# UNIT-II

# STRUCTURING THE DATA

# INTRODUCTION:-

Computer programs can be viewed as functions that are applied to values of certain input domains to produce results in some other domains. In conventional programming languages, this function is evaluated through a sequence of steps that produce intermediate data that are stored in program variables.

Languages do so by providing features to describe data, the flow of computation, and the overall program organization. This chapter is on mechanisms for structuring and organizing data values

Programming languages organize data through the concept of *type*. Types are

used as a way to classify data according to different categories. They are more, however, than pure sets of data. Data belonging to a type also share certain semantic behaviours.

A type is thus more properly defined as a set of values and a set of operations that can be used to manipulate them. For example, the type BOOLEAN of languages like Ada and Pascal consists of the values TRUE and FALSE; Boolean algebra defines operators NOT, AND, and OR for BOOLEANs.

BOOLEAN values may be created, for example, as a result of the application of relational operators (<, ð, >, Š, +, ¦) among INTEGER expressions. Programming languages usually provide a fixed, built-in set of data types, and mechanisms for structuring more complex data types starting from the elementary ones. Built-in types are discussed in Section 3.1. Constructors that allow more complex data types to be structured starting from built-in types are discussed in Section 3.2. Section 3.3 is about type systems, i.e., on the principles that underlie the organization of a collection of types. The type system adopted by a language affects the programming style enforced by the language.

It may also have a profound influence on the reliability of programs, since it may help prevent errors in the use of data. Moreover, understanding the type system of a language helps us understand subtle and complicated semantic issues. Section 3.4 reviews the type system of existing programming languages. Finally, Section 3.5 is about implementation models.

**3.1 Built-in types and primitive types**

Any programming language is equipped with a finite set of *built-in types* (or *predefined*) types, which normally reflect the behaviour of the underlying hardware. At the hardware level, values belong to the untyped domain of bit strings, which constitutes the underlying universal domain of computer data.

Data belonging to such universal domain are then interpreted differently by hardware instructions, according to different types. At the *hardware level*, a type may thus be considered as a view under which data belonging to the universal type may be manipulated.

As an example of a hypothetical micro computer, the bit string "01001010" might be interpreted as integer "74" (coded in two’s complement representation) when it is the argument of the machine instruction ADD (which does integer addition).

However, it would be interpreted as a bit string by the machine instruction CPL (which does bitwise complement). It might be interpreted as ASCII character "I" if printed by instruction PCH (which prints an ASCII character).

The built-in types of a programming language reflect the different views provided by typical hardware. Examples of built-in types are:

• Booleans, i.e., truth values TRUE and FALSE, along with the set of operations defined by

Boolean algebra;

• Characters, e.g., the set of ASCII characters;

• Integers, e.g., the set of 16-bit values in the range <-32768, 37767>; and

• Reals, e.g., floating point numbers with given size and precision.

Let us analyse what makes built-in types a useful concept. This discussion will help us identify the properties that types in general (i.e., not only the built-in ones) should satisfy. Built-in types can be viewed as a mechanism for *classifying* the data manipulated by a program.

Moreover, they are a way of *protecting* the data against forbidden, or nonsensical, maybe unintended, manipulations of the data. Data of a certain type, in fact, are only manipulable by the operations defined for the type. In more detail, the following are

**Advantages of built-in types:**

1. **Hiding of the underlying representation***.* This is an advantage provided by the abstractions of higher-level languages over lower-level (machine-level) languages. The programmer does not have access to the underlying bit string that represents a value of a certain type. The programmer may change such bit string by applying operations, but the change is visible as a new value of the built-in type, not as a new bit string. Invisibility of the underlying representation has the following benefits:

**Programming style:** The abstraction provided by the language increases program readability by protecting the representation of objects from undisciplined manipulation.This contrasts with the underlying conventional hardware, which does not enforce protection, but usually allows any view to be applied on any bit string. For example, a location containing an integer may be added to one containing a character, or even to a location containing an instruction.

**Modifiability:** The implementation of abstractions may be changed without affecting the programs that make use of the abstractions. Consequently, portability of programs is also improved, that is, programs can be moved to machines that use different internal data representations. One must be careful, however, regarding the precision of data representation that might change for different implementations. For example, the range of representable integer values is different for 16- and 32-bit machines.

Programming languages provide features to read and write values of built-in types, as well as for formatting the output. Such features may be either provided by language instructions or through predefined routines.

Machines perform input/output by interacting with peripheral devices in a complicated and machine-dependent way. High-level languages hide these complications and the physical resources involved in machine input/output (registers, channels, and so on).

**2. Correct use of variables can be checked at translation time**. If the type of each variable is known to the compiler, illegal operations on a variable may be caught while the program is translated. Although type checking does not prevent all possible errors to be caught, it improves our reliance on programs.

For example, in Pascal or Ada, it cannot ensure that J will never be zero in some expression I/J, but it can ensure that it will never be a character.

**3. Resolution of overloaded operators can be done at translation time.**

For readability purposes, operators are often overloaded. For example, + is used for both integer and real addition, \* is used for both integer and real multiplication.

In each program context, however, it should be clear which specific hardware operation is to be invoked, since integer and real arithmetic differs.

In a statically typed language, where all variables are bound to their type at translation time, the binding between an overloaded operator and its corresponding machine operation can be established at translation time, since the types of the operands are known.

This makes the implementation more efficient than in dynamically typed languages, for which it is necessary to keep track of types in run-time descriptors.

**4. Accuracy control*.***In some cases, the programmer can explicitly associate a specification of the accuracy of the representation with a type. For example, FORTRAN allows the user to choose between single and double-precision floating-point numbers. In C, integers can be short int, int, or long int.

Each C compiler is free to choose appropriate size for its underlying hardware, under the restriction that short int and int are at least 16 bits long, long int is at least 32 bits long, and the number of bits of short int is no more than the number of bits of int, which is no more than the number of bits of long int.

In addition, it is possible to specify whether an integer is signed or unsigned. Similarly, C provides both float (for single-precision floating point numbers) and double (for double precision floating point numbers).

Accuracy specification allows the programmer to direct the compiler to allocate the exact amount of storage that is needed to represent the data with the desired precision. Some types can be called *primitive* (or *elementary*). That is, they are not built from other types. Their values are atomic, and cannot be decomposed into simpler constituents.

In most cases, built-in types coincide with primitive types, but there are exceptions. For example, in Ada both Character and String are predefined. Data of type String have constituents of type Character, however. In fact, String is predefined as:

**type** String **is array** (Positive **range** <>) **of** Character

It is also possible to declare new types that are elementary. An example is given by enumeration types in Pascal, C, or Ada. For example, in Pascal one may write:

**type** color = (white, yellow, red, green, blue, black);

The same would be written in Ada as

**type** color **is** (white, yellow, red, green, blue, black);

Similarly, in C one would write:

enum color {white, yellow, red, green, blue, black};

In the three cases, new constants are introduced for a new type. The constants are ordered; i.e., white < yellow < . . . < black. In Pascal and Ada, the built-in successor and predecessor functions can be applied to enumerations.

For example, succ (yellow) in Pascal evaluates to red. Similarly color’pred (red) in Ada evaluates to yellow.

**3.2 Data aggregates and type constructors**

Programming languages allow the programmer to specify aggregations of elementary data objects and, recursively, aggregations of aggregates. They do so by providing a number of *constructors*. The resulting objects are called *compound objects*. A well-known example is the array constructor, which constructs aggregates of homogeneous-type elements.

An aggregate object has a unique name. In some cases, manipulation can be done on a single elementary component at a time, each component being accessible by a suitable selection operation. In many languages, it is also possible to manipulate (e.g, assign and compare) entire aggregates.

Older programming languages, such as FORTRAN and COBOL, provided only a limited number of constructors. For example, FORTRAN only provided the array constructor; COBOL only provided the record constructor.

In addition, through constructors, they simply provided a way to define a new *single* aggregate object, not a type. Later languages, such as Pascal, allowed new compound types to be defined by specifying them as aggregates of simpler types.

In such a way, any number of instances of the newly defined aggregate can be defined. According to such languages, constructors can be used to define both aggregate objects and new aggregate types.

Since in this chapter we concentrate on data types, we review constructors that generate compound data. One should not ignore, however, that routines can also be seen as constructors which allow elementary instructions to be combined to form new operations.

In addition, the distinction between data and routines vanishes in the case of programming languages that treat routines as first class objects, which can be assigned, passed as parameters, be members of data structures, etc.

Type constructors are discussed and exemplified in Section 3.2.1 through Section 3.2.6. Section 3.2.7 discusses how structured data values can be denoted in some languages. Section 3.2.8 will discuss how new types can be defined not only through composition of more elementary types, but also by specifying the operations to be used for their manipulation.

In the discussion, we will first describe the constructors abstractly in terms of a mathematical model, and then we will show how different programming languages provide concrete constructs to represent the abstract model.

