Chapter 4

Features of Programming Languages

The preceding chapter described the programming process as starting with a clearly specified task, expressing it mathematically as a set of algorithms, translating the algorithms in pseudocode, and finally, translating the pseudocode into a “real” programming language. The final stages of this prescription work because most (if not all) computational languages have remarkable similarities: They have statements, the sequencing of which are controlled by various loop and conditional constructs, and functions that foster program modularization. We indicated how similar MATLAB, C++, and Fortran are at this level, but these languages differ the more they are detailed. It is the purpose of this chapter to describe those details, and bring you from a superficial acquaintance with a computational language to fluency. Today, the practicing engineer needs more than one programming language or environment. Once achieving familiarity with one, you will find that learning other languages is easy.

When selecting a programming tool for engineering calculations, one is often faced with two different levels of need. One level is where you need to quickly solve a small problem once, such as a homework assignment, and computational efficiency is not important. You may not care if your code takes ten seconds or one hundred seconds to execute; you want convenience. At that level it may make sense to use an engineering environment like MATLAB, or Mathematica. At the other extreme you may be involved in doing a wide area weather prediction where a one-day run time, instead of a ten-day run time, defines a useful versus a non-useful product. You might be developing a hospital laboratory system for reporting test results to an emergency room physician where an answer in ten seconds versus an answer in ten minutes can literally mean the difference between life or death for a patient. For programming at this level one wants an efficient language. Since such projects can involve programming teams in different countries, you want your language to be based on an international standard. Then you would choose to program a language such as C++ or F90. Since most students have experienced only the first need level, they tend to overvalue the first approach and devalue the second. This chapter will illustrate that the skills needed for either approach are similar.

The structure of this chapter follows our usual progression to learning a language: What are variables, how can variables be combined into expressions, what constructs are available to control program flow, and how are functions defined so that we can employ modularity. The basics are described in Chapter 1; we assume you are familiar with the language basics described there. Initially, this chapter will parallel the program composition section of Chapter 1 as applied in the C++, F90, and MATLAB languages, and then it will bring in more advanced topics.

The features of F90 that are to be discussed here have been combined in a series of tables and placed in Appendix B. It is expected that we will want to refer to those tables as we read this section as well as later when we program. At times, references to C++ and MATLAB have been given to show the similarities between most languages and to provide an aid for when having to interface in reading codes in those languages.

4.1 Comments

In MATLAB and Fortran, a single character — ‘%’ in MATLAB, ‘!’ in F90 — located anywhere in a line of text means that the remainder of the text on that line comprises the comment. In C, an entirely different
Table 4.1: Comment syntax

structure for comments occurs. Comments begin with the two-character sequence ‘//’ and end with the
next occurrence of the two-character sequence ‘*/’. In C, comments can occur anywhere in a program;
they can consume a portion of a line, temporarily interrupting a statement, or they can span multiple
lines of text. C++ allows the use of the C comment syntax, but has added a more popular two-character
sequence ‘//’ to proceed a comment to the end of a line. Table 4.1 gives a summary of these comments
syntax. It is also in the “Fortran 90 Overview” for quick reference. Samples of comment statements are
shown in Fig. 1.3, which gives the corresponding versions of the classic “hello world” program included
in most introductory programming texts.

4.2 Statements and Expressions

Before introducing statements and expressions, a word about documenting what you program. We en-
courage the heavy usage of comments. The three languages of concern here all allow comment lines and
comments appended to the end of statements. Their form is given above in Fig. 1.3 and Table 4.1.

The above languages currently allow variable names to contain up to 31 characters and allow the use
of the underscore, ‘_’, to aid in clarity by serving as a virtual space character, as in my_name. Another
useful convention is to use uppercase first letters for words comprising part of a variable’s name: MyName.
Fortran and MATLAB allow a program line to contain up to 132 characters, while C++ has no limit on
line length. Since the old F77 standard was physically limited to holes punched in a card, it allowed only
a line length of 72 characters, a maximum name length of six characters, and did not allow the use of the
underscore in a name. In this text, we will usually keep line lengths to less than 65 characters in order to
make the programs more readable.

A statement in these three languages has a structure common to them all:

variable = expression

The built-in, or intrinsic, data types allowed for variables are summarized in Table 4.2. Additional user
defined types will be considered later. The expressions usually involves the use of arithmetic operators
and/or relational operators which are given in Tables 4.3 and 4.4, respectively. The order in which the
language applies these operators is called their precedence, and they are shown in Table 4.5. They are
also in the “Fortran 90 Overview” for quick reference.

In moving from MATLAB to high level languages one finds that it is necessary to define the type of
each variable. Fortran has a default naming convention for its variables and it allows an easy overriding
of that built in “implicit” convention. Since most engineering and mathematical publications used the
letters from “i” through “n” as subscripts, summation ranges, loop counters, etc. Fortran first was released
with implicit variable typing such that all variables whose name begin with the letters “i” through “n”,
inclusive, defaulted to integers, unless declared otherwise. All other variables default to be real, unless
declared otherwise. In other words, you can think of the default code as if it contained the statements:

IMPLICIT INTEGER (I-N) ! F77 and F90 Default
IMPLICIT REAL (A-H, O-Z) ! F77 and F90 Default

The effect is automatic even if the statements are omitted. Explicit type declarations override any given
IMPLICIT types. For example, if the code had the above implicit defaults one could also explicitly
identify the exceptions to those default rules, such as the statements:

INTEGER :: Temp_row
<table>
<thead>
<tr>
<th>Storage</th>
<th>MATLAB</th>
<th>C++</th>
<th>F90</th>
<th>F77</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>char</td>
<td>character::</td>
<td>character</td>
<td></td>
</tr>
<tr>
<td>integer</td>
<td>int</td>
<td>integer::</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>single precision</td>
<td>float</td>
<td>real::</td>
<td>real</td>
<td></td>
</tr>
<tr>
<td>double precision</td>
<td>double</td>
<td>real*8::</td>
<td>double precision</td>
<td></td>
</tr>
<tr>
<td>complex</td>
<td>* complex::</td>
<td>complex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Boolean</td>
<td>bool</td>
<td>logical::</td>
<td>logical</td>
<td></td>
</tr>
<tr>
<td>argument</td>
<td>parameter::</td>
<td>parameter</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pointer</td>
<td>* pointer::</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>structure</td>
<td>struct type::</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*aMATLAB 4 requires no variable type declaration; the only two distinct types in MATLAB are strings and reals (which include complex). Booleans are just 0s and 1s treated as reals. MATLAB 5 allows the user to select more types.  
bThere is no specific data type for a complex variable in C++; they must be created by the programmer.*

Table 4.2: Intrinsic data types of variables

<table>
<thead>
<tr>
<th>Description</th>
<th>MATLAB</th>
<th>C++</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>addition</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>subtraction'</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>multiplication</td>
<td>* and .*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>division</td>
<td>/ and ./</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>exponentiation</td>
<td>&quot; and .&quot;</td>
<td>pow&quot;</td>
<td>**</td>
</tr>
<tr>
<td>remainder</td>
<td>%</td>
<td></td>
<td>%</td>
</tr>
<tr>
<td>increment</td>
<td>++</td>
<td></td>
<td>++</td>
</tr>
<tr>
<td>decrement</td>
<td>--</td>
<td></td>
<td>--</td>
</tr>
<tr>
<td>parentheses (expression grouping)</td>
<td>( )</td>
<td>( )</td>
<td>( )</td>
</tr>
</tbody>
</table>

*aWhen doing arithmetic operations on matrices in MATLAB, a period (‘.’) must be put before the operator if scalar arithmetic is desired. Otherwise, MATLAB assumes matrix operations; figure out the difference between ‘*’ and ‘.*’. Note that since matrix and scalar addition coincide, no ‘.+’ operator exists (same holds for subtraction).  
bFortran 90 allows the user to change operators and to define new operator symbols.  
cIn all languages the minus sign is used for negation (i.e., changing sign).  
dIn C++ the exponentiation is calculated by function pow (x, y).*

Table 4.3: Arithmetic operators

REAL :: Interest = 0.04 ! declare and initialize  
CHARACTER (Len=8) :: Months_of_year(12)

We will also see that the programmer can define new data types and explicitly declare their type as well.  
The F90 standard discourages the use of any IMPPLICIT variables such as  

IMPLICIT COMPLEX (X-Z) ! Complex variables  
IMPLICIT DOUBLE PRECISION (A-H) ! Double Precision reals

and encourages the use of  

IMPLICIT NONE

which forces the programmer to specifically declare the type of each and every variable used, and is referred to as strong typing. However, you need to know that such default variable types exist because they are used in many millions of lines of older Fortran code and at some point you will need to use or change such an existing program.
<table>
<thead>
<tr>
<th>Description</th>
<th>MATLAB</th>
<th>C++</th>
<th>F90</th>
<th>F77</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equal to</td>
<td>==</td>
<td>==</td>
<td>==</td>
<td>.EQ.</td>
</tr>
<tr>
<td>Not equal to</td>
<td>˜=</td>
<td>!=</td>
<td>/=</td>
<td>.NE.</td>
</tr>
<tr>
<td>Less than</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>.LT.</td>
</tr>
<tr>
<td>Less or equal</td>
<td>&lt;=</td>
<td>&lt;=</td>
<td>&lt;=</td>
<td>.LE.</td>
</tr>
<tr>
<td>Greater than</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>.GT.</td>
</tr>
<tr>
<td>Greater or equal</td>
<td>&gt;=</td>
<td>&gt;=</td>
<td>&gt;=</td>
<td>.GE.</td>
</tr>
<tr>
<td>Logical NOT</td>
<td>˜</td>
<td>!</td>
<td>.NOT.</td>
<td>.NOT.</td>
</tr>
<tr>
<td>Logical AND</td>
<td>&amp;</td>
<td>&amp;&amp;</td>
<td>.AND.</td>
<td>.AND.</td>
</tr>
<tr>
<td>Logical inclusive OR</td>
<td>! &amp;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical exclusive OR</td>
<td>xor</td>
<td>.XOR.</td>
<td>.XOR.</td>
<td>.XOR.</td>
</tr>
<tr>
<td>Logical equivalent</td>
<td>==</td>
<td>==</td>
<td>.EQV.</td>
<td>.EQV.</td>
</tr>
<tr>
<td>Logical not equivalent</td>
<td>˜=</td>
<td>!=</td>
<td>.NEQV.</td>
<td>.NEQV.</td>
</tr>
</tbody>
</table>

**Table 4.4**: Relational operators (arithmetic and logical)

<table>
<thead>
<tr>
<th>MATLAB Operators</th>
<th>C++ Operators</th>
<th>F90 Operators(^a)</th>
<th>F77 Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>()</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>+ -</td>
<td>! ++ -- +</td>
<td>**</td>
<td>**</td>
</tr>
<tr>
<td>* /</td>
<td>* / %</td>
<td>* /</td>
<td>* /</td>
</tr>
<tr>
<td>+ _b</td>
<td>+ _b</td>
<td>+ _b</td>
<td>+ _b</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>&lt;&lt; &gt;&gt;</td>
<td>//</td>
<td>//</td>
</tr>
<tr>
<td>== &quot;=</td>
<td>== /= &lt; &lt;= &gt;</td>
<td>.EQ. .NE.</td>
<td>.GT. .LE.</td>
</tr>
<tr>
<td>-</td>
<td>== !=</td>
<td>.NOT.</td>
<td>.NOT.</td>
</tr>
<tr>
<td>&amp;</td>
<td>&amp;&amp;</td>
<td>.AND.</td>
<td>.AND.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>= = / = =</td>
<td>.EQV. .NEQV.</td>
<td>.EQV. .NEQV.</td>
</tr>
<tr>
<td>? :</td>
<td>= = = = = = =</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(^a\)User-defined unary (binary) operators have the highest (lowest) precedence in F90.

\(^b\)These are binary operators representing addition and subtraction. Unary operators + and - have higher precedence.

**Table 4.5**: Precedence pecking order
program main
!
Examples of simple arithmetic in F90
implicit none
integer :: Integer_Var_1, Integer_Var_2 ! user inputs
integer :: Mult_Result, Div_Result, Add_Result
integer :: Sub_Result, Mod_Result
real :: Pow_Result, Sqrt_Result

print *, 'Enter two integers:'
read *, Integer_Var_1, Integer_Var_2

Add_Result = Integer_Var_1 + Integer_Var_2
print *, Integer_Var_1,' + ', Integer_Var_2,' = ', Add_Result

Sub_Result = Integer_Var_1 - Integer_Var_2
print *, Integer_Var_1,' - ', Integer_Var_2,' = ', Sub_Result

Mult_Result = Integer_Var_1 * Integer_Var_2
print *, Integer_Var_1,' * ', Integer_Var_2,' = ', Mult_Result

Div_Result = Integer_Var_1 / Integer_Var_2
print *, Integer_Var_1,' / ', Integer_Var_2,' = ', Div_Result

Mod_Result = mod (Integer_Var_1, Integer_Var_2) ! remainder
print *, Integer_Var_1,' mod ', Integer_Var_2,' = ', Mod_Result

Pow_Result = Integer_Var_1 ** Integer_Var_2 ! exponentiation
print *, Integer_Var_1,' ˆ ', Integer_Var_2,' = ', Pow_Result

Sqrt_Result = sqrt( real(Integer_Var_1))
print *, 'Square root of ', Integer_Var_1,' = ', Sqrt_Result
!
end program main ! Running produces:
!
! Enter two integers:
! 25 + 4 = 29
! 25 - 4 = 21
! 25 * 4 = 100
! 25 / 4 = 6, note integer
! 25 mod 4 = 1
! 25 - 4 = 3.9062500E+05
! Square root of 25 = 5.0000000

Figure 4.1: Typical Math and Functions in F90

An example program that employs the typical math operators in F90 is shown in Fig. 4.1. It presents examples of addition (line 11), subtraction (line 14), multiplication (line 17), division (line 20), as well as the use of the remainder or modulo function (line 23), exponentiation (line 26), and square root operators (line 29). In addition it shows a way of inputting data from the default input device (line 9). The results are appended as comments (lines 33-40). Observe that a program must include one and only one segment that begins with the word program (line 1) and ends with the line end program (line 32). If a name is assigned to the program then it must be appended to both of these lines. Often the name of main is used, as here, but it is not required as it is in C++. A C++ formulation of this example is included for comparison in the appendix as there are several other examples from this chapter.

A special expression available in MATLAB and F90 uses the colon operator (:) to indicate forming a vector (row matrix) of numbers according to an arithmetic progression. In MATLAB, the expression b:i:e means the vector [b (b + i) (b + 2i) · · · (b + Ni)], where (b + Ni) is the largest number less than or equal to (greater than or equal to if i is negative) the value of the variable e. Thus, b means “beginning value”, i means the increment, and e the end value. The expression b:e means that the increment equals one. You can use this construct to excise a portion of a vector or matrix. For example, x(2:5) equals the vector comprised by the second through fifth elements of x, and A(3:5,1:3) creates a matrix from the third, fourth, and fifth rows, the third, fourth, and fifth columns of the matrix A. F90 uses the convention of b:e:i and has the same defaults when :i is omitted. This operator, also known as the subscript triplet, is described in Table 4.6.

