# UNIT-III

# STRUCTURING THE COMPUTATION

# INTRODUCTION:-

This chapter is devoted to a detailed analysis of how computations are structured in a programming languages in terms of the flow of control among the different components of a program.

Programs are often decomposed into units. For example, routines provide a way of hierarchically decomposing a program into units representing new complex operations. Once program units are constructed, it becomes necessary to structure the flow of computation among such units. The simplest kind of unit-level control regime is the routine call and return. Another kind of control regime is exception handling, which supports the ability to deal with anomalous situations which may arise even asynchronously, as the program is being executed.

**4.1 Expressions and statements:**

Expressions define how a value can be obtained by combining other values through operators. The values from which expressions are evaluated are either denoted by a literal, as in the case of the real value 57.73, or they are the r\_value of a variable.

Operators appearing in an expression denote mathematical functions. They are characterized by their aritiy (i.e., number of operands) and are invoked using the function’s signature. A unary operator is applied to only one operand. A binary operator is applied to two operands. In general, a n-array operator is applied to n operands. For example, ’-’ can be used as a unary operator to transform–say–the value of a positive expression into a negative value. In general, however, it is used as a binary operator to subtract the value of one expression from the value of another expression. Functional routine invocations can be viewed as n-array operators, where n is the number of parameters.

Regarding the operator’s notation, one can distinguish between infix, prefix, and postfix. Infix notation is the most common notation for binary operators: the operator is written between its two operands, as in x + y. Postfix and prefix notations are common especially for non-binary operators. In prefix notation, the operator appears first, and then the operands follow.

 In postfix notation the operands are followed by the corresponding operator. Assuming that the arity of each operator is fixed and known, expressions in prefix and postfix forms may be written without resorting to parentheses to specify subexpressions that are to be evaluated first. For example, the infix expression

**a \* ( b + c)** can be written in prefix form as

**\* a + b c** and in postfix form as

**a b c + \***

In C, the increment and decrement unary operators ++ and -- can be written both in prefix and in postfix notation. The semantics of the two forms, however, is different; that is, they denote two distinct operators. Both expressions ++k and k++ have the side effect that the stored value of k is incremented by one. In the former case, the value of the expression is the value of k incremented by one (i.e., first, the stored value of k is incremented, and then the value of k is provided as the value of the expression). In the latter case, the value of the expression is the value of k before being incremented.

Infix notation is the most natural one to use for binary operators, since it allows programs to be written as conventional mathematical expressions. For example, the convention adopted by most languages is such that

a + b \* c is interpreted implicitly as

a + (b \* c) i.e., multiplication has precedence over binary addition (as in standard mathematics). However, consider the Pascal expression

**a = b < c** and

the C expression

**a == b < c**

In Pascal, operators < and = have the same precedence, and the language specifies that application of operators with the same precedence proceeds left to right. The meaning of the above expression is that the result of the equality test (a=b), which is a Boolean value, is compared with the value of c (which must be a Boolean variable). In Pascal, FALSE is assumed to be less than TRUE, so the expression yields TRUE only if a is not equal to b, and c is TRUE; it yields FALSE in all other cases. For example, if a, b and c are all FALSE, the expression yields FALSE.

In C, operator "less than" (<) has higher precedence than "equal" (==). Thus, first b < c is evaluated. Such partial result is then compared for equality with the value of a. For example, assuming a = b = c = false (represented in C as zero), the evaluation of the expression yields 1, which in C stands for true.

Some languages, like C++ and Ada, allow operators to be programmer defined. For example, having defined a new type Set, one can define the operators + for set union and - for set difference. The ability of providing programmer-defined operators, as any other feature that is based on overloading, can in some cases make programs easier to read, and in other cases harder. Readability is improved since the programmer is allowed to use familiar standard operators and the infix notation also for newly defined types.

The effect of this feature, however, is such that several actions happen behind the scenes when the program is processed. This is good whenever what happens behind the scenes matches the programmer’s intuition; it is bad whenever the effects are obscure or counterintuitive to the programmer.

Functional programming languages are based heavily on expressions. In such languages, a program is itself an expression, defined by a function applied to operands, which may themselves be defined by functions applied to operands. Conventional languages, instead, make the values of expressions visible as a modification of the computation’s state, through assignment of expressions to variables.

An assignment statement, like x = y + z in C, changes the state by associating a new r\_value with x, computed as y + z. To evaluate the expression, the r\_values of variables y and z are used. The result of the expression (an r\_value) is then assigned to a memory location by using the l\_value of x. Since the assignment changes the state of the computation, the statement that executes next operates in the new state. Often, the next statement to be executed is the one that textually follows the one that just completed its execution. This is the case of a sequence of statements, which is represented in C as

**statement\_1;**

 **statement\_2;**

 **. . . statement\_n;**

 the sequence can be made into a compound statement by enclosing it between the pair of brackets {and}. In other languages, like Pascal and Ada, the keywords begin and end are used instead of brackets.

In many conventional programming languages, like Pascal, the distinction between assignment statements and expressions is sharp. In others, like C, an assignment statement is actually an expression with a side-effect. The value returned by an assignment statement is the one that is stored in the left operand of the assignment operator "=". A typical example is given by the following loop which reads successive input characters until the end of file is encountered:

**While ((c = get char ( ))! = EOF)** /\* assigns the character read to c and yields the read value, which is compared to the end of file symbol \*/ . . . Furthermore, in C the assignment operator associates from right to left. That is, the statement

**a = b = c = 0;** is interpreted as

**a = (b = (c = 0))**

Many programing languages, like Pascal, require the left-hand side of an assignment operator to be a simple denotation for an l\_value. For example, it can be a variable name, or an array element, or the cell pointed by some variable. More generally, other languages, like C, allow any expression yielding a modifiable l\_value to appear on the left-hand side. Thus, it is possible to write the following kind of statement

**( p> q) ? p\* : q\* = 0;**

which sets to zero the element pointed by the maximum of p and q.

As another example, one can write

**\*p++ = \*q++;**

the right-hand side expression yields the value pointed by q. The left-hand side is an expression which provides the r\_value of p, which is the reference, i.e., an l\_value. So the overall effect is that the value of the object pointed by q is copied into the object pointed by p. Both pointers are also incremented as a side effect. Since the above assignment is an expression, the value of the expression is that of the object pointed by q.

For example, the following concise piece of code copies a sequence of integers terminated by zero pointed by p into a sequence pointed by q.

While ((\*p++ = \*q++)! = 0) { }; Sequences, as shown before, are the simplest form of compound statements. Often, the syntax of the language requires each statement in a sequence to be separated from the next by a semicolon. For example, in Pascal a sequence can be written as:

**Begin**

**stat\_1;**

**stat\_2;**

**.stat\_n**

 **End**

Other languages, instead, require each statement to be terminated by a semicolon, and therefore do not need any special separator symbol. For example, in C we would write

**{**

**stat\_1;**

**stat\_2:**

**. . .**

**stat\_n;**

**}**

Although the choice between the two syntactic forms has no deep implications, pragmatically the latter can be more convenient, because one does not need to distinguish between the last statement of a sequence (which does not require the separator, and any other statements, since all are terminated by a semicolon.

Programming languages provide other kinds of compound statements in addition to sequences. We will survey them in Section 5.2. In the rest of this chapter, we implicitly concentrate on conventional languages, unless explicitly stated otherwise. Functional languages, which are not based on computations defined by successive state changes, will be studied in Chapter 7.

**4.2 Conditional execution and iteration**

Conditional execution of different statements can be specified in most languages by the **if** statement. Let us start with the example of the if statement as originally provided by Algol 60. Two forms are

Possible, as shown by the following examples:



In the first case, no alternative is specified for the case i ¦ 0, and thus nothing happens if i ¦ 0. In the latter, two alternatives are present. Since the case where i ¦ 0 is described by a sequence, it must be made into a compound statement by bracketing it between begin and end.

A syntactic variation that avoids this problem is adopted by Algol 68, Ada, and Modula-2, which use a special keyword as an enclosing final bracket of the if statement (fi in the case of Algol 68, end if in the case of Ada, end in the case of Modula-2). Thus, the above examples would be coded in Modula-2 as

**if i = 0**

**then i := j**

 **else i := i + 1;**

 **j := j – 1**

 **end**

and

**if x > 0 then if x < 10 then x := 0 else x := 1000 end end**

or

**if x > 0 then if x < 10 then x := 0 end else x := 1000 end**

depending on the desired interpretation.

Choosing among more than two alternatives using only if-then-else statements may lead to awkward constructions, such as

**if a**

**then S1**

**else**

 **if b**

 **then S2**

 **else**

**if c**

**then S3**

**else S4**

**end**

**end**

**end**

To solve this syntactic inconvenience, Modula-2 has an else-if construct that also serves as an end bracket for the previous if. Thus the above fragment may be written as

**if a**

**then S1**

 **else if b**

**then S2**

**else if c**

**then S3**

**else S4**

**end**

 C, Algol 68, and Ada provide similar abbreviations.

Most languages also provide an ad-hoc construct to express multiple-choice selection. For example, C++ provides the switch construct, illustrated by the following fragment:



Each branch is labelled by one (or more) constant values. Based on the value of the switch expression, the branch labelled by the same value is selected. If the value of the switch expression does not match any of the labels, the (optional) default branch is executed. If the default branch is not present, no action takes place. The order in which the branches appear in the text is immaterial. In the above example, an explicit break statement is used to terminate each branch; otherwise execution would fall into the next branch.

The same example may be written in Ada as



In Ada, after the selected branch is executed, the entire case statement terminates.Iteration allows a number of actions to be executed repeatedly. Most programming languages provide different kinds of loop constructs to define iteration of actions (called the loop body).

Often, they distinguish between loops where the number of repetitions is known at the start of the loop, and loops where the body is executed repeatedly as long as a condition is met. The former kind of loop is usually called a for loop; the latter is often called the while loop.

For-loops are named after a common statement provided by languages of the Algol family. For statements define a control variable which assumes all values of a given predefined sequence, one after the other. For each value, the loop body is executed.

Pascal allows iterations where control variables can be of any ordinal type: integer, Boolean, character, enumeration, or subranges of them. A loop has the following general appearance:

**for loop\_ctr\_var := lower\_bound to upper\_bound do statement A**

control variable assumes all of its values from the lower to the upper bound. The language prescribes that the control variable and its lower and upper bounds must not be altered in the loop. The value of the control variable is also assumed to be undefined outside the loop.

As an example, consider the following fragment:

**type day = (sun, mon, tue, wed, thu, fri, sat);**

**var week\_day: day;**

**. . . for week\_day := mon to fri do . . .**

 As another example, let us consider how for-loops can be written in C++, by examining the following fragment, where the loop body is executed for all values of i from 0 to 9

**for (int i = 0; i< 10; i++) {. . .}**

The statement is clearly composed of three parts: an initialization and two expressions. The initialization provides the initial state for the loop execution. The first of the two expressions specifies a test, made before each iteration, which causes the loop to be exited if the expression becomes zero (i.e., false). The second specifies the incrementing that is performed after each iteration. In the example, the statement also declares a variable i. Such variable’s scope extends to the end of the block enclosing the for statement.

In C++, either or both of the expressions in a for loop can be omitted. This is used to write an endless loop, as

**for ( ; ; ) { . . .}**

While loops are also named after a common statement provided by languages of the Algol family. A while loop describes any number of iterations of the loop body, including zero. They have the following general form

**while condition**

 **do statement**

For example, the following Pascal fragment describes the evaluation of the greatest common divisor of two variables a and b using Euclid’s algorithm



The end condition (a ¦ b) is evaluated before executing the body of the loop. The loop is exited if a is equal to b, since in such case a is the greatest common divisor. Therefore, the program works also when it is executed in the special case where the initial values of the two variables are equal. In C++, while statements are similar. The general form is:

**while (expression) statement**

Another way to write an endless loop in C++ is therefore

**while (1) {. . .}**

Often languages provide another similar kind of loop, where the loop control variable is checked at the end of the body. In Pascal, the construct has the following general form

**repeat**

**Statement**

**until condition**

In a Pascal repeat loop, the body is iterated as long as the condition evaluates to false. When it becomes true, the loop is exited.

C++ provides a do-while statement which behaves in a similar way:

**do statement**

**while (expression);**

In this case the statement is executed repeatedly until the value of the expression becomes zero (i.e., the condition is false).

Ada has only one general loop structure, with the following form

**Iteration\_specification loop**

**loop\_body**

**end loop**

where iteration\_specification is either

**while condition**

**or**

**for counting\_var in discrete\_range**

**or**

**for counting\_var in reverse discrete\_range**

An example is provided by the following fragment:

**for K in Index\_Range while A (K) /= 0 do B (K) := B (K) / A (K);**

Endless loops are easy to write, since iteration\_specification is optional. In addition, loops can be terminated by an unconditional exit statement

**exit;**

**or**

**by a conditional exit statement**

 exit when condition If the loop is nested within other loops, it is possible to exit an inner loop and any number of enclosing loops.



after the exit statement execution continues here In the example, control is transferred to the statement following the end of Main\_Loop when A is found to be equal to zero in the inner loop. The exit statement is used to specify a premature termination of a loop.

In some cases, it is useful to allow the programmer to define a mechanism to step through the elements of a given collection. To do so, a programming language might provide support for user-defined control structures, in much the same way as it provides support for user-defined types and operations. For example, having defined a set, the programmer might need to sequence through all elements in the set. User-defined control structures which sequence through elements of user-defined collections are sometimes called iterators.

Languages providing constructs for the implementation of abstract data types easily allow iterators to be defined. For example, in C++ let the generic "collection of elements of type T" be defined by a template. To define an iterator, we can design three operations that are exported by the template: start ( ), which initializes the loop by positioning a cursor on the first element of the collection (if any), more ( ), which yields true if there are elements left to examine in the collection, and next ( ), which yields the current element and positions the cursor on the next element of the collection (if any). A typical iteration on an instantiated collection X of elements of type T would be



**4.3 Routines**

Routines are a program decomposition mechanism which allows programs to be broken into several units. Routine calls are control structures that govern the flow of control among program units.

Most languages distinguish between two kinds of routines: procedures and functions. A procedure does not return a value: it is an abstract command which is called to cause some desired state change. The state may change because the value of some parameters transmitted to the procedure gets modified, or because some nonlocal variables are updated by the procedure, or because some actions are performed on the external environment (e.g., reading or writing). A function corresponds to its mathematical counterpart: its activation is supposed to return a value, which depends on the value of the transmitted parameters.

Pascal provides both procedures and functions. It allows formal parameters to be either by value or by reference. It also allows procedures and functions to be parameters, as shown by the following example of a procedure header:

**Procedure example (var x: T; y: Q; function f (z: R): integer);**

In the example, x is a by-reference parameter of type T; y is a by-value parameter of type Q; f is a function parameter which takes one by-value parameter z of type R and returns an integer.