**USER-DEFINED TYPES AND ABSTRACT DATA TYPES:**

Modern programming languages provide many ways of defining new types, starting from built-in types. The simplest way, mentioned in Section 3.1, consists of defining new elementary types by enumerating their values.

The constructors reviewed in the previous sections go one step further, since they allow complex data structures to be composed out of the built-in types of the language. Modern languages also allow aggregates built through composition of built-in types to be named as new types. Having given a type name to an aggregate data structure, one can declare as many variables of that type as necessary by simple declarations. For example, after the C declaration which introduces a new type name complex

struct complex {

float real\_part, imaginary\_part;

}

any number of instance variables may be defined to hold complex values:

complex a, b, c, . . .;

By providing appropriate type names, program readability can be improved. In addition, by factoring the definition of similar data structures in a type declaration, modifiability is also improved. A change that needs to be applied to the data structures is applied to the type, not to all variable declarations.

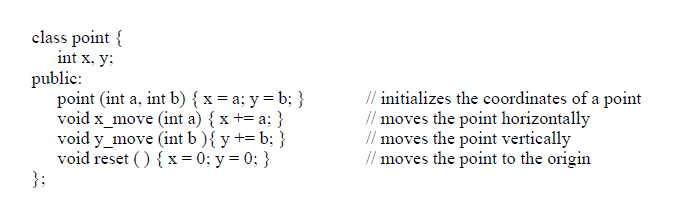
Factorization also reduces the chance of clerical errors and improves consistency. The ability to define a type name for a user defined data structure is only a first step in the direction of supporting data abstractions. As we mentioned in Section 3.1, the two main benefits of introducing types in a language are classification and protection.

Types allow the (otherwise unstructured) world ofdata to be organized as a collection of different categories. Types also allow data to be protected from undesirable manipulations by specifying exactly which operations are legal for objects of a given type and by hiding the concrete representation. Of these two properties, only the former is achieved by defining a user-defined data structure as a type. What is needed is a construct that allows both a data structure and operations to be specified for user defined types. More precisely, we need a construct to define abstract datatypes.

An *abstract data type* is a new type for which we can define the operations to be used for manipulating instances, while the data structure that implements the type is hidden to the users. In what follows we briefly review the constructs provided by C++ and by Eiffel to define abstract data types.

**3.2.8.1 Abstract data types in C++**

Abstract data types can be defined in C++ through the class construct. A class encloses the definition of a new type and explicitly provides the operations that can be invoked for correct use of instances of the type. As an example, Figure 33 shows a class defining the type of the geometrical concept of point.



A class can be viewed as an extension of structures (or records), where fields can be both data and routines. The difference is that only some fields (declared public) are accessible from outside the class. Non-public fields are hidden to the users of the class.

In the example, the class construct encapsulates both the definition of the data structure defined to represent points (the two integer numbers x and y) and of the operations provided to manipulate points. The data structure which defines a geometrical point (two integercoordinates) is not directly accessible by users of the class. Rather, points can only be manipulated by the operations defined as public routines, as shown by the following fragment:

point p1 (1, 3); // instantiates p1 and initializes its value

point p2 (55, 0); // instantiates p2 and initializes its value

point\* p3 = new point (0, 0); // p3 points to the origin

p1.x\_move (3); // moves p1 horizontally

p2.y\_move (99); // moves p2 vertically

p1.reset ( ); // positions p1 at the origin

The fragment shows how operations are invoked on points by means of thedot notation; that is, by writing, “object\_name.public\_routine\_name”.

The only exceptions are the invocations of constructors and destructors. We discuss constructors below; destructors will be discussed in a later example.

A *constructor* is an operation that has the same name of the new type being defined (in the example, point). A constructor is automatically invoked when an object of the class is allocated. In the case of points p1 and p2, this is done automatically when the scope in which they are declared is entered.

In the case of the dynamically allocated point referenced by p3, this is done when the new instruction is executed. Invocation of the constructor allocates the data structure defined by the class and initializes its value according to the constructor’s code.

A special type of constructor is a *copy constructor*. The constructor we haveseen for point builds a point out of two int values. A copy constructor is able to build a point out of an existing point. The signature of the copy constructor would be:

point (point&)

The copy constructor is fundamentally a different kind of constructor becauseit allows us to build a new object from an existing object without knowing the components that constitute the object.

It is also possible to define *generic abstract data types*, i.e., data types that are parametric with respect to the type of components. The construct provided to support this feature is the *template*.

As an example, the C++ template implements an abstract data type stack which is parametric with respect to the type of elements that it can store and manage according to a last-in first-out policy. The figure also describes a fragment that defines data objects of instantiated generic types:

template<class T> class Stack{

int size;

T\* top;

T\* s;

public:

Stack (int sz) {top = s = new T [size = sz];}

~Stack ( ) {delete [ ] s;} //destructor

void push (T el) {\*top++ = el;}

T pop ( ) {return \*--top;}

int length ( ) {return top - s;}

};

void fun ( ) {

Stack<int>int\_st (30);

Stack<item>item\_st (100);

. . .

int\_st.push (9);

. . .

}

**FIGURE 34.**A C++ generic abstract data type and its instantiation

Another class can become a client of POINT by declaring references to

objects of type POINT:

p1, p2: POINT;

Objects can be created and bound to such references, and then manipulated

according to the type’s operations:

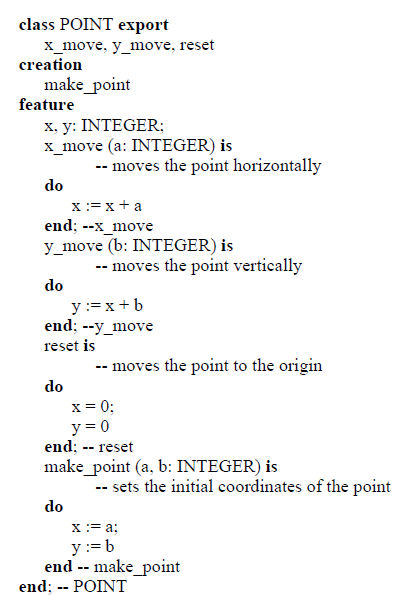
p1.make\_point (4, 7);

p2.make\_point (55, 0);

p1.move\_x (3);

p2.move\_y (99);

p1.reset ( );



C++ instances of an abstract data type can be either stack objects or heapobjects.

**class**POINT **export**

x\_move, y\_move, reset

**creation**

make\_point

**feature**

x, y: INTEGER;

x\_move (a: INTEGER) **is**

-- moves the point horizontally

**do**

x := x + a

**end**; --x\_move

y\_move (b: INTEGER) **is**

-- moves the point vertically

**do**

y := x + b

**end**; --y\_move

reset**is**

-- moves the point to the origin

**do**

x = 0;

y = 0

**end**; -- reset

make\_point (a, b: INTEGER) **is**

-- sets the initial coordinates of the point

**do**

x := a;

y := b

**end**-- make\_point

**end**; -- POINT

**FIGURE 35.**An Eiffel class defining point

That is, they can be associated both with automatic variables or be dynamically allocated and referred to by pointers. In the example in Figure33, the objects associated with variables p1 and p2 are (automatically) allocated on the stack; the objects to which p3 points is dynamically allocated on the heap.

In Eiffel, all objects (except for built-in elementary values like integers) are implicitly allocated on the heap and made accessible via pointers. In the example of Figure 35, p1 and p2 are in fact pointers to objects, which are allocated (and initialized) by the invocation of the creation operation. The Eiffel make\_point is analogous to the C++ constructor but must be called explicitly to create the object. The C++ concept of copy construction—creating a new object from an existing like object—is not associated with eachobject. Rather, the langauge provides a function named **clone** which can becalled with an object of any type to create a new object which is a copy of the original object.

The Eiffel language assumes a set of principles that should guide programmers in a disciplined and methodical development of programs. It is possible to associate a class with an *invariant property*, i.e., a predicate that characterizes all possible correct instances of the type.

For example, consider a variant NON\_AXIAL\_INT\_POINT of class POINT which describes the set of points with integer coordinates that do not belong to the axes x and y. The x- and y-coordinates of the elements of class NON\_AXIAL\_INT\_POINT cannot be zero; that is, the invariant property for such class is written as:

x \* y /= 0

To ensure that the invariant is satisfied, suitable constraints must apply to the exported routines of the class. This is stated in Eiffel by defining two predicates: a *precondition* and a *postcondition*. These two predicates characterize the states in which the routine can start and should end its execution.

A class is said to be consistent if it satisfies the following conditions:

1. for every creation routine, if its precondition holds prior to execution, the invariant holds

upon termination

2. for every exported routine, if the precondition and the invariant hold prior to execution, the postcondition and the invariant hold upon termination.

If these two rules are satisfied, by simple induction one can prove that the invariant will always be true for all reachable object states.