Of course, expressions often involve the use of functions. A tabulation of the built-in functions in our languages is given in Table 4.7 and the F90 overview, as are all the remaining tables of this chapter. The arguments of functions and subprograms have some important properties that vary with the language used. Primarily, we are interested in how actual arguments are passed to the dummy arguments in the subprogram. This data passing happens by either of two fundamentally different ways: by reference, or
Table 4.6: Colon Operator Syntax and its Applications.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>F90</th>
<th>MATLAB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>B: E: I</td>
<td>B: I: E</td>
</tr>
<tr>
<td>≥ B</td>
<td>B:</td>
<td>B:</td>
</tr>
<tr>
<td>≤ E</td>
<td>:E</td>
<td>:E</td>
</tr>
<tr>
<td>Full range</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Use</th>
<th>F90</th>
<th>MATLAB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array subscript ranges</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Character positions in a string</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Loop control</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Array element generation</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 4.7: Mathematical functions

<table>
<thead>
<tr>
<th>Description</th>
<th>MATLAB</th>
<th>C++</th>
<th>F90</th>
<th>F77</th>
</tr>
</thead>
<tbody>
<tr>
<td>exponential</td>
<td>exp(x)</td>
<td>exp(x)</td>
<td>exp(x)</td>
<td>exp(x)</td>
</tr>
<tr>
<td>natural log</td>
<td>log(x)</td>
<td>log(x)</td>
<td>log(x)</td>
<td>log(x)</td>
</tr>
<tr>
<td>base 10 log</td>
<td>log10(x)</td>
<td>log10(x)</td>
<td>log10(x)</td>
<td>log10(x)</td>
</tr>
<tr>
<td>square root</td>
<td>sqrt(x)</td>
<td>sqrt(x)</td>
<td>sqrt(x)</td>
<td>sqrt(x)</td>
</tr>
<tr>
<td>raise to power (x^r)</td>
<td>x.^r</td>
<td>pow(x,r)</td>
<td>x**r</td>
<td>x**r</td>
</tr>
<tr>
<td>absolute value</td>
<td>abs(x)</td>
<td>fabs(x)</td>
<td>abs(x)</td>
<td>abs(x)</td>
</tr>
<tr>
<td>smallest integer &gt; x</td>
<td>ceil(x)</td>
<td>ceil(x)</td>
<td>ceiling(x)</td>
<td></td>
</tr>
<tr>
<td>largest integer &lt; x</td>
<td>floor(x)</td>
<td>floor(x)</td>
<td>floor(x)</td>
<td></td>
</tr>
<tr>
<td>division remainder</td>
<td>rem(x,y)</td>
<td>fmod(x,y)</td>
<td>mod(x,y)</td>
<td></td>
</tr>
<tr>
<td>modulo</td>
<td>modulo(x,y)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>complex conjugate</td>
<td>conj(z)</td>
<td>conjg(z)</td>
<td>conjg(z)</td>
<td></td>
</tr>
<tr>
<td>imaginary part</td>
<td>imag(z)</td>
<td>imag(z)</td>
<td>aimag(z)</td>
<td></td>
</tr>
<tr>
<td>drop fraction</td>
<td>fix(x)</td>
<td>aint(x)</td>
<td>aint(x)</td>
<td></td>
</tr>
<tr>
<td>round number</td>
<td>round(x)</td>
<td>nint(x)</td>
<td>nint(x)</td>
<td></td>
</tr>
<tr>
<td>cosine</td>
<td>cos(x)</td>
<td>cos(x)</td>
<td>cos(x)</td>
<td>cos(x)</td>
</tr>
<tr>
<td>sine</td>
<td>sin(x)</td>
<td>sin(x)</td>
<td>sin(x)</td>
<td>sin(x)</td>
</tr>
<tr>
<td>tangent</td>
<td>tan(x)</td>
<td>tan(x)</td>
<td>tan(x)</td>
<td>tan(x)</td>
</tr>
<tr>
<td>arc cosine</td>
<td>acos(x)</td>
<td>acos(x)</td>
<td>acos(x)</td>
<td>acos(x)</td>
</tr>
<tr>
<td>arc sine</td>
<td>asin(x)</td>
<td>asin(x)</td>
<td>asin(x)</td>
<td>asin(x)</td>
</tr>
<tr>
<td>arc tangent</td>
<td>atan(x)</td>
<td>atan(x)</td>
<td>atan(x)</td>
<td>atan(x)</td>
</tr>
<tr>
<td>arc tangent²</td>
<td>atan2(x,y)</td>
<td>atan2(x,y)</td>
<td>atan2(x,y)</td>
<td>atan2(x,y)</td>
</tr>
<tr>
<td>hyperbolic cosine</td>
<td>cosh(x)</td>
<td>cosh(x)</td>
<td>cosh(x)</td>
<td>cosh(x)</td>
</tr>
<tr>
<td>hyperbolic sine</td>
<td>sinh(x)</td>
<td>sinh(x)</td>
<td>sinh(x)</td>
<td>sinh(x)</td>
</tr>
<tr>
<td>hyperbolic tangent</td>
<td>tanh(x)</td>
<td>tanh(x)</td>
<td>tanh(x)</td>
<td>tanh(x)</td>
</tr>
<tr>
<td>hyperbolic arc cosine</td>
<td>acosh(x)</td>
<td>acosh(x)</td>
<td>acosh(x)</td>
<td></td>
</tr>
<tr>
<td>hyperbolic arc sine</td>
<td>asinh(x)</td>
<td>asinh(x)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hyperbolic arc arctan</td>
<td>atan(x)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

²Differ for x < 0.

*a*atan2 (x, y) is used to calculate the arc tangent of x/y in the range [−π, +π]. The one-argument function atan(x) computes the arc tangent of x in the range [−π/2, +π/2].

by value. One should understand the difference between these two mechanisms.

“Passing by reference” means that the address in memory of the actual argument is passed to the subprogram instead of the value stored at that address. The corresponding dummy argument in the subprogram has the same address. That is, both arguments refer to the same memory location so any change to that argument within the subprogram is passed back to the calling code. A variable is passed by reference to a subroutine whenever it is expected that it should be changed by the subprogram. A related term is “dereferencing”. When you dereference a memory address, you are telling the computer to get the information located at the address. Typically, one indirectly gives the address by citing the

©2001 J.E. Akin 56
Conditionally execute statements
\[
\begin{array}{l}
\text{C++} \quad \text{F90} \quad \text{F77} \quad \text{MATLAB} \\
\text{if} \quad \text{if} \quad \text{if} \\
\quad \{ \} \quad \text{end if} \quad \text{end if} \quad \text{end if} \\
\end{array}
\]

Loop a specific number of times
\[
\begin{array}{l}
\text{for} \ k=1:n \quad \text{do} \ k=1:n \quad \text{do} \ # \ k=1:n \\
\quad \{ \} \quad \text{end do} \quad \text{end do} \quad \text{for} \ k=1:n \\
\end{array}
\]

Loop an indefinite number of times
\[
\begin{array}{l}
\text{while} \quad \text{do while} \quad - \quad \text{while} \\
\quad \{ \} \quad \text{end do} \quad - \quad \text{end} \\
\end{array}
\]

Terminate and exit loop
\[
\begin{array}{l}
\text{break} \quad \text{exit} \quad \text{go to} \quad \text{break} \\
\end{array}
\]

Skip a cycle of loop
\[
\begin{array}{l}
\text{continue} \quad \text{cycle} \quad \text{go to} \quad - \\
\end{array}
\]

Display message and abort
\[
\begin{array}{l}
\text{error()} \quad \text{stop} \quad \text{stop} \\
\end{array}
\]

Return to invoking function
\[
\begin{array}{l}
\text{return} \quad \text{return} \quad \text{return} \quad \text{return} \\
\end{array}
\]

Conditional array action
\[
\begin{array}{l}
- \quad \text{where} \quad - \quad \text{if} \\
\end{array}
\]

Conditional alternate statements
\[
\begin{array}{l}
\text{else} \quad \text{else} \quad \text{else} \quad \text{else} \\
\text{else if} \quad \text{elseif} \quad \text{elseif} \quad \text{elseif} \\
\end{array}
\]

Conditional array alternatives
\[
\begin{array}{l}
- \quad \text{elsewhere} \quad - \quad \text{else} \\
- \quad - \quad - \quad \text{elseif} \\
\end{array}
\]

Conditional case selections
\[
\begin{array}{l}
\text{switch} \{ \} \quad \text{select} \quad \text{if} \quad \text{if} \\
\quad \text{end select} \quad \text{end if} \quad \text{end} \\
\end{array}
\]

Table 4.8: Flow Control Statements.

name of a pointer variable or a reference variable.

“Passing by value” means that the value of the actual argument stored at its address in memory is copied and the copy is passed to the dummy argument in the subprogram. Thus any change to the argument within the subprogram is not passed back to the calling code. The two passing methods do not clearly show the intended use of the argument within the subprogram. Is it to be passed in for use only, passed in for changing and returned, or is it to be created in the subprogram and passed out for use in the calling code? For additional safety and clarity modern languages provide some way to allow the programmer to optionally specify such intent explicitly.

Both C++ and MATLAB use the pass by value method as their default mode. This means the value associated with the argument name, say arg\_name, is copied and passed to the function. That copying could be very inefficient if the argument is a huge array. To denote that you want to have the C++ argument passed by reference you must precede the argument name with an ampersand (&), e.g. &arg\_name, in the calling code. Then within the subprogram the corresponding dummy variable must be dereferenced by preceding the name with an asterisk (*), e.g. *arg\_name. Conversely, Fortran uses the passing by reference method as its default mode. On the rare occasions when one wants to pass by value simply surround the argument name with parentheses, e.g. (arg\_name), in the calling code. In either case it is recommended that you cite each argument with the optional “intent” statement within the subprogram. Examples of the two passing options are covered in Sec. 4.5.

4.3 Flow Control

The basic flow control constructs present in our selected engineering languages are loops—repetitive execution of a block of statements—and conditionals—diversions around blocks of statements. A typical set of flow control statement types are summarized in Table 4.8. Most of these will be illustrated in detail in the following sections.
4.3.1 Explicit Loops

The following discussion will introduce the important concept of loops. These are required in most programs. However, the reader is warned that today the writing of explicit loops are generally not the most efficient way to execute a loop operation in Fortran90 and MATLAB. Of course, older languages like F77 and C do require them, so that the time spent here not only covers the explicit loop concepts but aids one in reading older languages. Our pseudocode for the common loops is:
In engineering programming one often needs to repeatedly perform a group of operations. Most computer languages have a statement to execute this powerful and widely-used feature. In Fortran this is the \texttt{DO} statement, while in C++ and MATLAB it is the \texttt{FOR} statement. This one statement provides for the initialization, incrementing and testing of the loop variable, plus repeated execution of a group of statements contained within the loop. In Fortran77, the loop always cites a label number that indicates the extent of the statements enclosed in the loop. This is allowed in F90, but not recommended, and is considered obsolete. Instead, the \texttt{END DO} indicates the extent of the loop, and the number label is omitted in both places. F90 does allow one to give a name to a loop. Then the structure is denoted as \texttt{NAME:DO} followed by \texttt{END DO NAME}. Examples of the syntax for these statements for the languages of interest are given in Table 4.9.

A simple example of combining loops and array indexing is illustrated in Figs. 4.2 and 4.3. Note in Fig. 4.2 that the final value of a loop counter (called \texttt{Integer Var} here) upon exiting the loop (line 10) can be language or compiler dependent despite the fact that they are same here. In Fig. 4.3, we introduce for the first time a variable with a single subscript (line 5) and containing five numbers (integers) to be manually initialized (lines 8-10) and then to be listed in a loop (lines 12-15) over all their values. Note that C++ stores the first entry in an array at position zero (see appendix listing), MATLAB uses position one, and F90 defaults to position one.

C++ and Fortran 90 allow a special option to create loops that run “forever.” These could be used, for example, to read an unknown amount of data until terminated, in a non-fatal way, by the input statement. In C++, one omits the three loop controls, such as

\begin{verbatim}
for (;;) { // forever loop
  loop_block
} // end forever loop
\end{verbatim}

while in F90, one simply omits the loop control and gives only the DO command:

\begin{verbatim}
do ! forever
\end{verbatim}
Most of the time, an infinite loop is used as a \texttt{loop\_while\_true} or a \texttt{loop\_until\_true} construct. These will be considered shortly.

### 4.3.2 Implied Loops

Fortran and MATLAB have shorthand methods for constructing “implied loops.” Both languages offer the colon operator to imply an incremental range of integer values. Its syntax and types of applications are given in Table 4.6 (page 56). The allowed usages of the operator differ slightly between the two languages. Note that this means that the loop controls are slightly different in that the \texttt{do} control employs commas instead of colons. For example, two equivalent loops are

<table>
<thead>
<tr>
<th>Fortran</th>
<th>MATLAB</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{do k=B,E,I} \newline \texttt{A(k) = k**2} \newline \texttt{end do}</td>
<td>\texttt{for k=B:I:E} \newline \texttt{A(k) = k^2} \newline \texttt{end}</td>
</tr>
</tbody>
</table>

Fortran offers an additional formal implied \texttt{do} loop that replaces the \texttt{do} and \texttt{end do} with a closed pair of parentheses in the syntax:

\[
\texttt{(object, k = B,E,I)}
\]
where again the increment, I, defaults to unity if not supplied. The above implied do is equivalent to the formal loop

\[
\text{do } k = B, E, I \\
\text{define object} \\
\text{end do}
\]

However, the object defined in the implied loop can only be utilized for four specific Fortran operations: 1) read actions, 2) print and write actions, 3) data variables (not value) definitions, and 4) defining array elements. For example,

\[
\text{print }*, (4*k-1, k=1,10,3) ! 3, 15, 27, 39 \\
\text{read }*, (A(j,:), j=1,\text{rows}) ! \text{read A by rows, sequentially}
\]

The implied do loops can be nested to any level like the standard do statement. One simply makes the inner loop the object of the outer loop, so that

\[
((\text{object}_{j,k}, j=\text{min, max}), k=k1,k2,\text{inc})
\]

implies the nested loop

\[
\text{do } k = k1,k2,\text{inc} \\
\text{do } j = \text{min, max} \\
\text{use object}_{j,k} \\
\text{end do} ! \text{over j} \\
\text{end do} ! \text{over k}
\]

For example,

\[
\text{print }*, (((A(k)*B(j)+3), j=1,5), k=1,\text{max}) \\
\text{read }*, (((A(i,j,k) , j=1,\text{cols}), i=1,\text{rows}), k=1,\text{max})
\]

Actually, there is even a simpler default form of implied dos for reading and writing arrays. That default is to access arrays by columns. That is, process the leftmost subscript first. Thus, for an array with three subscripts,

\[
\text{read }*, A \Leftarrow \text{read }*, (((A(i,j,k), j=1,\text{cols}), i=1,\text{rows}), k=1,\text{planes})
\]

Both languages allow the implied loops to be employed to create an array vector simply by placing the implied loop inside the standard array delimit symbols. For example, we may want an array to equally distribute \( N + 1 \) points over the distance from zero to \( D \).

\[
\text{F90: } X = (/ (k,k=0,N)/) * D / (N+1) \\
\text{MATLAB: } X = [0:N] * D / (N+1),
\]

which illustrates that MATLAB allows the use of the colon operator to define arrays, but F90 does not.

In addition to locating elements in an array by the regular incrementing of loop variables, both Fortran90 and MATLAB support even more specific selections of elements: by random location via vector subscripts, or by value via logical masks such as where and if in F90 and MATLAB, respectively.

### 4.3.3 Conditionals

Logic tests are frequently needed to control the execution of a block of statements. The most basic operation occurs when we want to do something when a logic test gives a true answer. We call that a simple IF statement. When the test is true, the program executes the block of statements following the IF. Often only one statement is needed, so C++ and Fortran allow that one statement to end the line that begins with the IF logic. Frequently we will nest another IF within the statements from a higher level IF. The common language syntax forms for the simple IF are given below in Table 4.10, along with the examples of where a second true group is nested inside the first as shown in Table 4.11.

The next simplest case is where we need to do one thing when the answer is true, and a different thing when the logic test is false. Then the syntax changes simply to an IF {true group} ELSE {false group} mode of execution. The typical IF-ELSE syntaxes of the various languages are given in Table 4.12. Of course, the above statement groups can contain other IF or IF-ELSE statements nested within them. They can also contain any valid statements, including DO or FOR loops.

The most complicated logic tests occur when the number of cases for the answer go beyond the two (true-false) of the IF-ELSE control structure. These multiple case decisions can be handled with the IF-ELSEIF-ELSE control structures whose syntax is given in Table 4.13. They involve a sequence of logic
Table 4.10: IF Constructs. The quantity \( l_{\text{expression}} \) means a logical expression having a value that is either TRUE or FALSE. The term true statement or true group means that the statement or group of statements, respectively, are executed if the conditional in the if statement evaluates to TRUE.

Table 4.11: Nested IF Constructs.

Table 4.12: Logical IF-ELSE Constructs.

tests, each of which is followed by a group of statements that are to be executed if, and only if, the test answer is true. There can be any number of such tests. They are terminated with an ELSE group of default statements to be executed if none of the logic tests are true. Actually, the ELSE action is optional. For program clarity or debugging, it should be included even if it only prints a warning message or contains a comment statement. Typical “if” and “if-else” coding is given in Figs. 4.4, 4.5, and 4.6. Figure 4.4 simply uses the three logical comparisons of “greater than” (line 9), “less than” (line 12), or “equal to” (line 15), respectively. Figure 4.5 goes a step further by combining two tests with a logical “and” test (line 9), and includes a second else branch (line 11) to handle the case where the if is false. While the input to these programs were numbers (line 7), the third example program in Fig. 4.6 accepts logical input (lines 6,8) that represents either true or false values and carries out Boolean operations to negate an input (via NOT in line 9), or to compare two inputs (with an AND in line 11, or OR in line 17, etc.) to produce a third logical value.

Since following the logic of many IF-ELSEIF-ELSE statements can be very confusing both the C++ and Fortran languages allow a CASE selection or “switching” operation based on the value (numerical or character) of some expression. For any allowed specified CASE value, a group of statements is executed. If the value does not match any of the specified allowed CASE values, then a default group of statements are executed. These are illustrated in Table 4.14.
Fortran90 offers an additional optional feature called construct names that can be employed with the above IF and SELECT CASE constructs to improve the readability of the program. The optional name, followed by a colon, precedes the key words IF and SELECT CASE. To be consistent, the name should also follow the key words END IF or END SELECT which always close the constructs. The construct name option also is available for loops where it offers an additional pair of control actions that will be explained later. Examples of these optional F90 features are given in Table 4.15.

