9 Programming Languages

Objectives

After studying this chapter, the student should be able to:

- Describe the evolution of programming languages from machine language to high-level languages.
- Understand how a program in a high-level language is translated into machine language.
- Distinguish between four computer language paradigms.
- Understand the procedural paradigm and the interaction between a program unit and data items in the paradigm.
- Understand the object-oriented paradigm and the interaction between a program unit and objects in this paradigm.
- Define functional paradigm and understand its applications.
- Define a declaration paradigm and understand its applications.
- Define common concepts in procedural and object-oriented languages.

9-1 EVOLUTION

To write a program for a computer, we must use a computer language. A computer language is a set of predefined words that are combined into a program according to predefined rules (***syntax***). Over the years, computer languages have evolved from ***machine language*** to ***high-level languages***.

Machine languages

In the earliest days of computers, the only programming languages available were **machine languages**. Each computer had its own machine language, which was made of streams of 0s and 1s. In Chapter 5 we showed that in a primitive hypothetical computer, we need to use eleven lines of code to read two integers, add them and print the result. These lines of code, when written in machine language, make eleven lines of binary code, each of 16 bits, as shown in Table 9.1.

The only language understood by a computer is machine language.

Table 9.1 Code in machine language to add two integers

Hexadecimal	*Code in machine language*			
$(1FEF)_{16}$	0001	1111	1110	1111
$(240F)_{16}$	0010	0100	0000	1111
$(1FEF)_{16}$	0001	1111	1110	1111
$(241F)_{16}$	0010	0100	0001	1111
$(1040)_{16}$	0001	0000	0100	0000
$(1141)_{16}$	0001	0001	0100	0001
$(3201)_{16}$	0011	0010	0000	0001
$(2422)_{16}$	0010	0100	0010	0010
$(1F42)_{16}$	0001	1111	0100	0010
$(2FFF)_{16}$	0010	1111	1111	1111
$(0000)_{16}$	0000	0000	0000	0000

Assembly languages

The next evolution in programming came with the idea of replacing binary code for instruction and addresses with symbols or mnemonics. Because they used symbols, these languages were first known as symbolic languages. The set of these mnemonic languages were later referred to as assembly languages. The assembly language for our hypothetical computer to replace the machine language in Table 9.2 is shown in Program 9.1.

The only language understood by a computer is machine language.

Table 9.2 Code in assembly language to add two integers

Code in assembly language				*Description*
LOAD	**RF**	**Keyboard**		Load from keyboard controller to register F
STORE	**Number1**	**RF**		Store register F into Number1
LOAD	**RF**	**Keyboard**		Load from keyboard controller to register F
STORE	**Number2**	**RF**		Store register F into Number2
LOAD	**R0**	**Number1**		Load Number1 into register 0
LOAD	**R1**	**Number2**		Load Number2 into register 1
ADDI	**R2**	**R0**	**R1**	Add registers 0 and 1 with result in register 2
STORE	**Result**	**R2**		Store register 2 into Result
LOAD	**RF**	**Result**		Load Result into register F
STORE	**Monitor**	**RF**		Store register F into monitor controller
HALT				Stop

High-level languages

Although assembly languages greatly improved programming efficiency, they still required programmers to concentrate on the hardware they were using. Working with symbolic languages was also very tedious, because each machine instruction had to be individually coded. The desire to improve programmer efficiency and to change the focus from the computer to the problem being solved led to the development of high-level languages.

Over the years, various languages, most notably BASIC, COBOL, Pascal, Ada, C, C++ and Java, were developed. Program 9.1 shows the code for adding two integers as it would appear in the C++ language.

Program 9.1 Addition program in C++

```
/*      This program reads two integers from keyboard and prints their sum.
        Written by:
        Date:
*/
#include <iostream.h>
using namespace std;
int main (void)
{
        // Local Declarations
        int number1;
        int number2;
        int result;
        // Statements
        cin >> number1;
        cin >> number2;
        result = number1 + number2;
        cout << result;
        return 0;
} // main
```

9-2 TRANSLATION

Programs today are normally written in one of the high-level languages. To run the program on a computer, the program needs to be translated into the machine language of the computer on which it will run. The program in a high-level language is called the source program. The translated program in machine language is called the object program. Two methods are used for translation: **compilation** and **interpretation**.

Compilation

A compiler normally translates the whole **source program** into the **object program**.

Interpretation

Some computer languages use an interpreter to translate the source program into the object program. Interpretation refers to the process of translating each line of the source program into the corresponding line of the object program and executing the line. However, we need to be aware of two trends in interpretation: that used by some languages before Java and the interpretation used by Java.

Translation process

Compilation and interpretation differ in that the first translates the whole source code before executing it, while the second translates and executes the source code a line at a time. Both methods, however, follow the same translation process shown in Figure 9.1.

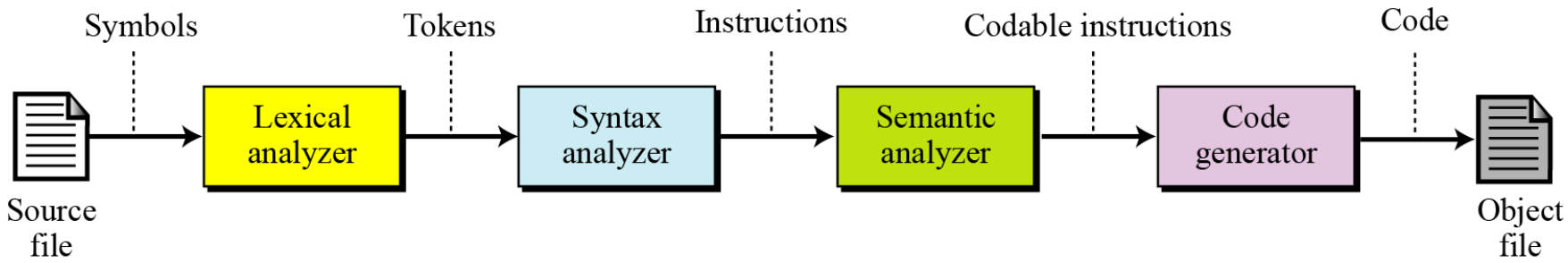

Figure 9.1 **Source code translation process**

9-3 PROGRAMMING PARADIGMS

Today, computer languages are categorized according to the approach they use to solve a problem. A **paradigm**, therefore, is a way in which a computer language looks at the problem to be solved. We divide computer languages into four paradigms: *procedural*, *object-oriented*, *functional* and *declarative*. Figure 9.2 summarizes these.

