• From Code to Execution

The purpose of this division is to track how a program written in a high-level programming language ends up running on a machine. We begin with a survey of the most important languages for scientific computing — Fortran 77, Fortran 95, C, and C++. To become a running process, a code must be compiled, assembled, and loaded, which is the subject of the second and third sections of this division. A running process, however, must coordinate a number of tasks implicit in the original code — for example, invoking subprograms and managing memory allocation.

Before we start, it will pay to set the background and terminology that informs the entire division. The following is a list of the stages by which a program in a high-level language becomes an executing program. For purposes of illustration, we will use Unix extension conventions.

High-level code with preprocessors statements. This is the starting point for a program. It is organized in files with extensions specifying the language; e.g., `.c` or `.f95`. The files are sent a **preprocessor** to produce

Pure high-level code. These preprocessed files contain only code in the high-level languages in question. They are passed to a **compiler** that produces

Assembly language code. We have discussed assembly language briefly in §3.2. Assembly language files have the extension `.s`. An assembly language file is passed to an **assembler** to produce

Machine language code. This is code in machine language but with cross references to other programs and library routines unresolved. Files containing such code are called **object** files and often have the extension `.o`. The several object files that compose a program are passed to a **linker** to produce

An executable program. This file can have any name (on Unix systems the default is usually `a.out`). On command the operating system will invoke a **loader** to move the file into virtual memory and will start its execution to produce

A running process. During its execution, the process is supported by a **run-time environment,** which, among other things, links the process dynamically to **shared libraries** and aids in storage allocation and deallocation.

Of course, this list is a simplification. For example, some languages do not have preprocessors. Again, some compilers may jump directly to machine language code or may produce code for a virtual machine whose execution is performed by a simulation program for a particular machine. Finally, as we go down the list, the items become increasingly dependent on the **run-time environment** — the collection of conventions and routines that are particular to the machine and its operating system. In spite of these exceptions, the list represents the typical passage from high-level code to a running program.

6. Languages

In this section we are going to treat four high-level programming language that are widely used in scientific computing: Fortran 77, Fortran 95, C, and C++. The purpose of this treatment is not to make you a proficient programmer in each of these language — that would be a long undertaking indeed. Instead it is intended to acquaint you with the main features of the languages and their relevance to scientific computing. We will begin with a discussion of the features more or less common to all four languages and then turn to treatments of the individual languages.

A notable absence in this section is a treatment of I/O. All our languages have excellent I/O constructs that can accommodate the needs of most scientific programs. However, the price for this flexibility is a steep learning curve, one that we cannot climb in this book. Fortunately, there are many excellent treatments of I/O in Fortran and C.

6.1. Commonalities

Synopsis: The four programming languages we will study have many common features. Although they differ somewhat in format, programs written in them have a similar appearance. They all work with typed variables such as integers, characters, and floating-point numbers, which can also be arranged in arrays. Variables can be combined in arithmetic and logical expressions, and their values can be assigned to other variables. Flow through a program is mediated by control constructs such as the if constructs and switches as well as constructs for looping.

The languages allow separately compiled subprograms to perform specialized tasks. Information is passed to subprograms through an argument list by either a call-by-value or a call-by-name protocol. New, internal variables can be defined in a subprogram. Static internal variables retain their values between subprogram invocations. Global variables can be accessed by all subprograms. Except in Fortran 77, subprograms can invoke themselves recursively.

Again excepting Fortran 77, our languages provide the wherewithal to obtain extra memory on demand — either from the stack or from the heap. They all have a preprocessor that can define constants and conditionally modify the program itself. They all come with standard libraries to provide functionality not in the language itself — for example, to evaluate mathematical functions.

—

It is customary to associate high-level programming languages with the Tower of Babble and the discordant languages that God visited on a prideful humanity. And indeed, the world of programming languages has been populated by a bewildering variety of tongues, many of which are deservedly extinct. But the four languages of scientific computing treated in this section — Fortran 77 Fortran 95, C and C++ — have as much in common as not. It will therefore simplify this section if we begin with a treatment of their common aspects, in which we also discuss some of their differences. This will also give us an opportunity to introduce in one place some of the concepts and nomenclature from programming languages. For the sake of brevity the unqualified term 'Fortran' will mean both Fortran 77 and Fortran 95. The term 'C family' will mean C and C++. We will refer to the totality as 'our languages'.

6.1.1. Format

Although programs written in our languages have strikingly different appearances, they all consist of sequences of characters divided into statements, which in turn consist of lexical units — e.g., names, operators, and delimiters. In many instances simple rules map the lexical units from one language to another, so that familiarity with one makes it easy to read another. Unfortunately, ease of reading does not extend to active programming.

A important difference between Fortran and C families is that Fortran uses line endings to delimit statements while C uses explicit delimiters. In practice, however, programmers tend to use standard indenting and line-breaking conventions, so that programs in Fortran and the C family have a similar overall appearance.

Another difference is in the matter of reserved words. Programming languages use two kinds of words. **Keywords** are words like `if` or `return` that are part of the language itself. **Identifiers** are words generated by the programmer as names for variables, functions, files, etc. In the C family keywords are **reserved** in the sense that they cannot be used as identifiers. In Fortran, on the contrary, there are no reserved words. Keywords are distinguished from identifiers by context. However, the practical difference between Fortran and the C family in this regard is not great. Using `return` and other syntactically meaningful words as variable names is to invite confusion, and good Fortran programmers seldom do it.

6.1.2. Data types and specification statements

All our languages have the following data types: integer, single and double-precision floating-point, logical (Fortran only), and character. The C family has a richer variety of integers. Fortran supports floating-point complex numbers, which the C family does not.n The Fortran character type is really a string of characters in the sense of §2.2, whereas in the C family it is an ordinary character. None of our languages commit themselves to a particular character code.

The languages provide constants for all types. Their forms vary in minor ways from language to language.

All types can be declared as **arrays** of various sizes and dimensions. However, array usage is quite different in Fortran and the C family, and we will treat them in detail in the subsections for the individual languages.

In all but Fortran 77 it is possible to define ensembles of types, called **derived types** in Fortran 95 and **structures** in the C family. Instances of these ensembles can be assigned to variables. Thus we might define an ensemble `point` consisting of two double precision numbers representing the x and y coordinates of a point in the plane. We can then declare variables like `A`, `B`, or `endpoint` to be `point`'s. We will treat these ensembles in the subsections for the individual languages.

All but Fortran 77 provide **pointers,** that contain the addresses of other variables. As we will see later, pointers are very powerful but also dangerous; and they can interfere with the compiler optimization of a program.

All our languages have **specification statements** or **declarations** that associate a data type with a variable — a process called **typing.** A peculiarity of Fortran that a variable that is not explicitly typed may be typed implicitly by the compiler. The use of this feature is deprecated. Fortran 95 provides an elaborate mechanism to specify the precision and other properties of its types.

6.1.3. Expressions

Expressions are build up of operators acting on constants and variables. All our languages support the **arithmetic operations** of addition, subtraction, multiplication, and division. The operators are the usual ones (`+, -, *, /`). Fortran has an exponentiation operator (`**`), and the C family has a modulus operator (`%`). The basic four operators have the usual **precedence** with `*` and `/` evaluated before `+` and `-`. For example, `b+c*d` evaluates as `b+(c*d)`, not `(b+c)*d`. However, parentheses can be used to override precedence. The basic four evaluate from left to right; e.g., `a+b+c` evaluates as `(a+b)+c`.

When the operands of an operation are of different type, the operand of a 'lower' type (e.g., integer) is converted to the operand of the 'higher' type (e.g., floating-point) before the operation is evaluated.

The four language provide **relational operations**. For example, in Fortran the expression `A.LT.B` evaluates to `.TRUE.` if `A` is less than `B` and to `.FALSE.` otherwise. In C the corresponding statement is `a<b`, which evaluates to `0` if `a` is less than `b` and to `1` otherwise. Note that the C family regards `0` as 'false' and everything else as 'true'.

Logical operations can be used to combine relational operations. For example in Fortran `A.LE.B. .AND. B.LT.C` is `.TRUE.` if and only if $\mathtt{A} \leq \mathtt{B} < \mathtt{C}$. The corresponding C statement is `a<=b && c<d`.

All our languages provide operations and functions for manipulating characters and strings.

In general the C family has a richer set of operations than Fortran. For example, `x++` increments `x` by one. Hence the name C++ for the object oriented extension of C. Many of the operations of the C family — for example, shifting — are performed by intrinsic functions in Fortran.

6.1.4. Assignment

An expression may be **assigned** to a variable, thus changing the value of the variable. The basic assignment operator in all four languages is `=`, although the C family has additional forms. When the type of the variable differs from the type of the expression, the latter is converted to the type of the former. Conversion of a floating-point number to integer — say, `i=x` — always truncates the number. Note that this conversion results in a loss of precision when `x` does not have an integer value. The converse assignment `x=i` can also result is a loss of precision if `i` is too large.

6.1.5. Control constructs

Programs in our languages are executed one statement after another. This sequential execution is insufficient for all but the simplest tasks because it does not allow for the behavior of a program to depend on the values of its variables. Consequently, all our languages have **control constructs** or **control statements** that allow the natural order of execution to be overridden. The commonly occurring constructs are the **if** construct, the **select case** (Fortran 95) or **switch** (C family) construct, the **for** (C family) or **do** (Fortran) construct, the **while** (C family) or **do while** (Fortran 95) construct, and the **goto** statement. In addition, invoking and returning from subprograms are also control operations. To avoid putting undue emphasis on any one of our languages, we will use Matlab to illustrate these statements.

The `if` construct has the form

```
if <logical expression>
   <statements>
elseif <logical expression>
   <statements>
.  .  .  .  .  .  .  .                                   (6.1)
elseif <logical expression>
   <statements>
else
   <statements>
end
```

The construct executes the first `<statements>` for which their `<logical expression>` evaluates to `true`. Otherwise the construct executes the `<statements>` following the `else`. The `elseif` statements the may be omitted, as may be the `else`. In the latter case, the construct does nothing if none of the `<logical expression>`'s evaluate to `true`.

The `switch` construct has the form

```
switch <switch expression>
   case <case expression>
      <statements>
   .  .  .  .  .  .  .
   case <case expression>                                (6.2)
      <statements>
   otherwise
      <statemets>
end
```

The `<statements>` for the first case for which the `<case expression>` matches the `<switch expression>` is executed. If there is no such case, the `<statements>` for the `otherwise` are executed. The `<case>`'s may be omitted, as may be the `otherwise`.