While C++ and MATLAB do not formally offer this option, the same enhancement of readability can
program main
!
Example of Logical operators in F90
implicit none

logical :: Logic_Var_1, Logic_Var_2
print *, 'Print logical value of A (T or F):'ead *, Logic_Var_1
print *, 'Print logical value of B (T or F):'ead *, Logic_Var_2
print *, 'NOT A is ', (.NOT. Logic_Var_1)
if ( Logic_Var_1 .AND. Logic_Var_2 ) then
  print *, 'A ANDed with B is true'
elsa
  print *, 'A ANDed with B is false'
end if ! for AND
if ( Logic_Var_1 .OR. Logic_Var_2 ) then
  print *, 'A ORed with B is true'
elsa
  print *, 'A ORed with B is false'
end if ! for OR
if ( Logic_Var_1 .EQV. Logic_Var_2 ) then
  print *, 'A EQUivalent with B is true'
elsa
  print *, 'A EQUivalent with B is false'
end if ! for EQV
if ( Logic_Var_1 .NEQV. Logic_Var_2 ) then
  print *, 'A Not EQUivalent with B is true'
elsa
  print *, 'A Not EQUivalent with B is false'
end if ! for NEQV
end program main
!
Running with T and F produces:
! Print logical value of A (T or F): T
! Print logical value of B (T or F): F
! NOT A is F
! A ANDed with B is false
! A ORed with B is true
! A EQUivalent with B is false
! A Not EQUivalent with B is true

Figure 4.6: Typical Logical Operators in F90

<table>
<thead>
<tr>
<th>F90</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT CASE (expression)</td>
<td>switch (expression) {</td>
</tr>
</tbody>
</table>
| CASE (value 1)            |     case value 1 :
| group 1                   |         group 1 |
| CASE (value 2)            |         break;
| group 2                   |     case value 2 :
| ;                         |         group 2 |
| CASE (value n)            |         break;
| group n                   |     :       |
| CASE DEFAULT              |     case value n :
| default group             |         group n |
| END SELECT                |         break;
|                           |     default:   |
|                           |         default group |
|                           |         break;    |

Table 4.14: Case Selection Constructs.

be achieved by using the trailing comment feature to append a name or description at the beginning and end of these logic construct blocks.

Both C++ and Fortran allow statement labels and provide controls to branch to specific labels. Today you are generally advised not to use a GO TO and its associated label! However, they are common in many F77 codes. There are a few cases where a GO TO is still considered acceptable. For example, the pseudo-while construct of F77 requires a GO TO.
<table>
<thead>
<tr>
<th>F90 Named IF</th>
<th>F90 Named SELECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: IF (logical_1) THEN</td>
<td>name: SELECT CASE (expression)</td>
</tr>
<tr>
<td>true group A</td>
<td>CASE (value 1)</td>
</tr>
<tr>
<td>ELSE IF (logical_2) THEN</td>
<td>group 1</td>
</tr>
<tr>
<td>true group B</td>
<td>CASE (value 2)</td>
</tr>
<tr>
<td>ELSE</td>
<td>group 2</td>
</tr>
<tr>
<td>default group C</td>
<td>CASE DEFAULT</td>
</tr>
<tr>
<td>ENDIF name</td>
<td>default group</td>
</tr>
<tr>
<td></td>
<td>END SELECT name</td>
</tr>
</tbody>
</table>

Table 4.15: F90 Optional Logic Block Names.

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>DO 1 ...</td>
<td>for (...) {</td>
</tr>
<tr>
<td>DO 2 ...</td>
<td>for (...) {</td>
</tr>
<tr>
<td>IF (disaster) THEN</td>
<td>...</td>
</tr>
<tr>
<td>GO TO 3</td>
<td>if (disaster)</td>
</tr>
<tr>
<td>END IF</td>
<td>go to error</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2 END DO</td>
<td>}</td>
</tr>
<tr>
<td>1 END DO</td>
<td>error:</td>
</tr>
<tr>
<td>3 next statement</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.16: GO TO Break-out of Nested Loops. This situation can be an exception to the general recommendation to avoid GO TO statements.

<table>
<thead>
<tr>
<th>F77</th>
<th>F90</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>DO 1 I = 1,N</td>
<td>DO I = 1,N</td>
<td>for (i=1; i&lt;n; i++) {</td>
</tr>
<tr>
<td>...  IF (skip condition) THEN</td>
<td>... IF (skip condition) THEN</td>
<td>}</td>
</tr>
<tr>
<td>GO TO 1</td>
<td>CYCLE ! to next I</td>
<td>if (skip condition)</td>
</tr>
<tr>
<td>ELSE</td>
<td>ELSE</td>
<td>continue; // to next</td>
</tr>
<tr>
<td>false group</td>
<td>false group</td>
<td>else if</td>
</tr>
<tr>
<td>END IF</td>
<td>END IF</td>
<td>false group</td>
</tr>
<tr>
<td>1 continue</td>
<td>END DO</td>
<td>end</td>
</tr>
</tbody>
</table>

Table 4.17: Skip a Single Loop Cycle.

The GO TO can also be effectively utilized in both Fortran and C++ to break out of several nested loops. This is illustrated in Table 4.16. The “break-out” construct can be used in the situation when, as a part of a subroutine, you wanted the program exit the loop and also exit the subroutine, returning control to the calling program. To do that, one would simply replace the GO TO statement with the RETURN statement. In F90, one should also append the comment “!” to calling program” to assist in making the subroutine more readable.

You may find it necessary to want to skip a cycle in loop execution and/or exit from a single loop. Both Fortran and C++ provide these control options without requiring the use of a GO TO. To skip a loop cycle, Fortran90 and C++ use the statements CYCLE and continue, respectively, and EXIT and break to abort a loop. These constructs are shown in Tables 4.17 and 4.18. Other forms of the GO TO in F77 were declared obsolete in F90, and should not be used. The Fortran abort examples could also use the RETURN option described above in the rare cases when it proves to be more desirable or efficient.

As mentioned earlier, F90 allows the programmer to use “named” DO constructs. In addition to im-
proving readability, this feature also offers additional control over nested loops because we can associate the CYCLE and EXIT commands with a specific loop (Table 4.19). Without the optional name, the CYCLE and EXIT commands act only on the inner-most loop in which they lie. We will see later that Fortran90 allows another type of loop called WHERE that is designed to operate on arrays.

4.3.3.1 Looping While True or Until True

It is very common to need to perform a loop so long as a condition is true, or to run the loop until a condition becomes true. The two are very similar and both represent loops that would run forever unless specifically terminated. We will refer to these two approaches as WHILE loops and UNTIL loops. The WHILE logic test is made first in order to determine if the loop will be entered. Clearly, this means that if the logic test is false the first time it is tested, then the statement blocks controlled by the WHILE are never executed. If the WHILE loop is entered, something in the loop must eventually change the value of a variable in the logic test or the loop would run forever. Once a change causes the WHILE logic test to be false control is transferred to the first statement following the WHILE structure. By way of comparison, an UNTIL loop is always entered at least once. Upon entering the loop, a beginning statement group is executed. Then the logic test is evaluated. If the test result is true, the loop is exited and control is passed to the next statement after the group. If the test is false, then an optional second statement group is executed before the loop returns to the beginning statement group. The pseudo-code for these two similar structures are given as follows:

<table>
<thead>
<tr>
<th>while true</th>
<th>until true</th>
</tr>
</thead>
<tbody>
<tr>
<td>logic_variable = true</td>
<td>logic_variable = false</td>
</tr>
<tr>
<td>begin:</td>
<td>begin:</td>
</tr>
<tr>
<td>if (logic_variable) then</td>
<td>statements</td>
</tr>
<tr>
<td>% true</td>
<td>if (logic_variable) then</td>
</tr>
<tr>
<td>true_group</td>
<td>exit the loop</td>
</tr>
<tr>
<td>re-evaluate logic_variable</td>
<td>else % false</td>
</tr>
<tr>
<td>go to begin</td>
<td>false_group</td>
</tr>
<tr>
<td>else % false</td>
<td>re-evaluate logic_variable</td>
</tr>
<tr>
<td>exit loop</td>
<td>go to begin</td>
</tr>
<tr>
<td>end if</td>
<td>end if</td>
</tr>
</tbody>
</table>

Table 4.18: Abort a Single Loop.

Table 4.19: F90 DOs Named for Control.
Since these constructs are commonly needed, several programming languages offer some support for them. For example, Pascal has a \texttt{REPEAT UNTIL} command and C++ has the \texttt{DO-WHILE} pair for the until-true construct. For the more common while-true loops, C++ and \textsc{Matlab} offer a \texttt{WHILE} command, and Fortran 90 includes the \texttt{DO WHILE}. F77, however, only has the obsolete \texttt{IF-GO TO} pairs as illustrated in a previous example. Many current programmers consider the \texttt{WHILE} construct obsolete because it is less clear than a \texttt{DO-EXIT} pair or a “for-break” pair. Indeed, the F90 standard has declared the \texttt{DO WHILE} as obsolete and eligible for future deletion from the language. We can see how the loop-abort feature of C++ and F90 includes both the \texttt{WHILE} and \texttt{UNTIL} concepts. For example, the F90 construct

\begin{verbatim}
initialize logical_variable
DO WHILE (logical_variable) ! is true
  true_group
  re-evaluate logical_variable
END DO ! while true
::
\end{verbatim}

is entirely equivalent to the aborted endless loop

\begin{verbatim}
initialize logical_variable
DO ! forever while true
  IF (.NOT. logical_variable) EXIT ! as false
  true_group
  re-evaluate logical_variable
END DO ! while true
::
\end{verbatim}

Likewise, a minor change includes the \texttt{UNTIL} construct.

\begin{verbatim}
DO ! forever until true
beginning statements and initialization
  IF (logical_expression) EXIT ! as true
  false_group
  re-evaluate logical_variable
END DO ! until true
\end{verbatim}

When approached in the C++ language, we have the \texttt{WHILE} loop.

\begin{verbatim}
initialize logical_variable
while (logical_variable)
  // is true
  true_group
  re-evaluate logical_variable
} // end while true
\end{verbatim}

Recalling the standard for syntax,

\begin{verbatim}
for (expr_1; expr_2; expr_3)
{
  true_group
} // end for
\end{verbatim}

could be viewed as equivalent to the above \texttt{WHILE} in \texttt{for} form.

\begin{verbatim}
expr_1;
while (expr_2)
  { // is true
    true_group
    expr_3;
} // end while true
\end{verbatim}

If one omits all three \texttt{for} expressions, then it becomes an “infinite loop” or a “do forever” which can represent a \texttt{WHILE} or \texttt{UNTIL} construct by proper placement of the \texttt{break} command. Furthermore, C has the \texttt{do-while} construct that is equivalent to Pascal’s \texttt{REPEAT-UNTIL}.

\begin{verbatim}
do // forever until true
  statements
  evaluate logical_variable
  while (logical_variable) // is true
\end{verbatim}

The syntax for the classical \texttt{WHILE} statements in C++, Fortran and \textsc{Matlab} are given in Table 4.20. Fortran90 has declared the \texttt{DO WHILE} as obsolete, and recommends the \texttt{DO-EXIT} pair instead! Using infinite loops with clearly aborted stages is a less error-prone approach to programming.
MATLAB | C++
---|---
initialize test | initialize test
while l_expression | while (l_expression)
true group | {
change test | true group
end | change test

| F77 | F90 |
---|---|
initialize test | initialize test
do while (l_expression) | 
true group | true group
change test | change test
go to # | end do

Table 4.20: Looping While a Condition is True.

<table>
<thead>
<tr>
<th>Function Type</th>
<th>MATLAB</th>
<th>C++</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>program</td>
<td>statements</td>
<td>main(argc,char **argv)</td>
<td>program main</td>
</tr>
<tr>
<td></td>
<td>[y_{1\ldots n}]=f(a_{1\ldots m})</td>
<td></td>
<td>type y</td>
</tr>
<tr>
<td></td>
<td>[end of file]</td>
<td></td>
<td>type a_{1\ldots m}, type am</td>
</tr>
<tr>
<td></td>
<td>statements</td>
<td>y = f(a_{1,I,am});</td>
<td>statements</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
<td>y = f(a_{1\ldots m});</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
<td>call s(a_{1\ldots am})</td>
</tr>
<tr>
<td>subroutine</td>
<td>void f</td>
<td>subroutine s(a_{1\ldots am})</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(type a_{1\ldots am})</td>
<td>type a_{1\ldots am}</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
<td>statements</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
<td>end</td>
</tr>
<tr>
<td>function</td>
<td>function [r_{1\ldots n}]=f(a_{1\ldots m})</td>
<td>type f (type a_{1\ldots am})</td>
<td>function f(a_{1\ldots am})</td>
</tr>
<tr>
<td></td>
<td>statements</td>
<td>{ statements }</td>
<td>type f</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>type a_{1\ldots am}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>statements</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>end</td>
</tr>
</tbody>
</table>

*Every function or program in MATLAB must be in separate files.*

Table 4.21: Function definitions. In each case, the function being defined is named \( f \) and is called with \( m \) arguments \( a_{1\ldots am} \).

### 4.4 Subprograms

The concept of modular programming requires the use of numerous subprograms or procedures to execute independent segments of the calculations or operations. Typically, these procedures fall into classes such as functions, subroutines, and modules. We will consider examples of the procedures for each of our target languages. These are shown in Table 4.21.

Recall that Table 8.6 compared several intrinsic functions that are common to both F90 and MATLAB. For completeness, all of the Fortran90 functions are listed both alphabetically and by subject in Appendix B. Similar listings for MATLAB can be found in the MATLAB Primer.

#### 4.4.1 Functions and Subroutines

Historically, a function was a subprogram that employed one or more input arguments and returned a single result value. For example, a square root or logarithm function would accept a single input value and return a single result. All of the languages of interest allow the user to define such a function, and they
One-Input, One-Result Procedures

<table>
<thead>
<tr>
<th>Language</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATLAB</td>
<td><code>function out = name (in)</code></td>
</tr>
<tr>
<td>F90</td>
<td><code>function name (in) ! name = out</code></td>
</tr>
<tr>
<td>C++</td>
<td><code>function name (in) result (out)</code></td>
</tr>
</tbody>
</table>

Multiple-Input, Multiple-Result Procedures

<table>
<thead>
<tr>
<th>Language</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATLAB</td>
<td><code>function [inout, out2] = name (in1, in2, inout)</code></td>
</tr>
<tr>
<td>F90</td>
<td><code>subroutine name (in1, in2, inout, out2)</code></td>
</tr>
<tr>
<td>C++</td>
<td><code>name(in1, in2, inout, out2)</code></td>
</tr>
</tbody>
</table>

* Other arrangements acceptable

Table 4.22: Arguments and return values of subprograms.

all provide numerous intrinsic or built-in functions of this type. As you might expect, such a procedure is called a function in C++, Fortran and MATLAB. As an example of such a procedure, consider the calculation of the mean value of a sequence of numbers defined as

$$\text{mean} = \frac{1}{n} \sum_{k=1}^{n} x_k.$$ 

In Fortran90, a subprogram to return the mean (average) could be

```fortran
function mean(x)
    real :: mean, x(:)
    mean = sum(x)/size(x)
end function mean
```

Note that our function has employed two other intrinsic functions: `size` to determine the number of elements in the array `x`, and `sum` to carry out the summation of all elements in `x`. Originally in Fortran, the result value was required to be assigned to the name of the function. That is still a valid option in F90, but today it is considered better practice to specify a result value name to be returned by the function. The `mean` function is a MATLAB intrinsic and can be used directly.

To apply these two functions to an array, say `y`, we would simply write `y_ave = mean(y)`, and `y_mid = mid_value(y)`, respectively. While Fortran allows a “function” to return only a single object, both C++ and MATLAB use that subprogram name to return any number of result objects. Fortran employs the name “subroutine” for such a procedure. Such procedures are allowed to have multiple inputs and multiple outputs (including none). The syntax of the first line of these two subprogram classes are shown in Table 4.22. Note that a typical subprogram may have no arguments, multiple input arguments (`in1, in2, inout`), multiple result arguments (`inout, out2`), and arguments that are used for both input and result usage (`inout`). These example names have been selected to reflect the fact that a programmer usually intends for arguments to be used for input only, or for result values only, or for input, modification, and output. It is considered good programming practice to declare such intentions to aid the compiler in detecting unintended uses. F90 provides the `INTENT` statement for this purpose, but does not require its use.

Having outlined the concepts of subprograms, we will review some presented earlier and then give some new examples. Figure 1.3 presented a clipping function which was earlier expressed in pseudocode. A corresponding Fortran implementation of such a clipping function is given in Fig. 4.7. Note that it is very similar to the pseudocode version.

©2001 J.E. Akin

69
program main
!

clip the elements of an array

implicit none

real, parameter :: limit = 3
integer, parameter :: n = 5
real :: y(n), x(n)
!

