

7 Data Types

7.9 Files and Input/Output

The first two subsections below are devoted to interactive and file-based I/O, respectively. Section ©7.9.3 then considers the common special case of text files.

7.9.1 Interactive I/O

On a modern machine, interactive I/O usually occurs through a graphical user interface (GUI: “gooey”) system, with a mouse, a keyboard, and a bit-mapped screen that in turn support windows, menus, scrollbars, buttons, sliders, and so on. GUI characteristics vary significantly among, say, Microsoft Windows, the Macintosh, and Unix’s X11; the differences are one of the principal reasons it is difficult to port applications across platforms.

Within a single platform, the facilities of a GUI system usually take the form of library routines (to create or resize a window, print text, draw a polygon, and so on). Input events (mouse move, button push, keystroke) may be placed in a queue that is accessible to the program, or tied to *event handler* subroutines that are called by the run-time system when the event occurs. Because the handler is triggered from outside, its activities must generally be *synchronized* with those of the main program, to make sure that both parties see a consistent view of any data shared between them. We will discuss events further in Section 8.7, and synchronization in Section 12.3.

A few programming languages—notably Smalltalk and Java—attempt to incorporate a standard set of GUI mechanisms into the language. The Smalltalk design team was part of the original group at Xerox’s Palo Alto Research Center (PARC) that invented mouse-and-window based interfaces in the early 1970s. Unfortunately, while the Smalltalk GUI is successful within the confines of the language, it tends not to integrate well with the “look and feel” of the host system on which it runs. In a similar vein, Java’s original GUI facilities (the Abstract Window Toolkit—AWT) had something of a “least common denominator” look to them.

Smalltalk's GUI is a fundamental part of the language; Java's takes the form of a standard set of library routines. The Java routines and their interface have evolved significantly over time; the current "Swing" libraries have a "pluggable" look and feel, allowing them to integrate more easily with (and port more easily among) a variety of window systems.

The "parallel execution" of the program and the human user that characterizes interactive systems is difficult to capture in a functional programming model. A functional program that operates in a "batch" mode (taking its input from a file and writing its output to a file) can be modeled as a function from input to output. A program that interacts with the user, however, requires a very concrete notion of program ordering, because later input may depend on earlier output. If both input and output take the form of an ordered sequence of tokens, then interactive I/O can be modeled using lazy data structures, a subject we considered in Section 6.6.2. More general solutions can be based on the notion of *monads*, which use a functional notion of sequencing to model side effects. We will consider these issues again in Sections 10.4 and 10.7.

7.9.2 File-Based I/O

Persistent files are the principal mechanism by which programs that run at different times communicate with each other. A few language proposals (e.g., Argus [LS83] and χ [SH92]) allow ordinary variables to persist from one invocation of a program to the next, and a few experimental operating systems (e.g., Opal [CLFL94] and Hemlock [GSB⁺93]) provide persistence for variables outside the language proper. In addition, some language-specific programming environments, such as those for Smalltalk and Common Lisp, provide a notion of *workspace* that includes persistent named variables. These examples, however, are more the exception than the rule. For the most part, data that need to outlive a particular program invocation need to reside in files.

Like interactive I/O, files can be incorporated directly into the language, or provided via library routines. In the latter case, it is still a good idea for the language designers to suggest a standard library interface, to promote portability of programs across platforms. The lack of such a standard in Algol 60 is widely credited with impeding the language's widespread use. One of the principal reasons to incorporate I/O into the language proper is to make use of special syntax. In particular, several languages, notably Fortran and Pascal, provide built-in I/O facilities in order to obtain type-safe "subroutines" that take a variable number of parameters, some of which may be optional.

