4 Elementary Data Types

4.1 Integer types

The word *integer* is familiar from mathematics where it denotes the unbounded, ordered sequence of numbers:

$$\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$$

The term is used in programming to denote something quite different: a specific data type. Recall that a data type is a set of values and a set of operations on those values. Let us begin by defining the set of values of the Integer data type.

Given a word of memory, we can simply define a set of values by the usual interpretation of the bits of the word as a binary value. For example, if a word with 8 bits contains the sequence 10100011, it is interpreted as:

$$(1 \times 2^7) + (1 \times 2^5) + (1 \times 2^1) + (1 \times 2^0) = 128 + 32 + 2 + 1 = 163$$

The range of possible values is 0..255, or in general $0..2^B - 1$ for a word of B bits. A data type with this set of values is called an *unsigned integer* and a variable of this type can be declared in C as:

```
unsigned int v;
```
C

Note that the number of bits in a value of this type varies from one computer to another.[1] Today the most common word size is 32 bits and an (unsigned) integer value is in the range $0..2^{32} - 1 \approx 4 \times 10^9$. Thus, the set of mathematical integers is unbounded while integer types are limited to a finite range of values.

Since they cannot represent negative numbers, unsigned integers are mostly useful as representations of values read from external equipment. For example, a temperature probe may return 10 bits of information; the unsigned value in the range 0..1023 must then be translated into ordinary (positive and negative) numbers. Unsigned integers are also used to represent characters (see below). Unsigned integers should not be used for ordinary computation, because most computer instructions work on signed integers and a compiler may emit extra instructions for unsigned values.

The range of values that a variable can have may not fit into a single word, or it may fit into a partial word. Length specifiers can be added to indicate different integer types:

[1] In fact, unsigned int need not be the same size as a word of memory.

C

```
unsigned int v1;
unsigned short int v2;
unsigned long int v3;
```

In Ada, additional types such as Long_Integer are predefined along with ordinary Integer. The actual interpretation of length specifiers like long and short varies from one compiler to another; some compilers may even give the same interpretation to two or more specifiers.

The representation of signed numbers in mathematics uses a special character "–" followed by the usual representation of the absolute value of the number. This representation is not convenient for computer instructions to work with. Instead most computers represent signed integers in a notation called *two's complement*. A positive number is represented by a leading zero bit followed by the usual binary representation of the value. It follows that the largest positive integer that can be represented in a word with w bits is only $2^{w-1}-1$.

To obtain the representation of $-n$ from the binary representation $B = b_1 b_2 \ldots b_w$ of n:

- Take the logical *complement* of B, that is, change each b_i from zero to one or from one to zero.
- Add one.

For example, the representations of -1, -2 and -127 as 8-bit words are:

$$\begin{array}{rcccccccc} 1 & = & 00000001 & \rightarrow & 11111110 & \rightarrow & 11111111 & = & -1 \\ 2 & = & 00000010 & \rightarrow & 11111101 & \rightarrow & 11111110 & = & -2 \\ 127 & = & 01111111 & \rightarrow & 10000000 & \rightarrow & 10000001 & = & -127 \end{array}$$

A negative value will always have a one in the high-order bit.

Two's complement is convenient because if you do ordinary binary arithmetic on values, the result is correct:

$$\begin{array}{rcl} (-1)-1 & = & -2 \\ 11111111-00000001 & = & 11111110 \end{array}$$

Note that the bit string 10000000 cannot be obtained from any positive value. It represents -128, even though its positive counterpart 128 cannot be represented as an 8-bit two's complement number. This asymmetry in the range of integer types must be kept in mind, especially when dealing with short types.

An alternative representation of signed numbers is *one's complement*, in which the representation of $-n$ is just the complement of n. The set of values is then symmetric, but there are two representations of zero: 00000000 called positive zero, and 11111111 called negative zero.

If you do not use a special syntax like unsigned in a declaration, the default is to use signed integers:

```
int i;                  /* Signed integer in C */
I: Integer;             -- Signed integer in Ada
```

Integer operations

Integer operations include the four basic operations: addition, subtraction, multiplication and division. They can be used to form *expressions*:

```
a + b / c - 25 * (d - e)
```

Normal mathematical rules of precedence apply to integer operations; parentheses can be used to change the order of computation.

The result of an operation on a signed integer must not be outside the range of values; otherwise, overflow will occur as discussed below. For unsigned integers, arithmetic is cyclic. If short int is stored in one 16-bit word, then:

C

```
unsigned short int i;      /* Range of i = 0..65535 */
i = 65535;                 /* Largest possible value */
i = i + 1;                 /* Cyclic arithmetic, i = 0 */
```

The designers of Ada 83 made the mistake of not including unsigned integers. Ada 95 generalizes the concept of unsigned integers to *modular types*, which are integer types with cyclic arithmetic on an arbitrary modulus. A standard unsigned byte can be declared:

Ada

```
type Unsigned_Byte is mod 256;
```

while a modulus that is not a power of two can be used for hash tables or random numbers:

Ada

```
type Random_Integer is mod 41;
```

Note that modular types in Ada are portable since only the cyclic range is part of the definition, not the *size* of the representation as in C.

Division

In mathematics, division of two integers a/b produces two values, a *quotient* q and a *remainder* r such that:

$$a = q * b + r$$

Since arithmetical expressions in programs return a single value, the "/" operator is used to return the quotient, and a distinct operator (called "%" in C and rem in Ada) returns the remainder. The expression $54/10$ yields the value 5, and we say that the result of the operation has been *truncated*. Pascal uses a separate operator div for integer division.

The definition of integer division is not trivial when we consider negative numbers. Is the expression $-54/10$ equal to -5 or -6? In other words, is truncation done to smaller (more negative)

values or to values closer to zero? One choice is to truncate towards zero, since the equation for integer division can be maintained just by changing the sign of the remainder:

$$-54 = -5 * 10 + (-4)$$

There is another mathematical operation called *modulo* which corresponds to truncation of negative numbers to smaller (more negative) values:

$$\begin{aligned} -54 &= -6 * 10 + 6 \\ -54 \bmod 10 &= 6 \end{aligned}$$

Modulo arithmetic has applications whenever arithmetic is done on finite ranges, such as error-correcting codes used in communications systems.

The meaning of "/" and "%" in C is implementation dependent, so programs using these integer operators may not be portable. In Ada, "/" always truncates towards zero. The operator rem returns the remainder corresponding to truncation towards zero, while the operator mod returns the remainder corresponding to truncation towards minus infinity.

Overflow

An operation is said to *overflow* if it produces a result that is not within the range of defined values. The following discussion is given in terms of 8-bit integers for clarity.

Suppose that the signed integer variable i has the value 127 and that we increment i. The computer will simply add one to the integer representation of 127:

$$01111111 + 00000001 = 10000000$$

to obtain -128. This is not the correct result and the error is called overflow.

Overflow may cause strange bugs:

C

```
for (i = 0; i ¡ j*k; i++) ...
```

If the expression j*k overflows, the upper bound may be negative and the loop will not be executed.

Suppose now that the variables a, b and c have the values 90, 60 and 80, respectively. The expression (a–b+c) evaluates to 110, because (a–b) evaluates to 30, and then the addition gives 110. However, the optimizer may choose to evaluate the expression in a different order, (a+c–b), giving an incorrect answer because the addition (a+c) gives the value 170 which overflows. If your high-school teacher told you that addition is commutative and associative, he/she was talking about mathematics and not about programming!

Some compilers have the option of checking every integer operation for overflow, but this may be prohibitive in terms of run-time overhead unless the detection of overflow is done by hardware. Now that most computers use 32-bit memory words, integer overflow is rarely a problem, but you must be aware of it and take care not to fall into the traps demonstrated above.

Implementation

Integer values are stored directly in memory words. Some computers have instructions to compute with partial words or even individual bytes. Compilers for these computers will usually map short int into partial words, while compilers for computers that recognize only full words will implement all integer types identically. long int will be mapped into two words to achieve a larger range of values.

Addition and subtraction are compiled directly into the corresponding instructions. Multiplication is also implemented as a single instruction but requires significantly more processing time than addition and subtraction. Multiplication of two words stored in registers R1 and R2 produces a result that is two words long and will require two registers to store. If the register containing the high-order value is not zero, the result has overflowed.

Division requires the computer to perform an iterative algorithm similar to "long division" done by hand. This is done in hardware and you need not concern yourself with the details, but if efficiency is important it is best to avoid division.

Arithmetic on long int takes more than twice the amount of time as it does on int. The reason is that an instruction must be used to transfer a possible "carry" from the low-order words to the high-order words.

4.2 Enumeration types

Programming languages such as Fortran and C define data in terms of the underlying computer. Real-world data must be explicitly mapped onto data types that exist on the computer, in most cases one of the integer types. For example, if you are writing a program to control a heater you might have a variable dial that stores the current position of the dial. Suppose that the real-world dial has four positions: *off*, *low*, *medium*, *high*. How would you declare the variable and denote the positions? Since the computer does not have instructions that work on four-valued memory words, you will choose to declare the variable as an integer, and you will choose four specific integers (say 1, 2, 3, 4) to denote the positions:

C

```
int dial;                    /* Current position of dial */
if (dial ¡ 4) dial++;        /* Increment heater level*/
```

The obvious problem with using integers is that the program becomes hard to read and maintain. You must write extensive documentation and continually refer to it in order to understand the code. The first improvement that can be made is to document the intended values internally:

C

```
#define  Off      1
#define  Low      2
#define  Medium   3
#define  High     4
```

```
int dial;
if (dial ¡ High) dial++;
```

Improving the documentation, however, does nothing to prevent the following problems:

C

```
dial = −1;              /* No such value */
dial = High + 1;        /* Undetected overflow */
dial = dial * 3;        /* Meaningless operation */
```

To be precise, representing the four-position dial as an integer allows the programmer to assign values that are not within the intended range, and to execute operations that are meaningless for the real-world object. Even if the programmer does not intentionally create any of these problems, experience has shown that they frequently arise from misunderstandings among software team members, typographical errors and other mistakes typical in the development of complex systems.

The solution is to allow the program designer to create *new* types that correspond exactly to the real-world objects being modeled. The requirement discussed here—a short ordered sequence of values—is so common that modern programming languages support creation of types called *enumeration types*. In Ada, the above example would be:

Ada

```
type Heat is (Off, Low, Medium, High);
Dial: Heat;

Dial := Low;
if Dial ¡ High then Dial := Heat'Succ(Dial);

Dial := −1;                     -- Error
Dial := Heat'Succ(High);        -- Error
Dial := Dial * 3;               -- Error
```

Before explaining the example in detail, note that C has a construct that is superficially the same:

C

```
typedef enum {Off, Low, Medium, High} Heat;
```

However, variables declared to be of type Heat are still *integers* and none of the above statements are considered to be errors (though a compiler may issue a warning):

C

```
Heat dial;

dial = −1;              /* Not an error! */
dial = High + 1;        /* Not an error! */
dial = dial * 3;        /* Not an error! */
```

In other words, the enum construct is just a means of documentation that is more convenient than a long series of define's, but it does not create a new type.

Fortunately, C++ takes a stricter interpretation of enumeration types and does not allow assignment of an integer value to a variable of an enumeration type; the above three statements are in error. However, values of enumeration types can be implicitly converted to integers so the type checking is not complete. Unfortunately, C++ provides no operations on enumeration types so there is no predefined way to increment a variable of enumeration type. You can write your own function that takes the result of an integer expression and then explicitly converts it back to the enumeration type:

C++

```
dial = (Heat) (dial + 1);
```

Note the implicit conversion of dial to integer, followed by the explicit conversion of the result back to Heat. The "++" and "−−" operators can be overloaded in C++ (Section 10.2), so they can be used for defining operations on enumeration types that are syntactically the same as those of integer types.

In Ada, the type definition creates a new type Heat. The values of this type are *not* integers. Any attempt to exceed the range or to use integer operations will be detected as an error. If your finger slips and you enter Higj instead of High, an error will be detected, because the type contains exactly the four values that were declared; if you had used integers, 5 would be as valid an integer as 4.

Enumeration types are types just like integers: you can declare variables and parameters to be of these types. However, they are restricted in the set of operations that can be performed on values of the type. These include assignment (:=), equality (=) and inequality (/=). Since the set of values in the declaration is interpreted as an ordered sequence, the relational operators (<,>,>=,<=) are defined.

Given an enumeration type T and a value V of type T, the following functions called *attributes* are defined in Ada:

- T'First returns the first value of T.
- T'Last returns the last value of T.
- T'Succ(V) returns the successor of V.
- T'Pred(V) returns the predecessor of V.
- T'Pos(V) returns the position[2] of V within the list of values of T.
- T'Val(I) returns the value at the I'th position in T.

Attributes make the program robust to modifications: if values are added to an enumeration type, or if the values are rearranged, loops and indices remain unaltered:

[2]The position of a value is defined by its position (starting from zero) in the declaration of the type.

Ada

```
for I in Heat'First .. Heat'Last – 1 loop
   A(I) := A(Heat'Succ(I));
end loop;
```

Not every language designer believes in enumeration types. Eiffel does not contain enumeration types for the following reasons:

- A desire to keep the language as small as possible.
- The same level of safety can be obtained using assertions (Section 11.5).
- Enumeration types are often used with variant records (Section 10.5); when inheritance (Section 14.3) is used correctly there is less need for enumeration types.

Enumeration types should be used whenever possible in preference to ordinary integers with lists of defined constants; their contribution to the reliability of a program cannot be overestimated. C programmers do not have the benefit of type-checking as in Ada and C++, but they should still use enum for its significant contribution to readability.

Implementation

I will let you in on a secret and tell you that the values of an enumeration type are represented in the computer as consecutive integers starting from zero. The Ada type checking is purely compile-time, and operations such as "¡" are ordinary integer operations.

It is possible to request the compiler to use a non-standard representation of enumeration types. In C this is specified directly in the type definition:

C

```
typedef enum {Off=1, Low=2, Medium=4, High=8} Heat;
```

while in Ada a representation specification is used:

Ada

```
type Heat is (Off, Low, Medium, High);
for Heat use (Off => 1, Low => 2, Medium => 4, High => 8);
```

4.3 Character type

Though computers were originally invented in order to do numerical calculations, it quickly became apparent that non-numerical applications are just as important. Today non-numerical applications, such as word-processors, educational programs and databases, probably outnumber mathematical applications. Even mathematical applications such as financial software need text for input and output.

From the point of view of a software developer, text processing is extremely complicated because of the variety of natural languages and writing systems. From the point of view of programming languages text processing is relatively simple, because the language assumes that the set of characters forms a short, ordered sequence of values, that is, characters can be defined as an enumeration type. In fact, except for languages like Chinese and Japanese that use thousands of symbols, the 128 or 256 possible values of a signed or unsigned 8-bit integer suffice.

The difference in the way that Ada and C define characters is similar to the difference in the way that they define enumeration types. In Ada, there is a predefined enumeration type:

Ada

```
type Character is (..., 'A', 'B', ...);
```

and all the usual operations on enumeration types (assignment, relations, successor, predecessor, etc.) apply to characters. In Ada 83, Character is limited to the 128 values of the American standard ASCII, while in Ada 95, the type is assumed to be represented in an unsigned byte, so the 256 values required by international standards are available.

In C, char is just a very small integer type and all the following statements are accepted since char and int are essentially the same:

C

```
char c;
int i;
c = 'A' + 10;          /* Convert char to int and back */
i = 'A';               /* Convert char to int */
c = i;                 /* Convert int to char */
```

char is a distinct type in C++, but is convertible to and from integers so the above statements remain valid.

For non-alphabetic languages, 16-bit characters can be defined. These are called wchar_t in C and C++, and Wide_Character in Ada 95.

The only thing that distinguishes characters from ordinary enumeration or integer types is the special syntax ('A') for the set of values, and more importantly the special syntax that exists for arrays of characters called *strings* (Section 5.6).

4.4 Boolean type

Boolean is a predefined enumeration type in Ada:

Ada

```
type Boolean is (False, True);
```

The Boolean type is extremely important because:

- Relational operators (=, >, etc.) are functions which return a value of Boolean type.

- The if-statement takes an expression of Boolean type.
- Operators for Boolean algebra (and, or, not, xor) are defined on the Boolean type.

C does not define a separate type; instead integers are used with the following interpretation:

- Relational operators return 1 if successful and 0 if not.
- The if-statement takes the false branch if the integer expression evaluates to zero, and the true branch otherwise.

There are several methods for introducing Boolean types into C. One possibility is to define a type which will allow declaration of functions with Boolean result:

C

```
typedef enum {false, true} bool;

bool data_valid(int a, float b);

if (data_valid(x, y)) ...
```

but of course this is purely for documentation and readability, because a statement like:

C

```
bool b;
b = b + 56;              /* Add 56 to "true" ?? */
```

is still accepted and can cause obscure bugs.

In C++, bool is predefined *integer* type (not an enumeration type), with implicit conversion between non-zero values and a literal true, and between zero values and false. A C program with bool defined as shown above can be compiled in C++ simply by removing the typedef.

Even in C it is better not to use the implicit conversion of integer to Boolean, but rather to use explicit equality and inequality operators:

C

```
if (a+b == 2) ...        /* This version is clearer than */
if (!(a+b-2)) ...        /* ... this version. */
if (a+b != 0) ...        /* This version is clearer than */
if (a+b) ...             /* ... this version. */
```

Finally, C supports Boolean algebra using short-circuit operators which we will discuss in Section 6.2.

4.5 * Subtypes

In the previous sections we have discussed integer types which allow computations to be performed on the large range of values that can be represented in a memory word, and enumeration

types which work with smaller ranges but do not allow arithmetic computation to be done. However, in many cases we would like to do computations on smaller ranges of integers. For example, there should be some way of detecting errors such as:

```
Temperature: Integer;
Temperature := −280;      -- Below absolute zero!

Compass_Heading: Integer;
Compass_Heading := 365;  -- Compass shows 0..359 degrees!
```

Suppose we try to define a new class of types:

```
type Temperatures is Integer range −273 .. 10000; Not Ada!
type Headings is Integer range 0 .. 359;          -- Not Ada!
```

This will solve the problem of checking errors caused by values outside the range of the type, but the question remains: are these two different types or not? If they are, then:

```
Temperature * Compass_Heading
```

is a valid arithmetic expression on an integer type; if not, type conversion must be used.