Define x values that will be clipped

x = (/ (-8. + 3.*k, k = 1,n) /) ! an implied loop
!
doi=1 ,n
!
y(i) = clip (x(i), limit)
!
end do
!
print *, x
!
print *, y
!
contains ! methods
!
function clip (x, L) result (c)
!
clip (x, L) - clip the variable x, output
!
x = scalar variable, input
L = limit of the clipper, input
!
real, intent(in) :: x, L ! variable types
real :: c ! variable types
intent (in) x, L ! argument uses
!
if ( abs(x) <= L ) then ! abs of x less than or equal L
!
c = x; ! then use x
!
else ! absolute of x greater than L ?
!
c = sign(L,x) ! sign of x times L
!
end if ! of value of x
!
end function ! clip
!
end program main
!
!
contains ! methods
!
function clip (x, L) result (c)
!
clip (x, L) - clip the variable x, output
!
x = scalar variable, input
L = limit of the clipper, input
!
real, intent(in) :: x, L ! variable types
real :: c ! variable types
!
intent (in) x, L ! argument uses
!
if ( abs(x) <= L ) then ! abs of x less than or equal L
!
c = x; ! then use x
!
else ! absolute of x greater than L ?
!
c = sign(L,x) ! sign of x times L
!
end if ! of value of x
!
end function ! clip
!
end program main
!
!
produces:

-5.00000000 -2.00000000 1.00000000 4.00000000 7.00000000

Figure 4.7: Clipping a Set of Array Values in F90

For the purpose of illustration an alternate F90 version of the Game of Life, shown earlier in Chapter 1 as pseudocode, is given in the assignment solutions section. Clearly we have not introduced all the features utilized in these example codes so the reader should continue to refer back to them as your programming understanding grows.

A simple program that illustrates program composition is maximum.f90, which asks the user to specify several integers from which the program finds the largest. It is given in Fig. 4.8. Note how the main program accepts the user input (lines 15,20), with the maxint function (line 22) finding the maximum (lines 25-34). Perhaps modularity would have been better served by expressing the input portion by a separate function. Of course, this routine is not really needed since F90 provides intrinsic functions to find maximum and minimum values (maxval, minval) and their locations in any array (maxloc, minloc). A similar C++ program composition is shown for comparison in the appendix.
program maximum ! of a set of integers (see intrinsic maxval)
implicit none
interface ! declare function interface prototype
function maxint (input, input_length) result(max)
  integer, intent(in) :: input_length, input(:)
  integer :: max
end function maxint
end interface

integer, parameter :: ARRAYLENGTH=100
integer :: integers(ARRAYLENGTH);
integer :: i, n;
!
read in the number of integers
print *, 'Find maximum; type n: '; read *, n
if ( n > ARRAYLENGTH .or. n < 0 ) &
  stop 'Value you typed is too large or negative.'
do i = 1, n ! Read in the user's integers
  do i = 1, input_length ! note could be only 1
    print *, 'Integer ', i, '?'; read *, integers(i)
  end do ! over n values
print *, 'Maximum: ', maxint (integers, n)
end program maximum

function maxint (input, input_length) result(max)
!
Find the maximum of an array of integers
integer, intent(in) :: input_length, input(:)
integer :: i, max
max = input(1); ! initialize
do i = 1, input_length ! note could be only 1
  if ( input(i) > max ) max = input(i);
end do ! over values
end function maxint
!
Figure 4.8: Search for Largest Value in F90
### Global Variable Declaration

<table>
<thead>
<tr>
<th>MATLAB</th>
<th>global list of variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>F77</td>
<td>common /set_name/ list of variables</td>
</tr>
<tr>
<td>F90</td>
<td>module set_name</td>
</tr>
<tr>
<td></td>
<td>save type (type_tag) :: list of variables</td>
</tr>
<tr>
<td></td>
<td>end module set_name</td>
</tr>
<tr>
<td>C++</td>
<td>extern list of variables</td>
</tr>
</tbody>
</table>

### Access to Global Variables

<table>
<thead>
<tr>
<th>MATLAB</th>
<th>global list of variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>F77</td>
<td>common /set_name/ list of variables</td>
</tr>
<tr>
<td>F90</td>
<td>use set_name, only subset of variables</td>
</tr>
<tr>
<td></td>
<td>use set_name2 list of variables</td>
</tr>
<tr>
<td>C++</td>
<td>extern list of variables</td>
</tr>
</tbody>
</table>

Table 4.23: Defining and referring to global variables.

### 4.4.2 Global Variables

We have seen that variables used inside a procedure can be thought of as dummy variable names that exist only in the procedure, unless they are members of the argument list. Even if they are arguments to the procedure, they can still have names different from the names employed in the calling program. This approach can have disadvantages. For example, it might lead to a long list of arguments, say 20 lines, in a complicated procedure. For this and other reasons, we sometimes desire to have variables that are accessible by any and all procedures at any time. These are called global variables regardless of their type.

Generally, we explicitly declare them to be global and provide some means by which they can be accessed, and thus modified, by selected procedures. When a selected procedure needs, or benefits from, access to a global variable, one may wish to control which subset of global variables are accessible by the procedure. The typical initial identification of global variables and the ways to access them are shown in Table 4.23, respectively.

An advanced aspect of the concept of global variables are the topics of inheritance and object-oriented programming. Fortran90, and other languages like C++, offer these advanced concepts. In F90, inheritance is available to a module and/or a main program and their “internal sub-programs” defined as those procedures following a contains statement, but occurring before an end module or the end program statement. Everything that appears before the contains statement is available to, and can be changed by, the internal sub-programs. Those inherited variables are more than local in nature, but not quite global; thus, they may be thought of as territorial variables. The structure of these internal sub-programs with inheritance is shown in Fig. 4.9

Perhaps the most commonly used global variables are those necessary to calculate the amount of central processor unit (cpu) time, in seconds, that a particular code segment used during its execution. All systems provide utilities for that purpose but some are more friendly than others. MATLAB provides a pair of functions, called tic and toc, that act together to provide the desired information. To illustrate the use of global variables we will develop a F90 module called tic_toc to hold the necessary variables along with the routines tic and toc. It is illustrated in Fig. 4.10 where the module constants (lines 2-6) are set (lines 17, 26) and computed (line 28) in the two internal functions.
module or program name_inherit
    Optional territorial variable, type specification, and calls
    contains

    subroutine Internal_1
    territorial specifications and calls
    contains

        subroutine Internal_2
        local computations
        end subroutine Internal_2

        subroutine Internal_3
        local computations
        end subroutine Internal_3

    end subroutine Internal_1

end name_inherit

Figure 4.9: F90 Internal Subprogram Structure.

module tic_toc
! Define global constants for timing increments
implicit none
integer :: start ! current value of system clock
integer :: rate ! system clock counts/sec
integer :: finish ! ending value of system clock
real :: sec ! increment in sec, (finish-start)/rate
! Usage: use tic_toc ! global constant access
! call tic ! start clock
! . . . ! use some cpu time
! cputime = toc () ! for increment
contains ! access to start, rate, finish, sec
subroutine tic
! -------------------------------------------------
! Model the matlab tic function, for use with toc
! ------------------------------------------------- 
implicit none
call system_clock ( start, rate ) ! Get start value and rate
end subroutine tic

function toc ( ) result(sec)
! -------------------------------------------------
! Model the matlab toc function, for use with tic
! -------------------------------------------------
implicit none
real :: sec
call system_clock ( finish ) ! Stop the execution timer
sec = 0.0
if ( finish >= start ) sec = float(finish - start) / float(rate)
end function toc
end module tic_toc

Figure 4.10: A Module for Computing CPU Times
### Bit Functions

We have discussed the fact that the digital computer is based on the use of individual bits. The subject of bit manipulation is one that we do not wish to pursue here. However, advanced applications do sometimes require these abilities, and the most common uses have been declared in the so-called military standards USDOD-MIL-STD-1753, and made part of the Fortran90 standard. Several of these features are also a part of C++. Table 4.24 gives a list of those functions.

### Exception Controls

An exception handler is a block of code that is invoked to process specific error conditions. Standard exception control keywords in a language are usually associated with the allocation of resources, such as files or memory space, or input/output operations. For many applications we simply want to catch an unexpected result and output a message so that the programmer can correct the situation. In that case we may not care if the exception aborts the execution. However, if one is using a commercial execute only program then it is very disturbing to have a code abort. We would at least expect the code to respond to a fatal error by closing down the program in some gentle fashion that saves what was completed before the error and maybe even offer us a restart option. Here we provide only the minimum form of an exceptions module that can be used by other modules to pass warnings of fatal messages to the user. It includes an integer flag that can be utilized to rank the severity of possible messages. It is shown in Fig. 4.11. Below we will summarize the F90 optional error flags that should always be checked and are likely to lead to a call to the exception handler.

#### Dynamic Memory:

The ALLOCATE and DEALLOCATE statements both use the optional flag STAT = to return an integer flag that can be tested to invoke an exception handler. The integer value is zero after a successful (de)allocation, and a positive value otherwise. If STAT = is absent, an unsuccessful result stops execution.

#### File Open/Close:

The OPEN, CLOSE, and ENDFILE statements allow the use of the optional keyword IOSTAT = to return an integer flag which is zero if the statement executes successfully, and a positive value otherwise. They also allow the older standard exception keyword ERR = to be assigned a positive integer constant label number of the statement to which control is passed if an error occurs. An exception handler could be called by that statement.

#### File Input/Output:

The READ, WRITE, BACKSPACE, and REWIND statements allow the IOSTAT = keyword to return a negative integer if an end-of-record (EOR) or end-of-file (EOF) is encountered, a zero if there is no error, and a positive integer if an error occurs (such as reading a character during an

```markdown
<table>
<thead>
<tr>
<th>Action</th>
<th>C++</th>
<th>F90</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitwise AND</td>
<td>&amp;</td>
<td>iand</td>
</tr>
<tr>
<td>Bitwise exclusive OR</td>
<td>^</td>
<td>ieor</td>
</tr>
<tr>
<td>Bitwise exclusive OR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Circular bit shift</td>
<td></td>
<td>ishftc</td>
</tr>
<tr>
<td>Clear bit</td>
<td></td>
<td>ibclr</td>
</tr>
<tr>
<td>Combination of bits</td>
<td></td>
<td>mvbits</td>
</tr>
<tr>
<td>Extract bit</td>
<td></td>
<td>ibits</td>
</tr>
<tr>
<td>Logical complement</td>
<td>~</td>
<td>not</td>
</tr>
<tr>
<td>Number of bits in integer</td>
<td>sizeof</td>
<td>bit_size</td>
</tr>
<tr>
<td>Set bit</td>
<td></td>
<td>ibset</td>
</tr>
<tr>
<td>Shift bit left</td>
<td>&lt;</td>
<td>ishft</td>
</tr>
<tr>
<td>Shift bit right</td>
<td>&gt;</td>
<td>ishft</td>
</tr>
<tr>
<td>Test on or off</td>
<td></td>
<td>btest</td>
</tr>
<tr>
<td>Transfer bits to integer</td>
<td></td>
<td>transfer</td>
</tr>
</tbody>
</table>
```

**Table 4.24:** Bit Function Intrinsics.
module exceptions

contains

subroutine exception (program, message, flag)
    character(len=*) :: program
    character(len=*) :: message
    integer, optional :: flag
    error_count = error_count + 1
    print *, 'Exception Status Thrown'
    print *, ' Program :', program
    print *, ' Message :', message
    if ( present(flag) ) then
        print *, ' Level :', flag
        if ( flag > max_level ) max_level = flag
    end if ! flag given
end subroutine exception

subroutine exception_status ()
    print *
    print *, "Exception Summary:"
    print *, " Exception count = ", error_count
    print *, " Highest level = ", max_level
end subroutine exception_status
end module exceptions

Figure 4.11: A Minimal Exception Handling Module

integer input). They also allow the ERR = error label branching described above for the file open/close operations.

In addition, the READ statement also retains the old standard keyword END = to identify a label number to which control transfers when an end-of-file (EOF) is detected.

Status Inquiry: Whether in UNIT mode or FILE mode, the INQUIRE statement for file operations allows the IOSTAT = and ERR = keywords like the OPEN statement. In addition, either mode supports two logical keywords: EXISTS = to determine if the UNIT (or FILE) exists, and OPENED = to determine if a (the) file is connected to this (an) unit.

Optional Arguments: The PRESENT function returns a logical value to indicate whether or not an optional argument was provided in the invocation of the procedure in which the function appears.

Pointers and Targets: The ASSOCIATED function returns a logical value to indicate whether a pointer is associated with a specific target, or with any target.

4.5 Interface Prototype

Compiler languages are more efficient than interpreted languages. If the compiler is going to correctly generate calls to functions, or subprograms, it needs to know certain things about the arguments and returned values. The number of arguments, their type, their rank, their order, etc. must be the same. This collection of information is called the “interface” to the function, or subprogram. In most of our example codes the functions and subprograms have been included in a single file. In practice they are usually stored in separate external files, and often written by others. Thus, the program that is going to use these external files must be given a “prototype” description of them. In other words, a segment of prototype, or interface, code is a definition that is used by the compiler to determine what parameters are required by the subprogram as it is called by your program. The interface prototype code for any subprogram can usually be created by simply copying the first few lines of the subprogram (and maybe the last one) and placing them in an interface directory.

To successfully compile a subprogram modern computer science methods sometimes require the programmer to specifically declare the interface to be used in invoking a subprogram, even if that subprogram is included in the same file. This information is called a “prototype” in C and C++, and an “interface” in F90. If the subprogram already exists, one can easily create the needed interface details by making
a copy of the program and deleting from the copy all information except that which describes the arguments and subprogram type. If the program does not exist, you write the interface first to define what will be expected of the subprogram regardless of who writes it. It is considered good programming style to include explicit interfaces, or prototype code, even if they are not required.

If in doubt about the need for an explicit interface see if the compiler gives an error because it is not present. In F90 the common reasons for needing an explicit interface are: 1) Passing an array that has only its rank declared. For example, \( A(:,:) \), \( B(1) \). These are known as “assumed-shape” arrays; 2) Using a function to return a result that is: a) an array of unknown size, or b) a pointer, or c) a character string with a dynamically determined length. Advanced features like optional argument lists, user defined operators, generic subprogram names (to allow differing argument types) also require explicit operators.

In C++ before calling an external function, it must be declared with a prototype of its parameters. The general form for a function is

\[
\text{function} \ 	ext{type} \functionname \ ( \ \text{argument} \ 	ext{type} \ \text{list});
\]

where the argument type list is the comma separated list of pairs of type and name for each argument of the function. These names are effectively treated as comments, and may be different from the names in the calling program, or even omitted. The use of a prototype was shown in Fig. 4.8 and is used again in Fig. 4.12 which also illustrates passing arguments by reference or by value.

An interface block for external subprograms was not required by F77 (thereby leading to hard to find errors), but is strongly recommended if F90 and is explicitly required in several situations. The general form for a F90 interface is

\[
\text{interface} \ \text{interface} \ 	ext{name} \\
\text{function} \ 	ext{interface} \ 	ext{body} \\
\text{subroutine} \ 	ext{interface} \ 	ext{body} \\
\text{module} \ 	ext{procedure} \ 	ext{interface} \ 	ext{body} \\
\text{end} \ \text{interface} \ \text{interface} \ 	ext{name}
\]

where a typical function interface body would be

\[
\text{function} \ 	ext{type} \functionname \ ( \ \text{argument} \ 	ext{name} \ \text{list}) \ 	ext{result} \ ( \ \text{name}) \\
\text{implicit none} \\
\text{argument} \ 	ext{type}, \intent_class :: \text{name} \ \text{list} \\
\text{end} \ \text{function} \ \text{name}
\]

where the argument name list is the comma separated list of names. Of course, the function type refers to the result argument name. These names may be different from the names in the calling program. A typical subroutine interface body would be

\[
\text{subroutine} \ \text{subroutine} \ 	ext{name} \ ( \ \text{argument} \ 	ext{name} \ \text{list}) \\
\text{implicit none} \\
\text{argument} \ 	ext{type}, \intent_class :: \text{name} \ \text{list} \\
\text{end} \ \text{subroutine} \ \text{subroutine} \ 	ext{name}
\]

where the argument name list is the comma separated list of names. The topic of a module procedure is covered elsewhere. The use of a interface block was shown in Fig. 4.8 and used in two new codes, shown in Fig. 4.12, and the corresponding C++ code in the appendix, which also illustrate passing arguments by reference (line 23) and by value (line 19) in both F90 and C++. The important, and often confusing, topic of passing by reference or value was discussed in Sec. 4.2 and is related to other topics to be considered later, such as the use of “pointers” in C++ and F90, and the “intent” attribute of F90 arguments. Passing by reference is default in F90 while passing by value is default in C++.

