm for General Solution

The general solution gives the value of the two roots of the quadratic equation, when the value of the coefficient a is not zero ($a \neq 0$). The values of the two roots are:

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

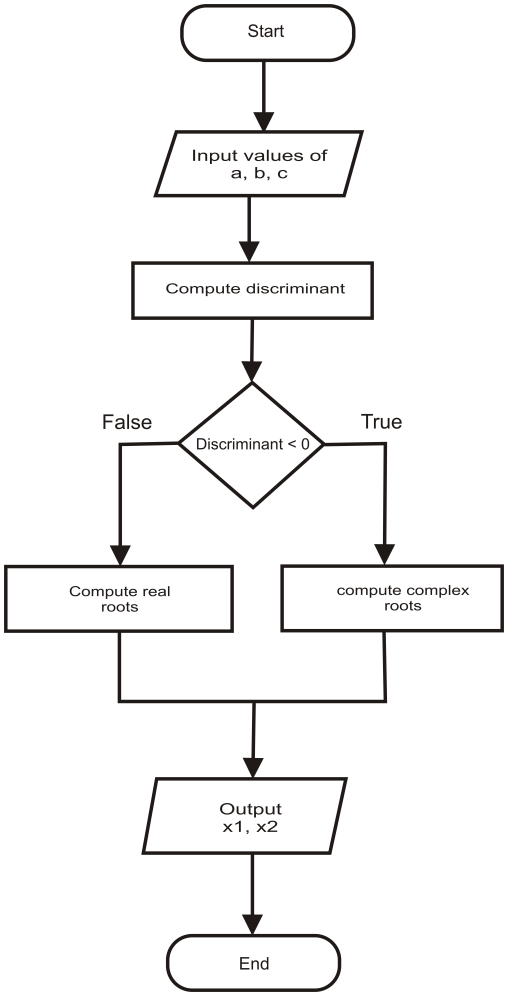

Figure 5.12: High-level flowchart for solving a quadratic equation.

The expression inside the square root, $b^2 - 4ac$, is known as the *discriminant*. If the discriminant is negative, the solution will involve complex roots. Figure 6.3 shows the flowchart for the general solution and the following listing is a high-level pseudo-code version of the algorithm.

```
Input the values of coefficients  a,  b, and  c
Calculate value of the discriminant
if the value of the discriminant is less than zero
     then calculate the two complex roots
     else calculate the two real roots
endif
display the value of the roots
```

5.8.3 Detailed Algorithm

The algorithm in pseudo-code notation for the solution of the quadratic equation is:

```
read the value of a from the input device
read the value of b from the input device
read the value of c from the input device
compute the discriminant, disc = b^2 - 4ac
if discriminant less than zero
then
      // roots are complex
      compute x1 = (-b + sqrt(disc))/2a
      compute x2 = (-b - sqrt(disc))/2a
else
      // roots are real
      compute x1 = (-b + sqrt(disc))/2a
      compute x2 = (-b - sqrt(disc))/2a
endif
display values of the roots:  x1 and x2
```

Listing 5.1a shows the SPSCL program that implements the algorithm for the solution of the quadratic equation, which is stored in the file `solquad.scl`.

Listing 5.1a SPSCL program that computes the roots of a quadratic equation.

```
 1 /*
 2  Program     : solquad.scl
 3  Author      : Jose M Garrido, Nov 21 2021.
 4  Description : Compute the roots of a quadratic equation.
 5     Read the value of the coefficients: a, b, and c from
 6     the input console, display value of roots.
 7 */
 8 import "scl.h"
 9 implementations
10 function main is
11    variables
12      define a of type double    // coefficient a
13      define b of type double
14      define c of type double
```

```
 15      define disc of type double  // discriminant
 16      define x1r of type double   // real part of root 1
 17      define x1i of type double   // imaginary part of root 1
 18      define x2r of type double
 19      define x2i of type double
 20 begin
 21      input "Enter value of cofficient a: ", a
 22      display "Value of a: ", a
 23      input "Enter value of cofficient b: ", b
 24      display "Value of a: ", b
 25      input "Enter value of cofficient c: ", c
 26      display "Value of a: ", c
 27      set disc = b^2 - 4.0 * a * c
 28      display "discreminant: ", disc
 29      if (disc < 0.0)  then
 30           // complex roots
 31           set disc = -disc
 32           set x1r = -b/(2.0 * a)
 33           set x1i = sqrt(disc)/(2.0 * a)
 34           set x2r = x1r
 35           set x2i = -x1i
 36           display "Complex roots "
 37           display "x1r: ", x1r, " x1i: ",  x1i
 38           display "x2r: ", x2r, " x2i: ", x2i
 39      else
 40           // real roots
 41           set x1r = (-b + sqrt(disc))/(2.0 * a)
 42           set x2r = (-b - sqrt(disc))/(2.0 * a)
 43           display "Real roots:"
 44           display "x1: ", x1r, " x2: ", x2r
 45      endif
 46      return 0      // execution terminates ok
 47 endfun main
```

Listing 5.1b shows the C program that implements the algorithm for the solution of the quadratic equation, which is stored in the file `solquad.c`.

Listing 5.1b C program that computes the roots of a quadratic equation.

```
/*
 Program     : solquad.c
 Author      : Jose M Garrido, Feb 3 2022.
 Description : Compute the roots of a quadratic equation.
    Read the value of the coefficients: a, b, and c from
    the input console, display value of roots.
*/
#include "scl.h"
int main(void) {
     double a;   /* coefficient a */
     double b;
     double c;
```

```
     double disc; /* discriminant */
     double x1r;  /* real part of root 1  */
     double x1i;  /* imaginary part of root 1 */
     double x2r;
     double x2i;
     printf("Enter value of cofficient a: ");
     scanf("%lf", &a);
     printf("\nValue of a: %lf \n", a);
     printf("Enter value of cofficient b: ");
     scanf("%lf", &b);
     printf("\nValue of a: %lf \n", b);
     printf("Enter value of cofficient c: ");
     scanf("%lf", &c);
     printf("\nValue of a: %lf \n", c);

     disc = pow(b, 2) - 4.0 * a * c;
     printf("disc: %lf \n", disc);
     if (disc < 0.0)  {
          /* complex roots */
          disc = -disc;
          x1r = -b/(2.0 * a);
          x1i = sqrt(disc)/(2.0 * a);
          x2r = x1r;
          x2i = -x1i;
          printf ("Complex roots \n");
          printf ("x1r: %lf x1i: %lf \n", x1r, x1i);
          printf ("x2r: %lf x2i: %lf \n", x2r, x2i);
     }
     else {
          /* real roots */
          x1r = (-b + sqrt(disc))/(2.0 * a);
          x2r = (-b - sqrt(disc))/(2.0 * a);
          printf("Real roots: \n");
          printf("x1: %lf x2: %lf \n", x1r, x2r);
     }
     return 0;      /* execution terminates ok */
}
```

```
C:\lang_SPSCL >spscl solquad.scl
SPSCL  v 1.1 File: solquad.scl Mon Feb 03 17:22:43 2022
File: solquad.scl no syntax errors, lines processed: 48

C:\lang_SPSCL >gcc solquad.c -o solquad.exe -lm
C:\lang_SPSCL >solquad
Enter value of cofficient a: 1.25
Value of a:   1.250000
Enter value of cofficient b: 2.5
Value of a:   2.500000
Enter value of cofficient c: 2.85
Value of a:   2.850000
discreminant:   0.000013
Real roots:
x1:   -0.998539  x2:   -1.001461
```

The following shell commands compile, link, and execute the C program `solquadra.c`. The program prompts the user for the three values of the coefficients, calculates the roots, then displays the value of the roots.

```
$ gcc -Wall solquad.c -lm
$ ./a.out
Enter value of cofficient a: 1.5

Value of a: 1.500000
Enter value of cofficient b: 2.25

Value of a: 2.250000
Enter value of cofficient c: 5.5

Value of a: 5.500000
disc: -27.937500
Complex roots
x1r: -0.750000 x1i: 1.761865
x2r: -0.750000 x2i: -1.761865
```

5.9 Multi-Level Selection

The multi-level selection involves more than two alternatives. The general **if** statement with multiple paths is used to implement this structure.

5.9.1 General Multi-Path Selection

In SPSCL, the **elseif** clause is used to expand the number of alternatives. The **if** statement with n alternative paths has the general form:

```
if ⟨ condition ⟩
   then
      ⟨ block1 ⟩
   elseif ⟨ condition2 ⟩
   then
      ⟨ block2 ⟩
   elseif ⟨ condition3 ⟩
   then
      ⟨ block3 ⟩
   else
      ⟨ blockn ⟩
endif
```

Each block of statements is executed when that particular path of logic is selected. This selection depends on the conditions in the multiple-path **if** statement that are evaluated from top to bottom until one of the conditions evaluates to true. The following example shows the **if** statement with several paths.

```
if y > 15.50
then increment x
elseif y > 4.5
then add 7.85 to x
elseif y > 3.85
then set x = y*3.25
elseif y > 2.98
then set x = y + z*454.7
else
    set x = y
endif
```

In C, this example is written as the follows:

```
if (y > 15.50)
    x ++;
else if  (y > 4.5)
   x = x + 7.85;
else if  (y > 3.85)
   x = y * 3.25;
else if  (y > 2.98)
   x = y + z * 454.7;
else
   x = y;
```

5.9.2 Case Structure

The case structure is a simplified version of the selection structure with multiple paths. In SPSCL , the **case** statement evaluates the value of a single variable or simple expression of a number or a text string and selects the appropriate path. The case statement also supports a block of multiple statements instead of a single statement in one or more of the selection options. The general case

structure is:

```
case ⟨ select_var ⟩ of
   value  var_value1 :  ⟨ block1 ⟩
   value  var_value2 :  ⟨ block2 ⟩
     ...
   value  var_valuen :  ⟨ blockn ⟩
endcase
```

In the following example, the pressure status of a furnace is monitored and the pressure status is stored in variable *press_stat*. The following case statement first evaluates the pressure status and then displays an appropriate text string and the value of variable press_stat.

```
case press_stat of
   value 'D': display "Very dangerous", press_stat
   value 'A': display "Alert", press_stat
   value 'W': display "Warning", press_stat
   value 'N': display "Normal" , press_stat
   value 'B': display "Below normal", press_stat
endcase
```

The default option of the **case** statement can also be used by writing the keywords **default** or **otherwise** in the last case of the selector variable. For example, the previous example can be enhanced by including the default option in the case statement:

```
case press_stat of
   value 'D': display "Very dangerous", press_stat
   value 'A': display "Alert", press_stat
   value 'W': display "Warning", press_stat
   value 'N': display "Normal" , press_stat
   value 'B': display "Below normal", press_stat
   otherwise
       display "Pressure rising", press_stat
endcase
```

In C, this structure is implemented with the **switch** statement. The general form of this statement is:

```
switch { ⟨ select_var ⟩
   case  var_value1 :    ⟨ block1 ⟩
   case  var_value2 :  ⟨ block2 ⟩
     ...
   case  var_valuen :  ⟨ blockn ⟩
   default
         ⟨ blockn ⟩
}
```

In the previous example of the pressure status of a furnace in variable *press_stat*. The following switch statement first evaluates the pressure status and then displays an appropriate text string and variable *press_stat*.

```
switch press_stat {
   case 'D' :
       printf ("Very dangerous %c \n", press_stat);
   case 'A' :
       printf("Alert %c \n", press_stat);
   case 'W' :
       printf ("Warning %c \n", press_stat);
   case 'N' :
       printf ("Normal %c \n", press_stat);
   case 'B' :
       printf ("Below normal %c \n", press_stat);
   default :
       printf ("Pressure rising %c \n", press_stat);
}
```

5.10 Time and Space in Algorithms

Using algorithms often involves selecting an appropriate one with respect to the time and space requirements of the algorithm; this reflects its efficiency. Technically, these requirements are analyzed and are known as space and time complexity.

As mentioned in previous sections of this chapter, an algorithm is a finite sequence of well-defined steps for reaching a solution to a computational problem. When implemented in a program and run, the *space complexity* of the algorithm measures (directly or indirectly) the amount of memory space taken with respect to the length of the input.

The *time complexity* of an algorithm indirectly refers to the time it takes to run. More specifically, the time complexity is the number of computations the algorithm executes when running in relation to the size of the input. It is the variation (growth) in the number of computations when the size of the input grows. For example, in a search algorithm for finding a given value in a list, the number of comparisons grow as the size of the list grows. This relation is denoted as $O()$ and indicates the order of growth with respect to the length of the input, N. It is also known as *Big O Notation*. There are different types of time complexities commonly used and these are:

- Constant time, denoted $O(1)$. The number of computations does not depend on the problem size.
- Linear time, denoted $O(N)$. In linear search, the number of comparisons grow linearly as the size of the list grows.
- Logarithmic time, denoted $O(\log N)$. With binary search, the growth rate of number of comparisons as the number of elements (N) in the list is $\log N$.
- Quadratic time, denoted $O(N^2)$. Algorithms with nested loops exhibits this time complexity order where for one loop takes O(N) and if the function involves a loop within a loop, then it is $O(N) * O(N) = O(N^2)$ order.
- Cubic time, denoted $O(N^3)$
- Other time complexities can be used, such as: exponential, quasilinear, factorial time, and more.

Summary

An algorithm is a precise, detailed, and complete description of a solution to a problem. The notations to describe algorithms are flowcharts and pseudo-code. Flowcharts are a visual representation of the execution flow of the various instructions in the algorithm. Pseudo-code is an English-like notation to describe algorithms.

The design structures are sequence, selection, repetition, and input-output. These algorithmic structures are used to specify and describe any algorithm.

The selection structure is also known as alternation. It evaluates a condition then follows one of two (or more) paths. The two general selection statements **if** and **case** statements are explained in pseudo-code and in SPSCL . The first one is applied when there are two or more possible paths in the algorithm, depending on how the condition evaluates. The case statement is applied when the value of a single variable or expression is evaluated, and there are multiple possible values.

The condition in the **if** statement consists of a conditional expression, which evaluates to a truth-value (true or false). Relational operators and logical operators are used to form more complex conditional expressions. As seen in this and previous chapters, the SPSCL language is very similar to pseudo-code, therefore writing an algorithm in pseudo-code can actually be avoided.

Key Terms

algorithm	flowcharts	pseudo-code	variables
constants	declarations	structure	sequence
action step	selection	repetition	input/output
statements	data type	identifier	design
alternation	condition	if statement	logical expression
case statement	relational operator	logical operator	truth-value
then	else	endif	endcase
otherwise	elseif	end	switch statement

Exercises

Exercise 5.1 Develop an SPSCL program that computes the conversion from gallons to liters and from liters to gallons. Include a flowchart, pseudo-code design and a complete implementation in C. The user inputs the string: “gallons” or “liters”; the model then computes the corresponding conversion.

Exercise 5.2 Develop an SPSCL program to calculate the total amount to pay for movie rental. Include a flowchart, pseudo-code design and a complete implementation in C. The movie rental store charges $3.50 per day for every DVD movie. For every additional period of 24 hours, the customer must pay $0.75.

Exercise 5.3 Develop an SPSCL program that finds and displays the largest of several numbers, which are read from the input device. Include a flowchart, pseudo-code design and a complete

implementation in C.

Exercise 5.4 Develop an SPSCL program that finds and displays the smallest of several numbers, which are read from the input device. Include a flowchart, pseudo-code design and a complete implementation in C.

Exercise 5.5 Develop an SPSCL program program that computes the gross and net pay of the employees. The input quantities are employee name, hourly rate, number of hours, percentage of tax (use 14.5%). The tax bracket is $ 115.00. When the number of hours is greater than 40, the (overtime) hourly rate is 40% higher. Include a flowchart, pseudo-code design and a complete implementation in C.

Exercise 5.6 Develop an SPSCL program that computes the distance between two points. A point is defined by a pair of values (x, y). Include a flowchart, pseudo-code design and a complete implementation in C. The distance, d, between two points, $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$ is defined by:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Exercise 5.7 Develop an SPSCL program that computes the fare in a ferry transport for passengers with motor vehicles. Include a flowchart, pseudo-code design and a complete implementation in C. Passengers pay an extra fare based on the vehicle's weight. Use the following data: vehicles with weight up to 780 lb pay $80.00, up to 1100 lb pay $127.50, and up to 2200 lb pay $210.50.

Exercise 5.8 Develop an SPSCL program that computes the average of student grades. The input data are the four letter grades for various work submitted by the students. Include a flowchart, pseudo-code design and a complete implementation in the C programming language.

6. The Selection Control Structure

6.1 Introduction

To fully describe algorithms, four design structures are used: sequence, selection, repetition, and input/output. This chapter explains two selection statements in the SPSCL language. The statements are the **if** and the **case** statements.

Conditions are used in the selection statements. Conditions are expressions of type **Boolean** that evaluate to a truth-value (true or false). Simple conditions are formed with relational operators for comparing two data items. Compound conditions are formed by joining two or more simple conditions with the *logical* operators.

For examples of applying the selection statements, two problems are discussed: the solution of a quadratic equation and the solution to of the salary problem.

6.2 Selection Structure

The selection design structure is also known as alternation, because alternate paths are considered based on a condition.

6.2.1 Flowchart of Selection Structure

The selection structure provides the capability for decision-making. Because the selection design structure is better understood using a flowchart, Figure 6.1 is repeated here. The figure shows two possible paths for the execution flow. The condition is evaluated, and one of the paths is selected. If the condition is true, then the left path is selected and *Action step1* is performed. If the condition is false, the other path is selected and *Action step2* is performed.

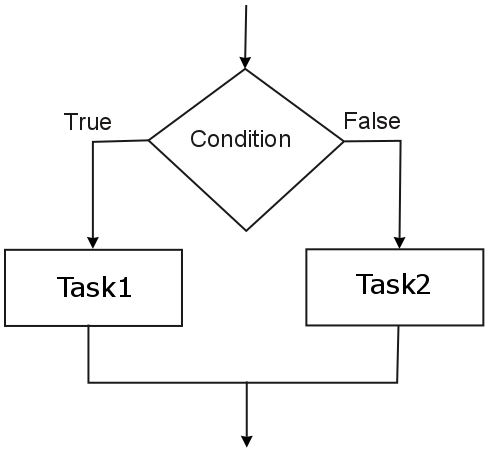

Figure 6.1: Flowchart of the Selection structure

6.2.2 IF Statement

The selection structure is written with an **if** statement, also known as an if-then-else statement. This statement includes three sections: the condition, the then-section, and the else-section. The else-section is optional. Several keywords are used in this statement: **if**, **then**, **else**, and **endif**. The general form of the **if** statement is:

if ⟨ *condition* ⟩

then

⟨ *first sequence of statements* ⟩

else

⟨ *alternate sequence of statements* ⟩

endif

The **if** statement that corresponds to the selection structure in Figure 6.1 is:

```
if condition is true
   then
      perform instructions in Action step1
   else
      perform instructions in Action step2
endif
```

When the **if** statement executes, the condition is evaluated and only one of the two alternatives will be carried out: the one with the statements in *Task1* or the one with the statements in *Task2*.

6.2.3 Boolean Expressions

Boolean expressions are written to form conditions. A condition consists of an expression that evaluates to a truth-value, **true** or **false**.

A simple Boolean expression can be formed with the values of two data items and a relational operator. The following list of relational operators can appear in a Boolean expression:

- Less than, $<$
- Less or equal to, $<=$
- Greater than, $>$
- Greater or equal to, $>=$
- Equal, ==
- Not equal, !=

Examples of simple conditions are:

```
height > 23.75

a <= b

p == q
```

SPSCL supports not only conditions with the syntax in the previous example, but also provides additional keywords can be used for the relational operators. For example, the previous Boolean expressions can also be written as:

```
height  greater than  23.75

a  less or equal to  b

p  equal to  q
```

6.2.4 Example of Selection

In this example, the condition to be evaluated is: $p \leq q$, and a decision is taken on how to update variable p. Figure 6.2 shows the flowchart for part of the algorithm that includes this selection structure. The SPSCL code for this example is:

```
if p <= q
then
    set p = q * x + 24.5
else
    set p = q
    set q = 12.75
```

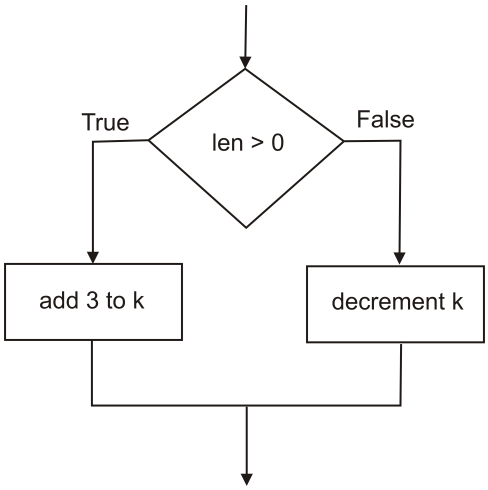

Figure 6.2: Example of selection structure

endif

6.3 Example Program

The following problem involves developing a solution of a quadratic equation, which is a simple mathematical model of a second-degree equation. The solution to the quadratic equation in general involves complex numbers.

6.3.1 Analysis of the Problem

The goal of the solution to the problem is to compute the two roots of the equation. The mathematical model is defined in the general form of the quadratic equation (second-degree equation):

$$ax^2 + bx + c = 0$$

The given data for this problem are the values of the coefficients of the quadratic equation: a, b, and c. Because this mathematical model is a second degree equation, the solution consists of the value of two roots: x_1 and x_2.

6.3.2 Algorithm for General Solution

The general solution gives the value of the two roots of the quadratic equation, when the value of the coefficient a is not zero ($a \neq 0$). The values of the two roots are:

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

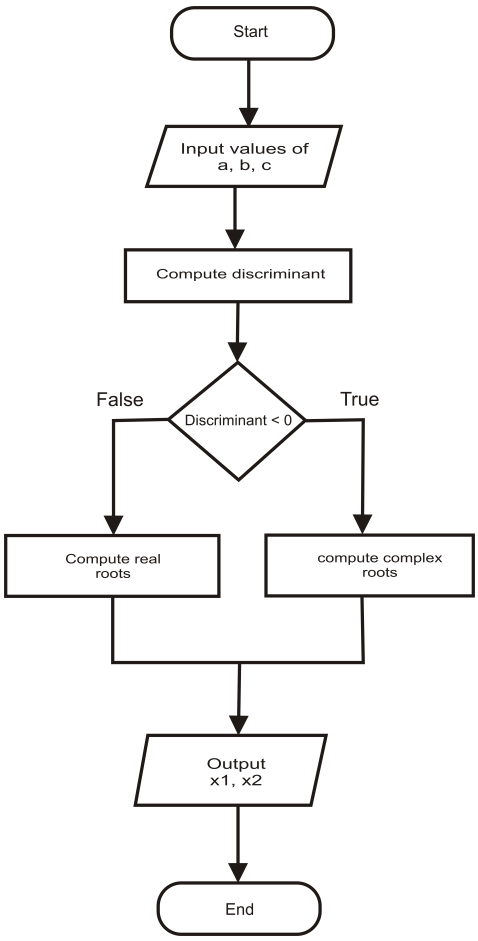

Figure 6.3: High-level flowchart for solving a quadratic equation.

The expression inside the square root, $b^2 - 4ac$, is known as the *discriminant* and if it is negative, the solution will involve complex roots. Figure 6.3 shows the flowchart for the general solution and the following listing is a high-level pseudo-code version of the algorithm.

```
Input the values of coefficients  a,  b, and  c
Calculate value of the discriminant
if the value of the discriminant is less than zero
     then calculate the two complex roots
     else calculate the two real roots
endif
display the value of the roots
```

6.3.3 Detailed Algorithm

The algorithm in pseudo-code notation for the solution of the quadratic equation is:

```
read the value of a from the input device
read the value of b from the input device
read the value of c from the input device
compute the discriminant, disc = b^2 - 4ac
if discriminant less than zero
then
      // roots are complex
      compute x1 = (-b + sqrt(disc))/2a
      compute x2 = (-b - sqrt(disc))/2a
else
      // roots are real
      compute x1 = (-b + sqrt(disc))/2a
      compute x2 = (-b - sqrt(disc))/2a
endif
display values of the roots:  x1 and x2
```

Listing 6.1 shows the SPSCL program that implements the algorithm for the solution of the quadratic equation, which is stored in the file `solquad.scl`.

Listing 6.1 SPSCL program that computes the roots of a quadratic equation.

```
/*
 Program     : solquad.scl
 Author      : Jose M Garrido, Nov 21 2021.
 Description : Compute the roots of a quadratic equation.
    Read the value of the coefficients: a, b, and c from
    the input console, compute and display value of roots.
*/
import "scl.h"
implementations
function main is
   variables
     define a of type double       // coefficient a
     define b of type double
     define c of type double
     define disc of type double  // discriminant
```

```
     define x1r of type double   // real part of root 1
     define x1i of type double   // imaginary part of root 1
     define x2r of type double
     define x2i of type double
begin
     input "Enter value of cofficient a: ", a
     display "Value of a: ", a
     input "Enter value of cofficient b: ", b
     display "Value of a: ", b
     input "Enter value of cofficient c: ", c
     display "Value of a: ", c
     set disc = b^2 - 4.0 * a * c
     display "discreminant: ", disc
     if (disc < 0.0)  then
          // complex roots
          set disc = -disc
          set x1r = -b/(2.0 * a)
          set x1i = sqrt(disc)/(2.0 * a)
          set x2r = x1r
          set x2i = -x1i
          display "Complex roots "
          display "x1r: ", x1r, " x1i: ",  x1i
          display "x2r: ", x2r, " x2i: ", x2i
     else
          // real roots
          set x1r = (-b + sqrt(disc))/(2.0 * a)
          set x2r = (-b - sqrt(disc))/(2.0 * a)
          display "Real roots:"
          display "x1: ", x1r, " x2: ", x2r
     endif
     exit      // execution terminates ok
endfun main
```

The following shell commands translate the SPSCL program, compile, link, and execute the corresponding C program `solquadra.c`. The program prompts the user for the three values of the coefficients, calculates the roots, then displays the value of the roots.

```
$ spscl solquad.scl
$ gcc -Wall solquad.c -lm
$ ./a.out
Enter value of cofficient a: 1.5

Value of a: 1.500000
Enter value of cofficient b: 2.25

Value of a: 2.250000
Enter value of cofficient c: 5.5

Value of a: 5.500000
disc: -27.937500
Complex roots
x1r: -0.750000 x1i: 1.761865
x2r: -0.750000 x2i: -1.761865
```

6.4 If Statement with Multiple Paths

The **if** statement with multiple paths is used to implement decisions involving more than two alternatives. The additional **elseif** clause is used to expand the number of alternatives. The general form of the *if* statement with *k* alternatives is:

if ⟨ *condition* ⟩

then

⟨ *sequence1 of statements* ⟩

elseif ⟨ *condition2* ⟩

then

⟨ *sequence2 of statements* ⟩

elseif ⟨ *condition3* ⟩

then

⟨ *sequence3 of statements* ⟩

. . .

else

⟨ *sequencek of statements* ⟩

endif

The conditions in a multiple-path **if** statement are evaluated from top to bottom until one of the conditions evaluates to true. The following example applies the **if** statement with four paths.

```
if height > 6.30
  then increment group1
elseif height > 6.15
  then increment group2
elseif height > 5.85
  then increment group3
elseif height > 5.60
  then increment group4
else
  increment group5
endif
```

6.5 Using Logical Operators

With the logical operators, complex Boolean expressions can be constructed. The logical operators are: **and**, **or**, and **not**. These logical operators help to construct complex (or compound) conditions from simpler conditions. The general form of a complex conditions using the **or** operator and two simple conditions, *cond1* and *cond2*, is:

`cond1` **and** `cond2`

`cond1` **or** `cond2`

not `cond1`

The following example includes the **and** logical operator:

if `x >= y` **and** `p == q`

then

⟨ *statements_1* ⟩

else

⟨ *statements_2* ⟩

endif

The condition can also be written in the following manner:

if `x` **greater or equal to** `y` **and** `p` **is equal to** `q`

...

The following example applies the **not** operator:

not `(p > q)`

6.6 The Case Statement

The case structure is a simplified version of the selection structure with multiple paths. The **case** statement evaluates the value of a single variable or simple expression of type **integer** or of type **character** and selects the appropriate path. The case statement also supports compound statements, that is, multiple statements instead of a single statement in one or more of the selection options. The general form of this statement is:

case ⟨ *selector_variable* ⟩ **is**

  when *variable_value* : ⟨ *statements* ⟩

  ...

endcase

In the following example, the temperature status of a boiler is monitored and the temperature status is stored in variable *temp_status*. The following case statement first evaluates the temperature status in variable *temp_status* and then assigns an appropriate string literal to variable *mess*, finally this variable is displayed on the console.

The example assumes that the variables involved have an appropriate declaration, such as:

```
variables
   define temp_status of type character
   define mess of type string
  . . .
  case temp_status is
      when 'E': set mess = "Extremely dangerous"
      when 'D': set mess = "Dangerous"
      when 'H': set mess = "High"
      when 'N': set mess = "Normal"
      when 'B': set mess = "Below normal"
      default :  set mess = "Temp rising"
  endcase . . .
  display "Temperature status: ", mess
```

The default option of the **case** statement can also be used by writing the keywords **default** or **otherwise** in the last case of the selector variable.