Ada provides both procedures and functions. Parameter passing mode is specified in the header of an Ada routine as either in, out, or in out. If the mode is not specified, in is assumed by default. A formal in parameter is a constant which only permits reading of the value of the corresponding actual parameter. A formal in out parameter is a variable and permits both reading and updating of the value of the associated actual parameter. A formal out parameter is a variable and permits updating of the value of the associated actual parameter.

In C all routines are functional, i.e., they return a value, unless the return type is void, which states explicitly that no value is returned. Parameters can only be passed by value. It is possible, however, to achieve the effect of call by reference through the use of pointers. For example, the following routine

**void proc (int\* x, int y);**

 **{**

 **\*x = \*x + y;**

 **}**

increments the object referenced by x by the value of y. If we call proc as follows

**proc (&a, b);** /\* &a means the address of a \*/ x is initialized to point to a, and the routine increments a by the value of b.

C++ introduced a way of directly specifying call by reference. The previous example would be written in C++ as follows.



**proc (a, b);** -- no address operator is needed in the call

 While Pascal only allows routines to be passed as parameters, C++ and Ada get closer to treating routines as first-class objects. For example, they provide pointers to routines, and allow pointers to be bound dynamically to different routines at run time.

**4.4 Exceptions**

Programmers often write programs under the optimistic assumption that nothing will go wrong when the program executes. An arithmetic expression may cause a division by zero, or the square root operation may be executed with a negative argument. A request for new memory allocation issued by the run-time system might exceed the amount of storage available for the program execution

For example, to check that an index never exceeds the array bounds, one would need to explicitly test the value of the index before any indexing takes place, and insert appropriate response code in case the bounds are violated. Alternatively, one would like the run-time machine to be able to trap such anomalous condition, and let the response to it be programmable in the language. This would be more efficient under the assumption that bound violations are the exceptional case.

To cope with this problem, programming languages provide features for exception handling. According to the standard terminology, an exception denotes an undesirable, anomalous behavior which supposedly occurs rarely. The language can provide facilities to define exceptions, recognize them, and specify the response code that must be executed when the exception is raised (exception handler).

Earlier programming languages (except PL/I) offered no special help in properly handling exceptional conditions. Most modern languages, however, provide systematic exception-handling features. With these features, the concern for anomalies may be moved out of the main line of program flow, so as not to obscure the basic algorithm.

To define exception handling, the following main decisions must be taken by a programming language designer:

1. What are the exceptions that can be handled? How can they be defined?

 2. What units can raise an exception and how?

3. How and where can a handler be defined?

4. How does control flow after an exception is raised in order to reach its handler?

5. Where does control flow after an exception has been handled?

The solutions provided to such questions, which can differ from language to language, affect the semantics of exception handling, its usability, and its ease of implementation. In this section, we will analyze the solutions provided by C++, Ada, Eiffel, and ML. The exception handling facilities of PL/I and CLU are shown in sidebars.

**4.4.1 Exception handling in Ada**

 Ada provides a set of four predefined exceptions that can be automatically trapped and raised by the underlying run-time machine:

**• Constraint\_Error**: failure of a run-time check on a constraint, such as array index out of bounds, zero right operand of a division, etc.;

**• Program\_Error**: failure of a run-time check on a language rule. For example, a function is required to complete normally by executing a return statement which transmits a result back to the caller. If this does not happen, the exception is raised;

 • **Storage\_Error**: failure of a run-time check on memory availability; for example, it may be raised by invocation of new;

 • **Tasking\_Error**: failure of a run-time check on the task system

A program unit can declare new exceptions, such as

For Example:



In the example, a list of handlers is attached to the block. The list is prefixed by the keyword exception, and each handler is prefixed by the keyword when.

If the unit that raises the exception provides a handler for it, control is transferred immediately to that handler: the actions following the point at which the exception is raised are skipped, the handler is executed, and then the program continues execution normally from the statement that follows the handler. If the currently executing unit U does not provide a handler, the unit terminates and the exception is propagated.

As an example, consider the program sketched in Figure 51. The figure shows the overall structure of the program, ignoring all internal details. In particular, procedures are described by showing the scope they define by using solid lines, while blocks’ scopes are shown by dashed lines.

Suppose that the following sequence of unit activations occurs:

• Main is activated

• block 1 is entered

 • block 2 is entered

 • Proc1 is called

 • Proc2 is called

 • block 3 is entered

• block 4 is entered

 If an exception is raised at this stage, execution of block 4 is abandoned and a



check is performed to see if the block provides an exception handler that can handle the exception. If a handler is found, the handler is executed and, if no further exceptions are raised, execution continues from the statements that

FIGURE 51.An example of an Ada program which raises an exception

Main Proc2 Proc1

block1

block 2

block 3

block 4

block 5

Instruction raising the exception

call to Proc1

call to Proc2

follow block 4.

 If not, the exception is propagated to the enclosing block 3. That is, execution of block 3 is abandoned, and a check for an exception handler provided by block 3 is performed. If a handler is provided, and its execution terminates normally, procedure Proc2 returns to its caller normally. If not, the exception is propagated to the caller, and thus execution of block 2 is abandoned. If no exception handlers are provided by procedure Proc1, block 2, and block 1, eventually the Main program terminates abnormally.

Each descriptor in the table contains

1. the internal exception name handled by the handler;

 2. a pointer to the handler body.

 When an exception is raised, its code is used to search for a handler in the handler table. If it is found there, control is transferred to its body. If not, the activation record is deleted from the stack, and the search is performed in the caller’s handler table using the address of the return point.

**4.4.2 Exception handling in C++**

 Exceptions may be generated by the run-time environment (e.g., due to a division by zero) or may be explicitly raised by the program. An exception is raised by a throw instruction, which transfers an object to the corresponding handler. A handler may be attached to any piece of code (a block) which needs to be fault tolerant. To do so, the block must be prefixed by the keyword try. As an example, consider the following simple case:



A throw expression causes the execution of the block to be abandoned, and control to be transferred to the appropriate handler. It also initializes a temporary object of the type of the operand of throw and uses the temporary to initialize the variable named in the handler. In the example, Help (MSG1) actually invokes the constructor of class Help passing a parameter which is used by the constructor to initialize field kind. The temporary object so created is used to initialize the formal parameter msg of the matching catch, and control is then transferred to the first branch (case MSG1) of the switch in the first handler attached to the block.

The above block might call routines which, in turn may raise exceptions. If one such routine raises a–say–help request and does not provide a handler for it, the routine’s execution is abandoned and the exception is propagated to the point of call within the block. Execution of the block, in turn, is abandoned, and control is transferred to the handler as in the previous case. In other terms, C++, like Ada, propagates unhandled exceptions. Like Ada, a caught exception can be propagated explicitly, by simply saying throw. Also, as in Ada, after a handler is executed, execution continues from the statement that follows the one to which the matched handler is attached.

C++ routines may list in their interface the exeception they may raise. This feature allows a programmer to state the intent of a routine in a precise way, by specifying both the expected normal behavior (the data it can accept and return), and its abnormal behaviors. For example

void foo ( ) throw (Help, Zerodivide); might be the interface of a function foo which is called within the above fault tolerant block. Knowing that the used function foo may indeed raise exceptions, the client code may guard against anomalous behaviors by providing appropriate exception handling facilities, as we did.

If an exception is repeatedly propagated and no matching handler is ever found, the special function terminate ( ) is called automatically. Its default behavior, which can be redefined by the programmer, eventually aborts the program execution.

Since the exceptions that can be raised in C++ are expressions of a given type, one can use the general facilities available to structure types (and abstract data types) to organize exceptions. For instance, one can use enumerations to structure and classify exceptions in groups. In the previous examples, if only the specific kind of needed help must be provided to handle exceptions of type Help, the following definition would suffice

enum Help {MSG1, MSG2, ...};

and the corresponding catch statement would be rewritten as

catch (Help msg)

 {

switch (msg)

 {

case MSG1:

 . . .;

case MSG2:

 . . .;

. . . }

 . . . }

 Other interesting ways of organizing exceptions can be achieved by organizing the corresponding classes according to subtype hierarchies, by means of subclasses.

Intuitively, an abstract implementation of the C++ mechanism can be similar to what we outlined for Ada. When an exception is raised, the dynamic chain is unwound until the appropriate handler is found. Further comments will be provided in Section 4.4.5.

**4.4.3 Exception handling in Eiffel**

The features provided by Eiffel to support exception handling have been strongly influenced by a set of underlying software design principles that programmers should follow.Thus, exceptions may arise in Eiffel because an assertion is violated (assuming that the program has been compiled under the option that sets runtime checking on). They can also arise because of anomalous states caught by the underlying abstract machine (memory exhausted, dereferencing an uninitialized pointer, ...). Finally, they can arise because a called routine fails (see below for what this means).

To respond to an exception, an exception handler (rescue clause) may be attached to any routine. There are two possible approaches to exception handling, which comply with the contract-based methodology underlying Eiffel programming. The first approach is called organized panic.

As an example, consider the abstract data type NON\_AXIAL\_INT\_POINT that was defined in Figure 4.4 and suppose that the program is compiled with the option "check assertion" on. If any of the operations is called by a client module with parameters that do not satisfy the corresponding precondition (e.g., one of the parameters of make\_point is zero), control is transferred to the implicit empty rescue clause that is attached to all exported operations. This causes propagation of the failure to the object that called the operation with improper arguments. To explain the reason of the failure to the programmer, one might attach rescue clauses to the routines of the class in Figure 4.4 which print out a message describing the reason for the failure, i.e., violation of the precondition.

An alternative approach to organized panic is called retrial. This means that the handler can find an alternative way to fulfil the object’s contract. This is achieved by a statement retry which may appear in the rescue clause and would cause re-execution of the routine’s body. In such a case, if re-execution does not raise an exception, the routine does not fail and the object’s contract would be fulfilled. As an example, suppose that several methods are available to solve a specific task, so that if one of them fails, another can be tried instead; the task only fails if none of the available methods succeeds. This strategy can be stated in Eiffel according to the following scheme:



It is easy to verify that routine try\_several\_methods only fails if all possible methods fail. Otherwise, if one of the methods succeeds, the routine returns normally to its caller.

\*\*\*\*\*\*start sidebar PL/I\*\*\*\*\*\*

PL/I was the first language to introduce exception handling. Exceptions are called CONDITIONS in PL/I. Exception handlers are declared by ON statements:

ON CONDITION (exception\_name) exception\_handler where exception\_handler can be a simple statement or a block. An exception is explicitly raised by the statement

 SIGNAL CONDITION (exception\_name);

 ON ZERODIVIDE BEGIN;

 . . .

END;

Handlers are bound to exceptions dynamically. When an ON unit is encountered during execution, a new binding takes place between an exception and a handler. Once this binding is established, it remains valid until it is overridden by the execution of another ON statement for the same exception, or until termination of the block in which the ON statement is executed. If more than one ON statement for the same exception appears in the same block, each new binding overrides the previous one. If a new ON statement for the same exception appears in an inner block, the new binding remains in force only until the inner block is terminated. When control exits a block, the bindings that existed prior to block entry are reestablished.

When an exception is raised (either automatically or by a SIGNAL statement), the handler currently bound to the exception is executed as if it were a subprogram invoked explicitly at that point. Therefore, unless otherwise specified by the handler, control subsequently will return to the point that issued the SIGNAL.

Enabling a previously disabled exception (or an exception that is not enabled by default) can be specified by prefixing a statement, block, or procedure with the exception name, for example

(ZERODIVIDE) : BEGIN

 . . .

END;

The scope of the prefix is static; it is the statement, block, or procedure to which it is attached. An enabled exception can be explicitly disabled by prefixing a statement, block, or procedure with NO exception\_name. For example

(NOZERODIVIDE) : BEGIN;

 . . .

 END

 \*\*\*\*\*\*end sidebar PL/I\*\*\*\*\*\*

\*\*\*sidebar start Exception handling in CLU

In CLU, exceptions can only be raised by procedures. That is, if a statement raises an exception, the procedure containing the statement returns abnormally by raising the exception. A procedure cannot handle an exception raised by its execution: its caller should be in charge of handling it. The exceptions that a procedure may raise are to be declared in the procedure’s header. This choice is a consequence of the design method that CLU wishes to enforce. Exception handlers can be attached to statements by except clauses having the following syntactic form

<statement> except <handler\_list> end

where<statement> can be any (compound) statement of the language. If the execution of a procedure invocation within <statement> raises an exception, control is transferred to <handler\_list>. A<handler\_list> has the following form

when<exception\_list\_1>: <statement\_1>

 . . .

when<exception\_list\_n>: <statement\_n>

If the raised exception belongs to <exception\_list\_i>, then <statement\_i> (the handler body) is executed. When the execution of the handler body is completed, control passes to the statement that follows the one to which the handler is attached. If statement\_i> contains a call to a unit, another exception may be raised. In such a case, control flows to the except statement that encloses <statement>. If the raised exception is not named in the exception list that should handle it, it is propagated to the enclosing statements. If no handler is found within the procedure that issued the call, the procedure implicitly signals a language-defined exception failure and returns.

\*\*\*sidebar end

**4.4.4 Exception handling in ML**

The functional language ML allows exceptions to be defined, raised, and handled. There are also exceptions that are predefined by the language and raised automatically by the runtime machine while the program is being executed.

As an example, the following declaration introduces an exception

*exceptionNeg*

*which can be raised subsequently in the following function declaration*

*fun fact (n) =*

*if n < 0 then raise Neg*

*else if n = 0 then 1*

*else n \* fact (n - 1)*

A call such as fact (-2) would cause the evaluation of the function to be abandoned, the exception raised and, since no handler is provided, the program to stop by writing the message "Failure: Neg".

Suppose we wish to handle the exception by returning 0 when the function is called with a negative argument. This can be done, for example, by defining the following new function

fun fact\_0 (n) = fact (n) handle Neg => 0;

which uses fact as a subsidiary function. Exceptions that are not handled in a chain of function calls are implicitly propagated. That is, suppose that function fact is called by some function f which does not provide a handler for Neg;

function f, in turn is called by function g, which provides a handler for Neg, in the same way as function fact\_0 does. In such a case, if the evaluation of the following expression:

g (f (fact (-33))) results in 0.

**4.4.5 A comparative evaluation**

The languages we surveyed in the previous sections are good representatives of the different approaches followed by programming languages to provide exception handling. Although the field has matured in the past years and the main design decision to be faced by language designers are now basically restricted to a limited number of possible choices, still there are differences and there is no consensus on a common scheme that languages should adopt. We will compare and evaluate the different solutions adopted by existing languages by examining the questions we posed at the beginning of our discussion, that is:

1. What are the exceptions that can be handled? How can they be defined?

 2. What units can raise an exception and how?

 3. How and where can a handler be defined?

 4. How does control flow after an exception is raised in order to reach its handler (if any)?

 5. Where does control flow after an exception has been handled?

Regarding questions 1 and 2, all languages (except Eiffel) are quite similar. They all allow both built-in and programmer-defined exceptions. The main differences are whether an exception can carry information and how it can do so. In Ada (and PL/I) an exception is basically a named signal, and thus it does not allow any additional information to be passed to the handler along with it. In C++ any desirable data may be passed along with the exception1.