**class**NON\_AXIAL\_POINT **export**

x\_move, y\_move

**creation**

make\_point

**feature**

x, y: INTEGER;

x\_move (a: INTEGER) **is**

-- moves the point horizontally

**require**

x + a /= 0

**do**

x := x + a

**ensure**

x /= 0

**end**; --x\_move

y\_move (b: INTEGER) **is**

-- moves the point vertically

**require**

y + b /= 0

**do**

y := x + b

**ensure**

y /= 0

**end**; --y\_move

make\_point (a, b: INTEGER) **is**

-- sets the initial coordinates of the point

**require**

a \* b /= 0

**do**

x := a;

y := b

**end**-- make\_point

**invariant**

x \* y /= 0

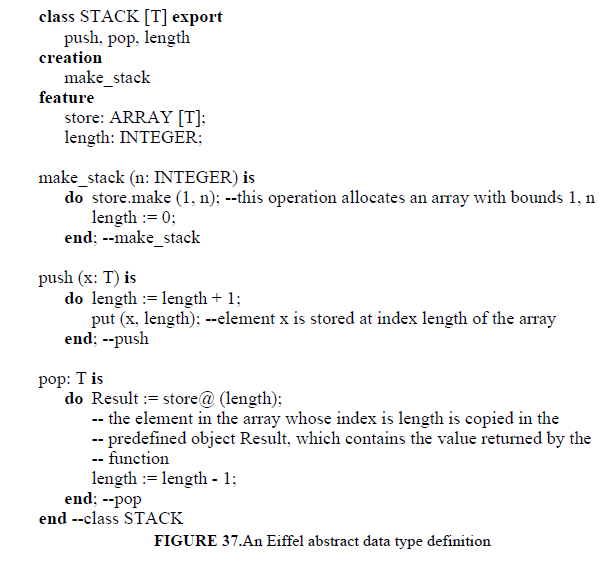
**end**; -- NON\_AXIAL\_POINT

**FIGURE 36.** An Eiffel class defining a point that may not lie on the axes x and y are optional, although their use is good programming practice. If they are present, an Eiffel implementation can check such properties at runtime.

This is an effective way of debugging Eiffel programs. As we will see in Chapter4, it also supports systematic programmed ways of error handling.

Eiffel supports the implementation of generic abstract data types, via *generic classes*. As an example, Figure 37 shows an implementation of a genericstack abstract data type in Eiffel.

The definition of preconditions, postconditions,and invariants are left to the reader as an exercise.



adopted by a language, defined as the set of rules used by the language to structure and organize its collection of types. Understanding the type system adopted by a language is perhaps the major step in understanding the language’s semantics.

Our treatment in this section is rather abstract, and does not refer to any specific programming language features. The only assumption made is that a type is defined as a set of values and a set of operations that can be applied to such values.

As usual, since values in our context are stored somewhere in the

memory of a computer, we use the term *object* (or *data object*) to denote boththe storage and the stored value. The operations defined for a type are theonly way of manipulating its instance objects: they protect data objects from any illegal uses. Any attempt to manipulate objects with illegal operations is a *type error*.

A program is said to be *type safe* (or *type secure*) if all operations

in the program are guaranteed to always apply to data of the correct type, i.e., no type errors will ever occur.

**3.3 Type Systems:**

Types are a fundamental semantic concept of programming languages. Moreover, programming languages differ in the way types are defined and behave, and typing issues are often quite subtle. Having discussed type concepts informally in different languages so far, we now review the foundations for a theory of types.

The goal is to help the reader understand the *type system* adopted by a language, defined as the set of rules used by the language to structure and organize its collection of types. Understanding the type system adopted by a language is perhaps the major step in understanding the language’s semantics.

Our treatment in this section is rather abstract, and does not refer to any specific programming language features. The only assumption made is that a type is defined as a set of values and a set of operations that can be applied to such values.

As usual, since values in our context are stored somewhere in the

memory of a computer, we use the term *object* (or *data object*) to denote both the storage and the stored value.

The operations defined for a type are the only way of manipulating its instance objects: they protect data objects from any illegal uses. Any attempt to manipulate objects with illegal operations is a *type error*. A program is said to be *type safe* (or *type secure*) if all operations in the program are guaranteed to always apply to data of the correct type, i.e., no type errors will ever occur.

**3.3.1 Static versus Dynamic Program Checking:**

Before focusing our discussion on type errors, a brief digression is necessary to discuss more generally the kinds of errors that may occur in a program, the different times at which such errors can be checked, and the effect of checking times on the quality of the resulting programs.

Errors can be classified in two categories: language errors and application errors. *Language errors* are syntactic and semantic errors in the use of the programming language. *Application errors* are deviations of the program behaviour with respect to specifications (assuming specifications capture the required behaviour correctly).

The programming language should facilitate both kinds of errors to be identified and removed. Ideally, it should help prevent them from being introduced in the program. In general, programs that are readable and well structured are less error prone and easier to check.

Hereafter we concentrate on language errors. A discussion of application errors is out of the scope of this book: software design methods address application errors. Therefore, here the term “error” implicitly refers to “language error”.

Error checking can be accomplished in different ways, that can be classified in two broad categories: static and dynamic. *Dynamic checking* requires the program to be executed on sample input data. *Static checking* does not. In general, if a check can be performed statically, it is preferable to do so instead of delaying the check to run-time for two main reasons.

First, potential errors are detected at run time only if one can provide input data that cause the errorto be revealed. For example, a type error might exist in a portion of the program that is not executed by the given input data. Second, dynamic checking

slows down program execution.

Static checking is often called compile-time (or translation-time) checking. Since programs may be subject to separate compilation

and some static checks might occur at link time.

For example, the possible mismatch between a routine called by one module and defined in another might be checked at link time. Conventional linkers, unfortunately, seldom perform such checks. For simplicity, we will continue to use the terms static checking and compile-time (or translation-time) checking interchangeably.

Static checking, though preferable to dynamic checking, does not uncover all language errors. Some errors only manifest themselves at run time. For example, if div is the operator for integer division, the compiler might check that both operands are integer. However, the program would be erroneous if the value of the divisor is zero. This possibility, in general, cannot be checked by the compiler.

**3.3.1 Strong typing and Type Checking:**

The type system of a language was defined as the set of rules to be followed to define and manipulate program data. Such rules constrain the set of legal programs that can be written in a language. The goal of a type system is to prevent the writing of type unsafe programs as much as possible.

A type system is said to be *strong* if it guarantees type safety; i.e., programs written by following the restrictions of the type system are guaranteed not to generate type errors. A language with a strong type system is said to be a *stronglytyped language*. If a language is strongly typed, the absence of type errors from programs can be guaranteed by the compiler. A type system is said to be *weak* if it is not strong. Similarly, a *weakly typed language* is a language that is not strongly typed. Such languages are said to obey to a *static type system*. Precisely, such a type system requires that the type of every expressions be known at compile time. An example of a static type system can be achieved by requiring that

1. only built-in types can be used;

2. all variables are declared with an associated type;

3. all operations are specified by stating the types of the required operands and the type of the result.

A statically typed language is a strongly typed language, but there are

strongly typed languages that are not statically typed. For example, Binding between a variable and its type cannot be established at compile time, and yet the rules of the type system guarantee type safety; i.e., they guarantee that correctly compiled programs will execute without generating type errors.

In general, we may observe that many strong type systems exist. Since all of them guarantee type safety, how should a language designer choose a type system when defining a new programming language? There are two conflicting requirements to be accommodated in such a design decision: the size of

the set of legal programs and the efficiency of the type checking procedure performed by the compiler.

Since a type system restricts the set of programs that can be written, we might come out with rules that allow only very simple programs. In principle, a type system which restricts the set of legal programs to the empty set is a strong type system. It is also trivial to check. But it is obviously useless. The previous example of static typing allows for a simplechecking procedure, but it is overly restrictive. Dynamic typing, is a very powerful programming facility thatcan be combined with strong typing. In such a case, however, the is required to perform a complex type checking procedure.

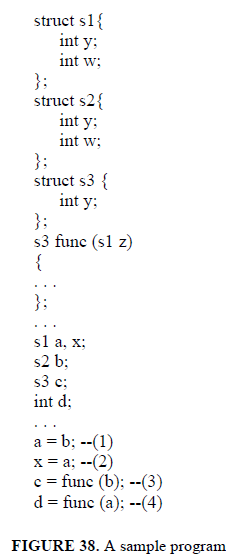
**3.3.3 Type Compatibility:**

A strict type system might require operations that expect an operand of a type T to be invoked legally only with a parameter of type T. Languages, however, often allow more flexibility, by defining when an operand of another type–say Q–is also acceptable without violating type safety. In such a case, we say that the language defines whether, in the context of a given operation, type Q is *compatible* with type T. Type compatibility is also sometimes called *conformance*or *equivalence*.

When compatibility is defined precisely by the type system, a type checking procedure can verify that all operations are always

invoked correctly, i.e., the types of the operands are compatible with the types expected by the operation.