In practice, both of these interpretations are useful. Computation concerning the physical world tends to involve calculations among values of many ranges. On the other hand, indices into tables or serial numbers do not require computation between types: it makes sense to ask for the next index into a table, but not to add a table index to a serial number. Ada provides two separate facilities for these two classes of type definition.

A *subtype* is a restriction on an existing type. Discrete (integer and enumeration) types can have a *range constraint*:

```
subtype Temperatures is Integer range −273 .. 10000;
Temperature: Temperatures;

subtype Headings is Integer range 0 .. 359;
Compass_Heading: Headings;
```

The type of a value of a subtype S is still that of the underlying *base* type T; here, the base type of both Temperatures and Headings is Integer. The type is determined at *compile-time*. The value of a subtype has the same representation as that of the base type and is acceptable wherever a value of the base type is needed:

```
Temperature * Compass_Heading
```

is a valid expression, but the statements:

```
Temperature := −280;
Compass_Heading := 365;
```

cause errors because the values are not within the ranges of the subtypes. Violations of the range of a subtype are checked at *run-time*.

Subtypes can be defined on any type that can have its set of values restricted in a meaningful way:

```
subtype Upper_Case is Character range 'A' .. 'Z';
U: Upper_Case;
C: Character;

U := 'a';                -- Error, out of range
C := U;                  -- Always OK
U := C;                  -- May cause an error
```

Subtypes are essential in defining arrays as will be discussed in Section 5.5. In addition, a named subtype can be used to simplify many statements:

```
if C in Upper_Case then ...        -- Range check
for C1 in Upper_Case loop ...      -- Loop bounds
```

4.6 * Derived types

The second interpretation of the relation between two similar types is that they represent separate types that cannot be used together. In Ada, such types are called *derived* types and are indicated by the word new in the definition:

```
type Derived_Character is new Character;
C: Character;
D: Derived_Character;

C := D;                  -- Error, types are different
```

When one type is derived from another type (called its *parent* type), it inherits a *copy* of the set of values and a *copy* of the set of operations, but the types remain distinct. However, it is always possible to explicitly convert between types that are derived from one another:

```
D := Derived_Character(C);         -- Type conversion
C := Character(D);                 -- Type conversion
```

It is even possible to specify a different representation for the derived types; type conversion will then convert between the two representations (Section 5.9).

A derived type may include a restriction on the range of values of the parent type:

```
type Upper_Case is new Character range 'A' .. 'Z';
U: Upper_Case;
```

```
C: Character;

C := Character(U);        -- Always valid
U := Upper_Case(C);       -- May cause an error
```

Ada 83 derived types implement a weak version of inheritance, which is a central concept of object-oriented languages (see Chapter 14). The revised language Ada 95 implements true inheritance by extending the concept of derived types; we will return to study them in great detail.

Integer types

Suppose that we have defined the following type:

```
type Altitudes is new Integer range 0 .. 60000;
```

This definition works correctly when we program a flight simulation on a 32-bit workstation. What happens when we transfer the program to a 16-bit controller in our aircraft avionics? Sixteen bits can represent signed integers only up to the value 32,767. Thus the derived type would be in error (as would a subtype or a direct use of Integer), violating the portability of programs which is a central goal of the Ada language.

To solve this problem, it is possible to derive an *integer* type without explicitly specifying the underlying parent type:

```
type Altitudes is range 0 .. 60000;
```

The compiler is required to choose a representation that is appropriate for the requested range—Integer on a 32-bit computer and Long_Integer on a 16-bit computer. This unique feature makes it easy to write numeric programs in Ada that are portable between computers with different word lengths.

The drawback of integer types is that each definition creates a new type and we can't write calculations that use different types without type conversions:

```
I: Integer;
A: Altitude;

A := I;                   -- Error, different types
A := Altitude(I);         -- OK, type conversion
```

Thus there is an unavoidable tension between:

- Subtypes are potentially unsafe because mixed expressions may be written and portability is hard to achieve.

- Derived types are safe and portable, but may make a program difficult to read because of extensive type conversions.

4.7 Expressions

An expression may be as simple as a literal (24, 'x', True) or a variable, or it may be a complex composition involving operations (including calls to system- or user-defined functions). When an expression is *evaluated* it produces a *value*.

Expressions occur in many places in a program: assignment statements, Boolean expressions in if-statements, limits of for-loops, parameters of procedures, etc. We first discuss the expression by itself and then the assignment statement.

The value of a literal is what it denotes; for example, the value of 24 is the integer represented by the bit string 0001 1000. The value of a variable V is the contents of the memory cell it denotes. Note the potential for confusion in the statement:

```
V1 := V2;
```

V2 is an *expression* whose value is the contents of a certain memory cell. V1 is the *address* of a memory cell into which the value of V2 will be placed.

More complex expressions are created from a function or an operator and a set of parameters or operands. The distinction is mostly syntactical: a function with parameters is written in prefix notation sin(x), while an operator with operands is written in infix notation a+b. Since the operands themselves can be expressions, expressions of arbitrary complexity can be created:

```
a + sin(b) * ((c - d) / (e + 34))
```

When written in prefix notation, the order of evaluation is clearly defined except for the order of evaluation of the parameters of a single function:

```
max(sin(cos(x)), cos(sin(y)))
```

It is possible to write programs whose result depends on the order of evaluation of the parameters of a function (see Section 7.3), but such order dependencies should be avoided at all costs, since they are a source of obscure bugs when porting or even when modifying a program.

Infix notation brings its own problems, namely those of precedence and associativity. Almost all programming languages adhere to the mathematical standard of giving higher precedence to the multiplying operators ("*", "/") over the adding operators ("+", "−"), and other operators will have arbitrary precedences defined by the language. Two extremes are APL which does not define *any* precedences (not even for arithmetic operators), and C which defines 15 levels of precedence! Part of the difficulty of learning a programming language is to get used to the style that results from the precedence rules.

An example of a non-intuitive precedence assignment occurs in Pascal. The Boolean operator and is considered to be a multiplying operator with high precedence, whereas in most other languages such as C it is given a precedence below the relational operators. The following statement:

```
if a > b and b > c then ...
```
Pascal

is in error, because the expression is interpreted to be:

```
if a > (b and b) > c then ...
```
Pascal

and the syntax is not correct.

The meaning of an infix expression also depends upon the *associativity* of the operators, namely how operators of identical precedence are grouped: from left to right, or from right to left. In most cases, but not all, it does not matter (except for the possibility of overflow as discussed in Section 4.1). However, the value of an expression involving integer division can depend on associativity because of truncation:

```
int i = 6, j = 7, k = 3;
i = i * j / k;                  /* Is the result 12 or 14 ? */
```
C

In general, binary operators associate to the left so the above example is compiled as:

```
i = (i * j) / k;
```
C

while unary operators associate to the right: in C !++i is computed as if written !(++i).

All precedence and associativity problems can be easily solved using parentheses; they cost nothing, so use them whenever there is the slightest possibility of confusion.

While precedence and associativity are specified by the language, evaluation order is usually left to the implementation to enable optimizations to be done. For example, in the following expression:

```
(a + b) + c + (d + e)
```

it is not defined if a+b is computed before or after d+e, though c will be added to the *result* of a+b before the *result* of d+e. The order can be significant if an expression uses *side-effects*, that is, if the evaluation of a sub-expression calls a function that modifies a global variable.

Implementation

The implementation of an expression depends of course on the implementation of the operators used in the expression. Nevertheless, certain general principles are worth discussing.

Expressions are evaluated from the inside out; for example a*(b+c) is evaluated:[3]

[3]Note this order is not the only possible one; the optimizer might move the load of a to an earlier position or exchange the loads of b and c, without affecting the result of the computation.

```
load    R1,b
load    R2,c
add     R1,R2       Add b and c, result to R1
load    R2,a
mult    R1,R2       Multiply a and b+c, result to R1
```

It is possible to write an expression in a form which makes this evaluation order explicit:

```
b c + a *
```

Reading from left to right: the name of an operand means load the operand, and the symbol for an operator means apply the operator to the two most recent operands and replace all three (two operands and the operator) with the result. In this case, addition is applied to b and c; then multiplication is applied to the result and to a.

This form, called *reverse polish notation*[4] (*RPN*), may be used by the compiler: the expression is translated into RPN, and then the compiler emits instructions for each operand or operator as the RPN is read from left to right.

If the expression were more complex, say:

```
(a + b) * (c + d) * (e + f)
```

more registers would be needed to store the intermediate results: a+b, c+d, and so on. As the complexity increases, the number of registers will not suffice, and the compiler must allocate anonymous temporary variables in order to store the intermediate results. In terms of efficiency: up to a certain point, it is more efficient to increase the complexity of an expression rather than to use a sequence of assignment statements, so that unnecessary storing of intermediate results to memory can be avoided. However, the improvement quickly ceases because of the need to allocate temporary variables, and at some point the compiler may not even be able to handle the complex expression.

An optimizing compiler will be able to identify that the subexpression a+b in the following expression:

```
(a + b) * c + d * (a + b)
```

need only be evaluated once, but it is doubtful if a common subexpression would be identified in:

```
(a + b) * c + d * (b + a)
```

If the common subexpression is complex, it may be helpful to explicitly assign it to a variable rather than trust the optimizer.

Constant folding is another optimization. In the expression:

[4]Named for the Polish logician Lukasiewicz who was interested in a parenthesis-free logical calculus. The notation is *reversed* because he placed the operators before the operands.

```
2.0 * 3.14159 * Radius
```

the compiler will do the multiplication once at compile-time and store the result. There is no reason to lower the readability of a program by doing the constant folding yourself, though you may wish to assign a name to the computed value:

Ada

```
PI: constant := 3.14159;
Two_PI: constant := 2.0 * PI;
Circumference: Float := Two_PI * Radius;
```

4.8 Assignment statements

The meaning of the assignment statement:

```
variable := expression;
```

is that the value of expression should be stored in the memory address denoted by variable. Note that the left-hand side of the statement may also be an expression, as long as that expression evaluates to an address:

Ada

```
a(i*(j+1)) := a(i*j);
```

An expression that can appear on the left-hand side of an assignment statement is called an *l-value*; a constant is of course not an l-value. All expressions produce a value and so can appear on the right-hand side of an assignment statement; they are called *r-values*. The language will usually not specify the order of evaluation of the expressions on the left- and right-hand sides of an assignment. If the order affects the result, the program will not be portable.

C defines the assignment statement itself as an expression. The value of:

```
variable = expression;
```

is the same as the value of the expression on the right-hand side:

C

```
int v1, v2, v3;
v1 = v2 = v3 = e;
```

means assign (the value of) e to v3, then assign the result to v2, then assign the result to v1 and ignore the final result.

In Ada assignments are statements, not expressions, and there is no multiple assignment. The multiple declaration:

Ada

```
V1, V2, V3: Integer := E;
```

is considered to be an abbreviation for:

Ada

```
V1: Integer := E;
V2: Integer := E;
V3: Integer := E;
```

and not an instance of multiple assignment.

While C programming style makes use of the fact that assignment is an expression, it probably should be avoided because it can be a source of obscure programming errors. A notorious class of bugs is due to confusion between the assignment ("=") and the equality operator ("=="). In the following statement:

C

```
if (i = j) ...
```

the programmer probably intended just to compare i and j, but i is inadvertently modified by the assignment. Some C compilers regard this as such bad style that they issue a warning message.

A useful feature in C is the combination of an operator with an assignment:

Ada

```
v += e;                 /* This is short for ... */
v = v + e;              /*      this. */
```

Assignment operators are particularly important when the variable is complex, including array indexing, etc. The combination operator not only saves typing, but avoids the possibility of a bug if v is not written identically on both sides. Assignment operators are a stylistic device only, since an optimizing compiler can remove the second evaluation of v.

It is possible to prevent the assignment of a value to an object by declaring the object as a *constant*:

```
const int N = 8;            /* Constant in C */
N: constant Integer := 8;   -- Constant in Ada
```

Obviously a constant must be given an initial value.

There is a difference between a constant and a *static value*[5] which is known at compile-time:

Ada

```
procedure P(C: Character) is
   C1: constant Character := C;
   C2: constant Character := 'x';
begin
   ...
   case C is
      when C1 =>                -- Error, not static
      when C2 =>                -- OK, static
      ...
```

[5] Called a *constant expression* in C++.

```
    end case;
    ...
end P;
```

The local variable C1 is a constant object, meaning that its value cannot be changed *within* the procedure, even though its value will be different each time the procedure is called. On the other hand, the case statement selections must be known at compile time.

Unlike C, C++ considers constants to be static:

C++

```
const int N = 8;
int a[N];                    // OK in C++, not in C
```

Implementation

Once the expression on the right-hand side of an assignment statement has been evaluated, one instruction is needed to store the value of the expression in a memory location. If the left-hand side is complex (array indexing, etc.), additional instructions will be needed to compute the memory address that it specifies.

If the right-hand side is more than one word long, several instructions will be needed to store the value, unless the computer has a *block copy* operation which can copy a sequence of memory words given: the source starting address, the target starting address and the number of words to copy.

4.9 Exercises

1. Read your compiler documentation and list the precisions used for the different integer types.

2. Write $200 + 55 = 255$ and $100 - 150 = -50$ in two's complement notation.

3. Let a take on all values in the ranges 50..56 and $-56..-50$, and let b be either 7 or -7. What are the possible quotients q and remainders r when a is divided by b? Use both definitions of remainder (denoted rem and mod in Ada), and display the results in graphical form. Hint: if rem is used, r will have the sign of a; if mod is used, r will have the sign of b.

4. What happens if you execute the following C program on a computer which stores short int values in 8 bits and int values in 16 bits?

   C

   ```
   short int i;
   int j = 280;
   for (i = 0; i ¡ j; i++) printf("Hello world");
   ```

5. If a non-standard representation of an enumeration type is used, how would you implement the Ada attribute T'Succ(V)?

6. What will the following program print? Why?

C

```
int i = 2;
int j = 5;
if (i & j) printf("Hello world");
if (i && j) printf("Goodbye world");
```

7. What is the value of i after executing the following statements?

C

```
int i = 0;
int a[2] = {10,11};
i = a[i++];
```

8. C and C++ do not have an exponentiation operator; why?

9. Show how modular types in Ada 95 and unsigned integer types in C can be used to represent sets. How portable is your solution? Compare with the set type in Pascal.

10. Given an arithmetic expression such as:

```
(a + b) * (c + d)
```

Java specifies that it be evaluated from left to right, while C++ and Ada allow the compiler the evaluate the subexpression in any order. Why is Java more strict in its specification?

11. Compare the final construct in Java with constants in C++ and Ada.

5 Composite Data Types

From the beginning, programming languages have provided support for composite data types. Arrays are used to model vectors and matrices in mathematical models of the real world. Records are used in business data processing to model the forms and files that collect a variety of data into a single unit.

As with any type, we must describe the set of values of composite types and the operations on these values. In the case of composite types: how are they constructed from elementary values, and what operations can be used to access components of the composite value? The predefined operations on composite types are relatively limited, and most operations must be explicitly programmed from operations on the components of the composite type.

Since arrays are a specialization of records, we begin the discussion with records (called *structures* in C).

5.1 Records

A value of a record type is composed of a set of values of other types called *components* (Ada), *members* (C) or *fields* (Pascal). Within the type declaration, each field has a name and a type. The following declaration in C shows a structure with four components: one of type string, one of a user-defined enumeration type and two of integer type:

C

```
typedef enum {Black, Blue, Green, Red, White} Colors;

typedef struct {
    char        model[20];
    Colors      color;
    int         speed;
    int         fuel;
} Car_Data;
```

The equivalent declaration in Ada is:

Ada

```
type Colors is (Black, Blue, Green, Red, White);

type Car_Data is
```

```
record
   Model:      String(1..20);
   Color:      Colors:
   Speed:      Integer;
   Fuel:       Integer;
end record;
```

Once a record type has been defined, objects (variables and constants) of the type can be declared. Assignment is possible between variables of the same record type:

C

```
Car_Data c1, c2;
c1 = c2;
```

and in Ada (but not in C) it is also possible to check equality of values of the type:

Ada

```
C1, C2, C3: Car_Data;
if C1 = C2 then
   C1 = C3;
end if;
```

Since a type is a set of values, you would think that it is always possible to denote a *value* of a record type. Surprisingly, this is not possible in general; for example, C permits record values only in initializations. In Ada, however, it is possible to construct a value of a record type, called an *aggregate*, simply by supplying a value of the correct type for each field. The association of a value with a field may be by position within the record, or by name of the field:

Ada

```
if C1 = ("Peugeot", Blue, 98, 23) then ...
C1 := ("Peugeot", Red, C2.Speed, C3.Fuel);
C2 := (Model => "Peugeot", Speed => 76,
          Fuel => 46, Color => White);
```

This is extremely important because the compiler will report an error if you forget to include a value for a field; if you use individual assignments, it is easy to forget one of the fields:

Ada

```
C1.Model := "Peugeot";
                              -- Forgot C1.Color
C1.Speed := C2.Speed;
C1.Fuel := C3.Fuel;
```

Individual fields of a record can be *selected* using a period and the field name:

C

```
c1.speed = c1.fuel * x;
```

Once selected, a field of a record is a normal variable or value of the field type, and all operations appropriate to that type are applicable.

The names of record fields are local to the type definition and can be reused in other definitions:

C

```
typedef struct {
        float speed;          /* Reuse field name */
} Performance;

Performance p;
Car_Data c;
p.speed = (float) c.speed;  /* Same name, different field */
```

Single records as such are not very useful; the importance of records is apparent only when they form part of more sophisticated data structures such as arrays of records, or dynamic structures that are created with pointers (Section 8.2).