6.7 Summary

The selection design structure evaluates a condition then takes one of two (or more) paths. This structure is also known as alternation. The two selection statements explained are the **if** and **case** statements. The first one is applied when there are two or more possible paths in the algorithm, depending on how the condition evaluates. The case statement is applied when the value of a single variable or expression is evaluated, and there are multiple possible values.

The condition in the **if** statement consists of a Boolean expression, which evaluates to a truth-value (true or false). Relational operators and logical operators are used to form Boolean expressions.

Key Terms

selection	alternation	condition	if statement
case statement	relational operator	logical operator	truth-value
then	else	endif	endcase
otherwise	elseif	end	multi-path

Exercises

Exercise 6.1 Develop a program that computes the conversion from gallons to liters and from liters to gallons. Include a flowchart, design and a complete implementation in SPSCL. The user inputs the string: "gallons" or "liters"; the model then computes the corresponding conversion.

Exercise 6.2 Develop a program to calculate the total amount to pay for movie rental. Include a flowchart, design and a complete implementation in SPSCL. The movie rental store charges $3.50 per day for every DVD movie. For every additional period of 24 hours, the customer must pay $0.75.

Exercise 6.3 Develop a program that finds and displays the largest of several numbers, which are read from the input device. Include a flowchart, pseudo-code design and a complete implementation in SPSCL.

Exercise 6.4 Develop a program that finds and displays the smallest of several numbers, which are read from the input device. Include a flowchart, pseudo-code design and a complete implementation in SPSCL.

Exercise 6.5 Develop a program that computes the gross and net pay of the employees. The input quantities are employee name, hourly rate, number of hours, percentage of tax (use 14.5%). The tax bracket is $ 115.00. When the number of hours is greater than 40, the (overtime) hourly rate is 40% higher. Include a flowchart, pseudo-code design and a complete implementation in SPSCL.

Exercise 6.6 Develop a program that computes the distance between two points. A point is defined by a pair of values (x, y). Include a flowchart, pseudo-code design and a complete implementation in SPSCL. The distance, d, between two points, $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$ is defined by:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Exercise 6.7 Develop a program that computes the fare in a ferry transport for passengers with motor vehicles. Include a flowchart, pseudo-code design and a complete implementation in SPSCL. Passengers pay an extra fare based on the vehicle's weight. Use the following data: vehicles with weight up to 780 lb pay $80.00, up to 1100 lb pay $127.50, and up to 2200 lb pay $210.50.

Exercise 6.8 Develop a program that computes the average of student grades. The input data are the four letter grades for various work submitted by the students. Include a flowchart, pseudo-code design and a complete implementation in the SPSCL programming language.

7. Repetition

7.1 Introduction

This chapter presents the repetition structure and specifying, describing, and implementing algorithms in developing computational models. This structure and the corresponding statements are discussed with flowcharts, pseudo-code, and in the C programming language. The repetition structure specifies that a block of statements be executed repeatedly based on a given condition. Basically, the statements in the process block of code are executed several times, so this structure is often called a *loop* structure. An algorithm that includes repetition has three major parts in its form:

1. the initial conditions
2. the steps that are to be repeated
3. the final results.

There are three general forms of the repetition structure: the **while** loop, the *repeat-until* loop, and the **for** loop. The first form of the repetition structure, the *while* construct, is the most flexible. The other two forms of the repetition structure can be expressed with the **while** construct.

7.2 Repetition with the While Loop

The *while* loop consists of a conditional expression and block of statements. This construct evaluates the condition before the process block of statements is executed. If the condition is true, the statements in the block are executed. This repeats while the condition evaluates to true; when the condition evaluates to false, the loop terminates.

7.2.1 While-Loop Flowchart

A flowchart with the *while* loop structure is shown in Figure 7.1. The process block consists of a sequence of actions.

The actions in the process block are performed while the condition is true. After the actions in the process block are performed, the condition is again evaluated, and the actions are again performed if the condition is still true; otherwise, the loop terminates.

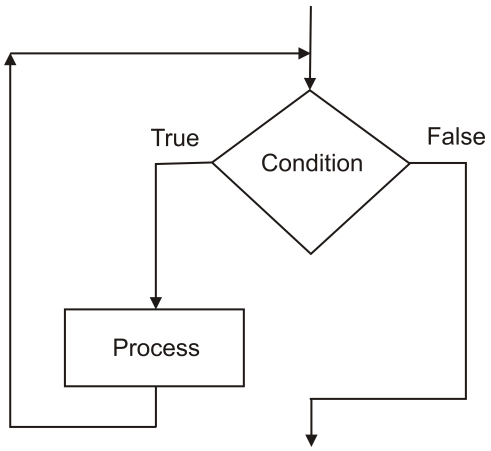

Figure 7.1: A flowchart with a while-loop.

The condition is tested first, and then the process block is performed. If this condition is initially false, the actions in the block are not performed.

The number of times that the loop is performed is normally a finite number. A well-defined loop will eventually terminate, unless it has been specified as a non-terminating loop. The condition is also known as the *loop condition*, and it determines when the loop terminates. A non-terminating loop is defined in special cases and will repeat the actions forever.

7.2.2 While Structure in SPSCL

The form of the **while** statement includes the condition, the actions in the process block written as statements, and the keywords **while**, **do**, and **endwhile**. The block of statements is placed after the **do** keyword and before the **endwhile** keyword. The following lines of pseudo-code show the general form of the while-loop statement that is shown in the flowchart of Figure 7.1.

```
while ⟨ condition ⟩ do
    ⟨ block of statements ⟩
endwhile
```

The following SPSCL example has a *while* statement with a condition that checks the value of variable *j*. The block of statements is performed repeatedly while the condition `j <= MAX_NUM` is true.

```
constants
  define MAX_NUM = 15 of type integer   // maximum number
                                        // of times through the loop
variables
  define j of type integer    // loop counter
  define sum of type float
  define x of type float
  define y of type float
  ...
begin
  set j = 1   // initial value
  set sum = 0
  set x = 10.25
  while j <= MAX_NUM do
      set sum = sum + 12.5
      set y = x * 2.5
      add 3 to j
  endwhile
  display "Value of sum: ", sum
  ...
```

7.2.3 While Loop in the C Language

The following lines of code show the general form of the while-loop statement in C; it is similar to the pseudo-code statement and follows the loop definition shown in the flowchart of Figure 7.1.

while ⟨ *condition* ⟩
  ⟨ *block of statements* ⟩
end

The previous example has a *while* statement with a condition that checks the value of variable *j*. The block of statements is enclosed in braces and repeats while the condition `j <= MAX_NUM` is true. The following lines of code show the C implementation.

```
while ( j <= MAX_NUM) {
     sum = sum + 12.5;
     y = x * 2.5;
     j = j + 3;
}
moutput('Value of sum: ', sum)
moutput('Value of y: ', y)
```

7.2.4 Loop Counter

As mentioned previously, in the while-loop construct the condition is tested first and then the statements in the statement block are performed. If this condition is initially false, the statements are not performed.

The number of times that the loop is performed is normally a finite integer value. For this, the condition will eventually be evaluated to false, that is, the loop will terminate. This condition is often known as the *loop condition*, and it determines when the loop terminates. Only in some very

special cases, the programmer can decide to write an infinite loop; this will repeat the statements in the repeat loop forever.

A *counter* variable stores the number of times (also known as iterations) that the loop executes. The counter variable is incremented every time the statements in the loop are performed. The variable must be initialized to a given value, typically to 0 or 1.

In the the following listing, there is a *while* statement with a counter variable with name *loop_counter*. This counter variable is used to control the number of times the block statement is performed. The counter variable is initially set to 1, and is incremented every time through the loop.

```
set Max_Num = 25      // maximum number of times to execute
set loop_counter = 1   // initial value of counter
while loop_counter < Max_Num do
       increment loop_counter
       display "Value of counter: ", loop_counter
endwhile
```

The first time the statements in the while- block are performed, the loop counter variable *loop_counter* has a value equal to 1. The second time through the loop, variable *loop_counter* has a value equal to 2. The third time through the loop, it has a value of 3, and so on. Eventually, the counter variable will have a value equal to the value of *Max_Num* and the loop terminates.

7.2.5 Accumulator Variables

An *accumulator* variable stores partial results of repeated calculations. The initial value of an accumulator variable is normally set to zero.

For example, an algorithm that calculates the summation of numbers from input, includes an accumulator variable. The following pseudo-code statement accumulates the values of *innumber* in variable *total* and it is included in the while loop:

```
set total = 0.0
while j < Max_num
    add innumber to total
endwhile
display "Total accumulated: ", total
```

After the *endwhile* statement, the value of the accumulator variable *total* is displayed using the pseudo-code statement to print the string “Total accumulated” and the value of *total*. In programming, each counter and accumulator variable serves a specific purpose. These variables should be well documented.

7.2.6 Summation of Input Numbers

The following simple problem applies the concepts and implementation of while loop and accumulator variable. The problem computes the summation of numeric values read from the main input device. Computing the summation should proceed while the input values are greater than zero.

The SPSCL code that implements the algorithm uses an input variable, an accumulator variable, a loop counter variable, and a conditional expression that evaluates whether the input value is greater than zero.

```
function main is
   variables
       define innumber of type double
       define msum of type double
   set sum = 0.0       // initialize accumulator variable
   display "Enter value of input number: "
   read innumer
   display " \nValue of input number: ", innumber
   while innumber > 0.0  do
          add innumber to msum
          display " \nEnter value of input number: "
          read innumber
          display " \nValue of input number: ", innumber
   endwhile
   display "Value of summation: ", msum
endfun main
```

Listing 7.1 shows the C program that implements the summation problem. The program is stored in file `summ.c`.

Listing 7.1 Source C program for summation.

```
  #include <stdio.h>
  #include <math.h>
  int main(void) {
     double innumber;   /* number to read and sum */
     double msum = 0.0;
     printf("Enter value of input number: ");
     scanf("%lf", &innumber);
     printf("\nValue of input number: %lf \n", innumber);
     while ( innumber > 0.0) {
         msum = msum + innumber;
         printf("Enter value of input number: ");
         scanf("%lf", &innumber);
         printf("\nValue of input number: %lf \n", innumber);
    }
    printf("Value of summation: %lf \n", msum);
    return 0;      /* execution terminates ok */
 }
```

The following output listing shows the shell commands used to compile and link the source file `summ.scl`, and execute the object file `summ.out`.

```
$ ./spscl summ.scl
$ gcc -Wall summ.c -o summ.out -lm
$ ./summ.out
Enter value of input number: 12.5

Value of input number: 12.500000
Enter value of input number: 10.0

Value of input number: 10.000000
Enter value of input number: 5.5

Value of input number: 5.500000
Enter value of input number: 4.25

Value of input number: 4.250000
Enter value of input number: 133.75

Value of input number: 133.750000
Enter value of input number: 0.0

Value of input number: 0.000000
Value of summation: 166.000000
```

7.3 Repeat-Until Loop

The *repeat-until* loop is a control flow structure that allows actions to be executed repeatedly based on a given condition. This construct consists of a process block of actions and the condition.

The actions within the process block are executed first, and then the condition is evaluated. If the condition is not true the actions within the process block are executed again. This repeats until the condition becomes true.

Repeat-until structures check the condition after the block is executed; this is an important difference from the while loop, which tests the condition before the actions within the block are executed. Figure 7.2 shows the flowchart for the repeat-until structure.

The SPSCL statement of the repeat-until structure corresponds directly with the flowchart in Figure 7.2 and uses the keywords *repeat*, *until*, and *endrepeat*. The following portion of code shows the general form of the repeat-until statement.

repeat
⟨ *statements in block* ⟩
until ⟨ *condition* ⟩
endrepeat

The following lines of code shows the pseudo-code listing of a repeat-until statement for the previously discussed problem.

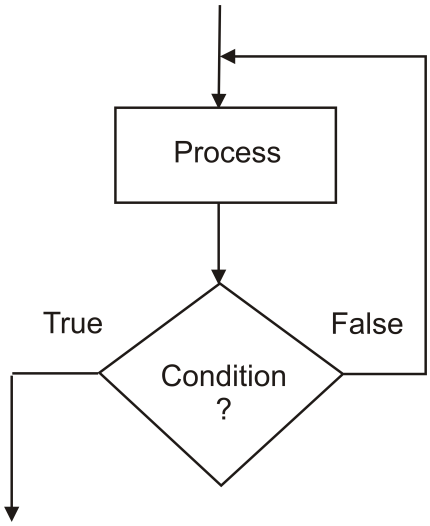

Figure 7.2: A flowchart with a repeat-until structure.

```
set innumber = 1.0  // dummy initial value
set l_counter = 0
set sum = 0.0       // accumulator variable
repeat
       display "Type number: "
       read innumer
       if innumber > 0.0
       then
           add innumber to sum
           increment l_counter
           display "Value of counter: ", l_counter
       endif
until innumber <= 0.0
endrepeat
display "Value of sum: ", sum
```

In C, the repeat-until loop has the general form:

```
do
{
      [ block of statements ]
}
while (condition);
```

7.4 For Loop Structure

The *for* loop structure explicitly uses a loop counter; the initial value and the final value of the loop counter are specified. The *for* loop is most useful when the number of times that the loop is carried

out is known in advance. In pseudo-code, the *for* statement includes the keywords: *for*, *to*, *downto*, *do*, and *endfor* and has the following general form:

```
for ⟨ counter ⟩ = ⟨ initial_val ⟩ to/downto ⟨ conditionl ⟩
[ with ⟨ expression ⟩ ]
do
      Block of statements
endfor
```

On every iteration, the loop counter is automatically incremented. The last time through the loop, the loop counter reaches the condition and the loop terminates. The *for* loop is similar to the *while* loop in that the condition is evaluated before carrying out the operations in the repeat loop. The *with* clause is optional; in this case, the counter variable is implicitly incremented or decremented by 1.

The following portion of code uses a *for* statement for the repetition part of the summation problem.

```
for j = 1 to j <= num do
      set sum = sum + 12.5
      set y = x * 2.5
endfor
display "Value of sum: ", sum
display "Value of y: ", y
```

Variable *j* is the counter variable, which is automatically incremented and is used to control the number of times the statements in a block will be performed. In C, the previous portion of code is equivalent to the following:

```
for ( j = 1; j <= num; j++ ) {
       sum = sum + 12.5;
       y = x * 2.5;
}
printf ("Value of sum: %lf \n", sum);
printf ("Value of y: %lf \n", y)
```

A more complete *for* statement in SPSCL is:

```
for j = 3 to j < 100 with j = 4 do
    . . .
endfor
```

7.4.1 Summation Problem with a For-Loop

Using the for-loop construct of the repetition structure, the algorithm for the summation of input data can be defined in a relatively straightforward manner in SPSCL. The most significant difference from the previous design is that the number of data inputs from the input device is included at the beginning of the algorithm.

```
import "scl.h"
```

```
implementations

function main is
constants
    define MAXNUM of type integer = 15
variables
    define j of type integer  // loop counter
    define innumber of type float
    define sum of type float
begin
    set sum = 0.0       // initialize accumulator variable
    for j = 1 to j <= MAXNUM do
       // display "Type number: ", j
       input "Type number: ", innumber
       if innumber > 0.0
       then
           add innumber to sum
       endif
    endfor
    display "Value of sum: ", sum
endfun main
```

Listing 7.2 shows the C source program that implements the summation problem with a for-loop. The source programs are stored in files `summfor.scl` and `summfor.c`.

Listing 7.2 C source program for summation with for-loop.

```
//SPSCL v1.0 File: summfor.c, Fri Aug 05 14:06:05 2022

#include "scl.h"
 int  main() {
   const int  MAXNUM = 15;
   int  j;   // loop counter
   float  innumber;
   float  sum;
   sum = 0.0;     // initialize accumulator variable
   for (j = 1 ; j <= MAXNUM; j++) {
      printf("Type number: ");
      scanf(" %f", &innumber);
      if ( innumber > 0.0) {
         sum += innumber;
      } // endif
   } // endfor
   printf("Value of sum:   %f\n",sum);
 } // end function main
```

7.4.2 Factorial Problem

The *factorial* operation, denoted by the symbol !, can be defined in a general and informal manner as follows:

$$y! = y\,(y-1)\,(y-2)\,(y-3)\,\ldots\,1$$

For example, the factorial of 5 is:

$$5! = 5 \times 4 \times 3 \times 2 \times 1$$

Mathematical Specification of Factorial

A mathematical specification of the factorial function is as follows, for $y \geq 0$:

$$y! = \begin{cases} 1 & \text{when } y = 0 \\ y\,(y-1)! & \text{when } y > 0 \end{cases}$$

The base case in this definition is the value of 1 for the function if the argument has value zero, that is $0! = 1$. The general (recursive) case is $y! = y\,(y-1)!$, if the value of the argument is greater than zero. This function is not defined for negative values of the argument.

Computing Factorial

In the following program, the factorial function is named *fact* and has one argument: the value for which the factorial is to be computed. For example, $y!$ is implemented as function *fact(y)*.

Listing 7.3. shows an SPSCL program, `factp.scl` that includes function *main* that calls function *fact* to compute the factorial of a number and displays this value on the console.

Listing 7.3 SPSCL source program for computing factorial.

```
/*
 Program     : factp.scl
 Author      : Jose M Garrido, Nov 28 2021.
 Description : Compute the factorial of a number.
*/
import <stdio.h>
import <math.h>

forward declarations
 function fact return type integer
    parameters y of type integer

implementations

function main is
 variables
   define num of type integer
   define r of type integer
```

```
 begin
     displayn  "Enter number: "
 read num
     display "\n Value of input number: ", num
     set r = fact(num)
     display " Factorial is: ", r
     return 0    // execution terminates ok
endfun main
//
function fact return type integer
   parameters y of type integer is
variables
   define f of type integer
   define n of type integer
   define res of type integer
 begin
   set f = 1
   if y > 0 then
        for n = 1 to n <= y do
            set f = f * n
endfor
        set res = f
   elseif y equal 0 then
            set  res = 1
   else
         set res = -1
   endif
   return res
endfun fact
```

Note that this implementation will always return -1 for negative values of the argument. The following Linux shell commands compile, link, and execute the program for several values of the input number.

```
$ ./spscl factp.scl
$ gcc -Wall factp.c -o factp.out
$ ./factp.out
Enter number: 5

Value of input number: 5
 Factorial of 5 is: 120

$ ./factp.out
Enter number: 8

Value of input number: 8
 Factorial of 8 is: 40320
$ ./factp.out
Enter number: 10

Value of input number: 10
```

```
 Factorial of 10 is: 3628800

$ ./factp.out
Enter number: -5

Value of input number: -5
 Factorial of -5 is: -1
```

Summary

The repetition structure is used in algorithms in order to perform repeatedly a group of action steps (instructions) in the process block. There are three types of loop structures: *while* loop, *repeat-until*, and *for* loop. In the *while* construct, the loop condition is tested first, and then the block of statements is performed if the condition is true. The loop terminates when the condition is false.

In the *repeat-until* construct, the group of statements in the block is carried out first, and then the loop condition is tested. If the loop condition is true, the loop terminates; otherwise the statements in the block are performed again.

The number of times the statements in the block are carried out depends on the condition of the loop. In the *for* loop, the number of times to repeat execution is explicitly indicated by using the initial and final values of the loop counter. Accumulator variables are also very useful with algorithms and programs that include loops. Simple examples of SPSCL and C programs are shown to execute.

Key Terms

repetition	loop	while	loop condition
do	endrepeat	block	loop termination
loop counter	endwhile	accumulator	repeat-until
for	to	downto	endfor
end	iterations	summation	factorial

Exercises

Exercise 7.1 Develop a program that computes the maximum value from a set of input numbers. Use a while loop in the algorithm and implement in SPSCL .

Exercise 7.2 Develop a program that computes the maximum value from a set of input numbers. Use a for loop in the algorithm and implement in SPSCL .

Exercise 7.3 Develop a program that computes the maximum value from a set of input numbers. Use a repeat-until loop in the algorithm and implement in SPSCL .

Exercise 7.4 Develop a program that finds the minimum value from a set of input values. Use a while loop in the algorithm and implement in SPSCL .

Exercise 7.5 Develop a program that finds the minimum value from a set of input values. Use a for loop in the algorithm and implement in SPSCL .

Exercise 7.6 Develop a program that finds the minimum value from a set of input values. Use a repeat-until loop in the algorithm and implement in SPSCL .

Exercise 7.7 Develop a program that computes the average of a set of input values. Use a while loop in the algorithm and implement in SPSCL .

Exercise 7.8 Develop a program that computes the average of a set of input values. Use a for loop in the algorithm and implement in SPSCL .

Exercise 7.9 Develop a program that computes the average of a set of input values. Use a repeat-until loop in the algorithm and implement in SPSCL .

Exercise 7.10 Develop a program that computes the student group average, maximum, and minimum grade. The computational model uses the input grade for every student. Use a while loop in the algorithm and implement in SPSCL .

Exercise 7.11 Develop a program that computes the student group average, maximum, and minimum grade. The computational model uses the input grade for every student. Use a for loop in the algorithm and implement in SPSCL .

Exercise 7.12 Develop a program that reads rainfall data in inches for yearly quarters from the last five years. The computational model is to compute the average rainfall per quarter, the average rainfall per year, and the maximum rainfall per quarter and for each year. Implement in SPSCL .

Exercise 7.13 Develop a program that computes the total inventory value amount and total per item. The progtam must read item code, cost, and description for each item. The number of items to process is not known. Implement in SPSCL .

8. Arrays

8.1 Introduction

An *array* stores multiple values of data with a single name and most programming languages include facilities that support arrays. The individual values in the array are known as *elements*. Many programs manipulate arrays and each one can store a large number of values in a single collection.

To refer to an individual element of the array, an integer value or variable known as the array *index* is used after the name of the array. The value of the index represents the relative position of the element in the array. Figure 8.1 shows an array with 13 elements. This array is a data structure with 13 cells, and each cell contains an element value. In the SPSCL and C programming languages, the index values start with 0 (zero).

Figure 8.1: A simple array.

An array is basically a static data structure because once the array is declared, its size cannot be changed. The size of an array is the number of elements it can store. An array can also be created at execution time using pointers. A program carries out the following steps in order to use arrays:

1. Declare the array with appropriate name, size, and type
2. Assign initial values to the array elements
3. Access or manipulate the individual elements of the array

A simple array has one dimension and is considered a row vector or a column vector. To manipulate the elements of a one-dimensional array, a single index is required. A vector is a single-dimension array and by default it is a row vector.

A *matrix* is a two-dimensional array and the typical form of a matrix is an array with data items organized into rows and columns. In this array, two indexes are used: one index to indicate the column and one index to indicate the row. Figure 8.2 shows a matrix with 13 columns and two rows. Because this matrix has only two rows, the index values for the rows are 0 and 1. The index values for the columns are from 0 up to 12.

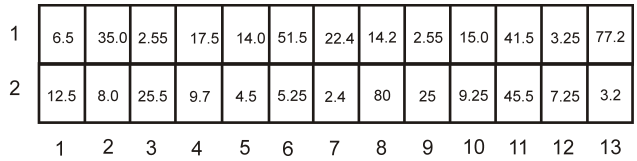

Figure 8.2: A two-dimensional array.

8.2 Declaring an Array

An array is declared with an appropriate name, type, and size and the optional initial values of the various elements. The size of the array is the number of elements it can store.

8.2.1 Declaring Arrays in SPSCL

In SPSCL , the general form for declaring an array is:

define ⟨ *array_name* ⟩ **array** [⟨ *size* ⟩] **of type** ⟨ *array_type* ⟩

The following SPSCL statement declares array *y*, with a capacity of 20 elements. Arrays of simple types must be declared in the `variables` section of data definitions.

```
variables
   define y array [20] of type float
```

The size of the array can be specified using a symbolic (identifier) constant. For example, given the constant `MAX_NUM` with value 20, the array *y* can be declared in SPSCL by the following statement:

```
constants
    define MAX_NUM of type integer = 20
variables
    define y array [MAX_NUM] of type float
```

8.2.2 Declaring Arrays in C

In the C programming language, an array is declared with its type, name, and size. For example, assume that the name of the one-dimensional array (vector) shown in Figure 8.1 is *precip*; it can be declared in C in the following manner:

```
double  rainf[13];
```

When the array is small, initial values can be given in the same statement that declares the array:

```
double  rainf[13] = {2.75 6.4 15.0 7.65 10.5 5.25 12.5 8.25 6.9
          3.5 15.5 5.5 9.4};
```

8.3 Operations on Arrays

A program can manipulate an array by accessing the individual elements of the array and performing some computation. Recall that an integer value known as the *index* is used with the name of the array to access an individual element of an array. In SPSCL and C, the range of index values starts with 0 and ends with the number equal to the size of the array minus 1.

8.3.1 Manipulating Array Elements in SPSCL

To refer to an individual element the name of the array is used followed by the index value in rectangular brackets. The following statement assigns a value of 8.25 to element 5 of array *z*.

```
set z[5] = 8.25
```

The index may be an integer constant, a symbolic constant (identifier), or an integer variable. The following statement assigns the value 4.5 to element 6 of array *y*, using variable *k* as the index.

```
constants
    define MAX_NUM = 25 of type integer
variables
   define k of type integer
   define y array [MAX_NUM] of type float
. . .
set k = 6
set y[k] = 4.5
```

8.3.2 Manipulating Elements of an Array in C

After declaring an array in C, the individual elements can be used in the same manner simple variables are used. To reference an individual element of an array, the appropriate index value is written in brackets. The following assignment statements may be used to initialize the elements of an array *rainf*:

```
rainf[0] = 2.75;
rainf[1] =  6.4;
rainf[2] =  15.0;
rainf[3] =  6.75;
rainf[4] =  4.5;
rainf[5] =  5.25;
rainf[6] =  2.4;
rainf[7] =  80.4;
rainf[8] =  25.96;
rainf[9] =  9.25;
rainf[10] =  45.5;
rainf[11] =  7.25;
rainf[12] =  8.5;
```

Accessing the elements of a vector in C is also known as indexing a vector. The values of the index of a one-dimensional array is an integer value that starts with zero, thus the first element of a simple array has an index value of zero. Accessing a particular element of an array uses the name of the array followed by the index value in brackets. The following assignment statement assigns a value of 14.5 to element 6 of array *y*.

```
y[5] = 14.5;
```

The index value used can be an integer constant, a symbolic constant (identifier), or an integer variable. The following lines of C code declare vector *height*, assign a value 4 to variable *k*, and assign the value 132.5 to element 5 (with index 4) of array *w*, using variable *k* as the index.

```
float w[5] = {12.5 8.25 7.0 21.35 6.55};
k = 4;
w[k] = 132.5;
```

In addition to accessing individual elements of an array in a simple assignment statement, more complex arithmetic operations are also supported in SPSCL and C. The following examples (in SPSCL and C) include a trigonometric operation with element 4 of vector *w*; the assignment sets the value to variable *x*. The C symbolic constant *M_PI* is the value for π.

```
set x = sin(0.04*w[4]/M_PI)
```

```
x = sin(0.04*w[4]/M_PI);
```

8.4 Arrays as Arguments of Functions

The arguments in a function call can be used as input values, output values, or input-output values by the function. In the SPSCL and C programming languages, there are two general techniques to pass arguments when calling a function:

- pass by value
- pass by reference

With *pass by value*, a copy of the value of the argument is passed to the corresponding parameter of the function. With this technique, the value of the argument is not changed by the function. Any attempt to change the value is only local to the function.

With *pass by reference*, an address of the argument is passed to the corresponding parameter. The function can change the value of this argument and when the function completes its execution, this argument will have a different value.

For input arguments, pass by value is used with simple variables. For output arguments, pass by reference is used. To pass simple variables by reference, *pointers* need to be used.

An array is always passed by reference and only the array name is used as the argument in the function call. The corresponding array parameter in the function definition does not include the & symbol before the parameter name. When the parameter is declared as a one-dimensional array, the size of the array need not be specified.