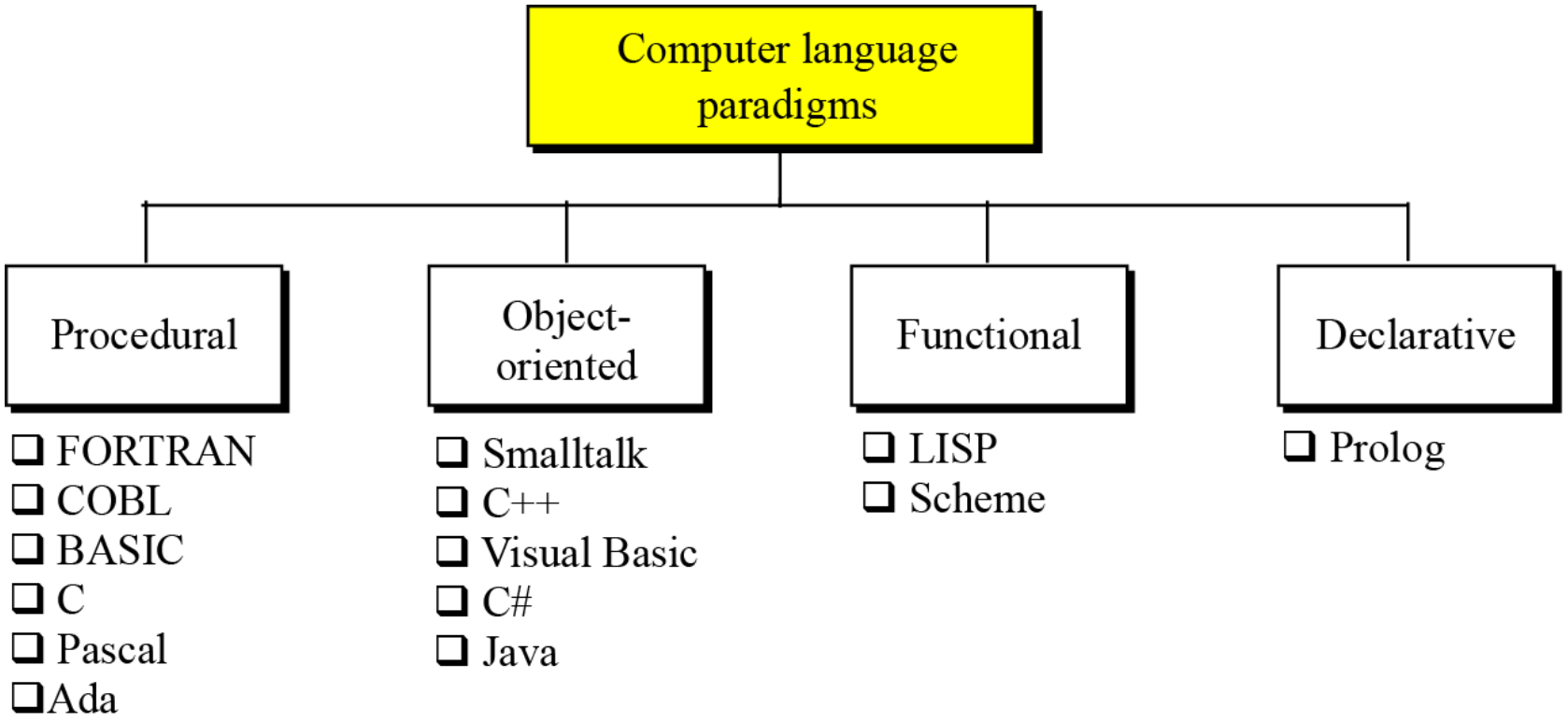

Figure 9.2 **Categories of programming languages**

The procedural paradigm

In the procedural paradigm (or imperative paradigm) we can think of a program as an active agent that manipulates passive objects. We encounter many passive objects in our daily life: a stone, a book, a lamp, and so on. A passive object cannot initiate an action by itself, but it can receive actions from active agents.

A program in a procedural paradigm is an active agent that uses passive objects that we refer to as data or data items. To manipulate a piece of data, the active agent (program) issues an action, referred to as a **procedure**. For example, think of a program that prints the contents of a file. The file is a passive object. To print the file, the program uses a procedure, which we call print.