Implementation

A record value is represented by a sufficient number of memory words to include all of its fields. The layout of the record Car_Data is shown in Figure 5.1. The fields are normally laid out in order of their appearance in the record type definition.

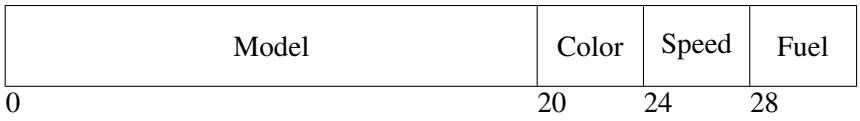

Figure 5.1: Implementation of a record

Accessing a specific field is very efficient because each field is placed at an offset from the beginning of the record which is constant and known at compile-time. Most computers have addressing modes that enable a constant to be added to an address register while the instruction is decoded. Once the address of the beginning of the record has been loaded into a register, subsequent accesses of fields do not require additional instructions:

```
load    R1,&C1      Address of record
load    R2,20(R1)   Load second field
load    R3,24(R1)   Load third field
```

Since a field may need an amount of memory that is not a multiple of the word size, the compiler may *pad* the record to ensure that each field is on a word boundary, because accesses not on a word boundary are much less efficient. On a 16-bit computer, the following type definition:

C

```
typedef struct {
        char  f1;           /* 1 byte, skip 1 byte */
        int   f2;           /* 2 bytes */
        char  f3;           /* 1 byte, skip 1 byte */
        int   f4;           /* 2 bytes */
};
```

may cause four words to be allocated to each record variable so that fields of type int are word-aligned, whereas:

C

```
typedef struct {
        int    f2;          /* 2 bytes */
        int    f4;          /* 2 bytes */
        char   f1;          /* 1 byte */
        char   f3;          /* 1 byte */
};
```

would require only three words. When using a compiler that packs fields tightly, you may be able to improve efficiency by adding dummy fields to pad out fields to word boundaries. See also Section 5.9 for ways of explicitly mapping fields. In any case, *never* assume a specific record layout, as that will make your program totally non-portable.

5.2 Arrays

An array is a record all of whose fields are of the same type. Furthermore, these fields (called *elements* or *components*) are not named by identifiers but by position within the array. The power of the array data type comes from the ability to efficiently access an element by using an *index*. Since all the elements are of the same type, it is possible to compute the location of a specific element by multiplying the index by the size of an element. Using indices it is easy to search through an array for a specific element, or to sort or otherwise rearrange the elements.

The index type in Ada may be any discrete type, that is any type on which "counting" is possible; these are integer types and enumeration types (including Character and Boolean):

Ada

```
type Heat is (Off, Low, Medium, High);
type Temperatures is array(Heat) of Float;
Temp: Temperatures;
```

C restricts the index type to integers; you specify *how many* components you want:

C

```
#define Max 4
float temp[Max];
```

and the indices are implicitly from 0 to one less than the number of components, in this case from 0 to 3. C++ allows any constant expression to be used for the array count, which improves readability:

C++

```
const int last = 3;
float temp[last+1];
```

The array component may be of any type:

C

```
typedef struct {...} Car_Data;
Car_Data database[100];
```

In Ada (but not in C) the operations of assignment and equality checking may be done on arrays:

Ada

```
type A_Type is array(0..9) of Integer;
A, B, C: A_Type;

if A = B then A := C; end if;
```

As with records, array values called aggregates exist in Ada with an extensive range of syntactic options:

Ada

```
A := (1,2,3,4,5,6,7,8,9,10);
A := (0..4 => 1, 5..9 => 2);        -- Half one's, half two's
A := (others => 0);                 -- All zeros
```

In C, array aggregates are limited to initial values.

The most important array operation is *indexing*, which selects an element of an array. The index value, which may be an arbitrary expression of the array index type, is written following the array name:

Ada

```
type Char_Array is array(Character range 'a' .. 'z') of Boolean;
A: Char_Array := (others => False);
C: Character := 'z';

A(C) := A('a') and A('b');
```

Another way of understanding arrays is to consider them to be a function from the index type to the element type. Ada (like Fortran but unlike Pascal and C) encourages this view by using the same syntax for function calls and for array indexing. That is, without looking at the declaration you cannot know whether A(I) is a call to a function or an indexing operation on an array. The advantage to this common syntax is that a data structure may be initially implemented as an array, and later on if a more complex data structure is needed, the array can be replaced by a function without modifying the array accesses. The use of brackets rather than parentheses in Pascal and C is mostly for the convenience of the compiler.

Records and arrays may be nested arbitrarily in order to construct complex data structures. To access an elementary component of such a structure, field selection and element indexing must be done one after the other until the component is reached:

C

```
typedef int A[10];          /* Array type */
typedef struct {            /* Record type */
   A      a;                /* Array within record */
   char   b;
   }  Rec;

Rec    r[10];               /* Array of records of arrays of int */
int    i,j,k;

k = r[i+1].a[j-1];          /* Index, then select, then index */
                            /* Final result is an integer value */
```

Note that partial selecting and indexing in a complex data structure yields values that are themselves arrays or records:

C

```
r                           Array of record of array of integer
r[i]                        Record of array of integer
r[i].a                      Array of integer
r[i].a[j]                   Integer
```

and these values can be used in assignment statements, etc.

5.3 Reference semantics in Java

Perhaps the worst aspect of C (and C++) is the unrestricted and excessive use of pointers. Not only are pointer manipulations difficult to understand, they are extremely error-prone as described in Chapter 8. The situation in Ada is much better because strong type-checking and access levels ensure that pointers cannot be used to break the type system, but data structures still must be built using pointers.

Java (like Eiffel and Smalltalk) uses *reference semantics* instead of *value semantics*.[1] Whenever a variable of non-primitive type[2] is declared, you do not get a block of memory allocated; instead, an implicit pointer is allocated. A second step is needed to actually allocate memory for the variable. We now demonstrate how reference semantics works in Java.

Arrays

If you declare an array in C, you are allocated the memory you requested (Figure 5.2(a)):

[1] Do not confuse these terms with the similar terms used to discuss parameter passing in Section 7.2.

[2] Java's primitive types are: float, double, int, long, short, byte, char, boolean. Do not confuse this term with *primitive operations* in Ada (Section 14.5).

```
int a_c[10];
```
C

while in Java, you only get a pointer that *can be* used to store an array (Figure 5.2(b)):

```
int[] a_java;
```
Java

Allocating the array takes an additional instruction (Figure 5.2(c)):

```
a_java = new int[10];
```
Java

though it is possible to combine the declaration with the allocation:

```
int[] a_java = new int[10];
```
Java

If you compare Figure 5.2 with Figure 8.4, you will see that Java arrays are similar to structures defined in C++ as `int *a` rather than as `int a[]`. The difference is that the pointer is implicit so you do not need to concern yourself with pointer manipulation or memory allocation. In fact, in the

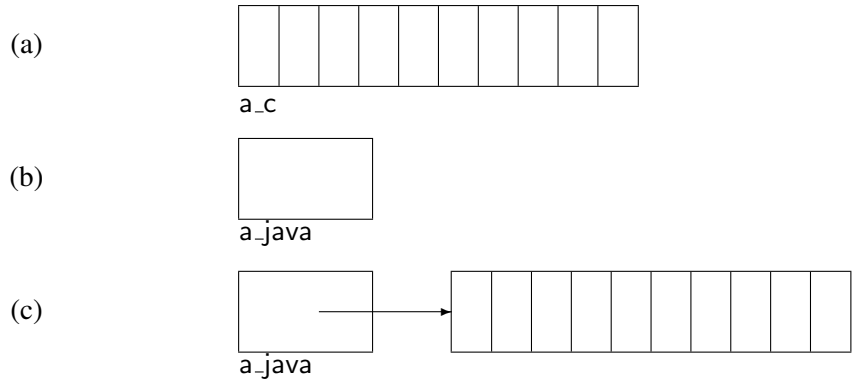

Figure 5.2: Value versus reference semantics

case of an array, the variable will be a dope vector (Figure 5.5), enabling bounds checking to be done when accessing the array.

Note that the Java syntax is rather easier to read than the C++ syntax: `a_java` is of *type* `int[]`, that is "integer array"; in C++, the component type `int` and the array indication `[10]` appear on opposite sides of the variable name.[3]

Under reference semantics, dereferencing of the pointer is implicit, so that once the array is created you access the array as usual:

[3]To ease the transition to Java, the language also accepts the C++ syntax.

Java

```
for (i = 1; i < 10; i++)
    a_java[i] = i;
```

Of course, this indirect access can be much less efficient than direct access unless optimized by the compiler.

Note that allocating an object and assigning it to a variable can be done in any statement, resulting in the following possibility:

Java

```
int[] a_java = new int[10];
...
a_java = new int[20];
```

The variable a_java, which pointed to a ten-element array, now points to a twenty-element array, and the original array is now garbage (Figure 8.9). The Java model specifies that a garbage collector must exist within the JVM.

5.4 Arrays and type checking

Probably the most common cause of difficult bugs is indexing that exceeds array limits:

C

```
int a[10];
for (i = 0; i ¡= 10; i++)
    a[i] = 2*i;
```

The loop will also be executed for i=10 but the last element of the array is a[9].

The reason that this type of bug is so common is that index expressions can be arbitrary expressions, but valid indices fall within the range defined in the array declaration. The slightest mistake can cause the index to receive a value that is outside this narrow range. The reason that the resulting bug is so disastrous is that an assignment to a[i] if i is out of range causes some *unrelated* memory location to be modified, perhaps even in the operating system area. Even if hardware protection restricts the modification to data within your own program, the bug will be difficult to find because the *symptoms* will be in the wrong place, namely in unrelated statements that use the modified memory rather than in the statement that caused the modification.

To demonstrate: if a numerical mistake causes a variable speed to receive the value 20 instead of 30:

C

```
int x = 10, y = 50;
speed = (x + y) / 3;        /* Compute average ! */
```

the symptom is the incorrect value of speed, and the cause (division by 3 instead of 2) is in a statement that calculates speed. The symptom is directly related to the mistake, and by using breakpoints or watchpoints you can quickly locate the problem. In the following example:

C

```
int a[10];
int speed;

for (i = 0; i ¡= 10; i++)
    a[i] = 2*i;
```

speed is a victim of the fact that it was arbitrarily declared just after a and thus modified in a totally unrelated statement. You can trace the computation of speed for days without finding the bug.

The solution to these problems is to check indexing operation on arrays to ensure that the bounds are respected. Any attempt to exceed the array bounds is considered to be a violation of type checking. The first language to propose index checking was Pascal:

Pascal

```
type A_Type = array[0..9] of Integer;
A: A_Type;
A[10] := 20;                (* Error *)
```

With array type checking, the bug is caught *immediately* where it occurs and not after it has "smeared" some unrelated memory location; an entire class of difficult bugs is removed from programs. More precisely: such errors become compile-time errors instead of run-time errors.

Of course you don't get something for nothing, and there are two problems with array type checking. The first is that there is a run-time cost to the checks which we will discuss in a later section. The second problem has to do with an inconsistency between the way we work with arrays and the way that type checking works. Consider the following example:

Pascal

```
type A_Type = array[0..9] of Real;   (* Array types *)
type B_Type = array[0..8] of Real;
A: A_Type;                           (* Array variables *)
B: B_Type;

procedure Sort(var P: A_Type);       (* Array parameter *)

sort(A);                             (* OK *)
sort(B);                             (* Error! *)
```

The two array type declarations define distinct types. The type of the actual parameter to a procedure must match the type of the formal parameter, so it seems that two different Sort procedures are needed, one for each type. This is inconsistent with our intuitive concept of arrays and array operations, because the detailed programming of procedures like Sort does not depend on the number of elements there are in an array; the array bounds should simply be additional parameters. Note that this problem does not occur in Fortran or C simply because those languages do not have array parameters! They simply pass the address of the start of the array and it is the responsibility of the programmer to correctly define and use the array bounds.

The Ada language has an elegant solution to this problem.[4] The type of an array in Ada is determined solely by its *signature*, that is the type of the index and the type of the element. Such a type is called an *unconstrained array type*. To actually declare an array, an *index constraint* must be appended to the type:[5]

Ada

```
type A_Type is array(Integer range <>) of Float;
                              -- Unconstrained array type declaration

A: A_Type(0..9);              -- Array with index constraint
B: A_Type(0..8);              -- Array with index constraint
```

The signature of A_Type is a one-dimensional array with Integer indices and Float components; the index bounds are *not* part of the signature.

As in Pascal, indexing operations are fully checked:

Ada

```
A(9) := 20.5;                 -- OK, index range is 0..9
B(9) := 20.5;                 -- Error, index range is 0..8
```

The importance of unconstrained arrays is apparent when we look at procedure parameters. Since the type of (unconstrained) array parameters is determined by the signature only, we can call a procedure with any actual parameter of that type regardless of its index constraint:

Ada

```
procedure Sort(P: in out A_Type);
                              -- Unconstrained array type parameter

Sort(A);                      -- Type of A is A_Type
Sort(B);                      -- Type of B is also A_Type
```

The question now arises: how can the Sort procedure access the array bounds? In Pascal, the bounds were part of the type and so were also known within the procedure. In Ada, the constraint of the actual array parameter is automatically passed to the procedure at run-time, and can be accessed using functions called *attributes*. Given any array A:

- A'First is the index of the first element in A.
- A'Last is the index of the last element in A.
- A'Length is the number of elements in A.
- A'Range is equivalent to A'First..A'Last.

For example:

[4]The Pascal standard defines *conformant array parameters* whose bounds are implicitly passed to a procedure, but we will focus on the Ada solution which introduces a new generalized concept not limited to array parameters.

[5]The symbol "<>" is read "box".

Ada

```
procedure Sort(P: in out A_Type) is
begin
   for I in P'Range loop
      for J in I+1 .. P'Last loop
   ...
end Sort;
```

The use of array attributes enables the programmer to write software that is extremely robust to modifications: any change in the array bounds is automatically reflected in the attributes.

To summarize: array type checking is a powerful tool for improving the reliability of programs; however, the definition of the array bounds should not be part of the static definition of a type.

5.5 * Array subtypes in Ada

The subtypes that we discussed in Section 4.5 were defined by appending a *range constraint* to a discrete type (integer or enumeration). Similarly, an array subtype may be declared by appending an *index constraint* to an unconstrained array type:

```
type A_Type is array(Integer range <>) of Float;
subtype Line is A_Type(1..80);
L, L1, L2: Line;
```

A value of this named subtype may be used as an actual parameter corresponding to a formal parameter of the underlying unconstrained type:

```
Sort(L);
```

In any case, the unconstrained formal parameter of Sort is dynamically constrained by the actual parameter each time the procedure is called.

The discussion of subtypes in Section 4.5 is applicable here. Arrays of different subtypes of the same type can be assigned to each other (provided that they have the same number of elements), but arrays of different types cannot be assigned to each other without an explicit type conversion. The definition of a named subtype is just a convenience.

Ada has powerful constructs called *slices* and *sliding*, which enable partial arrays to be assigned:

```
L1(10..15) := L2(20..25);
```

which assigns a slice of one array to another, sliding the indices until they match. The type signatures are checked at compile-time, while the constraints are checked at run-time and can be dynamic:

```
L1(I..J) := L2(I*K..M+2);
```

The difficulties with array type definitions in Pascal led the designers of Ada to generalize the solution for arrays to the elegant concept of subtypes: separate the static type specification from the constraint which can be dynamic.

5.6 String type

Basically strings are simply arrays of characters, but additional language support is required for programming convenience. The first requirement is to have a special syntax for strings, otherwise working with arrays of characters would be too tedious. Both of the following declarations are valid but of course the first form is much more convenient:

C

```
char s[] = "Hello world";
char s[] = {'H','e','l','l','o',' ','w','o','r','l','d', '\0'};
```

Next we have to find some way of dealing with string length. The above example already shows that the compiler can determine the size of a string without having the programmer specify it explicitly. C uses a convention on the representation of strings, namely that the first zero byte encountered terminates the string. String processing in C typically contains a while-loop of the form:

C

```
while (s[i++] != '\0') ...
```

The main drawback to this method is that if the terminating zero is somehow missing, memory can be smeared just like any statement that causes array bounds to be exceeded:[6]

C

```
char s[11] = "Hello world";/* No room for zero byte */
char t[11];
strcpy(t, s);                    /* Copy s to t. How long is s? */
```

Other disadvantages of this method are:

- String operations require dynamic allocation and deallocation of memory which is relatively inefficient.
- Calls to string library functions require that the string length be recalculated.
- The zero byte cannot be part of the string.

An alternative solution used by some dialects of Pascal is to include an explicit length byte as the implicit zero'th character of a string whose maximum length is specified when it is declared:

[6] This particular error would be caught in C++ which does not ignore extra members in an initialization.

Pascal

```
S: String[10];
S := 'Hello world';          (* Needs 11 bytes *)
writeln(S);
S := 'Hello';
writeln(S);
```

The first output of the program will be "Hello worl" because the string will be truncated to fit the declared length. The second output will be "Hello" because the writeln statement takes the implicit length into account. Unfortunately, this solution is also flawed because it is possible to directly access the hidden length byte and smear memory:

Pascal

```
s[0] := 15;
```

In Ada there is a predefined unconstrained array type called String whose definition is:[7]

Ada

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

Each string must be of fixed length and declared with a constraint:

Ada

```
S: String(1..80);
```

Unlike C where all string processing is done using library procedures like strcpy, Ada includes operators on strings such as concatenation "&", equality and relational operators like "¡". Type checking is strictly enforced, so a certain amount of gymnastics with attributes is needed to make everything work out:

Ada

```
S1: constant String := "Hello";
S2: constant String := "world";
T: String(1 .. S1'Length + 1 + S2'Length) := S1 & ' ' & S2;
Put(T);                          -- Will print Hello world
```

The exact length of T must be computed before the assignment is done! Fortunately, Ada supports array attributes and a construct for creating subarrays (called slices) that make it possible to do such computations in a portable manner.