Depending on the needs of the programmer and the capabilities of the host operating system, data in files may be represented in binary form, much as it is in memory, or as *text*. In a binary file, the number 1066_{10} would be represented by the 32-bit value 10000101010_2 . In a text file, it would probably be represented by the character string "1066". Temporary files are usually kept in binary form for the sake of speed and convenience. Persistent files are commonly kept in both

forms. Text files are more easily ported across systems: issues of word size, byte order, alignment, floating-point format, and so on do not arise. Text files also have the advantage of human readability: they can be manipulated by text editors and related tools. Unfortunately, text files tend to be large, particularly when used to hold numeric data. A double-precision floating-point number occupies only eight bytes in binary form, but can require as many as 24 characters in decimal notation (not counting any surrounding white space). Text files also incur the cost of binary to text conversion on output, and text to binary conversion on input. The size problem can be addressed, at least for archival storage, by using data compression. Mechanisms to control text/binary conversion tend to be the most complicated part of I/O; we discuss them in the following subsection.

When I/O is built into a language, files are usually declared using a built-in type constructor, as they are in Pascal:

```
var my_file : file of foo;
```

EXAMPLE 7.136

Files as a built-in type

If I/O is provided by library routines, the library usually provides an opaque type to represent a file. In either case, each file variable is generally bound to an external, operating system–supported file by means of an *open* operation. In C, for example, one says:

```
my_file = fopen(path_name, mode);
```

EXAMPLE 7.137

The open operation

The first argument to `fopen` is a character string that names the file, using the naming conventions of the host operating system. The second argument is a string that indicates whether the file should be readable, writable, or both, whether it should be created if it does not yet exist, and whether it should be overwritten or appended to if it does exist.

EXAMPLE 7.138

The close operation

When a program is done with a file, it can break the association between the file variable and the external object by using a *close* operation:

```
fclose(my_file);
```

In response to a call to `close`, the operating system may perform certain “finalizing” operations, such as unlocking an exclusive file (so that it may be used by other programs), rewinding a tape drive, or forcing the contents of buffers out to disk.

Most files, both binary and text, are stored as a linear sequence of characters, words, or records. Every open file then has a notion of *current position*: an implicit reference to some element of the sequence. Each `read` or `write` operation implicitly advances this reference by one position, so that successive operations access successive elements, automatically. In a *sequential* file, this automatic advance is the only way to change the current position. Sequential files usually correspond to media like printers and tapes, in which the current position has a physical representation (how many pages we’ve printed; how much tape is on each spool) that is difficult to change.

In other, *random-access* files, the programmer can change the current position to an arbitrary value by issuing a *seek* operation. In a few programming languages (e.g., Cobol and PL/I), random-access files (also called *direct* files) have no notion of current position. Rather, they are *indexed* on some key, and every read or write operation must specify a key. A file that can be accessed both sequentially and by key is said to be *indexed sequential*.

Random-access files usually correspond to media like magnetic or optical disks, in which the current position can be changed with relative ease. (Depending on technology, modern disks take anywhere from 5 to 200 ms to seek to a new location. Tape drives, by contrast, can take more than a minute. Note that 5 ms is still a very long time—ten million cycles on a 2 GHz processor—so seeking should never be taken casually, even on a disk.) A few languages—notably Pascal—provide no random-access files, though individual implementations may support random access as a nonstandard language extension.

7.9.3 Text I/O

It is conventional to think of text files as consisting of a sequence of *lines*, each of which in turn consists of characters. In older systems, particularly those designed around the metaphor of punch cards, lines are reflected in the organization of the file itself. A *seek* operation, for example, may take a line number as argument. More commonly, a text file is simply a sequence of characters. Within this sequence, control (nonprinting) characters indicate the boundaries between lines. Unfortunately, end-of-line conventions are not standardized. In Unix, each line of a text file ends with a *newline* (“control-J”) character, ASCII value 10. On the Macintosh, each line ends with a *carriage return* (“control-M”) character, ASCII value 13. In MS-DOS and Windows, each line ends with a carriage return/line feed pair. Text files are usually sequential.

Despite the muddled conventions for line breaks, text files are much more portable and readable than binary files.¹ Because they do not mirror the structure of internal data, text files require extensive conversions on input and output. Issues to be considered include the base for integer values (and the representation of nondecimal bases); the representation of floating-point values (number of digits, placement of decimal point, notation for exponent); the representation of enumerations and other nonnumeric, nonstring types; and positioning, if any, within columns (right and left justification, zero or white-space fill, “floating” dollar signs in Cobol). Some of these issues (e.g., the number of digits in a floating-point number) are influenced by the hardware, but most are dictated by the needs of the application and the preferences of the programmer.