### 4.6 Characters and Strings

All of our example languages offer convenient ways to manipulate and compare strings of characters. The characters are defined by one of the international standards such as ASCII, which is usually used on UNIX, or the EBCDIC set. These contain both printable and non-printable (control) characters. On a UNIX system, the full set can be seen with the command `man ascii`. In the 256-character ASCII set, the upper case letters begin at character number 65, ‘A’, and the corresponding lower case values are

©2001 J.E. Akin
program main
! declare the interface prototypes
interface
subroutine Change (Refer)
  integer :: Refer; end subroutine Change
subroutine No_Change (Value)
  integer :: Value; end subroutine No_Change
end interface

! illustrate passing by reference and by value in F90
integer :: Input_Val, Dummy_Val
print *, "Enter an integer: "; read *, Input_Val
! pass by value
  call No_Change (Input_Val) ! Use but do not change
  print *, "After No_Change it is ", Input_Val
! pass by reference
  call Change (Input_Val) ! Use and change
  print *, "After Change it is ", Input_Val
end program

subroutine Change (Refer)
  integer :: Refer
  Refer = 100;
  print *, "Inside Change it is set to ", Refer
end subroutine Change

subroutine No_Change (Value)
  integer :: Value
  Value = 100;
  print *, "Inside No_Change it is set to ", Value
end subroutine No_Change

! Running gives:
! Enter an integer: 12
! Input value was 12
! After No_Change it is 12
! Inside Change it is set to 100
! After Change it is 100

Figure 4.12: Passing Arguments by Reference and by Value in F90

32 positions higher (character 97 is 'a'). These printable characters begin at character 32, as shown in Table 4.25 for the ASCII standard. The first 33 characters are "non-printing" special control characters. For example, NUL = null, EOT = end of transmission, BEL = bell, BS = backspace, and HT = horizontal tab. To enter a control character, one must simultaneously hold down the CONTROL key and hit the letter that is 64 positions higher in the list. That is, an end of transmission EOT is typed as CONTROL-D. The code SP denotes the space character, and we will use the underscore "_" to represent a blank in strings.

We can employ the standard relational operators (e.g., less than) to compare strings and would find that 'bad' < 'dog' < 'same' == 'same___', that 'word' > 'WORD', and that 'four' < 'one' < 'two' while '1' < '2' < '4'. Note that the above equality occurred because trailing blanks are not considered in relational operations, but leading blanks are considered: 'same' ≠ '___same'. The F90 function adjustL removes leading blanks and appends them to the right end. Thus, it adjusts the string to the left, so that 'same' == adjustL('___same'). This and other F90 intrinsic character functions are summarized in Table 4.26.

All blanks are considered when determining the length of a character string. In F90 the intrinsic function LEN provides these data so that LEN('same') = 4, LEN('___same') = 6, and LEN('same___') = 7. There is another intrinsic function, LEN_TRIM, that provides the string length ignoring trailing blanks. By way of comparison: LEN_TRIM('same') = 4, LEN_TRIM('___same') = 6, and LEN_TRIM('same___') = 4. Each character in a string or any internal substrings may be referenced by the colon operator. Given a character variable we can define a substring, say sub as

```
sub = variable(K:L) for 0 < K,L <= LEN(variable)
   = null for K > L
```

©2001 J.E. Akin 77
Table 4.25: The ASCII Character Set

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACHAR (I)</td>
<td>Character number I in ASCII collating set</td>
</tr>
<tr>
<td>ADJUSTL (STRING)</td>
<td>Adjust left</td>
</tr>
<tr>
<td>ADJUSTR (STRING)</td>
<td>Adjust right</td>
</tr>
<tr>
<td>CHAR (I) #</td>
<td>Character I in processor collating set</td>
</tr>
<tr>
<td>IACHAR (C)</td>
<td>Position of C in ASCII collating set</td>
</tr>
<tr>
<td>ICHAR (C)</td>
<td>Position of C in processor collating set</td>
</tr>
<tr>
<td>INDEX (STRING, SUBSTRING)</td>
<td>Starting position of a substring</td>
</tr>
<tr>
<td>LEN (STRING)</td>
<td>Length of a character entity</td>
</tr>
<tr>
<td>LEN/TRIM (STRING)</td>
<td>Length without trailing blanks</td>
</tr>
<tr>
<td>LGE (STRING A, STRING B)</td>
<td>Lexically greater than or equal</td>
</tr>
<tr>
<td>LGT (STRING A, STRING B)</td>
<td>Lexically greater than</td>
</tr>
<tr>
<td>LLE (STRING A, STRING B)</td>
<td>Lexically less than or equal</td>
</tr>
<tr>
<td>LLT (STRING A, STRING B)</td>
<td>Lexically less than</td>
</tr>
<tr>
<td>REPEAT (STRING, NCOPIES)</td>
<td>Repeated concatenation</td>
</tr>
<tr>
<td>SCAN (STRING, SET) a</td>
<td>Scan a string for a character in a set</td>
</tr>
<tr>
<td>TRIM (STRING)</td>
<td>Remove trailing blank characters</td>
</tr>
<tr>
<td>VERIFY (STRING, SET) a</td>
<td>Verify the set of characters in a string</td>
</tr>
<tr>
<td>STRING_A//STRING_B</td>
<td>Concatenate two strings</td>
</tr>
</tbody>
</table>

aOptional arguments not shown.

Table 4.26: F90 Character Functions

- error for K or L > LEN(variable).

For example, given the string ‘howl’, then we can define bird = string(2:4) = ‘owl’, and prep = string(1:3) = ‘how’.

The F90 and F77 operator used to concatenate strings into larger strings is ‘//’. Continuing the last example, we see that the concatenation string(1:3)//’ ‘//string(2:4)//’?’ is ‘how_owl?’, while the concatenation ‘same’//’word’ becomes ‘same_word’ and ‘bad’//’’//’dog’ becomes ‘bad__dog’. Programs illustrating the reading and concatenating two strings are given in Fig. 4.13, and in the companion C++ code in the appendix.

Sometimes one needs to type in a non-printing character, such as a tab or a newline. To allow this, special transmissions have been allowed for, as summarized in Table 4.27.

Remember the ASCII character features: the uppercase letters correspond to numbers 65 through 90 in the list, while the lowercase letters are numbers 97 through 122, so that if we wanted to convert “G” to
1) program main
2) ! Compare two strings
3) ! Concatenate two character strings together
4) ! Get the combined length
5) implicit none
6) character(len=20) :: String1, String2
7) character(len=40) :: String3
8) integer :: length
9) print *, 'Enter first string (20 char max):'
10) read '(a)', String1 ! formatted
11) print *, 'Enter second string (20 char max):'
12) read '(a)', String2 ! formatted
13) ! compare
14) if ( String1 == String2 ) then
15) print *, "They are the same."
16) else
17) print *, "They are different."
18) end if
19) ! concatenate
20) String3 = trim (String1) // trim (String2)
21) print *, 'The combined string is:', String3
22) length = len(trim (String3))
23) print *, 'The combined length is:', length
24) end program main
25) ! Runnng with "red" and "bird" produces:
26) ! Enter first string (20 char max): red
27) ! Enter second string (20 char max): bird
28) ! They are different.
29) ! The combined string is: redbird
30) ! The combined length is: 7
31) ! Also "the red" and "bird" works

Figure 4.13: Using Two Strings in F90

<table>
<thead>
<tr>
<th>Action</th>
<th>ASCII Character</th>
<th>F90 Input</th>
<th>C++ Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alert (Bell)</td>
<td>7</td>
<td>Ctrl-G</td>
<td>\a</td>
</tr>
<tr>
<td>Backspace</td>
<td>8</td>
<td>Ctrl-H</td>
<td>\b</td>
</tr>
<tr>
<td>Carriage Return</td>
<td>13</td>
<td>Ctrl-M</td>
<td>\r</td>
</tr>
<tr>
<td>End of Transmission</td>
<td>4</td>
<td>Ctrl-D</td>
<td>Ctrl-D</td>
</tr>
<tr>
<td>Form Feed</td>
<td>12</td>
<td>Ctrl-L</td>
<td>\f</td>
</tr>
<tr>
<td>Horizontal Tab</td>
<td>9</td>
<td>Ctrl-I</td>
<td>\t</td>
</tr>
<tr>
<td>New Line</td>
<td>10</td>
<td>Ctrl-J</td>
<td>\n</td>
</tr>
<tr>
<td>Vertical Tab</td>
<td>11</td>
<td>Ctrl-K</td>
<td>\v</td>
</tr>
</tbody>
</table>

“Ctrl-” denotes control action. That is, simultaneous pressing of the CONTROL key and the letter following.

Table 4.27: How to type non-printing characters.

“g” we could use commands such as:

```fortran
character (len = 1) :: lower_g, UPPER_G
lower_g = achar(iachar('G') + 32)
```

or visa versa:

```fortran
UPPER_G = achar(iachar('g') - 32)
```

since they differ by 32 locations. Likewise, since the zero character “0” occurs in position 48 of the ASCII set we could convert a single digit to the same numerical value with:

```fortran
integer :: number_5
number_5 = iachar('5') - 48
```

and so forth for all ten digits. To convert a string of digits, such as ’5623’, to the corresponding number 5623, we could use a looping operation.
program main
! Convert a character string to an integer in F90
implicit none
character(len=5) :: AgeChar
integer :: age
print *, "Enter your age: " ! a character string
read *, AgeChar ! a character string
read (AgeChar, fmt = '(i5)') age ! convert to integer
print *, "Your integer age is ", age
print '(" Your binary age is ", b8)', age
print '(" Your hexadecimal age is ", z8)', age
print '(" Your octal age is ", o8)', age
end program main

Figure 4.14: Converting a String to an Integer with F90

character (len = 132) :: digits
integer :: d_to_n, power, number
! Now build the number from its digits
if (digits == ' ') then
print *, 'warning, no number found'
number = 0
else
number = 0
k = len_trim(digits)
do m = k, 1, -1 ! right to left
d_to_n = iachar(digits(m:m)) - 48
power = 10**(k-m)
number = number + d_to_n*power
end do ! over digits
print *, 'number = ', number

However, since loops can be inefficient, it is better to learn that, in F90, an “internal file” can be (and
should be) employed to convert one data type to another. Here we could simply code:

! internal file called convert
write(convert, "'(A)'") digit
read(convert, "'(I4)'") number
to convert a character to an integer (or real) number. Converting strings to integers is shown in the codes
given in Fig. 4.14 (line 11) and the corresponding C++ appendix routine. Similar procedures would be
used to convert strings to reals. The C++ version (see appendix) uses the intrinsic function “atoi” while
the F90 version uses an internal file for the conversion.

One often finds it useful to change the case of a string of characters. Some languages provide intrinsic
functions for that purpose. In C++ and MATLAB the function to convert a string to all lower case letters
are called tolower and lower, respectively. Here we define a similar F90 function called to_lower
which is shown in Fig. 4.15 along with a testing program in Fig. 4.16. Note that the testing program
uses an interface to tolower (lines 4-13) assuming that routine was compiled and stored external to the
testing program. The tolower function employs the intrinsic function index (line 16) to see if the k-th
character of the input string is an upper case letter. The intrinsic function len is also used (line 8) to
force the new_string to be the same length as the original string.

4.7 User Defined Data Types

Variables, as in mathematics, represent some quantity; unlike mathematics, many languages force the
programmer to define what type the variable is. Generic kinds of type are integer, floating point (single,
double, and quadruple precision), and complex-valued floating point. Table 4.2 (page 53) presents the
data types inherent in the various languages. Most beginning programmers find the requirement most
function to_lower (string)  result (new_string) ! like C
! ---------------------------------------------------------------------
! Convert a string or character to lower case
! (valid for ASCII or EBCDIC processors)
! ---------------------------------------------------------------------
implicit none
character (len = *), intent(in) :: string ! unknown length
character (len = len(string)) :: new_string ! same length
character (len = 26), parameter :: &
   UPPER = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ', &
   lower = 'abcdefghijklmnopqrstuvwxyz'
integer :: k ! loop counter
integer :: loc ! position in alphabet
new_string = string ! copy everything
do k = 1, len(string) ! to change letters
   loc = index ( UPPER, string(k:k)) ! first upper
   if (loc /= 0 ) new_string(k:k) = lower(loc:loc) ! convert it
end do ! over string characters
end function to_lower

program up_down ! test character case inversion functions
implicit none
character (len = 24) :: test='ABCDefgh1234abcdZYXWzyxw'
interface
   function to_lower (string) result (new_string)
      character (len = *) , intent(in) :: string
      character (len = len(string)) :: new_string
   end function to_lower
   function to_upper (string) result (new_string)
      character (len = *) , intent(in) :: string
      character (len = len(string)) :: new_string
   end function to_upper
end interface
print *, test
print *, to_lower (test)
print *, to_upper (test)
end program ! running gives

Figure 4.15: Converting a String to Lower Case with F90

languages impose of defining explicitly each variable’s type to be tedious, unnecessary, and a source of bugs. It’s tedious because the programmer must think not only about what the variable represents, but also how the computations calculate its value, unnecessary because mathematics doesn’t work that way (the variable \( x \) represents a quantity regardless whether it turns out to be an integer or a complex value), and bug-creating because computations involving different types and assigned to a typed variable can yield nonmathematical results (for example, dividing the integers 1 with 3 and assigning the results to an integer yields a zero value).

MATLAB is one language in which variables are not explicitly typed. (Beginning programmers cheer!) Internally, MATLAB represents numbers in double precision floating point. If a variable’s value corresponds to an integer, MATLAB will gleefully print it that way, effectively hiding its floating point representation. A surprise occurs when a calculation accidentality becomes complex: MATLAB will (silently) change what the variable represents from being real to being complex. For example, MATLAB will, without complaint, calculate \( x = \log(-1) \) and assign the value \( 3.14159i \) to \( x \). In many applications, the expression that yielded the value of \(-1\) because of an error, and MATLAB will let the error propagate. (Beginning programmers sigh!) Most, if not all typed languages will immediately announce the evaluation of the logarithm of a negative number, and halt execution. By explicitly defining the kinds of values a variable will assume helps programming clarity and run-time debugging to some degree.

C++ has four intrinsic (i.e., built-in) types of data—integer, single and double precision reals, and character—and F90 has the similar set: integer, real, complex, logical, and character. F90 also allows the user to create a specific precision level for integer and real data. C++ has specified byte sizes for three character, six integer, one single precision real, and two double precision real data types for a total of twelve intrinsic data types.

©2001 J.E. Akin
In addition to intrinsic types, C, C++ and F90 allow the formation of new types of data—structures—that are collections of values of not necessarily the same type. These procedures are named struct or type in C and F90, respectively.

To go along with this freedom, F90 allows you to define new operations to act on the derived types. While C++ retains the struct keyword, it is viewed as a class with only public data members and no functions. In other words, in C++ class is a generalization of struct and, thus, class is the preferred keyword to use. As an example of a task made easier by derived data, consider creating parts of a data structure to be used in an address book. We will need a variable that can have components and sub-components. They are referenced by a special syntax and defined as illustrated in Tables 4.28 and 4.29.

This procedure for defining a new type of data structure can be “nested” by referring to other derived type entities defined earlier in the program. These concepts are shown in Table 4.30. One should declare the data type of all variables used in a program module. This is also true for user defined data structures. Table 4.31 outlines the forms of these statements, how structures are initialized, and how component values are assigned.

There are times when either the derived type variable or its components, or both, could be subscripted objects (i.e., arrays). Then care must be taken in the interpretation of which variable or component is being addressed. Table 4.32 illustrates the typical combinations with the F90 syntax.

As a concrete example, consider a phone_type and address_type definition.
Table 4.31: Declaring, initializing, and assigning components of user-defined datatypes.

<table>
<thead>
<tr>
<th>Construct</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>derived</td>
<td>All components of all derived's elements</td>
</tr>
<tr>
<td>derived(j)</td>
<td>All components of ( j )th element of derived</td>
</tr>
<tr>
<td>derived(j)%name</td>
<td>All ( k )th components of ( \text{name} ) within ( j )th element of derived</td>
</tr>
<tr>
<td>derived%name(k)</td>
<td>Component ( k ) of the ( \text{name} ) array for all elements of derived</td>
</tr>
<tr>
<td>derived(j)%name(k)</td>
<td>Component ( k ) of the ( \text{name} ) array of ( j )th element of derived</td>
</tr>
</tbody>
</table>

Table 4.32: F90 Derived Type Component Interpretation.