Eiffel follows an original approach in that exception handling has been designed to fit a precise program development discipline. According to such discipline, an exception arises only if a routine fails because of some error. The language also explicitly and precisely defines what may cause a routine to fail. Thus, in most cases there is no need for naming exceptions, nor for providing a raise statement. All that matters is whether a failure that would

*1. Actually in Ada it is possible to pass to the handler information about the exception occurrence, and a number of predefined operations are provided to extract some limited information from the exception occurrence violate the object’s contract occurred in a routine1.*

Exception handlers in both Ada and C++ can be attached to any block. In Eiffel it can be attached to any routine. As an exception is raised, control is transferred to the appropriate handler. To match the raised exception with the corresponding handler, Ada and C++ unwind the run-time stack by following the dynamic chain until the relevant handler (if any) is found. In Eiffel, each routine provides its own handler (either explicitly or implicitly), and the stack is unwound only if the routine fails.

The combination of static scope rules for exception declarations, adopted by languages like Ada and C++, with dynamic binding between an exception raised by some unit and its handler cause subtlelties that can make programs hard to read. We illustrate the point in the case of C++, in order to show the reader how different language features may interfere with each other, thus making language semantics and language implementation more complex.



If when f is called by foo the exception is thrown and not handled by f, propagation reaches the catch point in File 2. The scope rules of the language, however, are such that the parameter of the catch and the object thrown are bound to different types, and therefore the match does not occur, and the exception is further propagated. Besides affecting understandability, ease and efficiency of the implementation are also affected. Type information, in fact, must be kept to perform the required run-time binding.

The last important point about exception handling is where control should flow after an exception is handled. There are essentially two possible solutions, which corresponds to different styles of handling exception: termination and resumption. The resumption scheme implies that the handler’s code may cause control to return to the point where the exception was raised, whereas the termination scheme does not allow that.

Eiffel is different from all other languages with respect to termination and resumption. Termination in Eiffel is stronger than in other languages. In fact, after control is transferred to a rescue clause which does not contain a retry, completion of the clause implies that the routine fails, and the failure is notified to the caller

**4.5 Pattern matching**

Pattern matching is a high level way of stating conditions, based on which, different actions are specified to occur. Pattern matching is the most important control structure of the string manipulation programming language SNOBOL4 (see sidebar). Pattern matching is also provided by most modern functional programming languages, like ML, Miranda, SASL, and is also provided by the logical language PROLOG and by rule-based systems.

Let us start by discussing the following simple definitions of a data type and a function:

*datatype day = Mon | Tue | Wed | Thu | Fri | Sat | Sun*

*funday\_off (Sun) = true*

*|day\_off (Sat )= true*

*|day\_off ( \_ ) = false*

In the example, function day\_off is defined by a number of cases. Depending on the value of the parameter with which the function will be invoked, the appropriate case will be selected. Cases are checked sequentially, from the first one on. If the first two cases are not matched by the value of the parameter with which the function is called, the third alternative will be selected, since the so-called wild card "\_" matches any argument.

As another example, consider the following function definition:

*fun reverse (nil) = nil*

 *| reverse (head::tail) = reverse(tail) @ [head]*

 In this case, if the argument is an empty list, then reverse is defined to return the empty list. Otherwise, suppose that the argument is the list [1, 0, 5, 99, 2]. As a result of pattern matching [1, 0, 5, 99, 2] with head::tail, head is bound to 1 and tail is bound to [0, 5, 99, 2]. Thus the result of reverse is the concatenation (operator @) of reverse ([0, 5, 99, 2]) with the list [1].

As a final example, suppose that a new operation to reverse lists is to be defined, such that a (sub)list remains unchanged if its first element is zero. The following function rev would do the job:

fun rev(nil) = nil | rev(0::tail) = [0] @ tail

| rev(head::tail) = rev(tail) @ [head]

In this case, since pattern matching examines the various alternatives sequentially, if the function is invoked with a non-empty list whose first element is zero, the second alternative would be selected. Otherwise, for a non-empty list whose first element is not zero, the third alternative would be selected. As the example shows, pattern matching has a two fold effect. On the one hand, it chooses the course of action based on the argument; on the other, since the pattern can be an expression with variables, it binds the variables in the pattern (if any) with the values that match.

The same bound variables can then be used in the expression that defines the value of the function. Pattern matching can thus be viewed as a generalization of conventional parameter passing. The value of actual parameters is used to match the pattern appearing in the formal parameter part. Thus the case selected by pattern matching can vary from call to call. More will be said on pattern matching for ML in Chapter 7. Chapter 8 addresses pattern matching in the case of Prolog.

SNOBOL4 is a string-oriented language in that character strings are the most important primitive data type with many built-in operations. A pattern is a data structure that specifies a set of strings. A pattern is used in pattern matching statements to examine a subject string for the presence of a pattern. For example, the statement

*MESSAGE PAT*

means "search the string MESSAGE for the occurrence of the pattern PAT." If, previous to this statement, we had executed these two assignment statements:

*MESSAGE = 'THERE ARE NO ERRORS HERE.'*

 *PAT = 'ERROR'*

then the above pattern-matching statement will succeed. The notion of success and failure of statements is used in SNOBOL4 to control the flow of execution in a program. Each statement can specify labels of target state- ments for success, failure, or unconditionally. For example

*MESSAGE PAT : S (OK) F (NOTFOUND)*

will transfer control to the statement labelled OK if the pattern-matching succeeds and to NOTFOUND otherwise. The pattern PAT is the simplest kind of pattern we can have–simply one string. We may specify a pattern as a choice among a number of patterns:

SUBJECT = 'I' | 'YOU' | 'WE'

 Now SUBJECT will match any string that contains 'I', ’YOU’ or 'WE'. A pattern may be defined also as a concatenation of other patterns:

SENTENCE = SUBJECT VERB OBJECT '.'

 The pattern SENTENCE will match any string that contains the patterns SUBJECT, VERB, OBJECT, followed by a period. We can then define patterns for SUBJECT, VERB, and OBJECT:

VERB = 'EAT' | 'TAKE'

 OBJECT = 'FOOD' | 'THE SPOON' | 'THE CAR'

 The set of patterns defines the grammar of a tiny and highly simplified subset of the English language. For example, the grammar can represent strings such as

I TAKE THE CAR

 YOU EAT FOOD

 Pattern matching can recognize the sentences that are grammatically correct. In fact, the statement

TEST SENTENCE

will succeed if a valid sentence (according to our grammar) occurs in the string TEST. This pattern will actually match a sentence anywhere in the string but SNOBOL4 provides facilities to constrain the pattern further, for example, to have one sentence and nothing more.

**4.6 Nondeterminism and backtracking:**

Problem solutions can often be described via and-or decompositions into subproblems. For example, to solve problem A, one needs to solve either B, C, or D; to solve–say–C, one needs to solve E, F, and G. This can be represented as an and/or tree (see Figure 52). Node A, which has no incoming arcs, is called a root node; nodes B, D, E, F, and G, which have no exiting arcs, are called leaf nodes. And/or decompositions can also be described–in the hypothetical syntax of some programming language–as





The solution of A is described as a disjunction of subproblems; the solution of C is described as a conjunction of subproblems. We can further assume B, D, E, F, and G to be problem solving routines, which can terminate in either a success or a failure state.

If the order in which subproblems are solved is unspecified (and irrelevant as far as the problem statement is concerned), we say that the program is nondeterministic. In the example, this means that the order in which B, C, or D are tried does not matter. Similarly, the way in which E, F, and G are tried does not matter. The only thing that matters is that a solution be found, if it exists, or a notification of failure is delivered to the request to solve A, if no solution exists. The latter case happens if all three subproblems in the disjunct fail, which means also that at least one of the subproblems of the conjunction failed.

**4.7 Event-driven computations**

In some cases, programs are structured conveniently as reactive systems, i.e., systems where certain events occurring in the environment cause certain program fragments to be executed. An example is provided by modern user interfaces where a number of small graphical devices (called widgets) are often displayed to mediate human-computer interaction. By operating on such widgets (e.g., by clicking the mouse on a push-button) the user generates an event. The event, in turn causes a certain application fragment to be executed. Execution of such a fragment may cause further screen layouts to be generated with a new context of available widgets on it.

Another common event-driven control paradigm is the one based on so-called triggers. Triggers became popular in recent years, in conjunction with new developments in the field of so-called active data bases. An active data base consists of a conventional underlying (passive) data base and a set of active rules (or triggers) of the following form

**on event**

**when condition**

**do action**

 When the event associated with the rule occurs, we say that the rule is triggered. A triggered rule is then checked to see if the condition holds. If this is the case, the rule can be executed.

As an example, the following trigger specifies that the total number of employees should be updated as a new employee record is inserted in the data base.

**on insert in EMPLOYEE**

**when TRUE**

**doemp\_number ++**

 As another example, in a database application, triggers may be used to specify some constraints that must be verified as new elements are inserted or existing elements are updated or deleted from the database. For example, a constraint might be that no employee can have a salary that is more than the average salary of managers. A trigger might watch that no insertion, update, or deletion violates the constraint; if that happens, some appropriate action would be undertaken.

A trigger-based problem solution can be viewed as a high-level design, which is then implemented in any programming language using the conventional constructs it provides.

**4.8 Concurrent computations**

Sometimes it is convenient to structure our software as a set of concurrent units which execute in parallel. This can occur when the program is executed on a computer with multiple CPU’s (multiprocessor). In such a case, if the number of processors coincides with the number of concurrent units.

Concurrency is an important area of computer science, which is often studied in different context: machine architectures, operating systems, distributed systems, databases, etc. In this section we give an overview of how programming languages support concurrency. Concurrent programs support concurrency by allowing a number of units (called processes) to execute in parallel

 (logically or physically).

If the abstract machine that executes the program does not support concurrency, it is possible to simulate it by transferring control explicitly from one unit to another. This low-level approach is supported bycoroutines, reviewed in the sidebar.

\*\*\*Coroutine sidebar start

Coroutines are a low-level construct for describing pseudo-concurrent units. They can be used to simulate parallelism on a uniprocessor by explicitly interleaving the execution of a set of units. Therefore, they do not describe a set of concurrent units, but a particular way of sharing the processor to simulate concurrency.

Coroutines can be viewed as program units that activate one another explicitly, via a resume primitive. At any time, only one unit is executing. When a unit is executing, control may be explicitly transferred to another unit (via resume), which resumes execution at the place where it last terminated. Consequently, units activate each other explicitly in an interleaved fashion, according to a predefined pattern of behavior.

As an example, consider the the two coroutines client and give\_me\_next shown in Figure 53, written in a hypothetical, self-explaining programing language. Unit client repeatedly activates unit give\_me\_next to get the next value of a variable. Each reactivation of unit give\_me\_next produces a new value, which depends on the previously generated value. The two units resume one another. There is a global variable i, which is shared by client and give\_me\_next. Unit main, which is activated initially, resumes client.



An abstract implementation model of coroutines differs from the case of routines. When a coroutineA issues a resume to a coroutine B, one must save (in A's activation record) the pointer to the instruction following the instruction resume B. Moreover, A's activation record is not deallocated. If coroutines can have nested units that may be activated recursively, each coroutine requires an activation record stack that can grow and shrink independently of the other stacks. In addition, as in the example, they may access the global environment.

\*\*\*Coroutine end

To support correct interaction among processes, a language should provide suitable synchronization statements (or primitives). We introduce this concept through an example. Suppose that a certain system contains concurrent activities of the following two kinds: producers and consumers. A producer produces a stream of values and places them into a suitable data structure (a buffer of a certain size, N). A consumer reads these values from the buffer in the same order as they are produced and then processes such values according to some policy.

This example is a classic standard problem that exhibits many relevant issues of concurrency. An abstract description of two producer and consumer processes is shown in Figure 54. A given system might contain many such processes, and all might interact through the same buffer.

unit client { intstop\_value = . . .; ... while (i != stop\_value) { . . . resume give\_me\_next; } }

FIGURE 53. An example of coroutines

unit give\_me\_next { int step ( ) { . . . }; . . . for ( ; ; ) { i += step ( ); resume client; }

main { resume client; }

inti = 0; //global variable



The two processes in Figure 54 are described by cyclic and, ideally, nonterminating program units, which cooperate to achieve the common goal of transferring data from the producer (which could be reading them from an input device) to the consumer (which could be storing them in a file). The buffering mechanism allows the two processes to proceed at their own speeds, by smoothing the effect of their variations. To guarantee the correctness of the cooperation, however, the programmer must ensure that no matter how quickly or slowly the producer and the consumer progress, there will be no attempts to write into a full buffer or to read from an empty buffer.

This can be accomplished by the use of synchronization statements. In general, synchronization statements allow a process to be delayed in the execution of an operation, whenever that is necessary for correct cooperation with other concurrent units. In the example, when the buffer is full, the producer is delayed if it tries to append an element, until the consumer removes at least one element. Similarly, when the buffer is empty, the consumer is delayed if it tries to remove an item, until the producer appends at least one new element.

Another, more subtle, need for synchronization may arise when several activities can legally have access to the same buffer. For example, suppose that append and remove are implemented by the fragments in Figure 55:



where t represents the total number of elements stored in the buffer, next\_in and next\_out are two operations that yield the value of the buffer index where the next element can be stored and where the next element is to be read from, respectively.

Let us assume that the individual statements in Figure 55 are indivisible instructions of the abstract machine, in the sense that if one such action starts to execute, it is guaranteed to finish before any other instruction execution is started. The sequences, however, cannot be assumed to be indivisible, i.e., the execution of their constituent actions may be interleaved by the underlying machine. As an example, suppose that the buffer is initially empty. A producer might start depositing into the buffer by performing the first two actions. The total number of buffered items becomes 1 and the index of the position where the item should be deposited is evaluated.

Suppose that at this point another producer gets access to the buffer (since the buffer is not full). If this producer completes all three actions, the value will be deposited in the second buffer slot (since the first one was acquired by the first producer who did not complete its own deposit). At this point, a consumer might access the buffer (which is not empty, since t = 2). This is an error, however, because the consumer would read its value from the position which was assigned to the first producer, but no assignment was ever performed to such position. To avoid this error, we say that the two statement sequences must be executed in mutual exclusion; synchronization primitives must allow mutual exclusion to be specified.