Because an array is passed by reference, the function being called can modify the value of the elements of the array. However, in the case that the function should not change the values of the elements of an array, the keyword **const** must be included before the parameter declaration in the function prototype and in the function definition. The following examples (in SPSCL and C) illustrate this notion in the function prototype for function *myfunct*.

```
function myfunct
   parameters x array[] of type double, xsize of type integer

void myfunct ( double x [], int xsize );
```

Function *linspace* assigns values to a single-dimension array. This function also assigns values that are equally spaced using the initial value, the final value, and the number of values in the vector. In the following example in SPSCL, a call to function *linspace* assigns vector *y* equally-spaced values starting with 3.15, a final value of 20.5, and a total of 20 values.

```
call linspace using y, 3.15, 20.5, 20
```

The following SPSCL and C source listings of function *linspace*; note that the first parameter is a one-dimensional array of type *double*.

```
description
     This function computes the num element values
     in the array starting from low to high.          */
function linspace
    parameters a array [] of type double, low of type double,
      high of type double, num of type integer  is
    variables
       define i of type integer    // used as index of the array
       define diff of type double
    begin
       set diff = (high - low)/(num - 1)
       set a[0] = low
```

```
        for i = 1 to i < num do
           set a[i] = a[i-1] + diff
        endfor
        return
endfun linspace
```

Function *linspace* source code in the C programming language:

```
/*  This function computes the num element values
     in the array starting from low to high.          */
void linspace (double  a[], double low, double high, int num) {
   int  i;        /* used as index of the array  */
   double diff;
   diff = (high - low)/(num-1);
   a[0] = low;
   for (i = 1; i < num; i++ ) {
       a[i] = a[i-1] + diff;
   }
   return;
}  // end linspace
```

8.5 Arithmetic Operations with Vectors

Simple arithmetic operations can be performed that relate a vector and a number, also known as a *scalar*. These operations are: addition, subtraction, multiplication, and division. These operations are applied to all elements of the vector. For example, the following for-loop statements, add the scalar 3.25 to all elements of vector *height*, which was previously declared with *N* elements.

```
for j = 0 to j < N do
     set hf[j] =  hf[j] + 3.25
endfor
```

Several array operations can be implemented in C. The simplest array operations are: addition, subtraction, multiplication, division, and exponentiation of two vectors. Element by element operations are usually inside a loop using similar notation for multiplication, division, and exponentiation. The following lines of code apply several simple array operations on two small row vectors.

```
  constants
     define N of type integer = 10
  variables
    define x array [6] of type double
    define y array [6] of type double
    define j of type integer
  begin
    set x[0] = 1.
    set x[1] = 6.25
    set x[3] = 3.15
    set y[0] = 4.35
    set y[1] = 5.7
```

```
 set y[2] = 6.85
 // vector addition
 for j = 0 to  j < N do
     set x[j] = x[j] + y[j]
     display " value of x: ", x[j]
     display " value of y: ", y[j]
 endfor
// vector subtraction  */
for j = 0 to j < N do
   set x[j] = x[j] - y[j]
   display " x: ", x[j]
   display " y: ", y[j]
endfor
// vector multiplication
for j = 0 to j < N do
   set x[j] = x[j] * y[j]
   display " x: ", x[j]
   display " y: ", y[j]
endfor
```

8.6 Multi-Dimensional Arrays

A multi-dimensional array is declared with one or more dimensions. A common name for a two-dimensional array is a matrix) and this is a mathematical structure with values arranged in columns and rows. Two index values are required, one for the rows and one for the columns. Figure 8.2 shows an example of a two-dimensional array.

8.6.1 Matrices

A matrix is a two-dimensional array of data values and is organized in rows and columns. The following array, Y, is a *two-dimensional array* organized as an $m \times n$ matrix; its elements are arranged in m rows and n columns.

$$Y = \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m1} & y_{m2} & \cdots & y_{mn} \end{bmatrix}$$

The first row of Y consists of elements: $y_{11}, y_{12}, \ldots, y_{1n}$. The second row consists of elements: $y_{21}, y_{22}, \ldots, y_{2n}$. The last row of Y consists of elements: $y_{m1}, y_{m2}, \ldots, y_{mn}$. In a similar manner, the elements of each column can be identified.

8.6.2 Matrix Basic Concepts

A matrix is defined by specifying the rows and columns of the array. An m by n matrix has m rows and n columns. A *square* matrix has the same number of rows and columns, n rows and n columns,

which is denoted as $n \times n$. The following example shows a 2×3 matrix, which has two rows and three columns:

$$\begin{bmatrix} 0.5000 & 2.3500 & 8.2500 \\ 1.8000 & 7.2300 & 4.4000 \end{bmatrix}$$

A matrix of dimension $m \times 1$ is known as a *column vector* and a matrix of dimension $1 \times n$ is known as a *row vector*. A vector is considered a special case of a matrix with one row or one column. A row vector of size n is typically a matrix with one row and n columns. A column vector of size m is a matrix with m rows and one column.

The elements of a matrix are denoted with the matrix name in lower-case and two indices, one corresponding to the row of the element and the other index corresponding to the column of the element. For matrix Y, the element at row i and column j is denoted by y_{ij} or by $y_{i,j}$.

The *main diagonal* of a matrix consists of those elements on the diagonal line from the top left and down to the bottom right of the matrix. These elements have the same value of the two indices. The diagonal elements of matrix Y are denoted by y_{ii} or by y_{jj}. For a square matrix, this applies for all values of i or all values of j. For a square matrix X (an $n \times n$ matrix), the elements of the main diagonal are:

$$x_{1,1},\ x_{2,2},\ x_{3,3},\ x_{4,4},\ \ldots,\ x_{n,n}$$

An *identity matrix* of size n, denoted by I_n is a square matrix that has all the diagonal elements with value 1, and all other elements (off-diagonal) with value 0. The following is an identity matrix of size 3 (of order n):

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

8.6.3 Multi-Dimensional Arrays Using SPSCL

To declare a two-dimensional array in SPSCL, two index numbers are used. The first number defines the range of values for the first index (rows) and the second number defines the range of values for the second index (columns). The following SPSCL statements declare a two-dimensional array named *rainf* with 30 rows and 10 columns.

```
constants
   define ROWS = 30 of type integer
   define COLS = 10 of type integer
variables
   define rainf array [ROWS][COLS] of type float
```

To reference individual elements of a two-dimensional array, two integer numbers are used as indexes. In the following pseudo-code statements each individual element is accessed using index variables *i* and *j*. The code assigns all the elements of array *rainf* to 2.4:

```
for j = 0 to COLS - 1 do
     for i = 0 to ROWS - 1 do
          set rainf [i][j] = 2.4
     endfor
endfor
```

In these operations with two-dimensional arrays, nested loops are used: an outer loop and an inner loop. The inner loop varies the row index *i* from 0 to value *ROWS-1*, and outer loop varies the row index *j* from 0 to *COLS-1*. The assignment sets the value to the element at row *i* and column *j*.

8.6.4 Multi-Dimensional Arrays in C

The following lines of C code declare a two-dimensional array named *rainf* with 30 rows and 10 columns, and assign the value 2.4 to every element of the array.

```
const int ROWS = 30;
const int COLS = 10;

double rainf[ROWS][COLS];
for (j = 1; j < COLS; j++)
   for (i = 0; i < ROWS; i ++)
         rainf [i][j] = 2.4;
```

The following lines of code declare vectors *x* and *w* and a two-dimensional array *z*. The element values of *x* are assigned from a starting value of 2 ending in 11 in increments of 2. For array *w*, the assignments of values starting at 1.0, a final value of 9.0, and in increments of 2.0.

The for-loop assigns values to the the elements of the double-dimension array *z* to the first row from the elements of row vector *x*, to the elements of the second row from the values in vector *w*.

```
constants
   define m of type integer = 5
variables
   define x[10] array of type double
   define w [10] array of type double
   define z[2][10] array of type double
begin
   set m = arraycol (x, 2.0, 11.0, 2.0)
   set m = arraycol (w, 1.0, 9.0, 2.0)
   for j = 0 to N-1 do
        set z[0][j] = x[i]
        set z[1][j] = w[i]
   endfor
```

The following is the corresponding code in C.

```
int m;
double x[10];
double w[10];
double z[2][10];
const int N = 5;
m = arraycol(x, 2.0, 11.0, 2.0);
m = arraycol(w, 1.0, 9.0, 2.0);
for(j = 0; j < N; j++)  {
         z[0][j] = x[i];
         z[1][j] = w[i];
}
```

To access individual elements of a two dimensional array, two indexes are used. The first number is the index for rows and the second number is the index for columns. To access the value of the element in array z defined previously at row 1 and column 3, the syntax is: $z[1][3]$. For example in SPSCL then in C:

```
set zz = z[1][3] + 13.5
. . .
zz = z[1][3] + 13.5;
```

8.6.5 Passing Multi-Dimensional Arrays

The parameter declaration in a function prototype for multi-dimensional arrays, all index sizes except the first one must be specified. This is illustrated in the following function prototype in SPSCL and the corresponding code in C.

```
function yfunct parameters y array [][200] of type double,
                           yrows of type integer
void yfunct (double y [][200], int yrows);
```

This function prototype has a declaration with two parameters, the first one *y* is a double-dimensional array with the size of the second dimension equal to 200. The first dimension is not indicated. The second parameter is an integer that is used by the function as the value of the size of the first dimension. Instead of the value of the second index, a symbolic constant can be used. The function prototype of function *yfunct* may be more conveniently declared in the following form:

```
function yfunct parameters y array [][N] of type double,
                           yrows of type integer
```

The following SPSCL statements declare the symbolic constants *M* and *N*, declare a two-dimensional array *myarray*, then call function *yfunct* with two arguments.

```
constants
    define M of type integer = 100     // number of rows
    define N of type integer = 200     // number of columns
. . .
variables
    define myarray array [M][N] of type double
. . .
    call yfunct using myarray, M
```

8.7 Applications Using Arrays

Several simple applications of arrays are discussed in this section. A few of these applications perform simple manipulation of arrays, other applications perform slightly more complex operations with arrays such as searching and sorting.

8.7.1 Computing The Average Value in an Array

To compute the average value in an array, an algorithm can be designed to first compute the summation of all the elements in the array. The accumulator variable *sum* is used to store this. Second, the algorithm computes the average value by dividing the value of *sum* by the number of elements in the array. The following listing has the pseudo-code description of the algorithm.

1. Initialize the value of the accumulator variable, *sum*, to zero.
2. For every element of the array, add its value to the accumulator variable *sum*.
3. Divide the value of the accumulator variable by the number of elements in the array, n.

The accumulator variable *sum* stores the summation of the element values in the array named *varr* with n elements. The average value *ave* of array *varr* using index j starting with $j = 0$ to $j = n-1$ is expressed mathematically as:

$$ave = \frac{1}{n} \sum_{j=0}^{n-1} varr_j$$

The following listing shows the implementation of the algorithm using an SPSCL program. The number of values in the array is (*num*) and the array elements are inputted from the input device. This program is stored in file: `avearray.scl`. This program can use only a single for-loop to shorten the program.

Listing 8.1: Function that computes average of values in an array.

```
implementations
description
   This program computes the average of the
   elements values in array marr.
   */
function main is
constants
```

```
     define N of type integer = 100    // max number of elements in the array
 variables
    define sum of type float
    define ave of type float    // average value
    define svalue of type float
    define j of type integer
    define num of type integer
    define varr array [N] of type float
 begin
    input "Enter number of values to store: ", num
    for j = 0 to j < num  do
        input "Enter element value: ", svalue
        set varr[j] = svalue
    endfor
    //
    set sum = 0.0
    for j = 0 to j < num do
         add marr[j] to sum
    endfor
    set ave = sum / num
    display "Average value in array: ", ave
endfun main
```

8.7.2 Maximum Value in an Array

Consider a problem that deals with finding the maximum value in an array. The algorithm with the solution to this problem examines all the elements of the array.

Variable *max_val* stores the maximum value found so far and the index variable is *i*. The following is the algorithm description.

1. Read the value of the number of values to store in the array, *num_elements*.
2. Read the values of the array elements. This and the previous step are performed in function *main*. The rest of the operations are performed in function *max_array*.
3. Initialize the variable *max_val* that stores the current largest value found (so far). This initial value is the value of the first element of the array.
4. Initialize the index variable (value zero).
5. For each of the other elements of the array, compare the value of the next array element; if the value of the current element is greater than the value of *max_elem* (the largest value so far), change the value of *max_val* to this element value.

The SPSCL code with function *max_array* appears in the following listing. Function *max_array* returns the maximum value in the array. The input parameters are the array itself and the number of elements in the array. The complete SPSCL program is stored in file `maxarray.scl`.

```
import "scl.h"
/* Program: maxarray.scl
   J Garrido 12-2-2021
*/
```

```
forward declarations

function max_array return type integer
 parameters
       a array [] of type integer, num_elements of type integer
global declarations
constants
   define ARRAY_SIZE  of type integer = 50

/*
  This program computes the maximum value in an array
*/
implementations

  description
      This is the main logic of the program       */
  function main is
    variables
      define marray array [ARRAY_SIZE] of type integer
      define marrval of type integer
      define max_val of type integer
      define kelements of type integer
      define j of type integer
    begin
      input "Number of elements to process: ", kelements
      for j = 0 to kelements -1 do
         input "Enter integer value: ", marrval
         set marray[j] = marrval
         // display "value entered: ", marray[j]
      endfor
      set max_val = max_array(marray, kelements)
      display "Max Integer in array: ", max_val
      exit
  endfun main
  //
  description
        The following function definition finds the maximum value
         in an integer array. To invoke the function, two arguments
         are used: the integer array and the current number of
         elements in the array.    */
  function max_array return type integer
   parameters
       a array[] of type integer,
       num_elements of type integer
   is
   variables                          // local variables
       define i of type integer       // used as index of the array
       define max_elem of type integer  // maximum value found so far
   begin
       set max_elem = a[0]
       set i = 1
```

```
        // display "initial max_elem: ", max_elem
        while i < num_elements do
            if a[i] > max_elem
            then
                set max_elem = a[i]
                // display "max_elem: ", max_elem
            endif
            increment i   // set i = i + 1
        endwhile
        return max_elem
    endfun max_array
```

8.7.3 Searching

Searching consists of looking for an array element for a particular value known as the key and involves examining some or all elements of an array. The search ends when and if an element of the array has a value equal to the value of the key. Two general techniques for searching are: linear search and binary search.

Linear Search

With linear search the elements of an array are examined in a *sequential* manner starting from the first element of the array, or from some specified element. Every array element is compared with the key value, and if an array element is equal to the key value, the algorithm has found the element and the search terminates. This may occur before the algorithm has examined all the elements of the array.

The index of the element in the array that is found is the result of the search. If the requested value is not found, the algorithm indicates this with a negative result or in some other manner. The following is an algorithm description of a general linear search using a search condition of an element equal to the value of a *key*.

1. Repeat for every element of the array:
   (a) Compare the current element with the requested value or key. If the value of the array element satisfies the condition, store the value of the index of the element found and terminate the search.
   (b) If values are not equal, continue search.
2. If no element with value equal to the value requested is found, set the result to value -1.

The following function in SPSCL searches the array for an element with the value of the key *keyval*. Function *lsearch* returns the index value of the element equal to the key. If no element with value equal to the requested value is found, the function returns a negative value. The complete program is stored in file `lsearch.scl`.

```
description
  This function implements a linear search
  of an array varr using a key value. The result is the index value of the
  element found, or -1 */
function lsearch return type integer
```

```
 parameters
       varr array [] of type double, num of type integer,
       keyval of type double is
 variables
      define j of type integer
      define found of type boolean = false
      define tolerance of type double = 0.0001
      define rel of type double
 begin
      set j = 0
      while j < num and found not equal true do
            set rel = abs(varr[j] - keyval)
            if rel <= tolerance then
                set result = j
                set found = true
            else
                increment j
            endif
       endwhile
       if found not equal true
       then
           set result = -1
       endif
endfun lsearch
```

With linear search, the number of comparison with respect to the number of elements N in the array defines a linear relation. This means that the number of comparisons grow linearly with respect to the growth of N, the big O notation for linear search is thus denoted as $O(N)$.

Binary Search

Binary search is a more complex search method, compared to linear search. This search technique is very efficient compared to linear search because the number of comparisons is smaller.

The prerequisite for binary search technique is that the element values in the array to search be sorted in ascending order. The array elements to include are split into two halves or partitions of about the same size. The middle element is compared with the key (requested) value. If the element with this value is not found, the search is continues on only one partition. This partition is again split into two smaller partitions until the element is found or until no more splits are possible because the element is not found. The informal description of the algorithm is:

1. Assign the lower and upper bounds of the array to *lower* and *upper*.
2. While the lower value is less than the upper value, continue the search.
   (a) Split the array into two partitions. Compare the middle element with the key value.
   (b) If the value of the middle element is equal to the key value, terminate search and the result is the index of this element.
   (c) If the key value is less than the middle element, change the upper bound to the index of the middle element minus 1. Continue the search on the lower partition.
   (d) If the key value is greater or equal to the middle element, change the lower bound to the index of the middle element plus 1. Continue the search on the upper partition.

3. If the key value is not found in the array, the result is -1.

The following function in SPSCL searches the array for an element with the value of the key *keyval*, using binary search. Function *bsearch* returns the index value of the element equal to the key. If no element with value equal to the requested value is found, the function returns a negative value. The complete program is stored in file `bsearch.scl`.

```
description
  This function implements a binary search
  of a sorted array using a key value. The result is the index value
  of the element found, or -1 if none found
  */
function bsearch return type integer
  parameters sarray array [] of type double, keyval of type double,
     lowidx of type integer, highidx of type integer is
constants
    set TOLER of type double = 0.0001
variables
    define index of type integer
    define mididx of type integer
begin
    set index = -1
    while lowidx <= highidx do
        set mididx = lowidx  + ((highidx - lowidx) / 2)
        if sarray[mididx] < keyval then
            set lowidx = mididx + 1
        elseif sarray[mid] > keyval then
            set highidx = mididx - 1
        elseif abs(sarray[mididx] - keyval) <= TOLER
        then
             set index = mididx
             return index
        endif
    endwhile
    return index
endfun bsearch
```

8.8 Structures

SPSCL and C support structures, also known as *records*. A structure type is a composite type because it allows its components to have different types. To define a structure type, the **struct** keyword and a type name are used and enclose the data components of the structure.

8.8.1 Structure Types

The following example declares a structure type *ComplexNum* with two data components of type *double*, in SPSCL the definition of this structure is:

```
struct ComplexNum is
```

```
    define realpart of type double
    define imagpart of type double
endstruct  ComplexNum
```

The definition of this structure in the C programming language is:

```
struct ComplexNum {
    double realpart;
    double imagpart;
};
```

This definition introduces *struct ComplexNum* as a new type. This programmer-defined type can be used to declare variables of this type. In the following example, two structures: *mstr1* and *mstr2* are declared. Note that the keywords **struct** and **endstruct** must be used when declaring these structures.

A component of a structure variable can be accessed by typing the name of the component, the keyword **in**, followed by the name of the structure variables. Using the *dot notation* can also be used, an individual component of a structure can be accessed by indicating the name of the structure, a dot, and the name of the component. In the example, a value is assigned to each individual components of the structure variable *mstr1*. Then the entire structure *mstr1* is copied to the structure *mstr2*.

```
define mstr1 of struct ComplexNum
define mstr2 of struct ComplexNum
. . .
set mstr1.real.part = 132.54      // assign value to component of var1
set imagpart in mstr1 = 37.32   // assign value to component of var1
set mstr2 = mstr1                    // copy the whole structure mstr1 to mstr2
```

The equivalent definition of struct variables in the C programming language is:

```
struct ComplexNum mstr1;
struct ComplexNum mstr2;
mstr1.realpart = 132.54;
mstr1.imagpart = 37.32;
mstr2 = mstr1;
```

Although assignment to a structure can be performed directly from another structure of the same type with an assignment statement, direct comparison is not allowed. Instead, the individual components are compared with an **if** statement.

As mentioned previously, this type of complex data structure allows its components to have different types. The data structure is also known as a *record* and the individual components are also known as *fields*.

8.8.2 New Type Names

Programs can provide alternative names for the primitive and program-defined data types. This does not create a new type. The keyword in SPSCL for renaming a type is **definetype** and provides a new name to a type and the syntax is:

definetype ⟨ *type* ⟩ ⟨ *new_type_name* ⟩

For example,

```
definetype struct ComplexNum ComplexT
```

This define a new name to *struct ComplesNum*. For the rest of the program this type can be used as *ComplexT*. The following declares *mstr3* with the new type name *ComplexT*.

```
define mstr3 of type ComplexT
```

C supports the definition of new type names, which basically provides new shorthand names for types. The **typedef** statement is used for defining type names and the syntax is:

typedef ⟨ *type* ⟩ ⟨ *type_name* ⟩

The following example defines the type name *ComplexT* as the name for the type *struct ComplexNum*.

```
typedef struct ComplexNum ComplexT;
. . .
ComplexType var;          // Declare struct variable var
```

8.8.3 Array of Structures

An array of structure type can be declared in a similar manner to an array of a simple or primitive type. To declare an array of a structure, a previously-defined structure type is used as the type of the array.

The following example declares an array with name *cmplxarray* of 150 elements using type *ComplexT* (defined previously). Then values are assigned to the individual components of the first two elements of the array.

```
define cmplxarray array [150] of type CompleT
. . .
set realpart in cmplxarray[0] = 54.85
set imagpart in cmplxarray[0] = 17.87
set cmplxarray[1].realpart = 54.85
set cmplxarray[1].imagpart = 17.87
```

The equivalent statements in the C programming language are:

```
ComplexT cmplxarray[200];
cmplxarray[0].realpart = 54.85;
cmplxarray[0].imagpart = 17.87;
cmplxarray[1].realpart = 54.85;
cmplxarray[1].imagpart = 17.87;
```

8.9 Sparse Matrices

A Sparse matrix is one that has a large number of elements with a zero value. Sparse data can be easily compressed, which in turn can significantly reduce the amount of memory required.

There are two types of sparse matrices. In the first type of sparse matrix, all elements above the main diagonal have a zero value. This type of sparse matrix is also called a *lower tridiagonal matrix* because all the elements with a non-zero value appear below the diagonal.

$$\begin{bmatrix} 0.5000 & 0 & 0 \\ 1.8000 & 7.2300 & 4.4000 \\ 3.500 & 1.2500 & 9.7500 \end{bmatrix}$$

To store a lower-triangular matrix, a one-dimensional array can be used and it stores only non-zero elements. The mapping between a two-dimensional matrix and a one-dimensional array can be done in any one of the following ways:

- Row-wise mapping, the contents of the array will be the non-zero elements of every row.
- Column-wise mapping, the contents of the array will be the non-zero elements of every column.

In an *upper-triangular matrix*, has n non-zero elements are those above the diagonal. Another variant of a sparse matrix is one in which elements with a non-zero value can appear only on the diagonal or immediately above or below the diagonal. This type of matrix is also known as a tri-diagonal matrix.

Summary

An array is a data structure that stores several values of the same type. Each of these values is known as an element. After the array has been declared the capacity of the array cannot be changed. To refer to an individual element an index is used to indicate the relative position of the element in the array. Structures is another implementation data structure that is very useful in more complex problems. array of structures is an important programming facility for slightly more advanced programs.

Searching an array consists of looking for a particular element value or key. Two common search algorithms are linear search and binary search. Computing an approximation of the rate of change and the area under a curve is much more convenient using arrays, as shown in the case study discussed.

Key Terms

declaring arrays	accessing elements	array capacity
index	array element	element reference
searching	linear search	binary search
key value	algorithm efficiency	summation
accumulator	structures	sparse matrix

Exercises

Exercise 8.1 Develop a program that computes the standard deviation of values in an array. Implement using the SPSCL programming language. The standard deviation measures the spread, or

dispersion, of the values in the array with respect to the average value. The standard deviation of array X with n elements is defined as:

$$std = \sqrt{\frac{sqd}{n-1}},$$

where

$$sqd = \sum_{j=0}^{n-1} (X_j - Av)^2.$$

Exercise 8.2 Develop a program that finds the minimum value element in an array and returns the index value of the element found. Implement using the SPSCL programming language.

Exercise 8.3 Develop a program that computes the average, minimum, and maximum rainfall per year and per quarter (for the last five years) from the rainfall data provided for the last five years. Four quarters of rainfall are provided, measured in inches. Use a matrix to store these values. Implement using the SPSCL programming language.

Exercise 8.4 Develop a computational model that sorts an array using the Insertion sort technique. This sort algorithm divides the array into two parts. The first is initially empty; it is the part of the array with the elements in order. The second part of the array has the elements in the array that still need to be sorted. The algorithm takes the element from the second part and determines the position for it in the first part. To insert this element in a particular position of the first part, the elements to right of this position need to be shifted one position to the right. Implement using the SPSCL programming language.

9. Pointers

9.1 Introduction

Pointers provide a very convenient and powerful facility to manipulate data such as arrays and linked lists, which store large number of values. SPSCL, C, and C++ programming languages support pointers. Using pointers provide a much more convenient manner for handling variables and arrays. There are several tasks that can be performed with pointers, such as dynamic memory allocation.

9.2 Pointer Fundamentals

In a computer system, every variable is stored in a particular memory location (allocated by the compiler and operating system) and every memory location is defined by an *address*.

A pointer variable is one that contains the location or address of memory where another variable, data value, or function is stored. A pointer is a variable whose value is the address of another variable. Pointers are important for implementing more advanced types of data as well. For example, a linked list is a data structure that uses pointers to link individual nodes.

9.3 Pointers in SPSCL

In SPSCL, the **pointer** keyword is used to declare pointer variables or parameters. The first declaration in the following example includes a definition of a pointer variable named *palpha* of type double; the second declaration defines a pointer variable, *ttime*, of struct *Timer*.

```
define palpha pointer of type double
define pbeta pointer of type float
define pstudent pointer of struct student
```

The **address** keyword is used to get the memory address of a variable. Usually, an address is assigned to a pointer variable. The following example gets the address of variable *alpha* and assigns it to pointer variable *palpha*, which was defined previously.

```
set alpha = 25.5
set  palpha = address alpha
```

Figure 9.1 shows that the value of variable *alpha* is 25.5 and the value of pointer variable *palpha* is the address of variable *alpha*.

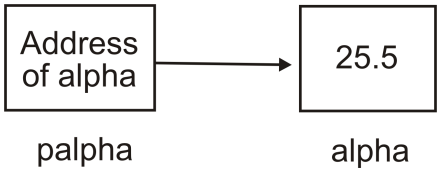

Figure 9.1: Pointer variable *palpha* pointing to variable *alpha*

The **deref** keyword is used to dereference a pointer variable. It gets the value of a variable or address pointed at by the specified pointer variable. In the following example, pointer variable *palpha* is dereferenced and the corresponding value is assigned to variable *alpha*:

```
set  alpha = deref palpha
```

9.4 Pointers in C

The C programming language the asterisk (*) is used to declare a pointer variable. A pointer is declared with the type of the variable it will point to.

A pointer variable of type *char* can be used to point to a single character or an array of characters. A similar type can be used for a *struct*. The following are examples of pointer variable declarations.

```
double *palpha;
struct node *nodep;
int * numnodes;            // an integer pointer variable

char* descP;                // declares a pointer of type char
```

To assign an address to a pointer variable the & operator is used. This operator gets the address of a variable to point at and this value can be assigned to a pointer variable. For example,

```
double alpha;
double *palpha;
alpha = 25.5;
palpha = & alpha;        // palpha now points to alpha
```

Once a pointer variable has an appropriate address, which means the pointer variable is actually pointing to a variable, it can be used to dereference a variable. For example, continuing the previous

lines of code, pointer variable *palpha* can be used to change the value of the variable it currently points to. The following assignment statement changes the value of variable *alpha* using pointer variable *palpha*.

```
*palpha = 75.25;
```

9.5 Simple Aritmetic Operations on Pointers

A pointer is an address that is a numeric value, there are a few basic arithmetic operations that can be performed on a pointer. These are:

- increment the value of the pointer with the ' ++ ' operator
- decrement the value of the pointer with the ' -- ' operator
- addition to the value of the pointer with the ' + ' operator
- subtraction from the value of the pointer with the ' - ' operator

The increment operator will add a value depending on the size of the type of the pointer variable. For integers that are four bytes in length, the increment operator adds 4 to the address value of the pointer variable.

9.6 Converting Types

When it becomes necessary to convert one type to another, *casting* can be used. The keyword **cast** is used on the right-hand side of an assignment statement, the general syntax for converting the type of a variable is:

cast ⟨ *new_type* ⟩ ⟨ *expression* ⟩

For example,

```
define keylen of type integer
define uklen of type unsigned integer
define ukeylen of type unsigned integer
. . .
set ukeylen = cast unsigned integer (keylen + 1)  // casting
set uklen = cast unsigned (keylen +1)
```

Using C, the new type is enclosed in parenthesis:

```
(type) variable
```

For example, in an assignment statement, the following lines of C code change the type of a variable from *int* to *long*.

```
long xx;
int xy;
xx = 76;
xy = (long) xx;
```

The type conversion with pointer variables is very similar. The following examples using SPSCL illustrate type conversion with pointers.

```
define xxp pointer of type long
define yyp pointer of type integer
define palpha pointer of type double
define pbeta pointer of type float
define pstudent pointer of struct student
define pdate pointer of struct datestr
. . .
set pbeta = cast pointer float palpha
set pstudent = cast pointer struct student pdate  // careful
```

Some castings are syntactically correct but can be semantically incorrect, so we have to be careful when using type conversion. When casting of pointer types in C, the asterisk (*) must be included. The general syntax is:

```
(type *) pointer_variable
```

For example, the type conversion from a pointer to an integer to a pointer to a *long* variable. The following lines of C code includes an example of pointer type conversion from *long* to *int*.

```
long xx
long *xxp;
int *yyp;
xxp = &xx;
yyp = (int *) xxp;
```

9.7 Pointer Parameters

A function call normally returns only one value at a time back to the calling function. As mentioned in a previous chapter, there are two general techniques to pass *arguments* when calling a function:

- pass by value
- pass by reference

The function can declare reference (or variable) *parameters* and these are as a pointer variables. The following function *swap_val* declares reference parameters *xp1* and *yp2*. This allows the arguments to be input-output.

```
import <stdio.h>
forward declarations

function swap_val parameters xpt1 pointer of type integer,
        yp2 pointer of type integer

function main is
variables
     define x of type integer = 5
     define y of type integer = 10
begin
     call swap_val using address x, address y
     display "x: ", x, "y: ", y
     return 0
endfun main
//
function swap_val parameters aptrpointer of type integer,
           bptr pointer of type integer
is
variables
     define mtemp of type integer
begin
     set mtemp = deref bptr
     set deref bptr = deref aptr
     deref aptr = mtemp
endfun swap_val
```

The following is the corresponding source code in C:

```
#include <stdio.h>
void swap_val (int *xp1, int *yp2);     // function prototype

int main()
{
     int x = 5;
     int y = 10;
     swap_val (&x, &y);
     printf (" %d and b = %d\n\n", x, y);
     return 0;
}
void swap_val (int *aptr, int *bptr)
{
     int mtemp;
     mtemp = *bptr;
     *bptr = *aptr;
     *aptr = temp;
}
```

9.8 Pointers with Value NULL

It is good programming practice to initialize a pointer variable with a NULL value so the pointer will not initially point to any variable. The NULL value is a constant with a value of zero defined in the C standard libraries. The following SPSCL source program illustrates the *NULL* value usage for pointer variables:

```
import "scl.h"
description
      This is the main logic of the program
      */
function main is
variables
    define xxp pointer of type integer
    define uy of type unsigned integer
begin
    set xxp = NULL
    if xxp == NULL then
         set uy = cast unsigned integer xxp
         display "value of uy: " , uy
    endif
endfun main
```

The correspong source code in the C language is:

```
1 /* This program displays the value of a pointer variable
2     Type conversion with casting is necessary to
3     'unsigned int' to use format '%x'
4 */
5 #include <stdio.h>
6 int main ()
7 {
8      int *xxp;
9      unsigned int uy;
10     intptr = NULL;
11     if (xxp == NULL) {
12         uv = (unsigned int) xxp;
13         printf("Value of uy: %x \n", uy);
14     }
15     return 0;
16 }
```

9.9 Arrays as Pointers

An array can be manipulated as a pointer and this is useful with array parameters and when an array is the return value in a function.