<table>
<thead>
<tr>
<th>F90</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>type phone_type</td>
<td>struct phone_type {</td>
</tr>
<tr>
<td>integer :: area_code, number, extension</td>
<td>int area_code, number, extension;</td>
</tr>
<tr>
<td>end type phone_type</td>
<td>}</td>
</tr>
<tr>
<td>type address_type</td>
<td>struct address_type {</td>
</tr>
<tr>
<td>integer :: number</td>
<td>int number;</td>
</tr>
<tr>
<td>character (len = 35) :: street, city</td>
<td>char street[35], city[35];</td>
</tr>
<tr>
<td>character (len = 2) :: state</td>
<td>char state[2];</td>
</tr>
<tr>
<td>integer :: zip_code</td>
<td>int zip_code;</td>
</tr>
<tr>
<td>end type address_type</td>
<td>}</td>
</tr>
</tbody>
</table>

These could be used to define part of a `person_type`

<table>
<thead>
<tr>
<th>F90</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>type person_type</td>
<td>struct person_type {</td>
</tr>
<tr>
<td>character (len = 50) :: name</td>
<td>char name[50];</td>
</tr>
<tr>
<td>type (phone_type) :: phone</td>
<td>struct phone_type phone;</td>
</tr>
<tr>
<td>type (address_type) :: address</td>
<td>struct address_type address;</td>
</tr>
<tr>
<td>integer :: born_year</td>
<td>int born_year;</td>
</tr>
<tr>
<td>end type person_type</td>
<td>}</td>
</tr>
</tbody>
</table>

We define two people with

<table>
<thead>
<tr>
<th>F90</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>type (person_type) :: sammy, barney</td>
<td>struct person_type sammy, barney;</td>
</tr>
</tbody>
</table>

or build an address book array filled with the above data structures by defining
F90

class, parameter :: number = 99
type (person_type), dimension (number) :: address book

C++

#define NUMBER 99
struct person_type address book[NUMBER];

and then initialize, or “construct” sammy’s phone and zip code as

F90

c = (713, 5278100, 0)
s = 770051892

C++

c = {713, 5278100, 0};
s = 770051892;

and print them with

F90

print *, c
print *, s

C++

printf("(%d)%d, extension %d",
c.area, c.number, c.extension);
printf("%d", s);

and then define specific members for barney with the “constructor”

F90

barney = person type("Barn Owl", &
c, &
s)

C++

barney = {"Barn Owl", {0, 0, 0},
c, s} ;

Note the difference in the defined type constructors. Two are actually used here because the second component must be defined as a phone type. C++ just uses brackets to enclose the supplied components of each user defined type. F90 has an intrinsic function that is created automatically by the type definition and it accepts all of the components required by the type. That is why the function name “phone type” appears in the intrinsic constructor routine “person type”. Finally, put them in the book.

F90

address book(1) = sammy
address book(2) = barney

C++

address book[1] = sammy;
address book[2] = barney;

Fig. 4.17 presents a sample code for utilizing user defined structure types using F90 (there is a C++ version in the appendix). First a “person” structure is created (lines 4-7) by using only the intrinsic types of integers and characters. It then is used in turn within an additional data structure (line 10). The components of the structures are read (lines 18, 21, 24) and output (lines 26, 27). For more general data, suggested in the comments, formatted input/output controls would be necessary.

4.7.1 Overloading Operators

As a complete short example of utilizing many of the new programming features that come with user defined data structures we will consider the use of a familiar old mathematics system, fractions. Recall that a fraction is the ratio of two integers. We will therefore define a new data type called Fraction. It
program main()
! Define structures and components, via F90
implicit none

type Person ! define a person structure type
  character (len=20) :: Name
  integer :: Age
end type Person

type WhoWhere ! use person type in a new structure
  type (Person) :: Guest
  character (len=40) :: Address
end type WhoWhere

! Fill a record of the WhoWhere type components
type (WhoWhere) Record;
  print *, "Enter your name: "
  read *, Record % Guest % Name
  print *, "Enter your city: "
  read *, Record % Address
  print *, "Enter your age: "
  read *, Record % Guest % Age
  print *, "Hello ", Record % Guest % Age, " year old ", 
          " in ", Record % Address
end program main

Figure 4.17: Using Multiple Structures in F90

will simply consist of two integer types, named num and denom, respectively. New data types can be defined in any program unit. For maximum usefulness we will place the definition in a module named Fractions. To use this new data type we will want to have subprograms to define a fraction, list its components, and multiply two fractions together, and to equate one fraction to another. In addition to the intrinsic constructor function fraction we will create a manual constructor function called assign and it will have two arguments, the numerator value, and denominator value, and will use them to return a fraction type. The listing subroutine, called list_Fraction, simply needs the name of the fraction to be printed. The function, mult_Fraction, accepts two fraction names, and returns the third fraction as their product. Finally, we provide a function that equates the components of one fraction to those in a new fraction.

This data structure is presented in Fig. 4.18. There we note that the module starts with the definition of the new data type (lines 2-4), and is followed with the "contains" statement (line 12). The subprograms that provide the functionality of the fraction data type follow the "contains" statement and are thus coupled to the definition of the new type. When we have completed defining the functionality to go with the new data type we end the module.

In this example the program to invoke the fraction type follows in Fig. 4.19. To access the module, which defines the new data type and its supporting functions, we simply employ a "use" statement at the beginning of the program (line 2). The program declares three Fraction type variables (line 3): x, y, and z. The variable x is defined to be 22/7 with the intrinsic type constructor (line 5), while y is assigned a value of 1/3 by using the function assign (line 7). Both values are listed for confirmation. Then we form the new fraction, z = 22/21, by invoking the mult_Fraction function (line 9),

\[ z = \text{mult}_\text{Fraction} (x, y) \]

which returns z as its result. A natural tendency at this point would be to simply write this as \( z = x \times y \). However, before we could do that we would have to tell the operators, "\( \times \)" and "\( = \)" how to act when provided with this new data type. This is known as overloading an intrinsic operator. We had the foresight to do this when we set up the module by declaring which of the "module procedure"s were equivalent to each operator symbol. Thus from the "interface operator (\( \times \))" statement block the system now knows that the left and right operands of the "\( \times \)" symbol correspond to the first and second arguments in the

©2001 J.E. Akin 85
module Fractions  ! F90 "Fraction" data structure and functionality
implicit none
type Fraction  ! define a data structure
  integer :: num, den ! with two "components"
end type Fraction

interface operator (*)  ! extend meaning to fraction
  module procedure mult_Fraction ; end interface

interface assignment (=)  ! extend meaning to fraction
  module procedure equal_Fraction ; end interface
contains  ! functionality
subroutine assign (name, numerator, denominator)
  type (Fraction), intent(inout) :: name
  integer, intent(in) :: numerator, denominator
  name % num = numerator ! % denotes which "component"
  if ( denominator == 0 ) then
    print *, "0 denominator not allowed, set to 1"
    name % den = 1
  else; name % den = denominator
  end if ; end subroutine assign

subroutine list(name)
  type (Fraction), intent(in) :: name
  print *, name % num, "/", name % den ; end subroutine list

function mult_Fraction (a, b) result (c)
  type (Fraction), intent(in) :: a, b
  type (Fraction) :: c
  c%num = a%num * b%num ! standard = and * here
  c%den = a%den * b%den ; end function mult_Fraction

subroutine equal_Fraction (new, name)
  type (Fraction), intent(out) :: new
  type (Fraction), intent(in) :: name
  new % num = name % num ! standard = here
  new % den = name % den ; end subroutine equal_Fraction
end module Fractions

Figure 4.18: Overloading operations for new data types

Function mult_Fraction. Likewise, the left and right operands of "=" are coupled to the first and second arguments, respectively, of subroutine equal_Fraction. The testing main and verification results are in Fig. 4.19 Before moving on note that the system does not yet know how to multiply a integer times a fraction, or visa versa. To do that one would have to add more functionality, such as a function, say int_mult_frac, and add it to the "module procedure" list associated with the "=" operator.

When considering which operators to overload for a newly defined data type one should consider those that are used in sorting operations, such as the greater-than, >, and less-than, <, operators. They are often useful because of the need to sort various types of data. If those symbols have been correctly overloaded then a generic sorting routine might be used, or require few changes.

4.7.2 User Defined Operators
In addition to the many intrinsic operators and functions we have seen so far, the F90 user can also define new operators or extend existing ones. User defined operators can employ intrinsic data types and/or user defined data types. The user defined operators, or extensions, can be unary or binary (i.e., have one or two arguments). The operator symbol must be included between two periods, such as '.op.' Specific examples will be given in the next chapter.

4.8 Pointers and Targets
The beginning of every data item must be stored in computer memory at a specific address. The address of that data item is called a pointer to the data item, and a variable that can hold such an address is called a pointer variable. Often it is convenient to have a pointer to a variable, an array, or a sub-array. F90, C++ and MATLAB provide this sophisticated feature. The major benefits of the use of pointers is that...
program main
use Fractions
implicit none
type (Fraction) :: x, y, z

x = Fraction (22,7) ! default constructor
write (*,'("default x = ")', advance='no') ; call list(x)
call assign(y,1,3) ! manual constructor
write (*,'("assigned y = ")', advance='no') ; call list(y)
z = mult (x,y) ! function use
write (*,'("z = x mult y = ")', advance='no') ; call list(z);
print *, "Trying overloaded * and = for fractions:"
write (*,'("y * x gives ")', advance='no') ; call list(y*x) ! multi

z = x*y ! new operator uses
write (*,'("z = x*y gives ")', advance='no') ; call list(z) ! add
end program main ! Running gives:
! default x = 22/7 ! assigned y = 1/3 ! x mult y = 22/21
! Trying overloaded * and = for fractions:
! y * x gives 22/21 ! z = x*y gives 22/21

Figure 4.19: Testing overloading for new data types

<table>
<thead>
<tr>
<th>C++</th>
<th>F90</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>type_tag *pointer_name;</td>
</tr>
<tr>
<td>Target</td>
<td>&amp;target_name</td>
</tr>
<tr>
<td>Examples</td>
<td>char *cp, c;</td>
</tr>
<tr>
<td></td>
<td>int *ip, i;</td>
</tr>
<tr>
<td></td>
<td>float *fp, f;</td>
</tr>
<tr>
<td></td>
<td>cp = &amp; c;</td>
</tr>
<tr>
<td></td>
<td>ip = &amp; i;</td>
</tr>
<tr>
<td></td>
<td>fp = &amp; f;</td>
</tr>
</tbody>
</table>

Table 4.33: Definition of pointers and accessing their targets.

they allow dynamic data structures, such as “linked lists” and “tree structures,” and they allow recursive algorithms. Note that rather than containing data themselves, pointer variables simply exist to point to where some data are stored. Unlike C and MATLAB the F90 pointers are more like the “reference variables” of the C++ language in that they are mainly an alias or synonym for another variable, or part of another variable. They do not allow one to easily get the literal address in memory as does C. This is why programmers that write computer operating systems usually prefer C over F90. But F90 pointers allow easy access to array partitions for computational efficiency, which C++ does not. Pointers are often used to pass arguments by reference.

The item to which a pointer points is known as a target variable. Thus, every pointer has a logical status associated with it which indicates whether or not it is currently pointing to a target. The initial value of the association is .false., or undefined.

4.8.1 Pointer Type Declaration

For every type of data object that can be declared in the language, including derived types, a corresponding type of pointer and target can be declared (Table 4.33).

While the use of pointers gives programmers more options for constructing algorithms, they also have a potential severely detrimental effect on the program execution efficiency. To ensure that compilers can produce code that execute efficiently, F90 restricts the variables, to which a pointer can point, to those specifically declared to have the attribute target. This, in part, makes the use of pointers in F90 and C++ somewhat different. Another major difference is that C++ allows arithmetic to be performed on the pointer address, but F90 does not.

So far, we have seen that F90 requires specific declarations of a pointer and an potential target. However, C++ employs two unary operators, & and *, to deal with pointers and targets, respectively. Thus, in C++ the operator &variable_name means “the address of” variable_name, and the C++ operator *pointer_name means “the value at the address of” pointer_name.
### Table 4.34: Nullifying a pointer to break target association.

<table>
<thead>
<tr>
<th>C, C++</th>
<th>F90, F95</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pointer_name = NULL</code></td>
<td><code>nullify (list_of_pointer_names)</code></td>
</tr>
<tr>
<td><code>pointer_name = NULL()</code></td>
<td><code>nullify (list_of_pointer_names)</code></td>
</tr>
</tbody>
</table>

#### Figure 4.20: Using F90 Pointers in Expressions.

```fortran
program pt_expression
 implicit none integer, POINTER :: p, q, r integer, TARGET :: i = 1, j = 2, k = 3 q => j ! q points to integer j p => i ! p points to integer i q - p + 2 ! means: j - i + 2 - 1 + 2 - 3 print *, i, j, k ! print target values p => k ! p now points to k print *, (q-p) ! means print j - k = 3 - 3 = 0 ! automatically substitues the target value: r => k ! now r points to k, also ! Check associations of pointers print *, associated (r) ! false print *, associated (q, r) ! true ! associated (p, i) ! false ! associated (p, k) ! true ! associated (r, k) ! true end program pt_expression
```

### 4.8.2 Pointer Assignment

F90 requires that a pointer be associated with a target by a single pointer assignment statement. C allows, but does not require, a similar statement. (See Table 4.33). After such a statement, the pointer has a new association status and one could employ the F90 intrinsic inquiry function `associated(pointer_name, target_name)` to return `.true.` as the logical return value. If one wishes to break or nullify a pointer’s association with a target, but not assign it another target, one can nullify the pointer as shown in Table 4.34.

### 4.8.3 Using Pointers in Expressions

The most important rule about using pointers in F90 expressions is that, where ever a pointer occurs, it is treated as its associated target. That is, the target is automatically substituted for the pointer when the pointer occurs in an expression. For example, consider the actions in Fig. 4.20 (where the results are stated as comments).

### 4.8.4 Pointers and Linked Lists

Pointers are the simplest available mechanism for dynamic memory management of arrays such as stacks, queues, trees, and linked lists. These are extraordinarily flexible data structures because their size can grow or shrink during the execution of a program. For linked lists the basic technique is to create a derived type that consists of one or more data elements and at least one pointer. Memory is allocated to contain the data and a pointer is set to reference the next occurrence of data. If one pointer is present, the list is a singly-linked list and can only be traversed in one direction: head to tail, or vice versa. If two pointers are present: the list is a doubly-linked list and can be traversed in either direction. Linked lists allow the data of interest to be scattered all over memory and uses pointers to weave through memory, gathering data as required. Detailed examples of the use of linked lists are covered in Chapter 8.

As a conceptual example of when one might need to use linked-lists think of applications where one never knows in advance how many data entries will be needed. For example, when a surveyor determines the exact perimeter of a building or plot of land, critical measurements are taken at each
angle. If the perimeter has $N$ sides, the surveyor measures the length of each side and the interior angle each side forms with the next. Often the perimeter has visual obstructions and offsets around them must be made, recorded, and corrected for later. Regardless of how careful the surveyor is, errors are invariably introduced during the measurement process. However, the error in angle measurements can be bounded.

The program for implementing the recording and correcting of the angles in a survey could be written using a singly linked list. A linked list is chosen because the programmer has no idea how many sides the perimeter has, and linked lists can grow arbitrarily. Because of the linked list’s ability to absorb a short or long data stream, the user does not have to be asked to count the number of legs in the traverse. The program begins by declaring a derived type that contains one angle measurement and a pointer to the next measurement. A count is kept of the number of legs in this loop and the forward pointer for the last angle read is cleared (set to null) to signal the end of list. After all the data are read, the entire list of angles is reviewed to get the total of the measurements. This starts by revisiting the head of the list and adding together all the angle measurements until a null pointer is encountered, signaling the end of list. Then the error can be computed and distributed equally among the legs of the traverse.

### 4.9 Accessing External Source Files and Functions

At times one finds it necessary, or efficient to utilize other software from libraries, other users, or different paths in your directories. Of course, you could always use the brute force approach and use a text editor to copy the desired source code into your program. However, this is unwise not only because it wastes storage, but more importantly gives multiple copies of a module that must all be found and changed if future revisions are needed or desired. Better methods of accessing such codes can be defined either inside your program, or external to it in the “linking” phase after compiling has been completed.

High level languages like C, C++, and F90 allow one or more approaches for accessing such software from within your code. One feature common to all these languages is the availability of an “include” statement which gives the system path to the desired code file. At compile time, and only then, a temporary copy of the indicated code from that file is literally copied and inserted into your program at the location of the corresponding “include” statement.

It is common practice, but not required, to denote such code fragments with name extensions of “.h” and “.inc”, in C++ and F90, respectively. For example, to use a program called “class__Person” one could insert the following statement in your program:

```c
C, C++: include <class__Person.h>
F90 : include 'class__Person.inc'
```

if the files, class__Person.h or class__Person.inc, were in the same directory as your program. Otherwise, it is necessary to give the complete system path to the file, such as,

```c
include '/home/caam211/Include/inv.f90'
include '/home/caam211/Include/SolveVector.f90'
```

which give source links to the caam211 course files for the function inv(A) for returning the inverse of a matrix A, and the function SolveVector(A,B) which returns the solution vector X for the matrix system $A\times X = B$.

In F90 one can also provide a “module” that defines constants, user defined types, supporting subprograms, operators, etc. Any of those features can be accessed by first including such a F90 module before the main program and later invoking it with a “use” statement which cites the “module” name. For example, the F90 program segments:

```f90
include '/home/caam211/Include/caam211_operators.f90'
Program Lab2_A2
  call test_matrix ( A, B, X ) ! form and invert test matrix
  ...
subroutine test_matrix ( A, B, X )
  use caam211_operators ! included above
  implicit none
  real :: A(:,,:), B(:,), X(:)
  real :: A_inv(size(A,1),size(A,1)) ! automatic array allocation
  A_inv = inv(A)
  X = A .solve. B ! like X = A \ B in Matlab
  ...
```

©2001 J.E. Akin
gives a source link to the caam211 course “module” source file named caam211_operators.f90 which contains subprograms, such as the function inv(), and operator definitions like .solve. which is equivalent to the “\" operator in MATLAB.