In general, synchronization primitives may be viewed as mechanisms that constrain the order in which operations performed by different processes are executed. Let {P1, P2, . . .,Pn} be a set of concurrent processes. Each process can be assigned for execution to an abstract machine, like SIMPLESEM that was discussed in Chapter 2. Let ipi be the value of the instruction pointer of the i-th abstract machine which executes Pi; ipi yields the address of the instruction Ci(ipi) which is to be executed next in each process i. If the processes are logically independent, at any instant, all machines can execute Ci(ipi). Synchronization, however, may force some abstract machines to remain in idle until some condition is met that allows them to resume execution.

**4.8.1 Processes**

A concurrent programming language must provide constructs to define processes. Processes can belong to a type, of which several instances can be created. The language must define how and by whom a process is initiated, i.e., a new independent execution flow is spawned by an executing unit. It also need to address the issue of process termination, i.e., how can a process be terminated, what happens after a process terminates, etc.

In this section we will briefly review the main concepts and solutions provided by Ada. In Ada, processes are called tasks. The execution of an Ada program consists of the execution of one or more tasks, each representing a separate computation that proceeds concurrently with other tasks, with which it may interact through synchronization statements.

Tasks can be defined by a task type, of which many instances can be declared. It is also possible to declare a task object (shortly, a task) directly. The declaration of a task (type) specifies how the task (or all instances of the type) can interact with other tasks. As we will see shortly, interaction with a task can be achieved by calling one of its entries, which must appear in its declaration. Thus, the declaration of a task type is a declaration of an abstract data type; entries represent the operations available for interaction with task objects. In Ada, the body of the task (type) , which describes the implementation of the task’s internal code, can be described separately from its declaration.

This is an example of a task type declaration:



Tasks A and B are activated as the block in which they are locally declared is entered at run time. The task pointed at by HER\_SERVER\_PTR is activated by the execution of the new operation.

The concept of task termination is more complex, and will not described in all its subtelties. For simplicity, let us assume that a task can terminate when it reaches the last statement of its body and

 (1) all of the locally declared task objects have terminated, and

 (2) tasks allocated by a new and referenced only by pointers local to the task have terminated.

**4.8.2 Synchronization and communication**

In this section we present some elementary mechanisms for process synchronization and interprocess communication: semaphores , signals and monitors, and rendezvous. Semaphores are low level synchronization mechanisms that are mainly used when interprocess communication occurs via shared variables. Monitors are higher level constructs that define abstract objects used for interprocess communication; synchronization is achieved via signals. Finally, rendevous is another mechanism that combines synchronization and communication via message passing.

***4.8.2.1 Semaphores***

A semaphore is a data object that can assume an integer value and can be operated on by the primitives P and V. The semaphore is initialized to a certain integer value when it is declared.

The definitions of P and V are

P (s): if s>0 then s = s - 1 else suspend current process

V (s): if there is a process suspended on the semaphore then wake up process else s = s + 1

The primitives P and V are assumed to be indivisible, atomic operations; that is, only one process at a time can be executing P or V operations on the same semaphore. This must be guaranteed by the underlying implementation, which should make P and V behave like elementary machine instructions.

The semaphore has (1) an associated data structure where the descriptors of processes suspended on the semaphore are recorded, and (2) a policy for selecting one process to be woken up when required by the primitive V. Usually, the data structure is a queue served on a first-in/first-out basis. However, it is also possible to assign priorities to processes and devise more complex policies based on such priorities.

The simple producer-consumer example of Figure 54 can be solved using semaphores as shown in (as usual, we adopt an arbitrary, self-explanatory C like notation). The keyword process starts the segments of code that can proceed concurrently. Three semaphores are introduced. Semaphores spaces and in are used to guarantee the logical correctness of the accesses to the buffer. In particular, spaces (number of available free positions in the buffer) suspends the producer when it tries to insert a new item into a full buffer. Similarly, in (number of items already in the buffer) suspends the consumer if it tries to remove an item from an empty buffer.



Semaphore mutex is used to enforce mutual exclusion of accesses to the buffer. We can see that semaphores are used both for pure synchronization, as in mutex, to ensure that only one process may use the buffer at a time, and for a kind of communication among processes. For example, V (spaces) by the consumer communicates to the producer that it has consumed an item and that more space is now available in the buffer.

Programming with semaphores requires the programmer to associate one semaphore with each synchronization condition. Our example shows that semaphores are a simple but low-level mechanism, their use can be awkward in practice, and the resulting programs are often difficult to design and understand. Moreover, little checking can be done statically on programs that use semaphores.

For example, a compiler would not be able to catch the incorrect use of a semaphore, such as one resulting from a change of V (mutex) into P (mutex) in the producer process (see Exercise 16). Catching such an error is impossible because it requires the translator to know the semantics of the program, that is, that the operations on the buffer are to be executed in mutual exclusion, and mutex is used to guarantee such mutual exclusion. Therefore, semaphores require considerable discipline on the part of the programmer. For example, one should not forget to execute a P before accessing a shared resource, or neglect to execute a V to release it.

Using semaphores for synchronization purposes other than mutual exclusion is even more awkward. In the producer-consumer example, process consumer suspends itself by executing P (spaces) when the buffer is full. It is the responsibility of some other piece of code, the consumer in this case) to provide the matching V operation. If the programmer forgets to write a V (spaces) after each consumption, the producer will become blocked forever.

Semaphores are often provided by operating systems to support systems programming. They have also been integrated into a number of existing programming languages, such as PL/I and Algol 68 (see sidebar).

\*\*\*sidebar start

PL/I was the first language to allow concurrent units, called tasks. A procedure may be invoked as a task, in which case it executes concurrently with its caller. Tasks also can be assigned priorities. Synchronization is achieved by the use of events, which are binary semaphores that only can assume one of two values: '0'B and '1'B (Boolean constants 0 and 1).

A P operation on a semaphore is represented by a WAIT operation on the completion of an event E: WAIT (E). A V operation is represented by signaling the completion of the event: COMPLETION (E) = '1'B. PL/I extends the notion of semaphores by allowing the WAIT operation to name several events and an integer expression e. The process will be suspended until any e events have been completed. For example, WAIT (El, E2, E3) (1) indicates the waiting for any one of the events: El, E2, or E3.

ALGOL 68 supports concurrent processes in a parallel clause whose constituent statements are elaborated concurrently. Synchronization can be provided by semaphores, which are data objects of type sema.

\*\*\*Sidebar end

***4.8.2.2 Monitors and signals***

Concurrent Pascal introduced the signal and monitor constructs into the programming languages. Signals are synchronization primitives; monitors describe abstract data types in a concurrent environment. The operations that manipulate the data structure are guaranteed to be executed in mutual exclusion by the underlying implementation. Cooperation in accessing the shared data structure must be programmed explicitly by using the monitor signal primitives delay and continue.

Using the notation of Concurrent Pascal, the program in Figure 57 illustrates the use of monitors in the producer-consumer example.



An instance of the monitor (i.e., a buffer) can be declared as var buffer: fifo storage and can be created by the statement init buffer. Monitor instances are abstract objects through which inter process communication and synchronization is coordinated.

The init statement allocates storage for the variables defined within the monitor definition (i.e., contents–the contents of the buffer, tot–the total number of buffered items, and in and out–the positions at which the next items will be appended and removed, respectively) and executes the initialization part (which sets tot to zero, and in and out to one). The monitor defines the two procedures, append and remove. They are declared with the keyword entry, which means that they are the only exported procedures that can be used to manipulate monitor instances. Cooperation between the producer and the consumer is achieved by using the synchronization primitive signals delay and continue.

The operation delay (sender) suspends the executing process (e.g., the producer) in the queue sender. The process loses its exclusive access to the monitor's data structure and its execution is delayed until another process (e.g., the consumer) executes the operation continue (sender). Similarly, with delay (receiver) a consumer process is delayed in the queue receiver if the buffer is empty, until the producer resumes it by executing the instruction continue (receiver). The execution of the continue (q) operation makes the calling process return from the monitor call and, additionally, if there are processes waiting in the queue q, one of them immediately will resume the execution of the monitor procedure that previously delayed it.

The structure of a Concurrent Pascal program that uses the above monitor to represent cooperation between a producer and a consumer is given in Figure 58.



Processes are described in the example as non terminating, cyclic activities (cycle...end). Two particular instances (me producer and you consumer) are declared as bound to an instance of the resource type fifo storage and subsequently activated as concurrent processes by the init statement.

***4.8.2.3 Rendezvous***

The examples given so far used shared memory for interprocess communication. A globally accessible buffer was used by producer and consumer processes, and suitable synchronization promitives were introduced to allow them to proceed safely. In the example using semaphores, synchronization and communication features were separate. Semaphores are used for synchronization; shared variables are used for communication. In the monitor example, the two issues were more intertwined, and the resulting construct is higher level.

One can view the monitor construct as defined by two logical components: an abstract object which is used for communication among processes in mutual exclusion, and a signal mechanism that supports synchronization (e.g., the ability to delay and resume processes, based on some logical condition). Note that while the first component is intrinsically based on a shared memory computation paradigm, the second is not, and might be used also in a decentralized scheme for concurrent computation.

In this section we illustrate the rendezvous concept introduced by the Ada programming language. The construct can be viewed as a high-level mechanism that combines synchronization and communication, where communication is based on the message passing conceptual paradigm. The construct, per se, can be naturally used to write software for distributed architectures, although its possible interaction with other features in Ada can make this quite difficult. Hereafter we concentrate on the basic properties of rendezvous; additional features and the interaction with other facilities provided by the language (such as scope rules and exception handling), will be ignored for the sake of simplicity.

The Ada task object in Figure 59 describes a process that handles the operations append and remove on a buffer. The declaration of task Buffer\_Handler specifies Append and Remove as entries. An entry can be viewed as a port, through which a task can send a message to another task, which can then accept it. The task can indicate its willingness to accept a message if it is an owner of the corresponding entry (i.e., the entry is declared in it). It does so by executing the accept statement. At this point, the sender and the receiver tasks can be viewed as meeting together (in French, they perform a rendezvous).

If the sender calls the entry (i.e., sends the message) before the receiver issues an accept, the sender is suspended until the rendezvous occurs. Similarly, a suspension of the receiver occurs if an accept statement is executed before the corresponding entry is called (i.e., before the message is sent). Note that a task can accept messages from more than one task; consequently, each entry potentially has a queue of tasks which sent messages to it.

The accept statement is similar to a routine. After a repetition of the header of the entry, the do. . .end part (accept body) specifies the statements to be executed at the rendezvous. Once a match between an entry call and the corresponding accept occurs, the sender is suspended until the accept body is executed by the called task. The accept body is the only place at which the parameters of the entry are accessible. Possible out parameters (as in the case of REMOVE) are passed back to the sender at the end of the rendezvous, that is, when the execution of the accept body is completed. Thereafter, the two tasks that met in the rendezvous can proceed in parallel.



The bodies of tasks PRODUCER and CONSUMER, which interact with BUFFER\_HANDLER in the producer-consumer example, are sketched in Figure 60.



In the example of Figure 59, accept statements are enclosed within a select statement. The select statement specifies several alternatives, separated by or, that can be chosen in a nondeterministic fashion. The Ada selection is specified by an accept statement, possibly prefixed (as in our example) by a when condition. Execution of the select statement proceeds as follows1.

1. The conditions of the when parts of all alternatives are evaluated. Alternatives with a true condition, or without a when part, are considered open; otherwise, they are considered closed. In the example, both alternatives are open if 0 < TOT < N.

 2. An open alternative can be selected if a rendezvous is possible (i.e. an entry call already has been issued by another task). After the alternative is selected, the corresponding accept body is executed.

 3. If there are open alternatives but none can be selected immediately, the task waits until a rendezvous is possible.

4. If there are no open alternatives, an error condition is signaled by the language-defined exception PROGRAM\_ERROR.

***4.8.2.4 Summing up***

Semaphores, monitors, and rendezvous are all primitives for modeling concurrent systems. As we pointed out, semaphores are rather low-level mechanisms: programs are difficult to read and write, and few checks on their correct use can be done automatically. Monitors, on the other hand, are a higher-level structuring mechanism. Using monitors, a typical system structuring proceeds by identifying (1) shared resources as abstract objects with suitable access primitives (passive entities), and (2) processes (active entities) that cooperate through the use of resources.

Resources are encapsulated within monitors. Mutual exclusion on the access to a shared resource is guaranteed automatically by the monitor implementation, but synchronization must be enforced by explicitly suspending and signaling processes via delay and continue statements. The distinction between active and passive entities (processes and monitors, respectively) disappears in a scheme based on rendezvous. Shared resources to be used cooperatively are represented by tasks, that is, by active components representing resource managers. A request to use a resource is represented by an entry call, i.e., by sending a message which must be accepted by the corresponding resource manager.

A system structured via monitors and processes can be re-structured via tasks and rendezvous, and vice versa; the choice between the two schemes is largely dependent on personal taste. As we mentioned, the latter scheme mirrors more directly the behavior of a concurrent system in a distributed architecture, where remote resources are actually managed by processes that behave as guardians of the resource.

 However, it can be somewhat awkward in case processes need to communicate via shared objects. In fact, early experience with the Ada programming language, which initially provided only rendezvous, showed that the need for additional tasks to manage shared data often led to poor performance. Therefore, Ada 95 introduced a kind of monitor construct–protected types–in addition to the rendezvous.

The use of an Ada protected type to implement our running example of a buffer type (Fifo\_Storage) is shown in Figure 61.



Similar to the monitor, operations defined for a protected type are executed by the underlying abstract machine in mutual exclusion. There are two kinds of possible operations: routines (i.e., procedures and functions) and entries. Entries (shown in the above example) have an associated barrier condition which is used for synchronization. Routines have no associated barriers. The difference with the monitor is that no explicit signals are issued. Rather, when an entry is called its barrier is evaluated; if the barrier is false then the calling process is suspended and queued.

At the end of the execution of an entry (or a routine) body, all barriers which have queued tasks are re-evaluated, thus possibly allowing a suspended task whose barrier became true to be resumed. The absence of explicit signals to be exchanged for synchronization purposes makes the construct simpler to use and the corresponding abstraction easier to understand than in the case of monitors.

Ada is perhaps the best example of a programming language which provides a coherent set of well integrated features supporting concurrent programming. Most other languages do not. Such languages often provide support for concurrent programming either via calls to low-level operating system primitives or via libraries added to language implementations.

\*\*\*Linda is a library for C???\*\*\*

\*\*\*To support distributed systems programming, there are libraries supporting remote procedure calls. In such a case\*\*\*\*\*\*\*

\*\*\*task library of C++\*\*\*

\*\*\*sidebar on data concurrency???\*\*\*

***4.8.3 Implementation models***

In a concurrent system, processes either are suspended (waiting on some synchronization condition) or are potentially active, that is, there are no logical obstacles to their execution. In general, only a subset of potentially active processes can be running, unless there are as many processors as there are potentially active processes. In the common case of a uniprocessor, only one of such processes can be running at a time. It is thus customary to say that processes can be in one of the following states (see also Figure 62).