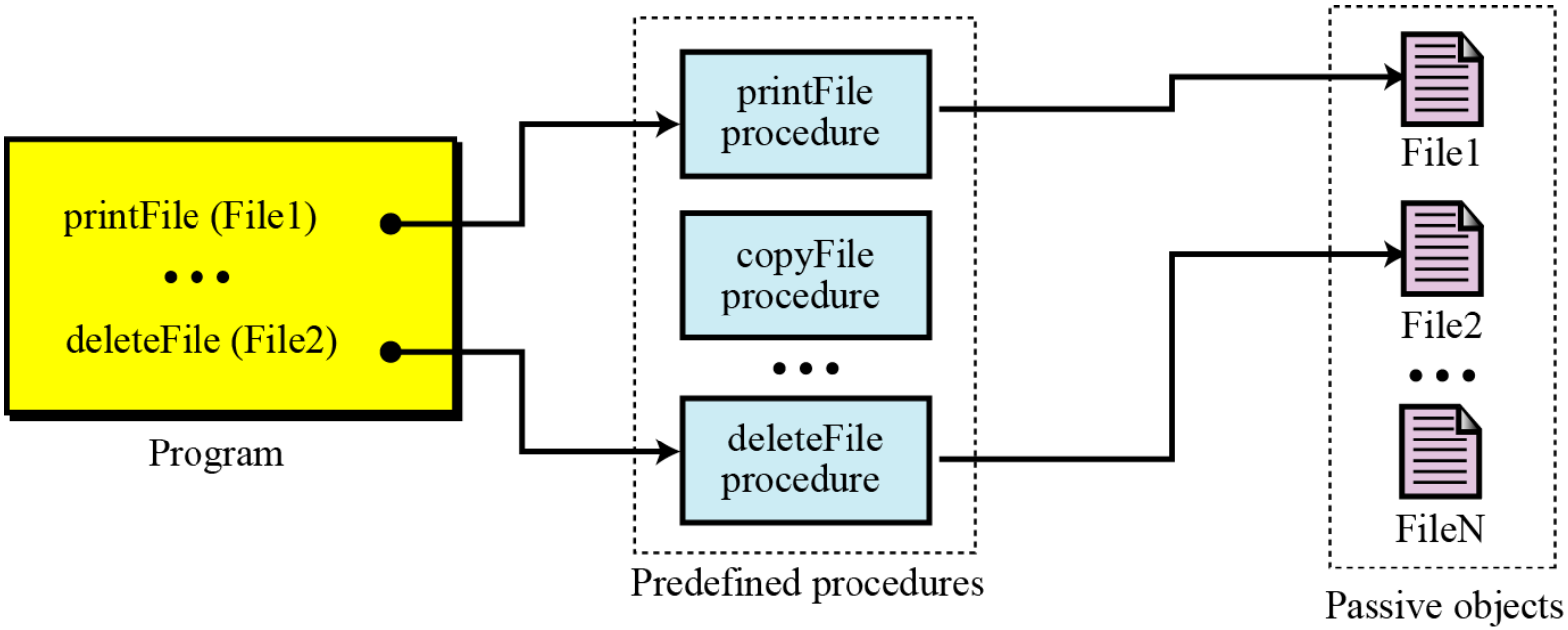

Figure 9.3 The concept of the procedural paradigm

A program in this paradigm is made up of three parts: *a part for object creation*, *a set of procedure calls* and *a set of code for each procedure*. Some procedures have already been defined in the language itself. By combining this code, the programmer can create new procedures.

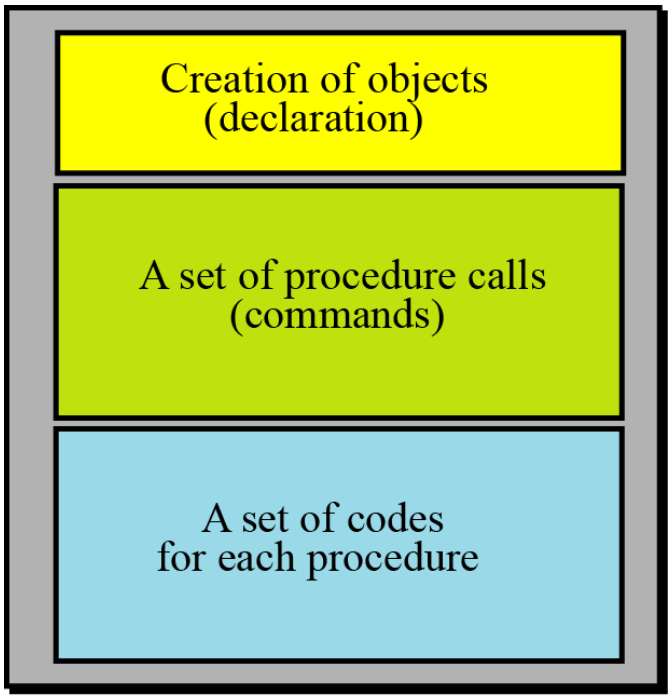

Figure 9.4 **The components of a procedural program**

Some procedural languages

- FORTRAN (FORmula TRANslation)
- COBOL (COmmon Business-Oriented Language)
- Pascal
- C
- Ada

The object-oriented paradigm

The object-oriented paradigm deals with active objects instead of passive objects. We encounter many active objects in our daily life: a vehicle, an automatic door, a dishwasher and so on. The action to be performed on these objects are included in the object: the objects need only to receive the appropriate stimulus from outside to perform one of the actions.

A file in an object-oriented paradigm can be packed with all the procedures—called methods in the object-oriented paradigm—to be performed by the file: printing, copying, deleting and so on. The program in this paradigm just sends the corresponding request to the object.

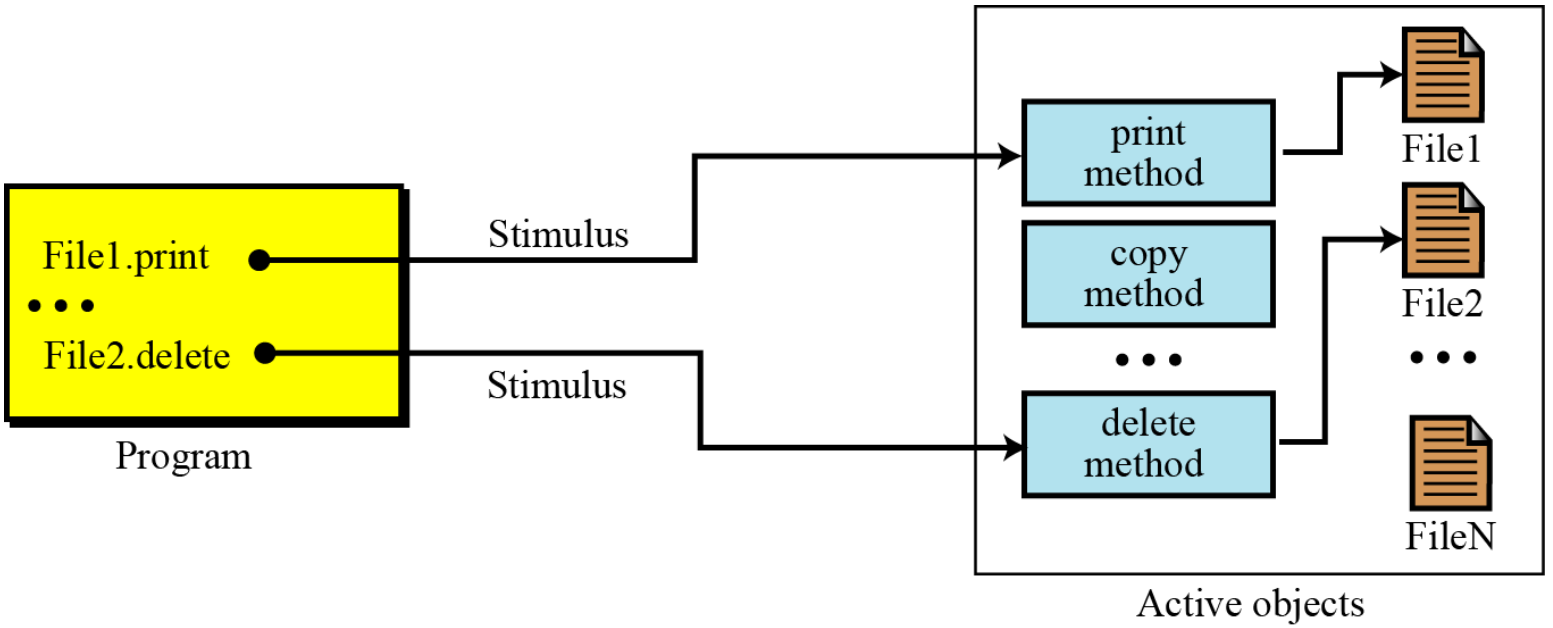

Figure 9.5 The concept of an object-oriented paradigm

Classes

As Figure 9.5 shows, objects of the same type (files, for example) need a set of methods that show how an object of this type reacts to stimuli from outside the object's "territories". To create these methods, a unit called a **class** is used (see Appendix F).