In the last example the omission of the “include” statement would require a compiler dependent statement to allow the system to locate the module cited in the “use” statement. For the National Algorithms Group (NAG) F90 compiler that link would be given as

```
f90 -o go /home/caam211/Include/caam211_operators.f90 my.f90
```

if the above segment was stored in the file named my.f90, while for the Cray F90 compiler a path flag, -p, to the compiled version is required, such as:

```
f90 -o go -p /home/caam211/Include/caam211_op_CRAY.o my.f90
```

Either would produce an executable file, named “go” in this example.

### 4.10 Procedural Applications

In this section we will consider two common examples of procedural algorithms: fitting curves to experimental data, and sorting numbers, strings, and derived types. Sorting concepts will be discussed again in Chapter 7.

#### 4.10.1 Fitting Curves to Data

We must often deal with measurements and what they result in: data. Measurements are never exact because they are limited by instrument sensitivity and are contaminated by noise. To determine trends (how measurements are related to each other), confirm theoretical predictions, and the like, engineers must frequently fit functions to data. The “curve” fit is intended to be smoother than a raw plot of the data, hopefully revealing more about the underlying relation between the variables than would otherwise be apparent.

Often, these functions take parametric form: The functional form is specified, but has unknown coefficients. Suppose you want to fit a straight line to a dataset. With $y$ denoting the measurement and $x$ the independent variable, we wish to fit the function $y = f(x) = mx + b$ to the data. The fitting process amounts to determining a few quantities of the assumed linear functional form—the parameters $m$ and $b$—from the data. You know that two points define a straight line; consequently, only two of the $(x_i,y_i)$ pairs need be used. But which two should be used? In virtually all real-world circumstances, the measurements do not precisely conform to the assumed functional form. Thus, fitting a curve by selecting a few values (two in the linear case) and solving for the function’s parameters produces a circumspect “fit”, to say the least. Instead, the most common approach is to use all the data in the curve fitting process. Because you frequently have much more data than parameters, you have what is known as an over-determined problem. In most cases, no parameter values produce a function that will fit all the data exactly. Over-determined problems can be solved by specifying an error criterion (what is an error and how large is the deviation of data from the assumed curve) and finding the set of parameter values that minimizes the error criterion. With this approach, we can justifiably claim to have found the best parameter choices.

The “Least Squares” Approach

Far and away the most common error criterion is the mean-squared error: Given measurement pairs $(x_i,y_i), i = 1,\ldots,N$, the mean squared error $e^2$ equals the average across the dataset of $(y_i - f(x_i))^2$, the squared error between the $i^{th}$ measurement and the assumed parametric function $f(x_i)$.

$$
e^2 = \frac{1}{N} \sum_{i=1}^{N} (y_i - f(x_i))^2$$

Least squares fitting of functions to data amounts to minimizing the dataset’s mean squared error with respect to the parameters.

©2001 J.E. Akin
To illustrate the least-squares approach, let’s fit a linear function to a dataset. Substituting the assumed functional form \( f(x) = mx + b \) into the expression for the mean-squared error, we have

\[
e^2 = \frac{1}{N} \sum_{i=1}^{N} (y_i - (mx_i + b))^2
\]

We can find a set of equations for the parameters \( m \) and \( b \) that minimize this quantity by evaluating the derivative of \( e^2 \) with respect to each parameter and setting each to zero.

\[
\frac{de^2}{dm} = \frac{1}{N} \sum_{i=1}^{N} -2x_i(y_i - (mx_i + b)) = 0
\]

\[
\frac{de^2}{db} = \frac{1}{N} \sum_{i=1}^{N} -2(y_i - (mx_i + b)) = 0
\]

After some simplification, we find that we have two linear equations to solve for the fitting parameters.

\[
m \cdot \left( \frac{1}{N} \sum_{i=1}^{N} x_i^2 \right) + b \cdot \left( \frac{1}{N} \sum_{i=1}^{N} x_i \right) = \frac{1}{N} \sum_{i=1}^{N} x_i y_i
\]

\[
m \cdot \left( \frac{1}{N} \sum_{i=1}^{N} x_i \right) + b = \frac{1}{N} \sum_{i=1}^{N} y_i
\]

Thus, finding the least-squares fit of a straight line to a set of data amounts to solving a set of two linear equations, the coefficients of which are computed from the data. Note that the four summations in the last equation have the same range count (N) and could be evaluated in a single explicit loop.

### An Aside

Because fitting data with a linear equation yields a set of two easily solved equations for the parameters, one approach to fitting nonlinear curves to data is to convert the nonlinear problem into a linear one. For example, suppose we want to fit a power law to the data: \( f(x) = ax^b \). Instead of minimizing the mean squared error directly, we transform the data so that we are fitting it with a linear curve. In the power law case, the logarithm of the fitting curve is linear in the parameters: \( \log f(x) = \log a + b \log x \). This equation is not linear in the parameter \( a \). For purposes of least-squares fits, we instead treat \( a' = \log a \) as the linear fit parameter, solve the resulting set of linear equations for \( a' \), and calculate \( a = \exp a' \) to determine the power law fitting parameter. By evaluating the logarithm of \( x_i \) and \( y_i \) and applying the least squares equations governing the fitting of a linear curve to data, we can fit a power-law function to data.

Thus, calculating a linear least squares fit to data underlies general approximation of measurements by smooth curves. For an insight to the types of relationships that can be determined, see the following summary.

<table>
<thead>
<tr>
<th>( x )-axis</th>
<th>( y )-axis</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>Linear</td>
<td>( y = mx + b )</td>
</tr>
<tr>
<td>Linear</td>
<td>Logarithmic</td>
<td>( \log y = mx + b )</td>
</tr>
<tr>
<td>Logarithmic</td>
<td>Linear</td>
<td>( y = m \log x + b )</td>
</tr>
<tr>
<td>Logarithmic</td>
<td>Logarithmic</td>
<td>( \log y = m \log x + b )</td>
</tr>
</tbody>
</table>

We can now specify the computations required by the least squares fitting algorithm mathematically.

### Algorithm: Least-Squares Fitting of Straight Lines to Data

1. **Given** \( N \) pairs of data points \((x_i, y_i)\)
2. **Calculate**¹ \( a_{11} = \frac{1}{N} \sum_{i=1}^{N} x_i^2, \ a_{12} = \frac{1}{N} \sum_{i=1}^{N} x_i, \ a_{21} = \frac{1}{N} \sum_{i=1}^{N} x_i, \ a_{22} = 1, \ c_1 = \frac{1}{N} \sum_{i=1}^{N} x_i y_i, \) and \( c_2 = \frac{1}{N} \sum_{i=1}^{N} y_i. \)

¹Note that these calculations can be performed in one loop rather than four.
3. Solve the set of linear equations

\[
\begin{bmatrix}
  a_{11} & a_{12} \\
  a_{21} & a_{22}
\end{bmatrix}
\begin{bmatrix}
  m \\
  b
\end{bmatrix}
= \begin{bmatrix}
  c_1 \\
  c_2
\end{bmatrix}
\]

which for two equations can be done by hand to yield

\[
m = (a_{12} \cdot c_2 - N \cdot c_1)/(a_{12} \cdot a_{21} - N \cdot a_{11})
\]

\[
b = (c_2 - m \cdot a_{12})/N
\]

4. Calculate the mean squared error $e^2 = \frac{1}{N} \sum_{i=1}^{N} (y_i - (mx_i + b))^2$.

Implementing the Least Squares Algorithm

In F90, such calculations can be performed two different ways: one expresses the looping construct directly, the other uses more efficient intrinsic array routines inside F90. Assuming the $\{x_i\}$ are stored in the vector $x$, the coefficient $a_{12}$ can be calculated (at least) two ways.

1. \[
\text{sum}_x = 0
\]
\[
N = \text{size}(x)
\]
\[
do \ i = 1,N
\]
\[
\text{sum}_x = \text{sum}_x + x(i)
\]
\[
end \ do
\]
\[
a_{12} = \text{sum}_x/N
\]

2. \[
a_{12} = \text{sum}(x)/\text{size}(x)
\]

Clearly, the second method produces a somewhat simpler expression than the first, and is vastly superior to the first. In the sample code that follows in Fig. 4.21 we use the intrinsic array functions but encourage the reader to check the results with a single loop that computes all six terms need to find $m$ and $b$.

There are a few new features demonstrated in this example code. In line 6 we have specified a fixed unit number to associate with the data file to be specified by the user. But we did not do an INQUIRE to see if that unit was already in use. We will accept a user input filename (lines 8, 25 and 28) that contains the data to be fitted. An interface (lines 12-21) is provided to external routines that will determine the number of lines of data in the file and the read those data into the two arrays. Those two routines are given elsewhere. Of course, the memory for the data arrays must be dynamically allocated (line 35) before they can be read (line 37). After the least squares fit is computed (line 40) and printed the memory space for the data is freed (line 44).

In the `lsq_fit` subroutine (line 47) the three items of interest are passed in the array `fit`. (Routine `lsq_fit` could have been written as a function, try it.) Observe that $y$ must be the same length as array $x$ so the `size` intrinsic was used to ensure that (line 56). The data summations are evaluated with the `sum` intrinsic (lines 62-64) and it is used again to evaluate the mean squared error `mse` (line 72) as described in step 4 of the algorithm. The test data (lines 78-89) and results (lines 92-96) are given as comments as usual. Since no explicit loops have been used this form would be more efficient on vector computers and some parallel computers.

4.10.2 Sorting

One of the most useful computational routines is sorting: Ordering a sequence of data according to some rule. For example, the alphabetized list of filenames produced by a system directory command is far easier to read than an unsorted list would be. Furthermore, data can be fruitfully sorted in more than one way. As an example, you can sort system files by their creation date.

Sorting algorithms have been well studied by computer scientists in a quest to find the most efficient. We use here the bubble sort algorithm, perhaps the oldest, but not most efficient. This algorithm makes multiple passes over a list, going down the list interchanging adjacent elements in the list if needed to put them in order. For example, consider the list $\{b, e, a, d, f, c\}$, shown in Fig. 4.22, that we
program linear_fit
! ------------------------------------------------------
! F90 linear least-squares fit on data in file specified by the user.
! ------------------------------------------------------
implicit none
integer, parameter :: filenumber = 1 ! RISKY
real, allocatable :: x(:), y(:) ! data arrays
character (len = 64) :: filename ! name of file to read
integer :: lines ! number of input lines
real :: fit(3) ! final results

interface
function inputCount(unit) result(linesOfInput)
integer, intent(in) :: unit ! file unit number
integer :: linesOfInput ! result
end function inputCount
end interface

interface
subroutine readData (inFile, lines, x, y)
integer, intent(in) :: inFile, lines ! file unit, size
real, intent(out) :: x(lines), y(lines) ! data read
end subroutine readData
end interface

! Get the name of the file containing the data.
write (*,*) 'Enter the filename to read data from:'
read (*,'(A64)') filename
!
! Open that file for reading.
open (unit = filenumber, file = filename)
!
! Find the number of lines in the file
lines = inputCount (filenumber)
write (*,*) 'There were ',lines,' records read.'
!
! Allocate that many entries in the x and y array
allocate (x(lines), y(lines)) ! data read
!
! Read data
call readData (filenumber, lines, x, y) ! Read data
!
! least-squares fit
call lsq_fit (x, y, fit) ! least-squares fit
print '*', "the slope is ", fit(1) ! display the results
print '*', "the intercept is ", fit(2)
print '*', "the error is ", fit(3)
deallocate (y, x)
contains

Fig. 4.21, A Typical Least Squares Linear Fit Program (continued)

wish to sort to alphabetical order. In the first pass, the algorithm begins by examining the first two list elements (b, e). Since they are in order, these two are left alone. The next two elements (e, a) are not in order; these two elements of the list are interchanged. In this way, we “bubble” the element a toward the top and e toward the bottom. The algorithm proceeds through the list, interchanging elements if need be until the last element is reached. Note that the bottom of the list at the end of the first pass contains the correct entry. This effect occurs because of the algorithm’s structure: The “greatest” element will always propagate to the list’s end. Once through the pass, we see that the list is in better, but not perfect, order. We must perform another pass just like the first to improve the ordering. Thus, the second pass need consider only the first \( n - 1 \) elements, the third \( n - 2 \), etc. The second pass does make the list better formed. After more passes, the list eventually becomes sorted. To produce a completely sorted list, the bubble-sort algorithm requires no more passes than the number of elements in the list minus one.

The following F90 routines illustrate some of the initial features of a simple procedural approach to a simple process like the bubble-sort algorithm. We begin by considering the sorting of a list of real numbers as shown in subroutine Sort_Reals in Fig. 4.22.

In line 1 we have passed in the size of the array, and the actual array (called database). Note that the database has intent (inout) because we plan to overwrite the original database with the newly sorted order, which is done in lines 18–20. For efficiency sake we have included an integer counter, swaps_Made, so that we can determine if the sort has terminated early. If we wished to apply the same bubble-sort algorithm to an integer array all we would have to do is change the procedure name and lines 6 and 10 that describe the type of data being sorted (try it).
subroutine lsq_fit (x, y, fit)

! ------------------------------------------------------
! Linear least-squares fit, A u = c
! ------------------------------------------------------
! fit = slope, intercept, and mean squared error of fit.
! lines = the length of the arrays x and y.
! x = array containing the independent variable.
! y = array containing the dependent variable data.
implicit none
real, intent(in) :: x(:), y(size(x))
real, intent(out) :: fit(3)
integer :: lines
real :: m, b, mse
real :: sumx, sumx2, sumy, sumxy

! Summations
sumx = sum ( x ) ; sumx2 = sum ( x**2 )
sumy = sum ( y ) ; sumxy = sum ( x*y )

! Calculate slope intercept
lines = size(x)
m = (sumx*sumy - lines*sumxy)/(sumx**2 - lines*sumx2)
b = (sumy - m*sumx)/lines

! Predicted y points and the sum of squared errors.
mse = sum ( (y - m*x - b)**2 )/lines
fit(1) = m ; fit(2) = b ; fit(3) = mse ! returned
end subroutine lsq_fit

end program linear_fit

Figure 4.21: A Typical Least Squares Linear Fit Program

That is true because the compiler knows how to apply the > operator to all the standard numerical types in the language. But what if we want to sort character strings, or other types of objects? Fortran has lexical operators (like LGE) to deal with strings, but user defined objects would require that we overload the > operator, if the expected users would not find the overloading to be confusing. In other words, you could develop a fairly general sort routine if we changed lines 6 and 10 to be

```
6) type (Object), intent(inout) :: database (lines)
10) type (Object) :: temp
```

and provided an overloading of > so that line 17 makes sense for the defined Object (or for selected component of it).

To illustrate the sort of change that is necessary to sort character strings consider subroutine Sort_String Fig. 4.23:

To keep the same style as the previous algorithm and overload the > operator we would have to have a procedure that utilizes the lexical operators in lines 24 and 25, along with the interface definition on lines 12 through 17, do define the meaning of > in the context of a string. While the concept of a “template” for a code to carry out a bubble-sort on any list of objects it may not always be obvious what > means when it is overloaded by you or some other programmer.

Note that in the two above sorting examples we have assumed that we had the authority to change the original database, and that it was efficient to do so. Often that is not the case. Imagine the case where the database represents millions of credit card users, each with a large number components of numbers,
Figure 4.22: Example passes of the bubble-sort algorithm through data.

```fortran
subroutine Sort_Reals (lines, database)
! Bubble Sort of (changed) Real Database
implicit none
integer, intent(in) :: lines ! number of records
real, intent(inout) :: database (lines) ! records in database
integer :: swaps_Made ! number of swaps made in one pass
integer :: count ! loop variable
real :: temp ! temporary holder for making swap
do ! Repeat this loop forever... (until we break out of it)
    swaps_Made = 0 ! Initially, we've made no swaps
    do count = 1, (lines - 1)
        ! Make one pass of the bubble sort algorithm
        if ( database (count) > database (count + 1) ) then
            temp = database (count)
            database (count) = database (count + 1)
            database (count + 1) = temp
            swaps_Made = swaps_Made + 1
        end if
    end do
    if ( swaps_Made == 0 ) exit ! do count swaps
end do
end subroutine Sort_Reals
```

Figure 4.23: Bubble Sort of a Real Array

character strings, or general objects. If many workers are accessing those data for various sorting needs you probably would not allow the original dataset to be changed for reasons of safety or security. Then we consider an alternative to moving around the actual database components. That is, we should consider using moving pointers to large data components, or pseudo-pointers such as an ordering array. The use of an ordering array is shown in Fig. 4.24 where subroutine Integer_Sort now includes an additional argument.