• - Waiting

 • - Ready (i.e., potentially active, but presently not runnning)

• - Running

The state of a process changes from running to waiting if there is some logical condition that prevents the process from continuing its execution. That is, the process is suspended by the execution of some synchronization statement (e.g., the buffer is full for the producer process). The state can later change from waiting to running if some other process performs a suitable synchronization statement (e.g., a consumer process signals that the buffer is not full any more).

In concurrent programming, the programmer has no direct control over the speed of execution of the processes. In particular, the user is not responsible for changing the state of a process from ready to running (operation of selection in Figure 62), which is done by the underlying implementation. Figure 62 shows that a process can leave the running state and enter the ready state as a consequence of the action of preemption.

Preemption is an action performed by the underlying implementation; it forces a process to abandon its running state even if, from a logical point of view, it could safely continue to be executed. A process can be preempted either after it performs a synchronizing statement that makes another suspended process enter the ready state (e.g., a V on a semaphore) or when some other condition occurs, such as the expiration of a specified amount of time (time slice).



After the preemption of one process, one of the ready processes can enter the running state. This kind of implementation allows the programmer to view the system as a set of activities that proceed in parallel, even if they are all executed by the same processor. Only one process at a time can be executed by the processor, but each process runs only for a limited amount of time, after which control is given to another process. It is possible to have nonpreemptive implementation of concurrency. In this case, execution switches to another process only when the currently executing process deliberately suspends itself or requires the use of an unavailable re-source.

The portion of run-time support of a concurrent language responsible for the implementation of the state transitions shown in Figure 62, is called the kernel. To illustrate the basic features of a kernel, consider the case of a single processor shared by a set of processes. For the sake of simplicity, we will ignore the problems of synchronizing processes with input/output devices and concentrate our attention on the interactions among internal processes. More complete discussions of these issues are traditionally (and more properly) addressed in textbooks on operating systems. Here we provide only a glimpse of the basic problems that are relevant to understanding concurrency features of programming languages.

The information about a process needed by the kernel is represented in a process descriptor, one for each process. The descriptor for a process is used to store all the information needed to restore the process from a waiting or blocked state to the running state. This information (called process status) includes the process priority (if priorities are used) and all information required to instruct the processor about the identity and point of execution of the process–notably, the contents of the machine registers (program counter, index registers, accumulator, and so on). Saving the status of the process when the process becomes suspended and restoring the status when the process becomes running is one of the kernel's jobs.

The kernel can be viewed as an abstract data type; it hides some private data structures and provides procedures that provide the only ways to use these data structures. All of the kernel’s operations are assumed to be executed in a noninterruptible way; i.e., all interrupts are disabled while they are being executed. The kernel's private data structures are organized as queues of process descriptors. The descriptors of ready processes are kept by the kernel in READY\_QUEUE. There is also one CONDITION\_QUEUE for each condition that might suspend a process, that is, there is one queue for each semaphore and one for each object declared to be of type queue in a monitor (and for each entry in a protected Ada type).

 Each such queue is used to store the descriptors of all processes suspended on the semaphore or delayed in the queue. A variable RUNNING denotes the descriptor of the running process. A typical snapshot of the kernel's data structures is shown in Figure 5.4. The queues used by the kernel can be considered as instances of an abstract data type whose operations are defined by the following signatures:

enqueue: Queue x Descriptor -> Queue -- inserts a descriptor into the queue

dequeue: Queue -> Queue x Descriptor -- extracts a descriptor from the queue

empty: Queue -> Boolean -- true if the queue is empty; false otherwise

In what follows, we discuss the basic operations performed by the abstract machine to execute concurrency constructs. The notation we use is a self explaining pseudo-code based on C++. Whenever necessary, additional comments are added to the pseudo-code.

Time slicing is implemented by a clock interrupt. Such an interrupt activates the following kernel operation Suspend-and-Select, which suspends the most recently running process into READY\_QUEUE and transfers a ready processinto the running state.



Operation Suspend-and-Select

RUNNING = process\_status; -- save status of running process into RUNNING READY\_QUEUE .enqueue (RUNNING); -- enqueue RUNNING into READY\_QUEUE RUNNING = READY\_QUEUE . dequeue ( ); -- move a descriptor from READY\_QUEUE into RUNNING process\_status = RUNNING; -- activate the new process)

***4.8.3.1 Semaphores***

If semaphores are provided by the language, primitives P and V can be implemented as calls to kernel procedures. A suspension on a condition c caused by a P operation is implemented by the following private operation of the kernel





***4.8.3.2 Monitors and signals***

In the case of monitors and Ada’s protected types, a simple way to implement the required mutual exclusion consists of disabling interrupts when a monitor procedure is called and enabling them on return from the call. We assume that interrupts are enabled and disabled by a single machine instruction, and that a special machine register determines whether interrupts are enabled or disabled. This register is part of the process status and must be saved in the process descriptor when the process is suspended. For the sake of simplicity,

we also assume that monitor procedures do not contain calls to other monitor procedures. When a process calls a monitor procedure, the value of the return point from the call is saved in an entry of the process descriptor. Operations delay and continue can be implemented by kernel procedures. In particular, delay is implemented by operation Suspend-on-Condition, and continue (c), where c is a condition queue, is implemented by the following operation.



Note that we did not state the policies for the management of queues in this abstract implementation. Queues might be handled according to a first-infirst-out policy, or one may even use sophisticated strategies which take into account waiting times and priorities. The part of the kernel responsible for choosing a policy is called the scheduler. In our scheme, the scheduler is a part of the implementation of the abstract data type that defines queues.

***4.8.3.3 Rendezvous***

In this section, we discuss some implementation issues of Ada's rendezvous mechanism. There is one queue of ready tasks (READY\_QUEUE). Each entry has a descriptor that contains the following fields.

• A boolean value O describing whether the entry is open (O = true indicates that the task owning the entry is ready to accept a call to this entry).

 • A reference W to a queue of descriptors of tasks whose calls to the entry are pending (waiting queue).

 • A reference T to the descriptor of the task owning the entry.

 • A reference I to the first instruction of the accept body (to simplify matters, we assume that no two accept statements for the same entry can appear in a select statement).

This reference is significant only if the task owning the entry is ready to accept a call to the entry (that is, O = true). For simplicity, we can assume that the value of this field is a constant, statically associated with the entry.

As usual, we assume that the implementation of the synchronization statements is done by kernel operations that are noninterruptible, that is, interrupts are disabled and enabled by the kernel before and after executing such statements. The problem of passing parameters across tasks is ignored for simplicity.

Let e be an entry that is called by a task, and let DESCR (e) be e’s descriptor. The implementation of a call to entry e can be done by the kernel as follows.







The actions to be executed as a consequence of an accept statement for entry e (not embedded in a select statement) are



To execute a select statement, a list of the open entries involved in the selection is first constructed. If this list is empty, then the exception PROGRAM\_ERROR is raised. Otherwise, the following kernel actions are required.



# STRUCTURING THE PROGRAM

# INTRODUCTION:-

The basic mechanisms described in previous chapters for structuring data (Chapter 3) and computation (Chapter 4) may be used for programming in the small. In Chapter 4, we also have seen the use of control structures for structuring large programs. This chapter deals strictly with issues of programming in the large. We describe the basic concepts for structuring large programs (encapsulation, interfaces, information hiding) and the mechanisms provided by languages to support it (packaging, separate compilation).

We also consider the concept of genericity in building software component libraries. We do not go deeply into object-oriented programming, which is the subject of the next chapter.

The production of large programs—those consisting of more than several thousand lines—presents challenging problems that do not arise when developing smaller programs. The same methods and techniques that work well with small programs just don’t “scale up.” To stress the differences between small and large systems production, we refer to “programming in the small ”and “programming in the large.”

Two fundamental principles—*abstraction* and *modularity*—underlie all approaches to programming in the large. Abstraction allows us to understand and analyze the problem by concentrating on its important aspects. Modularity allows us to design and build the program from smaller pieces called modules.

During problem analysis, we discover and invent abstractions that allow us to understand the problem. During program design and implementation, we try to discover a modular structure for the program. In general, if modules

that implement the program correspond closely to abstractions discovered during problem analysis, the program will be easier to understand and manage.

The principles of modularity and abstraction help us apply the well known problem solving strategy known as “divide and conquer.”

The concept of a “large program” is difficult to define precisely. We certainly do not want to equate the size of a program (e.g., the number of source statements) with its complexity.

Largeness relates more to the “size” and complexity of the problem being solved than to the final size of a program in terms of the number of source lines. Often, however, the size of a program is

a good indication of the complexity of the problem being solved.

Others may access the system to obtain statistical data about a particular flight or all flights. A problem of this magnitude imposes severe restrictions on the solution strategy and the following key requirements:

**•** The system has to function correctly. A seemingly small error, such as assignment to the wrong pointer, may lead to losing a reservation list or interchanging two different lists and be extremely costly.

**5.1 Software design methods**

To combat the complexities of programming in the large, we need a systematic design method that guides us in composing a large program out of smaller units—which we call *modules*. A good design is composed of modules that interact with one another in well-defined and controlled ways. Consequently, each module can be designed, understood, and validated independently of the other modules. Once we have achieved such a design, we need programming language facilities that help us in implementing these independent modules, their relationships, and their interactions.

The goal of software design is to find an appropriate modular decomposition of the desired system. Indeed, even though the boundaries between programming in the large and programming in the small cannot be stated rigorously, we may say that programming in the large addresses the problem of modular system decomposition, and programming in the small refers to the production of individual modules.

A good modular decomposition is one that is based on modules that are as independent from each other as possible. There are many methods for achieving such modularity. A well-known approach is *information hiding* which uses the distribution of “secrets” as the basis for modular decomposition. Each module hides a particular design decision as its secret.

The idea is that if design decisions have to be changed, only the module that “knows” the secret design decision needs to be modified and the other modules remain unaffected.

In Chapter 1 we discussed the importance of software design. If a design is composed of highly independent modules, it supports the requirements of large programs:

**•** Independent modules form the basis of work assignment to individual team members.

The more independent the modules are, the more independently the team members can proceed in their work.

**•** The correctness of the entire system may be based on the correctness of the individual modules. The more independent the modules are, the more easily the correctness of the individual modules may be established.

**•** Defects in the system may be repaired and, in general, the system may be enhanced more easily because modifications may be isolated to individual modules.

**5.2 Concepts in support of modularity**

To summarize the discussion of the last section, the key to software design is modularization. A good module represents a useful abstraction; it interacts with other modules in well-defined and regular ways; it may be understood, designed, implemented, compiled, and enhanced with access to only the specification (not the implementation secrets) of other modules.

Programming languages provide facilities for building programs in terms of constituent modules. In this chapter, we are interested in programming language concepts and facilities that help the programmer in dividing a program into subparts—modules—the relationships among those modules and the extent to which program decompositions can mirror the decomposition of the design.

We have already seen some units of program decomposition in Chapters 3 and 4. Procedures and functions are an effective way of breaking a program into two modules: one which provides a service and another which uses the service. We may say that the procedure is a server or service provider and the caller is a client. Even at this level we can see some of the differences between different types of modularization units. For example, if we provide a service as a function, then the client has to use the service in an expression.

On the other hand, if we provide the service in a procedure, then the client may not use it in an expression and is forced to use a more assignment-oriented or imperative style.

Procedures and functions are units for structuring small programs, perhaps limited to a single file. Sometimes, we may want to organize a set of related functions and procedures together as a unit. For example, we saw in Chapter 3 how the class construct of C++ lets us group together a data structure and related operations. Ada and Modula-2 provide other constructs for this purpose.

**5.2.1 Encapsulation**

A program unit provides a service that may be used by other parts of the program, called the *clients* of the service. The unit is said to *encapsulate* the service. The purpose of encapsulation is to group together the program components that combine to provide a service and to make only the relevant aspects visible to clients. *Information hiding* is a design method that emphasizes the importance of concealing information as the basis for modularization.

Encapsulation mechanisms are linguistic constructs that support the implementation of information hiding modules. Through encapsulation, a module is clearly described by two parts: the specification and the implementation. The specification describes how the services provided by the module can be accessed by clients. The implementation describes the module’s internal secrets that provide the specified services. For example, assume that a program unit implements a dictionary data structure that other units may use to store and retrieve <name, “id”> pairs.

This dictionary unit makes available to its clients operations for: inserting a pair, such as <“Mehdi”, 46>, retrieving elements by supplying the string component of a pair, and deleting elements by supplying the string component of a pair. The unit uses other helper routines and data structures to implement its service. The purpose of encapsulation is to ensure that the internal structure of the dictionary unit is *hidden* from the clients.

By making visible to clients only those parts of the dictionary unit that they need to know, we achieve two important properties.

• The client is simplified: clients do not need to know how the unit works in order to be able to use it; and

• The service implementation is independent of clients and may be modified without affecting the clients.

Different languages provide different encapsulation facilities. For example, in C, a file is the unit of encapsulation. Typically, the entities declared at the head of a file are visible to the functions in that file and are also made available to functions in other files if those functions choose to declare them.

 The declaration:

extern int max;

states that the variable max to be used here, is defined—and storage for it allocated—elsewhere. Variables declared in a C function are local and known only to that function. Variables declared outside of functions are assumed to be available to other units, if they declare them using the extern specifier.



This program declares five publicly available functions. As we know from Chapter 4, the first two functions, dict() and ~dict(), may be used to create and clean up a dictionary object, respectively. The other three functions may be used to access the dictionary object. The private part of the class defines the representation of a node of a dictionary and the root of the dictionary. This part of the declaration is not visible to the users of the class.

The module encapsulates the dictionary service, both providing access and hiding unnecessary details. As we have seen also in Chapter 3, the built-in types of a language are examples of encapsulated types. They hide the representation of the instances of those types and allow only legal operations to be performed on those instances.

**5.2.2 Interface and implementation**

A module encapsulates a set of entities and provides access to some of those entities. The available entities are said to be *exported* by the module. Each of the exported entities is available through an interface. The collection of the interfaces of the exported entities form the *module interface*. Clients request the services provided by a module using the module’s *interface*, which describes the module’s specification. The interface specifies the syntax of service requests. Some languages also support or require the specification of the

interface’s semantic requirements. The idea is that the interface is all that the client needs to know about the provider’s unit.

The implementation of the unit is hidden from the client. The separation of the interface from the implementation contributes to the independence of the client and the server from one another.

A service provider *exports* a set of entities to its clients. A client module *imports* those entities to be able to use the services of the provider module. The exported entities comprise the service provided by the module. Some languages have implicit and others explicit mechanisms for import and export of entities. Languages also differ with respect to the kinds of entities they allow

to be exported.