¹ We are speaking here, of course, of plain text ASCII or Unicode files. So-called “rich text” files, consisting of formatted text in particular fonts, sizes, and colors, perhaps with embedded graphics, are another matter entirely. Word processors typically represent rich text with a combination of binary and ASCII data, though ASCII-only standards such as Postscript, textual PDF, and RTF can be used to enhance portability.

In most languages the programmer can take complete control of input and output formatting by writing it all explicitly, using language or library mechanisms to read and write individual characters only. I/O at such a low level is tedious, however, and most languages also provide more high-level operations. These operations vary significantly in syntax and in the degree to which they allow the programmer to specify I/O formats. We illustrate the breadth of possibilities with examples from four imperative languages: Fortran, Ada, C, and C++.

Text I/O in Fortran

EXAMPLE 7.139

Formatted output in Fortran

In Fortran, we could write a character string, an integer, and an array of ten floating-point numbers as follows:

```
character s*20
integer n
real r (10)
...
write (4, '(A20, I10, 10F8.2)'), s, n, r
```

In the `write` statement, the 4 indicates a *unit number*, which identifies a particular output file. The quoted, parenthesized expression is called a *format*; it specifies how the printed variables are to be represented. In this case, we have requested a 20-column ASCII string, a 10-column integer, and 10 eight-column floating-point numbers (with two columns of each reserved for the fractional part of the value). Fortran provides an extremely rich set of these *edit descriptors* for use inside of formats. Cobol, PL/I, and Perl provide comparable facilities, though with a very different syntax. ■

EXAMPLE 7.140

Labeled formats

Fortran allows a format to be specified indirectly, so it may be used in more than one input or output statement:

```
write (4, 100), s, n, r
...
100 format (A20, I10, 10F8.2)
```

It also allows formats to be created at run time, and stored in strings:

```
character(len=20) :: fmt
...
fmt = "(A20, I10, 10F8.2)"
...
write (4, fmt), s, n, r
```

If the programmer does not know, or does not care about, the precise allocation of columns to fields, the format can be omitted:

```
write (4, *), s, n, r
```

EXAMPLE 7.14

Printing to standard output

In this case, the run-time system will use default format conventions. ■

To write to the standard output stream (i.e., the terminal or its surrogate), the programmer can use the `print` statement, which resembles a `write` without a unit number:

```
print*, s, n, r          ! * means default format
```

For input, `read` is used both for standard input and for specific files; in the former case, the unit number is omitted, together with the extra set of parentheses:

```
read 100, s, n, r
...
read*, s, n, r          ! * means default format
```

The star may be omitted in Fortran 90. ■

In the full form of `read`, `write`, and `print`, additional arguments may be provided in the parenthesized list with the unit number and format. These can be used to specify a variety of additional information, including a label to which to jump on end-of-file, a label to which to jump on other errors, a variable into which to place status codes returned by the operating system, a set of labels (a “namelist”) to attach to the output values, and a control code to override the usual automatic advance to the next line of the file. Because there are so many of these optional arguments, most of which are usually omitted, they are usually specified using *named* (keyword) parameter notation, a notion we defer to Section 8.3.3.

The variety of shorthand versions of `read`, `write`, and `print`, together with the fact that they operate on a variable number of program variables, makes it very difficult to cast them as “ordinary” subroutines. Fortran 90 provides default and keyword parameters (Section 8.3.3), but Fortran 77 does not, and even in Fortran 90 there is no way to define a subroutine with an *arbitrary* number of parameters.

In Pascal, as in Fortran 77, the parameters of every subroutine are fixed in number and in type. Pascal’s `read`, `readln`, `write`, and `writeln` “routines” are therefore built into the language; they are not part of a library. Each takes a variable number of arguments, the first of which may optionally specify a particular file. Unfortunately, Pascal’s formatting mechanisms are much less flexible than those of Fortran; programmers are often forced to implement formatting by hand, using `read` and `write` for input and output of individual characters only. In the design of Modula-2, Niklaus Wirth chose to move I/O out of the language proper, and to embed it in a standard library. The designers of Ada took a similar approach. The Modula-2 I/O libraries are relatively primitive: only a modest improvement over character-by-character I/O in Pascal. The Ada libraries are much more extensive, and make heavy use of overloading and default parameters.