The third argument has intent (out), as shown in line 7, and is an integer array of the same length as the original database which has now been changed to intent (in) so the compiler will not allow us to change the original data. If the data are properly sorted as supplied then it should not be changed and the new order should be the same as the original sequential input. That is why line 13 initializes the return order to a sequential list. Then we slightly change the previous sort logic so that lines 19 through 23 now check what's in an ordered location, and change the order number when necessary, but never change the original data. After exiting this routine you could list the information, in sorted order, without changing the original data simply by using vector subscripts in a print statement like:

```fortran
print *, database (order).
```

©2001 J.E. Akin
subroutine Sort_String (lines, database)
! Bubble Sort of (Changed) String Database
implicit none
integer, intent(in) :: lines ! input size
character(len=*) , intent(inout) :: database (lines) ! records
character (len = len(database (1))) :: temp ! swap holder
integer :: swaps_Made ! number of swaps in a pass
integer :: count ! loop variable
interface ! to lower
function to_lower (string) result (new_String)
  character(len = *), intent(in) :: string
  character (len = len(string)) :: new_String
end function to_lower
end interface ! to lower

do ! Repeat this loop forever... (until we break out of it)
  swaps_Made = 0 ! Initially, we've made no swaps
  ! Make one pass of the bubble sort algorithm
  do count = 1, (lines - 1)
    ! If the element is greater than the one after it, swap them
    if ( LGT (to_lower (database (count )) ,
            to_lower (database (count + 1))) ) then
      temp = database (count )
      database (count ) = database (count + 1)
      database (count + 1) = temp
      swaps_Made = swaps_Made + 1
    end if
  end do
  ! If we made no swaps, break out of the loop.
  if ( swaps_Made == 0) exit ! do count swaps
end do
end subroutine Sort_String

Figure 4.24: Bubble Sort of an Array of Character Strings

subroutine Integer_Sort (lines, database, order)
! Ordered Bubble Sort of (Unchanged) Integer Database
implicit none
integer, intent(in) :: lines ! number of records
integer, intent(in) :: database (lines) ! records in database
integer, intent(out) :: order (lines) ! the order array
integer :: swaps_Made ! number of swaps made in one pass
integer :: count ! loop variable
integer :: temp ! temporary holder for making swap
order = (/ (count, count = 1, lines) /) ! default order
do ! Repeat this loop forever... (until we break out of it)
  swaps_Made = 0 ! Initially, we've made no swaps
  ! Make one pass of the bubble sort algorithm
  do count = 1, (lines - 1)
    ! If item is greater than the one after it, swap them
    if ( database (order (count)) > database (order (count + 1))) then
      temp = database (count )
      database (count ) = database (count + 1)
      database (count + 1) = temp
      swaps_Made = swaps_Made + 1
    end if
  end do
  ! If we made no swaps, break out of the loop.
  if ( swaps_Made == 0) exit ! do count swaps
end do
end subroutine Integer_Sort

Figure 4.25: An Ordered Bubble Sort of an Integer Array

Clearly you could write a very similar program using a true “pointer” array since they are now standard in Fortran.

Next we will start to generalize the idea of sorting to include the sorting of objects that may have numerous components. Assume the each record object to be read is defined as in Fig. 4.25.

There may be thousands, or millions, of such records to be read from a file, sorted by name and/or number, and then displayed in sorted order. Program test_bubble, in Fig. 4.26 illustrates one approach to such a problem. Here since the database of records are to read from a file we do not yet know how many
module record_module
!
! record_module holds the "record" type
!
! record is a data structure with two names and an id number.
type record
  character (len=24) :: last_name ! last name
  character (len=24) :: first_name ! first name
  integer :: id ! id number
end type record
end module record_module

Figure 4.26: A Typical Record in a List to be Sorted

program test_bubble
!
! test_bubble asks for a filename for a file of names and id numbers, loads in the data from a file into the database, finds sorting orders, and prints sorted data.
!
! We define the database as an allocatable array of records.
type (record), allocatable :: database (:)
!
! These arrays hold the sorted order of the database entries.
type (record), allocatable :: sort_by_name (:)
type (record), allocatable :: sort_by_number (:)
!
character (len = 64) :: file_name ! file to read data from
integer, allocatable :: lines ! number of lines of input
integer, allocatable :: file_number ! the input file number
integer, allocatable :: loop_count ! loop counter
!
file_number = 1 ! arbitrarily set file_number to 1
write (*,*) 'Enter the filename to read data from:'
read (*,'(A64)') file_name
!
! Open our file and assign the number to 'file_number'
open (unit = file_number, file = file_name)
!
! Find the number of lines in the input file with input_count.
lines = input_count (file_number)
write (*,'(a, i)') 'There are ', lines, ' records.'
!
! Allocate that many entries in the database and order arrays.
allocate ( database (lines) )
allocate ( sort_by_name (lines), sort_by_number (lines) )
!
! Read the data from file into the database and close the file.
call read_data (file_number, lines, database)
close (file_number)
!
! Sort the database by name; the order will be in sort_by_name.
call string_sort (lines, database (:)%last_name, sort_by_name)
write (*,*) ; write (*,*) 'Data sorted by name: '; write (*,*)
!
! Print out the data in the database sorted by name
! call show_data (lines, database, sort_by_name)
write (*,*) ; write (*,*) 'Data sorted by number: '; write (*,*)
!
! Sort the database by id numbers; new order is sort_by_number.
call integer_sort (lines, database (:)%id, sort_by_number)
!
! Print out the data in the database sorted by number.
call show_data (lines, database, sort_by_number)
end program test_bubble

Figure 4.27: Testing of Ordered Bubble Sorts

there are to be stored. Therefore, it is declared allocatable in line 13, and allocated later in line 34 after we have evaluated the file size of a file named by the user. Although not generally necessary we have selected to have an order array for names and a different one for numbers. The are sort_by_name, and sort_by_number, respectively and are treated in a similar fashion to the database memory allocation as noted in lines 13–14, and line 35.

In line 21 we have arbitrarily set a unit number to be used for the file. That is okay for a very small code, but an unnecessary and unwise practice in general. The Fortran intrinsic inquire allows one to
determine which units are inactive and we could create a function, say Get\_Next\_Unit, to select a safe unit number for our input operation. After accepting a file name we open the unit, and count the number of lines present in the file (see line 30). Had the database been on the standard input device, and not contained any non-printing control characters, we could have easily read it with the statement

\begin{verbatim}
read *, database
\end{verbatim}

However, it does contain tabs (ASCII character number 9), and is in a user defined file instead of the standard input device so line 38 invokes subroutine read\_\_Data to get the database. Of course, once the tabs and commas have been accounted for and the names and id number extracted it uses an intrinsic constructor on each line to form its database entry like:

\begin{verbatim}
database (line\_Count) = Record (last, first, id)
\end{verbatim}

After all the records have been read into the database note that line 42 extracts all the last names with the syntax

\begin{verbatim}
database (:) last\_Name
\end{verbatim}

so they are copied into subroutine String\_Sort, as its second argument, and the ordered list sort\_by\_Name) is returned to allow operations that need a last name sort. Likewise, subroutine Integer\_Sort, shown above, is used in line 50 to sort the id numbers and save the data in order list sort\_by\_Number. The ordered lists are used in show\_\_Data, in lines 46 and 53, to display the sorted information, without changing the original data.

If the supplied file, say namelist, contained data in the format of (String comma String tab Number) with the following entries:

\begin{verbatim}
[ 1] Indurain, Miguel 5623
[ 2] van der Aarden, Eric 1245
[ 3] Rominger, Tony 3411
[ 4] Sorensen, Rolf 341
[ 6] Vandiver, Frank 45
[ 7] Smith, Sally 3821
[ 8] Johnston, David 3421
[ 9] Gillis, Malcolm 3785
[12] Johnson, Alexa 5190
[14] Butera, Robert 7253
[16] Hegg, Steve 9231
[17] LeBlanc, Lucien 23
[18] Peiper, Alan 5674
[19] Smith-Jones, Nancy 9082
\end{verbatim}

The output would be:

\begin{verbatim}
[ 1] ! Enter the filename to read data from: namelist
[ 2] ! There are 19 records.
[ 3] ! Data sorted by name:
[ 4] !
[ 7] ! Indurain Miguel 5623
[ 8] ! Gillis Malcolm 3785
[ 9] ! Hegg Steve 9231
[10] ! Johnston Jonathan 7234
[12] ! Johnston David 3421
[14] ! Kruger Charlotte 2345
[16] ! Peiper Alan 5674
[17] ! Rominger Tony 3411
[18] ! Smith Sally 3821
[19] ! Smith-Jones Nancy 9082
[20] ! Sorensen Rolf 341
[21] ! van der Aarden Eric 1245
[22] ! Vandiver Frank 45
[23] ! Yates Sean 8998
[24] !
[25] !
\end{verbatim}

and

©2001 J.E. Akin

98
Data sorted by number:

LeBlanc Lucien 23
Vandiver Frank 45
Sorensen Rolf 341
d van der Aarden Eric 1245
Kruger Charlotte 2345
Armstrong Lance 2374
Rominger Tony 3411
Johnston David 3421
Gillis Malcolm 3785
Smith Sally 3821
Johnson Alexa 5190
Indurain Miguel 5623
Peiper Alan 5674
Johns William 7234
Butera Robert 7253
Yates Sean 8998
Smith-Jones Nancy 9082
Hegg Steve 9231

<table>
<thead>
<tr>
<th>Pass 1</th>
<th>Pass 2</th>
<th>Pass 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Level</strong></td>
<td><strong>Sorted</strong></td>
<td><strong>Level</strong></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>b</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>e</td>
<td>e</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>e</td>
</tr>
<tr>
<td>d</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>f</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Is___Was</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

* Is___Was (j) = k. What is position j was position k.

Figure 4.28: Sorting via an Order Vector, Array (Is___Was) → a b c d e f

4.11 Exercises

1. Frequently we need to know how many lines exist in an external file that is to be used by our program. Usually we need that information to dynamically allocate memory for the arrays that will be constructed from the file data to be read. Write a F90 program or routine that will accept a unit number as input, open that unit, loop over the lines of data in the file connected to the unit, and return the number of lines found in the file. (A external file ends when the iostat from a read is less than zero.)

2. A related problem is to read a table of data from an external file. In addition to knowing the number of lines in the file it is necessary to know the number of entities (columns) per line and to verify that all lines of the file have the same number of columns. Develop a F90 program for that purpose. (This is the sort of checking that the MATLAB load function must do before loading an array of data.)
3 Write a program that displays the current date and time and uses the module `tic_toc`, in Fig. 4.10, to display the CPU time required for a calculation.

4 Develop a companion function called `to_upper` that converts a string to all upper case letters. Test it with the above program.

5 Develop a function that will take an external file unit number and count the number of lines in the file connected to that unit. This assumes that the file has been “opened” on that unit. The interface to the function is to be:

```fortran
interface
  function inputCount(unit) result(linesOfInput)
    integer, intent(in) :: unit ! file unit number
    integer :: linesOfInput ! result
  end function inputCount
end interface
```

6 Assume the file in the previous problem contains two real values per line. Develop a subroutine that will read the file and return two vectors holding the first and second values, respectively. The interface to the subroutine is to be:

```fortran
interface
  subroutine readData (inFile, lines, x, y)
    integer, intent(in) :: inFile, lines ! file unit, size
    real, intent(out) :: x(lines), y(lines) ! data read
  end subroutine readData
end interface
```

7 Written replies to the questions given below will be required. All of the named files are provided in source form as well as being listed in the text. The cited Figure number indicates where some or all of the code is discussed in the text.

(a) **Figure 1.3** — `hello.f90`
   What is necessary to split the printing statement so that “Hello,” and “world” occur on different program lines? That is, to continue it over two lines?

(b) **Figure 4.1** — `arithmetic.f90`
   What is the meaning of the symbol `(mod)` used to get the Mod_Result?
   What is the meaning of the symbol `(**) used to get the Pow_Result?

(c) **Figure 4.3** — `array_index.f90`
   Is it good practice to use a loop index outside the loop? Why?

(d) **Figure 4.4** — `more_or_less.f90`
   What does the symbol `(>) mean here?
   What does the symbol `(==)` mean here?

(e) **Figure 4.5** — `if_else.f90`
   What does the symbol `.and.` mean here? Can its preceding and following arguments be interchanged (is it commutative)?

(f) **Figure 4.6** — `and_or_not.f90`
   What does the symbol `.not.` mean here?
   What does the symbol `.or.` mean here? Can its preceding and following arguments be interchanged (is it commutative)?

(g) **Figure 4.7** — `clip.f90`
   What does the symbol `(<=)` mean here?

(h) **Figure 4.8** — `maximum.f90`
   What are the input and output arguments for the maxint function?
The vertical motion of a projectile at any time, $t$, has a position given by $y = y_0 + V_0 t - \frac{1}{2} g t^2$, and a velocity of $V = V_0 - g t$ when upward is taken as positive, and where the initial conditions on the starting position and velocity, at $t = 0$, are $y_0$ and $V_0$, respectively. Here the gravitational acceleration term, $g$, has been taken downward. Recall that the numerical value of $g$ depends on the units employed. Use metric units with $g = 9.81 m/s^2$ for distances measured in meters and time in seconds.

Write a C++ or F90 program that will accept initial values of $y_0$ and $V_0$, and then compute and print $y$ and $V$ for each single input value of time, $t$. Print the results for $y_0 = 1.5$ meters and $V_0= 5.0 m/s$ for times $t = 0.5, 2.0, \text{ and } 4.0$ seconds.

Modify the projectile program written in Problem 2 to have it print the time, position, and velocity for times ranging from 0.0 to 2.0 seconds, in increments of 0.05 seconds. If you use a direct loop do not use real loop variables. Conclude the program by having it list the approximate maximum (positive) height reached and the time when that occurred. The initial data will be the same, but should be printed for completeness. The three columns of numbers should be neat and right justified. In that case the default print format (print * in F90) will usually not be neat and one must employ a “formatted” print or write statement.

The Greatest Common Divisor of two positive integers can be computed by at least two different approaches. There is a looping approach known as the Euclidean Algorithm which has the following pseudocode:

Rank two positive integers as max and min.

Do while min > 0

Find remainder of max divided by min.

Replace max by min.

Replace min by the remainder

End do

Display max as the greatest common divisor.

Implement this approach and test with $max = 532 = 28 \times 19 \text{ and } min = 112 = 28 \times 8$. The names of the remainder functions are given in Table 4.7.

Another approach to some algorithms is to use a “recursive” method which employs a subprogram which calls itself. This may have an advantage in clarifying the algorithm, and/or in reducing the round off error associated with the computations. For example, in computer graphics Bernstein Polynomials are often used to display curves and surfaces efficiently by using a recursive definition in calculating their value at a point.

The Greatest Common Divisor evaluation can also be stated in terms of a recursive function, say gcd, having max and min as its initial two arguments. The following pseudocode defines the function:

\[\text{gcd}(\text{max}, \text{min}) \text{ is} \]

\[\begin{align*}
\text{a)} & \text{ max if } \text{min} = 0, \text{ otherwise} \\
\text{b)} & \text{ gcd}(\text{min}, \text{remainder of max divided by min}) \text{ if } \text{min} > 0
\end{align*}\]

Also implement this version and verify that it gives the same result as the Eulerian Algorithm. Note that F90 requires the use of the word "recursive" when defining the subprogram statement block. For example,

\begin{verbatim}
  recursive function gcd(...) result(g)
  ....
  end function gcd
\end{verbatim}
It is not uncommon for data files to be prepared with embedded tabs. Since it is a non-printing control character you can not see it in a listing. However, if you read the file expecting an integer, real, or complex variable the tab will cause a fatal read error. So one needs a tool to clean up such a file.

Write a program to read a file and output a duplicate copy, except that all tabs are replaced with a single space. One could read a complete line and check its characters, or read the file character by character. Remember that C++ and F90 have opposite defaults when advancing to a new line. That is, F90 advances to the next line, after any read or write, unless you include the format control, `advance = 'no'`, while C++ does not advance unless you include the new line control, `"<\n"`, and C does not advance unless you include the new line control, `"\n"`.

Engineering data files consisting of discrete groups of variable types often begin with a control line that lists the number of rows and columns of data, of the first variable type, that follow beginning with the next line. At the end of the data block, the format repeats: control line, variable type data block, etc. until all the variable types are read (or an error occurs where the end of file is encountered). Write a program that reads such a file which contains an integer set, a real set, and a second real set.

Neither C++ or F90 provides an inverse hyperbolic tangent function. Write such a function, called `arctanh`. Test it with three different arguments against the values given by MATLAB.

Often if one is utilizing a large number of input/output file units it may be difficult to keep up with which one you need. One approach to dealing with that problem may be to define a unit_CLASS or to create an units_MODULE to provide functionality and global access to file information. In the latter case assume that we want to provide a function to simply find a unit number that is not currently in use and utilize it for our input/output action:

```fortran
interface
  function get_next_io_unit () result (next)
    integer :: next ! the next available unit number
    end function get_next_io_unit
end interface
```

Use the Fortran INQUIRE statement to build such a utility. If you are familiar with Matlab you will see this is similar to its fopen feature.