 For example, some languages allow a type to be exported and others do not. The simplest interface, one that we have already seen in Chapter 4, is a procedure or function interface. A function declaration such as:

int max (int& x, int& y)

specifies to the clients that the function max may be called by passing to it two integers; the function will return an integer result.

We introduced the term *signature* to refer to these requirements on input and output for procedures and functions. Procedure signatures form the basis of type-checking across procedures. The name of the function, max, is intended to convey something about the semantics of the function, namely that the integer it will return is the maximum of the two integer input parameters. Ideally, the interface would specify the semantics and the requirements on parameters (for example that they must be positive integers). Most programming languages do not support such facilities,

The package body, as can be seen in the figure, defines both the implementation of the entities defined in the package interface and the implementation of other entities internal to the module. These entities are completely hidden from the clients of the package. The package specification and the package body may appear in different files and need not even be compiled together.

There are significant differences between the packages of Ada and classes of C++. Even from this simple example we can see a difference between the models supported by C++ and Ada. In C++, the client can declare several instances of the dictionary class. In Ada, on the other hand, a module may be declared once only and the client obtains access to only a single dictionary.

**package** Dictionary **is**

**procedure** insert (C:String; I: Integer);

**function** lookup(C:String): Integer;

**procedure** remove (C: String);

**end** Dictionary;

**FIGURE 65.**Package specification in Ada



**FIGURE 66.**Package body in Ada

**5.2.3 Separate and independent compilation**

The idea of modularity is to enable the construction of large programs out ofsmaller parts that are developed independently. At the implementation level, independent development of modules implies that they may be compiled and tested individually, independently of the rest of the program. This is referred to as *independent* compilation. The term *separate* compilation is used to refer to the ability to compile units individually but subject to certain ordering constraints. For example, C supports independent compilation and Ada supports separate compilation. In Ada, as we will see later, some units may not be compiled until other units have been compiled. The ordering is imposed to allow checking of interunit references. With independent compilation, normally there is no static checking of entities imported by a module. To illustrate this point, consider the program sketch in Figure 67, written in a hypothetical programming language. Separate compilation means that unit B,which imports routine X from unit A, must be compiled after A.

This allows any call to X issued by B to be checked statically against X’s definition in A. If the language allows module interfaces to be compiled separately from their bodies, only A’s interface must be compiled before B; its body can be compiled at any time after its corresponding interface has been compiled.



Independent or separate compilation is a necessity in the development of large programs because it allows different programmers to work concurrently on different parts of the program. It is also impractical to recompile thousands of modules when only a few modules have changed. Language concepts and features are available to allow implementations to determine the fewest number of units that must be recompiled. In general, programming languages define:

• the unit of compilation: what may be compiled independently?

• the order of compilation: are compilation units required to be compiled in any particular order?

• amount of checking between separately-compiled modules: are inter-unit interactions checked for validity?

The issue of separate compilation is at the border of the language definition and its implementation. Clearly, if the language requires inter-unit checking to be performed, this implies a programming environment that is able to check module implementations against the interfaces of compilation units from which they import services, for example a type-checking linker. Interface- checking of separately compiled modules is analogous to static type-checking for programming in the small: both are aimed at the development of safe and reliable programs.

**5.2.4 Libraries of modules**

We have seen that C++ class and Ada’s package make it possible to group related entities into a single unit. But large programs consist of hundreds or even thousands of such units. To control the complexity of dealing with the large number of entities exported by all these units, it is essential to be able to organize these units into related groups. For example, it is difficult to ensure that all the thousands of units have unique names! In general, we can always find groupings of units that are related rather closely.

A common example of a grouping of related services is a library of modules such as a library of matrix manipulation routines. A library collects together a number of related and commonly used services. Clients typically need to make use of different libraries in the same program and since libraries are written by different people, the names in different libraries may conflict.

For example, a library for manipulating lists and a library for manipulating dictionaries may both export procedures named insert. Mechanisms are needed for clients to conveniently distinguish between such identically-named services. We have seen that the dot notation helps with this problem at the module level. But consider trying to use two different releases of the same library at the same time. How can you use some of the entities from one release and some from the other? Both C++ and Ada have recent additions to the language to deal with these issues. We will describe these facilities when we discuss specific languages: namespaces of C++ on page 295 and child libraries of Ada on page 302.

**5.3 Language features for programming in the large**

We have so far discussed the concepts of programming in the large with isolated examples from programming languages. In this section we look at some interesting ways that existing programming languages support—or do not support—the programming in the large concepts. All programming languages provide features for decomposing programs into smaller and largely autonomous units.

We refer to such units as *physical* modules; we will use the term *logical* module to denote a module identified at the design stage. A logical module represents an abstraction identified at the design stage by the designer. A logical module may be implemented by one or more physical modules. The closer the relationship between the physical modules and logical modules is, the better the physical program organization reflects the logical design structure.

We will discuss the relevant aspects of each language based on the following points:

• **Module encapsulation**: What is the unit of modularity and encapsulation supported by the language, and how well does it support different programming paradigms?

• **Separation of interface from implementation:** What is the relationship among modules that form a program? What entities may be exported and imported by a module?

• **Program organization and module groupings:** How independently can physical modulesbe implemented and compiled? What are the visibility and access control mechanisms supported by the language?

We will discuss Pascal, C, C++, Ada, and ML. Pascal and C are viewed here as a representative of the class of traditional, minimalist, procedural languages. Our conclusions about them hold, with minor changes, for other members of the class such as FORTRAN. C++ is a representative of class based languages. Ada is a representative of module-based languages, although the 1995 version of the language has enhanced its object-orientation support. ML is reviewed as a representative of functional languages. A few comments on other languages will be given in Section 5.3.6. In general, our discussion here is not about programming paradigms. Object oriented and functional programming support will be covered in, respectively, Chapters 6 and 7.

**5.3.1 Pascal**

In this section we provide an assessment of Pascal’s features for programming in the large. Since many dialects and extensions of Pascal exist, here we consider the original version of the language. Most of the inconveniences discussed here have been eliminated by the enhancements provided by modern implementations.

The only features provided by Pascal for decomposing a program into modules are procedures and functions, which can be used to implement procedural abstractions. The language thus only supports procedural programming. Some later versions of the language have modified the original version of Pascal extensively by adding object-oriented programming features.

A Pascal program has the following structure.

**program** program\_name (files);

declarations of constants, types, variables, procedures and functions;

**begin**

statements (no declarations)

**end**.

A program consists of declarations and operations. The operations are either the built-in ones provided by the language or those declared as functions and procedures. A procedure or function itself may contain the declaration of constants, types, variables, and other procedures and functions. The organization of a Pascal program is thus a tree structure of modules (see static nesting tree in Section 2.6.4 on page 104). The tree structure represents the textual nesting of lower-level modules. Nesting is used to control the scope of names

declared within modules, according to the static binding rule presented in Section 2.6.4.

To assess the structure of Pascal programs, consider the following example. Suppose that the top-down modular design of a module A identifies two modules B and C providing subsidiary procedural abstractions. Similarly, module B invokes two private procedural abstractions provided by modules D and E.

Module C invokes a private procedural abstraction provided by F. Figure 68 shows a nesting structure for a program that satisfies the design constraints. A basic problem with the solution of Figure 68 is that the structure does not enforce the restrictions on procedure invocations found at the design stage.

Actually, the structure allows for the possibility of several other invocations. For example E can invoke D, B, and A; C can invoke B and A, and so on. On the other hand, the structure of Figure 68 imposes some restrictions that might become undesirable. For example, if we discover that module F needs the procedural abstraction provided by module E, the current structure is no longer adequate. Figure 69 shows a rearrangement of the program structure that is compatible with this new requirement. The problem with this new organization is that the structure no longer displays the hierarchical decomposition of abstractions. Module E appears to be a subsidiary abstraction used by A, although the only reason for its placement at that level in the tree is thatboth modules B and F need to refer to it. Similar problems occur for variables, constants and types. The tree structure provides indiscriminate access to variables declared in enclosing modules. In addition, if any two modules M and N need to share a variable, this variable must be declared in a module that statically encloses both M and N and thus the variable becomes accessible toany other modules enclosed by this enclosing module.



Further problems are caused by the textual layout of Pascal programs. The entire program is a single monolithic text. If the program is large, module boundaries are not immediately visible, even if the programmer uses careful conventions for indentation. A module heading can appear well before its body, because of intervening inner module declarations. Consequently, programs can be difficult to read and modify.

The problems with Pascal discussed in this section stem from block structure, and therefore hold for other ALGOL-like languages. Block structure is adequately



for programming in the small because it supports stepwise refinement quite naturally. It is not so valuable for structuring large programs. The program structure resulting from nesting may interfere with the logical structure found during design. This can impair the writability, readability, and modifiability of programs.

Another important question to address is how Pascal modules can be developed independently, and how long-lived and reusable they are. These goals are achieved by applying information hiding at the design stage to obtain a clean definition of module interfaces. In addition, it is desirable to support the separate implementation of modules. It should be possible to compile and certify modules separately. Separately compiled and tested modules should be kept in a library, ready for later reuse. The original Pascal Report does not address these issues, although most Pascal implementation provided their own solutions. Thus, a number of important questions are left unanswered by the original Report, such as

• What program entities can a separate compilation unit export?

• How is a unit interface specified?

• What amount of type checking across unit interfaces is prescribed to occur?

Different implementations have indeed adopted different solutions to thesepoints. As a result, Pascal programs developed on different platforms may beincompatible. For example, some implementations allow outer-level procedures and functions to be compiled independently. Independently compiledUnitsare assembled via a standard linker, which resolves the bindings between the entities imported by each module and the corresponding entities exported by other modules.

 No intermodule checkes are performed, however, to verify that, say, a call to an external procedure is inconsistent with the corresponding procedure declaration. Errors of this kind might remain uncaught, unfortunately. There are modern implementations of Pascal, such as Turbo Pascal, however, which provide safer separate-compilation facilities based on the notion of a module that encapsulates a set of constants, procedures and types.

**5.3.2 C**

C provides functions to decompose a program into procedural abstractions. In addition, it relies on a minimum of language features and a number of conventions to support programming in the large. These conventions are well recognized by C programmers and are even reflected in tools that have been developed to support the language. Indeed, a major portion of the programming in the large support is provided by the file-inclusion commands of the C pre-processor. Thus, even though the compiler does not provide any explicit support or checking for inter-module interaction, the combination of conventions and the pre-processor has proven in practice to be an adequate and popular way to support programming in the large.

The C unit of physical modularity is a file. A logical module is implemented in C by two physical modules (files) which we may roughly call the module’s interface and its implementation. The interface, called a “header” or an “include” file, declares all symbols exported by the module and thus available to the clients of the module. The header file contains the information necessary to satisfy the type system when the client modules are compiled. The implementation file of the module contains the private part of the module and implements the exported services.

A client module needing to use the functionality of another module “includes” the header file of the provider module. A header file may declare constants, type definitions, variables, and functions. Only the prototype of the function—its signature—is given by the declaration; the function definition appears in the implementation file. Functions may not be nested. Any names defined in a file are known throughout that file and may also be known outside of that file.

The header files are used to resolve inter-module references at compile-time.

At link-time, all implementation files are searched to resolve inter-module (i.e. inter-file) references. The header file is usually named with a .h extension and the implementation file is named with a .c extension. These conventions have largely overcome the lack of any explicit support for program organization.

Figure 70 shows the header and implementation files for a module providing a stack data structure. language provides no encapsulation facilities. For example, the main program in Figure 70 has complete access to the internal structure of the stacks s1 and s2. In fact, this property is used by the main program to initialize the stacks s1 and s2 to set their stack pointers (top) to 0.

There are ways to implement this program to reduce this interference between client and server but all depend on the care taken by the programmer. There is no control over what is exported: by default, all entities in a file are exported. Files may be compiled separately and inter-file references are resolved at link time with no type-checking. A file may be compiled as long as all the files it includes are available.

/\* file stack.h \*/

/\*declarations exported to clients\*/

typedef struct stack {

int elments[100]; /\* stack of 100 ints \*/

int top; /\*number of elements\*/

};

extern void push(stack, int);

extern int pop(stack);

/\* end of file stack.h \*/

/\*\*\*\*\*----------------------end of file \*\*\*\*/

/\*file stack.c \*/

/\*implementation of stack operations\*/

#include''stack.h''

void push(stack s, int i) {

s.elements[s.top++] = i;

};

int pop (stack s) {

return --s.top;

};

/\*\*\*\*\*----------------------end of file \*\*\*\*/

/\*file main.c \*/

/\*A client of stack\*/

#include ''stack.h''

void main(){

stack s1, s2; /\*declare two stacks \*/

s1.top = 0; s2.top = 0; /\* initialize them \*/

int i;

push (s1, 5); /\* push something on first stack \*/

push (s2, 6); /\* push something on second stack\*/

...

i = pop(s1); /\* pop first stack \*/

...

}

**FIGURE 70.**Separate files implementing and using a stack in C

Any names defined in the outer level of a file are implicitly known globally. These include the names of all the functions defined in the file and any other entities defined outside of those functions. There are two ways to control such indiscriminate dispersion of names.

• A module wanting to use an entity that is defined externally must declare such entities as being externally defined.

• A module wanting to limit the scope of one of its defined entities to be local to itself only may declare such an entity to be static.

The following two lines import the integer variable maximum\_length and hides the integer variable local\_size from other modules. extern int maximum\_length;

static int local\_size;

There are no explicit import/export facilities. All control over module independence relies on convention and implementer competence.

**5.3.3 C++**

C++ is based on C and it shares C’s reliance on conventions and unit of physical modularity as the file. As C, C++ provides functions as a decomposition construct to implement abstract operations. Nevertheless, C++’s most important enhancements to C are in the area of programming in the large. In particular, the class construct of C++ provides a unit of logical modularity that supports the implementation of information hiding modules and abstract data types. Combined with templates, classes may be used to implement generic abstract data types. The class provides encapsulation and control over interfaces.

In this chapter, we review the use of classes as modules. We will examine the use of classes to support object-oriented programming in Chapter 6.

#include ...various files...

global declarations

function definitions

void main (parameters)

{ ...one main function needed

...in a program

}

**FIGURE 72.**Structure of a C module

**5.3.3.1 Encapsulation in C++**

The unit of logical modularity in C++ is the class. A class serves several purposes including:

• A class defines a new (user-defined) data type.

• A class defines an encapsulated unit.

Entities defined by a class are either *public*—exported to clients—or *private*—hidden from clients. There are also *protected* variables which will be discussed in the next chapter.

Since a class defines a user-defined type, to use the services offered by a class, the client must create an instance of the class, called an *object*, and use that object. C++ supports the style of programming in which programmers write applications by extending the types of the language with user-defined types. Class derivation is a mechanism that supports the definition of new types based on existing types. We will examine this in more detail in the next chapter.