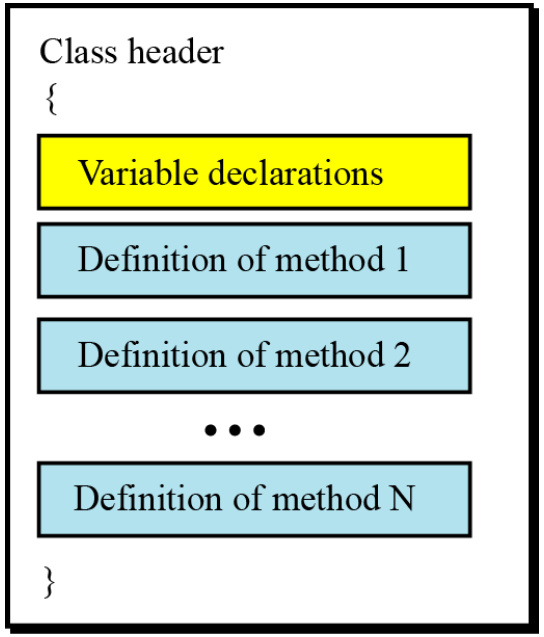

Figure 9.6 **The concept of an object-oriented paradigm**

Methods

In general, the format of methods are very similar to the functions used in some procedural languages. Each method has its header, its local variables and its statement. This means that most of the features we discussed for procedural languages are also applied to methods written for an object-oriented program. In other words, we can claim that object-oriented languages are actually an extension of procedural languages with some new ideas and some new features. The C++ language, for example, is an object-oriented extension of the C language.

Inheritance

In the object-oriented paradigm, as in nature, an object can inherit from another object. This concept is called inheritance. When a general class is defined, we can define a more specific class that inherits some of the characteristics of the general class, but also has some new characteristics. For example, when an object of the type GeometricalShapes is defined, we can define a class called Rectangles. Rectangles are geometrical shapes with additional characteristics.

Polymorphism

Polymorphism means “many forms”. Polymorphism in the object-oriented paradigm means that we can define several operations with the same name that can do different things in related classes. For example, assume that we define two classes, Rectangles and Circles, both inherited from the class GeometricalShapes. We define two operations both named area, one in Rectangles and one in Circles, that calculate the area of a rectangle or a circle. The two operations have the same name

Some object-oriented languages

- C++
- Java

The functional paradigm

In the functional paradigm a program is considered a mathematical function. In this context, a function is a black box that maps a list of inputs to a list of outputs.

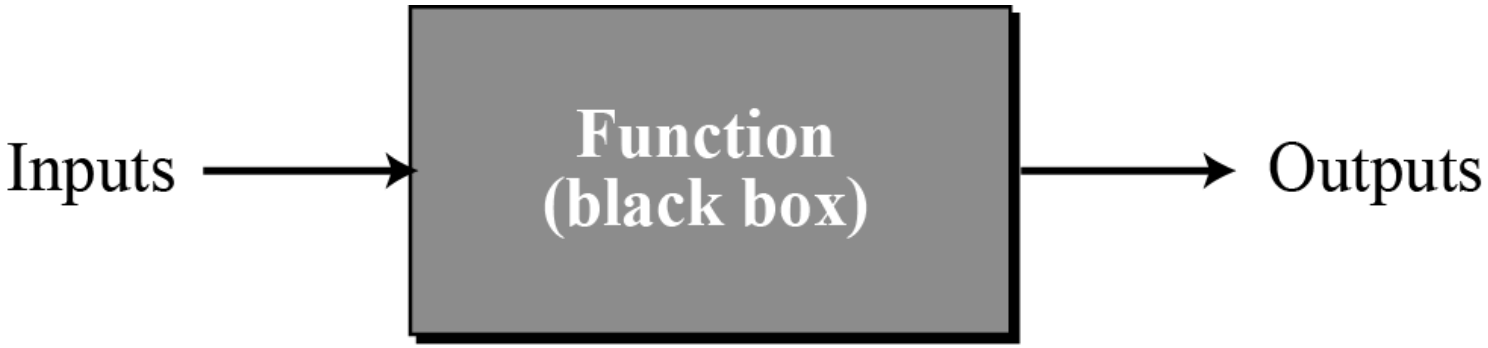

Figure 9.7 **A function in a functional language**

For example, we can define a primitive function called first that extracts the first element of a list. It may also have a function called rest that extracts all the elements except the first. A program can define a function that extracts the third element of a list by combining these two functions as shown in Figure 9.8.

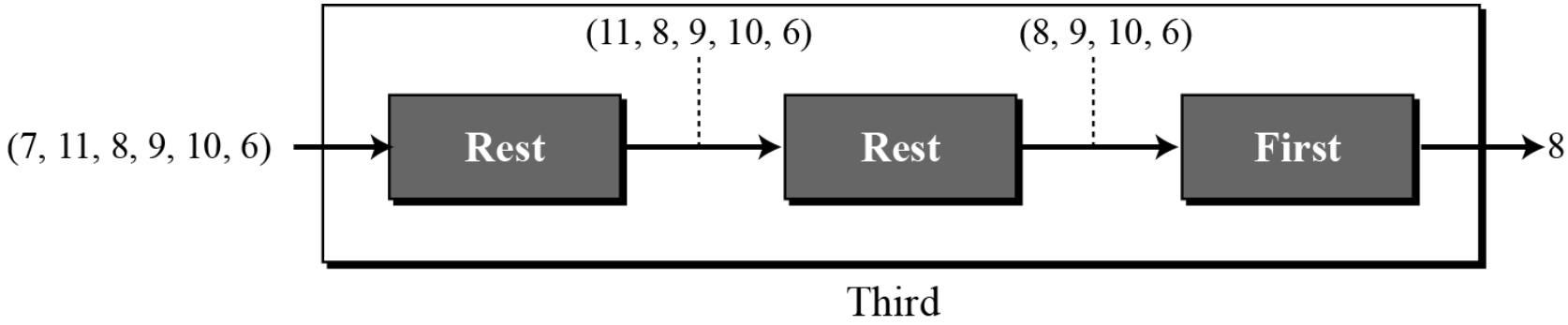

Figure 9.8 **Extracting the third element of a list**

Some functional languages

- LISP (LISt Programming)
- Scheme

The declarative paradigm

A declarative paradigm uses the principle of logical reasoning to answer queries. It is based on formal logic defined by Greek mathematicians and later developed into first-order predicate calculus.