Text I/O in Ada

Ada provides a suite of five standard library packages for I/O. The `sequential_IO` and `direct_IO` packages are for binary files. They provide generic file types that can be instantiated for any desired element type. The `IO_exceptions` and `low_level_IO` packages handle error conditions and device control, respectively. The `text_IO` package provides formatted input and output on sequential files of characters.

EXAMPLE 7.142

Formatted output in Ada

Using `text_IO`, our original three-variable Fortran output statement would look something like this in Ada:

```
s : array (1..20) of character;
n : integer;
r : array (1..10) of real;
...
set_output(my_file);
put(n, 10);
put(s);
for i in 1..10 loop put(r(i), 5, 2); end loop;
new_line;
```

In the `put` of an element of `r` (within the loop), the second parameter specifies the number of digits before the decimal point, rather than the width of the entire number (including the decimal point), as it did in Fortran. The `put` of `s` will use the string's natural length. If a different length is desired, the programmer will have to write blanks or put a substring explicitly. If precise output positioning is not desired for the integers and real numbers, the extra parameters in their `put` calls can be omitted; in this case the run-time system will use standard defaults. The programmer can use additional library routines to change these defaults if desired. A call to `set_output` invokes a similar mechanism: it changes the default notion of output file. ■

EXAMPLE 7.143Overloaded `put` routines

There are two overloaded forms of `put` for every built-in type. One takes a file name as its first argument; the other does not. The last five lines above could have been written:

```
put(my_file, n, 10);
put(my_file, s);
for i in 1..10 loop put(my_file, r(i), 5, 2); end loop;
new_line(my_file);
```

The programmer can of course define additional forms of `get` and `put` for arbitrary user-defined types. All of these facilities rely on standard Ada mechanisms; in contrast to Fortran, no support for I/O is built into the language itself. ■

Text I/O in C

C provides I/O through a library package called `stdio`; as in Ada, no support for I/O is built into the language itself. Many C implementations, however, build

knowledge of I/O functions into the compiler, so it can issue warnings when arguments appear to be used incorrectly.

EXAMPLE 7.144

Formatted output in C

Our example output statement would look something like this in C:

```
char s[20];
int n;
double r[10];
...
fprintf(my_file, "%20s%10d", s, n);
for (i = 0; i < 10; i++) fprintf(my_file, "%8.2f", r[i]);
fprintf(my_file, "\n");
```

The arguments to `fprintf` are a file, a format string, and a sequence of expressions. The format string has capabilities similar to the formats of Fortran, though the syntax is very different. In general, a format string consists of a sequence of characters with embedded “placeholders,” each of which begins with a percent sign. The placeholder `%20s` indicates a 20-character string; `%d` indicates an integer in decimal notation; `%8.2f` indicates an 8-character floating-point number, with two digits to the right of the decimal point. ■

As in Fortran, formats can be computed and stored in strings, and a single `fprintf` statement can print an arbitrary number of expressions. As in Ada, an explicit `for` loop is needed to print an array. Commonly the format string also contains labeling text and white space:

EXAMPLE 7.145

Text in format strings

```
strcpy(s, "four"); /* copy "four" into s */
n = 20;
char *fmt = "%s score and %d years ago\n";
fprintf(my_file, fmt, s, n);
```

A percent sign can be printed by doubling it:

```
fprintf(my_file, "%d%%\n", 25); /* prints "25%" */
```

EXAMPLE 7.146

Formatted input in C

Input in C takes a similar form. The `fscanf` routine takes as argument a file, a format string, and a sequence of pointers to variables. In the common case, every argument after the format is a variable name preceded by a “pointer to” operator:

```
fscanf(my_file, "%s %d %lf", &s, &n, &r[0]);
```