Classes may be nested. But as we saw in the case of Pascal, nesting may be used only for programming in the small and is of limited utility for programming in the large.

Both classes and functions may be generic, supporting a generic programming style. We will discuss generic units in Section 5.4.

**5.3.3.2 Program organization**

Classes define the abstractions from which the program is to be composed. The main program or a client creates instances of the classes and calls on them to perform the desired task. We saw the definition of a C++ template module implementing a generic abstract data type stack in Chapter 3. Figure 73 shows a class implementing a stack of integers1. The implementation separates the interface and the implementation in different files.

 /\* file stack.H \*/

/\*declarations exported to clients\*/

class stack {

public:

stack();

void push(int);

int push pop();

private:

int elments[100]; /\* stack represented as array \*/

int top = 0; /\*number of elements\*/

};

// the implementation follows and may be in a separate file

void stack::push( int i) {

elements[top++] = i;

};

int stack::pop (int i) {

return elements[--top];

};

/\*end of stack.H\*/

/\*main.c \*/

/\*A client of stack\*/

#include “stack.h”

main(){

stack s1, s2; /\*declare two stacks \*/

int i;

s1.push (5); /\* push something on first stack \*/

s2.push (6); /\* push something on second stack\*/

...

i = s1.pop(); /\* pop first stack \*/

... ...

}

**FIGURE 73.**Stack class in C++

C++ supports the development of independent modules (but does not enforce it):

1. A class’s interface and implementation may be separated and even compiled separately from each other. The implementation must include the interface definition and therefore must be compiled after the interface file exists.
2. Client modules may be compiled with access to only the interface modules of the service providers and not their implementation modules.
3. Any names defined in a class are local to the class unless explicitly declared to be public. Even so, client modules must use the class name to gain access to the names internal to the class.

**5.3.3.3 Grouping of units**

C++ has several mechanisms for relating classes to each other. First, classes may be nested. As we have said before, this is a programming in the small feature. Two other mechanisms, “friend” functions and namespaces, are discussed next.

**Friend functions:**

A class in C++ defines a user-defined type. As a result, the operations it defines as public are operations on objects of that type. Some operations do not naturally belong to one object or another. For example, if we define a class for complex numbers, it may have a data part that stores the real and imaginary parts of the number, along with exported operations that let clients create and manipulate objects of type complex. But what about an addition operation that takes two complex objects to add together? Which of the two complex objects is the operation a member of? Figure 74 shows the definition of a complex number class. The class defines the type complex which is internally composed of two doubles, representing the real and imaginary parts of a complex number. These are hidden from clients. The class exports a method of constructing a complex number out of two doubles. Thus, the following declaration creates two complex numbers:

complex x(1.0, 2.0), y(2.5, 3.5);

The other declarations state that the operator functions to be defined later (+, - , \*, and /) are friends of the class complex and thus may access the private parts of the class. They are not member functions of the class and they are notexported by the class. 

Defining these operators as friend functions allows the clients to naturally use these functions as binary operations such as:

complex c = x + y; If the operation + was made a member of the class, the notation for clients would be quite awkward. For example, we might have had to write something like:

c.add(x) in order to add the complex x to complex c. The requirement for friend functions is a direct consequence of C++’s use of classes as user-defined types.

In a language like Ada where the package is used not to define types but to group related entities, we would naturally group together type definitions for complex and its related functions in the same package. The functions automatically gain access to the private parts of the package because they are part of the package. In both cases, any changes to the representation of the data may require changes to the functions,whether they are part of a package or they are friend functions.

**Namespaces:**

In C and in C++, the unit of global naming is a file. Any names defined at the outer level of a file are known globally by default. For example, the names of all classes defined in a library are known to any client that includes that file. What if two libraries provide two classes with the samename? How can a client use both of those classes? How can a library provideradd a new service to its library and be sure that the new name of the service does not conflict with any existing uses of the clients? Since names are created

by independent people, a single global name space is a serious problem in the development of large programs. The solution of C++ is to partition the global name space into a smaller groups; each group is called a *namespace*.

The names defined in a namespace are independent from those in any other namespace and may be referenced by supplying the name of the namespace. This mechanisms enables library providers to provide their libraries in their own namespaces with a guarantee of independence from other library providers.

Of course, it is necessary for the names of the namespaces themselves to be unique.

For example, consider the XYZ Corp. that provides a library of classes for manipulating turbine engines. It might provide its library in a namespace

XYZCorp:

namespace XYZCorp {

typedef turbodiesel ...;

void start (turbodiesel);

//...other definitions

}

A client wanting to use the turbodiesel definition has several options. One is to directly name the definition. The :: operator is used to qualify the namespace in which to look for the desired object.

XYZCorp::turbodiesel t;

Another option is to first create a synonym for the name so that the namespace name does not need to be repeated:

using XYZCorp::turbodiesel; //creates a synonym turbodiesel

//...

turbodiesel t;

XYZCorp::start (t);

The final option is for a client that wants to import all the definitions from a namespace. The namespace may be opened by importing it:

using namespace XYZCorp; //this “opens” the namespace completely

turbodiesel t;

start (t);

The namespace mechanism is intended to help library providers become independent of other library providers, enable them to update their libraries without danger of interfering with client code, and even provide new releases of libraries that co-exist with older releases (each release lives in a different namespace).

The :: operator is used generally to deal with scope resolution. For example,

::x refers to x in the global environment. X::x refers to x in the scope X which may be a namespace or a class whose name, X, known in the current referencing environment.

**5.3.4 Ada**

Ada was designed specifically to support programming in the large. It has elaborate facilities for the support of modules, encapsulation, and interfaces. Rather than relying on convention as in C and C++, Ada makes an explicit distinction between specification and implementation of a module. A file may be compiled if the specifications of the modules it uses are available. Thus, Ada naturally supports a software development process in which module specifications are developed first and implementation of those modules may proceed independently. Ada also requires the existence of a compile-time library in which module specifications are compiled. A module may be compiled if all the module specifications it needs are already in the library.This library supports the checking of inter-module references at compile time.

**5.3.4.1 Encapsulation in Ada**

The package is Ada’s unit of modularity. An Ada module encapsulates a group of entities and thus supports module-based programming. We have already seen that the language’s explicit distinction between module specification and module body forces the programmer to separate what is expected by the module from what is hidden within the module. Additionally, Ada supports concurrent modules or tasks.

In addition to the conceptual modularity at the package level, Ada supports the separate compilation of procedures and functions as well as packages. We will see an example of this in the next section. All units in Ada may also be generic. We will discuss generic units in Section

.

**5.3.4.2 Program organization**

An Ada program is a linear collection of modules that can be either subprograms or packages. These modules are called units. One particular unit that implements a subprogram is the main program in the usual sense. Module declarations may be nested. Consequently, a unit can be organized as a tree structure of modules. Any abuse of nesting within a unit causes the same problems discussed for Pascal. These problems can be mitigated by the use of the subunit facility offered by the language. This facility permits the body of a

module embedded in the declarative part of a unit (or subunit) to be written separately from the enclosing unit (or subunit). Instead of the entire module, only a *stub* need appear in the declarative part of the enclosing unit.

The following example illustrates the concept of the subunit.

procedure X ( ... ) is --unit specification

W: INTEGER;

package Y is --inner unit specification

A: INTEGER;

function B (C: INTEGER) return INTEGER;

end Y;

package body Y is separate; --this is a stub

begin -- uses of package Y and variable W

...

...

...

end X;

------------------------------------next file--------------

separate (X)

package body Y is

procedure Z (...) is separate; --this is a stub

function B (C: INTEGER) return INTEGER is

begin --use procedure Z

...

...

...

end B;

end Y;

------------------------------------next file--------------

separate (X.Y)

procedure Z (...) is

begin

...

end Z;

The prefix **separate** (X) specifies package body Y as a subunit of unit X. Similarly, **separate** (X.Y) specifies procedure Z as a subunit of package Y nested within X. The subunit facility not only can improve the readability of programs, but supports a useful technique in top-down programming.

When writing a program at a certain level of abstraction, we may want to leave some details to be decided at a lower level. Suppose you realize that a certain procedure is required to accomplish a given task. Although calls to that procedure can be immediately useful when you want to test the execution flow, the body of the procedure can be written at a later time. For now, all you need is a *stub*. The subunit facility, however, does not overcome all the problems caused by the tree nesting structure. The textually separate subunit body is still considered to be logically located at the point at which the corresponding stub appears in the enclosing (sub)unit. It is exactly this point that determines the entities visible to the subunit. In the example, both subunits Y and Z can access variable W declared in unit X.

The interface of an Ada unit consists of the **with** statement, which lists the names of units from which entities are imported, and the *unit specification*(enclosed within a **is... end** pair), which lists the entities exported by the unit.

Each logical module discovered at the design stage can be implemented as a unit. If the top-down design was done carefully, logical modules should be relatively simple. Consequently, the nesting within units should be shallow or even nonexistent. Ada does not forbid an abuse of nesting within units. Actually, the entire program could be designed as a single unit with a deeply nested tree structure. It is up to the designer and programmer to achieve a more desirable program structure.

The last program structuring issue is how the interfaces (i.e., import/export relationships) among units are specified in Ada. A unit exports all the entities specified in its specification part. It can import entities from other units if and only if the names of such units are listed in a suitable statement (**with** statement)

that prefixes the unit. For example, the following unit lists unit X (a

subprogram) in its **with** statement. Consequently, it is legal to use X within T’s body.

**with** X;

**package** T **is**

C: INTEGER;

procedure D (...);

**end** T;

**package body** T **is**

...

...

...

**end** T;

Similarly, the following procedure U can legally call procedure T.D and access variable T.C. On the other hand, unit X is not visible by U.

**with** T;

**procedure**U (...) **is**

...

**end** U;

**5.3.4.3 Interface and implementation**

We have already seen in Section that Ada strictly separates the specificationand body of a package. In the previous section, we have seen how the **use** and**with** clauses are used to import services frompackages. These facilities are used also to support separate compilation. Recall that separate compilation, as

opposed to independent compilation, places a partial ordering on compilation units.

The set of units and subunits comprising a program can be compiled in one or more separate compilations. Each compilation translates one or more units and/or subunits. The order of compilation must satisfy the following constraints.

• A unit can be compiled only if all units mentioned in its with statement have been compiled previously.

• A subunit can be compiled only if the enclosing unit has been compiled previously.

In addition, unit specifications can be compiled separately from their bodies. A unit body must be compiled after its specification.

The specification of a unit U mentioned in the **with** statement of a unit W must be compiled before W. On the other hand, U’s body may be compiled either before or after W. These constraints ensure that a unit is submitted for compilation only after the compilation of unit specifications from which it can import entities. The compiler saves in a library file the descriptors of all entities exported by units.

When a unit is submitted for compilation, the compiler uses the library file to perform the same amount of type checking on the unit whether the program is compiled in parts or as a whole. Ada’s choice of a package as an encapsulation mechanism, together with its reliance on separate compilation, and the separation of specification and body creates an interesting issue when a package wants to export a type. This issue leads to the private type feature of Ada.

**The private type:**

In Figure 65, we declared a dictionary module that exports procedures and functions only. When the client declares its intention to use the dictionary package, the dictionary object is allocated. The representation of the object is not known to the client. From the package body, we can see that the entries in the dictionary are actually records that contain three different fields. What if we want to export to the client a type such as dictionary\_entry? This would enable the client to declare variables of type dictionary\_entry. We would like to export the type but not its representation.

From the language design point of view there is a conflict here. The Ada language specifies that a client may be compiled with the knowledge only of the specification of the provider module. But if the provider module is exporting a type and not its representation, the size of the type cannot be determined from the specification. Thus, when the compiler is compiling the client, it cannot determine how much memory to allocate for variables of the exported type. Ada’s solution to this problem is the **private type**. The specification must contain the representation of the type but as a private type.

If a package unit exports an encapsulated private data type, the type’s representation is hidden to the programmer but known to the compiler, thanks to the private clause appearing in the package specification. Consequently, the compiler can generate code to allocate variables for such types declared in other units submitted for compilation prior to the package body (but after its specification). When a unit is modified, it may be necessary to recompile several units. The change may potentially affect its subunits as well as all the

units that name it in their **with** statements. In principle, all potentially affected units must be recompiled.

The separate compilation facility of Ada supports an incremental rather than a parallel development of programs, because units must be developed according to a partial ordering. This is not an arbitrary restriction, but a conscious design decision in support of methodical program development. A unit can be submitted for compilation only after the interfaces of all used units are frozen. Consequently, the programmer is forced to postpone the design of a unit body until these interfaces have been designed. One of the goals of separate compilation is to support production of reusable software. Certified modules can be kept in a library and later combined to form different programs. The Ada solution is deficient on this point for package units exporting encapsulated (private) data types. The visible part (the specification) of such packages must include the type’s operations and a private clause that specifies the type’s internal representation.

This representation is not usable outside the package body; it is there only for supporting separate compilation. Logically, this information belongs in the package body, together with the procedure bodies implementing the type’s operations. Besides being aesthetically unpleasant, this feature has some unfortunate consequences:

• It violates the principle of top-down design. The representation must be determined at the same time as the specification of the data type, and both appear in the same textual unit.

• It limits the power of the language to support libraries of reusable modules, unless special care is taken in the implementation. For example, a module using FIFO queues is compiled and validated with respect to a FIFO queue package providing a specific representation for FIFO queues (e.g., arrays). The module must be recompiled if one

wants to reuse it in a different program in which FIFO queues are implemented by a different data structure, even though the interfaces for manipulating FIFO queues are the same in both cases.

**5.3.4.4 Grouping of units**

Ada has many features for supporting programming in the large. Two clauses, **use** and **with,** are used to import services from other packages. Child library units are used to group packages together in hierarchical organizations. These facilities are defined to enable safe separate compilation. **The with and use clauses.** The **with** clause is used by a client to import from a provider module. For example, if we want to write a module to manipulate telephone numbers and we want to use the dictionary module specified in Figure 65, we prefix the telephone module with a **with** clause: **with** dictionary;

**package** phone\_list **is**

...

--references to dictionary.insert(), etc.

...

302 Structuring the program Chap.5

**end** phone\_list;

Now, inside the phone\_list package, we may refer to the exported entities ofthe dictionary package. These references have to be prefixed by the name of the package from which they are imported. For example, dictionary.insert(...).

To gain direct visibility, and avoid the need to use the dotted name, Ada provides the **use** clause:

**with** dictionary; **use** dictionary;

**package** phone\_list **is**

...

--references to insert(), etc.

...