Thus a language defining a notion of type compatibility can still have a strong type system. Figure 38 shows a sample program fragment written in a hypothetical programming language.



The strict conformance rule where a type name is only compatible with itself is called *name compatibility*. Under name compatibility, in the above example, instruction (2) is type correct, since a and x have the same type name.

Instruction (1) contains a type error, because a and b have different types. Similarly, instructions (3) and (4) contain type errors. In (3) the function is called with an argument of incompatible type; in (4) the value returned by the function is assigned to a variable of an incompatible type.

*Structural compatibility* is another possible conformance rule that languagesmay adopt*.* Type T1 is structurally compatible with type T2 if they have the

same structure. This can be defined recursively as follows:

• T1 is name compatible with T2; or

• T1 and T2 are defined by applying the same type constructor to structurally compatible corresponding type components.

According to structural equivalence, instructions (1), (2), and (3) are type correct. Instruction (4) contains a type error, since type s3 is not compatible with int. Note that the definition we gave does not clearly state what happens with the field names of Cartesian products (i.e., whether they are ignored in the check or they are required to coincide and whether structurally compatible fields are required to occur in the same order or not).

For simplicity, we assume that they are required to coincide and to occur in the same order. In such a case, if we rename the fields of s2 as y1 and w1, or permute their occurrence, s2 would no longer be compatible with s1.

Name compatibility is easier to implement than structural compatibility, which requires a recursive traversal of a data structure. Name compatibility is also much stronger than structural compatibility.

Actually, structural compatibility goes to the extreme where type names are totally ignored in the check. Structural compatibility makes the classification of data objects implied by types exceedingly coarse.

For example, having defined the following two types:

struct complex {

float a;

float b;

};

struct point {

float a;

float b;

};

The programmer can instantiate variables to represent–say–points on a planeand values of a.c. voltage. The type system allows to use them interchangeably, although most likely the programmer has chosen two different type names in order to keep the different sets of objects separate.

In conclusion, name compatibility is often preferable. It prevents two types to be considered compatible just because their structure happens to be identical by coincidence. Often programming languages do not take much care in defining the adopted notion of type compatibility they adopt. This issue is left to be defined by the implementation.

An unfortunate consequence is that different implementations may adopt different notions, and thus a program accepted by a compiler might be rejected by another. This unfortunate case happened, for example, when Pascal was originally defined, although later ISO Pascal defined type compatibility rigorously, mainly based on name compatibility.

C adopts structural compatibility for all types, except structures, for which name compatibility is required. Type compatibility in Ada is defined via name compatibility. Since the language introduces the concept of a subtype (see also Section 3.3.5), objects belonging to different subtypes of the same type are compatible.

In Ada, when a variable is defined by means of a constructor, as in

IA: array (INTEGER range 1. .100) of INTEGER; a brand new anonymous type is implicitly introduced, followed by a variable declaration:

type ANONYMOUS\_1 is array (INTEGER range 1. .100) of INTEGER;

IA: ANONYMOUS\_1;

Thus, if two variables IA and IB are declared:

IA: array (INTEGER range 1. .100) of INTEGER;

IB: array (INTEGER range 1. .100) of INTEGER;

the two variables are considered to have noncompatible types, since theiranonymous type names would be different.

**3.3.4 Type Conversions:**

Suppose that an object of type T1 is expected by some operation at some point of a program. Also, suppose that an object of type T2 is available and we wish to apply the operation to such object. If T1 and T2 are compatible according to the type system, the application of the operation would be type correct. If they are not, one might wish to apply a type conversion from T2 to T1 in order to make the operation possible.

More precisely, let an operation be defined by a function fun expecting a parameter of type T1 and evaluating a result of type R1:

fun: T1 -> R1 Let x2 be a variable of type T2 and y2 of type R2. Suppose that T1 and T2 (R1 andR2) are not compatible. How can fun be applied to x2 and the result of the routine be assigned to y2? This would require two conversion functions to be available, t21 and r12, transforming objects of type T2 into objects of type T1 and objects of type R1 into objects of type R2, respectively:

t21: T2 -> T1

r12: R1 -> R2

Thus, the intended action can be performed by first applying t21 to x2, evaluating fun with such argument, applying r12 to the result of the function, and finally assigning the result to y2. That is:

(i) y2 = r12(fun (t21(x2)))

For some languages any required conversions are applied automatically by the compiler. Following the Algol 68 terminology, we will call such automatic conversions *coercions*. In the example, if coercions are available, the programmer might simply write

(ii) y2 = fun (x2)

and the compiler would automatically convert (ii) into (i).

In general, the kind of coercion that may occur at a given point (if any) depends on the context. For example, in C if we write

x = x + z;

where z is float and x is int, x is coerced to float to evaluate the arithmetic operator+ (which stands for real addition), and the result is coerced to int for the assignment.

That is, the arithmetic coercion is from int to float, but the assignment coercion is from float to int. C provides a simple coercion system. In addition, explicit conversions can be applied in C using the *cast* construct. For example, a cast can be used to override an undesirable coercion that would otherwise be applied in a given context.

For example, in the above assignment, one can force a conversion of z to int by writing

x = x + (int) z;

Such an explicit conversion in C is semantically defined by assuming that the expression to be converted is implicitly assigned to an unnamed variable of the type specified in the cast, using the coercion rules of the language.

Ada does not provide any coercions. Whenever a conversion is allowed by the language, it must be invoked explicitly. For example, if X is declared as a FLOAT variable and I is an INTEGER, assigning X to I can be accomplished by the instruction

I := INTEGER(X);

The conversion function INTEGER provided by Ada computes an integer from a floating point value by rounding to the nearest integer. The existence of coercion rules in a language has both advantages and disadvantages.

The advantage is that many desirable conversions are automatically provided by the implementation. The disadvantage is that since implicit transformations happen behind the scenes, the language becomes complicated and programs may be obscure.

In addition, coercions weaken the usefulness of type checking, since they override the declared type of objects with default,context sensitive transformations. For example, Algol 68 consistently applies the principle of implicit conversions to the extreme. The type of the value required at any given point in an Algol 68 program can be determined from the context. But the way coercions interact with other constructs of the language can make programs quite hard to understand. Unexpected difficulties, in particular, arise because of the interaction between coercions and overloading of operators and routines.

**3.3.4 Types and Subtypes:**

If a type is defined as a set of values with an associated set of operations, a subtype can be defined to be a subset of those values (and, for simplicity, the same operations). In this section we explore this notion in the context of conventional languages, ignoring the ability to specify user-defined operations for subtypes. The concept of subtype will have a richer semantics in the context of object-oriented languages, as will be discussed in Chapter 6. If ST is a subtype of T, T is also called ST’s supertype (or parent type).

We assume that the operations defined for T are automatically inherited by ST. A language supporting subtypes must define:

1. A way to define subsets of a given type;

2. Compatibility rules between a subtype and its supertype.

Pascal was the first programming language to introduce the concept of a subtype, as a subrange of any discrete ordinal type (i.e., integers, boolean, acter, enumerations, or a subrange thereof). For example, in Pascal one may define natural numbers and digits as follows:

**type**natural = 0. .maxint;

digit = 0. .9;

small = -9. .9;

wheremaxint is the maximum integer value representable by an implementation.

A Pascal program can only define a subset of contiguous values of a discrete type. For example, it cannot define a subtype EVEN of all even integers ormultiples of ten in the range -1000. .1000. Different subtypes of a given type are considered to be compatible among themselves and with the supertype.

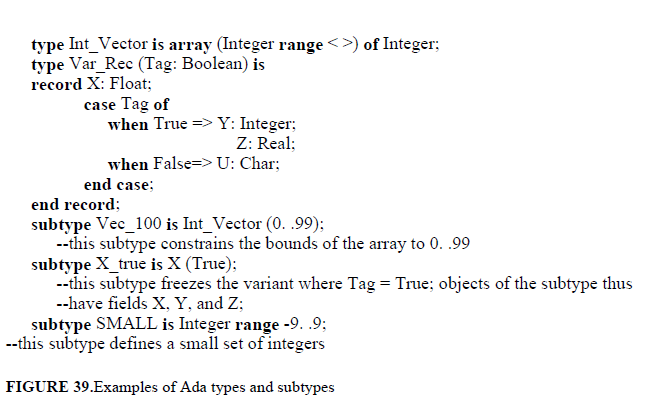
However, type safe operations are not guaranteed to evaluate with no error. No error arises if an object of a subtype is provided in an expression where an object of its supertype is expected. For example, if an expression requires an integer, one may provide a natural; if it expects a natural, one might provide a digit.

If, however, a small is provided where a digit is expected, an error

arises if the value provided is not in the range expected. That is, if an argument of type T is provided to an operation expecting an operand of type R, the expression is type safe if either R or T is a subtype of the other, or both are subtypes of another type Q. No value error will occur at run time if T is a subtype of R.