The essence of a **for** or **do** construct is that a group of statements is executed for a set of values of a variable. The ways by which the set is generated differ in Fortran and the C family, as they both do from Matlab. Here we will give a special case of the Matlab construct that is close to the Fortran construct.

```
for <variable>=i:j:k
   <statements>
end
```

Assuming that $\texttt{i} \leq \texttt{k}$ and $\texttt{j} > 0$, the construct forms the vector

```
(i, i+j, i+2*j, ...., i+n*j),
```

where `n` is the smallest integer such that $\texttt{i} + \texttt{n}*\texttt{j} \le \texttt{k}$. Then for each successive component of the vector, `<variable>` is assigned that component and the `<statements>` are executed. A `break` statement within the body of the for loop, terminates the loop. A `continue` statement cause the loop to restart with the next value of `<variable>`. If $\texttt{k} \le \texttt{i}$ and `j<0`, the loop runs backwards. Otherwise, the loop is not executed. If `j` is not present, it is assumed to be `1`.

For example, `for i=10:-1:2 a(i) = a(i-1); end` shifts the

```
1. for i=10:-1:2
2.     a(i) = a(i-1)
3. end
```

the first nine components of the vector `a` upward by one.

The `while` construct has the form

```
while <logical expression>
   <statements>
end
```

If the `<logical expression>` is `true`, then `<statements>` are executed (here, as in C, `true` means nonzero). This process is continued until the `<logical expression>` evaluates to `false`. As with the `for` construct, `break` exits the `while` loop while `continue` restarts it.

For example, here is a little program to compute the rounding unit.

```
u = 1.0
while (u+1 ~= 1)
   u = u/2;
end
u = 2*u
```

(Be warned, however, that some compilers, in a misguided attempt to improve or optimize, may generate code that does not work.)

The `goto` statement represents an unconditional transfer of control from one part of a program to another. Specifically, all our languages allow statements to be labeled. The statement

```
goto <label>
```

transfers control to the statement whose label is `<label>`.

Ever since Edsger Dijkstra published his classic note "Go To Statement Considered Harmful," [Comm. ACM, 11 (1968) 147–148] the `goto` statement has had bad press. It is true that the undisciplined use of `goto`'s has been responsible for much unreadable code, in which statements wrap around each other like so much spaghetti. It is further true that as more sophisticated control statements were introduced into programming languages, the need for `goto`'s declined greatly. But the `goto` will not go away. There are at least three situations in which it can be useful.

1. For simulating control statements that are not in the language at hand.
2. For breaking out of deeply nested constructs — especially loops.
3. For directing control from several places in a program to a common point — for example, to a clean-up before quitting a section of code or a subprogram.

In cases like these a well commented `goto` can actually improve the legibility of a program.

6.1.6. Subprograms

All our languages allow **subprograms** that can be compiled separately. A subprogram is invoked by a program or another subprogram, it does its job, and it returns to the statement just after its invocation. Information is conveyed to the subprogram in two ways. First, by arguments in an argument list attached to the invoking statement. Second, by means of global variables. Fortran has **functions,** which return a value that can be used in an expression, and **subroutines,** which return values only through the argument list. The C family has only functions.

As an example, consider the Matlab implementation of a stripped down version of the BLAS `dot`.

```
function [d] = dot(n, x, y)
   d = 0.0;
   for i=1:n
       d = d + x(i)*y(i);
   end
   n = 0;    % This statement is for purposes of
             % illustration only.
return
```
(6.3)

The variables `n`, `x`, and `y` are called **dummy arguments** (or **parameters**). As an example of an invocation of this function consider the following script.

```
size = 5;
vec1 = ones(size, 1); vec2 = 2*vec1;
a = dot(size, vec1, vec2)
size
```
(6.4)

The entries `size`, `vec1`, and `vec2` in the invoking argument list are called the **actual**

arguments. They are passed to the corresponding dummy arguments of `dot`, where they are used to calculate the inner product.

Arguments can be passed in two ways: by value or by reference. In the C family (and in Matlab) they are **passed by value** (a.k.a. **call by value**). This means that the actual arguments are physically copied to temporary locations and the copies associated with the corresponding dummy parameters. Any changes to the dummy parameters modifies only the copies, not the original. Thus, when (6.4) is executed, the result is

```
a =
    10
size =
     5
```

not `size=0`, as might be expected from line 6 in (6.3).

In Fortran (and as an option in C++) arguments are **passed by reference** (a.k.a. **call by reference**). This means that the memory addresses of the actual parameters are passed to the subprogram, where they are associated with the dummy arguments. Any changes in the dummy arguments are reflected in the actual arguments. Thus if Matlab were to pass arguments by reference rather than by value, the output of (6.4) would become

```
a =
    10
size =
     0
```

Each method of argument passing has its advantages and disadvantages. Call by value is safe. Arguments are protected against inadvertent changes. But it takes time and memory to copy extensive arguments, such as big arrays. Moreover, call by value makes it impossible to return results via the argument list. We will see later how the C family can use pointers to get around this limitation.

Call by reference avoids the disadvantages of call by value. But it can be dangerous. In particular, modifying array argument in a subprogram can overwrite not only the actual argument but also other variables in memory.

6.1.7. Internal and static variables

In addition to the variables that a subprogram obtains from via its argument list, a subprogram can declare **internal variables** of its own to use during its execution. The question arises of what happens when the subprogram returns to the invoking program. In all our languages, the assumption is that internal variables become undefined unless it is explicitly declared that their values are to be saved between invocations of the subprogram. Such internal variables are said to be **static.** All our languages provide mechanisms for declaring variables to be static.

Static variables are in some sense the memory of a subprogram. Just as a child fears fire only after it has been burnt, a subprogram with static variables may perform differently on two invocations, even though the actual arguments are unchanged. This is an example of a **side effect.** Side effects can be benign or even beneficial. But they make it difficult for a compiler to produce optimized code. In using static variables, make sure you know what you are doing.

6.1.8. Global variables

In extensive programs it often happens that a subprogram needs access to a large number of variables — so large that passing them through a parameter list is unwieldy. This problem can be resolved by introducing **global variables** that can be accessed from within subprograms. All our language have mechanisms for implementing global variables. Needless to say, global variables are a rich source of side effects and should be treated with respect.

6.1.9. Arrays

In scientific computing one frequently deals with vectors and matrices, which can be regarded a linear and rectangular arrays of numbers. All our languages provide methods for constructing arrays of various data types. For example, the Fortran 77 statement

```
double x(20), a(5,12)
```

defines `x` to be a linear array with 20 elements. The `i`th element is referenced by writing `x(i)`. Similarly, `a` is a two-dimensional array consisting of 5 rows and 12 columns. Its `(i,j)`-element is a(i,j).

Unfortunately, the arrays of the Fortran family and the C family are quite different constructs. One of the main difficulties in translating programs between the two families is the treatment of arrays.

6.1.10. Structures and derived types

A point in the plane can be represented by two coordinates that are traditionally called x and y. The fact that two variables are needed to specify a point complicates programs that use them. Instead of keeping track of several sets of variables, it would be nice to refer to them by individual variables — e.g., `A`, `B`, `C` that somehow contain the two coordinates of the point in question.

Fortran 95 and the C family provide the wherewithal to do this. For example, in Fortran 95 you can define a **derived type** `point` by

```
type point
   real x     ! The x-coordinate
   real y     ! The y-coordinate
end type point
```

Then if we define

```
type(point) A, B, C
```

We can refer to the x-coordinate of `A` by `A%x`. The corresponding construction in the C family is called a **structure.**

6.1.11. Pointers

A pointer is a variable that contains an address of a memory location. Languages that have pointers have a mechanism for retrieving and altering the memory locations they point to. For example, if `p` is a C pointer then `*p` represents the object pointed to by `p`.

Pointers have many uses. To give just one example, consider the C statement

```
file = fopen(filename,  mode)
```

which opens the file designated by the character string `filename`. The value returned is a pointer to a structure of type `FILE` that contains the properties of the file. The use of a pointer here is more efficient than returning a copy of the `FILE` structure. Moreover, when it comes time to close the file, the programmer can simply write

```
fclose(file)
```

and `fclose` not only closes the file, but if required deletes any storage it has allocated for `file`. `file`.

Pointers are useful and are therefore widely used. However, they are not without their disadvantages. For one thing, they are potentially expensive. Consider, for example, the following fragment of C code.

```
double a, *b
a = *b
```
(6.5)

To implement this, the pointer `b` must be loaded into a register, and then that register must be used to address `*b`. For example, MIPS code might go as follows.

```
LD  R3, b      ; Get the pointer b
L.D F4, 0(R3) ; Use the pointer to get *b
S.D a, F4     ; Store the result in a
```

Note that the store, which does not involve a pointer, requires only one memory reference, whereas it requires two memory references to load `b`. Thus the assignment statement requires three memory reference as opposed to two for an assignment without the pointer. Whether this extra work is important will depend on the context. For example, if the assignment in (6.5) occurs in the middle of a loop, the pointer can be loaded into a register before the beginning of the loop, so that the assignment requires only two memory reference for each iteration of the loop. But in some applications, especially those involving arrays of pointers, the overhead can be significant.

Another problem is that in C pointers can be used to write to any location in virtual memory. It the location is write protected, the result will be a fatal error. Finally, pointers can make it difficult for compilers to optimize code, a topic treated in §7.1.4. Pointers in Fortran 95 have restrictions on their usage that ameliorate these problems.

6.1.12. Recursion

A **recursive subprogram** is one that calls itself. For example, from the definition of the factorial

$$n! = n(n-1)\cdots(2)(1), \qquad 0! = 1,$$

we have that $n! = n(n-1)!$. This recursion suggests the following recursive Matlab function to compute the factorial.

```
function fct = fact(n)
   if n==0
      fct = 1;
   else                                              (6.6)
      fct = n*fact(n-1);
   end
return
```

Although it has its pitfalls, recursion is a valuable technique that can simplify programs. It is not nearly as expensive as it is reputed to be. Of our languages, only Fortran 77 does not support recursion.

6.1.13. Memory management

When a variable or an array is declared, storage is set aside to contain it. However, it may happen that in the course of execution the program requires more memory — for example, a work array whose size depends on the input to the program. In most systems, this extra storage comes from one of two places: the **stack** or the **heap**. When we discuss the run-time environment at the end of this topic, we will see how the stack and heap are implemented (see §8.2.3) and §**??**. For now, however, it is sufficient to know that getting memory from the stack is cheaper than getting it from the heap. But memory is always available from the heap, whereas it is available from the stack only in certain circumstances. Fortran 95, C, and C++ provide ways of using both sources. Fortran 77, on the other hand, can access neither: all storage must be allocated before the program starts executing.