9.9.1 Pointer Parameters for Arrays

An array can be used as an argument in a function call. In the function definition, the corresponding parameter is declared as a pointer. Arrays as arguments are considered to be pointers by the called functions receiving them and these functions can modify the original copy of the variable. The following example in SPSCL source code shows the arrays as pointers. The source file of the program is stored in file `arrayptr.scl`.

```
 description
    Program: arrayptr.scl
    This program uses an integer array as argument
     in the call function with myarr.
    The function declares the parameter as an integer pointer.
   */
 import <stdio.h>
 forward declarations

 function farraym
     parameters iarr pointer of type integer,
            factor of type integer, mcount of type integer

implementations

function main is
  variables
       define j of type integer
       define myarr array [] of type integer =
               {10, 20, 30, 40, 50, 60, 70, 78}
       define mcount of type integer = 8   //current number of elements
  begin
        call farraym using myarr, 10, mcount
        for j = 0 to j < mcount do
             display "Value: ", myarr[j], j
        endfor
        return 0
endfun main
//
function farraym parameters arrayptr pointer of type integer,
             lfactor of type integer is, lcount of type integer
    variables
         define j of type integer
    begin
        for j = 0 to lcount - 1 do
             set arrayptr [j] = arrayptr[j] * lfactor
        endfor
endfun farraym
```

The source code in the C programming language follows:

```
   /*
   Program: arrayptr.c
```

```
 This program uses an integer array as argument
 to call function with myarr.
 The function declares the parameter as an integer pointer.
 */
 #include <stdio.h>
 void farraym (int *iarr, int factor, int mcount); // prototype
 const int NUMELEM = 20;     /* array size */
 int main()
 {
        int j;
        int mcount = 8
        int myarr[NUMELEM] = {3, 10, 15, 17, 25, 12, 27, 35};
        farraymult (myarr, 15, mcount);
        for (j = 0; j < mcount; j++)
             printf("Element: %d value: %d \n ", j, myarr[j]);
        return 0;
}
void farraym (int *arrayptr, int lfactor, int lcount)
{
      int j;
      for (j = 0; j < lcount; j++)
            arrayptr[j] *= lfactor;
}
```

In function *farraym*, the first parameter is declared as an integer pointer, however, it can also be declared as an array parameter. In this case, the function declaration or prototype is:

```
function farraym
   parameters myarray array[] of type integer,
       pfactor of type integer, pcount of type integer
```

The function prototype in the C language is:

```
void farraym ( int myarray[], int pfactor, int pcount );
```

Integers are by default stored with four bytes, the increment operator adds 4 to the address value of the pointer variable. In the following example the increment operator used on a pointer variable. An array is really implemented as a pointer to the first element and it cannot directly be incremented so its address must be assigned to a pointer variable.

```
import <stdio.h>

function main is
 constants
     define M of type integer = 6
 variables
      define j of type integer
      define intptr pointer of type integer
      define myarr array [] of type integer = {1, 5, 10, 15, 20, 25}
```

```
begin
      set  intptr = myarr
      for  j = 0 to M - 1 do
           display "Element value: ", deref intptr, j
           increment intptr    // point to next element of array myarr
      endfor
      return 0
endfun main
```

The corresponding source code in the C language is:

```
 1 #include <stdio.h>
 2 int main ()
 3 {
 4    const int M = 6;
 5    int j;
 6    int *intptr;
 7    int myarr[] = {1, 5, 10, 15, 20, 25};
 8    intptr = myarr;
 9    for ( j = 0; j < M; j++) {
10       printf ("Current element: %d  value: %d \n", *intptr, j);
11       intptr++;  /* point to next element of array myarray */
12    }
13    return 0;
14 }   /* end function main */
```

9.9.2 Functions that Return Arrays

If a function returns an array it specifies a pointer type as the function return type. In the following example, function prototype *farraym2* has as return type of the function a pointer to an integer. This program is stored in file `arraym.scl`

```
description
   Program: arraym.c
   This program calls a function that returns a new array as an
   integer pointer.    The function uses an integer array as argument
  */
import <stdlib.h>
import <stdio.h>
forward declarations

function farraym2 return pointer of type integer
    parameters parray constant pointer of type integer,
   pfact of type integer

global declarations
constants
   define NUMARR of type integer = 6

implementations
```

```
function main is
variables
   define j of type integer
   define marray array [] of type integer = {1, 5, 10, 15, 20, 25}
   define parray pointer of type integer
begin
   set parray = farraym2 (marray, 15)
   for j = 0 to j < NUMARR  do
       display "Element value: ", parray[j], j
   endfor
   return 0
 endfun main
 //
 function farraym2 return pointer of type integer
      parameters larray constant pointer of type integer,
         lfact of type integer
 is
 variables
      define narray pointer of type integer
      define j of type integer
 begin
      set narray = malloc(NUMARR * sizeof(integer))
      for j = 0 to j < NUMARR do
          narray[j] = larray[j] * lfact
      endfor
      return narray
 endfun farraym2
```

The source code in C of the program follows.

```
   /*
   Program: arraym.c
   This program calls a function that returns a new array as an integer pointer
    uses an integer array as argument
 7 */
 8 #include <stdlib.h>
 9 #include <stdio.h>
10 int * farraym2 (const int *parray, int pfact);
11 const int NUMARR = 6;     /* array size */
12 int main()
13 {
14     int j;
15     int marray[] = {1, 5, 10, 15, 20, 25};
16     int *parray;
17     parray = farraym2 (marray, 15);
18     for (j = 0; j < NUMARR; j++)
19          printf("Element: %d value: %d \n ", j, parray[j]);
20     return 0;
21 }
22 int * farraym2 (const int *larray, int lfact)
```

```
23 {
24       int  * narray;
25       int j;
26       narray = malloc(NUMARR * sizeof(int));
27       for (j = 0; j < NUMARR; j++)
28             narray[j] = larray[j] * lfact;
29       return narray;
30 }
```

9.9.3 Dynamic Memory Allocation

When a program runs, it can request additional memory and the operating system can allocate the requested memory (if available) to the program. This technique of memory allocation is known as *dynamic memory allocation*. The operating system allocates a contiguous block of memory to the program and uses a pointer variable to refer to the starting address of the memory block allocated. This allocation of memory is performed by calling system function *malloc*. Memory allocated can be resized with system function *realloc* and memory can be de-allocated with system function *free*.

Calling function *malloc* requires one argument, which is the number of bytes of memory to allocate. The function returns a *void* pointer, which provides the address of the beginning of the allocated block of memory.

Calling function *realloc* function requires two arguments: the pointer to the memory block to be reallocated, and a number of *type size_t* that specifies the new size for the block. The function returns a void pointer to the newly reallocated block.

The deallocation of memory that has been allocated to a block is performed by calling function *free*. Calling this function requires only one argument, the pointer to the block to deallocate. The function does not return a value.

In the previous example, function *farraym2* creates an array of size *NUMARR* elements by calling the library function *malloc*. The number of bytes for an integer of type *int* is 4; therefore, the total number of bytes allocated is *NUMARR* times 4. Pointer variable *narray* points to the block of memory allocated. Function *malloc* is called in the following SPSCL instruction:

```
set narray = malloc(NUMARR * sizeof(integer))
```

The most practical command in SPSCL to allocate memory to a data object *data* of type *DataT* is using the **create** keyword:

```
create data type DataT
```

9.9.4 Pointers to Structures

A pointer variable to a structure can be declared by a statement similar to previous declaration of pointer variables. The declaration includes the structure type with an asterisk and a variable name. This is a pointer of a structure type previously defined. The following example shows a declaration of pointer struct variable *NodePtr*. In this case, the struct *Node* defined first.

```
struct Node is
variables
   define val of type double
   define indexval of type integer
endstruct  Node
. . .
define nodePtr pointer of struct Node
```

The source code in the C language follows.

```
struct Node {
   double val;
   int indexval;
};
. . .
struct Node *nodePtr;
```

The following program shows the definition of a structure type, declarations of structure variables, declaration of pointer variables, and use of pointer variables to structures. To access the components of a variable declared as a pointer to a struct, the keyword **of** is used. The program declares variables *fvar* and *svar* of type *PointT*. It declares a pointer variable *pointPtr* of type *PointT*. Note that the **in** keyword is used to access individual components of the structure variable *fvar*. To access variable *y* of pointer variable *pointPtr*, the keyword **of** is used.

The **in** is also used to display the value of the components of the struct variables *fvar* and *svar*. Dereferencing is used to display the individual components of the structure pointed at by *pointPtr*.

```
import "scl.h"

specifications

struct Point is
     define x of type double
     define y of type double
endstruct Point

definetype struct Point PointT

implementations

function main is
structures
     define fvar of type PointT
     define svar of type PointT
     define pointPtr pointer of type PointT
begin
     set x in fvar = 10.5
     set y in fvar = 7.35
     set pointPtr = address fvar
```

```
     set svar = deref pointPtr
     display "fvar x: ", x in fvar
     display  "fvar y: ", y of pointPtr
     display "svar x: ", x in svar
     display  "svar y: ", y in svar
     return 0
endfun main
```

The following language statements declare an array of pointers to *struct Point*, then assign the address of structure variable *pointv* to the element with index 10 of the pointer array *pointarray*.

```
definetype struct Point Ptstruct

define pointer Ptstruct structPtr
define pointarray array [100] of structPtr
. . .
set pointarray[10] = address pointv
```

The corresponding source code in the C language follows.

```
struct Point * pointarray[100];
pointarray[10] = &pointv;
```

The following notation can be used for dereferencing a pointer to a *struct*. The keyword **of** after the name of the pointer variable can access any of the fields in the pointer to *struct*. This is shown in the following statements:

```
set lx = x of structPtr
set ly = y of structPtr
```

In C, the notation is used as follows:

```
lx = structPtr->x;
ly = structPtr->y;
```

In following statements, a node is defined in a linked list. The third component of the structure is a pointer to a node itself.

```
definetype struct Node nodeT

struct Node is
     define field1 of type integer
     define field2 of type double
     define next_field pointer of type nodeT
endstruct Node
```

The source code in the C language is:

```
typedef struct Node nodeT;
struct Node {
     int field1;
     double field2;
     nodeT* next_field;
};
```

9.10 Arrays of Pointers

In an array of pointers, every element of the array is a pointer. Therefore, every element has to be assigned an appropriate address. The following is an array of *N* elements, declared as pointers of type integer.

```
constants
   define N of type integer = 10
variables
   define larray array [N] pointer of type integer
   define x of type integer
   define y of type integer
   define z of type integer
   . . .
   set x = 5
   set y = 10
   set x = 15
   set larray[0] = address x
   set larray[1] = address y
   set larray[2] = address z
   . . .
```

9.11 Enumerated Types

Enumerated types are arithmetic types and they are used to define variables that can only be assigned certain discrete integer values. The type is defined by enumerating the allowed values in symbolic form.

```
enum weekday is
    Monday, Tuesday, Wednesday, Thursday, Friday
endenum weekday

definetype pointer enum weekday wdtype

define day of type wdtype
. . .
set day = Tuesday
```

The C language source code follows.

```
enum weekday { Monday, Tuesday, Wednesday, Thursday, Friday };
typedef enum weekday wdtype;
wdtype day;
day = Tuesday;
```

In the following example in C, the first statement defines an enumerated type with name *boolean* that includes only two possible values 0 and 1. The symbolic name *False* has a value of 0 and the symbolic name *True* has a value of 1. The second statement defines the name *Bool* for the type *enum boolean*. The third line is a declaration of variable *flag* of type *Bool*. The fourth line is an assignment statement; the value *True* is assigned to variable *flag*.

```
enum boolean {False, True};
typedef enum boolean Bool;
Bool flag;
flag = True;
```

Summary

Using pointers allow a much more flexible, convenient, and powerful approach to manipulate data structures. These implementation data structures are mainly arrays and linked lists, which can handle large number of values in data collections Many computational tasks that are better performed with pointers, such as dynamic memory allocation.

Key Terms

declaring pointers	initializing pointers	address of a variable
dereferencing	reference parameter	dynamic memory allocation
structures	pointer to structures	type renaming
enumerated types	null pointer	type conversion

Exercises

Exercise 9.1 Develop a program with a function that declares two pointer parameters and an integer parameter. The function must compare the characters in the two strings pointed by the first two parameters and return 0 if the first *n* (defined by the third parameter) characters of the strings are equal, otherwise the function returns the index value of the first character that does not match.

Exercise 9.2 Develop a program with a function that declares two pointer parameters and an integer parameter. The function must copy the characters in the string pointed at by the first parameter to the second string. The function must copy the first *n* (defined by the third parameter) characters of the strings.

Exercise 9.3 Using the structure previously defined, *struct ComplexNum*, develop a program that performs complex addition, subtraction, multiplication, and division of the complex values of the

elements of array *x* with the complex values of the elements in array *y*. Include a function for each complex operation that returns a new array with the results.

Exercise 9.4 Using the structure *inventory*, develop a program that maintains inventory data in a warehouse in an array of structure data. The program must display the parts that have a number of units less than 5 and compute the total sales value of the inventory.

```
struct inventory is
    define part of type integer
    define units of type integer
    define price of type double
    define invdescription array [31] of type char
endstruct inventory
```

10. Recursion

10.1 Introduction

Recursion can be used to describe a solution in a simpler manner than with an iterative solution. Recursion is a design and programming technique by which a circular definition is employed, a structure is defined in terms of itself. Recursion can sometimes be simpler and clearer than an iterative solution to a problem. This allows the solution of a big problem by partitioning it into several smaller sub-problems of the same kind and then we can combine the solutions to these sub-problems. This way recursion can be used to describe complex algorithms.

The recursive approach to design and implement functions accomplishes the same goal as using the iterative approach for problems with repetitions. A function or method definition that contains a call to it is said to be recursive.

This chapter introduces the basic concepts that involve recursion and includes a problem-solving application of recursion using recursive function calls.

10.2 Defining Recursive Solutions

A recursive function calls itself from within its own body. A recursive operation achieves exactly what an operation with *iterations* achieves. In principle, any problem that can be solved recursively can also be solved iteratively. With iterations, a set of instructions is executed repeatedly until some *terminating condition* has been satisfied. Similarly with recursion, a set of instructions, most likely a part of a function, is invoked repeatedly unless some terminating condition has been satisfied. The unwinding phase returns the values from the base case to each of the previous calls. A recursive function definition consists of two parts:

1. One or more *base cases* that define the terminating conditions. In this part, the value returned by the function is specified by one or more values of the arguments.
2. One or more *recursive cases*. In this part, the value returned by the function depends on the value of the arguments and the previous value returned by the function. This is sometimes known as the winding phase.

10.3 Examples of Recursive Functions

This section describes and explains three examples of functions that can be defined and implemented with recursion. The functions are: sum of squares, exponentiation, and list reversal.

10.3.1 Sum of Squares

The first example of a simple recursive function is *sum of squares* of natural numbers. The function must add all the squares of numbers from 1 to n . The mathematical description of the function $ssq(m,k)$ as a series is:

$$ssq(n) = 1 + 2^2 + 3^2 + 4^2 + \cdots + (n-1)^2 + n^2$$

The strategy of solution is to decompose the problem into smaller sub-problems, such that the smaller problems can be solved with the same technique as that used to solve the overall problem. The final solution is computed by combining the solutions to the sub-problems. A mathematical specification of this recursive function is as follows, assuming that $n \geq 0$:

$$ssq(n) = \begin{cases} 1, & \text{when } n = 1 \\ 0, & \text{when } n = 0 \\ n^2 + ssq(n-1), & \text{when } n > 1 \end{cases}$$

The recursive implementation of function *ssq* is defined in file `sumsq.scl`. This function is invoked in function *main*.

```
description
   Tests sum of squares
   function sumsq is a test for sum of squares.
     It includes recursive calls  ssq
   */
import <stdio.h>
import <stdlib.h>
forward declarations

function ssq return type integer
  parameters m of type integer
//
implementations
```

```
function main is
  variables
       define n of type integer
       define res of type integer
  begin        // body of function starts here
       input "Enter vlue of n: ", n
       set res = call ssq using n
       display "results sum of squares: ", res
  endfun main
  //
  function ssq return type integer
     parameters m of type integer
  is
  variables
      define lres of type integer
  begin
      if m == 0    then
         return 0
      elseif m == 1 then
         return 1
      else
         // recursive call
         set lres = ssq( m-1) + m * m
         return lres
      endif
  endfun ssq
```

10.3.2 Exponentiation

Another example of a recursive function is *exponentiation*, y^n. The informal description of exponentiation is:

$$y^n = y \times y \times y \times y \times \ldots \times y$$

For example,

$$y^3 = y \times y \times y$$

A mathematical specification of the recursive exponentiation function is as follows, assuming that $k \geq 0$:

$$y^k = \begin{cases} 1, & \text{when } k = 0 \\ y \times y^{k-1}, & \text{when } k > 0 \end{cases}$$

The base case in this recursive definition is the value of 1 for the function if the argument has value zero, that is $y^k = 1$ for $k = 0$. The recursive case is $y^k = y \times y^{k-1}$, if the value of the argument is greater than zero.

The following program includes a recursive implementation for the exponentiation function, *expn(y, k)*. The source code defines the recursive function that is called by function *main*. The program is stored in file `recexpn.scl`.

```
description
  This program includes a recursive function to
  compute exponentiation of y to the power of k
  */
import <stdio.h>
import <stdlib.h>

forward declarations

 function expn return type double
    parameters y of type double , k of type integer

implementations

description
     This function is a test for exponentiation.
     It calls the recursive function, expn
     */
function main is
  variables
       define y of type double
       define n of type integer
   define res of type double
 begin        // body of function starts here
       input "Enter value of y: ", y
   input "Enter value of n: ", n
   set res = expn(y, n)
   display "results exponentiation: ", res
endfun main
  //
description
     This function computes the exponentiation of y
     Recursively. To the power of k,
     assuming that k > 0.
  */
function expn return type double
  parameters y  of type double, k of type integer is
 variables
      define res of type double
begin
     display "expn ", y, " power ", k
     // base case
     if k == 0 then
```

```
        set res = 1.0
    elseif k < 0 then
        // exceptional case
        display "Negative value of k"
        set res = 1.0
    else
        // recursive case
        // k greater than 0
        // recursive call
        set res = y * expn(y, k-1)
        display "y = ", y, " k = ", k, " expn ", res
    endif
    return res
endfun expn
```

The computation of the recursive function *expn* involve successive calls to the function continue until it reaches the base case. After this simple calculation of the base case, the function starts calculating the exponentiation of y with power 1, 2, 3, and then 4.

10.3.3 Reversing A Linked List

An interesting set of recursive problems is the recursive processing of a linked list. Recursive reversal of a linked list is simpler and more elegant than performing the same operation iteratively.

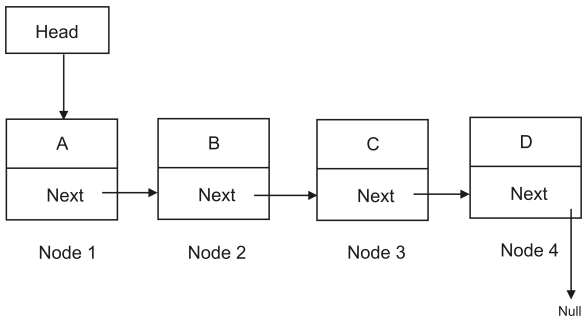

Figure 10.1: Linked list of characters

Consider a simple list of characters, List = ('A', 'B', 'C', 'D'). Figure 10.1 shows a linked list with four nodes, each with a character. The overall task for this problem is to change the list so that the nodes will appear in reverse order. The recursive solution to this problem applies a strategy of divide and conquer.

- The first step is to split the list into a head and tail sublists. The head sublist contains only the first node of the list. The second sublist contains the rest of the list.
- The second step is to reverse (recursively) the tail sublist.
- The third step is to concatenate the reversed tail sublist and the head.

Figure 10.2 illustrates the general steps of reversing a linked list recursively. After partitioning the

list, the head sublist contains only the first element of the list: *Hlist = ('A')*. The second sublist contains the rest of the list, *Tlist = ('B, 'C', 'D')*.

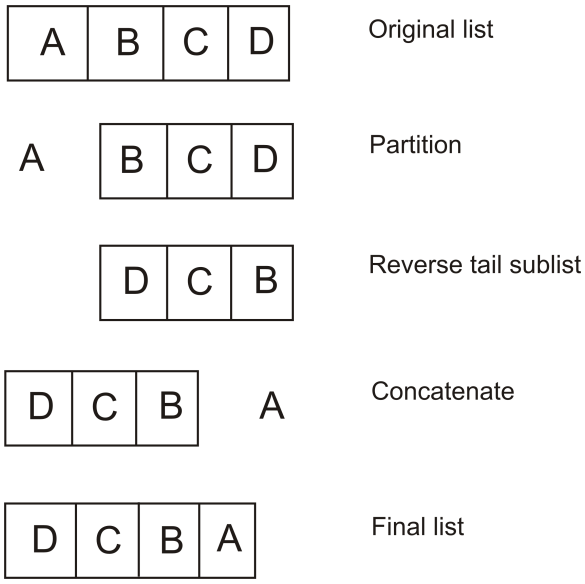

Figure 10.2: Reversing a linked list of characters

The recursive *reverse* operation continues to partition the tail sublist until a simple operation of reversing the sublist is found. Reversing a list containing a single node is the reverse of the list is the same list. If the sublist is empty, the reverse of the list is an empty list. These will be included as the two base cases in the overall solution to the problem. The overall solution is achieved by combining the concatenation or joining together the reversed tail sublist and the head sublist.

The solution to this problem requires three auxiliary functions: the first two are needed to divide the problem into smaller problems that are simpler to solve. The third function is needed to combine the partial solutions into the final overall solution to the problem.

The following SPSCL source code includes the implementation of the recursive reverse function and the auxiliary functions. The complete function implementations are stored in file `recrevlist.scl`. The following is a partial listing and shows only the recursive function *reverse*.

```
// Function to reverse the nodes of a list
function reverse return pointer type List
  parameters plist pointer of type List is
structures
    define tailt pointer of type List
    define headt pointer of type List
    define revlist pointer of type List
    define revplist pointer of type List
    define tpnode pointer of type ChNode
begin
    set tpnode = plist->head
```

```
    if plist equal NULL then
       return NULL
    elseif tpnode->next equal NULL then
       return plist
    else
       // partition list into tail and head sublists
       set headt = get_hsub (plist)
       // set tailt = get_tsub(plist)
       tailt = plist
       // reverse tail sublist
       set revplist = reverse (tailt) // recursive call
       // concatenate reversed tail sublist and head
       set revlist = conclist (revplist, headt)
       return revlist
    endif
endfun reverse
```

The following listing shows the commands (on Windows) that traslate the SPSCL source code to C, compile, link, and execute the sample program `TestChList.scl`. This program calls the various functions mention previously that are stored in file `recrevlist.scl`.

```
C:\SPSCL \scl_progs>spscl TestChList.scl
SPSCL  v 1.0 File: TestChList.scl Thu Mar 10 21:22:53 2022
File: TestChList.scl no syntax errors, lines processed: 53

C:\SPSCL \scl_progs>spscl recrevlist.scl
SPSCL  v 1.0 File: recrevlist.scl Thu Mar 10 21:22:59 2022
File: recrevlist.scl no syntax errors, lines processed: 191

C:\Books\SPSCL \scl_progs>gcc -c recrevlist.c
C:\Books\SPSCL \scl_progs>gcc TestChlist.c recrevlist.o

C:\SPSCL \scl_progs>a
Test list reversal
Enter number of nodes to process: 4
Enter character: A
Enter character: B
Enter character: C
Enter character: D
List nodes fist-->last:
Node data:   A
Node data:   B
Node data:   C
Node data:   D

List in reverse order:
List nodes fist-->last:
Node data:   D
Node data:   C
Node data:   B
Node data:   A
```

10.4 Recursive Executions

The executions of the iterative and the recursive solutions to a problem are quite different. For the recursive solution, we must understand its implementation well to be able to develop and use it. We must understand the *runtime stack*.

Recall from previous chapters that a stack is a data structure (a storage structure for data retention and retrieval) based on a last in first out (LIFO) discipline. For example, if a stack is constructed with integer values, the last value that was inserted is at the top of the stack and must be removed before any other can be removed.

When a value is inserted on the top of a stack, the operation is known as *push*, and the stack grows up. When a value is removed from the stack, the operation is known as *pop*, and the stack shrinks. In other words, a stack grows or shrinks depending upon the insertion or removal of items from the stack.

Execution of a computer program involves two broad steps: allocating storage to hold data and then attaching a set of instructions to perform computations on the data held by the allocated storage.

Generally, storage is provided for objects, constants, and functions that have local variables, constants, and parameter variables. Because recursion involves only a function, focus here is only on the associated storage of a function: local variables, constants, and parameter variables. Operating systems create a process for each program to be executed and allocate a portion of primary memory to each process. This block of memory serves as a medium for the runtime stack, which grows when a function is invoked and shrinks when the execution of a function is completed.

During the execution of a process, there may be several functions involved in processing; the first function is always the main function, which invokes other functions if specified by the instructions. Whenever a function is invoked, the runtime system dynamically allocates storage for its local variables, constants, and parameter variables declared within the function from the allocated chunk of memory. In fact, this data is placed in a data structure known as a *frame* or *activation record* and inserted on the stack with a *push* operation. After the execution of the invoked function is completed, the frame created for that function is removed from the top of the stack with *pop* operation.

Obviously, the first frame is always for function *main*, as processing starts with this function. A frame is created to provide storage for each function. The *runtime stack* refers to a stack-type data structure (LIFO) associated with each process provided and maintained by the system. The runtime stack holds data of all the functions that have been invoked but not yet completed processing. A runtime stack grows and shrinks depending on the number of functions involved in processing and the interaction between them.

If function *main* is the only function involved in processing, there is only one frame in the stack with storage for all the variables, locals and constants. If function *main* invokes another function, *A*, then the stack grows only by one frame. After the function *A* completes execution, the memory block allocated to its frame is released and returned to the system.

10.5 Summary

A recursive function definition contains one or more calls to the function being defined. In many cases, the recursive function is very powerful and compact in defining the solution to complex problems. The main disadvantages are that it demands a relatively large amount of memory and time to build stack.

As there is a choice between using iterative and recursive algorithms, so programmers must evaluate the individual situation and make a good decision for their use.

10.6 Key Terms

base cases	recursive case	recursive call
terminating condition	recursive solution	iterative solution
winding phase	unwinding phase	stack
recursive functions	list functions	auxiliary functions
LIFO	push operation	pop operation
Activation frame	activation record	runtime stack

Exercises

Exercise 10.1 Develop a program with a recursive function that computes and displays the summation of cubes of a number n.

Exercise 10.2 Develop a program with a recursive function that finds and displays n natural numbers that are even.

Exercise 10.3 Develop a program with a recursive function that displays the letters in a string in reverse order.

Exercise 10.4 Develop a program with a recursive function that reverses the letters in a string.

Exercise 10.5 Develop a program with a recursive function that checks whether a given string is a palindrome. A palindrome is a string that does not change when it is reversed. For example: “madam,”, “radar,”, and so on.

Exercise 10.6 Develop a program with a recursive function that performs a linear search in an array of integer values.

Exercise 10.7 Develop a program with a recursive function that performs a binary search in an array of integer values.

11. Linked Lists

11.1 Introduction

An array is a lower-level data structure that stores a collection of data items, however the array is static because once it is created, more elements cannot be added or removed. A *linked list* is another low-level data structure that consists of a sequence of data items of the same or similar types and each data item or *node* has one or more links to another node. This data structure is dynamic because the number of data items can change. A linked list can grow and shrink during the execution of the program that is manipulating it.