In all other cases, the operation must be checked at run time and an error may arise if the value transmitted does not belong to the expected type. Ada provides a richer notion of subtype than Pascal. A subtype of an array type can constrain its index; a subtype of a variant record type can freeze the variant; a subtype of a discrete ordinal type is a finite subset of contiguous values.

Examples of Ada types and subtypes are shown in Figure 39.



A possibility is provided by languages like Ada, C++, and Eiffel, where generic types must be explicitly instantiated at compile time by binding parameter types to “real” types, that are known at compile time. This achieves static typing for each instance of each generic type, and therefore each instance is statically checked to ensure type safety.

Instantiation, however, is not required in languages like ML. As we will see in Chapter 7, however, the language still ensures type safety statically.

**3.4 The type structure of Existing Languages:**

The type structure of a number of existing programming languages. The description provides an overall hierarchical taxonomy of the features provided by each language for data structuring. For a full understanding of language semantics, such description must be complemented by a precise understanding of the rules of the type system (strong typing, type compatibility, type conversion, subtyping, genericity, and polymorphic features).

**3.4.1 Pascal:**

The type structure of Pascal is described in Figure 42. A different decomposition of unstructured types would be in terms of ordinal types and real types. *Ordinal types* comprise integers, booleans, characters, enumerations, and sub ranges. Ordinal types are characterized by a discrete set of values, each of which has (at most) a unique predecessor and a unique successor, evaluated by the built-in functions pred and succ, respectively.

Figure 42 shows how structured types can be built in Pascal. Recursive data structures are defined via pointers. Cartesian products are defined by records. Unions and discriminated unions are described by variant records. Comments on these constructs, and particularly on their possible insecurities, were given earlier.

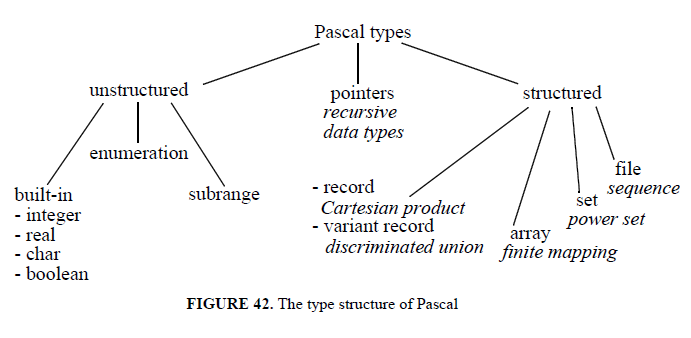
Finite mappings are described by *arrays*, whose index type must be an ordinal type. The size of an array is frozen when the array type is defined, and cannot change during execution. Pascal regards arrays with different index types as different types. For example, a1 and a2 below are different types.

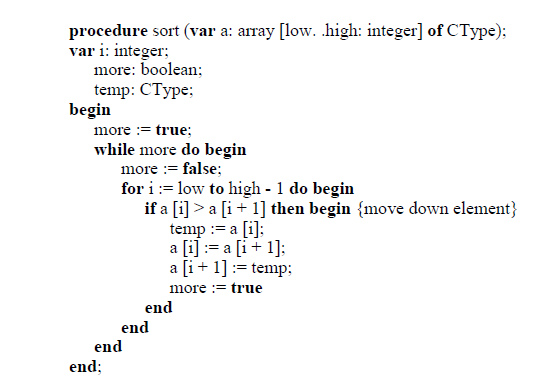
**type**a1 = **array** [1. .50] of integer;

a2 = **array** [1. .70] of integer;

This was a serious problem in Pascal as originally defined. Because procedures require formal parameters to have a specified type, it was not possible, for example, to write a procedure capable of sorting both arrays of type a1 and type a2.

This feature, called the *conformant array*, allows the formal array parameter of a procedure to conform to the size of the actual array parameter. The actual and formal parameters are required to have the same number of indexes and the same component type. The example illustrates the use of conformant arrays.





**FIGURE 41.**An example of conformant arrays in Pascal

When the procedure sort is called with a one-dimensional array parameter, low and high assume the values of the lower and upper bounds of the actual parameter, respectively. Another solution, not available in Pascal, could have been based on genericity(i.e., allowing a procedure to be generic with respect to the array bounds).

More generally, Pascal provides only limited forms of ad-hoc polymorphism. Some built-in operators, like + or succ, are overloaded. In fact, succis applicable to operators of any ordinal type. Similarly, + can be applied to integer operands, real operands, or even sets (in which case it denotes the union operator). The language also defines cases of coercion. For example, if an integer is added to a real, the integer is coerced to a real, and the addition is performed.

As we mentioned earlier, Pascal is not a strongly typed language. For example, its original definition did not carefully define the concept of type compatibility. Moreover, subtypes are defined by the language as new types, and thus the type of an expression may in general depend on the values of the operands at run time.

**3.4.2 C++**

The type structure of C++ is given in Figure 43. C++ distinguishes between to categories of types: *fundamental types* and *derived type*. Fundamental types are either *integral* or *floating*. Integral types comprise char, short int, int, long int, which can be used for representing integers of different sizes.

Floating-point types comprise float, double, and long double. New integral types may be declared via enumerations. For example

enum my\_small\_set {low = 1, medium = 5, high = 10}

Arrays are declared by providing a constant expression, which defines the number of elements in the array. For example

float fa [15];

declares an array of floating-point numbers that can be indexed with a value in the range 0.

**C++ distinguishes between pointers and references.**

A reference is an alias for an object. Therefore, once a reference is bound to an object, it cannot be made to refer to a different object. For example, having declared

int i = 5;

int & j = i;

i and j denote the same object, which contains the value 5.

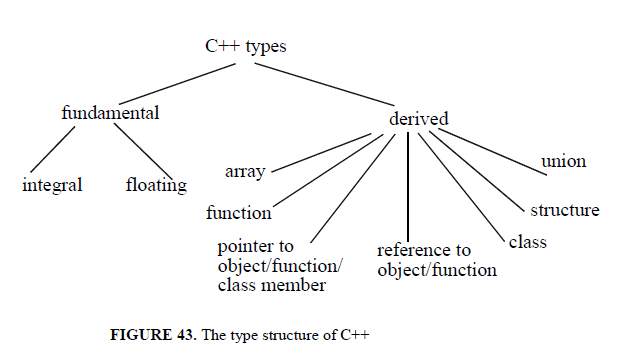
When the reference’s lifetime starts, it must be bound to an object. Thus, for example, one cannot write the following declaration

int& j;

It is possible, however, to bind a reference to an object through parameter passing. This is actually the way C++ supports parameter passing by reference. That is, it is possible to assign a new value to a pointer, not only to the object referenced by the pointer. Of course, it is possible to use pointers also in the case where a reference would do. In fact, references are not provided by C, but were added to the language by C++. As we mentioned, however, pointers are extremely powerful, but difficult to manage, and often dangerous to use.

As we discussed, this allows abstract data type implementations to be provided. If no protection is needed on the data declared in a class, classes without default access restrictions can be defined as structures, via the struct construct.

Finally, two other kinds of types can be derived in C++ by using the function and the union constructs. As we already observed, the function construct defines a new data object. It is thus possible to define pointers and references to functions, pass functions as parameters, etc.



**3.4.3 ADA:**

The type structure of Ada is described in Figure 44. Such structure is discussed and evaluated in this section, except for concurrency related types, which are discussed in Chapter 4, and tagged types, which are discussed in Chapters 5 and 6. *Unstructured* (*scalar*) types can be both numeric (i.e., integers and reals) and enumerations. All scalar types are ordered, i.e., relational operators are defined on them.

Enumeration types are similar to those provided by Pascal.*Integer* types comprise a set of consecutive integer values. An integer type may be either signed or modular. Any *signed* integer type is a subrange of System.Min\_Int..System.Max\_Int, which denote the minimum and maximum integer representable in a given Ada implementation.

A *modular* integer is an absolute value, greater than or equal to zero. The Ada language predefines a signed integer type, called Integer. Other integer types, such as Long\_Integeror Short\_Integer, may also be predefined by an implementation. Programmerdefined integer types may be specified as shown by the following examples:

**type**Small\_Int**is range** -10. .10; -- range bounds may be any static expressions

**type**Two Digit **is mod** 100; --the values range from 0 to 99;

--in general, the bound must be a static expression

As we mentioned, Ada allows *subtypes* to be defined from given types. Subtypes do not define a new type. They conform to the notion of subtype .

Two subtypes of Integer are predefined in Ada:

**subtype**Natural **is** Integer **range** 0. .INTEGER’LAST;

**subtype**Positive **is** Integer **range** 1. .INTEGER’LAST;

Ada provides a rich and elaborate set of facilities for dealing with *real* values; only the basic aspects will be reviewed here. Real types provided by the language are just an approximation of their mathematical counterpart (universal real, in the Ada terminology).

In fact, the fixed number of bits used by the implementation to represent real values makes it possible to store the exact value of only a limited subset of the universal reals. Other real numbers are approximated. Real types in Ada come in two forms: floating point and fixed point.