6.1.14. Preprocessors

A **preprocessor** is a program that takes high-level language code and changes it according to **preprocessor directives** before it is compiled. All our languages have preprocessors, which are descendents of the C preprocessor and hence are very much alike. For example, in all of them a directive starts with `#`.

An important use of the preprocessor is to include text from external files. For example, in C the preprocessor line

```
#include <stdio.h>
```
(6.7)

brings in a system file containing specification statements that are used in I/O.

Another use is to define constants. For example

```
#define PI 3.14159265
```
(6.8)

cause the occurrence of the name `PI` in the text of the program to be replaced by `3.14159265`. More elaborate definitions, called **macros**, can take arguments.

Although we will give no detailed treatment of preprocessors, we will describe specific uses wherever appropriate.

6.1.15. Separate compilation

Large programs — or even smaller ones — are usually divided into several **source files.** There are two reasons. First, managing a very large source file is an error-prone procedure. In fact, just finding something in a large file is not easy. Second, as a program is developed, parts of it become debugged and not subject to change. To recompile these parts whenever another part changes is wasteful in resources. For these reasons our languages all allow separate compilation of files. The results are then linked together into a single executable file, which is loaded onto the machine.

The parts on one file may need to know about what is in the other files. For example, if all files share global variables, the variables must be specified in each. Again, languages like C that automatically convert arguments to subprograms need to know the types of the arguments, even if the subprogram is in a different file. Each language meets these requirements in different ways. For now it suffices to say that Fortran 77 has only primitive facilities, C is versatile but messy, and Fortran 95 is versatile and elegant.

6.1.16. Libraries

To keep a programming language of manageable size the number of operators must be restricted. Functionality not provided by the language itself is provided by subprograms. In principle, one could place the responsibility for designing and coding such subprograms on the user community. In practice, however, some functions are so intimately connected with the language that the language must specify them to insure uniformity. These are the library functions of the language.

In Fortran, the members of the library are called **intrinsic functions.** C and `Cpp` have **standard libraries.** In what follows we will not treat these libraries in detail. But we will use library routines freely in our examples.

—

This completes the general treatment of our languages. The next four subsections will treat the individual languages: Fortran 77, Fortran 95, C, and C++.

6.2. FORTRAN 77

SYNOPSIS: Fortran 77 is the descendent of the oldest living programming language. In spite of its successor, Fortran 95, it is still an important language.

Fortran 77 statements are delimited by lines with fixed fields. Fortran 77 variables consist of up to six alphanumeric characters beginning with an alphabetic character. The short variable names often make Fortran 77 programs difficult to read.

Fortran 77 supports the following types: integer, single and double precision floating-point, complex single-precision floating-point, logical, and character. The character type is what one usually calls a string. In addition to explicit typing, Fortran 77 supports implicit typing by the first letter in a variable name.

Fortran 77 has a full complement of arithmetic and relational operators. It has a full if construct and a do loop construct. It does not have while loop or switch constructs.

The subprograms of Fortran 77 are subroutines and functions. Parameters are passed by reference. Subroutines return values through their calling sequence while functions return a scalar value. The latter can also return values through their calling sequences, though this may lead to undesirable side effects. Internal variables may be declared static. Global variables are provided by common blocks. Fortran 77 does not support recursion.

Fortran 77 arrays are stored in column-major order. When they are passed to subprograms, all dimensions but their last must also be passed to insure proper indexing. However, the declaration of an array in a subprogram need not be the same as that in the calling program, even the number of dimensions may vary. This, in effect, makes it to pass columns of matrices to subprograms.

—

Although **Fortran** (FORmula TRANslator) may not be the first high-level programming language, it is certainly the first successful one. It was developed at IBM by a team headed by John Backus and released in 1954, which makes it over a half century old. It owes its long life to the fact that it has been repeatedly extended. Fortran II, Fortran IV, Fortran 66, Fortran 77, and Fortran 90/95 (here Fortran 95 for brevity) form the sequence of loosely backward compatible languages that have dominated scientific computing for decades (although C and C++ have been enroaching on the territory). Fortran 2003, which has just been released, is a major extension of Fortran, containing, among other things, support for object oriented programming.

As of this writing, the living variants of Fortran are Fortran 77 and Fortran 95. There is a large gap between the two. Fortran 77, which was released in 1980, is the last of a line of what might be called fundamental Fortran. Fortran 95 is in many respects a different language, although it manages to include Fortran 77. Still, there are two good reasons for being conversant with Fortran 77. First, Fortran 95 has never completely caught on, and people still write programs in Fortran 77. Second, because of the backward compatibility of Fortran 77 with Fortran 95, little of the vast legacy of Fortran 77 programs has been translated into Fortran 95 — for example, the widely used LAPACK matrix package is coded entirely in Fortran 77. Hence a reading knowledge of Fortran 77 is essential to the scientific programmer.

In preparing this subsection I have relied heavily on the truly excellent *Professional Programmer's Guide to Fortran77* by Clive G. Page. Although, it is out of print, the entire text is available on the web.

Figure 6.1 contains code adapted from the reference BLAS function `ddot`, which computes the dot product (author, Jack Dongarra). We will use it to illustrate the features of Fortran 77.

6.2.1. Format

Unlike most programming languages, Fortran 77 is line oriented with distinct fields in each line. Here is a list of the fields and their uses along with illustrative lines in `ddot`.

cols	use	lines
1	comment	2, 3
2–5	label	26, 32
6	continuation	19, 21
7-72	statements	18, 22, 33

From this we see that statements are confined to columns 7–72. If a statement must be continued to another line, it is done by placing a character in column 6. A `C` or `*` in column 1 indicates the beginning of a comment. Finally, columns 2–5 are reserved for labels, which must be numeric.

The character set for Fortran 77 consists of the uppercase alphabetical characters, the digits from zero to nine, and a number of special characters like `+`, `(`, and `=`. A

```
      double precision function ddot(n,dx,incx,dy,incy)
c
c     forms the dot product of two vectors.
c
      double precision dx(*),dy(*),dtemp
      integer i,incx,incy,ix,iy,m,mp1,n
c
      ddot = 0.0d0
      dtemp = 0.0d0
      if(n.le.0)return
      if(incx.eq.1. and. incy.eq.1) go to 20
c
c     code for unequal increments or equal increments
c     not equal to 1
c
      ix = 1
      iy = 1
      if(incx.lt.0)
     *   ix = (-n+1)*incx + 1
      if(incy.lt.0)
     *   iy = (-n+1)*incy + 1
      do 10 i = 1,n
        dtemp = dtemp + dx(ix)*dy(iy)
        ix = ix + incx
        iy = iy + incy
   10 continue
      ddot = dtemp
      return
c
c     code for both increments equal to 1
c
   20 do 30 i=1,n
         dtemp = dtemp + dx(i)*dy(i)
   30 continue
      ddot = dtemp
      return
      end
```

Figure 6.1: Fortran 77 implementation of the BLAS routine `ddot`.

symbolic name consists of not more that six alphanumeric characters, the first of which must be alphabetical. The restriction to six characters makes it difficult to generate meaningful names for variables, and the nomenclature in Fortran 77 programs tends to be terse and cryptic — e.g., `INTGRL` for `INTEGRAL`.

Although lowercase letters are not part of the standard, all compilers accept them, and programmers often use them because they are easier to read. However, Fortran 77 is not case sensitive. The symbolic names `FARB`, `farb`, and `FaRb` are all the same to the compiler. For readability, all examples in this subsection will be in lower case.

6.2.2. Data types

In §2 we described the data types that computers commonly manipulate. High level programming languages have conventions for associating variables or identifiers with data types. We will now treat the conventions of Fortran 77

A **variable** is a symbolic name that is associated with a data type. The following table lists the types along with representative examples of their constants.

`integer`	`350, -25`
`real`	`3.0, -2.5, 4.789e+2 (= 478.9)`
`double precision`	`3.0d0, -2.5d0, 4.789d+2`
`complex`	`(5.4, 6.e2)`
`logical`	`.true., .false.`
`character`	`'j', 'Stewart'`

Note particularly the 'scientific notation' for real and double precision constants (it is obligatory for double precision constants). The character type is really a string; that is, an array of characters as described in §2.2.

The types of variables can be declared by **specification statement;** e.g.,

```
double precision dx(*),dy(*),dtemp
```

from line 6 of `ddot`. The size of the type integer is system dependent, but it is typically four or eight bytes. Nowadays the types real and double precision invariable mean single and double precision IEEE floating-point numbers. Fortran 77 has a complex type, but no double precision complex type (however, the specification `double complex` is widely available as an extension.) Logical variables are used in if statements.

If an object is not typed by a specification statement, Fortran 77 will give it an **implicit type.** Specifically, if a variable begins with `i`, `j`, `k`, `l`, `m`, or `n` it of type integer. Otherwise it is of type real. As a matter of good programming practice, all variables should be typed explicitly.

6.2.3. Expressions and assignment

The arithmetic operators in Fortran 77 are `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), and `**` (exponentiation). The relational operators are `.lt.` ($<$), `.le.` ($\leq$), `.eq.` ($=$), `.ge.` ($\geq$), `.gt.` ($>$), and `.ne.` ($\neq$). The logical operations are `.and.`, `.or.`, `.eqv.` (logical equivalence), `.neqv.` (nonequivalence aka exclusive or) and `.not.` (negation). The binding of the operators follows the natural mathematical conventions. Operands in mixed expressions are promoted in a natural way.

Fortran 77 provides a number of **intrinsic functions** to complement its operators. Of particular importance are functions like `SIN`, `COS`, etc. These functions are **polymorphic** in the sense that their behavior depends on the type of their argument. For example, if `x` is typed real, double, or complex, then `sin(x)` returns a real, double, or complex value.

Fortran 77 has primitive operations to extract substrings and concatenate strings. Other operations on strings are performed by intrinsic functions.

The assignment operator is `=`, as in

```
  z = sqrt(x**2 + y**2)
```

In mixed assignments the right-hand value is converted to the type of the left-hand variable, with a possible loss of precision.

6.2.4. Control constructs

Fortran 77 has two forms of the **if** construct. The if statement has the form

```
  if (<logical expression>) <statement>
```

If `<logical expression>` evaluates to `.TRUE.` then `<statement>` is executed; otherwise, not.