Logical reasoning is based on deduction. Some statements (facts) are given that are assumed to be true, and the logician uses solid rules of logical reasoning to deduce new statements (facts). For example, the famous rule of deduction in logic is:

If (A is B) and (B is C), then (A is C)

Using this rule and the two following facts,

Fact 1: Socrates is a human → A is B
Fact 2: A human is mortal → B is C

we can deduce a new fact:

Fact 3: Socrates is mortal → A is C

Prolog

One of the famous declarative languages is Prolog (PROgramming in LOGic), developed by A. Colmerauer in France in 1972. A program in Prolog is made up of facts and rules. For example, the previous facts about human beings can be stated as:

```
human (John)
mortal (human)
```

The user can then ask:

```
?-mortal (John)
```

and the program will respond with *yes*.

9-4 COMMON CONCEPTS

In this section we conduct a quick navigation through some procedural languages to find common concepts. Some of these concepts are also available in most object-oriented languages because, as we explained, an object-oriented paradigm uses the procedural paradigm when creating methods.

Identifiers

One feature present in all procedural languages, as well as in other languages, is the **identifier**—that is, the name of objects. Identifiers allow us to name objects in the program. For example, each piece of data in a computer is stored at a unique address. If there were no identifiers to represent data locations symbolically, we would have to know and use data addresses to manipulate them. Instead, we simply give data names and let the compiler keep track of where they are physically located.

Data types

A **data type** defines a set of values and a set of operations that can be applied to those values. The set of values for each type is known as the domain for the type. Most languages define two categories of data types: ***simple types*** and ***composite types***.

A simple type is a data type that cannot be broken into smaller data types.

A composite type is a set of elements in which each element is a simple type or a composite type.

Variables

Variables are names for memory locations. As discussed in Chapter 5, each memory location in a computer has an address. Although the addresses are used by the computer internally, it is very inconvenient for the programmer to use addresses. A programmer can use a variable, such as *score*, to store the integer value of a score received in a test. Since a variable holds a data item, it has a type.

Literals

A literal is a predetermined value used in a program. For example, if we need to calculate the area of circle when the value of the radius is stored in the variable *r*, we can use the expression $3.14 \times r^2$, in which the approximate value of π (pi) is used as a literal. In most programming languages we can have integer, real, character and Boolean literals. In most languages, we can also have string literals. To distinguish the character and string literals from the names of variables and other objects, most languages require that the character literals be enclosed in single quotes, such as 'A', and strings to be enclosed in double quotes, such as "Anne".

Constants

The use of literals is not considered good programming practice unless we are sure that the value of the literal will not change with time (such as the value of π in geometry). However, most literals may change value with time.

For this reason, most programming languages define constants. A **constant**, like a variable, is a named location that can store a value, but the value cannot be changed after it has been defined at the beginning of the program. However, if we want to use the program later, we can change just one line at the beginning of the program, the value of the constant.

Inputs and Outputs

Almost every program needs to read and/or write data. These operations can be quite complex, especially when we read and write large files. Most programming languages use a predefined function for input and output.

Data is input by either a statement or a predefined function such as ***scanf*** in the C language.

Data is output by either a statement or a predefined function such as ***printf*** in the C language.

Expressions

An expression is a sequence of operands and operators that reduces to a single value. For example, the following is an expression with a value of 13:

```
2 * 5 + 3
```

An **operator** is a language-specific token that requires an action to be taken. The most familiar operators are drawn from mathematics.

Table 9.3 shows some arithmetic operators used in C, C++, and Java.

Table 9.3 Arithmetic operators

Operator	*Definition*	*Example*
+ – * / %	Addition Subtraction Multiplication Division (the result is the quotient) Division (the result is the remainder)	3 + 5 2 – 4 Num * 5 Sum / Count Count % 4
++ – –	Increment (add 1 to the value of the variable) Decrement (subtract 1 from the value of the variable)	Count++ Count– –

Relational operators compare data to see if a value is greater than, less than, or equal to another value. The result of applying relational operators is a Boolean value (true or false). C, C++ and Java use six relational operators, as shown in Table 9.4:

Table 9.4 Relational operators

Operator	*Definition*	*Example*
<	Less than	Num1 < 5
<=	Less than or equal to	Num1 <= 5
>	Greater than	Num2 > 3
>=	Greater than or equal to	Num2 >= 3
==	Equal to	Num1 == Num2
!=	Not equal to	Num1 != Num2

Logical operators combine Boolean values (true or false) to get a new value. The C language uses three logical operators, as shown in Table 9.5:

Table 9.5 Logical operators

Operator	*Definition*	*Example*
!	Not	! (Num1 < Num2)
&&	And	(Num1 < 5) && (Num2 > 10)
||	Or	(Num1 < 5) || (Num2 > 10)

Statements

A statement causes an action to be performed by the program. It translates directly into one or more executable computer instructions. For example, C, C++ and Java define many types of statements.

An **assignment statement** assigns a value to a variable. In other words, it stores the value in the variable, which has already been created in the declaration section.

A **compound statement** is a unit of code consisting of zero or more statements. It is also known as a block. A compound statement allows a group of statements to be treated as a single entity.

Structured programming strongly recommends the use of the three types of control statements: ***sequence***, ***selection*** and ***repetition***, as we discussed in Chapter 8.