Ada 83 provides the framework for defining strings whose length is not fixed, but not the necessary library subprograms for string processing. To improve portability, Ada 95 defines standard libraries for all three categories of strings: fixed, varying (as in Pascal) and dynamic (as in C).

[7] Using predefined subtype Positive is Integer range 1..Integer'Last.

5.7 Multi-dimensional arrays

Multi-dimensional matrices are extensively used in mathematical models of the physical world, and multi-dimensional arrays have appeared in programming languages since Fortran. There are actually two ways of defining multi-dimensional arrays: directly and as complex structures. We will limit the discussion to two-dimensional arrays; the generalization to higher dimensions is immediate.

A two-dimensional array can be directly defined in Ada by giving two index types separated by a comma:

Ada

```
type Two is
   array(Character range <>, Integer range <>) of Integer;
T: Two('A'..'Z', 1..10);
I: Integer;
C: Character;

T('X', I*3) := T(C, 6);
```

As the example shows, the two dimensions need not be of the same type. Selecting an array element is done by giving both indices.

The second method of defining a two-dimensional array is to define a type that is an array of arrays:

Ada

```
type I_Array is array(1..10) of Integer;
type Array_of_Array is array (Character range <>) of I_Array;
T: Array_of_Array('A'..'Z');
I: Integer;
C: Character;

T('X')(I*3) := T(C)(6);
```

The advantage of this method is that by using one index operation, the elements of the second dimension (which are themselves arrays) can be accessed:

Ada

```
T('X') := T('Y');          -- Assign 10-element arrays
```

The disadvantage is that the elements of the second dimension must be constrained before they can be used to define the first dimension.

In C, only the second method is available and of course only for integer indices:

C

```
int a[10][20];
a[1] = a[2];               /* Assign 20-element arrays */
```

Pascal does not distinguish between a two-dimensional array and an array of arrays; since the bounds are considered to be part of the array type this causes no difficulties.

5.8 Array implementation

Arrays are represented by placing the array elements in sequence in memory. Given an array A the address of A(I) is (Figure 5.3):

$$addr\,(A) + size\,(element) * (i - A'First)$$

For example: the address of A(4) is $20 + 4 * (4 - 1) = 32$.

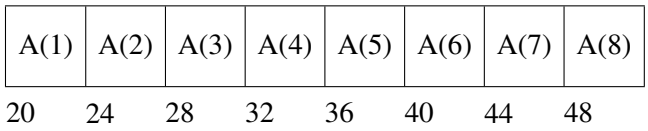

Figure 5.3: Implementation of array indexing

The generated machine code is:

```
load    R1,I          Get index
sub     R1,A'First    Subtract lower bound
multi   R1,size       Multiply by size -> offset
add     R1,&A         Add array addr. -> element addr.
load    R2,(R1)       Load contents
```

It may surprise you to learn how many instructions are needed for each array access!

There are many optimizations that can be done to improve this code. First note that if A'First is zero, we save the subtraction of the index of the first element; this explains why the designers of C specified that indices always start at zero. Even if A'First is not zero but is known at compile time, the address calculation can be rearranged as follows:

$$(addr\,(A) - size\,(element) * A'First) + (size\,(element) * i)$$

The first expression in parentheses can be computed at compile-time saving the run-time subtraction. This expression will be known at compile-time in a direct use of an array:

Ada

```
A: A_Type(1..10);
A(I) := A(J);
```

but not if the array is a parameter:

Ada

```
procedure Sort(A: A_Type) is
begin
   ...
   A(A'First+I) := A(J);
   ...
end Sort;
```

The main obstacle to efficient array operations is the multiplication by the size of the array element. Fortunately, most arrays are of simple data types like characters or integers whose size is a power of two. In this case, the costly multiplication operation can be replaced by an efficient shift, since shift left by n is equivalent to multiplication by 2^n. If you have an array of records, you can squeeze out more efficiency (at the cost of extra memory) by padding the records to a power of two. Note that the portability of the program is not affected, but the *improvement* in efficiency is not portable: another compiler might lay out the record in a different manner.

C programmers can sometimes improve the efficiency of a program with arrays by explicitly coding access to array elements using pointers rather than indices. Given definitions such as:

C

```
typedef struct {
    ...
    int field;
} Rec;
Rec a[100];
```

it may be more efficient (depending on the quality of the compiler's optimizer) to access the array using a pointer:

C

```
Rec* ptr;

for (ptr = &a; ptr < &a+100*sizeof(Rec); ptr += sizeof(Rec))
    ... ptr->field ...;
```

than using an indexing operation:

C

```
for (i = 0; i ¡ 100; i++)
    ... a[i].field ...;
```

However, this style of coding is extremely error-prone, as well as hard to read, and should be avoided except when absolutely necessary.

A possible way of copying strings in C is to use:

C

```
while (*s1++ = *s2++)
    ;
```

where there is a null statement before the semicolon. If the computer has block-copy instructions which move the contents of a block of memory cells to another address, a language like Ada which allows array assignment would be more efficient. In general, C programmers should use library functions which are likely to be implemented more efficiently than the naive implementation shown above.

Multi-dimensional arrays can be very inefficient because each additional dimension requires an extra multiplication to compute the index. When programming with multi-dimensional arrays, you must also be aware of the layout of the data. Except for Fortran, all languages store a two-dimensional array as a sequence of rows. The layout of:

Ada

```
type T is array(1..3, 1..5) of Integer;
```

is shown in Figure 5.4. This layout is consistent with the similarity between a two-dimensional

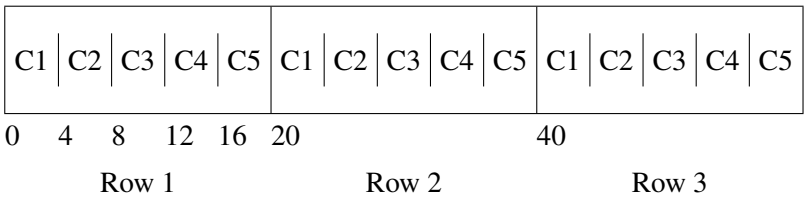

Figure 5.4: Implementation of a multi-dimensional array

array and an array of arrays. If you wish to compute a value for every element of a two-dimensional array, make sure that the last index is in the inner loop:

C

```
int matrix[100][200];

for (i = 0; i ¡ 100; i++)
    for (j = 0; j ¡ 200; j++)
        m[i][j] = . . . ;
```

The reason is that operating systems that use paging are vastly more efficient if successive memory accesses are close together.

If you wish to squeeze the best performance out of a program in C, you can ignore the two-dimensional structure of the array and pretend that the array is one-dimensional:

C

```
for (i = 0; i ¡ 100*200; i++)
        m[0][i] = . . . ;
```

Needless to say, this is not recommended and should be carefully documented if used.

Array type checking requires that the index be checked against the bounds before each array access. The overhead to this check is high: two comparisons and jumps. Compilers for languages like Ada have to invest significant effort to optimize the instructions for array processing. The main technique is to utilize available information. In the following example:

Ada

```
for I in A'Range loop
    if A(I) = Key then . . .
```

the index I will take on exactly the permitted values of the array so no checking is required. In general, the optimizer will perform best if all variables are declared with the tightest possible constraints.

When arrays are passed as parameters in a type-checking language:

Ada

```
type A_Type is array(Integer range <>) of Integer;
procedure Sort(A: A_Type) is ...
```

the bounds must also be implicitly passed in a data structure called a *dope vector* (Figure 5.5). The dope vector contains the upper and lower bounds, the size of an element and the address of

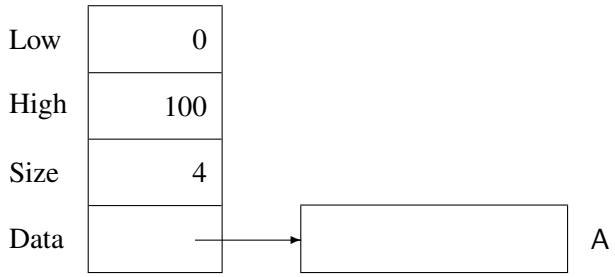

Figure 5.5: Dope vector

the start of the array. As we have seen, this is exactly the information needed to compute an array indexing operation.

5.9 * Representation specification

This book repeatedly stresses the importance of viewing a program as an abstract model of the real world. Nevertheless, many programs such as operating systems, communications packages and embedded software require the ability to manipulate data at the physical level of their representation in memory.

Bitwise computation

C contains Boolean operators which compute on individual bits of *integer* values: "&" (and), "—" (or), "^" (xor), "~" (not).

The Boolean operators in Ada: and, or, xor, not can also be applied to arrays of Boolean:

Ada

```
type Bool_Array is array(0..31) of Boolean;
B1: Bool_Array := (0..15 => True, 16..31 => False);
B2: Bool_Array := (0..15 => False, 16..31 => True);

B1 := B1 or B2;
```

However, just declaring a Boolean array does not ensure that it is represented as a bit string; in fact, a Boolean value is usually represented as a full integer. The addition of the directive:

Ada

```
pragma Pack(Bool_Array);
```

requests the compiler to pack the values of the array as densely as possible. Since only one bit is needed for a Boolean value, the 32 elements of the array can be stored in one 32-bit word. While this does provide the required functionality, it does lack some of the flexibility of C, in particular the ability to use octal or hexadecimal constants like 0xf00f0ff0 in Boolean computations. Ada does provide a notation for such constants, but they are integer values and not arrays of Boolean and so cannot be used in bitwise computation.

These problems are solved in Ada 95: modular types (Section 4.1) can be used for bitwise computation:

Ada

```
type Unsigned_Byte is mod 256;
U1, U2: Unsigned_Byte;

U1 := U1 and U2;
```

Subword fields

Hardware registers are usually composed of several subword fields. Classically, the way to access such fields is to use shift and mask; the statement:

C

```
field = (i >> 4) & 0x7;
```

extracts the three-bit field located four bits from the right of the word i. This programming style is dangerous because it is very easy to make mistakes in specifying the shift counts and masks. In addition, the slightest change in the layout of a word can require massive modification of the program.

An elegant solution to this problem was first introduced by Pascal: use ordinary records, but pack the fields into a single word.[8] An ordinary field access Rec.Field is automatically translated by the compiler into the correct shift and mask.

In Pascal, the mapping was implicit; other languages have a notation for explicit mapping. C allows bit-field specifiers on a structure field (provided that the fields are of integer type):

C

```
typedef struct {
    int     : 3;      /* Padding */
    int f1  : 1;
```

[8] The motivation for this came from the CDC 6000 series computers that were used in the first Pascal implementation. These computers, intended for numerical calculations, used a 60 bit word and had no instructions for subword access except shift and mask.

```
    int f2   : 2;
    int      : 3;      /* Padding */
    int f3   : 2;
    int      : 4;      /* Padding */
    int f4   : 1;
} reg;
```

and then ordinary assignment statements can be used, even though the fields are part of a word (Figure 5.6) and the assignment is implemented by the compiler using shift and mask:

C

```
reg r;
int i;

i = r.f2;
r.f3 = i;
```

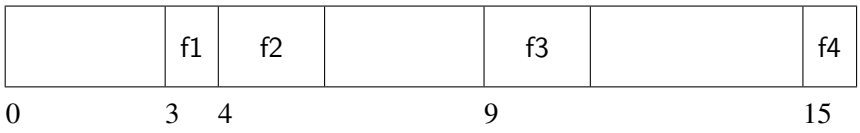

Figure 5.6: Subword record fields

Ada insists that type declarations be abstract; *representation specifications* use a different notation and are written separately from the type declaration. Given the record type declaration:

Ada

```
type Heat is (Off, Low, Medium, High);

type Reg is
    record
        F1: Boolean;
        F2: Heat;
        F3: Heat;
        F4: Boolean;
    end record;
```

a record specification can be appended:

Ada

```
for Reg use
    record
        F1 at 0 range 3..3;
        F2 at 0 range 4..5;
        F3 at 1 range 1..2;
        F4 at 1 range 7..7;
    end record;
```

The at clause specifies the byte within the record and range specifies the bit range assigned to the field, where we know that one bit is sufficient for a Boolean value and two bits are sufficient for a Heat value. Note that padding is not necessary because exact positions are specified.

Once the bit specifiers in C and representation specifications in Ada are correctly programmed, all subsequent accesses are ensured to be free of bugs.

Big endians and little endians

Conventionally, memory addresses start at zero and increase. Unfortunately, computer architectures differ in the way multibyte values are stored in memory. Let us assume that each byte can be separately addressed and that each word takes four bytes. How is the integer 0x04030201 to be stored? The word can be stored so that the most significant byte is first, an order called *big endian*, or so that the least significant byte has the lowest address, called *little endian*. Figure 5.7 shows the numbering of the bytes for the two options.

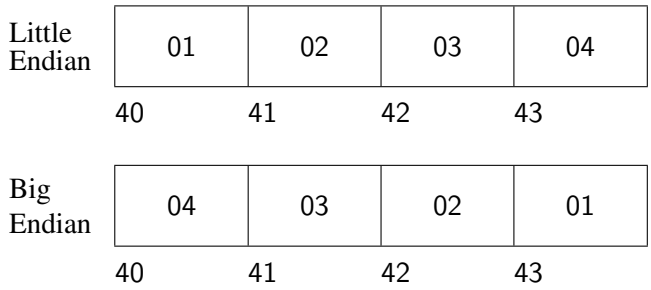

Figure 5.7: Big endian and little endian

This aspect of the computer architecture is handled by the compiler and is totally transparent to the programmer who writes abstractly using records. However, when representation specifications are used, the differences between the two conventions can cause a program to be non-portable. In Ada 95, the bit ordering of a word can be specified by the programmer, so a program using representation specifications can be ported by changing only one statement.

Derived types and representation specifications in Ada

A derived type in Ada (Section 4.6) is defined as a new type whose values and operators are a copy of those of the parent type. A derived type may have a representation that is different from the parent type. For example, once an ordinary type Unpacked_Register is defined:

Ada

```
type Unpacked_Register is
   record
      ...
   end record;
```

a new type may be derived and a representation specification associated with the derived type:

Ada

```
type Packed_Register is new Unpacked_Register;

for Packed_Register use
    record
        ...
    end record;
```

Type conversion (which is allowed between any types that are derived from each other) causes a change of representation, namely, packing and unpacking the subword fields of the record to ordinary variables:

Ada

```
U: Unpacked_Register;
P: Packed_Register;

U := Unpacked_Register(P);
P := Packed_Register(U);
```

This facility can contribute to the reliability of programs, because once the representation specification is correctly written, the rest of the program can be completely abstract.

5.10 Exercises

1. Does your compiler pack record fields or align them on word boundaries?

2. Does your computer have a block-copy instruction, and does your compiler use it for array and record assignments?

3. Pascal contains the construct with which opens the scope of a record so that the field names can be used directly:

   Pascal

   ```
   type Rec =
       record
           Field1: Integer;
           Field2: Integer;
       end;
   R: Rec;

   with R do Field1 := Field2;(* OK, direct visibility *)
   ```

   What are the advantages and disadvantages of the construct? Study the Ada renames construct and show how some of the same functionality can be obtained. Compare the two constructs.

4. Explain the error message that you get if you try to assign one array to another in C:

C

```
int a1[10], a2[10];
a1 = a2;
```

5. Write sort procedures in Ada and C, and compare them. Make sure that you use attributes in the Ada procedure so that the procedure will work on arrays with arbitrary indices.

6. Which optimizations does your compiler do on array indexing operations?

7. Icon has associative arrays called tables, where a string can be used as an array index:

   ```
   count["begin"] = 8;
   ```

   Implement associative arrays in Ada or C.

8. Are the following two types the same?

   Ada

   ```
   type Array_Type_1 is array(1..100) of Float;
   type Array_Type_2 is array(1..100) of Float;
   ```

   Ada and C++ use *name equivalence*: every type declaration declares a new type, so two types are declared. Under *structural equivalence* (used in Algol 68), type declarations that look alike define the same type. What are the advantages and disadvantages of the two approaches?

9. An array object of anonymous type (without a named type) can be defined in Ada. In the following example, is the assignment legal? Why?

   Ada

   ```
   A1, A2: array(1..10) of Integer;
   A1 := A2;
   ```

10. Compare the string processing capabilities of Ada 95, C++ and Java.

7 Subprograms

7.1 Subprograms: procedures and functions

A *subprogram* is a segment of a program that can be invoked from elsewhere within the program. Subprograms are used for various reasons:

- A segment of a program that must be executed at various stages within the computation can be written once as a subprogram and then repeatedly invoked. This saves memory and prevents the possibility of errors caused by copying the code from one place to another.
- A subprogram is a logical unit of program decomposition. Even if a segment is executed only once, it is useful to identify it in a subprogram for purposes of testing, documentation and readability.
- A subprogram can also be used as a physical unit of program decomposition, that is, as a unit of compilation. In Fortran, subprograms (called *subroutines*) are the only units of decomposition and compilation. Modern languages use the module, a group of declarations and subprograms, as the unit of physical decomposition (Chapter 13).

A subprogram consists of:

- A declaration which defines the interface to the subprogram. The subprogram declaration includes the name of the subprogram, the list of parameters (if any)[1] and the type of the value returned (if any).
- Local declarations which are accessible only within the body of the subprogram.
- A sequence of executable statements.

The local declarations and the executable statements form the *body* of the subprogram.

Subprograms that return a value are called *functions*; those that do not are called *procedures*. C does not have a separate syntax for procedures; instead you must write a function that returns void which is a type with no values:

[1] As a point of syntax, subprograms without parameters are usually declared without a parameter list (Ada, Pascal) or with an empty list (C++). C uses the explicit keyword void to indicate an absence of parameters.

```
void proc(int a, float b);
```
C

Such a function has the same properties as a procedure in other languages, so we will use the term procedure even when discussing C.