The **block if** construct differs only in minor syntactical details Matlab version (6.1).

The chief looping construct is the `do` construct. The following is a typical illustration

```
        do 100 i = 1,10,2
           <statements>                                    (6.9)
    100 continue
```

The variable `i` is called the **loop index.** The loop is executed for `i=1,3,5,\ldots,9`. Loops can run backward. For example

```
        do 100 i = 10,1,-3
```

executes its body for `2=10,7,4,1`. If the increment/decrement [i.e., 2 in (6.9)] is missing it is assumed to be one. Inconsistent loops are not executed. For example, a loop beginning

```
          do 100 i = 10,1
```

falls through its own body doing nothing.[8]

The loop index may not be redefined in the body of the loop. When the loop terminates — whether naturally or by using a `goto` to exit the loop — the value of the loop index is well defined, though the rules are a little complicated. Arithmetic expressions are permitted on the left of the equality sign; however, such expressions are evaluated only once just before the start of the loop. Floating-point expressions are permitted, as in

```
         do 100 x = 0.0, 1.0, 0.1
```

However, the use of floating point in do constructions is deprecated because rounding error may cause them to behave unpredictably. A better way to write the above loop is

```
         h = 1.0/n
         do 100, I=0,n
            x = i*h
         . . . . . .
```

In addition Fortran 77 has a **return** statement to exit subprograms and a **stop** statement to terminate execution. Unfortunately, Fortran 77 does not have the equivalent of a switch or case construct. Nor does it have a while construct.

6.2.5. Program organization and subprograms

Like all our programming languages, Fortran 77 allows programs to be subdivided into programming units. The three main units are the **program**, the **function**, and the **subroutine**. Each program must have a single program unit, which has the form

```
     program <name>
        <specification statements>
        <executable statements>
     end
```

When the program begins execution, it will start with the first executable statement in the program `<name>`. Note that in a program (as well as in a function or subroutine) specification statements must precede executable statements.

The subprogram `ddot` in Figure 6.1 is a typical function. The header

[8]There is a cautionary tale here. The original Fortran `do` construct tested for termination at the end of the loop because that was convenient for the IBM 704 — the target machine for Fortran. The result was that an inconsistent loop would always be executed once. This 'feature' was widely used, even after the standard for Fortran 66 decreed that the result of inconsistency was undefined. The community of macho programmers, however, refused to take the hint, and, when the Fortran 77 standard declared that inconsistent loops were not to be executed, their howls of anguish and rage could be heard far and wide.

```
double precision function ddot(n,dx,incx,dy,incy)
```

specifies the type of the returned value `ddot`. The dummy variables in the argument have their types specified in the next to statements.

Arguments are passed to functions and subroutines by reference, and hence a function can change the values of the actual parameters used when it is invoked. However, this practice is deprecated because of the possibility of side effects. Consider for example the two functions `fu` and `bar` defined as follows.

```
integer function fu(m)    integer function bar(m)
integer m                 integer m
   fu = 5                    bar = m
   m = 10                 end
end
```

When these functions are used in the sequence

```
m = 1
if (fu(m) .gt. bar(m)) ...
```

the result will depend on which order `fu` and `bar` are evaluated. If `fu` first, the relational expression in the if statement will evaluate to `.true.` If `bar` is evaluated first, the expression will evaluate to `.false.`. The Fortran 77 standard is of no help here: it does not specify an order for the evaluation `fu` and `bar`.

Subroutines can only return values through the argument list. Here is a short subroutine implementing a limited version of AXPY.

```
      subroutine daxpy(n, a, x, y)
      integer n
      double precision a, x(*), y(*)
         do 10 i=1,n
            y(i) = y(i) + a*x(i)
   10    continue
     end
```

Note that we cannot convert `daxpy` to a function, since functions can return only scalars, not arrays. A subroutine is invoked by a **call** statement: e.g.,

```
call daxpy(20, 3.0d0, u, v)
```

There is a **return** statement that transfers control back to the invoking procedure. Alternatively, a subprogram will return when it runs into its end statement. A `save` specification allows variables to retain their values between a return and the next invocation.

Fortran 77 allows subprograms to be passed to a function or subroutine through the parameter list. This feature can be used to write utility routines that operate on various functions. For example, a function

```
double precision function intgrl(a, b, func)
double precision a, b, func
```

that computes an approximation to the integral of `func` from `a` to `b`, might be invoked as follows

```
intrinsic sin
external myfunc
double precision iab1, iab2
iab1 = intgrl(0, 10, sin)
iab2 = intgrl(0, 10, myfunc)
```

This would cause `iab1` to be set to an approximation to $\int_0^{10} \sin(x)\,dx$ and `iab2` to be set to an approximation to $\int_0^{10} \texttt{myfunc}(x)\,dx$. If an intrinsic function is used in a parameter list, must be declared **intrinsic** in the invoking program. All other subprograms so used must be declared **external**. Further typing is unnecessary, since the invoked subprogram types its own parameters.

An important alternative to communicating through the argument list is to use **global variables.** Named blocks of global variables can be created using **common blocks**. An example of such a block is

```
common /myblk/ a(100), b(10), c, num
double precision a, b, c
integer num
save /myblk/
```

Any subprogram containing these declarations has access to the global variables `a`, `b`, `c`, and `num`. The save declaration insures that values of the variables of the block are saved on return from a subprogram. Common blocks are initialized via the **block data** construct.

Fortran 77 does not support recursive subprograms.

6.2.6. Arrays and subprograms

Fortran allows its types to be declared as arrays with up to seven dimensions. For example, consider the following specification statements.

```
integer n(1000)
real b(100), b(-5:5), c(25,30)
```

The first statement declares `n` to be 1-dimensional array of 1000 integers. Its elements are referenced as `c(1)`,...,`c(1000)`. In the second statement `a` is a 1-dimensional array of 100 reals. The variable `b` is a 1-dimensional array of 11 reals that are referenced as `b(-5)`, `b(-4)`, ..., `b(4)`, `b(5)`. The variable `c` is a 2-dimensional array of reals whose indexing begins with `c(1,1)`.

```
    a      a + 40   a + 80   a + 120
  A(1,1)   A(1,2)   A(1,3)    A(1,4)

  a + 8    a + 48   a + 88   a + 128
  A(2,1)   A(2,2)   A(2,3)    A(2,4)

  a + 16   a + 56   a + 96   a + 136
  A(3,1)   A(3,2)   A(3,3)    A(3,4)

  a + 24   a + 64  a + 104   a + 144
  A(4,1)   A(4,2)   A(4,3)    A(4,4)

  a + 32   a + 72  a + 112   a + 152
  A(5,1)   A(5,2)   A(5,3)    A(5,4)
```

Figure 6.2: Column-major storage of an array.

The elements of multidimensional arrays are stored consecutively in memory with the first index varying most rapidly and the last least rapidly. For example, if `c` is defined by

```
double precision c(2,2,2)
```

then the order of the elements of `c` in memory are

```
c(1,1,1), c(2,1,1), c(1,2,1), c(2,2,1),
          c(1,1,2), c(2,1,2), c(1,2,2), c(2,2,2)
```

Figure 6.2 shows how the elements of an a 5×4 array `A` of doubles is stored, assuming that the starting address of the array is `a`. A formula for the position of `A(i,j)` in memory is

```
a + 8*(5*(j-1) + i-1).
```

The `8` in this formula is the length of of a double-precision word. The `5` is equal to the size of the first dimension of `A`, and that is no coincidence. In general, if `A` `m`$\times$`n`, then the position of `A(i,j)` in memory is

```
a + size*(m*(j-1) + i-1).                                   (6.10)
```

The dimension `m`, which is called the **leading dimension** of `A` is critical to locating the elements of `A` in memory. Surprisingly, perhaps, the trailing dimension plays no role in determining where elements are stored.

To illustrate the interactions of subprograms and arrays, we will use the following code as a running example.

```
integer m, n, mmax, nmax
parameter (mmax=100, nmax=50)
data m/75/, n/25/                                          (6.11)
double precision a(nmax), b(nmax), c(mmax,nmax), d(mmax,nmax)
call fu(m, n, mmax, nnax, a, b(3), c, d(1,5))
```

The **parameter** specification defines and initializes two **named constants,** `mm` and `nn`, which are in turn used to specify the dimensions of `a` and `b`. The **data** specification is the standard way of initializing ordinary variables.

The call to `fu` illustrates a common practice in scientific computing. In the course of a computation the size of an object may change. For example, `a` in (6.11) may represent a vector whose size `n` changes, and `b` may represent a matrix whose row and column dimensions `m` and `n` also change. Since Fortran 77 cannot change storage allocation at execution time, it is customary to determine a maximum problem size — in this example defined by `mmax` and `nmax` — and make all relevant arrays large enough to contain the problem.

The calling sequence shows that there are two ways to pass an array — pass the entire array or pass an element of the array. The first is uncomplicated. Since Fortran 77 passes actual arguments by reference, the address of the first element in the array is passed to the subroutine. The second is open to two interpretations. In our example, `b(3)` could be regarded as an expression that evaluates to the value of the third element of `b`. In that case, the value of `b(3)` would be stored in a temporary location in memory and the address of that location would be passed. Fortran 77 does not do this. Instead it passes the address of the third element of `b`. This convention has important implication, as we shall see in a moment.

The declarations in `fu` can vary. Here is the most conservative option

```
subroutine fu(m, n, ldc, nmax, aa, bb, cc, dd)
integer m, n, ldb, nn
double precision aa(nmax), bb, cc(ldc, nmax), dd
```

In this case `aa` and `cc` are declared as arrays of the same size as their actual arguments `a` and `c`. The dimensions for this construction must be passed through the calling sequence of the subprogram. Such subprogram arrays are called **adjustable arrays.** An advantage of adjustable arrays is that the compiler, knowing their dimension, can generate code to check to see if an array reference is out of range.

We have renamed `mmax` as `ldc`, which stands for the **leading dimension of c**. This nomenclature is widely used in scientific computing. It follows from (6.10) that the leading dimension is absolutely necessary for the subroutine to compute array references, like `cc(i,j)` properly.

The dummy arguments `bb` and `dd` corresponding to `b(3)` and `d(1,5)` are declared as scalars. Modifying them will cause the `b(3)` and `d(1,5)` to be modified, but the arrays will be otherwise unaltered.