In this call, the `%s` placeholder will match a string of maximal length that does not include white space. If this string is longer than 20 characters (the length of `s`), then `fscanf` will write beyond the end of the storage for the string. (This weakness in `scanf` is one of the sources of the so-called “buffer overflow” bugs discussed in the sidebar on page 353.) The three-character `%lf` placeholder informs the library routine that the corresponding argument is a `double`; the two-character

sequence `%f` would read into a `float`.² Accidentally using a placeholder for the wrong size variable is a common error in older implementations of C; forgetting the ampersand on a trailing argument is another. While such mistakes will often be caught by a modern C compiler with special-case knowledge of `fscanf`, they would always be caught in a language with type-safe I/O. Note that we have read a single element of `r`; as with `fprintf`, a `for` loop would be needed to read the whole array. ■

We have noted above that the I/O routines of Fortran and Pascal are built into the language largely to permit them to take a variable number of arguments. We have also noted that moving I/O into a library in Ada forces us to make a separate call to `put` for every output expression. So how do `fprintf` and `fscanf` work? It turns out that C permits functions with a variable number of parameters (we will discuss such functions in more detail in Section 8.3.3). Unfortunately, the types of trailing parameters are unspecified, which makes compile-time type checking of variable-length argument lists impossible in the general case. Moreover, the lack of run-time type descriptors in C precludes run-time checking as well. At the same time, because the C library (including `fprintf` and `fscanf`) is part of the language standard, special knowledge of these routines can be built into the compiler—and often is: while the I/O routines of C are formally defined as “ordinary” functions, they are typically implemented in the same way as their analogues in Fortran and Pascal. As a result, C compilers will often provide good error diagnostics when the arguments to `fprintf` or `fscanf` do not match the format string.

To simplify I/O to and from the standard input and output streams, `stdio` provides routines called `printf` and `scanf` that omit the initial arguments of `fprintf` and `fscanf`. To facilitate the formatting of strings *within* a program, `stdio` also provides routines called `sprintf` and `sscanf`, which replace the initial arguments of `fprintf` and `fscanf` with a pointer to an array of characters. The `sscanf` function “reads” from this array; `sprintf` “writes” to it. Fortran 90 provides similar support for intraprogram formatting through so-called *internal files*.

Text I/O in C++

As a descendant of C, C++ supports the `stdio` library described in the previous subsection. It also supports a new I/O library called `iostream` that exploits the object-oriented features of the language. The `iostream` library is more flexible than `stdio`, provides arguably more elegant syntax (though this is a matter of taste), and is completely type safe.

² C’s `doubles` are double-precision IEEE floating-point numbers in most implementations; `floats` are usually single precision. The lack of safety for `%s` arguments is only one of several problems with `fscanf`. Others include the inability to “skip over” erroneous input, and undefined behavior when there is insufficient input. Instead of `fscanf`, seasoned C programmers tend to use `fgets`, which reads (length-limited) input into a string, followed by manual parsing using `strtol` (string-to-long), `strtod` (string-to-double), and so on.

EXAMPLE 7.147

Formatted output in C++

C++ *streams* use operator overloading to co-opt the << and >> symbols normally used for bit-wise shifts. The `iostream` library provides an overloaded version of << and >> for each built-in type, and programmers can define versions for new types. To print a character string in C++, one writes

```
my_stream << s;
```

To output a string and an integer one can write

```
my_stream << s << n;
```

This code requires that `my_stream` be an instance of the `ostream` (output stream) class defined in the `iostream` library. The << operator is syntactic sugar for the “operator function” `ostream::operator<<`, as described in Section 3.5.2. Because << associates left-to-right, the statement above is equivalent to

```
(my_stream.operator<<(s)).operator<<(n);
```

The code works because `ostream::operator<<` returns a reference to its first argument as its result (as we shall see in Section 8.3.1, C++ supports both a value model and a reference model for variables). ■

EXAMPLE 7.148

Stream manipulators

As shown so far, output to an `ostream` uses default formatting conventions. To change conventions, one may embed so-called *stream manipulators* in a sequence of << operations. To print `n` in octal notation (rather than the default decimal), we could write

```
my_stream << oct << n;
```