A procedure is invoked by a *call* statement. In Fortran, there is a special syntax:

```
call proc(x,y)
```
Fortran

while in other languages you simply write the name of the procedure followed by the actual parameters:

```
proc(x,y);
```
C

The semantics of a procedure call is as follows: the current sequence of instructions is suspended; the sequence of instructions within the procedure body is executed; upon completing the procedure body, the execution continues with the first statement following the procedure call. This description ignores parameter passing and scopes which will be the subject of extensive discussion in the next sections.

Since a function returns a value, the function declaration must specify the type of the returned value. In C, the type of a function precedes the function declaration:

```
int func(int a, float b);
```
C

while Ada uses a distinctive syntax for functions:

```
function Func(A: Integer; B: Float) return Integer;
```
Ada

A function call appears not as a statement, but as an element of an *expression*:

```
a = x + func(r,s) + y;
```
C

The result type of the function must be consistent with the type expected in the expression. Note that C does implicit type conversions in many cases, while in Ada the result type must exactly match the context. The meaning of a function call is similar to that of a procedure call: the evaluation of the expression is suspended; the instructions of the function body are executed; the returned value is then used to continue the evaluation of the expression.

The term function is actually very inappropriate to use in the context of ordinary programming languages. When used in mathematics, a function is just a mapping from one set of values to another. To use the technical term, a mathematical function is *referentially transparent*, because its "computation" is transparent to the point at which it is "called". If you have a value 3.6 and you ask for the value of $\sin(3.6)$, you will get the same unique result every time that the function appears in an equation. In programming, a function can perform an arbitrary computation including input-output or modification of global data structures:

C

```
int x,y,z;
int func(void)
{
   y = get( );              /* Modifies a global variable */
   return x*y;              /* Value based on global variable */
}

z = x + func(void) + y;
```

If the optimizer rearranged the order of the computation so that x+y were computed before the function call, a different result will be obtained, because the function modifies the value of y.

Since all C subprograms are functions, C programming style extensively uses return values in non-mathematical situations like input-output subprograms. This is acceptable provided that the possible difficulties with order dependencies and optimization are understood. Programming language research has developed exciting languages that are based on the mathematically correct concept of functions (see Chapter 16).

7.2 Parameters

In the previous section, we defined subprograms as segments of code that may be repeatedly invoked. Almost always, each invocation will require that the code in the subprogram body be executed using different data. The way to influence the execution of a subprogram body is to "pass" it the data that it needs. Data is passed to a subprogram in the form of a sequence of values called *parameters*. The concept is taken from mathematics where a function is given a sequence of *arguments*:[2] $\sin(2\pi r)$.

There are two concepts that must be clearly distinguished:

- A *formal parameter* is a declaration that appears in the declaration of the subprogram. The computation in the body of the subprogram is written in terms of formal parameters.
- An *actual parameter* is a value that the calling program sends to the subprogram.

In the following example:

C

```
int i, j;
char a;
void p(int a, char b)
{
   i = a + (int) b;
}
```

[2]Mathematical terminology uses *argument* for the value passed to a function, while *parameter* is usually used for values that are constant for a specific problem! We will, of course, use the programming terminology.

```
p(i, a);
p(i+j, 'x');
```

the formal parameters of the subprogram p are a and b, while the actual parameters of the first call are i and a, and of the second call, i+j and 'x'.

There are several important points that can be noted from the example. The first is that since the actual parameters are values, they can be constants or expressions, and not just variables. In fact, even when a variable is used as a parameter, what we really mean is "the current value stored in the variable". Secondly, the *name space* of each subprogram is distinct. The fact that the first formal parameter is called a is not relevant to the rest of the program, and it could be renamed, provided, of course, that all occurrences of the formal parameter in the subprogram body are renamed. The variable a declared outside the subprogram is totally independent from the variable of the same name declared within the subprogram. In Section 7.7, we will explore in great detail the relationship between variables declared in different subprograms.

Named parameter associations

Normally the actual parameters in a subprogram call are just listed and the matching with the formal parameters is done by position:

Ada

```
procedure Proc(First: Integer; Second: Character);
Proc(24, 'X');
```

However, in Ada it is possible to use named association in the call, where each actual parameter is preceded by the name of the formal parameter. The order of declaration of the parameters need not be followed:

Ada

```
Proc(Second => 'X', First => 24);
```

This is commonly used together with default parameters, where parameters that are not explicitly written receive the default values given in the subprogram declaration:

Ada

```
procedure Proc(First: Integer := 0; Second: Character := '*');
Proc(Second => 'X');
```

Named association and default parameters are commonly used in the command languages of operating systems, where each command may have dozens of options and normally only a few parameters need to be explicitly changed. However, there are dangers with this programming style. The use of default parameters can make a program hard to read because calls whose syntax is different actually call the same subprogram. Named associations are problematic because they bind the subprogram declaration and the calls more tightly than is usually needed. If you use only positional parameters in calling subprograms from a library, you could buy a competing library and just recompile or link:

Ada

```
X := Proc_1(Y) + Proc_2(Z);
```

However, if you use named parameters, then you might have to do extensive modifications to your program to conform to the new parameter names:

Ada

```
X := Proc_1(Parm => Y) + Proc_2(Parm => Z);
```

7.3 Passing parameters to a subprogram

The definition of the mechanism for passing parameters is one of the most delicate and important aspects of the specification of a programming language. Mistakes in parameter passing are a major source of difficult bugs, so we will go into great detail in the following description.

Let us start from the definition we gave above: the value of the actual parameter is passed to the formal parameter. The formal parameter is just a variable declared within the subprogram, so the obvious mechanism is to copy the value of the actual parameter into the memory location allocated to the formal parameter. This mechanism is called *copy-in semantics* or *call-by-value*. Figure 7.1 demonstrates copy-in semantics, given the procedure definition:

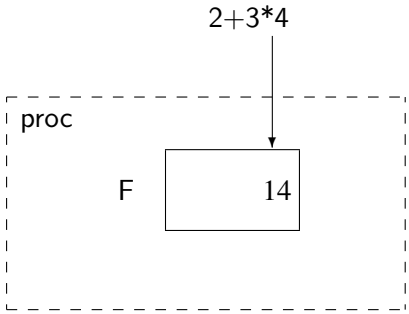

Figure 7.1: Copy-in semantics

Ada

```
procedure Proc(F: in Integer) is
begin
   ...;
end Proc;
```

and the call:

Ada

```
Proc(2+3*4);
```

The advantages of copy-in semantics are:

- Copy-in is the safest mechanism for parameter passing. Since only a copy of the actual parameter is passed, the subprogram cannot cause any damage to the actual parameter, which of course "belongs" to the calling program. If the subprogram modifies the formal parameter, only the copy is modified and not the original.
- Actual parameters can be constants, variables or expressions.
- Copy-in can be extremely efficient because once the initial overhead of the copy is done, all accesses to the formal parameter are to the local copy. As we shall see in Section 7.7, accesses to local variables are extremely efficient.

If copy-in semantics is so good, why are there other mechanisms? The reason is that we will often want to modify the actual parameter despite the fact that such modification is "unsafe":

- A function can only return a single result, so if the result of a computation is more complex, we may want to return several results. The way to do so is to provide a procedure with several actual parameters which can be assigned the results of the computation. Note that this situation can often be avoided by defining a function that returns a record as a result.
- Similarly, the purpose of the computation in the subprogram may be to modify data that is passed to it rather than to compute a result. This is common when a subprogram is maintaining a data structure. For example, a subprogram to sort an array does not compute a value; its only task is to modify the actual parameter. There is no point in sorting a copy of the actual parameter!
- A parameter may be so big that it is inefficient to copy. If copy-in is used for an array of 50,000 integers, there may simply not be enough memory available to make a copy, or the overhead of the copy may be excessive.

The first two situations can easily be solved using *copy-out semantics*. The actual parameter must be a variable, and the subprogram is passed the address of the actual parameter which it saves. A temporary local variable is used for the formal parameter, and a value must be assigned to the formal parameter[3] at least once during the execution of the subprogram. When the execution of the subprogram is completed, the value is copied into the variable pointed to by the saved address. Figure 7.2 shows copy-out semantics for the following subprogram:

Ada

```
procedure Proc(F: out Integer) is
begin
   F := 2+3*4;            -- Assign to copy-out parameter
end Proc;

A: Integer;
Proc(A);                  -- Call procedure with variable
```

[3] Ada 83 does not allow the subprogram to read the contents of the formal parameter. The more common definition of copy-out semantics, which is followed in Ada 95, allows normal computation on the (uninitialized) local variable.

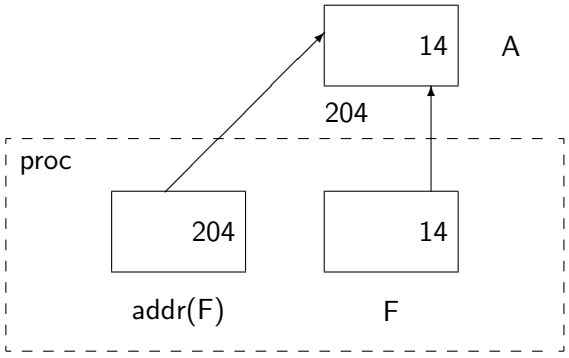

Figure 7.2: Copy-out semantics

When modification of the actual parameter is needed as in sort, *copy-in/out semantics* can be used: the actual parameter is copied into the subprogram when it is called and the final value is copied back upon completion.

However, copy-based parameter passing mechanisms cannot solve the efficiency problem caused by large parameters. The solution, which is known as *call-by-reference* or *reference semantics*, is to pass the address of the actual parameter and to access the parameter indirectly (Figure 7.3). Calling the subprogram is efficient because only a small, fixed-sized pointer is passed for each parameter; however, accessing the parameter can be inefficient because of the indirection.

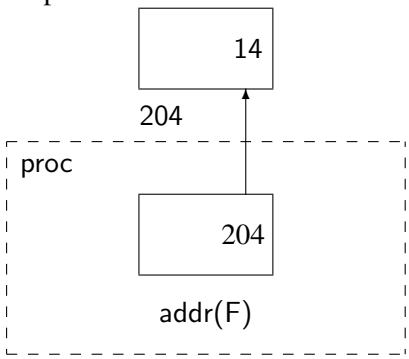

Figure 7.3: Reference semantics

In order to access the actual parameter, its address must be loaded and then an additional instruction is needed to load the value. Note that when using reference (or copy-out) semantics, the actual parameter must be a variable, not an expression, because a value will be assigned to it.

Another problem with call-by-reference is that it may result in *aliasing*: a situation in which the same variable is known by more than one name. In the following example, within the function f the variable global is also known by the alias *parm:

C

```
int global = 4;
int a[10];

int f(int *parm)
```

```
{
   *parm = 5;              /* Same variable as "global" */
   return 6;
}

x = a[global] + f(&global);
```

In the example, if the expression is evaluated in the order in which it is written, its value is a[4]+6, but because of aliasing, the value of the expression may be 6+a[5] if the compiler chooses to evaluate the function call before the array indexing. Aliasing is an important cause of non-portable behavior.

The real disadvantage of call-by-reference is that the mechanism is inherently unsafe. Suppose that for some reason the subprogram thinks that the actual parameter is an array whereas in fact it is just a single integer. This can cause an arbitrary amount of memory to be smeared, since the subprogram is working on the actual parameter, and not just on a local copy. This type of bug is extremely common, because the subprogram will typically have been written by a different programmer than the one calling the subprogram and misunderstandings always occur.

Safety of parameter passing can be improved by insisting on strong type checking which ensures that the types of the formal and actual parameters are compatible. Nevertheless, there is still room for misunderstanding between the programmer who wrote the subprogram and the programmer whose data is being modified. Thus we have an excellent parameter passing mechanism that is not always efficient enough (copy-in semantics), together with mechanisms that are necessary but unsafe (copy-out and reference semantics). The choice is complicated by constraints placed on the programmer by various programming languages. We will now describe the parameter passing mechanisms of several languages in detail.

Parameters in C and C++

C has only one parameter-passing mechanism, copy-in:

C

```
int i = 4;                 /* Global variables */

void proc(int i, float f)
{
   i = i + (int) f;        /* Local "i" */
}

proc(j, 45.0);             /* Function call */
```

In proc, the variable i that is modified is a local copy and not the global i.

In order to obtain the functionality of reference or copy-out semantics, a C programmer must resort to explicit use of pointers:

C

```
int i = 4;                  /* Global variables */

void proc(int *i, float f)
{
   *i = *i + (int) f;       /* Indirect access */
}

proc(&i, 45.0);             /* Address operator needed */
```

After executing proc, the value of the global variable i will be modified. The requirement that pointers be used for reference semantics is unfortunate, because beginning programmers must learn this relatively advanced concept at an early stage of their studies.

C++ has corrected this problem so that true call-by-reference is available using *reference parameters*:

C++

```
int i = 4;                  // Global variables

void proc(int & i, float f)
{
   i = i + (int) f;         // By reference access
}

proc(i, 45.0);              // No address operator needed
```

Note that the programming style is natural and does not use pointers artificially. This improvement in the parameter passing mechanism is so important that it justifies using C++ as a replacement for C.

You will often want to use pointers in C or references in C++ to pass large data structures. Of course, unlike copy-in parameters, there is a danger of accidental modification of the actual parameter. Read-only access to a parameter can be specified by declaring them const:

```
void proc(const Car_Data & d)
{
   d.fuel = 25;             // Error, cannot modify const
}
```

const declarations should be used whenever possible both to clarify the meaning of parameters to readers of the program, and to catch potential bugs.

Another problem with parameters in C is that arrays cannot be parameters. If an array must be passed, the address of the first element of the array is passed, and the procedure has the responsibility for correctly accessing the array. As a convenience, using an array name as a parameter is automatically considered to be the use of a pointer to the first element:

C

```
int b[50];                  /* Array variable */

void proc(int a[])          /* "Array parameter" */
{
   a[100] = a[200];         /* How many components ? */
}

proc(&b[0]);                /* Address of first element */
proc(b);                    /* Address of first element */
```

C programmers quickly get used to this but it is a source of confusion and bugs. The problem is that since the parameter is actually a pointer to a single element, *any* pointer to a variable of a similar type is accepted:

C

```
int i;
void proc(int a[]);         /* "Array parameter" */
proc(&i);                   /* Any pointer to integer is OK !! */
```

Finally, in C no type checking is done between files so that it is possible to declare:

C

```
void proc(float f) { ... }  /* Procedure definition */
```

in one file and:

C

```
void proc(int i);           /* Procedure declaration */
proc(100);
```

in another file, and then spend a month looking for the bug.

The C++ language requires that parameter type checking be performed. However, the language does not require that implementations include a library facility as in Ada (see Section 13.3) that can ensure type checking across separately compiled files. C++ compilers implement type checking by cooperation with the linker: parameter types are encoded in the external name of the subprogram (a process called *name mangling*), and the linker will make sure that calls are linked only to subprograms with the correct parameter signature.[4] Unfortunately, this method cannot catch all type mismatches.

Parameters in Pascal

In Pascal, parameters are passed by value unless reference semantics is explicitly requested:

[4]For details, see Section 7.2c of the Annotated Reference Manual.

Pascal

```
procedure proc(P_Input: Integer; var P_Output: Integer);
```

The keyword var indicates that the following parameter is called by reference, otherwise call-by-value is used even if the parameter is very large. Parameters can be of any type including arrays, records or other complex data structures. The one limitation is that the result type of a function must be a scalar. The types of actual parameters are checked against the types of the formal parameters.

As we discussed in Section 5.4, there is a serious problem in Pascal because the array bounds are considered part of the type. The Pascal standard defines *conformant array parameters* to solve this problem.

Parameters in Ada

Ada takes a novel approach in defining parameter passing in terms of intended use rather than in terms of the implementation mechanism. For each parameter you must explicitly choose one of three possible *modes*:

in The parameter may be read but not written (default).

out The parameter may be written but not read.

in out The parameter may be both read and written.

For example:

Ada

```
procedure Put_Key(Key: in Key_Type);
procedure Get_Key(Key: out Key_Type);
procedure Sort_Keys(Keys: in out Key_Array);
```

In the first procedure, the parameter Key must be read so that it can be "put" into a data structure (or output device). In the second, a value is obtained from a data structure and upon completion of the procedure, the value is assigned to the parameter. The array Keys to be sorted must be passed as in out, because sorting involves both reading and writing the data of the array.

Ada restricts parameters of a function to be of mode in only. This does not make Ada functions referentially transparent because there is still no restriction on accessing global variables, but it can help the optimizer to improve the efficiency of expression evaluation.

Despite the fact that the modes are not defined in terms of implementation mechanisms, the Ada language does specify some requirements on the implementation. Parameters of elementary type (numbers, enumerations and pointers) must be implemented by copy semantics: copy-in for in parameters, copy-out for out parameters, and copy-in/out for in out parameters. The implementation of modes for composite parameters (arrays and records) is not specified, and a compiler may choose whichever mechanism it prefers. This introduces the possibility that the correctness of an

Ada program depends on the implementation-chosen mechanism, so such programs are simply not portable.[5]

Strong type checking is done between formal and actual parameters. The type of the actual parameter must be the same as that of the formal parameter; no implicit type conversion is ever performed. However, as we discussed in Section 5.4, the subtypes need not be identical as long as they are compatible; this allows an arbitrary array to be passed to an unconstrained formal parameter.

Parameters in Fortran

We will briefly touch on parameter passing in Fortran because it can cause spectacular bugs. Fortran can pass only scalar values; the interpretation of a formal parameter as an array is done by the called subroutine. Call-by-reference is used for all parameters. Furthermore, each subroutine is compiled independently and no checking is done for compatibility between the subroutine declaration and its call.