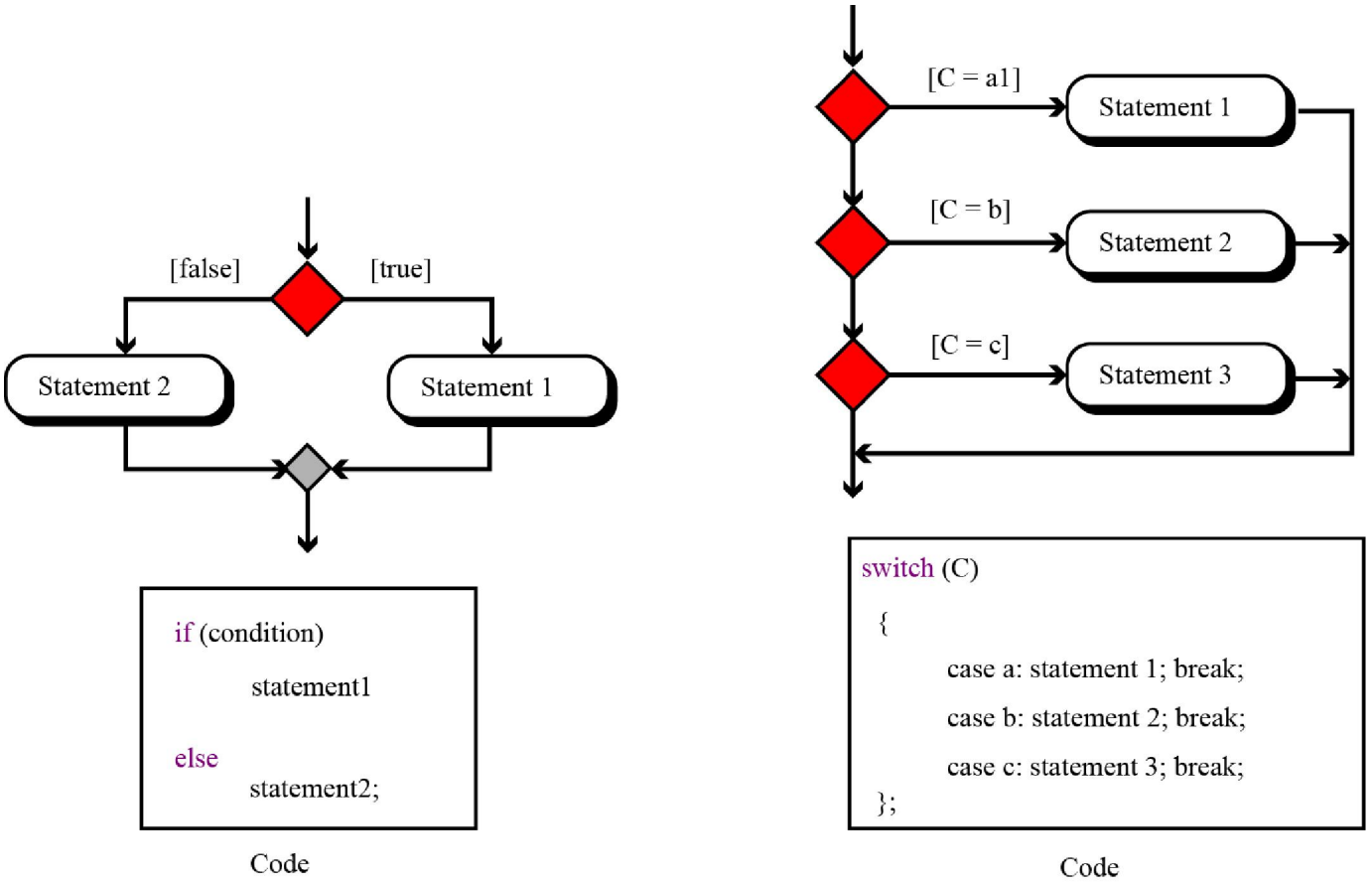

Figure 9.9 Two-way and multi-way decisions

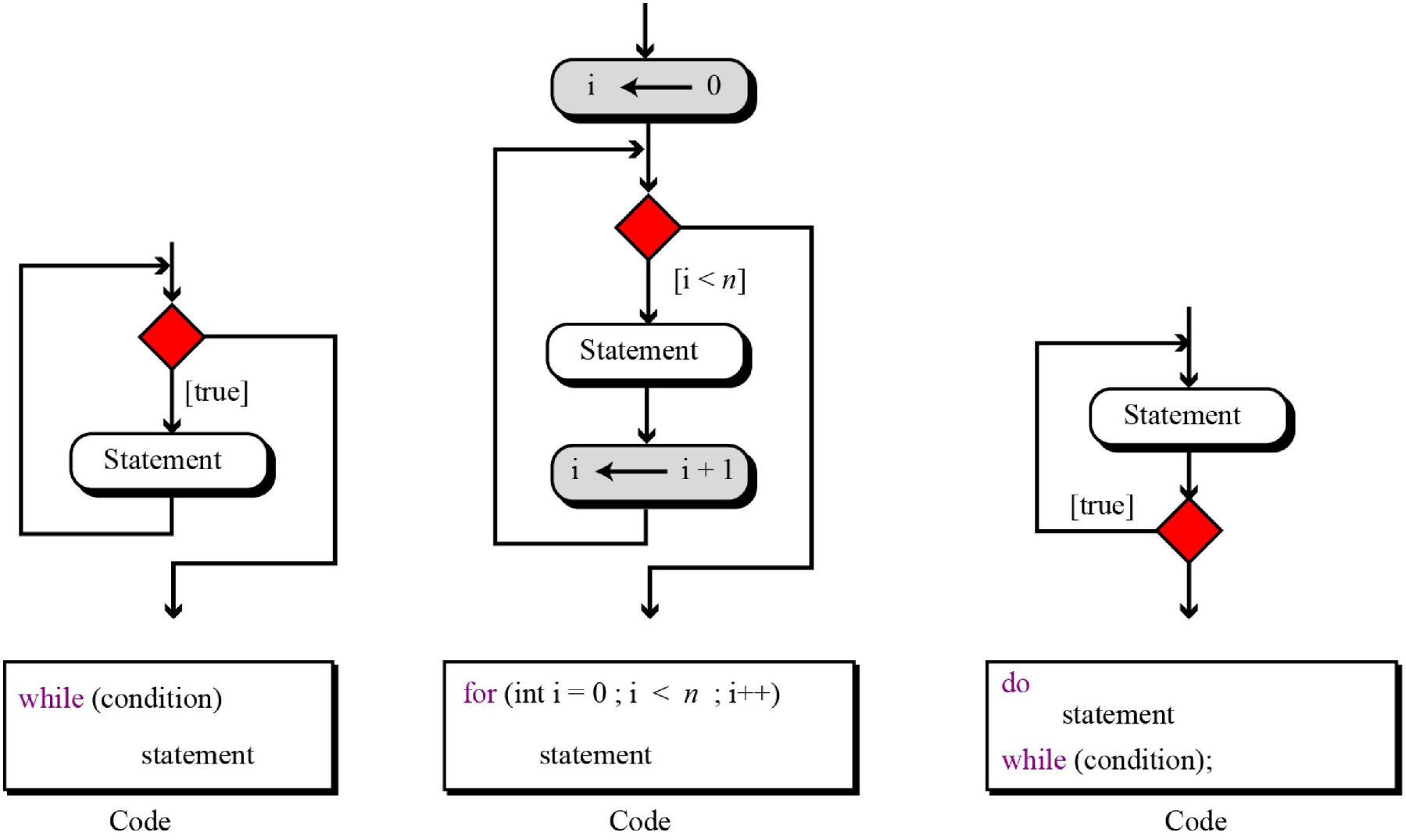

Figure 9.10 Three types of repetition

Subprograms

The idea of subprograms is crucial in procedural languages and to a lesser extent in object-oriented languages. This is useful because the subprogram makes programming more structural: a subprogram to accomplish a specific task can be written once but called many times, just like predefined procedures in the programming language.