Linked lists and arrays are considered *low-level* data structures because these are used to implement *higher-level* data structures. Examples of simple higher-level data structures are *stacks* and *queues* and each one exhibits a different behavior implemented by an appropriate algorithm. More advanced and complex higher-level data structures are priority queues, trees, graphs, sets, and others.

This chapter discusses the basic forms of simple linked lists, double-ended linked lists, and multiple linked lists. The operations possible on linked lists and higher-level data structures, such as stacks and queues, implemented with linked lists are also discussed. Abstract data types (ADTs) are discussed and defined for queues and stacks.

11.2 Nodes and Linked List

A linked list is a collection of nodes that are linked or connected. A *node* is a relatively smaller data structure that contains data and one or more link fields that are used to connect the node to one more other nodes in the list. In graphical form, a node may be depicted as a box, which is divided into two types of components:

- A data field or block that stores one or more *data components* of some specified type.
- One or more *link* fields that connect the node to other nodes.

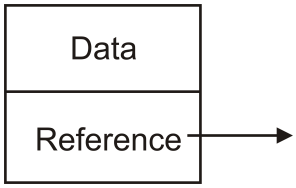

Figure 11.1: Structure of a node.

Figure 11.1 shows a representation of a simple node. It consists of two components: the data component and a reference or link to another node. A linked list is a data structure that consists of a chain (or sequence) of nodes connected in some manner. The last node on a simple linked list has a link with value *NULL*. Figure 11.2 illustrates the general form of a simple linked list. Note that head pointer *Head* points to the first node (the head) of the linked list. The last node (*Node 3* in Figure 11.2) has a link that points nowhere. When comparing linked lists with arrays, the main differences observed are:

- Linked lists are dynamic in size because they can grow and shrink; arrays are static in size.
- In linked lists, nodes are linked by using pointers or references and may include many nodes; whereas, an array is a large block of memory with the elements located contiguously.
- The nodes in a linked list are referenced by relationship not by position; to find a data item, always start from the first item (no direct access). Recall that access to the elements in an array is carried out using an index.

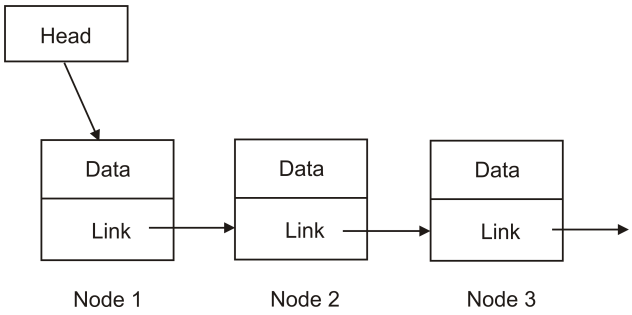

Figure 11.2: A simple linked list.

11.2.1 Nodes

As mentioned previously, a simple node in a linked list has a data block and a link that connects it to another node. These nodes can be located anywhere in memory and do not have to be stored contiguously in memory. The following listing shows the SPSCL code with the definition of the types *NodeType* and *NodePtr*, and the structure for a node. This structure for the node includes a pointer *datablock* as the data component of the node.

```
definetype struct node NodeType
definetype pointer NodeType  NodePtr
struct node is
      define datablock pointer of type DataT
      define link of type NodePtr
endstruct node
```

A general data block is defined in the structure and depends on the application. This is normally defined with several variables, each of a different type. A program declares a variable of type *DataT* and assigns the values of each component of the data block. The following listing in SPSCL and C illustrate this.

```
definetype struct Datablock DataT
struct Datablock  is
      define name array [31] of type char
      define age of type integer
      definet jobcode of type integer
endstruct Datablock
. . .
  define pdata pointer of type DataT
  // set pdata = malloc(sizeof (DataT))
  create pdata type DataT
  call strcpy using name of pdata, "Joseph Hunt"
  set age of pdata = 45
  set jobcode of pdata = 6524
```

In the C language:

```
typedef struct Datablock DataT;
struct Datablock {    // data block type
     char name [31];
     int age;
     int jobcode;
};

DataT *pdata;  // declare a pointer data block
pdata = malloc(sizeof(DataT));
strcpy(pdata->name, "Joseph Hunt");
pdata->age = 45;
pdata->jobcode = 6524;
```

To create a new node with the structure previously defined for a node, the keyword **create** is used. The pointer variable of type *NodePtr* is declared, dynamic memory allocation is performed for the size of the node, and its address is assigned to the pointer. The following lines of statements declare a pointer *newPtr* of type *NodeType*, allocate memory for a new node, assign the address of the node to *newPtr*, and assign the value of the pointer *datablock* of the node.

```
define NodePtr of type newPtr
. . .
create newPtr type NodeType
set datablock of newPtr = pdata
set link of newPtr = NULL
```

The corresponding code using the C language is:

```
NodePtr newPtr;
newPtr = malloc (sizeof(NodeType));
newPtr->datablock = pdata;
newPtr->link = NULL;
```

11.2.2 Manipulating Linked Lists

To complete a *specification* or *interface*, a set of operations is specified for to creating and manipulating the linked list. Some of the basic operations defined on a simple linked list are:

- Create an empty linked list
- Insert a new node at the front of the linked list
- Insert a new node at the back of the linked list
- Insert a new node at a specified position in the linked list
- Get a copy of the data in the node at the front of the linked list
- Get a copy of the data in the node at a specified position in the linked list
- Remove the node at the front of the linked list
- Remove the node at the back of the linked list
- Remove the node at a specified position in the linked list
- Traverse the list to display all the data in the nodes of the linked list Check whether the linked list is full
- Check whether the linked list is empty
- Find a node of the linked list that contains a specified data item

As mentioned previously, the pointer variable *Head* always points to the first node of the linked list. This node is also known as the *head node* because it is the front of the linked list. If the list is empty, then the value of *Head* is *NULL*. To create an empty list, all that is needed is the assignment as follows:

```
define Head of type Nodeptr
define numnodes of type integer  // current number of nodes
. . .
set Head = NULL
set numnodes = 0
```

To check if a list is empty, the value of the head pointer *Head* needs to be compared for equality with the constant value *NULL*.

```
if  Head equal NULL then
   . . .
endif
```

A function can be defined with name *isEmpty* that checks whether the list is empty by examining the value of the head pointer *Head*, which is a reference to the first node in the list. The value of this pointer has a value *NULL* when the list is empty.

To insert a new node as the first node of an *empty* linked list, then the head of the list should point to this node. Assuming that pointer variable *newPtr* points to a newly-created node, the following assignment statement inserts the node and it becomes the head of the linked list:

```
set Head = newPtr
```

To insert a new node at the head of a *non-empty* linked list, the head pointer *Head* is pointing at the current first node, so the value of *Head* will be copied to the link component of the new node. The new value of the head pointer *Head* will be value of the pointer to the new node. After creating and initializing a new node, the following statements insert the new node in the front of the linked list.

```
set link of newPtr = Head
set Head = newPtr
```

In the C language:

```
newPtr->link = Head;
Head = newPtr;
```

To remove and delete the node at the front of the linked list, the head pointer *H* will point to the second node. The node to be removed has to be destroyed after its data have been copied to another variable.

```
define cnodeptr of type NodePtr
. . .
set cnodeptr = Head         // pointer to current head node
set Head = link of Head    // pointer to second node
set pdata = datablock of cnodeptr  // get data
call free using cnodeptr    // de-allocate memory, destroy node
```

In the C language:

```
NodePtr cnodeptr;
cnodeptr = Head;       // pointer to current head node
Head = Head->link;     // pointer to second node
pdata = cnodeptr->datablock; // get data
free(cnodeptr);        // de-allocate memory, destroy node
```

Traversing a linked list involves accessing every node in the list by following the links to the next node until the last node. Recall that the value of the link of the last node is *NULL*. The following portion of code traverses a linked list to display the data of every node.

```
set cnodeptr = Head                  // point to first node
while  cnodeptr not equal NULL  do       // traverse list
     call display_data using cnodeptr        // display data of node
     set cnodeptr = link of cnodeptr         // point to next node
endwhile
```

In the C language:

```
cnodeptr = Head;   // point to first node
while ( cnodeptr != NULL ) {       // traverse list
      display_data (cnodeptr);       // display data of node
      cnodeptr = cnodeptr->link;   // point to next node
}
```

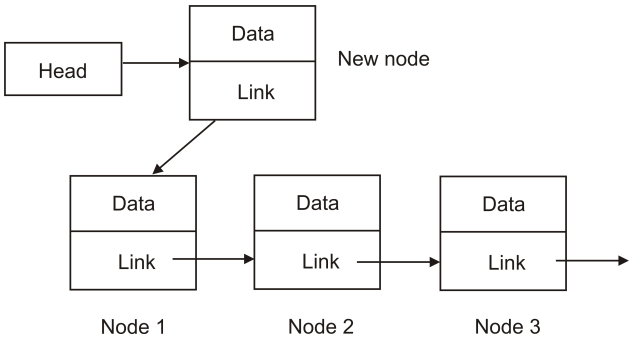

Figure 11.3: A new node inserted in the front of a linked list.

A node can be inserted to the linked list at the front, at the back, or in any other place specified. For a simple linked list, insertion at the front of the list is simpler. Function *insert_front* can be defined that creates a new node and inserts the new node to the front of the linked list. Figure 11.3 shows the insertion of a new node to the front of the list.

More flexibility is obtained by having the operation to insert a node at a specified position in the linked list. For example insert a new node after current node 2. Figure 11.4 illustrates changing the links so that a new node is inserted after node 2. Note that that in this figure, the pointer to the first node is H.

11.2.3 Example of Manipulating a Linked List

Listing 11.1 is the source code of a program that includes a global data declaration that defines the structure of a node, pointer types of the node structure, and various functions that implement the operations discussed on a linked list. Function *main* calls these functions to create and manipulate a linked list. The SPSCL source code of this program is stored in file `datablist.scl` and

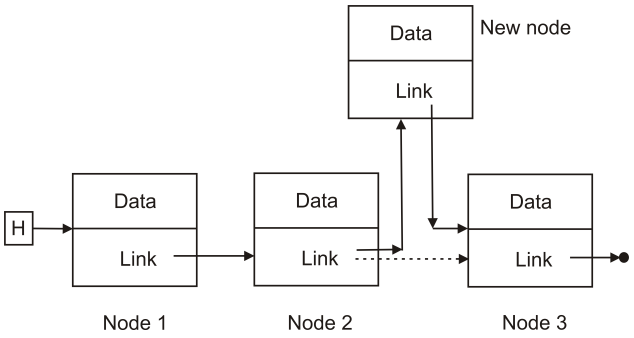

Figure 11.4: A new node inserted after node 2.

`datalist.c`. The files that define the linked list structures are: the interface stored in `linkedh.scl` and the implementation of the functions stored in file `linkedlib.scl`.

Listing 11.1: SPSCL program for creating and manipulating a linked list.

```
description
  datablist.scl
  This program shows how to create nodes and various ways
  to link them in a linked list using data blocks.
  The functions of linked list are implemented in library 'linkedlib.scl'.
  J. M. Garrido Feb. 2022
  */
import <stdio.h>
import <stdlib.h>
import <string.h>
import "linkedh.c"

specifications

struct Datablock is              // data block type
     define stname array [31] of type char
     define age of type integer
     define jobcode of type integer
endstruct Datablock

definetype struct Datablock DataT

forward declarations

function make_dblock return pointer type DataT parameters
      nname pointer of type char, nage of type integer, njobcode of type integer
function display_data return type void parameters pdata pointer of type DataT
function traverse_display return type void parameters plist pointer of type listT

implementations

function main is
```

```
// local data definitions
variables
   define max_nodes of type integer
   define cname array [30] of type char
   define tnum of type integer
structures
   // define mlist of type listT
   define clist pointer of type listT
   define dblock pointer of type DataT
   define nodePtr of type NodePtr
begin
   // set clist = address mlist
   set clist = NULL
   create clist type listT
   call init_list using clist, 40, "Data list"      // create empty linked list
   display "created list"
   // create data block
   set dblock = make_dblock("John Doe", 38, 1523)
   display "created a data block"
   // create node with data block
   create nodePtr type NodeType
   display "created a node with a data block"
   set datablock of nodePtr = dblock
   // now node to front of list
   display "Inserting node to front of list"
   call insert_front using clist, nodePtr
   set tnum = numnodes of clist
   display "A num nodes: ", tnum
   call printf using "clist %p \n", clist
   // create another data block 2
   set dblock = make_dblock("Joseph Hunt", 59, 1783)
   // create node with data block
   create nodePtr type NodeType

   set datablock of nodePtr = dblock
   call printf using "nodePtr %p \n", nodePtr
   call insert_front using clist, nodePtr
   // data block 3
   set dblock = make_dblock("William Dash", 49, 2581)
   create nodePtr type NodeType
   set datablock of nodePtr = dblock
   display "To insert front"
   call insert_front using clist, nodePtr

   call traverse_display using clist    // display all blocks in list
   display "Removing front: "
   set nodePtr = remove_front(clist)
   set dblock = datablock of nodePtr
   call display_data using dblock

   // node and data block to be deleted
```

```
    destroy nodePtr
    destroy dblock
    call traverse_display using clist    // display all blocks in list
    set dblock = make_dblock("David Winemaker", 48, 1854)
    call display_data using dblock
    create nodePtr type NodeType
    set datablock of nodePtr = dblock
    display "Inserting node to pos 2: "
    call insert_node using clist, nodePtr, 2
    call traverse_display using clist
    set dblock = make_dblock("Mary Washington", 54, 1457)
    display "Inserting 3: "
    call display_data using dblock
    create nodePtr type NodeType
    set nodePtr->datablock = dblock
    call insert_node using clist, nodePtr, 3
    call traverse_display using clist

    display "Inserting front: "
    set dblock = make_dblock("Ann Smith", 44, 1568)
    call display_data using dblock

    create nodePtr type NodeType
    set datablock of nodePtr = dblock

    call insert_front using clist, nodePtr
    call traverse_display using clist

    display "Remove node 2 "
    set nodePtr = remove_node(clist, 2)
    set dblock = datablock of nodePtr
    call display_data using dblock
    call traverse_display using clist

    destroy nodePtr
    destroy dblock

    display "Removing last node"
    set nodePtr = remove_last(clist)
    set dblock = datablock of nodePtr
    call display_data using dblock

    call traverse_display using clist
    destroy nodePtr
    destroy dblock
    return 0
endfun main
//
function make_dblock return pointer type DataT
    parameters pname pointer of type char, page of type integer,
       pjobcode of type integer is
```

```
structures
    define ldatab pointer of type DataT
begin
    // allocate memory for datablock
    create ldatab type DataT

    call strcpy using stname of ldatab, pname
    set age of ldatab = page
    set jobcode of ldatab = pjobcode
    return ldatab
endfun make_dblock
//
// display data block
function display_data return type void parameters pblock pointer of type DataT is
variables
     define lname array[30] of type char
     define ljobc of type integer
     define lage of type integer
begin
    call strcpy using lname, pblock->stname
    set ljobc = jobcode of pblock
    set lage = age of pblock
    display "Data name: ", lname
    display "Data age: ", lage
    display "Data job code: ", ljobc
endfun display_data
//
function traverse_display parameters plist pointer of type listT is
variables
   define i of type integer
structures
   define lblock pointer of type DataT
   define cnodePtr of type NodePtr
begin
   display "Traverse and display data in list"
   set i = numnodes of plist
   display "(traverse_display) Num nodes ", i
   if numnodes of plist <= 0 then
       return
   endif
   set cnodePtr = get_front(plist)       // get node at front of list
   set lblock = datablock of cnodePtr    // get data block from node
   call display_data using lblock
   for i = 2 to i <= numnodes of plist do
       display "display the next node "
       set cnodePtr = get_next(plist)        // get next node from list
       set lblock = datablock of cnodePtr    // get data block
       call display_data using lblock
   endfor
endfun traverse_display
```

The following listing includes the Linux shell commands to translate the SPSCL to C, compile, link, and execute the program `datablist.c`, and get the results produced.

```
$ ./spscl datablist.scl
$ ./spscl linkedlib.scl
$ gcc -Wall datablist.c linkedlib.c
$ ./a.out
List all nodes in linked list
Data from node: William Dash 49 2581
Data from node: Joseph Hunt 59 1783
Data from node: John Doe 38 1523
Data removed front node: William Dash 49 2581
insert nodes at pos 2, 3, and front
List all nodes in linked list
Data from node: Ann Smith 44 1568
Data from node: Joseph Hunt 59 1783
Data from node: John Doe 38 1523
Data from node: David Winemaker 48 1854
Data from node: Mary Washington 54 1457
Data removed node 2: Joseph Hunt 59 1783
List all nodes in linked list
Data from node: Ann Smith 44 1568
Data from node: John Doe 38 1523
Data from node: David Winemaker 48 1854
Data from node: Mary Washington 54 1457
Removing last node
List all nodes in linked list
Data from node: Ann Smith 44 1568
Data from node: John Doe 38 1523
Data from node: David Winemaker 48 1854
```

11.3 Linked List with Two Ends

The linked lists discussed previously have only one end, which points to the first node, and this pointer is also known as the head of the linked list. In addition to the head node, providing a pointer to the last node gives the linked list more flexibility. With two ends, a linked list has two pointers: a pointer to the first node – the *Head* or front of the list, and a pointer to the last node – the *Tail* of the linked list. Figure 11.5 illustrates a linked list with a head pointer *H* and a tail pointer *T*.

With a pointer to the last node (the *Tail*), in addition to the pointer to the first node (the *Head*), the linked list now provides the ability to directly add a new node to the back of the linked list without traversing it from the front. In a similar manner, the last node of a linked list can be removed without traversing it from the front.

To create an empty linked list with two ends, two pointers have to be declared and initialized, the head and the tail of the list. The structure of a node used previously does not change for a linked list with two ends. The structure of the linked list with two ends is:

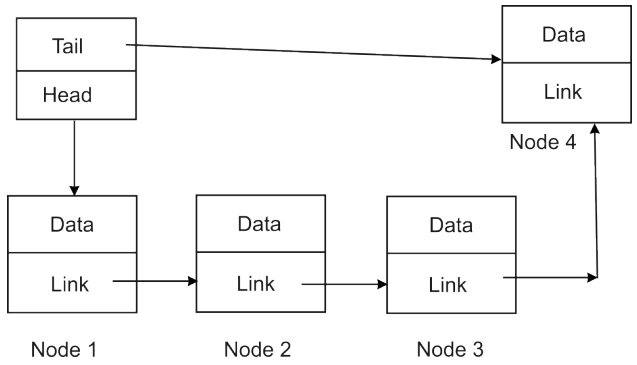

Figure 11.5: A linked list with Head and Tail nodes.

```
struct llist is
   define numnodes of type integer
   define Head of type NodePtr     // head of list
   define Tail of type NodePtr     // tail of list
endstruct llist
```

Linked lists with two ends are very useful and convenient for implementing higher-level data structures such as *queues*.

11.4 Double-Linked Lists

Linked lists previously described have nodes with only one link, a pointer to the next node, and can only traverse the linked list in one direction, starting at the front and toward the tail of the list. To further enhance the flexibility of a linked list, a second link is included in the definition of the nodes. This second link is a pointer to the *previous* node. These linked lists are also known as *doubly linked lists*. Figure 11.6 illustrates the general form of a linked list with nodes that have two links: a pointer to the next node and a pointer to the previous node.

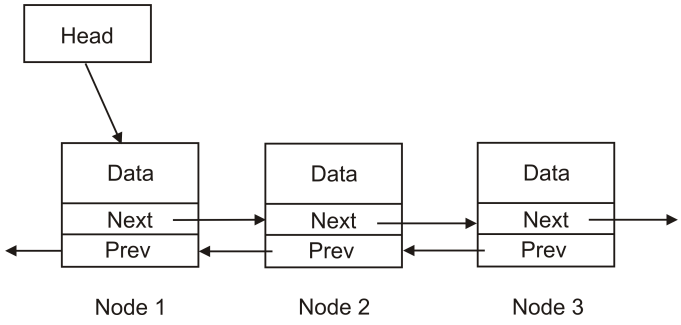

Figure 11.6: A linked list with two links per node.

The following SPSCL statements define a node type with two links, *next* that points to the next node

in the linked list and *prev* that points to the previous node in the linked list. This is an example of a node structure with more than one link components, in addition to the data block.

```
definetype struct node NodeType
definetype pointer NodeType  NodePtr
struct node is
     define datablock pointer of type DataT
     define next of type NodePtr       // pointer to next node
     define prev of type NodePtr       // pointer to previous node
endstruct node
```

The linked list library, *linked2lib*, defines double-linked lists with two ends. The interface is stored in file `link2h.scl` and the implementation is stored in file `linked2lib.scl`. The following listing shows the SPSCL code in the interface (header) file of this library.

```
// interface for double-linked list with two ends
// file: link2h.scl
import "scl.h"

specifications

definetype struct Datablock DataT
definetype struct node NodeType
definetype pointer NodeType NodePtr
struct node is       // node type
     define datablock pointer of type DataT
     define next pointer of type NodeType  // pt to next node
     define prev pointer of type NodeType  // pt to previous node
endstruct node
//
definetype struct llist listT
struct llist is
     define numnodes of type integer
     define maxnodes of type integer
     define lname array [30] of type char
     define Head of type NodePtr  // head of list
     define Tail of type NodePtr  // tail of list
     define current of type NodePtr
endstruct llist
//
forward declarations

function init_list parameters
    plist pointer of type listT,
    maxn of type integer, pmame pointer of type char
function insert_front parameters
    plist pointer of type listT,
    cdata pointer of type DataT
function remove_front return pointer of type DataT
   parameters plist pointer of type listT
function traverse_display parameters
  plist pointer of type listT
function empty_list return type bool parameters
  plist pointer of type listT
function full_list return type bool parameters
  plist pointer of type listT
function insert_node parameters plist pointer of type listT,
  ndata pointer of type DataT, pos of type integer
function insert_last parameters plist pointer of type listT,
     ndata pointer of type DataT
function remove_node return pointer of type DataT
    parameters plist pointer of type listT, pos of type integer
function get_data return pointer of type DataT
   parameters plist pointer of type listT, pos of type integer
function get_front return pointer of type DataT parameters
   plist pointer of type listT    // get data from first node
```

```
function get_next return pointer of type DataT parameters
   plist pointer of type listT    // copy data from next node
function remove_last return pointer of type DataT
  parameters plist pointer of type listT  // remove last node
function listSize return type integer
   parameters plist pointer of type listT
```

Summary

Linked lists are dynamic data structures for storing a sequence of nodes. The data is part of the structure known as a node. After the list has been declared and created, it can grow or shrink by adding or removing nodes. Linked lists can have one or two ends. Each node may have one or more links. A link is a pointer to another node of the linked list. Abstract data types (ADTs) of higher-level data structures are implemented with linked lists. Queues and stacks are examples of higher-level data structures and can be implemented with arrays or with linked lists.

Key Terms

linked lists	nodes	links
dynamic structure	low-level structures	high-level structures
next	previous	queues
stacks	data block	abstract data type

Exercises

Exercise 11.1 Design and implement a program that uses a linked list in which each node represents a flight stop in a route to a destination. The data in each node is the airport code (an integer number). The airline can add or delete intermediate flight stops. It can also calculate and display the number of stops from the starting airport to the destination airport.

Exercise 11.2 The *linked2* library for linked lists defines nodes with only one link and with only one end. Develop a *stack2* library that uses double-linked lists with two ends. Develop a complete program that creates and manipulates three stacks.

Exercise 11.3 The *linked2* library for linked lists defines nodes with only one link and with only one end. Add a function that searches for a node with a given name and returns a copy of the data block. Develop a complete program that uses the modified library.

Exercise 11.4 Develop a library that defines an array of N linked lists. This can be used if the linked list is used to implement a priority queue with N different priorities. A node will have an additional component, which is the priority defined by an integer variable. Develop a complete SPSCL program.

12. Data Structures

12.1 Introduction to Data structures

A data structure is a resource used to store and organize data in the computer's memory, so that the data can be manipulated efficiently. Data may be arranged in many different ways such first, it must be well and conveniently organized in structure to reflect the actual relationships of the data with the real world object. Second, the interface to manipulate the data should be simple enough so that any module can efficiently process the data each time it is necessary. Data structures are subdivided in various categories:

- Lower-level data structures or implementation data structures; these are arrays and linked lists
- Higher-level data structures; these are application oriented and are subdivided into:
  - Linear Data Structure
  - Non-linear Data Structure

Lower-level data structures are used to define and implement the higher-level data structures. A data structure is said to be linear if its elements combine to form any specific order. The common examples of higher-level data structure that are linear include:

- Queues
- Stacks

Non-linear data structures are mostly used for representing data that contains a hierarchical relationship among various elements. Examples of higher level data structures that are non-linear include:

- Family of trees
- Graphs

In trees, data often contain a hierarchical relationship among various elements. The data structure that reflects this relationship is termed as rooted tree graph or a tree. In graphs, data often hold a relationship between the pairs of elements which is not necessarily following the hierarchical structure.

12.2 Modular Design

Data structures must be developed by applying a good modular design approach. In software development, it is important to design every module correctly in order to manage complexity. One of the fundamental principles in modular design is: the module specification must be separate from the module implementation. Several other principles should also be applied, such as:

- *Strong cohesion* refers to the level of intra-dependency among the components of a software module and should be high. The components of a module are functionally grouped into a logical unit and they are related together as a logical unit – this promotes flexibility and re-usability. This is related to the principle of *separation of concern*.
- *Weak coupling* refers to how closely the modules are connected to each other determines the degree of interdependence that exists between software modules and should be low or should be minimized.

The *interface* of a data structure, also known as an abstract data type (ADT), is a specification that indicates:

- What data can be stored in the data structure
- What operations are available to manipulate the data

The *implementation* of the data structure is the representation of the data and includes indication how is the data stored and the algorithms that support the various operations to manipulate the data.

12.3 Abstract Data Types

An abstract data type (ADT) is a concept used in software development that involves modularity, abstraction, and data hiding. An ADT is a collection of data definitions and well-defined operations that allows programmers to apply the ADT without being concerned about the lower-level implementations. The programmers' perceptions are at a high-level of abstraction and they should mainly understand the function specifications (function prototypes). The lower-level details, which are the function implementations (function definitions), will normally be hidden from the programmers.

In the case of a linked list, the possible operations on a linked list in the form of function prototypes are visible to all users so they can manipulate the linked list. In SPSCL and C, function prototypes are normally stored in header files. The various function implementations are stored in a separate file.

12.4 Higher-level Linear Data Structures

Higher-level linear data structures are those used directly in problem solving and normally implemented by lower-level data structures. This section discusses and describes the structure and operations of two simple and widely-known higher-level linear data structures: *queues* and *stacks*. These can be implemented with low-level data structures such as arrays and linked lists.

12.4.1 Stacks

A stack is a higher-level dynamic data structure that is linear and that stores a collection of data items. The stack is a dynamic data structure because the number of nodes can grow and decrease. The data in a stack may be a data block with variables of a simple type, pointers to data objects, or a combination of these. Each node in a stack includes a data block and one or more links. This data structure is also known as a last in first out (LIFO) data structure. A stack has only one end: the *top* of the stack. The main characteristics of a stack are:

- Data blocks can only be inserted at the top of the stack (TOS)
- Data blocks can only be removed from the top of the stack
- Data blocks are removed in reverse order from that in which they are inserted into the stack. This is why a stack is also known as a last in and first out (LIFO) data structure.

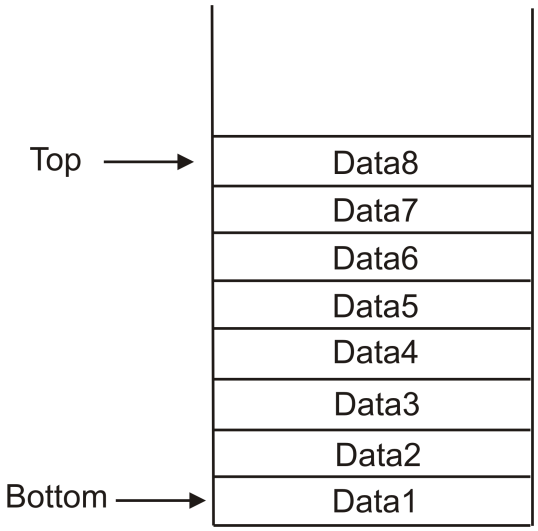

Figure 12.1: A stack as a dynamic data structure.

Figure 12.1 illustrates the form of a stack. It shows the top of the stack as the insertion point and the removal point. The operations that manipulate a stack ADT are:

- *create_stack*, create an empty stack.
- *push*, inserts a new node to the top of the stack.
- *pop*, removes the node from the top of the stack.
- *sEmpty*, returns true if the stack is empty; otherwise returns false.
- *sFull*, returns true if the stack is full; otherwise returns false.

- *top*, returns a copy of the data block at the top of the stack without removing the node from the stack.
- *stackSize*, returns the number of nodes currently in the stack.

The most direct way to implement a stack is with a single-list linked list in which insertions and deletions are performed at the front of the linked list. The linked list library *linkedlib* already implemented is used to implement a stack library, which is stored in files `stackh.scl` and `stacklib.scl`. Listing 12.1 shows the source code of the interface or stack ADT in the library that is used to create and manipulate a stack.