Another variant in the declarations in `fu` is based on the fact that the the value of `nn` is not needed to make array references. This leads to the following specification statements.

```
subroutine fu(m, n, ldc, nmax, aa, bb, cc, dd)
integer m, n, ldb, nn
double precision aa(*), bb, cc(ldc,*), dd
```

The arrays `aa` and `cc` are called **assumed-size arrays**. Range checking cannot be performed on assumed-size arrays; nor can they be used in any construct where the size of the arrays must be known. But the value of `nn` does not have to be passed in the argument list. Note that we still have to pass `mmax` (a.k.a. `ldc`) to recover entries in the array `cc`.

A further level of subtlety is illustrated by the following code.

```
subroutine fu(m, n, ldc, nmax, aa, bb, cc, dd)
integer m, n, ldc, nn
double precision aa(*), bb(*), cc(ldc,*), dd(ldc,*)
```

Now `bb` has become a 1-dimensional array beginning at the address of `b(3)` in the calling program. Thus a reference to `bb(i)` in the subroutine corresponds to a reference to `b(3+(i-1))`. Equivalently, `bb(1:n-3+1)` corresponds to `b(3:n)`. Analogously, `dd(1:m,1:n-5+1)` corresponds to `d(1:m,5:n)`. As above, `ldc` is still necessary to insure proper indexing. In this manner, Fortran 77 allows subarrays to be passed to subprograms.

Here is the bottom line.

```
subroutine fu(m, n, ldc, nmax, aa, bb, cc, dd)
integer m, n, ldb, nn
double precision aa(*), bb(*), cc(ldc,*), dd(*)
```

The change is in the declaration of `dd`, which has become a linear array, whose starting address is that of `d(1,5)`. In other words `dd(1:m)` is simply the fifth column of `dd`.

The treatment of `dd` in the last two examples is what makes the basic linear algebra subprograms (BLAS) work (see §4.3.3), since it enables one to pass submatrices to them. For a nontrivial example, suppose we want to compute $B = A^{\mathrm{T}}A$, where A is an $m{\times}n$ matrix. Partitioning $A = (a_1\ a_2\ \cdots\ a_n)$ by columns, we have

$$B = A^{\mathrm{T}}A = \begin{pmatrix} a_1^{\mathrm{T}} \\ a_2^{\mathrm{T}} \\ \vdots \\ a_n^{\mathrm{T}} \end{pmatrix} (a_1\ a_2\ \cdots\ a_n) = \begin{pmatrix} a_1^{\mathrm{T}}a_1 & a_1^{\mathrm{T}}a_2 & \cdots & a_1^{\mathrm{T}}a_n \\ a_2^{\mathrm{T}}a_1 & a_2^{\mathrm{T}}a_2 & \cdots & a_2^{\mathrm{T}}a_n \\ \vdots & \vdots & & \vdots \\ a_n^{\mathrm{T}}a_1 & a_2^{\mathrm{T}}a_2 & \cdots & a_n^{\mathrm{T}}a_n \end{pmatrix}$$

Thus the (i,j) element of B is just the dot product of the ith and jth columns of A. If we use `ddot` (Figure 6.1), we get the following code.

```
      do 20 j=1,n
         do 10 i=1,j
            b(i,j) = ddot(m, a(1,i), 1, a(1,j), 1)
            b(j,i) = b(i,j)                              (6.12)
10       continue
20    continue
```

This is simplicity itself. Note that we do not need a leading dimension for `a`, since we are traversing columns of `a` in `ddot`, which in Fortran 77 implies a stride of one. Things would be rather different if we were computing AA^{T}; see Exercise ??.

Passing the address of an element of an array has an admittedly ad-hoc flavor. However, as we have seen, it can be useful in manipulating subarrays, and it is quite safe when it is used with well-tested subprograms like the BLAS. The alternative is to pass the array, its leading dimension, and the four boundaries of the subarray, which is error prone and does not result in code that is easy to read. For example, the calling sequence of `ddot` in (6.12) would have to become something like

```
ddot(a, 1, m, i, i, 1, a, 1, m, j, j, 1)
```

The code for `ddot` also becomes more elaborate.

6.3. C

Synopsis: C is a language developed in the early 1970's that is oriented primarily toward systems programming. Nonetheless it is used in scientific computing, and is especially useful where the manipulation of data structures is as important as number crunching.

C is a free-format language with explicitly delimited statements. Simple statements end with a semicolon. Statements can be grouped together into a compound statement or a block.

C supports the following types: integer, single and double precision floating-point, and characters. C has no complex type, a serious omission. C also has no logical type, but for any type nonzero represents true and zero represents false. Strings are arrays of characters terminated by a delimiter. C has no implicit typing.

Variables may be declared within blocks. Storage for these automatic variables is allocated on the stack, and they go away when the block is exited unless they have been declared static.

C is very rich in operators It has the usual arithmetic (save exponentiation) and relational operations, and in addition binary operations. It has a variety of increment, decrement, and assignment operators. Operations on strings are performed by the standard library.

C has if and switch constructs as well as while and for loops.

The only subprogram type in C is the function. Parameters are passed by value, but passing by value can be simulated by using pointer. C supports recursive functions. Every program must start with a function called `main`.

The unit of compilation of C is the file. In addition to functions, files can contain variables local to the file and global variables. The rules for how files communicate with one another are complicated. Some of the complexity can be reduced by the use of header files.

C supports pointers, which are essentially addresses of variables. It has operators for to reference variables through pointers and to initialize pointers to the address of a variable. Pointer arithmetic allows one to use a pointer to move about in memory.

C arrays are intimately related to pointers, and array references can be replaced by pointer arithmetic. A major failing of C is that it is impossible to pass a two-dimensional array to a function unless the function knows at compile time the trailing dimension of the array.

Structures combine variables (and other structures) into one object. They can be used to implement data structures, such as linked lists and queues. The use of structures is illustrated by the compressed representation of a sparse vector.

—

C was developed in the early 1970's at Bell Laboratories by Dennis Richie. For background I can do no better than to quote the summary of Richie's excellent *Development of the C Language.*

> Ken Thompson created the B language in 1969-70; it was derived directly from Martin Richards's BCPL. Dennis Ritchie turned B into C during 1971-73, keeping most of B's syntax while adding types and many other changes, and writing the first compiler. Ritchie, Alan Snyder, Steven C. Johnson, Michael Lesk, and Thompson contributed language ideas during 1972-1977, and Johnson's portable compiler remains widely used. During this period, the collection of library routines grew considerably, thanks to these people and many others at Bell Laboratories. In 1978, Brian Kernighan and Ritchie wrote the book that became the language definition for several years. Beginning in 1983, the ANSI X3J11 committee standardized the language.

C was designed to code the unix operating system. As such, it is oriented to systems programming and is not as well suited as Fortran for numerical work. Nonetheless, it is a more supple language than Fortran and a lot more fun to write code in. Consequently, C and its object oriented successor C++ are widely used in scientific computing, especially in situations where the manipulation of data structures is as important as floating-point arithmetic.

C is not a subset of C++; but with minor precautions, a C++ compiler will work with C programs. Since its initial standardization, C has been extended to C99, a new standard. C99 removes many of the infelicities of C. From the standpoint of scientific computing, the most important is the introduction of variable array dimensions. Unfortunately these very improvements make C strongly incompatible with the C++ standard. The crystal ball is cloudy, but one scenario is that C++ will extend itself along the lines of C99.

Figure 6.3 contains a C implementation of `ddot`. We will use it in as a running example in what follows.

```
double ddot(int n, double x[], int incx, double y[], int incy){

   /* Form the dot product of two vectors. */

   double dot;
   int i, ix, iy;

   dot = 0.0;
   if (n <= 0) return 0;
   if (incx!=1 || incy!=1){

      /* Code for unequal increments or equal increments
         not equal to 1. */

      ix = (incx <0) ? (1-n)*incx : 0;
      iy = (incy <0) ? (1-n)*incy : 0;
      for (i=0; i<n; i++){
         dot += x[ix]*y[iy];
         ix += incx;
         iy += incy;
      }
   }
   else{

      /* Code for both increments equal to 1. */

      for (i=0; i<n; i++)
         dot += x[i]*y[i];
   }
   return dot;
}
```

Figure 6.3: C implementation of the BLAS routine **ddot**.

6.3.1. Format

Although `ddot` is broken up into lines for purposes of legibility, C treats the end of a line as **whitespace**, which also includes blanks, horizontal and vertical tabs, new lines, and form feeds. Thus we could write the two declarations beginning at line 5 as

```
double dot; int i, ix
             , iy;
```

On the other hand, white space is significant in separating identifiers and keywords. Thus we cannot code

```
inti, ix, iy;
```

An **identifier** in C consists of any number of alphanumeric characters and underscores. The first character must be a letter or an underscore. C is case sensitive; e.g., the identifiers `tom` and `Tom` are different. C has 32 keywords that may not be used as identifiers. In Figure 6.3 the keywords in order of appearance are `double`, `int`, `if`, `return`, `for`, and `else`.

C is divided into statements. Simple statements end with a semicolon; e.g.,

```
dot = dot + x[ix]*y[iy];
```

Groups of simple statements may be grouped by curly braces into a **compound statement** or **block.** The three statements starting on line 18 in `ddot` form a block.

Comments are delimited by `/*` and `*/`, as in lines 12 and 13 of `ddot`. C++ and many C compilers permit `//` start a comment that ends with the line. This convention is especially effective for making short running comments about the code, as in

```
for (i=0; i<10; i++){ // Initialize x and y
    x[i] = i;
    y[i] = 1;
}
```

But it is not standard C, and you use it at your own risk.

6.3.2. Data types

C supports the types **int**, **char**, **float**, and **double**, representing respectively integers, characters, and single- and double-precision floating point numbers. Declarations may be include other characteristics. For example,

```
unsigned short int a;
```

declares a to be a short integer — on today's machines most likely consisting of two bytes — that has no sign.

C has no logical type. However, whenever an expression must be interpreted in a logical context, it is true if it is nonzero and false if it is zero.

A variable typed char is represented a word of a size that can contain the (implementation defined) character code. There are a number of special character constants that are represented with a backslash escape. For example, '\n' is **newline** and '\0' is the character whose code is zero. Naturally, '\\' is the backslash character.

C has no **string** type. Strings are represented as arrays of characters. A string is always terminated by the character code '\0'. Thus the string

```
"I am longer than you think."
```

contains 28 characters, not just the 27 that you see. Note the distinction between 'a', which is a char, and "a", which is a string. Characters and strings are manipulated by functions in the standard library.