A *floating-point* real type approximates a universal real with an error

that is relative to the number’s absolute value. A *fixed-point* real approximatesa universal real with an error that is independent of the value being represented.The language predefines one floating-point real type, called Float.

It is left to the implementation whether additional real types, such as Short\_Floator Long\_Float, should be provided. The programmer can define additional floating-point real types, such as:

**type** Float\_1 **is digits** 10;

The digits clause specifies the minimum number of significant decimal digits required for the type. Such minimum number of digits defines the relative error bound in the approximate representation of universal reals. Given a floating point real type, attribute Digits gives the minimum number of digits associated with the type. Thus, Float\_1’Digits yields 10, whereas Float’Digitsyields an implementation dependent value. Fixed-point real types provide another way of approximating universal reals, where the approximation error is independent of the value being represented.

Such error bound is specified as the delta of the fixed-point real. An ordinary fixed-point real type is declared in Ada as

**type**Fix\_Pt**is delta** 0.01 **range** 0.00. .100;

The declaration defines both the delta and the range of values. A decimal fixed-point type is specified by providing the delta and the number of decimal digits. For example

**type**Dec\_Pt**is delta** 0.01 **digits** 3; includes at least the range -99.9. .99.9

Ada’s *structured* (or *composite*) types comprise arrays and records. *Arrays* can have statically known bounds, as in Pascal. For example

**type** Month **is** (JAn, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);

**type**Yearly\_Pay**is array** (Month) of Integer;

**type**Summer\_Pay**is array** (Month range Jul. .Sep) of Integer;

Such array types are also said to be constrained. Constraints, however, do not need to be statically defined; that is, Ada supports dynamic arrays as we shownext.

First, one can declare unconstrained array types by stating an unspecified range, indicated by the symbol <> (box):

**type**Some\_Period\_Pay**is array** (Month **range** <>) **of** Integer;

**type**Int\_Vector**is array** (Integer **range** <>) **of** Integer;

**type**Bool\_Matrix**is array** (Integer **range** <>, Integer **range** <>) **of** Boolean;

In Ada, array types are characterized by the types of the components, the number of indices, and the type of each index; the values of the bounds are not considered to be a part of the array type, and thus may be left unspecified at compile-time.

The values of the bounds, however, must become known

when an object is created. For example, one can declare the following variables:

Spring\_Salary: Some\_Period\_Pay (Apr. .Jun);

Z: Int\_Vector (-100. .100);

W: Int\_Vector (20. .40);

Y: Bool\_Matrix (0. .N, 0. .M);

Notice that the values of the bounds need not be given by a static expression.

It is only required that the bounds be known at run time when the object declarations are processed.

An interesting way of instantiating the bounds of an array is by parameter passing. For example, the following function receives an object of type

Int\_Vector and sums its components:

**function**Sum (X: Int\_Vector) **return** Integer;

Result: Integer := 0; --declaration with initialization

**begin**

**for**I **in** X’First. .X’Last**loop**

--attributes First and Last provide the lower and upper bounds of the index

Result := Result + X (I);

**end loop**;

**return**Result;

**end**Sum;

The function can thus be called with array parameters of different sizes; for

example

A := Sum (Z) + Sum (W);

Ada views *strings* as arrays of characters of the following predefined type:

**type**String **is array** (Positive **range** <>) **of** characters;

A line of 80 characters, initialized with all blanks, can be declared as follows:

Line: String (1. .80) := (1. .80 => ’ ’);

Similar to Pascal, Ada records can support both Cartesian products and (discriminated)

unions. An example of a *Cartesian product* is

**type**Int\_Char**is**

X: Integer **range** 0. .100;

Y: Character;

**end record**;

Ada provides a safe version of *discriminated unions* through variant records.

For example, one may write the following Ada declarations (corresponding to the example discussed in Section 3.2.3)

**type**Address\_Type**is** (Absolute, Offset);

**type**Safe\_Address**is record** (Kind: Address\_Type := Absolute)

**case**Kind **is**

**when**Absolute =>

Abs\_Addr: Natural);

**when**Offset =>

Off\_Addr: Integer;

**end case**;

**end record**;

Type Safe\_Address has a discriminant Kind that defines the possible variants of an address. The default initial value of the discriminant is declared in the example to be Absolute. Thus an object declared as

X: Safe\_Address;

is an absolute address by default. The discriminant of a variable initialized bydefault can be changed only by assignment of the record as a whole, not by assignment to the discriminant alone. This forbids the producing of inconsistent data objects, and makes variant records a safe representation of discriminated unions.

The discriminant of a variable can also be initialized explicitly when a variable is declared, as in the following case:

Y: Address (Offset);

In such a case, the variant for the object is frozen, and cannot be changed later. The compiler can reserve the exact amount of space required by the variant for the constrained variable. The following assignments

X := Y;

X := (Kind => Offset, Off\_Addr => 10);

are legal and change the variant of variable X to Offset. The following assignment, which would change the variant for Y, is illegal

Y := X;

Access to the variant of an object whose discriminant is initialized by default, such as

X.Off\_Addr := X.Off\_Addr + 10;

requires a run-time check to verify that the object is accessed correctly according to its current variant. In the example, if the address is not an offset, the error exception Constraint\_Error is raised.

*Access types* (*pointers*) are used mainly to allocate and deallocate data dynamically. As an example, the following declarations define a binary tree:

**type**Bin\_Tree\_Node; --incomplete type declaration

**type**Tree\_Ref**is access** Bin\_Tree\_Node;

**type**Bin\_Tree\_Node**is**

**record**

Info: Character;

Left, Right: Tree\_Ref:

**end**;

(Note that an incomplete type declaration is needed when recursive types are being defined.)

If P and Q are two pointers of type Tree\_Ref, the Info component of the node referenced by P is P.Info. The node itself is P.all. Thus, assignment of the node pointed by P to the node pointed by Q is written as

Q.all := P.all;

If T is a pointer of type TREE\_REF, allocation of a new node pointed by T can be accomplished as follows:

T :=**new** Bin\_Tree\_Node;

The following assignment

T.all := (Info => 0, Left => null, Right => null);

initializes T to point to a node whose Left and Right pointers are null, i.e., they do not refer to any entity. The language defined value null denotes a null pointer value.

Ada also allows pointers to refer to routines. For example, the following declaration defines type Message\_Routine to be any procedure with an input parameter of type String

**type**Message\_Routine**is access procedure** (M: String);

If Print\_This is a previously defined procedure with an input parameter M of type String, one can write

Give\_Message: Message\_Routine; --declares a pointer to a routine

. . .

Give\_Message := Print\_This’Access; --access yields a reference to the routine

. . .

Give\_Message.all (’This Is A Serious Error’);

--invokes Print\_This; ".all" (dereferencing) is optional

Finally, it is also possible in Ada to define pointers which refer to named objects, fields of records, or array elements. Such referenceable objects (or parts of an object) must be declared as aliased (dynamically allocated data are aliased).

As the name indicates, such elements are accessible via several possible names (aliases). If an element is declared as aliased, the attribute Access can be applied to provide a pointer to the element.

This is avoided in Ada by run-time checking that an object referenced by a pointer is allocated in an activation record that is more recently allocated than the activation record of the unit in which the access type is declared.

This check ensures that the object will live at least as long as the access type, which in turn ensures that the access values cannot refer to objects that do not exist.

The Ada type system is largely based on Pascal, but is richer, more complex, and more systematic. It also solves several of the problems and insecurities of the original Pascal definition. As a result, the language gets close to the goal of strong typing.

Type compatibility (based on name compatibility) is explicitly specified. The notion of subtype and the necessary run-time checks are precisely defined. If a new type is to be defined from an existing type, a derived type construct is provided. For example

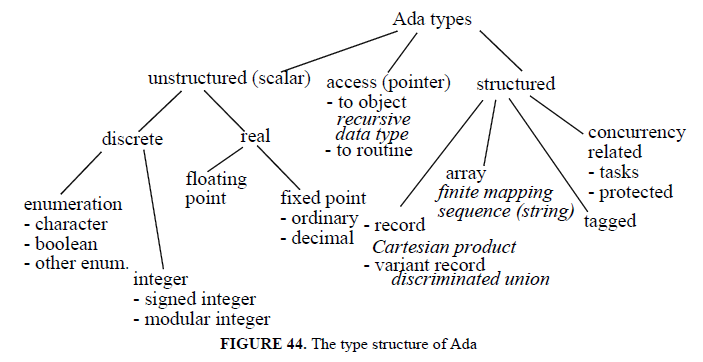
**type**Celsius **is new** Integer;

**type**Farenheit**is new** Integer;

define two new type; Integer is their parent type. New types inherit all properties (values, operations, and attributes) from their parent type, but they are considered as different types.

Overloading and coercion are widely available in Ada. The language also provides for inclusion polymorphism in the case of subtypes and type extensions.