**end** phone\_list;

**Child libraries.**

The Ada package groups together a set of related entities. Clients may import either selective services from a package or all the services provided by the package by using the **use** clause. The package is inadequate as a structural mechanism for grouping a collection of library modules. Here are some examples of problems that could occur:

• Suppose a client uses two different libraries, encapsulated in packages A and B. Since the client expects to make extensive use of both libraries, it uses the **use** clause to import all the library services. But if libraries A and B export entities with the same name, the client

would encounter name clashes at compile time. Ada provides a renaming facility to get around this problem.

• More serious is the case where there are no name clashes. The client compiles and works properly. But suppose that a new version of library B is released with new functionality.

It happens that one of the new functions introduced in B has a name identical to a name provide by A. The next time that the client code is compiled, compilation errors will show up due to name clashes. These errors would be particularly confusing because the previously working client code appears to not work even though it may not have been changed.

• In the previous case, after the release of the new version of the library B, the client code has to be recompiled even though it does not make use of the new functionality of the library B.

The recompilation is necessary only to satisfy Ada’s rules on the order of compilation. Ada 95 has addressed these problems by introducing the notion of childlibraries which allow packages to be hierarchically organized. The idea is that if new functionality is added to an existing library package, the new functionality may itself be organized as new package that is a child of the original library. The child package can be implemented using the facilities of the parent package. But the clients of the original library are not affected by the introduction of a child package.

The child package makes it possible to add functionality to a package without disturbing the existing clients of the package. In general, a library developer may provide a number of packages organized as a tree. Each package other than the root package has a parent package. An existing library may be extended by adding a child library unit to one of its existing nodes. The parent library unit, nor any clients of the parent need to be recompiled. For example, if the library Root exists, we may add Root.Child without disturbing Root or clients of Root. The Root.Child may be compiled separately. It has visibility to Root and to Root’s siblings.

**package** Root **is**

--specification of Root library

--...

**end** Root;

-----------

**package** Root.Child **is**

--specification of a child library unit

--...

**end** Root.Child;

-----------

**package body** Root.Child **is**

--implementation of Root.Child

--...

**end** Root.Child;

-----------

Each of the above segments may be compiled separately. The clients of Rootneed not be recompiled if they do not use Root.Child.

**5.3.5 ML**

Modularity is not only the province of imperative languages. The notion ofmodule is important in any language that is to be used for programming in the large. For example, ML is a functional programming language with extensive support for modularity and abstraction. In Chapter 7, we will study the basics of functional programming and ML. In Chapter 7, we will see ML’s support

for defining new types and abstract data types, which also help in programming in the large. In this section we give a brief overview of ML’s support for modules.

**5.3.5.1 Encapsulation in ML**

A module is a separately compilable unit. A unit may contain structures, signatures, and functors. Structures are the main building blocks; signatures are used to define interfaces for structures; functors are used to build a new structure out of an existing structure. We will discuss these more in Chapter 7.

Here, we only examine the structure construct as a packaging unit.

The ML *structure* is somewhat like the Ada package, used to group together a set of entities. For example, our dictionary example package of Figure 66

may be written in ML as given in Figure 75. Recall the syntax and case analysis style of programming from Chapter 5. We will describe the details of the functions in Chapter 7. Here, we are only interested in what is exported by the structure, that is, module.

Such a structure definition corresponds to the package body in that it gives the implementation for the entities being defined. It also has the property that all the entities are exported. This structure exports an exception, NotFound, a variable root, and two functions insert and lookup. To use the structure, a client uses the dot notation:

val D = Dictionary.create; (\*create an empty dictionary \*)

val newD = Dictionary.insert (“Mehdi”, 46, D); (\*insert a pair\*)

...

D.lookup(“Mehdi”, D); (\*produces value 46\*)



***5.3.5.2 Interface and implementation***

The signature of a structure definition consists of the signatures and types of all the entities defined in the structure. ML also provides a construct to define a signature independently of any structure. A signature may be viewed as a specification for a module. For example, Figure 76 gives the signature of a module that exports an exception called Not Found and a function called lookup. A signature may be used as a specification for a structure. For example, we may use the signature of Figure 76 to restrict the exported entities of

the structure of Figure 75. The system will do type checking to ensure that the structure provides at least what the signature requires.



We can use the structure and signature we have to create a new module with a restricted interface and use it accordingly:

structure LookupDict: DictLookupSig = Dictionary;

val L = LookupDict.create; (\* not allowed, must be done by someone else using a different

interface \*)

lookupDict.lookup(“Mehdi”, L);

lookupDict.insert(“Carlo”, 50, L); --error, insert not available

We can see that the ability to define signatures means that we can provide different interfaces to the same implementation, something not possible in Ada or C++. We can also provide different implementations to meet the same interface.

ML also supports the concept of *generic* modules or structures. The signature facility may be combined with generic structures to instantiate a structure for particular types. For example, the dictionaries that we have defined so far, both in Ada and in ML have been specific to <string, integer> pairs. In ML, we can remove the occurrences of the terms string and int from Figure 75 and have a generic dictionary. We will see this in Section 5.4.3.

**5.3.6 Abstract data types, classes, and modules**

We have discussed an abstract data type as a program modularization concept. Languages that provide a class construct, such as C++, support the implementation of abstract data types directly. For example, Figure 73 shows a class that implements an abstract data type stack and a client that declares instances of the stack and uses them. The name of the class is used as the type name to instantiate the objects necessary. Operations are performed directly on the instantiated objects, e.g. s1.push(...). Module-based languages such as Ada or Modula-2, however, do not support objects directly and are operation-oriented. We use a module to implements an abstract data type by packaging the type and its operations together. But the client creates instances of the abstract data type and passes them to operations as necessary, rather than calling the operations associated with theobject. That is, rather than calling s1.push(...) the client calls push(s1, ...). In

object-based languages, the object is an implicit first parameter automatically passed to the operation.

In a module-based language, the need for a construct such as friend functions does not appear: we simply put all the related functions and types in the same module and they gain visibility to each other. In an object-based language, the requirement to package a single type and its operations together makes it difficult to deal with operations that do not belong clearly to a single type. We generally have to make such operations global. Java resolves this dichotomy by supporting both modules and classes. Classes defined together in the same

module have visibility to one another’s internal structure.

A final comparison of Ada and C++ styles concerns the export of types. In both languages, a client may instantiate a variable of a type defined by another module (or class). Given a declaration of the form s: T in a client, an important question for the compiler is how much storage to allocate for the instance of s. Even though logically this information is part of the implementation of the module that implements the type, not part of its specification, the compiler needs the information at the time it compiles the client. This is the reason, as we have seen, that Ada requires the private clause in the specification part of a package. The information in the private part is there only for the compiler. C++ also requires the same information in the private part of a class definition.

Requiring the data representation in the specification of a module means that if the representation changes, the clients will have to be recompiled. This is a serious cost in large system development. To address this problem, Modula-2 introduced the notion of *opaque* export which allows types to be exported without the details of their representation.

**5.4 Generic units**

In this chapter, we have considered the issue of modularity as a support for developing large programs. One important approach to developing large programs is to build them from existing modules. Traditional software libraries offer examples of how such existing modules may be packaged and used. One of the criteria for evaluating the suitability of a language for programming in the large is whether it provides language mechanisms that enable the construction of independent components, the packaging together of related components, the use and combination of multiple libraries by clients, etc. We have seen the namespaces of C++ and the libraries and child libraries of Ada 95 as language mechanisms explicitly developed for the support of such packaging of related and independent software components. In this section, we concentrate on genericity as a mechanism for building individual modules that are general and thus usable in many contexts by many clients.

**5.4.1 Generic data structures**

Let us first consider the development of libraries of standard data structures, for example, stacks and queues. What should be the types of elements stored in these structures? Early typed languages such as Pascal and C require the designer to define one structure for each data type to be supported. This is an unsatisfactory solution for two reasons: one is that the solution only works for only the types the library designer knows about and not for any types to be defined by the user of the library; the second is that the solution forces the

library designer towards code duplication. C++ templates and Ada generics allow us to avoid such code duplication and define a template data structure that is independent of the type of the element to be stored in the structure. For example, we can define a generic *pair* data structure in C++:

template <class T1, class T2>

class pair {

public:

T1 first;

T2 second;

pair (T1 x, T2 y) : first(x), second(y) { }

};

The template parameters T1 and T2 stand for any type. We may “instantiate”a particular pair by supplying concrete types for T1 and T2. For example, we may create a pair of integers or a string, integer pair or a pair of employees:

pair<int, int> intint(2, 1456);

pair<string, int> stringint(“Mehdi”, 46);

pair<employee\_t, employee\_t> (jack, jill); /\*pair of user-defined type employee\_t\*/

We may refer to pair as a parameterized or generic type. The template of C++ allows us to define such a parameterized type which may later be used to create concrete types such as pair<int, int>. C++’s template facility is particularly general because it uses classes as parameters and classes represent types uniformly:

we may instantiate a template with either user-defined or primitive

types. Eiffel supports a similar scheme for generic classes, with which, for example, we can define a class stack [T] and then instantiate an intstack from

stack[integer]. In Chapter 3 we saw examples of generic stacks both in C++ and

Eiffel.

**5.4.2 Generic algorithms**

Templates may also be used to define generic algorithms. For example, in Chapter 2, we saw the following generic function swap which interchanges the values of its two parameters:

template <class T>

void swap(T& a, T& b)

{

T temp = a;

a = b;

b = temp;

}

This function may be used for any two parameters of the same type that support the “=” operation. Therefore, we can use it to swap integers, reals, and even user-defined types such as pairs. This is quite a useful facility because it gives us the possibility to write higher-level generic functions such as sort if they only use generic functions. The ability to write such generic functions is helped in C++ by the fact that generic functions do not have to be instantiated to be used.

To use a template data structure, C++ requires explicit instantiation of the structure, as we saw, for example, in pair<int,int>. For functions, on the other hand, explicit instantiation is not necessary. The compiler will infer the instance required and generate it automatically. For example, the following program fragment is valid:

int i, j;

char x, y;

pair<int,string> p1, p2;

...

swap(i, j); //swap integers

swap(x, y); //swap strings

swap(p1, p2); //swap pairs

the compiler will generate three different swap functions, for integers, strings and pairs of (int, string). To generate an appropriate function, the compiler checks at generation time that the parameters meet the expected requirements. Examination of the body of swap shows that the parameters passed must support assignment, that is, to be able to be passed and to be able to be assigned. (Exercise 22 asks you to explain why pairs meet this requirement.)

The implicit parameter requirements in C++ are made explicit in Ada generic functions. The same swap function is defined in Ada as:

**generic**

**type** T **is private**;

**procedure** swap (x, y: T) **is**

**begin**

temp: T = x;

x = y;

y = temp;

**end** swap;

The generic is explicitly stated to be based on a type T which is **private**. The private indication means that the type supports assignment and equality. In general, if other operations are required of the type, they have to be stated. For example, a generic max function will require its operands to support an

order operations such as “>”:

**generic**

**type** T **is private**;

**with function** “<“ (x, y: T) **return** BOOLEAN **is** <>;

**function** max (x, y: T) **return** BOOLEAN **is**

**begin**

**if** x<y

**then return** x;

**else return** y;

**end if**;

**end** max;

To use the function, we have to first instantiate an instance of it:

**function** int\_max **is new** max (INTEGER);

The type parameters passed at instantiation time are checked to ensure that they support the required operations. After instantiation, we have a new function that we may call:

m := int\_max (3, 6);

The Ada view is that different functions are generated and used while the C++ view is that there is just one function max which is generic. It is the compiler’s job to generate as many instances as it needs to satisfy all the calls to the function. The C++ approach is more flexible and is more supportive of generic programming because generic functions are not treated any differently from nongeneric functions: you simply call them. Ada treats generic functions as a special type of function that you must instantiate before you can call.

We will examine ML’s generic functions (called polymorphic) in Chapter 7. In summary, generic routines allow us to parameterize algorithms and achieve a higher level of generality by capturing an algorithm in a type-independent way.

**5.4.3 Generic modules**

Collections of data and algorithms may also be packaged together and collectively made to depend on some generic type parameter. Both C++ classes and Ada packages may be defined as generic in the types they use. We saw a generic stack class in Chapter 3.

The ML support for generic modules is particularly interesting because of the separation of structures and signatures. Recall the ML dictionary module in Section 5.3.5. The signature definition of Figure 75 can be defined in a generic way by not making any mention of specific types such as int and string. We have defined such a generic structure in Figure 77. The signature of this module is independent of specific types. Can we apply the signature of Figure 76 to this structure? That signature definition indeed matches this structure because the structure is more general than the signature requires. By

applying the signature, we are restricting the view of the structure. Applying a signature to a polymorphic structure is similar to package instantiation in Ada.



**5.4.4 Higher levels of genericity**

We have seen that we may define a generic algorithm that works on any type of object passed to it. For example, the max algorithm may be applied to any ordered type. This facility allows us to write one algorithm for n different data types rather than n different algorithms. It leads to great savings for writers of libraries. But consider a higher level of generality. Suppose we want to write an algorithm that works on different types of data *structures*, not just different data types. For example, we may want to write one algorithm to do a linearsearch in any “linear” data structure. Of course, we have to capture the notion

of linearity somehow but intuitively, we want to be able to find an element in a collection regardless of whether the collection is implemented as an array, a list. The goal of the generic programming paradigm is to develop exactly these kinds of units. In Chapter 3, we saw one kind of iterator for stepping through a collection. Here we will examine a different kind of iterator. A high level of genericity is usually associated with functional languages and

we will see it in the context of ML in Chapter 7. There are no particular language facilities in Ada or C++ for this kind of programming. However, the flexibility of C++ templates, combined with overloading of operators supports a high degree of generic programming. For example, consider the following function find:

template<class Iter, class T>

Iter find ( Iter f, Iter l, T x)

{

while (\*f != last && \*f != x)

++f;

return f;

}

We might think of this function as accepting two pointers into a sequence of elements. It sequences through the elements by using the ++ on the first pointer until either the value x is found or the sequence is exhausted. So, the following code fragment looks through the first half of an integer array:

int a[100];

int x;

int \*r;

...

r= find(x, &a[0], &a[50]);

if (r == &a[5])

// not found

...

But we might imagine writing a list object that also provides an Iter type object which supports ++, \*, ==, and != operations with the same semantics as those of pointers into arrays. More importantly, any time a library writer provides a new linear structure, he can also provide it with such iterators. In this way, any generic operations will be immediately usable with the library’s new data structures. What we are doing is to treat operations such as !=, \*, and ++ as generic operations and writing a higher level operation find in terms of them. This style of generic programming is possible in C++ and likely will be the way standard libraries are provided.

The advantages of such an approach for programming in the large is the reduction of the amount of code that needs to be written because one generic unit may be customized automatically depending on the context of its use. It is a form of modularity in which we modularize based on common properties and specific properties. Object-oriented programming is another approach to achieving this same kind of modularity. That will be the subject of the next chapter.