Declarations can be placed inside of any block, and the variables so declared are said to **automatic** or **local.** Storage is allocated to them on the stack when the block is entered. Local variables loose their current values when their block terminates, unless they have been declared **static**.

6.3.3. Expressions and assignments

The arithmetic operators in C are + (addition), - (subtraction), * (multiplication), / (division), and % (modulus). C has no exponential operator. To evaluate a power, you have to use the standard library function pow. The relational operators are <, <=, ==, >=, > != ($\neq$). The logical operators are && (and), || (or), and ! (negation). These operators have their natural mathematical precedence. In mixed expressions, operands are promoted as usual.

In addition C has bitwise operators for and, or, exclusive or, and one's complement. It also has left and right shift operators

The **conditional operator** is a compact if statement. An example, occurs in line 15 of ddot

```
(incx <0) ? (1-n)*incx : 0;
```

If incx is negative, the expression evaluates to (1-n)*incx; Otherwise it evaluates to zero.

A widely used operator is the *increment operator*. Specifically, the expression

```
i++
```

evaluates to `i` but then increases the value of `i` by one. There is a variant `++i` that increments `i` and evaluates to the incremented value. There are also two decrement operators: `i--` and `--i`.

Another important operator is the **cast**, which converts between types. For example, in the code

```
float x;
(double) x
```

the expression `(double) x` evaluates to a double-precision number that has the value of the single-precision variable `x`.

The basic **assignment operator** is `=`. Thus

```
a = b
```

assigns the value of `b` to `a`, with appropriate conversions if the `a` and `b` are not of the same type. There are variants of this assignment that perform operations along with the assignment. For example, the statement

```
dot += x[ix]*y[iy];
```

from line 18 in Figure 6.3 is equivalent to

```
dot = dot + x[ix]*y[iy];
```

There are similar assignment statements for other operations.

A curious fact about assignment statements is that they are also expressions which evaluate the left-hand side of the assignment. They are often seen in statements like

```
if ((a = func(b)) > 0) ...
```

which assigns the value of `func(b)` to `a` and then tests if `a` is positive.

6.3.4. Control statements

C has a while statement of the form

```
while (<expression>)
   <statement>
```

The `<statement>` (which may be compound) is evaluated as long as the `<expression>` evaluates to true.

One of the most widely used control constructs is the **for** statement. It has the form

```
for (<expr1>; <expr2>; <expr3>)
   <statement>
```

This statement is equivalent to

```
<expr1>;
while (<expr2>){
   <statement>
   <expr3>;
```

Thus

```
for (i=0; i<n; i++)
   dot += x[i]*y[i];
```

from Figure 6.3, line 27, updates `dot` for `i = 0,1,...,n-1`. It is permitted change loop variables in in the body of the loop. Such changes, however, are hard to decipher and may get in the way of compiler optimization. The **break** statement is used to leave the innermost `for` or `while` loop. The **continue** statement causes the loop to restart.

C has a standard **if statement**, complete with `else` and `else if` constructs. The `switch statement`, which is close to the Matlab switch, is illustrated by the following code fragment.

```
int a[10], i, m, n0=0, n1=0, n2=0;
for (i=0; i<10; i++){
   m = a[i]%3
   switch (m){
      case 0: n0++; break;
      case 1: n1++; break;
      case 2: n2++; break;
      default: /* Can’t get here */ ;
   }
}
```

This fragment counts the number of elements in `a` that are `0`, `1`, and `2` modulo 3. Assuming that the elements of `a` are positive, the variable `m` can assume only the values 0, 1, or 2. The switch on `m` match `m` against the constants following the `case`’s and executes the corresponding statement when a match is found. Otherwise, the `default` is executed. The switch expression (here `m`) does not have to be an int — char is a common alternative.

A feature of the C switch statement is that if a case does not end with a break then execution falls through to the next case. Thus if we remove the break statements, `n0` will contain the number of elements for which `m` is less than or equal to zero; `n1`, the number of elements for which `m` is less than or equal to one; and `n2`, the number of elements for which `m` is less than or equal to two. The pros and cons of this feature are not easily resolved.

C has statement labels and a `goto` statement. For a discussion of this construct see p. 107.

6.3.5. Functions

The basic computational unit of C is the function. It has the structure

```
<type> <name>(<parameter list>){
   <declarations>
   <statements>
}
```

The `type` may be omitted and the **parameter list** may be empty. An example is the function

```
double ddot(int n, double x[], int incx, double y[], int incy)
```

in Figure 6.3. Note how the parameters are typed in the parameter sequence. In what follows a **parameter** is a variable in the declaration of a function, and an **argument** is a variable or expression in a function invocation.

There are two formal ways to terminate execution of a function. The first is by the **return** statement which has the form

```
return <expression>;
```

This statement causes the function to return to its invoking program with the value of `<expression>`. If `<expression>` is missing, the returned value is undefined.

The standard library function `exit(int)` shuts down the whole program. By convention an argument of `0` represents a successful run, while a nonzero indicates an error.

Arguments to a function are passed by value and therefore cannot be modified. We will see later how we can get around this limitation by passing pointers to objects.

C allows recursive functions — that is subroutines that invoke themselves. For an example of such a function, see (6.6).

6.3.6. Program organization

Each program must have a **main function** of the form

```
int main(int argc, char *argv[])
```

This is the function the operating system invokes to start a program. The parameters `argc` and `argv` are used by the operating system to pass information — for example, command line arguments — to `main`. They may be omitted, as may the specification `int`.

The unit of compilation, called a **translation unit,** is the file. A typical file will have the following elements, usually in the following order.

1. Preprocessor includes

2. Preprocessor definitions
3. Declaration or definition of external objects
4. Functions

The first three items are called the **header** of the file.

Large programs will usually broken up into two or more files, which will depend on one another. How these dependencies are resolved is too big a topic to treat in detail. To illustrate some of the ideas we will consider the program in Figure 6.4, which is distributed among two program files (`file1.c` and `file2.c`) and a header file (`file.h`).

The first line of the two program files brings in the header file. We will treat this file toward the end of this discussion.

The second line in each program file is a preprocessor macro defining a constant `MAX`. Whenever the identifier `MAX` is encountered, the preprocessor will replace it with its definition. A definition is valid only within the file in which it occurs and only from the point of definition to the end of the file. In particular, there is no inconsistency in having a different value for `MAX` in `file1.c` and `file2.c`. By convention uppercase letters are used in such definitions.

The next two lines require some new terminology. A variable declared in a file but outside a function is called an **external** variable. It's declaration is valid in the file from the declaration point on, and it can be seen even in the body of functions.

It is important to make a distinction between declarations and definitions. In `file1.c` the variable `a` is both declared and defined. The initialization makes it a defined variable. On the other hand in `file2.c` the variable `a` is declared but not defined. The storage class specifier `extern` says to look elsewhere for a definition — perhaps in the same file, perhaps in another.

A variable may not be defined more than once. If we replace the declaration of `a` in `file2.c` with

```
int a = 10;
```

then an error would result, even though the definitions are the same. You could replace it with

```
int a;
```

but that is another story.

Note that `extern` is not just an abbreviation for external. The definition of `a` in `file1` makes `a` an external variable, even though it is not qualified by `extern`.

The variable `c` appears to be defined in both `file1.c` and `file2.c`, but the qualifier `static` makes each variable local to the file in question. Thus there can be no clash between the two definitions.

```
    file1.c                          file2.c
                              |
#include "file.h"             |   #include "file.h"
                              |
#define MAX 100               |   #define MAX 50
                              |
int a=10;                     |   extern int a;
extern int b;                 |   int b = 7;
                              |
static int c=5;               |   static int c=10;
                              |
int fu(int, int);             |   int fu(int x, int y){
                              |      static int z=10;
int bar(int a , int d){       |      return a*x + b*y + c*z;
   int c;                     |   }
   c = a + TWO*d;             |
   c = (c > MAX) ? MAX: c;    |   int bar(int, int);
   return c*c;                |
}                             |
```

```
                          file.h

                       #define TWO 2
                       extern int a;
                       extern int b;
                       int fu(int, int);
                       int bar(int, int);
```

Figure 6.4: Three program files

The two functions fu and bar are declared in in both program files, but fu is defined in file1.c while bar is defined in file2.c. There is no need for externs here, since function declarations not accompanied by a body of code can be a definition.

In a function arguments and internal variables declared within the function override external variables. For example, a and c in the function bar are independent of their external declarations above. On the other hand, external variables that are not so overridden are available to the function. This is the case for the a, b, and c in the function `fu` in `file2.c`. The static internal variable `z` in the same function will be initialized the first time `fu` is invoked and will retain its most recent value when `fu` is reinvoked.

Much of the above can be restated succinctly in terms of scope. The **scope** of a variable is the part of the program where its declaration is valid. The scope of an external variable is the entire program, or at least those files that declare or define the variable. The scope of a static external variable is confined to the file in which it is declared — more precisely the part of the file from the declaration on. The scope of a function parameter is the body of the function and it overrides any external variables. The scope of an automatic variable is the block in which it is declared, and it overrides external variables and variable declared in containing blocks.

Declarations and definitions in separate files are brought together by the **linker,** which takes separately compiled programs and turns them into a single executable **object program.** We will treat linkers later in this division.

Large programs will often have large, overlapping headers. Changes in one header will have to be echoed in the other headers. This process is obviously and invitation to wholesale blunders. The cure is to collect all external declarations into a common **header file** and include it at the beginning of each program file. In Figure 6.4 the file `file.h` is such a header file. It defines `TWO` as the constant `2` and declares the external variables `a` and `b` and functions `fu` and `bar`. Note that there is nothing wrong in redeclaring or defining a variable already declared, provided the new declarations are consistent with the old. Thus the header file causes no errors in `file1.c` and `file2.c`.

C provides special header files for programs in the standard library. Of particular importance for scientific computing is `math.h`.

The organization of large programs is something of an art. You are well advised to study some specific large programs or systems, before undertaking the writing of one yourself.

6.3.7. Pointers

C provides a means of determining the memory address of an object. Specifically, if `a` is a variable, then `&a` is its address. Since `&a` points to the location of the value of `a`, it is called a **pointer.** If `b` is a pointer, then `*b` is equivalent to a variable whose value is found at `b`. The operator `&` is called the **address operator,** and the operator `*` is called the **indirection** or **dereferencing operator.**

The declaration

```
int a, *b;
```

declares `a` to be of type int and `b` to be a pointer of type int. The assignment

```
b = &a
```

makes `b` point to `a`. The dereferenced pointer `*b` can be used in place of `a`. For example, the expressions

`a/5` and `*b/5`

produce the same result.[9] The dereferenced pointer `*b` can even be used as the target of an assignment. The statements

`a = 5;` and `*b = 5;`

have the same effect.