The language specifies that if a formal parameter is assigned to, the actual parameter must be a variable, but because of independent compilation this rule cannot be checked by the compiler. Consider the following example:

Fortran

```
Subroutine Sub(X, Y)
Real X,Y
X = Y
End

Call Sub(−1.0, 4.6)
```

The subroutine has two parameters of type Real. Since reference semantics is used, Sub receives pointers to the two actual parameters and the assignment is done directly on the actual parameters (Figure 7.4). The result is that the memory location storing the value −1.0 is modified! There

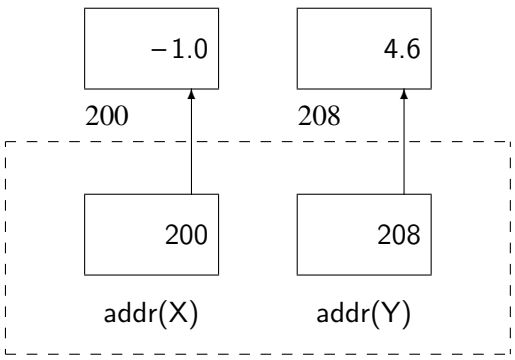

Figure 7.4: Smearing a constant in Fortran

[5] Ada 95 requires that certain categories of parameters be passed by reference; these include task types and tagged types (Section 14.5).

is literally no way to "debug" this bug, since debuggers only allow you to examine and trace variables, not constants. The point of the story is that correct matching of actual and formal parameters is a cornerstone of reliable programming.

7.4 Block structure

A *block* is an entity consisting of declarations and executable statements. A similar definition was given for a subprogram body and it is more precise to say that a subprogram body *is* a block. Blocks in general and procedures in particular can be nested within one another. This section will discuss the relationships among nested blocks.

Block structure was first defined in the Algol language which includes both procedures and unnamed blocks. Pascal contains nested procedures but not unnamed blocks; C contains unnamed blocks but not nested procedures; and Ada returns to support both.

Unnamed blocks are useful for restricting the scope of variables by declaring them only when needed, instead of at the beginning of a subprogram. The trend in programming is to reduce the size of subprograms, so the use of unnamed blocks is less useful than it used to be.

Nested procedures can be used to group statements that are executed at more than one location within a subprogram, but refer to local variables and so cannot be external to the subprogram. Before modules and object-oriented programming were introduced, nested procedures were used to structure large programs, but this introduces complications and is not recommended.

The following is an example of a complete Ada program:

Ada

```
procedure Main is
   Global: Integer;

   procedure Proc(Parm: in Integer) is
      Local: Integer;
   begin
      Global := Local + Parm;
   end Proc;

begin -- Main
   Global := 5;
   Proc(7);
   Proc(8);
end Main;
```

An Ada program is a *library* procedure, that is, a procedure that is not enclosed within any other entity and hence can be stored in the Ada library. The procedure begins with a procedure declaration for Main,[6] which serves as a definition of the interface to the procedure, in this case the

[6] Unlike C, the main procedure need not be called main.

external name of the program. Within the library procedure there are two declarations: a variable Global and a procedure Proc. Following the declarations is the sequence of executable statements for the main procedure. In other words, the procedure Main consists of a procedure declaration and a block. Similarly, the local procedure Proc consists of a procedure declaration (the procedure name and the parameters) and a block containing variable declarations and executable statements. Proc is said to be *local* to Main or *nested* within Main.

Each declaration has associated with it three properties:[7]

Scope The scope of a variable is the segment of the program within which it is defined.

Visibility A variable is visible within some subsegment of its scope if it can be directly accessed by name.

Lifetime The lifetime of a variable is the interval during the program's execution when memory is assigned to the variable.

Note that lifetime is a dynamic property of the run-time behavior of a program, while scope and visibility relate solely to the static program text.

Let us demonstrate these abstract definitions on the example above. The scope of a variable begins at the point of declaration and ends at the end of the block in which it is defined. The scope of Global includes the entire program while the scope of Local is limited to a single procedure. The formal parameter Parm is considered to be like a local variable and its scope is also limited to the procedure.[8]

The visibility of each variable in this example is identical to its scope: each variable can be directly accessed in its entire scope. Since the scope and visibility of the variable Local is limited to the local procedure, the following is not allowed:

Ada

```
begin -- Main
   Global := Local + 5;    -- Local is not in scope here
end Main;
```

However, the scope of Global includes the local procedure so the access within the procedure is correct:

Ada

```
procedure Proc(Parm: in Integer) is
   Local: Integer;
begin
   Global := Local + Parm;      -- Global is in scope here
end Proc;
```

The lifetime of a variable is from the beginning of the execution of its block until the end of the execution of its block. The block of the procedure Main is the entire program so Global exists for

[7] To keep the discussion concrete, it will be given in terms of variables even though the concepts are more general.

[8] Ada allows named parameter associations, so this statement is not completely precise.

the duration of the execution of the program. Such a variable is called *static*: once it is allocated it lives until the end of the program. The local variable has two lifetimes corresponding to the two calls to the local procedure. Since these intervals do not overlap, the variable may be allocated at a different location each time it is created. Local variables are called *automatic* because they are automatically allocated when the procedure is called (the block is entered), and released when the procedure returns (the block is left).

Hiding

Suppose that a variable name that is used in the main program is repeated in a declaration in a local procedure:

Ada

```
procedure Main is
   Global: Integer;
   V: Integer;                    -- Declaration in Main

   procedure Proc(Parm: in Integer) is
      Local: Integer;
      V: Integer;                 -- Declaration in Proc
   begin
      Global := Local + Parm + V; -- Which V is used ?
   end Proc;

begin -- Main
   Global := Global + V;          -- Which V is used ?
end Main;
```

In this case, the local declaration is said to *hide* the global declaration. Within the procedure, any reference to V is a reference to the locally declared variable. In technical terms, the scope of the global V extends from the point of declaration to the end of Main, but its visibility does not include local procedure Proc.[9]

Hiding of variable names by inner declarations is convenient in that the programmer can reuse natural names like Current_Key and not have to invent strange-sounding names. Furthermore, it is always possible to add a global variable without worrying that this will clash with some local variable name used by one of the programmers on your team. The disadvantage is that a variable name could be accidentally hidden, especially if large include-files are used to centralize global declarations, so it is probably better to avoid hiding variable names. However, there is no objection to reusing a name in different scopes since there is no way of accessing both variables simultaneously, regardless of whether the names are the same or different:

[9] In Ada (but not in Pascal) the hidden variable is accessible using the syntax Main.V. Similarly, in C++ (but not in C), ::V can be used to access a hidden global variable.

Ada

```
procedure Main is
   procedure Proc_1 is
      Index: Integer;              -- One scope
      ...
   end Proc_1;
   procedure Proc_2 is
      Index: Integer;              -- Non-overlapping scope
      ...
   end Proc_2;
begin -- Main
   ...
end Main;
```

Depth of nesting

There is no conceptual limit to the depth of nesting, though a compiler may arbitrarily limit the depth. Scope and visibility are determined by applying the rules given above: a variable's scope is from its point of declaration to the end of the block, and its visibility is the same unless hidden by an inner declaration. For example:

Ada

```
procedure Main is
   Global: Integer;

   procedure Level_1 is
      Local: Integer;       -- Outer declaration of Local

      procedure Level_2 is
         Local: Integer;    -- Inner declaration of Local
      begin -- Level_2
         Local := Global; -- Inner Local hides outer Local
      end Level_2;

   begin -- Level_1
      Local := Global;      -- Only outer Local in scope
      Level_2;
   end Level_1;

begin -- Main
   Level_1;
   Level_2;                 -- Error, procedure not in scope
end Main;
```

The scope of the variable Local defined in procedure Level_1 extends until the end of the procedure, but it is hidden within procedure Level_2 by the declaration of the same name.

The procedure declarations themselves are considered to have scope and visibility similar to variable declarations. Thus the scope of Level_2 is from its declaration in Level_1 until the end of Level_1. This means that Level_1 can *call* Level_2 even though it cannot access variables within Level_2. On the other hand, Main cannot call Level_2 directly, since it cannot access declarations that are local to Level_1.

Note the potential for confusion since the variable Local accessed by the statement in Level_1 is declared *further* away in the program text than the occurrence of Local enclosed within Level_2. If there were a lot of local procedures, it might be difficult to find the correct declaration. To prevent confusion, it is best to limit the depth of nesting to two or three levels below the main program.

Advantages and disadvantages of block structure

The advantage of block structure is that it provides an easy and efficient method of decomposing a procedure. If you avoid excessive nesting and hidden variables, block structure can be used to write reliable programs since related local procedures can be kept together. Block structuring is especially important when complex computations are being done:

Ada

```
procedure Proc(...) is
      -- Lots of declarations
begin
   -- Long computation 1
   if N ¡ 0 then
         -- Long computation 2 version 1
   elsif N = 0 then
         -- Long computation 2 version 2
   else
         -- Long computation 2 version 3
   end if;
   -- Long computation 3
end Proc;
```

In this example, we would like to avoid writing Long computation 2 three times and instead make it an additional procedure with a single parameter:

Ada

```
procedure Proc(...) is
      -- Lots of declarations
      procedure Long_2(I: in Integer) is
      begin
         -- Access declarations in Proc
      end Long_2;
begin
```

```
    -- Long computation 1
    if N ¡ 0 then Long_2(1);
    elsif N = 0 then Long_2(2);
    else Long_2(3);
    end if;
    -- Long computation 3
end Proc;
```

However, it would be extremely difficult to make Long_2 an independent procedure because we might have to pass dozens of parameters so that it could access local variables. If Long_2 is nested, it needs just the one parameter, and the other declarations can be directly accessed according to normal scope and visibility rules.

The disadvantages of block structure become apparent when you try to program a large system in a language like standard Pascal that has no other means of program decomposition:

- Small procedures receive excessive "promotions". Suppose that a procedure to convert decimal digits to hexadecimal digits is used in many deeply-nested procedures. That utility procedure must be defined in some common ancestor. Practically, large block-structured programs tend to have many small utility procedures written at the highest level of declaration. This makes the program text awkward to work with because it is difficult to locate a specific procedure.

- Data security is compromised. Every procedure, even those declared deeply nested in the structure, can access global variables. In a large program being developed by a team, this makes it likely that errors made by one junior team member can cause obscure bugs. The situation is analogous to a company where every employee can freely examine the safe in the boss's office, but the boss has no right to examine the file cabinets of junior employees!

These problems are so serious that every commercial Pascal implementation defines a (non-standard) module structure to enable large projects to be constructed. In Chapter 13 we will discuss in detail constructs that are used for program decomposition in modern languages like Ada and C++. Nevertheless, block structure remains an important tool in the detailed programming of individual modules.

It is also important to understand block structure because programming languages are implemented using stack architecture, which directly supports block structure (Section 7.6).

7.5 Recursion

Most (imperative) programming is done using *iteration*, that is loops; however, *recursion*, the definition of an object or computation in terms of itself, is a more primitive mathematical concept, and is also a powerful, if often under-used, programming technique. Here we will survey how to program recursive subprograms.

The most elementary example of recursion is the factorial function, defined mathematically as:

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)! \end{aligned}$$

This definition translates immediately into a program that uses a recursive function:

C

```
int factorial(int n)
{
    if (n == 0) return 1;
    else return n * factorial(n – 1);
}
```

What properties are required to support recursion?

- The compiler must emit *pure code*. Since the same sequence of machine instructions are used to execute each call to factorial, the code must not modify itself.
- During run-time, it must be possible to allocate an arbitrary number of memory cells for the parameters and local variables.

The first requirement is fulfilled by all modern compilers. Self-modifying code is an artifact of older programming styles and is rarely used. Note that if a program is to be stored in read-only memory (ROM), by definition it cannot modify itself.

The second requirement arises from consideration of the lifetime of the local variables. In the example, the lifetime of the formal parameter n is from the moment that the procedure is called until it is completed. But before the procedure is completed, another call is made and that call requires that memory be allocated for the new formal parameter. To compute factorial(4), a memory location is allocated for 4, then 3 and so on, five locations altogether. The memory cannot be allocated before execution, because the amount depends on the run-time parameter to the function. Section 7.6 shows how this allocation requirement is directly supported by the stack architecture.

At this point, most programmers will note that the factorial function could be written just as easily and far more efficiently using iteration:

C

```
int factorial(int n)
{
    int i = n;
    result = 1;
    while (i != 0) {
        result = result * i;
        i−−;
    }
    return result;
}
```

So why use recursion? The reason is that many algorithms can be elegantly and reliably written using recursion while an iterative solution is difficult to program and prone to bugs. Examples are the Quicksort algorithm for sorting and data structure algorithms based on trees. The language concepts discussed in Chapters 16 and 17 (functional and logic programming) use recursion exclusively instead of iteration. Even when using ordinary languages like C and Ada, recursion should probably be used more often than it is because of the concise, clear programs that result.

7.6 Stack architecture

A *stack* is a data structure that stores and retrieves data in a Last-In, First-Out (LIFO) order. LIFO constructions exist in the real world such as a stack of plates in a cafeteria, or a pile of newspapers in a store. A stack may be implemented using either an array or a list (Figure 7.5). The advantage of the list is that it is unbounded and its size is limited only by the total amount of

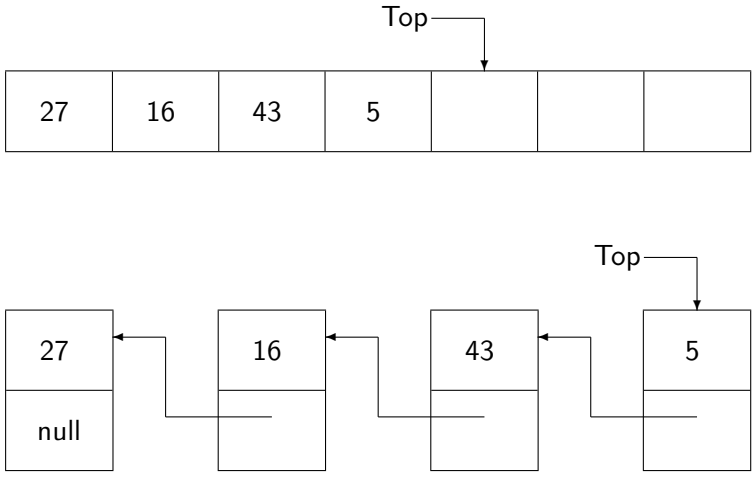

Figure 7.5: Stack implementation

available memory. Arrays are much more efficient and are implicitly used in the implementation of programming languages.

In addition to the array (or list), a stack contains an additional piece of data—the *top-of-stack pointer*. This is an index to the first available empty position in a stack. Initially, a variable top will point to the first position in the stack. The two possible operations on a stack are push and pop. push is a procedure that receives an element as a parameter, which it places on the top of the stack, incrementing top. pop is a function that returns the top element in the stack, decrementing top to indicate that that position is the new empty position.

The following C program implements a stack of integers as an array:

C

```
#define Stack_Size 100
int stack[Stack_Size];
```

```
int top = 0;

void push(int element)
{
    if (top == Stack_Size) /* Stack overflow, do something! */
    else stack[top++] = element;
}

int pop(void)
{
    if (top == 0) /* Stack underflow, do something! */
    else return stack[--top];
}
```

A stack can underflow if we try to pop from an empty stack, and it can overflow if we try to push onto a full stack. Underflow is always due to a programming error since you store something on a stack if and only if you intend to retrieve it later on. Overflow can occur even in a correct program if the amount of memory is not sufficient for the computation.

Stack allocation

How is a stack used in the implementation of a programming language? A stack is used to store information related to a procedure call, including the local variables and parameters that are automatically allocated upon entry to the procedure and released upon exit. The reason that a stack is the appropriate data structure is that procedures are entered and exited in a LIFO order, and any accessible data belongs to a procedure that occurs earlier in the chain of calls.

Consider a program with local procedures:

Ada

```
procedure Main is
    G: Integer;

    procedure Proc_1 is
        L1: Integer;
    begin ... end Proc_1;

    procedure Proc_2 is
        L2: Integer;
    begin ... end Proc_2;

begin
    Proc_1;
    Proc_2;
end Main;
```

When the Main begins executing, memory must be allocated for G. When Proc_1 is called, additional memory must be allocated for L1 without releasing the memory for G (Figure 7.6(a)). The memory for L1 is released before memory is allocated for L2, since Proc_1 terminates before

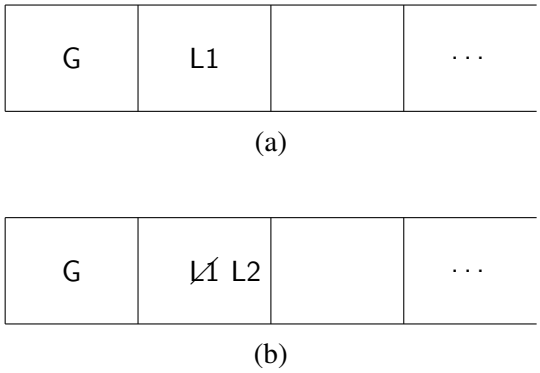

Figure 7.6: Allocating memory on a stack

Proc_2 is called (Figure 7.6(b)). In general, no matter how procedures call each other, the first memory element to be released is the last one allocated, so memory for variables and parameters can be allocated on a stack.

Consider now nested procedures:

Ada

```
procedure Main is
   G: Integer;

   procedure Proc_1(P1: Integer) is
      L1: Integer;

      procedure Proc_2(P2: Integer) is
         L2: Integer;
      begin
         L2 := L1 + G + P2;
      end Proc_2;

   begin -- Proc_1
      Proc_2(P1);
   end Proc_1;

begin -- Main
   Proc_1(G);
end Main;
```

Proc_2 can only be called from within Proc_1. This means that Proc_1 has not terminated yet, so its memory has not been released and the memory assigned to L1 must still be allocated (Fig-

ure 7.7). Of course, Proc_2 terminates before Proc_1 which in turn terminates before Main, so memory can be freed using the pop operation on the stack.

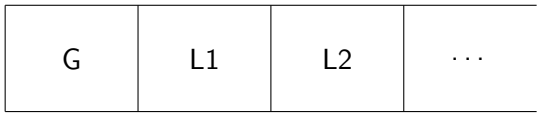

Figure 7.7: Nested procedures

Activation records

The stack is actually used to support the entire procedure call and not just the allocation of local variables. The segment of the stack associated with each procedure is called the *activation record* for the procedure. In outline,[10], a procedure call is implemented as follows (see Figure 7.8):

1. The actual parameters are pushed onto the stack. They can be accessed as offsets from the start of the activation record.
2. The *return address* is pushed onto the stack. The return address is the address of the statement following the procedure call.
3. The top-of-stack index is incremented by the total amount of memory required to hold the local variables.
4. A jump is made to the procedure code.