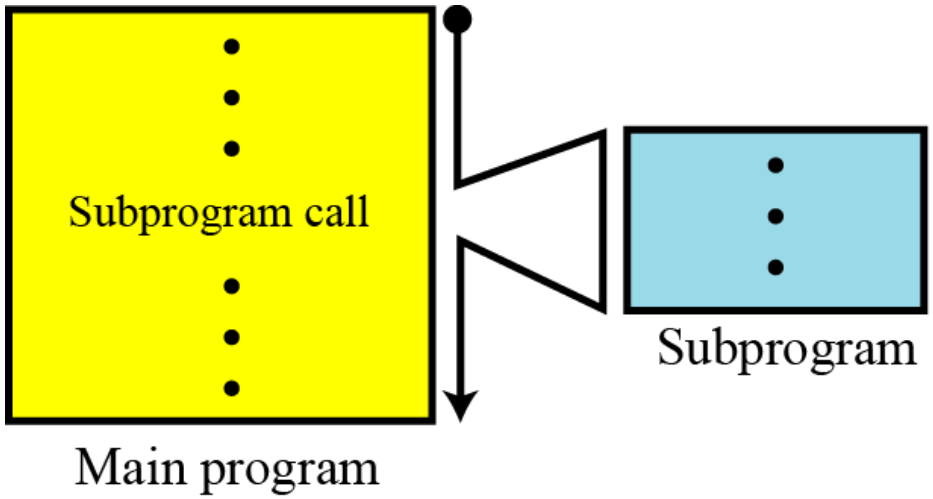

Figure 9.11 **The concept of a subprogram**

In a procedural language, a subprogram, like the main program, can call predefined procedures to operate on local objects. These local objects or **local variables** are created each time the subprogram is called and destroyed when control returns from the subprogram. The local objects belong to the subprograms.

It is rare for a subprogram to act only upon local objects. Most of the time the main program requires a subprogram to act on an object or set of objects created by the main program. In this case, the program and subprogram use **parameters**. These are referred to as actual parameters in the main program and formal parameters in the subprogram.

Pass by value

In parameter pass by value, the main program and the subprogram create two different objects (variables). The object created in the program belongs to the program and the object created in the subprogram belongs to the subprogram. Since the territory is different, the corresponding objects can have the same or different names. Communication between the main program and the subprogram is one-way, from the main program to the subprogram.

Example 9.1

Assume that a subprogram is responsible for carrying out printing for the main program. Each time the main program wants to print a value, it sends it to the subprogram to be printed. The main program has its own variable X, the subprogram has its own variable A. What is sent from the main program to the subprogram is the value of variable X.

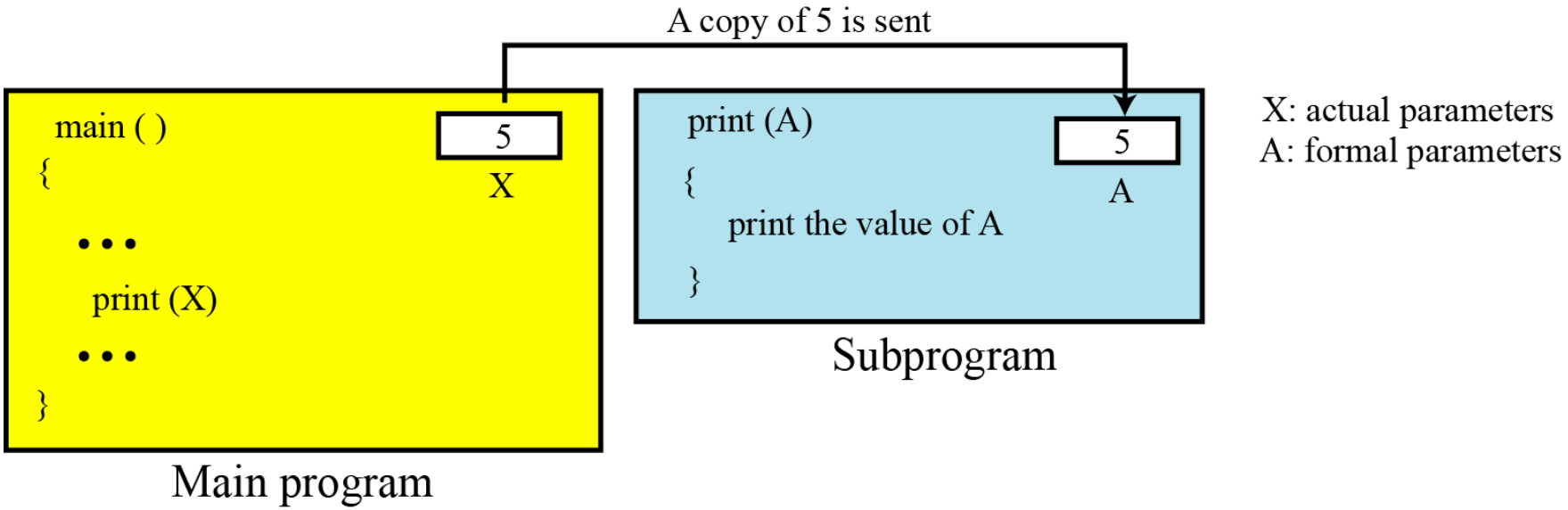

Figure 9.12 **An example of pass by value**

Example 9.2

In Example 9.1, since the main program sends only a value to the subprogram, it does not need to have a variable for this purpose: the main program can just send a literal value to the subprogram. In other words, the main program can call the subprogram as print (X) or print (5).

Example 9.3

An analogy of pass by value in real life is when a friend wants to borrow and read a valued book that you wrote. Since the book is precious, possibly out of print, you make a copy of the book and pass it to your friend. Any harm to the copy therefore does not affect the original.

Example 9.4

Assume that the main program has two variables X and Y that need to swap their values. The main program passes the value of X and Y to the subprogram, which are stored in two variables A and B. The swap subprogram uses a local variable T (temporary) and swaps the two values in A and B, but the original values in X and Y remain the same: they are not swapped.