An important use of pointers is to get around the limitations of call by value. As a silly example, suppose we want to compute the Fibonacci sequence defined by

$$a_0 = 0, \quad a_1 = 1, \quad a_{i+1} = a_i + a_{i-1} \quad i = 1, 2, \ldots .$$

Here is a function that tries to replace (a_{i-1}, a_{i-2}) with (a_i, a_{i-1}).

```
void fib(int ai, int aim1){
    int temp;
    temp = ai;
    ai = ai + aim1;
    aim1 = temp;
}
```

The hope is that the program

```
int f=1; ff=0;
for (i=2; i<=10, i++)
   fib(f, ff);
```

will compute the first 10 Fibonacci numbers.

Unfortunately, the function does not work because the changes in the parameters `ai` and `aim1` are not reflected in the arguments `f` and `ff`. However, if we rewrite our original function in the form

```
void fib(int *ai, int *aim1){
    int temp;
    temp = *ai;
    *ai = *ai + *aim1;
    *aim1 = temp;
}
```

[9]There is a deadly inconsistency in the C syntax regarding pointers. The expression `5/*b` does not divide `5` by the value of `b`. It is `5` followed by the beginning of a comment. You must write `5/(*b)`.

then the sequence

```
int f=1, ff=0, i;
for (i=2; i<=10, i++)                                    (6.13)
   fib(&f, &ff);
```

does compute the Fibonacci numbers. The reason is that we have passed *pointers* to `f` and `ff` instead of their values. The use of the dereferencing operator in `fib` causes causes the values of `f` and `ff` to be changed.

The general procedure for altering the argument of a function has three parts.

1. Pass a pointer to the argument instead of the argument.
2. In the function type the parameter corresponding to the argument as a pointer of the appropriate type.
3. In the body of the function use the dereferencing operator with the parameter.

Pointers can also be used to pass a function as an argument to a function. As a trivial example, suppose we want a function `squarefx` that computes the square of functions of `x`. This can be done as follows.

```
squarefx(double x, double (*f)(double)){
   double temp;
   temp = (*f)(x);
   return temp*temp;
}
```

If we define

```
double g(double y){
   return y/2;
}
```

then

```
squarefx(8, g);                                          (6.14)
```

returns 16.

The quirky notation

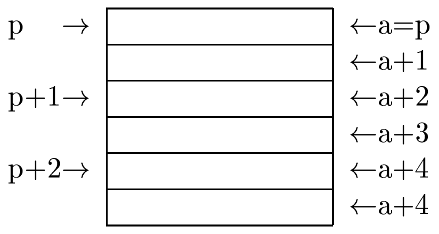
```
double (*f)(double)
```

says that `f` is a pointer to a function of a double that returns a double. This is to be contrasted with the declaration

```
double *f(double)
```

which says that `f` is a function of a double that returns a pointer to a double. There is no need to apply the address operator to `g` in (6.14); since `g` is known to be a function, the compiler does the right thing.

One can add integers to pointers. The result is best illustrated by an example. Suppose `p` is a pointer to a type that consists of two bytes. Consider the following memory diagram.



On the left is shown the memory location pointed to by `p`, `p+1`, etc. On the right is shown the actual address of that memory location in bytes. Of course the memory addresses are initially the same. But each increment of `p` by one skips two bytes of memory.

What is going on here is that C looks at the type of `p`, determines its size of its type, and changes the pointer by multiples of that size. Thus if our object were of type double and we were to imagine an array of doubles stored consecutively in memory beginning at `p`, then `p+1` would point to the second double, `p+2` to the third, `p+3` to the fourth, and so on. All this suggests a close relation between pointer and arrays, which we will now explore.

6.3.8. Arrays

We have already met with **linear** or **1-dimensional arrays** in our `ddot` example. The statement

```
double x[10], y[5];
```

declares `x` to be an array of doubles of length 10 and `y` to be an array of doubles of length 5. In C array indexing begins at zero, so the last element in `x` is `x[9]`. The standard requires that the elements of an array be stored consecutively in memory. An important restriction is that the dimensions defining an array must constant or expressions made up of constants. Thus the construction

```
int fu(int n){
   int work[n];
   . . . . .
}
```

is illegal. Otherwise put, 1-dimensional arrays cannot have variable length.

The variable name `x` standing alone is a pointer. It is a constant pointer; that is, it's value cannot be changed, as can the value of a declared pointer. It points to the beginning of the array, so that `*x` is the same as `x[0]`. Moreover, from our comments on pointer arithmetic it follow that for any integer `*(x+i)` and `x[i]` are the same.

The converse is true. If `a` is a pointer, not necessarily associated with an array, then the expression `a[i]` has the same value as `*(a+i)`.

When an array name appears in an argument list, its value *as a pointer* is passed to the invoked function. This means that the the function can change the elements of the array. For example, here is a simple C version of DAXPY.

```
void daxpy(int n, double a, double x[], double y[]){
   int i;
   for (i=0; i<n; i++)
      y[i] = y[i] + a*x[i];
}
```

We could also do the same thing with pointer arithmetic.

```
void daxpy(int n, double a, double x[], double y[]){
   int i;
   for (i=0; i<n; i++)
      *(y+i) = *(y+i) + a*(*(x+i));
}
```

Thus far the C array looks like a Fortran 77 array with pointer embellishments. But there is an important difference. In Fortran 77 a call like

```
ddot(n-i, x[i], y[i]);
```

will compute the dot product of the last `n-i` elements of `x` and `y`. But this will not work in C, which passes the *values* of `x[i]` and `y[i]`. If you have been careful, and typed `ddot` properly as

```
void ddot(int, double, double[], double[]);
```

then the compiler will catch the inconsistency. If not, you will probably get a memory exception when `ddot` attempts to use a double as an address. The cure is to write

```
ddot(n-i, &x[i], &y[i]);
```

which passes pointers to the appropriate part of the array.

C also has multi-dimensional arrays. The declaration

```
double x[5][7];
```

defines `x` to be a doubly subscripted array of doubles. Two dimensional arrays are stored in row-major order, as illustrated in Figure 3.5. If `x` is an `m`×`n` array, then the address of `x[i][j]` is

```
&x[i][j] = x + j + n*i.                                    (6.15)
```

Because pointer arithmetic is relative to the size of the pointer type, this formula is valid whatever the type of `x`. Note that the expression depends only on `n`—that is the trailing dimension of `x`. This is in contrast with column-major order, in which the address depends on the leading dimension, as in (6.10).

Unfortunately, the restriction that the dimensions in the declaration of an array must be constants or constant expressions means that doubly subscripted arrays are difficult to use effectively. Suppose, for example, that `x` is declared as above, and we wish to pass it to a function—say `fu`. Then somehow we must associate the trailing dimension of `x` with the parameter corresponding to `x`. A Fortran-like construction might define `fu` as

```
double fu(int ldt, double y[][ldt]){
```

and invoke it with

```
fu(7, x);
```

Unfortunately this construction is illegal in C, since `ldt` is not a constant expression. The only proper construction is

```
double fu(double y[][7]){
```

which makes it impossible for `fu` to be used with any 2-dimensional array whose trailing dimension is other than 7.

This inability to pass dimensions through a calling sequence is one of C's prime defects. A cure is to pass a pointer to the array in question along with its trailing dimension and use pointer arithmetic, a la (6.15). A really good optimizing compiler should be able to produce machine code that is as efficient as that produced by ordinary array indexing. However, pointer arithmetic is much harder to read that array indexing. An ad-hoc solution that uses the preprocessor to simulate array indexing is treated in Exercises ??.

6.3.9. Structures

C structures are a way of combining objects of several types into one object. We will begin with a simple example of a structure.

A vector x is **sparse** if most of its elements are zero. It would be inefficient to represent a sparse vector as 1-dimensional C array, since most of the array would be devoted to storing zeros. An alternative is called **compressed representation,** which we now describe.

Suppose our vector x is of dimension `n` and has `nnz` nonzero elements. Then we store the nonzero elements of x in an array `x` of size at least `nnz`. In a parallel array `ix` (for index) we store the indices of the nonzero elements. Thus the vector

$$x = (0, 5, 0, 0, 2, 0, 0, 1)$$

would be stored as

```
x :  5.0  2.0  1.0
ix:    2    5    8
```

Note that the entries in `ix` are required to be strictly increasing. The vector x is completely represented by the variables `n`, `nnz`, `x`, and `ix`.

The trouble with this representation is that to represent more than one vector, we need new names for `n`, `nnz`, `x`, and `ix`. If we have many such vectors, any naming scheme will break down. C solves this problem by arranging these variables in a **`structure`** which can be given a name of its own.

Specifically, we can declare a structure for compressed representation by

```
struct crvec{
   int n;
   int nnz;
   double x[MAXSIZE];
   int ix[MAXSIZE];
};
```
(6.16)

Here `MAXSIZE` is presumed to have been `#defined` previously. We can then declare a number of compressed vectors by the statement

```
struct crvec v1, v2, v3;
```

The `members` of the structure `v1` can be referenced by by writing `v1.n`, `v1.nnx`, `v1.x` and `v1.ix`.

Figure 6.5 contains a function that computes the dot product of `v1` and `v2`. The idea of the program is that only elements `v1[i1]` and `v2[i2]` for which

```
v1.ix[i1]==v2.ix[i2]
```

contribute to the inner product (why?). Thus `spdot` increases now `i1`, now `i2` looking for indices that satisfy the above condition. The code is best understood by working through a simple example to see how `i1` and `i2` leapfrog over each other.

There is one problem with this code. When a structure is passed by value — as it is in `dot` — the entire structure is copied. For large structures this is wasteful of time and memory. An alternative is to pass a pointers to the structures, in which case the function becomes

```
double spdot(struct crvec v1, struct crvec v2){
   int i1=0, i2=0;
   double sum = 0.0;

   while (i1 < v1.nnz && i2 < v2.nnz){
      if (v1.ix[i1] < v2.ix[i2])
          while (v1.ix[i1] < v2.ix[i2])
            i1++;
      else
         while (v2.ix[i2] < v1.ix[i1])
            i2++;

      if (v1.ix[i1] == v2.ix[i2]){
         sum = sum + v1.x[i1]*v2.x[i2];
         i1++; i2++;
      }
   }
   return sum;
}
```

Figure 6.5: Sparse dot product

```
double dot(struct crvec *v1, struct crvec *v2){
```

The notation for reference a member of a structure by a pointer is unusual. For example, to reference `nnz` in `v1` we must write

```
(*v1).nnz
```

The parentheses are required because `.` has precedence over `*`. Since this notation is awkward, C provides an equivalent that is easier to read: namely,

```
v1->ix[i1].
```

Thus to convert the code in Figure 6.5 all one needs to do is replace `.`'s with `->`'s.