To control the number of columns occupied by `s` and `n`, we could write

```
my_stream << setw(20) << s << setw(10) << n;
```

The `oct` manipulator causes the stream to print all subsequent numeric output in octal. The `setw` manipulator causes it to print its next string or numeric output in a field of a specified minimum width (behavior reverts to the default after a single output). ■

The `oct` manipulator is declared as a function that takes an `ostream` as a parameter and produces a reference to an `ostream` as its result. Because it is not followed by empty parentheses, the occurrence of `oct` in the output sequence above is *not* a call to `oct`; rather, a reference to `oct` is passed to an overloaded version of << that expects a manipulator function as its right-hand argument. This version of << then calls the function, passing the stream (the left-hand argument of <<) as argument.

The `setw` manipulator is even trickier. It is declared as a function that returns a reference to what we might call an “object closure”—an object containing a reference to a function and a set of arguments. In this particular case, `setw(20)`

is a call to a *constructor* function that returns a closure containing the number 20 and a pointer to the `setw` manipulator. (We will discuss constructors in detail in Section 9.3, and object closures in Section 3.6.3.) The `iostream` library provides an overloaded version of `<<` that expects an object closure as its right-hand argument. This version of `<<` calls the function inside the closure, passing it as arguments the stream (the left-hand argument of `<<`) and the integer inside the closure.

The `iostream` library provides a wealth of manipulators to change the formatting behavior of an `ostream`. Because C++ inherits C's handling of pointers and arrays, however, there is no way for an `ostream` to know the length of an array. As a result, our full output example still requires a `for` loop to print the `r` array:

EXAMPLE 7.149

Array output in C++

```
char s[20];
int n;
double r[10];
...
my_stream << setw(20) << s << setw(10) << n;
for (i = 0; i < 10; i++)
    my_stream << setiosflags(ios::fixed)
        << setw(8) << setprecision(2) << r[i];
my_stream << "\n";
```

Here the manipulators in the output sequence in the `for` loop specify fixed format (rather than scientific) for floating-point numbers, with a field width of eight, and two digits past the decimal point. The `setiosflags` and `setprecision` manipulators change the default format of the stream; the changes apply to all subsequent output. ■

EXAMPLE 7.150

Changing default format

To avoid calling stream manipulators repeatedly, we could modify our example as follows:

```
my_stream.flags(my_stream.flags() | ios::fixed);
my_stream.precision(2);
for (i = 0; i < 10; i++) my_stream << setw(8) << r[i];
```

The `setw` manipulator affects the output width of only a single item. To facilitate the restoration of defaults, the `flags` and `precision` functions return the previous value:

```
ios::fmtflags old_flags =
    my_stream.flags(my_stream.flags() | ios::fixed);
int old_precision = my_stream.precision(2);
for (i = 0; i < 10; i++) my_stream << setw(8) << r[i];
my_stream.flags(old_flags);
my_stream.precision(old_precision);
```

Formatted input in C++ is analogous to formatted output. It uses `istream`s instead of `ostream`s, and the `>>` operator instead of `<<`. It also supports a suite of

manipulators comparable to those for output. I/O on the standard input and output streams does not require different functions; the programmer simply begins an input or output sequence with the standard stream name `cin` or `cout`. (In keeping with C tradition, there is also a standard stream `cerr` for error messages.) To support intraprogram formatting of character strings, the `stringstream` library provides `istringstream` and `ostringstream` object classes that are derived from `istream` and `ostream`, and that allow a stream variable to be bound to a string instead of to a file.

✓ CHECK YOUR UNDERSTANDING

79. Explain the differences between interactive and file-based I/O, between temporary and persistent files, and between binary and text files. (Some of this information is in the main text.)
 80. What are the comparative advantages of *text* and *binary* files?
 81. Describe the end-of-line conventions of Unix, Windows, and Macintosh files.
 82. What are the advantages and disadvantages of building I/O into a programming language, as opposed to providing it through library routines?
 83. Summarize the different approaches to text I/O adopted by Fortran, Ada, C, and C++.
 84. Describe some of the weaknesses of C's `scanf` mechanism.
 85. What are *stream manipulators*? How are they used in C++?
-