Finally, Ada makes extensive use of *attributes*. Attributes are used to designate properties of data objects and types. As we saw in many examples so far, the value of an attribute is retrieved by writing the name of the entity whose attribute is being sought, followed by a ’ and the name of the attribute. Ada predefines a large number of attributes; more can be defined by an implementation.



**3.5 Implementation models**

This section reviews the basic implementation models for data objects. The description is language independent. It is intended to complement the conceptual model of programming language processing provided in Chapter 2, by showing how data can be represented and manipulated in a machine.

It is notintended, however, to provide a detailed account of efficient techniques forrepresenting data objects within a computer, which can be highly dependent on the hardware structure. Rather, straightforward solutions will be presented, along with some comments on alternative, more efficient representations.

Following Chapter 2, data will be represented by a pair consisting of a

*descriptor*and a *data object*. Descriptors contain the most relevant attributes that are needed during the translation process. Additional attributes might be appropriate for specific purposes.

Typically, descriptors are kept in a symbol table during translation, and only a subset of the attributes stored there needs to be saved at run time. Again, we will pay little attention to the physical layout

of descriptors, which depends on the overall organization of the symbol table.

**3.5.1 Built-in and enumerations**

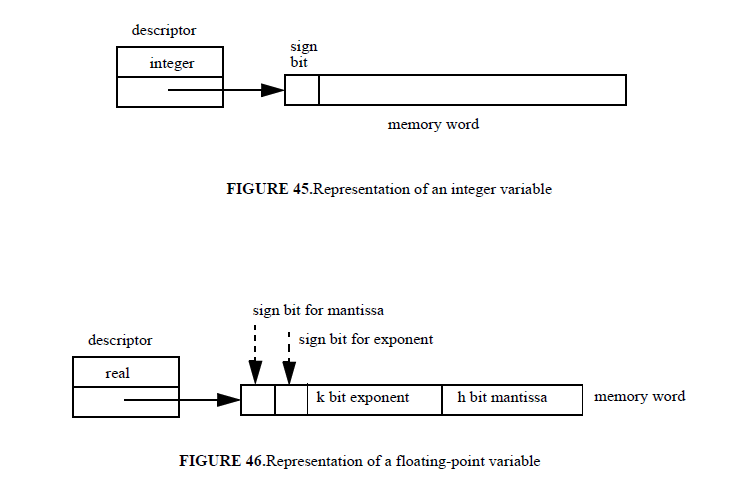
Integers and reals are hardware-supported on most conventional computers, which provide fixed and floating-point arithmetic. Existing computers may also provide different lengths for the supported arithmetic types.

In a language like C, these are reflected by long and short prefixes. If the language provides different lengths for arithmetic types and the underlying hardware does not, an implementation is usually free to ignore the prefixes.

In the case of C, it is only required that short should not be longer than int, which should not be longer than long. If we ignore the issue of different lengths for arithmetic types, for simplicity, integer and real variables may be represented as shown in Figure 45 and Figure 46.

Values of an enumeration type ENUM can be mapped, one-to-one, to integer values in the range 0. .n-1, n being the cardinality of ENUM. This mapping does not introduce any possibility of mixing up values of type ENUM with values of any other enumeration type, if all run-time accesses are routed via a descriptor containing the type information.

The use of the descriptor is of course not necessary for typed languages. Note that, in a language like C, the mapping of enumeration types to integers is not just part of the implementation of the type (and as such, invisible to the programmer), but it is explicitly stated in the language definition. This allows the programmer to take advantage of this knowledge, and find ways to break the protection shield provided by the type to access the representation directly.



Booleans and characters can be viewed as enumeration types, and implemented as above. To save space, characters and booleans can be stored instorage units smaller than a word (e.g., bits or bytes), which might be addressable by the hardware, or may be packaged into a single word and then retrieved by suitable word manipulation instructions that can disassemble the contents of a word. In such a case, accessing individual characters of Booleans may be less efficient, but it would save memory space. Thus the choice of the implementation model depends on a trade-off between speed and space.

**3.5.2 Structured types**

In this section we review how to represent structured types, built via the constructors discussed in Section 3.2. Our discussion will not be dependent on the specific syntax adopted by an existing language. Rather, it will refer to a hypothetical, self-explaining, programming notation.

For simplicity, we will assume that variables are declared by providing an explicit type name. For example, this means that a declaration of–say– a finite mapping X:

X: float array [0. .10]; is a shorthand for a declaration of a type followed by a declaration of an array variable:

typeX\_type is float array [0. .10];

X: X\_type;

Similarly, we assume that if a type declaration contains a structured component,such component is separately defined by a type declaration.

For example, if a field of a Cartesian product is a finite mapping, there are two type declarations: the declaration of an array type T and the declaration of a structure, with a field of type T. As a consequence of these assumptions, each component of a structured type is either a built-in type, or it is a user-defined type.

As for built-in types, each variable is described by a descriptor and its storage layout. The descriptor contains a description of the variable’s type. In an actual implementation, for efficiency reasons, all variables of a given type might have a simplified descriptor which points to a separately stored descriptor for that type.

**3.5.2.1 Cartesian product**

The standard representation of a data object of a Cartesian product type is a sequential layout of components. The descriptor contains the Cartesian product type name and a set of triples (name of the selector, type of the field, reference to the data object) for each field. If the type of the field is not a built-in type, the type field points to an appropriate descriptor for the field.

Figure 47 illustrates the representation for the following case of a variable of Cartesian product type with a field which is itself of a Cartesian product type:

typeanother\_type Y is struct{

float A;

int B;

};

typesample\_type is struct {

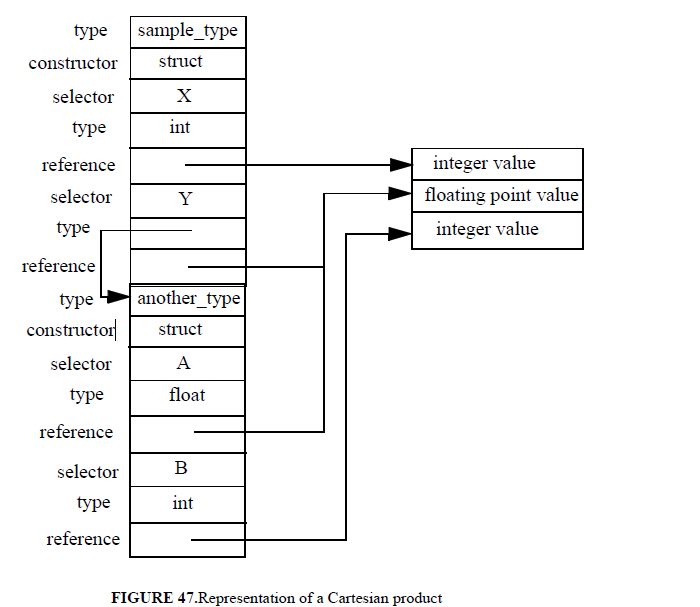
int X;

another\_type Y;

};

sample\_type X;

Note that each component of the Cartesian product occupies a certain number of addressable storage units (e.g., words). In a statically typed language, if each component is guaranteed to occupy a fixed memory size, known by the compiler, the descriptor is not needed at run time, and the reference to each field can be evaluated statically by the compiler as a fixed offset with respect to the initial address of the composite object.



**3.5.2.2 Finite mapping**

A conventional representation of a finite mapping allocates a certain number of storage units (e.g., words) for each component. The descriptor contains the finite-mapping type name; the name of the domain type, with the values of the lower and upper bound; the name of the range type (or the reference to its descriptor); the reference to the first location of the data area where the data object is stored.

For example, given the declarations

typeX\_type is float array [0. .10];

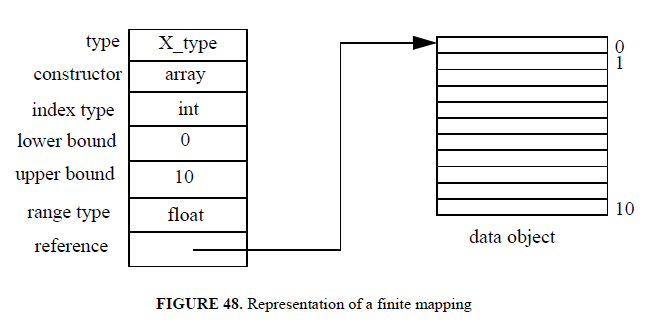
X\_type X;

the corresponding representation is given in Figure 48.

A reference to X[I] is computed as an offset from the address A\_X of the first location of the array. Let the domain type be the integer subrange M. .N. Let K be the number of words occupied by each element of the array (this is known from the type of the range, but might be stored in the descriptor to avoid computing such value each time it is necessary). The offset to be evaluated for accessing A[I] is K(I - M).

In a statically typed language with arrays of statically known index bounds, the descriptor does not need to be saved at run time. The only exceptions are index bounds, which may be used to check at run time that the index value belongs to the stated range. As we discussed in Chapter 2 (Section 2.6.5), in a language that supports dynamic arrays, the value of the array in the activation record is composed of two parts.