Listing 12.1: Source code of the stack ADT library.

```
// interface for the stack
// using the single-linked list library
// in 'linkedlib.scl' and 'linkedh.scl'
import "linkedh.c"

specifications

definetype struct llist stackType

forward declarations

function create_stack parameters stack pointer of type stackType,
    maxn of type integer,  sname pointer of type char
function push parameters stack pointer of type stackType,
    pnode of type NodePtr

function pop return type NodePtr parameters
     stack pointer of type stackType
function top return type NodePtr parameters
     stack pointer of type stackType
function sEmpty return type bool parameters
     stack pointer of type stackType
function sFull return type bool parameter
     s stack pointer of type stackType
function stackSize return type integer parameters
     stack pointer of type stackType
```

12.4.2 Applications of Stacks

There are many application that use stacks. Two interesting applications are: conversion of expressions to postfix notation and evaluation of postfix expressions. A compiler will normally include functions with very similar algorithms.

Conversion of Arithmetic Expressions to Postfix

Infix, postfix, and prefix notations are three notations that are used when writing algebraic expressions. Writing an arithmetic expression using infix notation, the operator is placed in between the operands. For example, A+B; the plus operator is placed between the two operands A and B. Using infix

notation, compilers need more processing to evaluate the expression. Compilers can process these expressions more efficiently when these are written using prefix and postfix notations.

Postfix notation was developed as a parenthesis-free prefix notation (also known as Polish notation) and a postfix notation, which is better known as Reverse Polish Notation or RPN. In postfix notation, the operator is placed after the operands. For example, if an expression is written as $A+B$ in infix notation, the same expression can be written as $AB+$ in postfix notation. The order of evaluation of a postfix expression is always from left to right.

The expression (A + B) * C is converted to: $AB+C*$, which is in the postfix notation. In algebra, multiplication and division have precedence over addition and subtraction. The following is a modified initial infix expression: $A+B*C$. This is converted to $ABC*+$. The following is a slightly more complex example:

Convert the expression $P+Q*R-S+T$ to postfix. The resulting postfix expression is: $PQR*+ST+-$. For simple expressions, the priority of operators are (from highest to lowest):

1. Multiplication and division
2. Addition and subtraction
3. Parenthesis

The following is an informal description of an algorithm for conversion of expression from infix to postfix:

```
Create a stack
while there are terms in the expression loop
   if term is left parenthesis then
        push term to stack
   elseif  term is right parenthesis then
        pop term from stack
        while term not left parenthesis loop
            append term to output expression
            pop term from stack
       endwhile
   elsif term is operator then
       compare precedence of operator at TOS
       get term at TOS to term_top
       while stack not empty and precedence of term <= term_top loop
          pop term from stack
          append term to output expression
          get term at TOS to term_top
      endwhile
      push term to stack
   else
      append term to output expression
   endif
endwhile
while stack not empty loop
  pop term from stack
```

```
    append term to output expression
 endwhile
```

A complete SPSCL program that converts infix arithmetic expressions to postfix notation is stored in file `infix_postfix.scl`. This progtam uses the stack ADT and its function implementations developed previously.

Evaluating Postfix Expressions

To evaluate a postfix expression, a stack is also used. Given the postfix expression: $XYZ+*$, the goal is to calculate its resulting value.

The operands are pushed into the stack. When an operator, we pop the two operands at the top of the stack and perform the operation. We then push the value back into the stack to be used later. Figure 3-16 traces the operation of our expression. (Note that we push the operand values into the stack, not the operand names. We therefore use the values in the figure.)

When the expression has been completely evaluated, its value is in the stack. The following is the algorithm in high-level pseudo-code.

```
Algorithm postFix_eval (expr)
The algorithm evaluates a postfix expression and returns resulting value.
createStack (stack)
loop (for each character symbol from input)
if (character is operand)
   pushStack (stack, character)
else
   popStack (stack, oper2)
   popStack (stack, oper1)
   operator = character
   set value calculate (oper1, operator, oper2)
   pushStack (stack, value)
end if
end loop
popStack (stack, result)
return (result)
end postFix_eval
```

A complete SPSCL program that implements this algorithm and that uses the stack ADT is stored in file `postfix_eval.scl`.

12.4.3 Queues

A queue is another higher level data structure that is linear. It stores a collection of data items or nodes and has two ends: the *head* and the *tail*. The queue is also a dynamic data structure because the number of nodes can grow and decrease. The rules for basic behavior of a queue are:

- Data items can only be inserted at the tail of the queue
- Data items can only be removed from the head of the queue

- Data items must be removed in the same order as they were inserted into the queue. This data structure is also known as a first in and first out (FIFO) data structure.

Figure 12.2 illustrates the form of a queue. It shows the insertion point at the tail and the removal point at the head of the queue. The operations that manipulate a queue are:

- *isEmpty*, returns true if the queue is empty; otherwise returns false.
- *isFull*, returns true if the queue is full; otherwise returns false.
- *copyHead*, returns a copy of the data object at the head of the queue without removing the object from the queue.
- *removeHead*, removes the head item from the queue
- *insertTail*, inserts a new data item into the tail of the queue.
- *queueSize*, returns the number of data items currently in the queue.

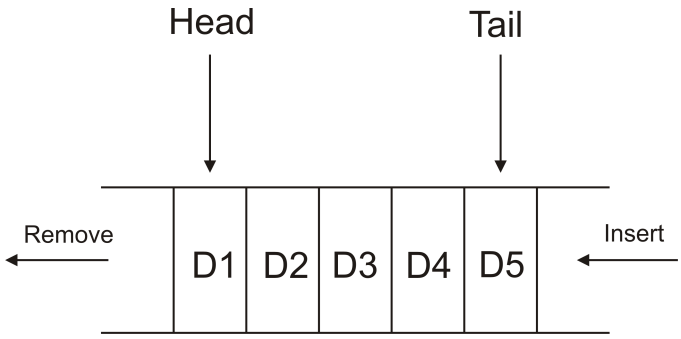

Figure 12.2: A queue as a dynamic data structure.

Queues can be implemented with single-linked lists, but are more efficiently or conveniently implemented with double-linked lists with two ends. The linked list library *linked2* defines double-linked lists with two ends and it is stored in files `linked2h.scl` and `linked2lib.scl`; and is used mainly to implement queues.

Listing 12.2 shows the source code of the ADT for a queue. It shows the dara definitions and the function prototypes for the queue. The queue ADT is is stored in files `queueh.scl` and the implementation file is `queuelib.scl`.

Listing 12.2: SPSCL source code of the queue ADT.

```
// interface for a queue using double-linked list with two ends
// file: queueh.scl
import "link2h.c"
specifications

definetype struct llist queueT
definetype struct Datablock DataT
definetype struct node NodeType
definetype pointer NodeType NodePtr

forward declarations
```

```
function create_queue parameters queue pointer of type queueT,
  maxn of type integer, sname pointer of type char
function enqueue parameters queue pointer of type queueT,
    cdata pointer of type DataT
function dequeue return pointer of type DataT parameters
    queue pointer of type queueT
// get a copy of node at head of queue
function head return pointer of type DataT parameters
    queue pointer of type queueT
function sEmpty return type bool parameters
    queue pointer of type queueT
function sFull return type bool parameters
    queue pointer of type queueT
function queueSize return type integer parameters
    queue pointer of type queueT
```

An example of using the queue interface and library is stored in file `dataqueue.scl`. The following listing shows the Linux shell commands that compile, link, and execute the program. The results produced by the program execution are also shown.

```
$ ./spscl dataqueue.scl
$ gcc -Wall dataqueue.c queuelib.o linked2lib.o
$ ./a.out
Enqueue data: Data: John Doe 38 1523
Enqueue data: Data: Joseph Hunt 59 1783
Enqueue data: Data: William Dash 49 2581
Queue size: queueA 3
Copy Head data from queue: queueA
Data: John Doe 38 1523
Enqueue: Data: David Winemaker 48 1854
Size of queue: 4
Dequeue from queue: Data: John Doe 38 1523
```

12.4.4 Applications of Queues

Queues are found in operating systems, networks, and many other areas. In all client-server model, queues are used. In a computer system, a queue is needed to process jobs and for system services such as print spools.

A very common application of queues is a simple first-come-first-serve (FCFS) model. This is used in simulation of computer systems and many other systems to study its basic operations using various performance metrics.

The following is an example of a Car-Wash System Model, using a FCFS model approach.

- Arriving cars join the tail of a queue to wait for service.
- The car at the head of the queue is the one that is next serviced by the server (Carwash machine).

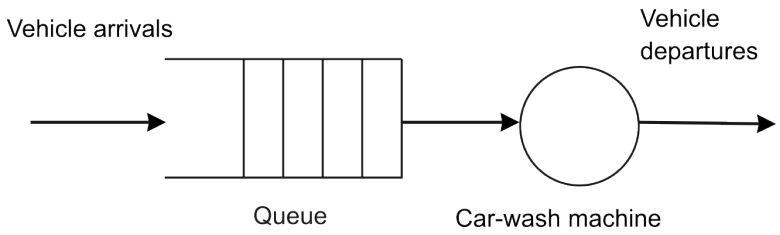

Figure 12.3: Graphical view of the Carwash system

- After the car wash service is completed, the car leaves the system.
- Cars are considered the customers of the system, as they are the entities requesting service from the server.

12.5 Priority Queues

A priority queue is a data structure in which each node include a priority component. The priority of the node determines the order in which the nodes will be processed. The order of how nodes are processed are:

- A node with a higher priority is processed before a node with a lower priority.
- Two nodes with the same priority are processed in order first-come-first-served (FCFS).

In a priority queue when a node has to be removed from the queue, the one with the highest-priority is retrieved first. The priority are assigned according to the type of application. Priority queues are widely used in various application areas, such as simulation models and operating systems.

Implementations using arrays combined with linked lists will result in an approximate time complexity of O(log n). An array of pointers to linked lists is a good practical solution for implementation. Every array pointer corresponds to a specific priority and there are several simple linked lists, one for each priority. Insertion and deletion of nodes are performed in the regular manner for queues,

12.6 Time Complexity of Algorithms

The efficiency of algorithms depend on several factors, mainly on the time complexity of the computations.The time complexity of algorithms depend on how the number of computations varies with the size of the data.

12.6.1 Constant Time

If a function does not have loops or recursions, its efficiency only depends on the number of instructions to execute. Therefore, the function is said to have constant time complexity.

12.6.2 Linear Time

Functions that have loops or recursion vary widely in efficiency and therefore their time complexity will be expressed as function of the size of the data to process.

In the case of the following simple loop, the number of times the loop is repeated is n. Given that j, and *sum* are integers, the time complexity of this segment of code is directly proportional to the number of iterations, (n).

```
for j = 0 to j < n do
      set sum = sum + 1
endfor
```

12.6.3 Simple Logarithmic Time

The following example includes a slightly more complex loop, the counter variable j is multiplied by 2 on each iteration. Therefore, the number of iterations is less than in the previous example. The time complexity is $(\log n)$.

```
set j= 0
while j < n do
    set sum = sum + j
    set j = j * 2
endwhile
```

12.6.4 Linear Logarithmic Time

With algorithms that have *nested loops*, the total number of iterations is the product of the iterations in the outer loop times the number of iterations of the inner loop. The following segment of code shows two nested loops, node that in the inner loop i is multiplied by 2. The time complexity is $(n \log n)$.

```
set j= 0
while j < n do
   set i = 0
   while i < n do
       set sum = sum + j
       set i = i * 2
   endwhile
   increment j
endwhile
```

12.6.5 Quadratic Time

Logarithms that have nested loops that have the same upper bound for the counter variable. The number of iterations of each loop is the same. The following segment of code shows these nested loops. The time complexity is (n^2).

```
set j= 0
while j < n do
   set i = 0
   while i < n do
```

```
        set sum = sum + j
        increment i
    endwhile
    increment j
endwhile
```

12.6.6 More Time Complexities

Other algorithms exhibit time complexities that grow much faster than the previous ones, as the size of the data increases. The most commonly known time complexities in this group are:

- Polynomial time, expressed as (n^k), where k is a constant.
- Exponential time, expressed as (k^n), where is k is a constant.

12.6.7 Big O Notation

The big O notation is used to express the asymptotic behavior of functions that refer to the growth of the number of computations with respect to increase in the size of the data. For example, an algorithm that has quadratic time complexity is denoted as the rate of growth of the function and is in the order of n^2, or $O(n^2)$.

With a search algorithm, the efficiency of the algorithm is determined by the number of operations, which compare the element values in the array, with respect to the size of the array. The average number of comparisons with linear search for an array with N elements is $N/2$, and if the element is not found, the number of comparisons is N. With binary search, the growth rate of number of comparisons as the number of elements (N) in the list grow is $\log N$. Using the big O notation for binary search is $O(\log N)$.

12.7 Summary

A data structure is an organized data storage in memory and requires one or more algorithms for efficient manipulation of the data. Applications will use the interface of the data structure. Abstract data types (ADTs) of higher-level data structures are implemented with linked lists. Queues and stacks are examples of higher-level data structures and can be implemented with arrays or by linked lists. The time complexity of an algorithm is expressed using the big O notation. This describes the rate of growth of the number of computations with respect to the program size.

Key Terms

stacks	queues	data structures
dynamic structure	low-level data structures	high-level data structures
next	previous	time complexity
big-O notation	data block	abstract data type

Exercises

Exercise 12.1 Develop a program with algorithm that copies the contents of one stack into another. The algorithm passes two stacks, the source stack and the destination stack. The order of the stacks

must be identical.

Exercise 12.2 Develop a program with a new stack ADT function that concatenates the contents of one stack on top of another. Test the function by writing a program that uses the ADT to create two stacks and prints them. It should then concatenate the stacks and print the resulting stack.

Exercise 12.3 Develop a library that defines an array of *N* linked lists. This can be used if the linked list is used to implement a priority queue with *N* different priorities. A node will have an additional component, which is the priority defined by an integer variable. Develop a complete C program.

Exercise 12.4 Develop a program using the algorithms in the queue ADT, that copies the contents of one queue to another.

Exercise 12.5 Develop a program using another queue ADT function that copies a queue using the queue ADT.

Exercise 12.6 Develop a program using the queue ADT, that concatenates two queues.

Exercise 12.7 Develop a program that implements the previous problem.

Exercise 12.8 Develop a program that calculates and prints the sum and the average of the integers in a queue without changing the contents of the queue.

Exercise 12.9 Indicate the order of the following expressions that represent time complexity, from the most efficient to the least efficient:

1. $2n$
2. $n!$
3. n^5
4. $10,000$
5. $n\log(n)$

Exercise 12.10 Determine the big-O notation for the following:

1. $7n7/2+n27$
2. $6log(n)+9n$
3. $3n4+n\log(n)$
4. $5n2+n3/2$

Exercise 12.11 Determine the time complexity of the following program segment:

```
for i = 1 to n do
   display "value of i: ", i
endfor
```

Exercise 12.12 Determine the time complexity of the following program segment:

```
for i = 1 to i < n do
   for j = 1 to j < n do
     for k = 1 to k < n do
        display i, " " , j, " ", k
     endfor
   endfor
endfor
```

Exercise 12.13 Given the function *myfun* with time complexity of $5n$, determine the time complexity of the following program segment:

```
for i = 1 to i < n do
    myfun ()
endfor
```

Exercise 12.14 If the efficiency of function *myfunct* can be expressed as $O(n) = n^2$, determine the efficiency of the following program segment:

```
for i = 1 to i < n do
   for j = 1 to n-1 do
       call myfunct
   endfor
 endfor
```

Exercise 12.15 If the efficiency of the function *myfunct* can be expressed as $O(n) = n^2$, determine the efficiency of the following program segment:

```
set  i = 1
while i < n   do
   call myfunct
   set i = i * 2
endwhile
```

Exercise 12.16 An implementation of an algorithm computes a given input of size n. If n is 4096, the run time is 512 milliseconds. If n is 16,384, the run time is 2048 milliseconds. What is the efficiency? What is the big-O notation?

13. Trees

13.1 Introduction

Trees are used extensively in computer science to represent algebraic formulas; as an efficient method for searching large, dynamic lists; and for such diverse applications as artificial intelligence systems and encoding algorithms. In this chapter we discuss the basic concepts behind the computer science application of trees. Then, in the following four chapters, we develop the application of trees for especific problems.

This chapter discusses the basic concepts of trees, their program implementation, and the computer science application of trees. Trees are non-linear higher-level data structures and are normally implemented with linked lists. Abstract data types (ADTs) are discussed and defined for tree implementations.

13.2 Basic Concepts of Trees

Because trees are normally implemented with linked lists, trees consist of a collection of nodes that are linked or connected. The branches of a tree are the connections between the nodes. The total number of branches that connect a node is the *degree* of the node. Branches can be edges arrive at a node and edges that leave a node. Figure 13.1 shows the general structure of a tree.

The node at the top of a non-empty tree is known as the *root* and has no branches arriving at it. All the remaining nodes can be partitioned into non-empty sets each of which is a *sub-tree* of the root. Every node in the tree can be reached by following a unique path starting from the root. Basic definitions:

- A *leaf node* has no branches that leave the node, so it has no children.
- A *path* is a sequence of consecutive branches.

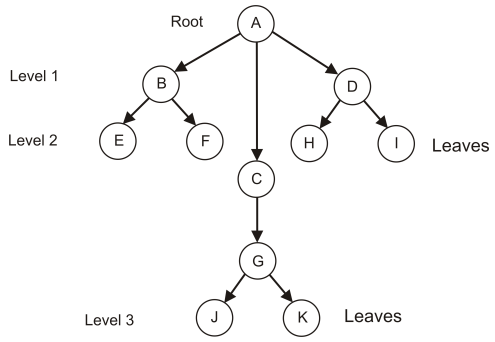

Figure 13.1: General Structure of a Tree

- The *level number* of a node is the number of branches from the root. node. The root has level zero.
- The level of the tree is its distance from the root.
- The *in-degree* of a node is the number of arriving branches of the node.
- The *out-degree* is the number of branches leaving a node.
- The *height* of a tree is the level in the longest path from the root plus 1.
- A subtree is any connected structure under the root.

13.3 Binary Trees

A binary tree is one in which no node can have more than two sub-trees; the maximum out-degree for a node is two. A node can have zero, one, or two sub-trees, the left sub-tree and the right sub-tree. Note that each sub-tree is itself a binary tree. Figure 13.2 shows a binary tree.

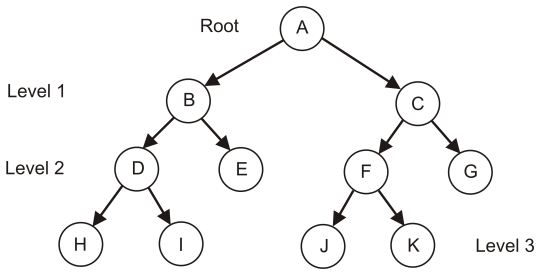

Figure 13.2: A Binary Tree

The implementation of a binary tree uses a linked list and every node consists of three components:

- A data element
- A link to the left child
- A link to the right child.

The general structure of a node of a binary tree is shown in Figure 13.3. The root of the binary tree is pointed at by a *tree pointer* and if it is equal to NULL, then it means the binary tree is empty.

A binary tree is recursive by definition as every node in the tree contains a left sub-tree and a right sub-tree. The leaf nodes contain an empty left sub-tree and an empty right sub-tree. In Figure 13.2, the leaf or terminal nodes are H, I, E, J, K, and G. These nodes have no successors and thus are said to have empty sub-trees.

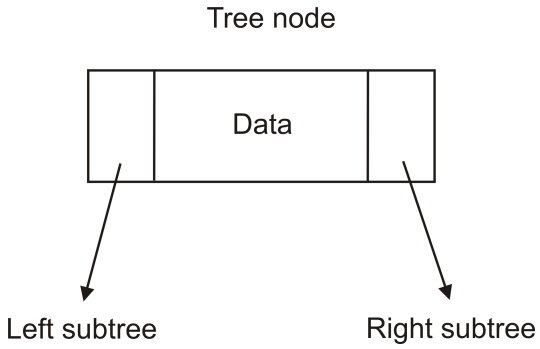

Figure 13.3: A Node of a Binary Tree

A binary tree of *height h* has at least h nodes and at most $2^h - 1$ nodes. This is because every level will have at least one node and can have at most 2 nodes. Therefore, if every level has two nodes then a tree with height h will have at the most $2^h - 1$ nodes as at level 0, there is only one node known as the root. The height of a binary tree with n nodes is at least $\log(n+1)$ and at most n.

The *balance factor* of a binary tree is the difference in height between its left and right sub-trees. Given the height of the left subtree as H_L and the height of the right sub-tree as H_R, the balance factor of the tree, B, is determined by the following formula: $B = H_L - H_R$. A *balanced binary tree* will have the height of its sub-trees differ by no more than one, and the sub-trees will also be balanced.

Complete Binary Tree

A complete tree is one in which every level, except possibly the last, is completely filled and all nodes appear as far left as possible. Figure 13.4 shows a complete binary tree. Level L of the binary tree can have at most 2^L nodes. For example, level 3 can have at most 2^3 nodes.

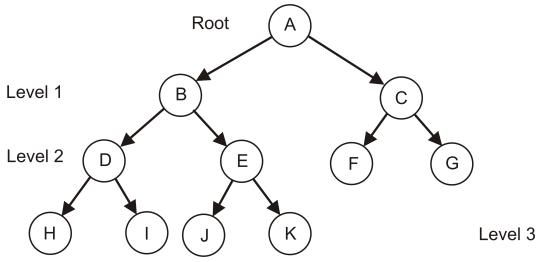

Figure 13.4: A Complete Binary Tree

13.4 Implementation of a Binary Tree

Every binary tree has a tree pointer *Root*, which points to the root node of the tree. The tree is a represented by the following structure.

```
struct bintree is
  variables
     define numnodes of type integer
     define maxnodes of type integer
     define tree_name array [30] of type char
  structures
     define Root of type NodePtr
endstruct bintree
```

The nodes of the binary tree in Figure 13.4 are labeled from 1 to 11. For a parent node, M its left child can be calculated as $2 \times M$ and its right child can be calculated as $2 \times M + 1$. The parent of the node M can similarly be calculated as $\lfloor M/2 \rfloor$. The height of a tree that has exactly N nodes is given as: $H_n = \lfloor \log 2(N+1) \rfloor$.

The implementation of a binary tree is carried out by using a double linked list. As mentioned previously, every node will have three components: the data element, a link (pointer) to the left subtree, and a link (pointer) to the right subtree. So in SPSCL, a node of a binary tree is represented as follows.

```
struct node is
  structures
     define datablock pointer of type DataT
     define left pointer of type NodeType   // pointer to left subtree
     define right pointer of type NodeType  // pointer to right subtree
endstruct node
```

13.5 Traversal of Binary Trees

In the traversal of binary trees, every node is visited once and only once in some predetermined sequence. There are two important types of traversal:

- *Depth first traversal*. The procedure follows the path from the root through one child to the most distant descendant of that first child before visiting a second child. The traversal proceeds visiting all of the descendants of a child before visiting the next child.
- *Breadth first traversal*. The procedure follows a horizontal path from the root to all of its children, then to its children's children, and so on until all nodes have been visited. With breadth-first traversal, each level is completely processed before the next level.

There are three variants of the depth first traversal, these are:

- Preorder traversal
- Inorder traversal
- Postorder traversal

13.5.1 Preorder Traversal

In this tree traversal, the root node is visited first, then the left subtree, then the right subtree. This traversal can be described by a recursive algorithm.

```
preOrder (root)
// Traverse a binary tree in preorder
  if (root is not null)
      visit (root)
      preOrder (leftSubtree)
      preOrder (rightSubtree)
  endif
end preOrder
```

13.5.2 Inorder and Postorder Traversals

The inorder and postorder tree traversal are similar to the preorder. Only the order on which a node is visited changes. With inorder traversal, the left subtree is traversed recursively, then the root node is visited, then the right subtree is traversed. With postorder traversal, the left subtree is traversed, then the right subtree, and finally the root node is visited.

13.5.3 Breadth-first Traversals

In the breadth-first traversal of a binary tree, all of the children of a node are visited before proceeding with the next level. Starting with a root at level n, all nodes at level n are visited before proceeding with the nodes at level $n+1$. A queue is used to traverse a tree in breadth-first order. The high-level algorithm for a breadth-first traversal of a binary tree is the following:

```
breadthFirst (root)
// Process tree using breadth-first traversal.
// Pre root is node to be processed Post tree has been processed
assign root to currentNode
createQueue (queue)
while currentNode not equal null  do
   process (currentNode)
   if left subtree not equal null then
        enqueue (queue, left subtree)
    endif
    if right subtree not equal null
        enqueue (queue, right subtree)
    endif
    if not emptyQueue(queue)
        set currentNode = dequeue (queue)
    else
    set currentNode = null
    endif
endwhile
destroyQueue (queue)
end breadthFirst
```

13.6 Application of Trees

One interesting application of binary trees is the representation of algebraic expressions. This is used to store and manipulate arithmetic expressions. In compiler design and implementation, general

trees are used to store and manipulate programming language statements. Figure 13.5 shows the tree representation of the simple expression: $(X+Y)-Z$.

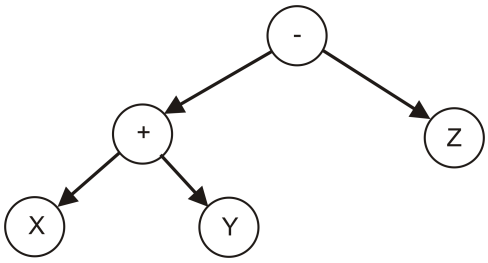

Figure 13.5: Arithmetic Expresion with Binary Tree

13.7 Binary Search Trees

A binary search tree is a binary trees in which the nodes are arranged in a specific order. Every node includes a *key* in its data component. In a binary search tree, all the nodes in the left sub-tree have a key value less than the key value of the root node. All the nodes in the right sub-tree have a key value either equal to or greater than the key value of root node. A binary search tree is also known as an ordered binary tree.

Figure 13.6 shows an example of a binary search key. Note that the key value of the root node is 12, so all the nodes in its left subtree have key values less then 12. In a similar manner, all nodes in its right subtree have key values greater than 12. The same applies to the roots of the subtrees.

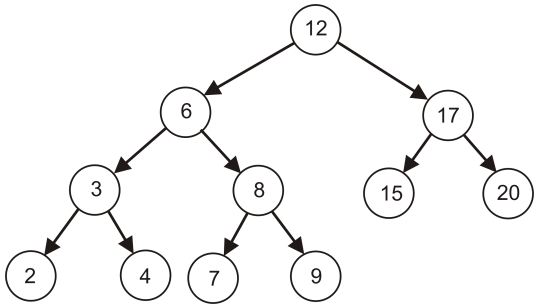

Figure 13.6: A Binary Search Tree

13.7.1 Traversals of a Binary Search Tree

The three types of traversals in a binary search tree generate the following results:

1. Using *preorder* traversal of the tree in Figure 13.6 produces the following sequence of keys: $12,6,3,2,4,8,7,9,17,15,20$.
2. Using *postorder* traversal of the tree in Figure 13.6 produces the following sequence of keys: $2,4,3,7,9,8,6,15,20,17,12$.
3. Using *inorder* traversal of the tree in Figure 13.6 produces the following sequence of keys: $2,3,4,6,7,8,9,12,15,17,20$. This sequence has the values of the keys in ascending order.

Various operations can be performed on a binary search tree. Most of these operations rely on comparison of the key values in the nodes of the tree.

13.7.2 Searches on a Binary Search Tree

Because the nodes in a binary search tree are ordered, the search a node with a given key value element in the tree is relatively efficient. To search for a node with a given key value, there is no need to traverse the entire tree and only one of the sub-trees at each level of the tree is searched. For example, to search for key with value 7 in the tree shown in Figure 13.6, only the left sub-tree will be traversed. Thus, the average time complexity of a search is $O(\log n)$.

Binary search trees are efficient data structures when compared with sorted linear arrays and linked lists. In a sorted array, searching can be performed in $O(\log n)$ time, but insertions and deletions are quite expensive. Inserting and deleting nodes in a linked list is relatively efficient, however, searching for an element has a time complexity of $O(n)$. In the worst case, a binary search tree has a time complexity of $O(n)$ in search for a key value. This will occur when the tree is a linear chain of nodes.