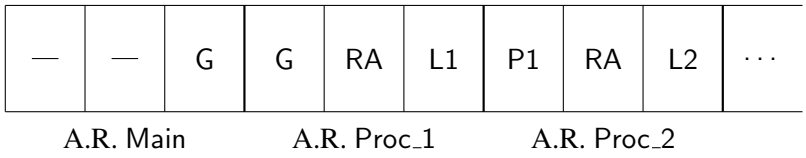

Figure 7.8: Activation records

Upon completion of the procedure the above steps are reversed:

1. The top-of-stack index is decremented by the amount of memory allocated for the local variables.
2. The return address is popped and the instruction pointer reset.
3. The top-of-stack index is decremented by the amount of memory allocated for the actual parameters.

[10] See Section 7.8 for more details.

While this code may seem complicated, it can actually be done very efficiently on most computers. The amount of memory needed for variables, parameters and call overhead is known at compile-time, and the above processing just requires modification of the stack index by a constant.

Accessing values on the stack

In a true stack, the only permissible operations are push and pop. The execution stack we have described is a more complex structure because we want to be able to efficiently access not only the most recent value pushed, but also all the local variables and all the parameters. One possibility would be to access these data relative to the top-of-stack index:

C

```
stack[top − 25];
```

However, the stack may hold other data besides that associated with a procedure call (such as temporary variables, see Section 4.7), so it is customary to maintain an additional index called the *bottom pointer* which points to the start of the activation record (see Section 7.7). Even if the top-of-stack index varies during the execution of the procedure, all the data in the activation record can be accessed at fixed offsets from the bottom pointer.

Parameters

There are two methods for implementing the passing of parameters. The simpler method is just to push the parameters themselves (whether values or references) onto the stack. This method is used in Pascal and Ada because in those languages the number and type of each parameter is known at compilation time. From this information, the offset of each parameter relative to the beginning of the activation record can be computed at compile-time, and each parameter can be accessed at this fixed offset from the bottom pointer index:

```
load    R1,bottom-pointer
add     R1,#offset-of-parameter
load    R2,(R1)         Load value whose address is in R1
```

If the bottom pointer is kept in a register, this code can usually be collapsed into a single instruction. When leaving the subprogram, *stack clean-up* is done by having the subprogram reset the stack pointer so that the parameters are effectively no longer on the stack.

There is a problem using this method in C, because C allows a procedure to have a variable number of arguments:

C

```
void proc(int num_args, ...);
```

Since the subprogram does not know how many parameters there are, it cannot clean-up the stack. The responsibility for stack clean-up is thus shifted to the caller which does know how many parameters were passed. This causes some memory overhead because the clean-up code is duplicated at *every* call instead of being common to all calls.

When the number of parameters is not known, an alternative method of parameter passing is to store the actual parameters in a separate block of memory, and then to pass the address of this block on the stack. An additional indirection is required to access a parameter, so this method is less efficient than directly pushing parameters on the stack.

Note that it may not be possible to store a parameter directly on the stack. As you will recall, a formal parameter in Ada can be an unconstrained array type whose bounds are not known at compilation time:

Ada

```
procedure Proc(S: in String);
```

Thus the actual parameter cannot be pushed directly onto the stack. Instead a dope vector (Figure 5.5) which contains a pointer to the array parameter is placed on the stack.

Recursion

The stack architecture directly supports recursion because each call to a procedure automatically allocates a new copy of the local variables and parameters. For example, each recursive call of the function for factorial needs one memory word for the parameter and one memory word for the return address. The higher overhead of recursion relative to iteration comes from the extra instructions involved with the procedure entry and exit. Some compilers will attempt an optimization called *tail-recursion* or *last-call* optimization. If the only recursive call in a procedure is the last statement in the procedure, it is possible to automatically translate the recursion into iteration.

Stack size

If recursion is not used, the total stack usage can theoretically be computed before execution by adding the activation record requirements for each possible chain of procedure calls. Even in a complex program, it should not be hard to make a reasonable estimate of this figure. Add a few thousand spare words and you have computed a stack size that will probably not overflow.

However, if recursion is used, the stack size is theoretically unbounded at run-time:

C

```
i = get( );
j = factorial(i);
```

In the Exercises, we describe Ackermann's function which is unconditionally guaranteed to overflow any stack you allocate! In practice, it is usually not difficult to make an estimate of stack size even when recursion is used. Suppose the size of an activation record is about 10 and the depth of recursion no more than a few hundred. Adding an extra 10K to the stack will more than suffice.

Readers who have studied data structures will know that recursion is convenient to use on tree-structured algorithms like Quicksort and priority queues. The depth of recursion in tree algorithms is roughly $\log_2$ of the size of the data structure. For practical programs this bounds the depth of recursion to 10 or 20 so there is very little danger of stack overflow.

Whether recursion is used or not, the nature of the system will dictate the treatment of potential stack overflow. A program might completely ignore the possibility and accept the fact that in extreme circumstances the program will crash. Another possibility is to check the stack size before each procedure call, but this might be too inefficient. A compromise solution would be to check the stack size periodically and take some action if it fell below some threshold, say 1000 words.

7.7 More on stack architecture

Accessing variables at intermediate levels

We have discussed how local variables are efficiently accessed at fixed offsets from the bottom pointer of an activation record. Global data, that is data declared in the main program, can also be accessed efficiently. The easiest way to see this is to imagine that global data is considered "local" to the main procedure; thus memory for the global data is allocated at the main procedure entry, that is, at the beginning of the program. Since this location is known at compile-time, or more exactly when the program is linked, the actual address of each element is known either directly or as an offset from a fixed location. In practice, global data is usually allocated separately (see Section 8.5), but in any case the addresses are fixed.

Variables at intermediate levels of nesting are more difficult to access:

Ada

```
procedure Main is
   G: Integer;

   procedure Proc_1 is
      L1: Integer;

      procedure Proc_2 is
         L2: Integer;
      begin L2 := L1 + G; end Proc_2;
      procedure Proc_3 is
         L3: Integer;
      begin L3 := L1 + G; Proc_2; end Proc_3;

   begin -- Proc_1
      Proc_3;
   end Proc_1;

begin -- Main
   Proc_1;
end Main;
```

We have seen that accessing the local variable L3 and the global variable G is easy and efficient,

but how can L1 be accessed in Proc_3? The answer is that the value of the bottom pointer is stored at procedure entry and is used as a pointer to the activation record of the enclosing procedure Proc_1. The bottom pointer is stored at a known location and can be immediately loaded, so the overhead is an additional indirection.

If deeper nesting is used, each activation record contains a pointer to the previous one. These pointers to activation records form the *dynamic chain* (Figure 7.9). To access a *shallow variable* (one that is less deeply nested), instructions have to be executed to "climb" the dynamic chain. This potential inefficiency is the reason that accessing intermediate variables in deeply nested procedures is discouraged. Accessing the immediately previous level requires just one indirection and an occasional deep access should not cause any trouble, but a loop statement should not contain statements that reach far back in the chain.

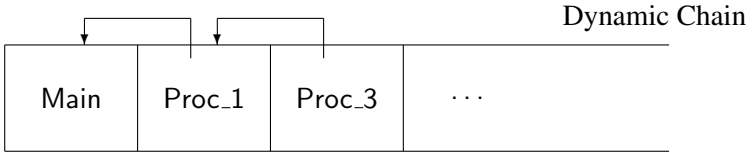

Figure 7.9: Dynamic chain

Calling shallow procedures

Accessing intermediate variables is actually more complicated than the discussion above would indicate because a procedure is allowed to call other procedures that are at the same level of nesting or lower. In the example, Proc_3 calls Proc_2. The activation record for Proc_2 will store the bottom pointer of Proc_3 so that it can be restored, but the variables of Proc_3 are *not* accessible in Proc_2 by the rules of scope.

Somehow the program must be able to identify the *static chain*, the link of activation records that defines the static context of the procedure according to the rules of scope, as opposed to the dynamic chain of procedure calls at execution time. As an extreme example, consider a recursive procedure: there may be dozens of activation records in the dynamic chain (one for each recursive call), but the static chain will consist only of the current record and the record for the main procedure.

One solution is to store the static level of nesting of each procedure in the activation record, because the compiler knows what level is needed for each access. In the example, if the main program is level 0, Proc_2 and Proc_3 are both at level 2. When searching up the dynamic chain, the level of nesting must decrease by one to be considered part of the static chain, thus the record for Proc_3 is skipped over and the next record, the one for Proc_1 at level 1, is used to obtain a bottom index.

Another solution is to explicitly include the static chain on the stack. Figure 7.10 shows the static chain just after Proc_3 calls Proc_2. Before the call, the static chain is the same as the dynamic chain while after the call, the static chain is shorter and contains just the main procedure and Proc_2.

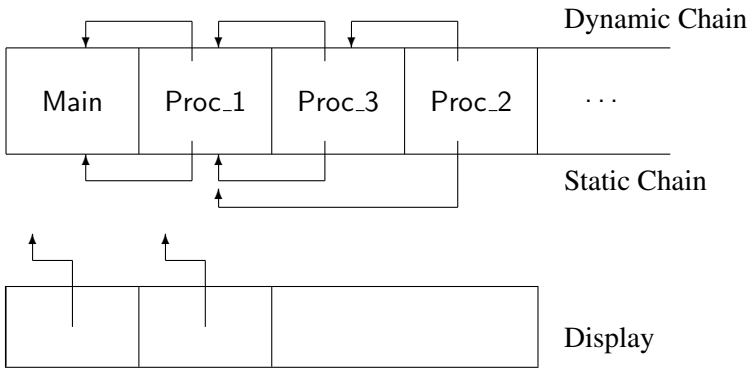

Figure 7.10: Variables at intermediate levels

The advantage of using an explicit static chain is that a static chain is often shorter than a dynamic chain (again think of a recursive procedure as an extreme case). However, we still have to do the search for each access of an intermediate variable. A solution that can be more efficient is to use a *display* which is an array that holds the current static chain, indexed by the nesting level (Figure 7.10). Thus to access a variable at an intermediate level, the nesting level is used as an index to obtain a pointer to the correct activation record, then the bottom pointer is obtained from the record and finally the offset is added to obtain the variable address. The disadvantage of a display is that additional overhead is needed to update the display at procedure entry and exit.

The potential inefficiencies of accessing intermediate variables should not deter the use of nested procedures, but programmers should carefully consider such factors as the depth of nesting and the trade-offs between using parameters as opposed to direct access to variables.

7.8 * Implementation on the 8086

To give a more concrete idea of how a stack architecture is implemented, we describe the actual machine code for procedure entry and exit on the Intel 8086 series of processors. The example program is:

Ada

```
procedure Main is
   Global: Integer;

   procedure Proc(Parm: in Integer) is
      Local1, Local2: Integer;
   begin
      Local2 := Global + Parm + Local1;
   end Proc;

begin
   Proc(15);
end Main;
```

The 8086 has built-in push and pop instructions which assume that the stack grows from higher to lower addresses. Two registers are dedicated to stack operations: the sp register which points to the "top" element in the stack, and the bp register which is the bottom pointer that identifies the location of the start of the activation record.

To call the procedure the parameter is pushed onto the stack and the call instruction executed:[11]

```
mov     ax,#15      Load value of parameter
push    ax          Store parameter on stack
call    Proc        Call the procedure
```

Figure 7.11 shows the stack after executing these instructions—the parameter and the return address have been pushed onto the stack.

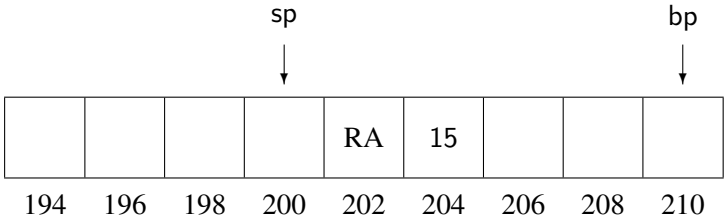

Figure 7.11: Stack before procedure entry

The next instructions are part of the code of the procedure and are executed at procedure entry; they store the old bottom pointer (the dynamic link), set up the new bottom pointer and allocate memory for the local variable by decrementing the stack pointer:

```
push    bp          Save the old dynamic pointer
mov     bp,sp       Set the new dynamic pointer
sub     sp,#4       Allocate the local variables
```

The stack now appears as shown in Figure 7.12.

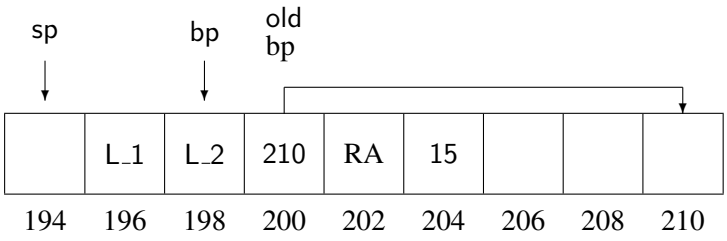

Figure 7.12: Stack after procedure entry

Now the body of the procedure can be executed:

[11] The extra mov instruction is there because stack operations work only to and from registers, in this case ax.

```
mov     ax,ds:[38]      Load the variable Global
add     ax,[bp+06]      Add the parameter Parm
add     ax,[bp-02]      Add the variable Local1
mov     ax,[bp]         Store in the variable Local2
```

Global variables are accessed as offsets from a special area of memory pointed to by the ds (data segment) register. The parameter Parm which is "lower" down in the stack than the start of the activation record is accessed at a *positive* offset from bp. The local variables which are "higher" up in the stack are accessed at a *negative* offset from bp. What is important to note is that since the 8086 processor has registers and addressing modes designed for common stack-oriented computations, all these variables can be accessed in a single instruction.

Procedure exit must reverse the effects of the procedure call:

```
mov     sp,bp           Release all local variables
pop     bp              Restore the old dynamic pointer
ret     2               Return and release parameters
```

The stack pointer is reset to the value of the bottom pointer thus effectively releasing memory allocated for the local variables. Then the old dynamic pointer is popped from the stack so that bp now points at the previous activation record. The only remaining tasks are to return from the procedure using the return address, and to release the memory allocated for the parameters. The ret instruction performs both of these tasks; the operand of the instruction indicates how many bytes of parameter memory must be popped from the stack. To summarize: procedure exit and entry require just three short instructions each, and access to local and global variables and to parameters is efficient.

7.9 Exercises

1. Does your Ada compiler use value or reference semantics to pass arrays and records?
2. Show how last-call optimization is implemented. Can last-call optimization be done on the factorial function?
3. McCarthy's function is defined by the following recursive function:

   Ada

   ```
   function M(I: Integer) return Integer is
   begin
      if I > 100 then return I-10;
      else return M(M(I+11));
   end M;
   ```

   (a) Write a program for McCarthy's function and compute M(I) for $80 \leq I \leq 110$.
   (b) Simulate by hand the computation for M(91) showing the growth of the stack.
   (c) Write an iterative program for McCarthy's function.
4. Ackermann's function is defined by the following recursive function:

Ada

```
function A(M, N: Natural) return Natural is
begin
   if M = 0 then return N + 1;
   elsif N = 0 then return A(M – 1, 1);
   else return A(M – 1, A(M, N – 1));
end A;
```

(a) Write a program for Ackermann's function and check that A(0,0)=1, A(1,1)=3, A(2,2)=7, A(3,3)=61.

(b) Simulate by hand the computation for A(2,2)=7 showing the growth of the stack.

(c) Try to compute A(4,4) and describe what happens. Try the computation using several compilers. Make sure you save your files before doing this!

(d) Write a non-recursive program for Ackermann's function.[12]

5. How are variables of intermediate scope accessed on an 8086?

6. There is a parameter passing mechanism called *call-by-name* in which each access of a formal parameter causes the actual parameter to be re-evaluated. This mechanism was first used in Algol but does not exist in most ordinary programming languages. What was the motivation for call-by-name in Algol and how was it implemented?

[12]Solutions can be found in: Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974, p. 201 and p. 235.

9 Real Numbers

9.1 Representations of real numbers

In Chapter 4 we discussed how integer types are used to represent a subset of the mathematical integers. Computation with integer types can cause overflow—a concept which has no meaning for mathematical integers—and the possibility of overflow means that the commutativity and associativity of arithmetical operations is not ensured during the computation.

Representation of real numbers on computers and computation with their representations are extremely problematical, to the point where a specialist should be consulted during the construction of critical programs. This chapter will explore the basic concepts of computation with real numbers; the superficial ease with which real-number computations can be written in a program must not obscure the underlying problems.

First of all let us note that decimal numbers cannot be accurately represented in binary notation. For example, 0.2 (one-fifth) has no exact representation as a binary number, only the repeating binary fraction:

$$0.0011001100110011\ldots$$

There are two solutions to this problem:

- Represent decimal numbers directly, for example by assigning four bits to each decimal numeral. This representation is called *binary-coded decimal* (*BCD*).
- Store binary numbers and accept the fact that some loss of accuracy will occur.

BCD wastes a certain amount of memory because four bits could represent 16 different values and not just the 10 needed for decimals. A more important disadvantage is that the representation is not "natural" and calculation with BCD is much slower than with binary numbers. Thus the discussion will be limited to binary representations; the reader interested in BCD computation is referred to languages such as Cobol which support BCD numbers.

Fixed-point numbers

To simplify the following discussion, it will be given in terms of decimal numbers, but the concepts are identical for binary numbers. Suppose that we can represent seven digits, five before and two after the decimal point, in one 32-bit memory word:

$$12345.67, \quad -1234.56, \quad 0.12$$

This representation is called *fixed-point*. The advantage of fixed-point numbers is that the *accuracy*, that is the absolute error, is fixed. If the above numbers denote dollars and cents, then any error caused by the limited size of a memory word is at most one cent. The disadvantage is that the *precision*, that is the relative error which is the number of significant digits, is variable. The first number uses all available seven digits of precision, while the last uses only two. More importantly, the varying precision means that many important numbers such as the $\$1,532,854.07$ that you won in the lottery, or your \$0.00572 income-tax refund, cannot be represented at all.