In designing structures, it is important to keep in mind that its members appear in memory in the order they appear in the structure definition and they are properly aligned. This can cause **memory fragmentation,** of which (2.2) is an example. To drive the point home, consider the following two structures.

```
struct x{            struct y{
   double a;            double a;
   double b;            int c;
   int c;               double b;          (6.17)
   int d;               int d;
};                    };
```

Assuming that the size of a double is eight bytes and the size of an int is four bytes, the structure `x` requires 24 bytes where as the functionally equivalent structure `y` requires 32 bytes. The difference is unimportant if only a few of these structures are to be declared. But if a large array is needed, then the savings for using `x` over `y` can be considerable.

There are more to structures that this. For example, a structure cannot have itself as member, but it can have a pointer to itself. This is useful in creating linked lists of structures. Moreover, there are other constructions that bind things together like structures. But the present exposition gives the flavor of this important construct.

6.3.10. Memory allocation

A difficulty with the structure `crvec` is that its arrays are all of length `MAXSIZE`. There are two ways in which this is unsatisfactory. First, if we encounter a vector with `nnz > MAXSIZE` we cannot use the structure `prvec`. Second, it wastes memory if most of the vectors at hand have fewer than `MAXSIZE` nonzero components. What we would really like is to have a function

```
construct(struct prvec *v, int nnz)
```

that initializes the storage for the arrays `v->x` and `v->ix`. We can write such a function using the standard library function MALLOC.

The first thing that we must do is change the definition of `prvec` to get rid of the initialization by `MAXSIZE`:

```
struct prvec{
   int n;
   int nnz;
   double *x;
   int *ix;
};
```

Because of the relation between pointers and vectors, we can still use the notation `x[i]` to refer to the `i`th element of `x`. The code for `construct` is now

```
void construct(struct prvec *v, int nnz){
   v->nnz = nnz;
   v->x = (double *) malloc(nnz, sizeof(double));
   v->ix = (int *) malloc(nnz, sizeof(int));
}
```

To see what is happening, let us look at what the statement

```
      v->x = (double *) malloc(nnz, sizeof(double));
```

is doing. The heart of the matter is the invocation of `malloc`, which allocates storage. The first argument `nnz` specifies the number of aligned storage units to return. The second argument says that the size of a storage unit is to be the size of the type double. Thus `malloc` returns (for IEEE double) a pointer to `8*nnz` consecutive bytes of memory properly aligned for double. The pointer is of type void, which is not the same as a pointer of type double, consequently the cast `(double *)` is needed to convert it before it is assigned to `v->x`.

The function `malloc` obtains storage from the **heap**, a pool of free memory. The heap is not infinite, however, and one should free up heap memory when it is not in use. For our structure we might write a special function `destruct` to free the values in a `prvec`. Here is how it is coded.

```
void destruct(struct prvec *v){
   free(v->x);
   free(v->ix);
}
```

Note that if `v` were repeatedly constructed but never destructed, say in a loop in which vector sizes vary, then the heap will eventually run out of memory. This situation is called a **memory leak** and is a frequent source of program failures in C.

6.4. Fortran 95

Fortran 95 is a minor extension of Fortran 90, which in turn is a major reworking of Fortran 77. Although it is backward compatible with Fortran 77, it looks and programs like a different language. It is widely used in Europe; less widely so in the United States. But it is well worth considering for scientific computing. Its designers crafted its new features, such as pointers, so that the language remains easy to optimize. The Fortran 95 module is a flexible device for organizing large programs and packages. And its array features are more supple than those of any other programming language. However, it not really suitable for object oriented programming (which we will treat in the next subsection in connection with C++).[10]

[10]A new extension, Fortran 2003 supports full object oriented programming. But it has just been released and compilers are not readily available.

```
integer parameter wp = kind(0.0D0) ! Working precision

subroutine daxpy1(n, a, x, y)

!  axpy computes y = y + a*x, where

   integer  intent(in)    :: n     ! The size of the vectors.
                                   ! x and y.
   real(wp) intent(in)    :: a,  & ! a constant.
                             x(:)  ! The vector x.
   real(wp) intent(inout) :: y(:)  ! The vector y.

  jloop: do j=1,n
            if (a == 0) exit jloop
            do i=1,n
               y(i) = y(i) + a*x(i)
            end do
         end do jloop
  end subroutine daxpy1
```

Figure 6.6: Stride 1 AXPY

Space does not allow us to discuss Fortran 95 (and later C++) at the level of detail we treated Fortran 77 and C. We will begin with a quick sketch of some of the basic extensions of Fortran 77 and then turn to the new features important for scientific computing.

6.4.1. Basic extensions

Here we will treat extensions under the heading of format, types, and control.

• **Format** Fortran 95 remains a line oriented language: statements are terminated by an end of line. However it has shed the fixed fields of Fortran 77, and has more flexible commenting and continuation conventions. This is illustrated in Figure 6.6 which contains a stripped down AXPY with a default stride of one. A comment begins with an exclamation point and continues to the end of its line. An ampersand (`&`) causes the statement to be continued to the next line.

• **types** Fortran 95 has the same basic types as Fortran 77: `integer`, `real`, `complex`, `logical` and `character`. But it provides a new mechanism for specifying variations of these basic types. Specifically, each type supported by a system has a system-defined number called its kind associated with it, which can be retrieved by the `kind` function. For example, line 1 in Figure 6.6 creates an integer parameter `wp` (for 'working precision') that is the kind of a double precision real. Its appearance in line 9 causes `a` and `x` to be declared to be double precision real. Just changing line 1 to

```
integer parameter wp = kind(0.0E0)
```

would change `daxpy1` to a single precision routine.

The specification statement has a new form that allow you to pile up specifications. Of these, the `intent` specification tells the compiler the read and write status of a dummy argument.

The **defined type** is the equivalent of the C structure. For example, the Fortran equivalent of the structure `crvec` in (6.16) is

```
type crvec
   integer  :: n
   integer  :: nnz
   real(wp) :: x(MAXSIZE)
   integer  :: ix(MAXSIZE)
```

Here is is assumed that `wp` and `MAXSIZE` have been previously defined. Variables of type `crvec` can be declared as follows.

```
type(crvec) :: x, y, z
```

The implicit typing of Fortran 77 can be suppressed by the

```
implicit none
```

statement, which must appear at the start of the programming unit. Its universal use is strongly recommended.

• **Control** Fortran has completed the usual complement of control constructs by the addition of a switch-case statement. The `do` construct has been overhauled in several ways.

1. A `do` can be terminated by an `end do`, which eliminates the need for numeric labels.
2. The statement `do` alone produces and infinite loop.
3. A `do while(<exp>)` has been added. The loop is executed as long as `<exp>` evaluates to true.
4. An `exit` statement causes the innermost loop containing the statement to be terminated. A `cycle` statement causes the innermost loop containing the statement to be restarted.

5. A name, which is not to be confused with a label, can be attached to a do, as illustrated by lines 13 and 18 in Figure 6.6. Passively, this device makes it easy for the compiler to check for correct loop nesting. Actively, the `exit` and `cycle` can name the loop they affect. This is illustrated in line 14, although in this case the name is redundant.

6.4.2. Program organization

The basic programming units for Fortran 95 are the program, the subroutine, the function, and the module. Fortran 95 inherits the program statement as well as subroutines and functions of Fortran 77. An important extension is that both subroutines and functions can be invoked recursively.

Another useful extension is the ability to declare arguments to be optional. For example, consider the following subroutine.

```
subroutine getvals(A, v1, v2, v3, v4)
   type(source)       :: A       ! A derived type from which
                                 !    v1,...,v4 is to be computed.
   real(wp)           :: v1, v2  ! Values that are always returned.
   real(wp), optional :: v3, v4  ! Values that are optionally
                                 ! returned.
```

The calling sequence

```
call getvals(A, v1, v2, v3)
```

will return the values `v1`, `v2`, and `v3`, but not `v4`. If you want `v4` but not `v3` `v3`, you can write.

```
call getvals(A, v1, v2, v4=valfour),
```

which will return the fourth value in the variable `valfour`. The presence of an optional argument can be test by the `present` function.

A subroutine or function may contain `internal functions` or `subroutines`. These subprograms can see the variables declared in the containing subprogram. They are useful for performing tasks that are particular to the containing subprogram.

The Fortran 95 **module,** which is new to Fortran, is an elegant way of dividing programs into manageable pieces. It has the form

```
module <module name>
   <definitions and specifications>
contains
   <subprograms>
end module <module name>
```

For example, suppose we want to bundle `axpy` and `dot` into a single package. Then we might proceed as follows.

```
module axpydot
implicit none
   integer parameter :: wp = kind(0.0D0)
contains
   <code for the subroutine axpy>
   <code for the function dot>
end module axpydot
```

A programming unit can invoke a module by the `use` statement. For example a program that needs both `axpy` and `dot` might begin

```
subroutine fubar(<parameter list>)
  use axpydot
  implicit none
  ...
end subrouting fubar
```

Programs using a module can access the definitions and subprograms contained in the module. This fact largely obviates the need for global variables and `common` statements. Variables and functions that must be shared among programming units can be placed in modules where they can be `use`d as required. Because modules can use other modules, the modularization of a program or package can be quite fine-grained. Commonly used groups of modules can then be collected into larger modules, so that a programmer can access a group with one `use` statement.

Exercises

1. The following doubly recursive Matlab function, computes the sequence of Fibonacci numbers.

```
function m = fib(n)
   if n==0 | n==1,
      m=1;
   else
      m = fib(n-1) + fib(n-2);
   end
return
```

1. Run this program for `n=1:5;30`. What (approximately) is the ratio `fib(n+5)/fib(n)`?

2. Assume that `fib(0)` and `fib(1)` take time $T_1 = T_2 = 1$ to execute, and that for larger values of n the time T_n is the sum of the times to evaluate `fib(n-1)` and `fib(n-2)`. What is T_n.
3. What are the implications of the first two parts for the function `fib`? What about double recursion in general.