The following listing shows a high-level algorithm to search for a node with a given key value, in a binary search tree. Note that the algorithm is recursive and the search starts from the root of the tree.

```
SearchKey (Root, keyval)
  if key of Root equal keyval or Root equal NULL then
      return Root
  elseif keyval < key of Root then
      return SearchKey (left of Root, keyval)
   else
      return SearchKey (right of Root, keyval)
   endif
end SearchKey
```

13.7.3 Inserting Nodes in a Binary Search Tree

Insertion of a node involves adding a new node with a given key to a binary search tree. This new node must be inserted at the correct position in the binary search tree. The first step is to find the correct position where the the new node is to be inserted. The next step is to add the node at that position. The following listing shows a high-level algorithm to insert a node with a given key value in a binary search tree. Note that this algorithm is also recursive. The algorithm returns a possible new root of the tree.

```
InsertNode (Root, keyval)
  if Root equal NULL then
      create a new node Root
      set key of Root = keyval
      set left of Root = NULL
      set right of Root = NULL
      return Root
  elseif keyval < key of Root then
      return InsertNode (left of Root, keyval)
  else
      return InsertNode (right of Root, keyval)
```

```
  endif
end InsertNode
```

13.7.4 Finding the Node with the Largest Key Value

To find the node with largest key value in a binary search key is almost straightforward. The search proceeds to find the node in the rightmost position in the right subtree. In the case that the right subtree is empty, then the root of the tree will be the node with the largest key value. The corresponding algorithm is very short, as shown in the following listing:

```
findLargestKey (Root)
  if Root equal Null or right of Root equal Null  then
     return Root
  else
     return findLargestKey (right of Root)
  endif
end findLargestKey
```

13.7.5 Finding the Height of a Binary Search Tree

The corresponding algorithm computes recursively the height of the left subtree and the height of the right subtree. The largest of these two plus 1 is the height of the tree. The following listing shows the high-level definition of the algorithm.

```
height_tree (Root)
if Root equal NULL then
    return 0
else
    set leftH = height_tree(left of Root)
    set rightH = height_tree(right of Root)
    if leftH > rightH then
        return leftH + 1
    else
        return rightH + 1
    endif
endif
end height_tree
```

13.7.6 Deleting a Node from a Binary Search Tree

There are four cases to consider when deleting a node with a given key value from a binary search key:

1. The node to delete has no children. This is the simplest case and the node is deleted directly.
2. The node to delete has only a left subtree. The node is removed from the tree and the link to the left subtree is connected to the parent of the deleted node.
3. The node to delete has only a right subtree. The node is removed from the tree and the link to the right subtree is connected to the parent of the deleted node.
4. The node to delete has two subtrees. This is the most complicate case because the tree must retain its balance as much as possible.

The delete node function requires time proportional to the height of the tree in the worst case. It's time complexity is $O(\log n)$ for the average case and $\Omega(n)$ in the worst case. The following listing shows a simplified version of a function to delete a node given its key value from a binary search key.

```
implementations
description
  This is a simplified version of a function in SPSCL to
     delete a node, given its key value from a binary search key.
*/
function deleteNode
    parameters Root of type NodePtr, keyval of type integer
is
structures
   define tnode of type NodePtr
begin
  if Root equal NULL then
     display "root is empty; key not found"
  elseif keyval < key of Root then
     call deleteNode using left of Root, keyval
  elseif keyval > key of Root then
     call deleteNode using right of Root, keyval
  elseif left of Root not equal Null and
        right of Root not equal Null then
     set tnode = findLargestNode(left of Root)
     set key of Root = key of tnode
     call deleteNode using left of Root, key of tnode
  else
     set tnode = Root
     if left of Root equal NULL and right of Root equal NULL then
         set Root = NULL
     elseif left of Root not equal NULL then
         set Root = left of Root
     else
         set Root = right of Root
     endif
     destroy tnode
  endif
endfun deleteNode
```

13.8 Interface for Manipulating Binary Search Trees

The interface for binary trees include the data structures mentioned previously and a set of operations specified for to creating and manipulating the binary tree. Some of the basic operations defined are:

- Create an empty binary tree
- Construct a complete binary tree with n nodes
- Insert a node given its key value to a search binary tree
- Traverse the binary tree inorder to display all the data in the nodes
- Search for a node given its key value in a binary search tree

- Find the node with the largest key value in a binary search key
- Delete a node given its key value from a search binary tree
- Get a copy of the data in the root node of the tree or a subtree
- Find the height of the binary search tree
- Check whether the binary tree is empty
- Check whether the binary tree is full

As mentioned previously, the pointer variable *Root* always points to the top node of the binary tree. If the binary tree is empty, then the value of *Root* is *NULL*. To create an empty list, all that is needed is the assignment as follows:

```
define Root of type Nodeptr
define numnodes of type integer  // current number of nodes
. . .
set Root = NULL
set numnodes = 0
```

To check if a binary tree is empty, the value of the tree root pointer must to be compared for equality with the constant value *NULL*.

```
if  Root equal NULL then
   . . .
endif
```

Function *isEmpty* checks whether the tree is empty by examining the value of the root pointer, which is a reference to the first node in the list. The value of this pointer has a value *NULL* when the tree is empty.

The following listing includes the complete interface for the SPSCL implementation of binary search trees. The interface is stored in file `btreeh.scl`, the function implementations in file `btreelib.scl`, and the test program in file `databtree.scl`.

```
description
 This is the interface for binary search trees */
import "scl.h"
specifications

definetype struct Datablock DataT
definetype struct node NodeType
definetype pointer NodeType NodePtr
struct node is
  structures
     define datablock pointer of type DataT
     define left pointer of type NodeType   // pointer to left node
     define right pointer of type NodeType // pointer to right node
endstruct node
//
definetype struct bintree btreeT
```

```
struct bintree is
  variables
     define numnodes of type integer
     define maxnodes of type integer
     define lname array [30] of type char
  structures
     define Root of type NodePtr
endstruct bintree
//
struct Datablock is              // data block type
 variables
     define keyval of type integer
     define nodenum of type integer
endstruct Datablock

definetype struct Datablock DataT

forward declarations

function init_tree return pointer of type btreeT parameters
    maxn of type integer, pname pointer of type char
function build_tree return type NodePtr
     parameters ptree pointer of type btreeT

function insert_node return type NodePtr
    parameters root of type NodePtr, ndata pointer of type DataT
// function insert_node parameters root of type NodePtr,
//       n of type integer
function preOrder_display parameters root of type NodePtr

function traverse_display parameters
  ptree pointer of type btreeT
function empty_list return type bool parameters
  ptree pointer of type btreeT
function full_list return type bool parameters
  ptree pointer of type btreeT
// function insert_node parameters ptree pointer of type btreeT,
//  ndata pointer of type DataT, pos of type integer

function findLargestKey return type NodePtr
     parameters Root of type NodePtr

function searchKey return type NodePtr
   parameters Root of type NodePtr, keyval of type integer

function get_data return pointer of type DataT
   // parameters ptree pointer of type btreeT, pos of type integer

function listSize return type integer
   parameters ptree pointer of type btreeT
```

```
function height_tree return type integer
   parameters Root of type NodePtr

function deleteNode
    parameters Root of type NodePtr, keyval of type integer

function destroy_tree parameters Root of type NodePtr
```

13.9 AVL Trees

An AVL tree is a binary search tree that is a *balanced*; the heights of the subtrees differ by no more than 1. The search time complexity is $O(\log n)$ in a balanced binary search tree. The time complexity in a totally unbalanced tree is $O(n)$. The most important property of an AVL tree is that the height of its left subtree, H_L, and the height of its right subtree, H_R, differ by not more than 1 and is denoted as: $|H_L - H_R| \leq 1$. The AVL tree is also known as a height-balanced tree.

In an AVL tree, the *balance factor* is an additional data component in every node of the tree. The balance factor of a node is calculated by subtracting the height of its right sub-tree from the height of its left sub-tree. A binary search tree in which every node has a balance factor of -1, 0, or 1 is said to be *height balanced*. A node with any other value of the balance factor is considered to be *unbalanced*. The balance factor is calculated as: $BF = H_L - H_R$.

An AVL tree has all subtrees with a balance factor of 0, 1, or -1. The AVL tree in Figure 13.7 has a balance factor of 0, because its two subtrees have height 3. Node 17 a balance factor of -1, and node 20 has a height factor of 1.

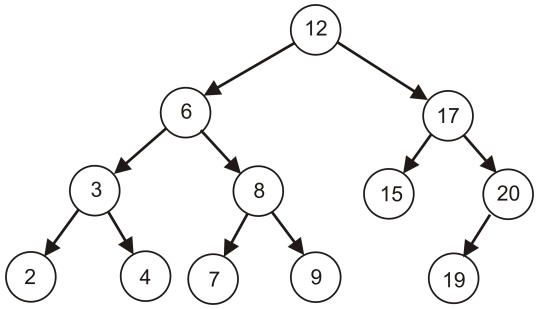

Figure 13.7: An AVL Tree

13.9.1 Seaching a Node in an AVL tree

Searching a node in an AVL tree is performed in the same manner as the search in a binary search tree. The corresponding time complexity is $O(\log n)$.

13.9.2 Inserting and deleting a Node in an AVL Tree

Inserting a node to an AVL tree involves an additional step to the insertion of a node in a binary search tree. This additional step is a *rotation* that might be needed to restore the balance of the tree. Rebalancing of the tree is performed by *rotation* at the critical node. There are four types of rotations: LL rotation, RR rotation, LR rotation, and RL rotation.

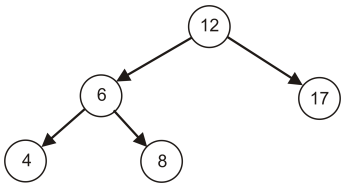

Figure 13.8: A Simple AVL Tree

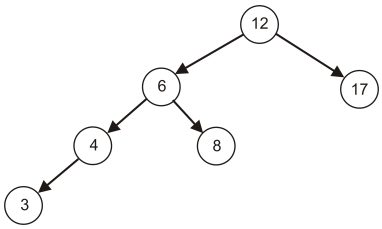

Figure 13.9: An Unbalanced Binary Tree

If the balance factor of a node is 0, the height of the left subtree is equal to the height of its right subtree. If the balance factor of a node is 1, the height of the left subtree is one greater than the height of its right subtree. If the balance factor of a node is -1, the height of the left subtree is one less than the height of its right subtree.

Consider the AVL tree in Figure 13.8. A new node with value 3 is inserted and the tree becomes unbalanced. The balance factor of node 12, the tree root node, is 2 and this unbalanced tree is shown in Figure 13.9.

The unbalanced tree will require a rotation on the node closest to the newly inserted node that does not have a balance factor of 1, 0, or -1. This is sometimes known as the *critical node*. In Figure 13.9, the critical node is node 12. It is necessary to perform a *rotation* of node 12 to the right, the tree becomes balanced and is shown in Figure 13.10.

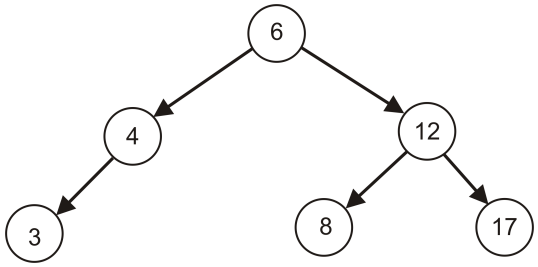

Figure 13.10: After Rebalancing the Tree

Trees that have become unbalanced because of the insertion or deletion of a node need to be rebalanced by rotating critical nodes either to the left or to the right. There are four types of rotations:

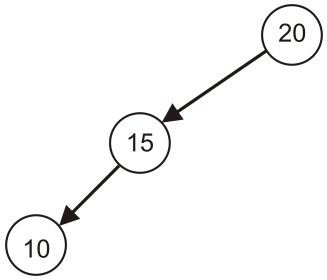

Figure 13.11: A left heavy Tree

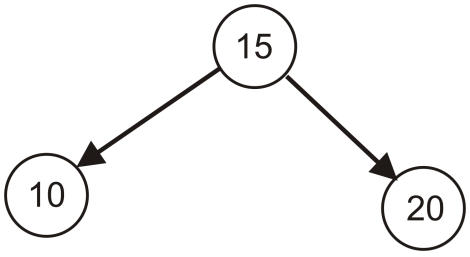

Figure 13.12: AVL Tree After Right Rotation

- *Left left case*. The unbalance has been caused by a left high subtree of a left high tree, rotating the critical node to the right is necessary to restore the balance of the tree. See Figure 13.11 and Figure 13.12. Node 20 is rotated to the right to balance the tree.
- *Left right case*. A new node is inserted in the right of the left sub-tree. Figure 13.13. A left rotation is first applied to node 15; see Figure 13.14. Then a right rotation of node 20 is performed to restore balance of the tree in Figure 13.15.

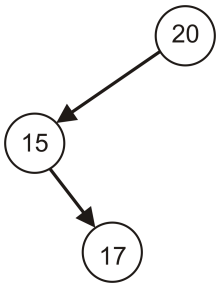

Figure 13.13: A left heavy Tree

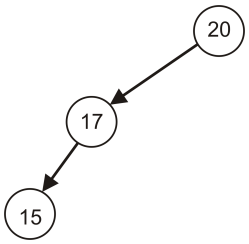

Figure 13.14: Tree After Left Rotation

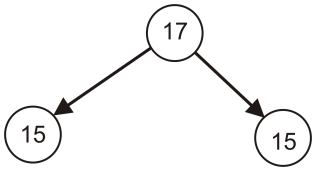

Figure 13.15: AVL Tree After Right Rotation

- *Right right case*. The critical node in Figure 13.16 is node 15, which has a balance factor of -2. A left rotation of this node is performed; see Figure 13.17.

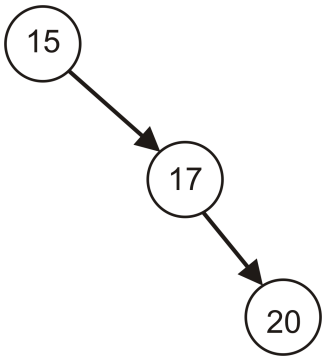

Figure 13.16: A right heavy Tree

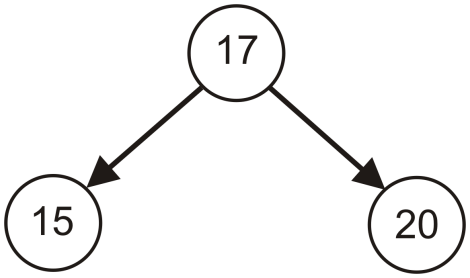

Figure 13.17: AVL Tree After Left Rotation

- *Right left case*. There is a node on the left subtree of the parent right subtree that causes imbalance in the tree; see Figure 13.18. Node 15 has a balance factor of -2. A right rotation is first performed on node 17; see Figure 13.19 . Then a left rotation is performed on node 15; see Figure 13.20.

The following pseudo-code describes a high-level specification of the algorithm for interting a note in an AVL tree. Note that this algorithm actually applies the BST insertion algorithm, then it includes operation for balancing the tree.

```
Algorithm avl_insert (root, newNode)
    //Return root returned recursively up the tree
    if subtree empty then
```

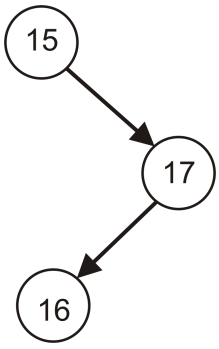

Figure 13.18: A right heavy Tree

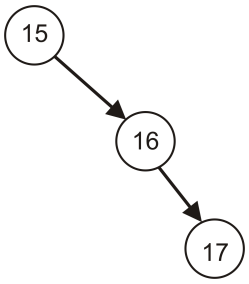

Figure 13.19: Tree After Right Rotation

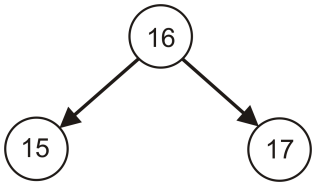

Figure 13.20: AVL Tree After Left Rotation

```
        insert newNode at root
        return root
    endif
    if newNode < root then
         avl_insert (left subtree, newNode)
         if left subtree taller
             leftBalance (root)
         endif
    else
         newNode  >= root data
         avl_insert (right subtree, newPtr)
         if right subtree taller
             rightBalance (root)
         endif
    endif
    return root
end avl_insert
```

The algorithm for balancing a tree uses the four cases of rotation of nodes in order to balance the tree. Note that this balancing operation is also used for node deletion. The following pseudo-code describes the algorithm.

```
algorithm function balance (node)
  // balance factor
  if bf of node equal *2 then
     if bf of leftsubtree of node <= 0 then
         return leftLeft_rotation (node)
     else
         return leftRight_rotation (node)
    endif
  elseif bf of node equal +2 then
     if bf of right subtree of node >= 0 then
        return rightRight_rotation (node)
     else
        return rightLeft_rotation (node)
```

```
      endif
    endif
    return node
 end balance
```

The SPSCL code that implements the building and manipulating AVL trees are stored in files `avltree.scl` and `avlh.scl`. The source code very closely follows the algorithm shown. Running the test run (on Windows) of SPSCL program of AVL trees produces the following console output:

```
C:\SCL\scl_progs>spscl avlh.scl
SCL v 1.0 File: avlh.scl Mon Mar 28 13:50:19 2022

File: avlh.scl no syntax errors, lines processed: 79
C:\SCL\scl_progs>spscl avltree.scl
SCL v 1.0 File: avltree.scl Mon Mar 28 13:54:43 2022

File: avltree.scl no syntax errors, lines processed: 383
C:\SCL\scl_progs>gcc avltree.c
C:\SCL\scl_progs>a

 Inserting nodes:
  4
  2
  1
  3
  7
  5
  8

 Delete node 7

After deletion:
  4
  2
  1
  3
  8
  5
```

Summary

Trees are commonly implemented with linked lists. The most common type of tree is the binary tree in which a node includes link to its left subtree and a link to its right subtree. So a tree is a non-linear data structure. The important binary tree operations are building an empty tree, inserting nodes, deleting nodes, and the various types of tree traversals. An AVL tree is a binary search tree that is balanced. After inserting or deleting nodes in a binary, it may become unbalanced, therefore, several rotation operations may be necessary.

Key Terms

binary trees	branches	leaves
substrees	tree traversals	preorder
postorder	inorder	depth-first
breadth-first	tree root	node key

Exercises

Exercise 13.1 Figure 13.6 shows a binary search tree. Remove the leave nodes and verify the resulting tree by running the program in `databtree.scl`.

Exercise 13.2 On the tree shown in Figure 13.6, perform the three tree traversals discussed for binary search trees. Show by running the program in `databtree.scl`.

Exercise 13.3 Build a binary search tree using the following sequence of keys: 50, 40, 55, 15, 53, 75, 10, 35, 33. 65, 85, 80. Perform the three tree traversals discussed for binary search trees. Show by running the program in `databtree.scl`.

Exercise 13.4 Figure 13.7 shows an AVL tree. Insert node with key 18 and show whether the tree remains balanced. In not, perform the corresponding rotations. Verify your by running the AVL program in file `avltree.scl`.

Exercise 13.5 Add node with key 10 to the tree shown in Figure 13.7. show whether the tree remains balanced. In not, perform the corresponding rotations. Verify your by running the AVL program in file `avltree.scl`.

Exercise 13.6 Add node with key 16 to the tree shown in Figure 13.7. show whether the tree remains balanced. In not, perform the corresponding rotations. Verify your by running the AVL program in file `avltree.scl`.

14. Heaps

14.1 Introduction

A heap is a binary tree whose left and right sub-trees have key values less than their parent. The root of a heap is guaranteed to have the node with the largest key in the tree; its sub-trees contain key values that are less than the root. A difference with the binary search tree, is that the lesser-valued nodes of a heap can be placed on either the right or the left sub-tree. This type of heap is also known as *max-heap*. If the heap has the smallest key value node at the root of the tree, then the heap is known as a *min-heap*

Heaps are usually implemented in an array rather than a linked list. With an an array implementation, the location of the left and right sub-trees can be directly calculated. Conversely, given the location of a node, its parent location can also be calculated directly. This results in efficient processing.

14.2 Concepts and Definitions

A heap is a binary tree of which:

- The tree is complete or nearly complete.
- The key value of each node is greater than or equal to the key value in each of its sub-trees, which are themselves heaps.

Figure 14.1 shows a simple heap. Note that the third level is being filled from the left; it is a nearly complete tree. The heap invariant is that the root element has a value always greater or equal to the elements of its left and right sub-trees.

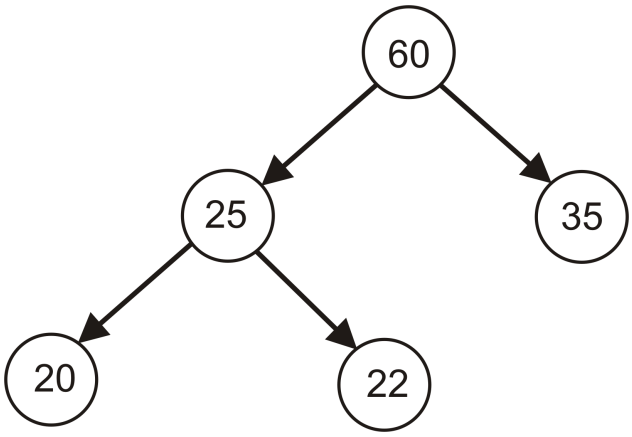

Figure 14.1: A Simple Heap

14.3 Operations on Heaps

The two important operations on a heap are: insertion and deletion of elements. However, two additional operations needed are move up and move down of the element in order to restore the heap invariant. The time complexity of insertion and deletion operations is $O(\log n)$.

14.3.1 Insertion of an Element to a Heap

Insertion of a node consists of the following sequence of steps:

1. A new element is inserted in the next available position at the bottom of the heap, starting from left to right. This is to maintain the property of a heap that it is a complete or nearly complete binary tree. All insertions are performed at the bottom of the heap.
2. The next step is to move up the newly inserted element if the value of its key is greater than the key value of its parent node. Each move up is done by swapping the element with its parent element.
3. Repeat the move up step until the element is not greater then its parent element. When this condition is satisfied, the newly inserted node is a correct location because the heap invariant is satisfied.

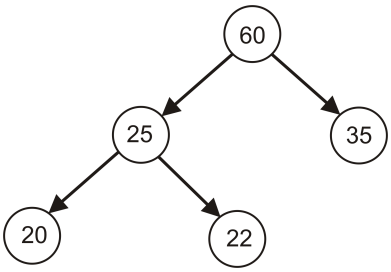

Figure 14.2: A Simple Heap

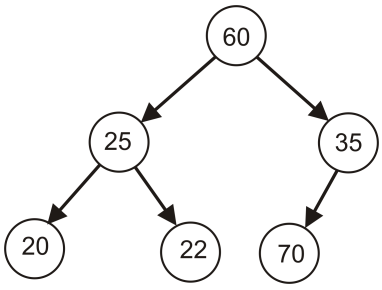

Figure 14.3: An Element Inserted to the Heap

The heap in Figure 14.2 has element at the root with value 60. A new element with key value 70 is

inserted in the next available position in the heap, as show in Figure 14.3. The key value 70 of this new element is greater than the key value 35 of its parent node, therefore the node with 70 moves up and is swapped with the node with key value 35. This is shown in Figure 14.4.

The element with value 70 is under the root, which has key value 60. So again the element 70 has to move up and swapped with the root element. The heap with this last move is shown in Figure 14.5 and now the invariant of the heap has been restored.

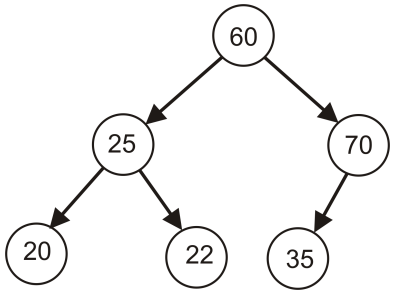

Figure 14.4: Inserted Element Moved Up

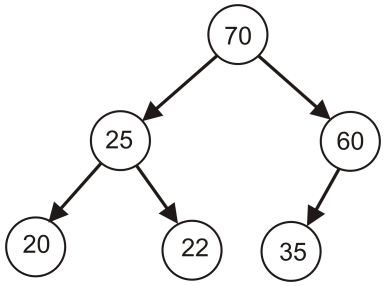

Figure 14.5: Invariant Restored in the Heap

14.3.2 Deleting an Element from a Heap

An element of a heap to delete is always the one at the root of the heap. This operation involves the following sequence of steps:

1. Remove the element at the root of the heap. In Figure 14.2, the root element is the one with value 60.
2. Replace with the element at the bottom of the heap, which was last inserted. This is to retain the property that a heap is a complete or nearly complete binary tree. In Figure 14.6, this is the element with value 22 now at the root.
3. The new root of the heap violates the heap invariant; that an element's value must be greater that the value of its child elements. The element must move down and be swapped with the child element that has the largest value. This is element with value 35 shown in Figure 14.7
4. The previous step must be repeated until the heap invariant is restored; that is, an element's value is greater than the values in its sub-trees.

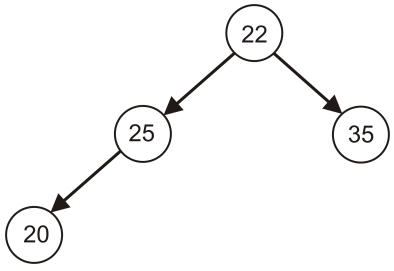

Figure 14.6: Root Replaced

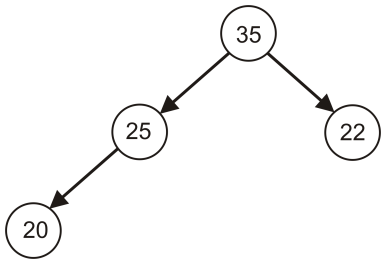

Figure 14.7: Element 22 Moved Down

14.4 Implementation of Heaps

The implementation of heaps is usually done with arrays. Because of the properties of heaps, the position of an element and its children can be calculated directly using the indices in an array. The calculations are performed as follows:

- An element is located at index j in the array. Its left child is located at index $2j+1$, and the location of its right child is located at index $2j+2$.
- The parent of an element is located at index $\lfloor (j-1)/2 \rfloor$.

Figure 14.8 shows the tree representation of a heap and its corresponding array implementation. The relations of the array indices for calculating the position of every element of the heap are used to access any element of the heap.

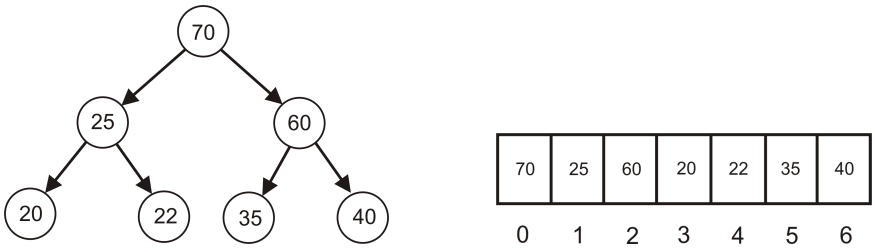

Figure 14.8: Heap Array Implementation

The SPSCL program that builds the heap and inserts new elements is stored in file `heap_insert.scl`. The following listing shows function *moveUp*, which is called from function *insertElem*.

```
// Function to move up ith element in a Heap with
// numel elements
function moveUp parameters harr array[] of type integer,
   i of type integer
is
variables
    define parent of type integer
begin
    // Find parent
    set parent = (i - 1) / 2

    if harr[parent] > 0 then
        // For Max-Heap
        // If current element is greater than its parent
        // Swap both of them and call moveUp again
        // for the parent
        if harr[i] > harr[parent] then
            call swap using address harr[i], address harr[parent]
            // Recursively move up the parent node
            call moveUp using harr, parent
        endif
    endif
endfun moveUp
```

The following listing shows the translation, compilation/linking, and running of the program that builds and inserts an element with key value 70 and one with key value 65 to a heap as shown in Figure 14.9.

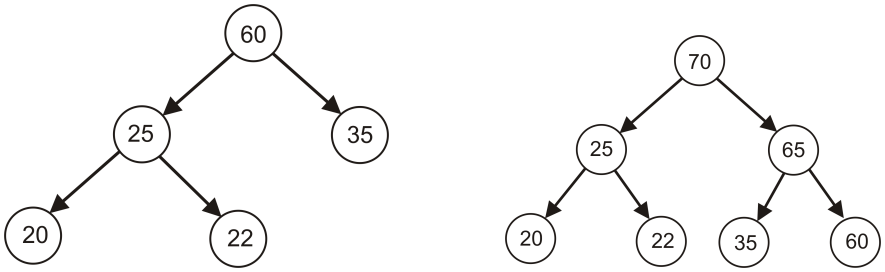

Figure 14.9: Elements 70 and 65 Inserted to Heap

```
C:\SCL\scl_progs>spscl heap_insert.scl
SCL v 1.0 File: heap_insert.scl Sun Apr 03 09:17:25 2022

File: heap_insert2.scl no syntax errors, lines processed: 132
C:\SCL\scl_progs>gcc -o heap_insert heap_insert.c

C:\SCL\scl_progs>heap_insert
   60
   25
   35
   20
   22

inserting element 70:
   70
   25
   60
   20
   22
   35

inserting element 65:
   70
   25
   65
   20
   22
   35
   60
```

The SPSCL program that deletes elements from a heap is stored in file `heap_delete.scl`. The following listing shows the implementation of function *deleteRoot*.

```
function deleteRoot parameters harr array[] of type integer
```

```
is
variables
    define lastElement of type integer
begin
    // Get the last element
    set lastElement = harr[numel - 1]

    // Replace root with last element
    set harr[0] = lastElement

    // Decrease size of heap by 1
    set numel = numel - 1

    // move down the root node
    call moveDown using harr, 0
endfun deleteRoot
```

The following listing shows the translation, compilation/linking, and running of the program that deletes the element with key value 60 from the heap as shown in Figure 14.10.