Fixed-point numbers are used in applications where total accuracy is essential. For example, accounting computations are usually done in fixed-point, since the required precision is known in advance (say 12 or 16 digits) and the account must balance to the last cent. Fixed-point numbers are also used in control systems where sensors and actuators communicate with the computer in fixed-length words or fields. For example, speed could be represented in a 10-bit field with range 0 to 102.3 km/hour; one bit will represent 0.1 km/hour.

Floating-point numbers

Scientists who have to deal in a wide range of numbers use a convenient notation called *scientific notation*:

$$123.45 \times 10^3, \;\; 1.2345 \times 10^{-8}, \;\; -0.00012345 \times 10^7, \;\; 12345000.0 \times 10^4$$

How can we use this concise notation on a computer? First note that there are three items of information that have to be represented: the *sign*, the *mantissa* (123.45 in the first number) and the *exponent*. On the surface it would seem that there is no advantage to representing a number in scientific notation, because of the varying precisions needed to represent the mantissa: five digits in the first and second numbers above, as opposed to eight digits for the other two.

The solution is to note that trailing zero digits of a mantissa greater than 1.0 (and leading zero digits of a mantissa less than 1.0) can be discarded in favor of changes in the value (not the precision!) of the exponent. More precisely, the mantissa can be multiplied or divided repeatedly by 10 until it is in a form that uses the maximum precision; each such operation requires that the exponent be decremented or incremented by 1, respectively. For example, the last two numbers can be written to use a five-digit mantissa:

$$-0.12345 \times 10^4, \;\; 0.12345 \times 10^{12}$$

It is convenient for computation on a computer if every such number has this standard form, called *normalized* form, where the first non-zero digit is the "tenth's" digit. This also saves on space in the representation, because the decimal point is always in the same position and need not be explicitly represented. The representation is called *floating-point* because we require that the decimal point "float" left or right until the number can be represented with maximum precision.

What is the main disadvantage of computation with floating-point numbers? Consider 0.12345×10^{10} which is normalized floating-point for

$$1,234,500,000$$

and suppose that this is how the bank represented your deposit of

$$\$1,234,567,890$$

Your bank manager would be proud that the *relative error*:

$$\frac{67,890}{1,234,567,890}$$

is a very small fraction of one percent, but you would justifiably demand your $\$67,890$, the *absolute error*.

In scientific calculations relative error is much more important than absolute error. In a program that controls the speed of a rocket, the requirement may be that the error be no more than 0.5% even though this translates into a few km/h during launch and a few hundred km/h nearing orbit. Floating-point computation is much more common than fixed-point, because relative accuracy is required more often than absolute accuracy. For this reason most computers have hardware that directly implements floating-point computation.

Representation of floating-point numbers

Floating-point numbers are stored as *binary* numbers in the normalized form we have described:

$$-0.101100111 \times 2^{15}$$

A typical implementation on a 32-bit computer would assign 1 bit to the sign, 23 bits to the mantissa and 8 bits to the exponent. Since it takes $\log_2 10 \approx 3.3$ bits to store one decimal digit, $23/3.3 \approx 7$ digits of precision are obtained. If more precision is needed, a 64-bit double word with a 52-bit mantissa can achieve about 15 digits of precision.

There is a "trick" that is used to represent more numbers than would otherwise be available. Since all floating-point numbers are normalized, and since the first digit of a normalized mantissa is necessarily 1, this first digit need not be explicitly represented.

The signed exponent is represented by *biasing* the exponent so that it is always positive, and the exponent is placed in the high-order bits of a word next to the sign bit. This makes comparisons easier because ordinary integer comparison may be done without extracting a signed two's-complement exponent field. For example, an 8-bit exponent field with values in the range 0..255 represents exponents in the range $-127..128$ with a bias of 127.

We can now decipher a bit string as a floating point number. The string:

$$1 \;\; 1000\,1000 \;\; 0110\,0000\,0000\,0000\,0000\,000$$

is deciphered as follows:

- The sign bit is 1 so the number is negative.
- The exponent is $1000\,1000 = 128 + 8 = 136$. Removing the bias gives

$$136 - 127 = 9$$

- The mantissa is $0.10110\ldots$ (note the hidden bit restored), which is
$$\frac{1}{2}+\frac{1}{8}+\frac{1}{16}=\frac{11}{16}$$
- Thus the number stored is $2^9 \times 11/16 = 352$.

Just as with integers, floating point overflow occurs when the result of the computation is too large:

$$(0.5\times 10^{70})\cdot(0.5\times 10^{80}) = 0.25\times 10^{150}$$

Since the largest exponent that can be represented is 128, the computation overflows.

Consider now the computation:

$$(0.5\times 10^{-70})\cdot(0.5\times 10^{-80}) = 0.25\times 10^{-150}$$

The computation is said to *underflow* because the result is too small to be represented. You may claim that such a number is so small that it might as well be zero, and a computer may choose to treat underflow by storing a zero result, but underflow does represent an error and should be either handled or accounted for.

9.2 Language support for real numbers

All programming languages have support for floating-point calculations. Variables can be declared to be of type float,[1] and floating-point literals are represented using a form similar to scientific notation:

C

```
float f1 = 7.456;
float f2 = -46.64E-3;
```

Note that the literals need not be in binary notation or in normalized form; the conversion is done by the compiler.

A minimum of 32 bits is needed for meaningful floating-point calculations. Often more precision is needed, and languages support declaration and computation with higher precisions. At a minimum *double-precision* variables using 64 bits are supported, and some computers or compilers will support even longer types. Double precision floating-point types are called double in C and Long_Float in Ada.

The notation for double-precision literals varies with the language. Fortran uses a separate notation, replacing the E that precedes the exponent with a D: −45.64D−3. C chooses to store *every* literal in double precision, allowing a suffix F if you want to specify single precision. Be careful if you are storing a large array of floating-point constants.

Ada introduces a new concept, *universal types*, to handle the varying precision of literals. A literal such as 0.2 is stored by the compiler in potentially infinite precision (recall that 0.2 cannot be exactly represented as a binary number). When a literal is actually used, it is converted into a constant of the correct precision:

[1] Pascal and Fortran use the name real.

Ada

```
PI_F:  constant Float      := 3.1415926535;
PI_L:  constant Long_Float := 3.1415926535;
PI:    constant            := 3.1415926535;
F: Float      := PI;       -- Convert number to Float
L: Long_Float := PI;       -- Convert number to Long_Float
```

The first two declarations declare constants of the named types. The third declaration for PI is called a *named number* and is of universal real type. When PI is actually used in the initializations, it is converted to the correct precision.

The four arithmetic operators (+, −, * and /) as well as equality and the relational operators are defined for floating-point types. Mathematical functions such the trigonometric functions may be defined within the language (Fortran and Pascal), or they be supplied by subprogram libraries (C and Ada).

Portability of floating-point

Programs that use floating-point can be difficult to port because different definitions are used for the type specifiers. There is nothing to prevent a compiler for C or Ada from using 64 bits to represent float (Float) and 128 bits to represent double (Long_Float). Both porting directions are problematical: when porting from an implementation which uses a high-precision representation of float to one which uses a low-precision representation, all float's must be converted to double's to retain the same level of precision. When porting from a low-precision to a high-precision implementation, the opposite modification may be needed, because using excess precision wastes execution time and memory.

An elementary partial solution is to declare and use an artificial floating-point type; then only a few lines need be changed when porting the program:

```
typedef double Real;        /* C */
subtype Real is Long_Float;-- Ada
```

See Section 9.4 for Ada's comprehensive solution to the problem of portable computation with real numbers.

Hardware and software floating-point

Our discussion of the representation of floating-point numbers should have made it clear that arithmetic on these values is a complex task. The words must be decomposed, the exponent biases removed, the multiword arithmetic done, the result normalized and the result word composed. Most computers use special hardware for efficient execution of floating-point calculations.

A computer without the appropriate hardware can still execute floating-point calculations using a library of subprograms that *emulate* floating-point instructions. Attempting to execute a floating-point instruction will cause a "bad-instruction" interrupt which will be handled by calling the

appropriate emulation. Needless to say this can be very inefficient, since there is the interrupt and subprogram overhead in addition to the floating-point computation.

If you think that your program may be extensively used on computers without floating-point hardware, it may be prudent to avoid floating-point computation altogether. Instead you can explicitly program fixed-point calculations where needed. For example, a financial program can do all its computation in terms of "cents" rather than fractions of dollars. Of course this runs the risk of overflow if the Integer or Long_Integer types are not represented with sufficient precision.

Mixed arithmetic

Mixed integer and real arithmetic is very common in mathematics: we write $A = 2\pi r$ and not $A = 2.0\pi r$. When computing, mixed integer and floating-point operations must be done with some care. The second form is preferable because 2.0 can be stored directly as a floating-point constant, whereas the literal 2 would have to be converted to a floating representation. While this is usually done automatically by the compiler, it is better to write exactly what you require.

Another potential source of difficulty is the distinction between integer division and floating-point division:

Ada

```
I: Integer := 7;
J: Integer := I / 2;
K: Integer := Integer(Float(I) / 2.0);
```

The expression in the assignment to J calls for integer division; the result, of course, is 3. In the assignment to K, floating-point division is required: the result is 3.5 and it is converted to an integer by rounding to 4.

Languages even disagree on how to convert floating-point values to integer values. The same example in C:

C

```
int i = 7;
int j = i / 2;
int k = (int) ((float i) / 2.0);
```

assigns 3 to both j and k, because the floating-point value 3.5 is truncated rather than rounded!

C implicitly performs mixed arithmetic, converting integers to floating-point types if necessary and converting lower precision to higher precision. Also, values are implicitly converted upon assignment. Thus the above example could be written:

C

```
int k = i / 2.0;
```

While the *promotion* of the integer i to floating-point is clear, programs will be more readable if explicit conversions are used on assignment statements (as opposed to initializations):

```
k = (int) i / 2.0;
```
C

Ada forbids *all* mixed arithmetic; however, any value of a numeric type can be explicitly converted from a value of any other numeric type as shown above.

If efficiency is important, rearrange a mixed expression so that the computation is kept as simple as possible for as long as possible. For example (recalling that literals are considered double in C):

```
int i, j, k, l;
float f = 2.2 * i * j * k * l;
```
C

would be done by converting i to double, performing the multiplication 2.2 * i and so on, with every integer being converted to double. Finally, the result would be converted to float for the assignment. It would be more efficient to write:

```
int i, j, k, l;
float f = 2.2F * (i * j * k * l);
```
C

to ensure that the integer variables are first multiplied using fast integer instructions and that the literal be stored as float and not as double. Of course, these optimizations could introduce integer overflow which would not occur if the computation was done in double precision.

One way to improve the efficiency of any floating-point computation is to arrange the algorithm so that only part of the computation has to be done in double precision. For example, a physics problem can use single precision when computing the motion of two objects that are close to each other (so the distance between them can be precisely represented in relatively few digits); the program can then switch to double precision as the objects get further away from each other.

9.3 The three deadly sins

Every floating-point operation produces a result that is possibly incorrect in the least significant digit because of rounding errors. Programmers writing numerical software should be well-versed in the methods for estimating and controlling these errors. In these paragraphs, we will summarize three serious errors that can occur:

- Negligible addition
- Error magnification
- Loss of significance

Negligible addition occurs when adding or subtracting a number that is very small relative to the first operand. In five-digit decimal arithmetic:

$$0.1234 \times 10^3 + 0.1234 \times 10^{-4} = 0.1234 \times 10^3$$

It is unlikely that your high-school teacher taught you that $x+y=x$ for non-zero y, but that is what has occurred here!

Error magnification is the large absolute error that can occur in floating-point arithmetic even though the relative error is small. It is usually the result of multiplication or division. Consider the computation of $x \cdot x$:

$$0.1234 \times 10^3 \cdot 0.1234 \times 10^3 \; = \; 0.1522 \times 10^5$$

and suppose now that during the calculation of x, there had been a one-digit error, that is an absolute error of 0.1:

$$0.1235 \times 10^3 \cdot 0.1235 \times 10^3 = 0.1525 \times 10^5$$

The error is now 30 which is 300 times the size of the error before the multiplication.

The most serious error is complete loss of significance caused by subtracting nearly equal numbers:

C

```
float f1 = 0.12342;
float f2 = 0.12346;
```

In mathematics, $f2-f1=0.00004$ which is certainly representable as a four-digit floating-point number: 0.4000×10^{-4}. However, a program computing f2–f1 in four-digit floating-point will give the answer:

$$0.1235 \times 10^0 \; - \; 0.1234 \times 10^0 \; = \; 0.1000 \times 10^{-3}$$

which is not even close to being a reasonable answer.

Loss of significance is more common than might be supposed because equality calculations are usually implemented by subtracting and then comparing to zero. The following if-statement is thus totally unacceptable:

C

```
f1 = ... ;
f2 = ... ;
if (f1 == f2) ...
```

The most innocent rearrangement of the expressions for f1 and f2, whether done by a programmer or by an optimizer, can cause a different branch of the if-statement to be taken. The correct way of checking equality in floating-point is to introduce a small error term:

C

```
#define Epsilon 10e–20

if ((fabs(f2 – f1)) ¡ Epsilon) ...
```

and then compare the absolute value of the difference to the error term. For the same reason, there is no effective difference between ¡= and ¡ in floating-point calculation.

Errors in floating-point calculations can often be reduced by rearrangement. Since addition associates to the left, the four-digit decimal computation:

$$1234.0+0.5678+0.5678=1234.0$$

is better if done:

$$0.5678 + 0.5678 + 1234.0 = 1235.0$$

to avoid negligible addition.

As another example, the arithmetic identity:

$$(x+y)(x-y) = x^2 - y^2$$

can be used to improve the accuracy of a computation:[2]

Ada

```
X, Y: Float_4;
Z: Float_7;

Z := Float_7( (X+Y)*(X-Y) );          -- This way ?
Z := Float_7( X*X - Y*Y );            -- or this way ?
```

If we let $x = 1234.0$ and $y = 0.6$, the correct value of this expression is 1522755.64. Evaluated to eight digits, the results are:

$$(1234.0 + 0.6) \cdot (1234.0 - 0.6) \;=\; 1235.0 \cdot 1233.0 \;=\; 1522755.0$$

and

$$(1234.0 \cdot 1234.0) - (0.6 \cdot 0.6) \;=\; 1522756.0 - 0.36 \;=\; 1522756.0$$

When $(x+y)(x-y)$ is evaluated, the small error resulting from the addition and subtraction is greatly modified by the magnification caused by the multiplication. By rearranging, the formula $x^2 - y^2$ avoids the negligible addition and subtraction and gives a more correct answer.

9.4 * Real types in Ada

Note: the technical definition of real types was significantly simplified in the revision of Ada from Ada 83 to Ada 95, so if you intend to study the details of this subject, it would be best to skip the older definition.

Floating-types in Ada

In Section 4.6 we described how an integer type can be declared to have a given range, while the implementation is chosen by the compiler:

```
type Altitude is range 0 .. 60000;
```

Similar support for portability of floating-point computations is given by the declaration of arbitrary floating-point types:

[2] Float_4 and Float_7 are assumed to be types with four and seven digit precision, respectively.

```
type F is digits 12;
```

This declaration requests 12 (decimal) digits of precision. On a 32-bit computer this will require double precision, while on a 64-bit computer single precision will suffice. Note that as with integer types, this declaration creates a new type which cannot be used in operations with other types without explicit conversions.

The Ada standard describes in great detail conforming implementations of such a declaration. Programs whose correctness depends only on the requirements of the standard, and not on any quirks of a particular implementation, are assured of easy porting from one Ada compiler to another, even to a compiler on an entirely different computer architecture.

Fixed-point types in Ada

A fixed-point type is declared as follows:

```
type F is delta 0.1 range 0.0 .. 1.0;
```

In addition to a range, when writing a fixed-point type declaration you indicate the absolute error required by writing a fraction after the keyword delta.

Given delta D and a range R, an implementation is required to supply a set of numbers, called *model numbers*, at most D from one another which cover R. On a binary computer, the model numbers would be multiples of a power of two just below the value of D, in this case $1/16 = 0.0625$. The model numbers corresponding to the above declaration are:

$$0,\ 1/16,\ 2/16,\ \ldots,\ 14/16,\ 15/16$$

Note that even though 1.0 is specified as part of the range, that number is not one of the model numbers! The definition only requires that 1.0 be within 0.1 of a model number, and this requirement is fulfilled because $15/16 = 0.9375$ and $1.0 - 0.9375 < 0.1$.

There is a predefined type Duration which is used for measuring time intervals. Fixed-point is appropriate here because time will have an absolute error (say 0.0001 second) depending on the hardware of the computer.

For business data processing, Ada 95 defines *decimal fixed-point types*:

```
type Cost is delta 0.01 digits 10;
```

Unlike ordinary fixed-point types which are represented by powers of two, these numbers are represented by powers of ten and so are suitable for exact decimal arithmetic. The type declared above can hold values up to $99,999,999.99$.

9.5 Exercises

1. What floating-point types exist on your computer? List the range and precision for each one. Is exponent bias used? Normalization? Hidden high-order bit? What infinite or other unusual values exist?

2. Write a program that takes a floating-point number and prints the sign, mantissa and exponent (after removing any bias).
3. Write a program for infinite precision integer addition and multiplication.
4. Write a program to print the binary representation of a decimal fraction.
5. Write a program for BCD arithmetic.
6. Write a program to emulate floating-point addition and multiplication.
7. Declare various fixed-point types in Ada and check how values are represented. How is Duration represented?
8. In Ada there are limitations on fixed-point arithmetic. List and motivate each limitation.