The first part (often called *dope vector*) contains a reference to the data object (which, in general, can only be evaluated at run time) and the values of the bounds (to be used for index checking). The second part is the array itself, which is accessed indirectly through the dope vector.



**3.5.2.3 Union and discriminated union**

Union types do not require any special new treatment to be represented. A variable of a union type has a descriptor that is a sequence of the descriptors of the component types. Instances of the values of the component types share the same storage area.

Discriminated union types are provided by existing programming languages as extensions of the Cartesian product. For example Pascal and Ada provide discriminated unions as variant records. The variant record is a construct can be viewed as the conjunction of several fields plus a disjunction of fields, prefixed by the definition of a tag field.

When the conjunction of fields is empty, we obtain a discriminated union. As an example, the reader may consider the following Pascal-like fragment. Since all variants share the same storage area, a variable Z of type Z\_type can be represented as in Figure 49. Note that the various variants are accessible via a case table, according to the value of the tag field.

typeX\_type is float array [0. .10];

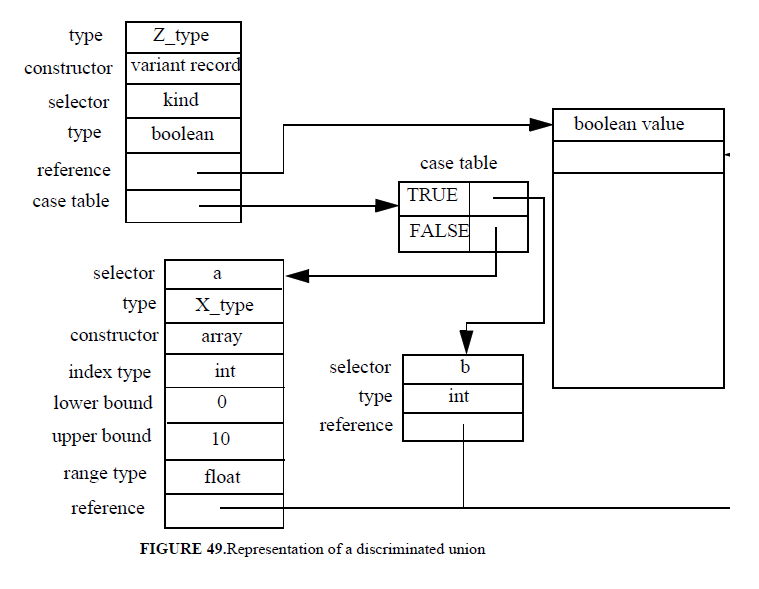
typeZ\_type = record

case kind: BOOLEAN of

TRUE: (a: integer);

FALSE: (b: X\_type)

End



***3.5.2.4 Powersets***

Powersets can be implemented efficiently, in terms of access time, manipulation time, and storage space, provided that a machine word has at least as many bits as there are potential members of the set. (i.e., the cardinality of the base type). The presence of the i-th element of the base type in a certain set S is denoted by a “1” as the value of the i-th bit of the word associated with S.

The empty set is represented by all zeros in the word. The union between sets is easily performed by an OR between the two associated words, and the intersection by an AND. If the machine does not allow bit-level access, test for membership requires shifting the required bit into an accessible word position (e.g., the sign bit), or using a mask.

The existence of such an appealing representation for powersets is responsible for the implementation defined limits for the cardinality of representable sets, which is normally equal to the size of a memory word.

**3.5.2.5 Sequences**

Sequences of elements of arbitrary length on a secondary storage are represented as files. File management is ignored here, being out of scope. Strings, as supported by many languages, are just array of characters. In other languages, such as SNOBOL4 and Algol 68, strings may vary arbitrarily in length, having no programmer-specified upper bound. This kind of array with dynamically changing size must be allocated in the heap.

**3.5.2.6 Classes**

User-defined types specified via the simple class construct introduced in Section 3.2.8 are easy to represent as extensions of structures. The differences are:

1. only public fields are visible outside the construct. Such fields must be tagged as public

in the descriptor.

2. some fields may be routines. It is thus necessary to be able to store routine signatures in

descriptors.

The reader can easily extend the representation scheme presented in Section 3.5.2.1 to keep these new requirements into account. Since the code of the routines encapsulated in a class is shared by all class instances, routine fields are represented as pointers to the routines.

For example, in C++ after class point is declared as in Figure 33, the following declarations are possible in some function f:

point x (1.3, 3.7);

point\* p = new point (1.1, 0.0);

In the first case, x is a named variable that is allocated in f’s activation record on the stack. In the second, p is allocated in f’s activation record, while the data structure for the point is allocated on the heap. Heap management is discussed next.

**3.5.2.7 Pointers and garbage collection**

A pointer holds as a value the address of an object of the type to which the pointer is bound to. Pointers usually can have a special null value (e.g., void in C and C++, nil in Pascal). Such a null value can be represented by an address value that would cause a hardware-generated error trap to catch an inadvertent reference via a pointer with null value. For example, the value might be an address beyond the physical addressing space into a protected area.

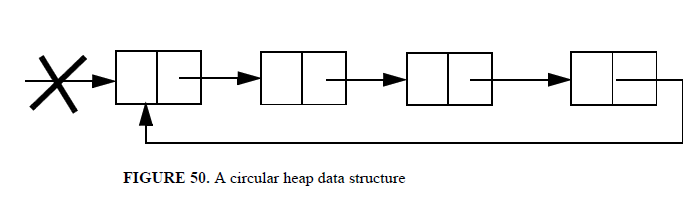
Pointer variables are allocated as any other variable on the activation record stack. Data objects that are allocated via the run-time allocation primitive new are allocated in the heap. Heap management is a crucial aspect of a programming language implementation. In fact, the free storage might quickly be exhausted, unless there is some way of returning allocated storage to the free area.

The memory allocated to hold a heap object can be released for later reuse if the object is no longer accessible. Such an object is said to be *garbage*. Precisely, an object is garbage if no variable in the stack points to it and–recursively– no other garbage points to it. There are two main aspects involved inheap management. One is the recognition that some portion of allocated memory is garbage; the other is to recycle such memory for new memory allocation requests issued by the user.

Garbage recognition often relies on the programmer, who is required to notify the system that a given object became useless. For example, in order to state that the object pointed by p is garbage, in C++ one would write delete(p) and in Pascal one would write dispose(p).

Another strategy is to let the run-time system take care of discovering garbage, by means of a suitable automatic *garbage collection* algorithm. Automatic garbage collection is vital for languages that make heavy use of dynamically generated variables, like LISP. In fact, garbage collection was invented for implementing the first LISP systems. Eiffel, which uniformly treats all objects as referenced by pointers, provides an automatic garbage collector.

This method thusreleases an object as soon as it is found to become unreferenced. The problemwith this method, however, is that it does not work for circular heap datastructures (see Figure 50). If a pointer to the head of a circular list is deallocated,the nodes of the list are not found to be garbage, because the referencecount for each node is one.

**

A non-incremental strategy for automatic garbage collection consists of allocating free cells from the heap until the free space is exhausted. Only at that point the system enters a garbage collecting phase. We describe one such scheme under the simplifying assumption that:

• the heap data objects have fixed size

• it is known a-priori which fields of a heap object contain pointers to other heap data objects, and

• it is possible to find all the pointers from the stack into the heap.

The following method for garbage collection allows all reachable heap data objects to be distinguished from garbage objects. To do so, a working set of pointers T may be used. Initially, T contains the stack values which point into the heap.

An element E is repeatedly extracted from T, the objects referenced

by E are marked, and E is replaced by the pointers to the node(s) contained inE, if they are not marked. When T becomes empty, all reachable heap data objects have been marked. All other objects are garbage.

A number of variations have been proposed in the literature to make this garbage collection method more efficient. Its main problem, however, is that “useful” processing time is lost when the garbage collector is invoked.

In an interactive system, this may be perceived by the programmer as an unexpected slow-down of the application, which occurs at unpredictable times. In a real-time system, this can be particularly dangerous, because an urgent request for service might arrive from the environment just after the garbage collector has started its rather complex activity. Garbage collection time is distributed more uniformly over processing time by using the reference counting scheme, but unfortunately such scheme works only partially.

An appealing solution, which cannot be reviewed here, is based on a parallel execution scheme, where the garbage collector and the normal language processor execute in parallel.

Having discovered which heap data objects are garbage (either by explicit notification by the programmer, or by running a garbage collector), one should decide how to recycle them by adding them to the free storage.One possibility is to link all free areas in a free list. In such a case, each block would contain at least two cells: one for the block size, and one for the pointer to the next block.

It is convenient to keep the list ordered by increasing block address, so that as a block is ready to be added to the list, it can be merged with possible adjacent blocks in the list. As a new storage area is to be allocated, the free list is searched for a matching block., according to some policy.