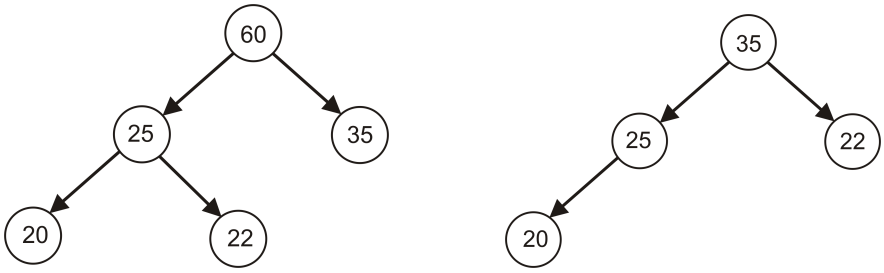

Figure 14.10: Elements 60 Deleted from Heap

```
C:\SCL\scl_progs>spscl heap_delete.scl
SCL v 1.0 File: heap_delete.scl Sun Apr 03 11:22:13 2022
File: heap_delete.scl no syntax errors, lines processed: 112

C:\SCL\scl_progs>gcc -o heap_delete heap_delete.c
C:\SCL\scl_progs>heap_delete
   60
   25
   35
   20
   22

After deleting element 60:

   35
   25
   22
   20
```

14.5 Applications of Heaps

There are several applications of heaps, the most common ones are: priority queues, selection algorithms, and sorting. This section includes short discussion of priority queues.

Priority Queues

The main idea in priority queues is to give priority to an element over the others. This is used in operating systems (OS) where one process has priority over other processes, so the OS will schedule this process before the others.

The heap is a very convenient structure to implement a priority queue. An item that enters the queue with a given priority number that determines its position relative to the other events already in the queue. This item enters the heap in the first empty leaf position and then rises to its correct position relative to all other items in the heap. A high-priority item rises to the top of the heap and becomes the next item to be processed. A low-priority item remains relatively low in the heap.

An item usually is designed to have an encoded priority number that consists of the priority plus a sequential number representing the item's position within the queue. One example of using this is a queue with nine priority classes, the first digit of the priority number represents the queue priority, 1 through 9, and the rest of the number represents the serial placement within the priority.

Using a heap, the serial number must be in descending order within each priority. For example, for a maximum of 1000 items for any priority at any one time, the lowest priority items have sequential numbers in the range 1999 down to 1000, the second-lowest priority to the sequential numbers in the range 2999 to 2000, the third-lowest priority to the numbers in the range 3999 to 3000, and so forth.

The following listing shows the heap interface for a priority queue implementation. This interface is stored in file `heaph.scl`. The general implementation of the heap is the same as the one previously discussed and in programs `heap_insert.scl` and `heap_delete.scl`. The difference is in the structure of an item.

```
description
 This is the interface of a Heap
 for a Priority Queue */
import "scl.h"

symbol MAX 10000 // Max size of Heap

specifications

definetype struct item ItemType
definetype pointer ItemType ItemPtr
struct item is
  variables
     define priority of type integer
 define serial of type integer
 define itemID of type integer
endstruct node
//
```

```
definetype struct heap HeapT

struct heap is
  variables
     define numitems of type integer
     define maxitems of type integer
endstruct heap
//
global declarations

variables
   define numel of type integer  // number of elements in heap
structures
   define heap_array array [] of type ItemType

forward declarations

function init_heap return pointer of type HeapT parameters
    maxn of type integer

function insertElem parameters
    Key of type integer, sequenceNumber of type integer

// Function to delete the root from Heap
function deleteElem

// function to display elements of heap of size numel
function displayHeap parameters heap pointer of type HeapT
```

Summary

A heap is a complete binary tree. Its root is an element with key value greater than its children. These are the two important properties of heaps. The most common operations are element insertion and element deletion. Heaps are more conveniently implemented with arrays.

Key Terms

arrays	move up	move down
heap elements	complete tree	max-heap
min-heap	insertion	delete
heap invariant	root element	priority
serial		

Exercises

Exercise 14.1 Explain the main differences between a binary tree and a tree. Explain the differences between an AVL tree and a heap.

Exercise 14.2 Figure 14.5 shows a heap after inserting element 70. Insert additional elements with key values smaller and greater than the root and verify the resulting heap by running the program in `heap_insert.scl`.

Exercise 14.3 On the heap shown in Figure 14.5, perform the three deletions and verify by running the program in `heap_delete.scl`.

Exercise 14.4 Build a heap using the following sequence of keys: 50, 40, 55, 15, 53, 75, 10, 35, 33. 65, 85, 80. Draw the heap and verify your result by running the program in `heap_insert.scl`.

Exercise 14.5 Perform four deletions from the heap of the previous problem and verify by running the program in `heap_delete.scl`.

Exercise 14.6 Use the heap interface for a priority queue implementation and complete a program of full heap implementation. The heap interface is stored in file `heaph.scl`.

15. Multi-way Trees

15.1 Introduction

Multiway trees have multiple entries in each node so may have multiple sub-trees. These trees are found in applications such as internal search trees, spell checkers, and external file indexes.

15.2 Concepts and Definitions

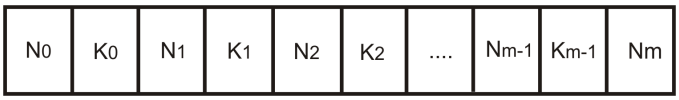

Figure 15.1: A Node Structure in an M-way Tree

A multiway tree such as an *M-way* tree has 0 to m sub-trees. In such a tree, M is known as the degree of the tree. Every internal node of an M-way search tree consists of pointers to M sub-trees and contains $M-1$ keys, where $M > 2$. An m-way tree has the following properties:

- Every node can have 0 to m sub-trees.
- As mentioned previously, a node with $j < m$ sub-trees has $j-1$ data fields (key values).
- The key values of the first sub-tree are less than the key value in the first entry. The key values in the sub-tree pointed at by N_0 are smaller than the key value K_0. Similarly, the key values of the sub-tree pointed at by N_j are smaller than the key value K_j.
- The key values are stored in ascending order, that is, $K_{j-1} < K_j$ for $0 \leq j \leq m-1$.
- All sub-trees are also M-way trees.

Figure 15.1 shows the structure of a node in an m-way tree. Note that a N_j field is a pointer to the jth sub-tree. The field K_j is the jth key in the node. A simple and small 3-way tree has nodes with three pointers to three sub-trees and two key values. See Figure 15.2.

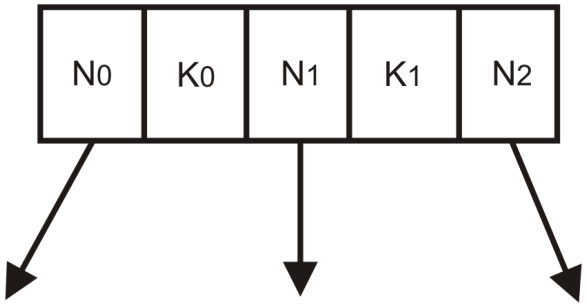

Figure 15.2: A Node Structure in a Three-way Tree

15.3 B-trees

A B-tree is an M-way search tree in which a large number of keys can be stored in a single node. A B-tree has the following additional properties:

1. A node has a maximum of m pointers to sub-trees.
2. All internal nodes have at least $\lceil m/2 \rceil$ non-null sub-trees.
3. All leaf nodes are at the same level; therefore, the tree is perfectly balanced.
4. A leaf node has at least $\lceil m/2 \rceil - 1$ and at most $m-1$ entries.

The time complexity of insertion and deletion operations is $O(\log n)$.

15.3.1 Insertion of an Element to a B-tree

Insertion of a node consists of the following sequence of steps:

1. Similar to the binary search tree, B-tree insertion is at a leaf node. Locate the leaf node for the data to insert.
2. If the leaf node has fewer than $m-1$ entries, the new data are inserted in sequence in the node.
3. When the leaf node is full, then it reaches the condition known as *overflow* and requires that the leaf node be split into two nodes, each containing half of the data. A new node is created from the available memory and then the data is copied from the end of the full node to the new node. After the data have been split, the new entry is inserted into either the original or the new node, depending on its key value. Then the median data entry is inserted into the parent node.

B-trees grow in a balanced manner starting from the bottom. A new root node is created and the tree grows one level, when the root node of a B-tree reaches the condition of overflow and the median entry is moved up.

Example of Insertion to an Empty B-tree

Starting with an empty B-tree of degree five, the data to insert have key values: 10, 25, 15, and 55. These key values are inserted in ascending order in a newly created node that becomes the root node

of the B-tree. This node now has the four key values and the node is full. The node is shown in Figure 15.3.

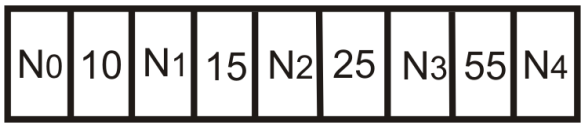

Figure 15.3: Node Structure of the B-tree with Degree Five

A new insertion to the B-tree is key value 75. Because the current node is full, a new node is created and becomes a right sub-tree. The keys in the upper half of the previous node are moved to the new node including key 75.

The median-valued key, which is key value 25, is moved up to a parent node that is now the new root of the B-tree, therefore this tree now has on more level. See Figure 15.4.

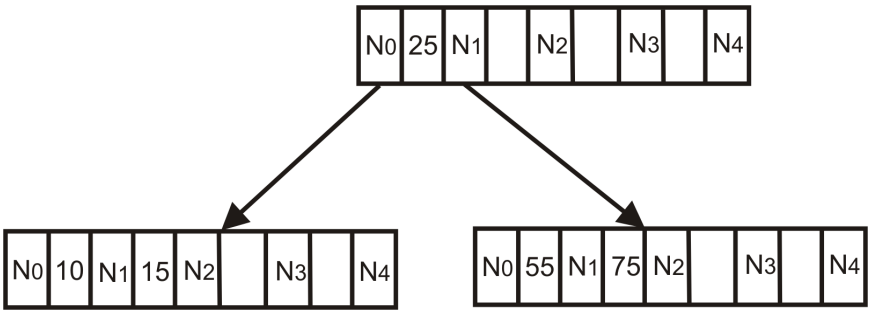

Figure 15.4: B-tree After Insertion of Key 75

15.3.2 Deleting a Key from a B-tree

Deletion of keys are performed from the leaf nodes and there are two cases: either deletion of a leaf node will need to be performed, or deletion of an internal node will need to be performed. When deleting a leaf node, the following steps are involved:

1. Find the leaf node that has the key to be deleted.
2. If the leaf node contains more than more than $m/2$ keys, then delete the key.
3. If the leaf node does not contain $m/2$ keys, then fill the node by taking a key either from the left or from the right sibling.
    - If the left sibling has more than the minimum number of key values, move up its largest key into its parent's node and move down the intervening key from the parent node to the leaf node where the key is deleted.
    - If the right sibling has more than the minimum number of key values, move up its smallest key into its parent node and move down the intervening key from the parent node to the leaf node where the key is deleted.
4. If both left and right siblings contain only the minimum number of elements, then a new leaf node is created by combining the two leaf nodes and the intervening key of the parent node. If

moving down the intervening key from the parent node leaves it with less than the minimum number of keys in the node, then propagate the process upwards, therefore the height of the B-tree is reduced.

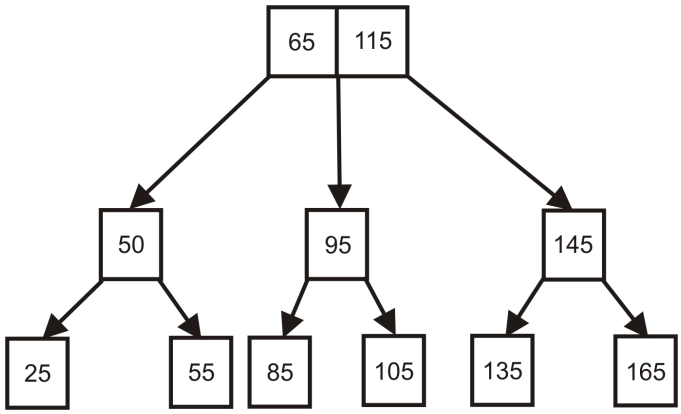

Figure 15.5: B-tree with Initial Keys

To delete an internal node, promote the successor or predecessor of the key to be deleted to occupy the position of the deleted key. This predecessor or successor will always be in the leaf node; the processing is performed as if a value from the leaf node has been deleted. Figure 15.5 shows a B-tree with initial keys. Figure 15.6 shows the B-tree after deleting key 135.

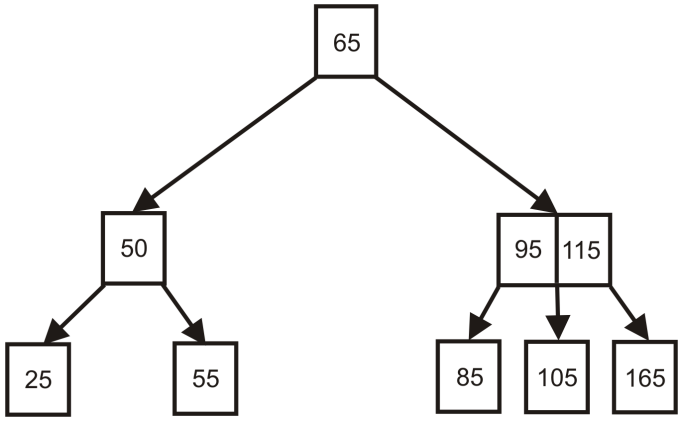

Figure 15.6: B-tree After Deleting Key 135

The following listing includes the interface for the program that implements a B-tree and is stored in file `b-tree1h.scl`. The complete program with the implementation of the B-tree is stored in file `b-tree1.scl`.

```
// Interface of a B-Tree

import "scl.h"
symbol NUMKEYS 24
```

```
symbol NUMCHILD 12

specifications

// A BTree node
definetype struct B_treeNode BTreeNode
struct B_treeNode is
 variables
    define keys array [NUMKEYS] of type integer  // An array of keys
    define mindeg of type integer                // Minimum degree
    define numkeys of type integer         // Current number of keys
    define leaf of type bool               // true when node is leaf
 structures
     define Child array [NUMCHILD] pointer of type BTreeNode
endstruct B_treeNode

definetype struct B_Tree BTree

struct B_Tree is
 variables
    define mindeg of type integer   // Minimum degree
 structures
    define root pointer of type BTreeNode
endstruct B_Tree

forward declarations

// Initialize tree
function BTree_init parameters pmindeg of type integer,
      tree pointer of type BTree

// tree traverse
function ttraverse parameters tree pointer of type BTree

// search a key in this tree
function tsearch return pointer of type BTreeNode parameters
      pkey of type integer, tree pointer of type BTree

// Insert a new key in this B-Tree
function insert parameters pkey of type integer,
      tree pointer of type BTree

// Remove a new key from thie B-Tree
function tremove_key parameters pkey of type integer,
      tree pointer of type BTree
```

The following listing shows the translation from SPSCL to C, compiling (and linking) the C program, and execution of the program that implements a B-tree.

```
C:\SCL\scl_progs>spscl b-tree1h.scl
SCL v 1.0 File: b-tree1h.scl Thu Apr 14 12:46:49 2022
```

```
File: b-tree1h.scl no syntax errors, lines processed: 50

C:\SCL\scl_progs>spscl b-tree1.scl
SCL v 1.0 File: b-tree1.scl Thu Apr 14 12:47:02 2022

File: b-tree1.scl no syntax errors, lines processed: 743

C:\SCL\scl_progs>gcc b-tree1.c -o btree1.exe
C:\SCL\scl_progs>btree1
Traversal of B-tree after inserting keys

    1
    2
    3
    6
    7
    10
    11
    12
    13
    17
    18
    20
    22

Traversal of B-tree after removing key 6

    1
    2
    3
    7
    10
    11
    12
    13
    17
    18
    20
    22


Traversal of B-tree after removing key 13

This key does not exist in the tree
  27
Traversal of B-tree after removing key 27

    1
    2
    3
```

```
7
10
11
12
17
18
20
22
```

15.4 Variations to B-tree

The main memory of a computer system is volatile in nature and cannot store a large amount of data, therefore, data is stored on secondary storage devices. The access time of retrieving data from magnetic disks is 10,000 to 1,000,000 times longer than the access time from the main memory.

B-trees are often used to index the data and provide better access time. Indexing large amounts of data can provide significant improvement to the performance of search operations.

B+trees

A variant of a B-tree is the B+tree, which stores sorted data in a manner that results in more efficient insertion, retrieval, and removal of records. A *key* identifies each record. A B+tree stores all the records of a file at the leaf level of the tree; only keys are stored in the interior nodes. The leaf nodes of a B+ tree are often linked to one another in a linked list. This enhances data queries in that these are simpler and more efficient.

The secondary storage (magnetic disk) is used to store the leaf nodes of trees and the internal nodes of trees are stored in the main memory. B+ trees store data only in the leaf nodes. All other nodes (internal nodes) are called *index nodes* or i-nodes and store index values.

B+trees are used in data base systems and simpler file management systems. The most application of B-trees and B+trees is indexed file organization. An index is maintained that points to the location on disk where the data of the record is found. In the simplest case, an index file is created in addition to the corresponding data file. Insert takes constant time for the data itself plus $\log 2n$ for the (smaller) index Select, Update, Delete take $\log 2n$ lookup on the index followed by constant time to access data record. In indexed files, a new data record is inserted at the end of the data file; this file is not ordered.

An example of an application of B+trees is the following SPSCL program that uses the PBL library. The following listing includes the interface for using the library, which implements B+tree structure for indexing files. The interface is stored in file `indxFh.scl` and a complete program implementation is stored in file `file_proc6.scl`.

```
/*
  Interface package for indexed files
  using the PBL btree library
  Updated May 2022. J Garrido
  File:  indxFh.scl */
```

```
import <stdio.h>
import <stdlib.h>
import <string.h>

import <errno.h>
import <stdlib.h>
import <ctype.h>

import "pbl.h"

specifications

struct idxFile  is
 variables
    define dataFileName array[30] of type char
define keyFileName array[30] of type char
 structures
define dataFilep pointer of type FILE
define keyFile pointer of type pblKeyFile_t
endstruct idxFile

definetype struct idxFile idxFileT

struct param_record is
 variables
define datarec pointer of type void
define datareclen of type unsigned integer
define numblk of type unsigned integer
define keyf pointer of type char
define keyflen of type unsigned integer
define recno pointer of type char
define recnolen of type unsigned integer
define ival of type integer
endstruct param_record

definetype struct param_record precT

forward declarations

// Create a data file and its corresponding index (key) file
function idxFCreate return type integer
  parameters idxfp pointer of type idxFileT, dfilepath pointer of type char,
    filesettag pointer of type void

// Open the data file and corresponding key file
//  update = 1;     // to write or update. otherwise is 0
function idxFOpen return type integer
  parameters idxfp pointer of type idxFileT,
     openMode pointer of type char, update of type integer

// Write data record and corresponding key record
```

```
function idxFWrite return type integer
  parameters idxfp pointer of type idxFileT, precPtr pointer of type precT

// Given a key value, find and read the record from data file
function idxFReadRec return type integer
  parameters idxfp pointer of type idxFileT, precPtr pointer of type precT

// Read the next record from data file, by key value
function idxFReadNext return type integer
  parameters idxfp pointer of type idxFileT, precPtr pointer of type precT

// Read the first record from data file and key file
function idxFReadFirst return type integer
  parameters idxfp pointer of type idxFileT, precPtr pointer of type precT

// Delete a record given its key, from the data file and from the key file
function idxFDeleteRec return type integer
  parameters idxfp pointer of type idxFileT, precPtr pointer of type precT

// close data file and thye corresponding key file
function idxFClose return type integer
  parameters idxfp pointer of type idxFileT
```

Summary

Multiway trees have multiple entries in each node so may have multiple sub-trees. These trees are found in applications such as internal search trees, spell checkers, and external file indexes. B-trees and B+trees are the structures used for efficient manipulation of files.

Key Terms

index	key	files
secondary storage	main memory	data access
B-tree	B+tree	file management

Exercises

Exercise 15.1 Explain the main differences between a binary tree and a multi-way tree. Explain the differences .

Exercise 15.2 On the tree shown in Figure 15.6, perform the three deletions and verify by running the program in `b-tree1.scl`.

Exercise 15.3 Build a B-tree using the following sequence of keys: 50, 40, 55, 15, 53, 75, 10, 35, 33. 65, 85, 80. Draw the tree and verify your result by running the program in `b-tree1.scl`.

Exercise 15.4 Perform four deletions from the tree of the previous problem and verify by running the program in `b-tree1.scl`.

Exercise 15.5 Perform the indicated modification to the SPSCL program `file_proc6.scl`. Include four additional searches given appropriate values of keys. Show the resulting listing of the data file content.

Exercise 15.6 Perform the indicated modification to the SPSCL program `file_proc6.scl`. Include five deletion of records given appropriate values of keys. Show the resulting listing of the data file content.

16. Graphs

16.1 Introduction

A graph is an abstract data structure that consists of a collection of *vertices* (also known as *nodes*) and *edges* that connect these vertices. Graphs are used to model any situation where objects are related to each other in pairs. For example, the following can be modeled by graphs: Computer networks, transportation networks in which nodes are airports, intersections, ports, etc.

16.2 Basic Definitions

A graph G is defined as an ordered set $G = (V, E)$, where V represents the set of vertices and E represents the edges that connect these vertices. The edges are each defined by a pair of vertices $E_{k,n} = (V_k, V_n)$. Simple graphs are non-directed (also known as *undirected graphs*), there is no implied direction from one vertex to another vertex in an edge. Figure 16.1 shows a graph with six vertices $V = \{a, b, c, d, e, f\}$ and eight edges $E = \{(a,b), (a,c), (b,c), (b,d), (b,e), (d,e), (e,f), (c,f)\}$.

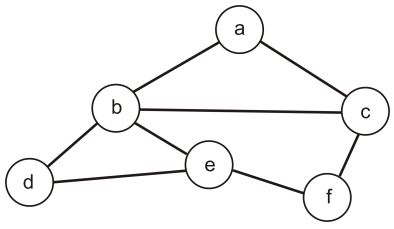

Figure 16.1: A simple non-directed graph.

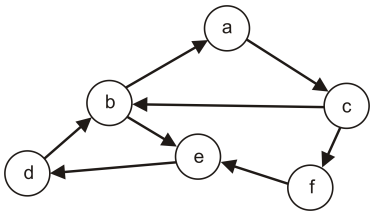

Figure 16.2: A directed graph.

A *directed graph* has an indicated direction of an edge from one vertex to the next. The edges are each defined by an ordered pair of vertices $E_{k,n} = (V_k, V_n)$. An arrow is drawn from one vertex to the next to show the direction. Figure 16.2 shows a directed graph with six vertices and eight edges:

$$V = \{a,b,c,d,e,f\} \quad E = \{(a,c),(b,a),(c,b),(b,e),(e,d),(d,b),(c,f),(f,e)\}$$

Additional Graph Terminology

- *Adjacent vertices* are the vertices that are connected via an edge. For every edge, $e = (m,n)$ the vertices m and n are adjacent.
- The *degree of a vertex* is the total number of edges incident to the vertex. In Figure 16.1, vertex a has a degree of 2, the degree of vertex b is 4.
- A *path* from a node m to node n is defined as a sequence of $(k+1)$ nodes. The path from vertex v_0 to vertex v_k is denoted as $P = (v_0, v_1, v_2, \ldots, v_k)$, of length k
- A *cycle* is a path in which the first and the last vertices are same. A simple cycle has no repeated edges or vertices (except the first and last vertices). A tree is a special graph that has no cycles.
- A *complete graph* is a graph with all its vertices that are fully connected. That is, there is a path from one vertex to every other vertex. A complete graph with n vertices has $n(n-1)/2$ edges.
- A *weighted graph* is a graph with a value assigned to every edge. The weight of an edge denoted by $w(e)$ is a positive value that indicates the cost of traversing the edge.
- The *out-degree* of a vertex in a directed graph, is the number of edges that originate at the vertex.
- The *in-degree* of a vertex in a directed graph, is the number of edges that terminate at the vertex.
- The *degree of a vertex* in a directed graph, is the sum of in-degree and out-degree of the vertex. For vertex v, this is denoted $deg(v) = indeg(v) + outdeg(v)$.
- A *loop* is an edge that connects a vertex to itself. This is drawn as an arrow) with both ends at the same vertex. Figure 16.3 shows a loop at vertex c.

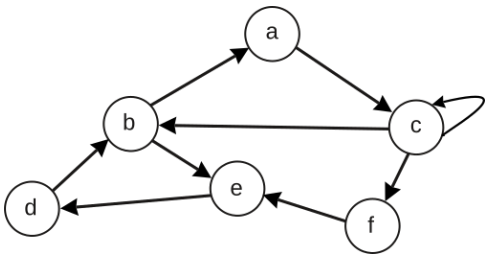

Figure 16.3: A graph with a loop.

16.3 Graph Implementation

There are three general approaches to represent graphs:

1. Sequential representation by using an *adjacency matrix*.
2. Linked representation by using an *adjacency list* that stores the neighbors of a node using a linked list.
3. Adjacency multi-list, which is an extension of a linked representation.

16.3.1 Adjacency Matrix Representation

An adjacency matrix is used to represent whether a specified edge is present, or which vertices are adjacent to one another. Recall that two vertices are adjacent if there is an edge connecting them. In a directed graph G, if vertex m is adjacent to vertex n, then there is an edge from m to n. For any graph G with k vertices, the adjacency matrix will have the dimension of $k \times k$.

In an adjacency matrix, the rows and columns are labeled by graph vertices. An element of the matrix in row i and column j will have a value of 1 if vertices i and j are adjacent, otherwise the matrix element will have a value of 0. The adjacency matrix for a weighted graph contains the weights of the edges connecting the vertices.

The following is the adjacency matrix for the undirected graph shown in Figure 16.1, it is the matrix on the left-hand side using numbers for the vertex labels instead of lower-case letters. The adjacency matrix for the directed graph shown in Figure 16.2 is the matrix on the right-hand side.

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

The following listing shows a function for creating the adjacency matrix for a graph. The complete SPSCL program is stored in file `adjmatrix2.scl`.

```
 // Graph with N vertices and M Edges
variables
    define N of type integer
    define M of type integer

implementations

// Create Adjacency Matrix
function createAdjMatrix parameters Adj array [] [N + 1] of type integer,
    edge array [][2] of type integer
is
variables
    define j of type integer
    define k of type integer
    define i of type integer
begin
    // Initialise Adjacency matrix
```

```
    for i = 0 to N do
        for j = 0 to N do
            set Adj[i][j] = 0
        endfor
    endfor

    // Setup the edges in matrix
    display "Edges: \n"
    for i = 0 to M-1 do

        // Edges
        set j = edge[i][0]
        set k = edge[i][1]
        display j," ", k

        // Set matrix element
        set Adj[j][k] = 1
        set Adj[k][j] = 1
    endfor
endfun createAdjMatrix
```

16.3.2 Adjacency Lists Representation

Adjacency lists or edge lists are also used for the representation of a graph. A linked list is used for each vertex; every element correspond to an adjacent vertex of the specified vertex. In Figure 16.4, an array is shown for all vertices in the graph. Each element of this array has a link to a linked list to all its adjacent vertices. This corresponds to the graph in Figure 16.1 using numbers for the vertex labels instead of lower-case letters.

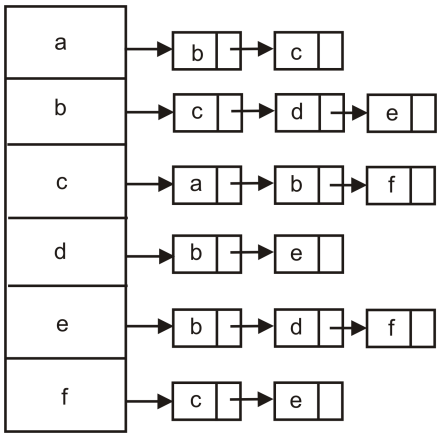

Figure 16.4: Adjacency lists.

The following listing shows the interface file for the adjacency lists implementation of a graph, it is stored in file `graph2h.scl`. The complete program is stored in file `graph2b.scl`.

```
// Interface for creating and processing a graph
```

```
// using adjacency list implementation
import "scl.h"
specifications

definetype struct node nodeStruct

struct node is
 variables
    define vertex of type integer
 structures
    define next pointer of type nodeStruct
endstruct node

definetype pointer nodeStruct nodePtr

struct dsgraph is
 variables
    define numVertices of type integer
define visited pointer of type bool
 structures
    define nodeLists pointer of type nodePtr
endstruct dsgraph

definetype struct dsgraph graphstruct
definetype pointer graphstruct graphPtr

forward declarations

function createNode return type nodePtr
   parameters vertex of type integer

// create a graph with n vertices
function createGraph return type graphPtr
   parameters nvertices of type integer

// Add edge from source to destination
// On an empty graph
function addEdge parameters graph of type graphPtr,
    source of type integer, destination of type integer

// Depth first traversal on the specified graph
function depthFirstT parameters graph of type graphPtr,
     vertex of type integer
//
// Breadth first traversal
function breadthFirst parameters graph of type graphPtr,
   startVertex of type integer

// Display the graph
function displayGraph parameters graph of type graphPtr
```

Summary

A graph is an abstract data structure that consists of a collection of *vertices* (also known as *nodes*) and *edges* that connect these vertices. A graphs can be implemented using an *adjacency matrix*, or an adjacency multi-list, which is an extension of a linked representation.

Key Terms

undirected graph	directed graph	vertices	edges
degree of vertex	adjacent vertices	path	complete graph
weighted graph	adjacency matrix	adjacency lists	graph traversal

Index