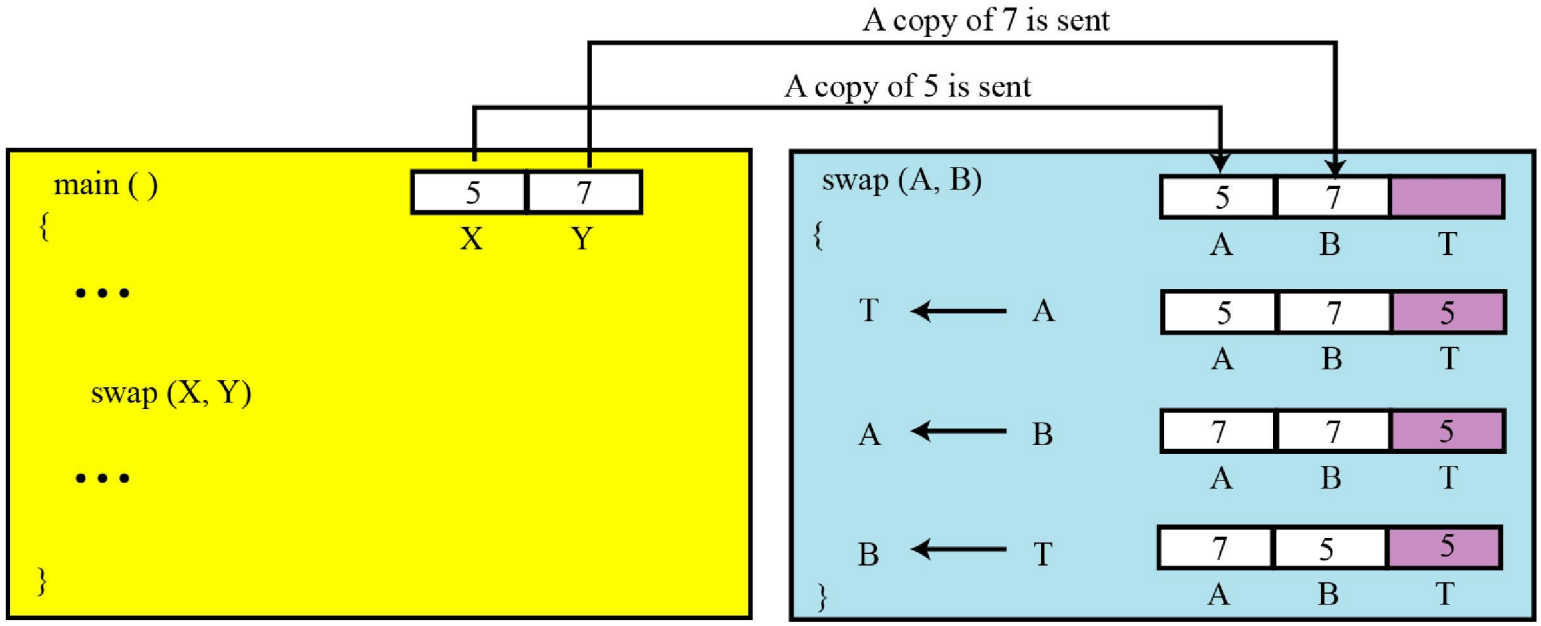

Figure 9.13 An example in which pass by value does not work

Pass by reference

Pass by reference was devised to allow a subprogram to change the value of a variable in the main program. In pass by reference, the variable, which in reality is a location in memory, is shared by the main program and the subprogram. The same variable may have different names in the main program and the subprogram, but both names refer to the same variable. Metaphorically, we can think of pass by reference as a box with two doors: one opens in the main program, the other opens in the subprogram. The main program can leave a value in this box for the subprogram, the subprogram can change the original value and leave a new value for the program in it.

Example 9.5

If we use the same swap subprogram but let the variables be passed by reference, the two values in X and Y are actually exchanged.

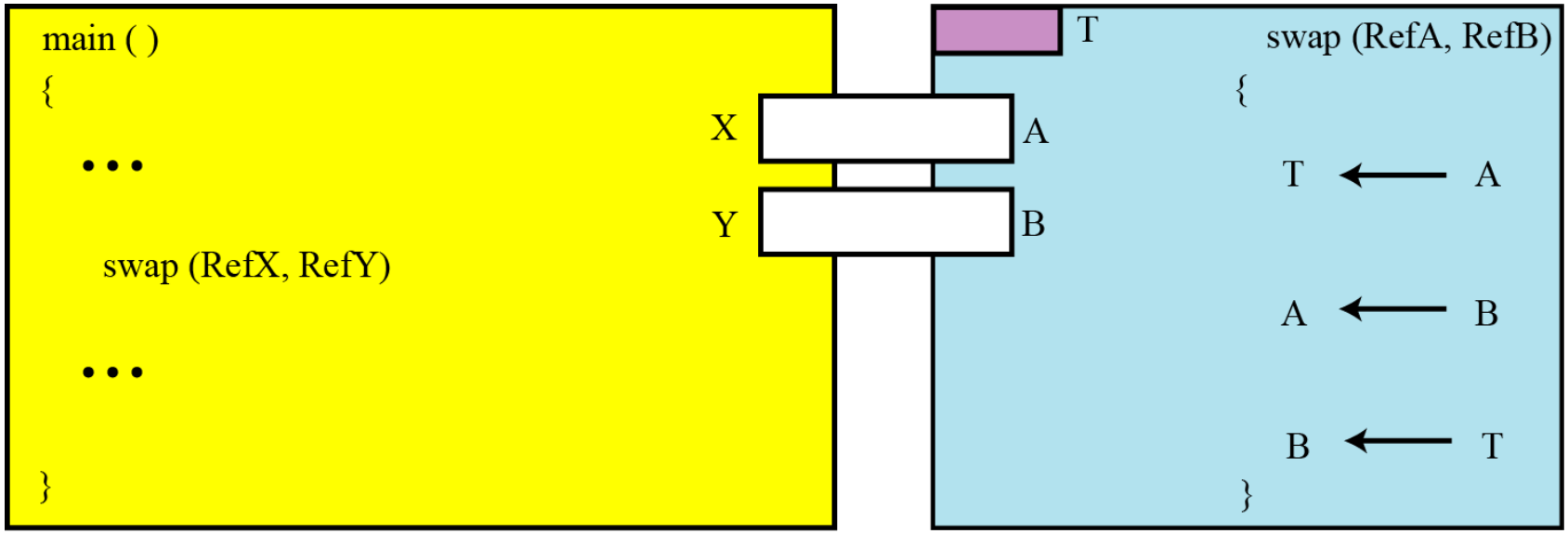

Figure 9.14 An example of pass by reference

Returning values

A subprogram can be designed to return a value or values. This is the way that predefined procedures are designed. When we use the expression C ← A + B, we actually call a procedure add (A, B) that returns a value to be stored in the variable C.