Chapter 4

More on Intrinsic Scalar Data Types

Fortran was developed primarily for computations and as such has built-in capabilities to deal with data types common in mathematical fields as varied as complex numbers, linear algebra, optimization, and differential equations. Here we present briefly some of the other intrinsic data types we will encounter, complex numbers and character constants. We also give some details as to how computers represent integers and floating point numbers.

4.1 Complex

Complex number computations requires special handling for the real and imaginary parts as shown in the calculation below:

\[
(1 + 2i) + (3 + 4i)(5 + 6i) = 1 + 2i + 15 + 18i + 20i + 24i^2 = (1 + 15 - 24) + (2 + 18 + 20)i = -8 + 40i
\]

First the cross products are found, and then remembering that \(i^2 = -1\) we collect the real and imaginary components separately. We could code this logic using REAL data types only but it makes for complicated code. Fortran being a programming language for numerical computations has built-in support for complex arithmetic using the complex versions of the algebraic operators defined on REALs. This is much more convenient code-wise. The following code illustrates how complex numbers are declared, assigned values and manipulated:

```fortran
program cmplx
complex :: a=(1.0,2.0), b=(3.0,4.0), c=(5,6), d
d = a + b*c
```

25
print *,d
print *, 'The real and imaginary parts are=',real(d), aimag(d)
stop
end program cmplx

Fortran stores complex numbers as pairs of REALs, and/or pairs of DOUBLE-PRECISIONs. The real and imaginary parts of a complex number can be extracted using intrinsic functions real and aimag.

4.2 Character

FORTRAN provides the intrinsic data type character to represent alphabetical characters, and strings of characters. We will give a brief overview of this data type as it is not central to numerical computing. A single character constant occupies one byte of memory and is declared as followed:

```
character :: letter='a'
character(len=4) :: letters='abcd'
character(len=21) :: hello='Hello World!'
```

The variable letter is declared as a character and is assigned the single letter a. To store multiple characters, like a string, a length must be assigned to the character variables, essentially the number of bytes allocated for that variable, as shown in the letters and hello declarations. In the first one letters is given enough space to have 4 characters within it whereas hello has 21. Notice that the string Hello World! has only 12 characters; the remaining characters 13–21 are left blank. Substrings of characters can be referenced using a colon notation:

```
print *,letters(1:2) ! will print ab
print *,letters(3:4) ! will print cd
```

It is an error to reference a character beyong the limit defined in the variable declaration. The only operation permitted on characters is concatenation and uses the double slash symbol //:

```
character(len=4) :: first='Jim ', last='Kirk'
character(len=8) :: fullname
fullname = first//last
print *,fullname ! would print "Jim Kirk"
```

4.3 Computer Representation of Integers

Computers store and manipulate numbers in binary form (using 0s and 1s). The positional notation is used to build the value of a given number, whereby the
4.4 COMPUTER REPRESENTATION OF REALS

Figure 4.1: Binary representation of the number 19 using integer*4 data types.

different numerals are the coefficient of powers of 2, instead of powers of 10 like in the decimal system. For example, the number 19 can be represented as:

\[
\text{decimal system} \quad 19_{10} = 1 \times 10^1 + 9 \times 10^0 \\
\text{binary system} \quad 10011_2 = 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \quad (4.4)
\]

The symbols in the binary system are called \textbf{bits} just like symbols in the decimal system are called digits.

Variables and constants of the type we have been referring to as real are stored in memory as a contiguous sequence of 32 bits called a \textbf{word} or \textbf{full word}. The bits of a word are grouped in 4 8-bit \textbf{bytes}. Because a fullword integer is 4 bytes long, FORTRAN names that data type \texttt{INTEGER*4}; so what we have been casually referring to as type integers, is more formally \texttt{integer*4}. This is in fact the default integer type in FORTRAN. An immediate consequence of the finite number of bits used is that arbitrary large integers cannot be represented on a computer. What is the largest integer that can be stored requires a bit more details.

Let us focus on the \texttt{integer*4} as shown in figure 4.3. The left most bit marked \textit{s} is the sign bit: it is 0 for positive integers and 1 for negative integers. If the number is positive the right-most 31 bits represent the binary value of the integer. If the number is negative then all bits, including the sign bit of 1, represent the 2’s complement of \(-N\), which is \(2^{32} - N\) and hence is positive. The largest positive value that can be stored in an \texttt{integer*4} is evidently

\[
2^{30} + 2^{29} + 2^{28} + \cdots + 2^1 + 2^0 = 2^{31} - 1 = 2147483647 \quad (4.5)
\]

Larger values cannot fit within an \texttt{integer*4}. If that happens we then speak of a \textbf{fixed-point overflow}. The results of calculations after an overflow cannot be trusted.

To represent larger integers FORTRAN has the \texttt{integer*8} data type which extends the range of possible integers.

4.4 Computer Representation of Reals

A real number has a fractional part (which could be zero). Different computers manufacturers had their own format to represent \textbf{floating point numbers}. Here we will present the one based on the IEEE standard. FORTRAN default reals are composed of 4 bytes and hence 32 bits. The interpretation of the bits is however
different from the integers. The left most bit still denotes the sign. The next 8
bits are the binary value of a biased exponent \( p \) and the last 23 bits are a binary
fraction \( f \). The bit pattern represented by the bit pattern is:

\[
r = (-1)^s \times 2^{p-127} \times (1 + f)
\]

The 127 in the 2 exponent is the exponent bias. The 8-bit exponent can represent
values of \( p \) from 0 to 255, and hence the exponents range from -127 to 128. Biasing
the exponent permits to represent number with positive and negative exponents.
The fractional value is also based on a positional notation but the exponent are
now negatives. Thus

\[
f = .100011_2 \\
= 2^{-1} + 2^{-5} + 2^{-6} = \frac{1}{2} + \frac{1}{32} + \frac{1}{64}
\]

To add real values that have different exponents we would move (or float) the
exponent in the smaller number to the left until the exponents were the same
before adding the fractions. For this reason reals are also referred to as floating
point numbers. A floating point number is said to be normalized when the left
most bit of its fraction \( f \) is 1. Otherwise it is called unnormalized.

The most important facts about floating numbers is that they cannot represent
the continuum of the mathematical real set \( R \). Some fraction numbers can be
represented exactly, like 1.5, but 3.1 can only be approximated since its fractional
part repeats indefinitely. A fortran real*4 data type has 6 or 7 significant digits.
Like integers, reals also have a largest representable value, which when exceeded
results in a floating point overflow. However, in this case the standard calls
for storing a special value to represent \( \pm \infty \). There is another special bit pattern
called a NaN (stands for Not A Number) to warn users against illegal floating
point operations, such as 0.0/0.0 and sqrt(-1). It is quite common for floating
point operations to underflow producing a nonzero value smaller then can be
represented using the data type. In most computations replacing this underflow
with 0 is usually harmless.

Large or small reals are cumbersome to write solely in decimal notation, and
so an exponential notation is used. Here are a few examples:

\[
\begin{align*}
\text{real, parameter} & : \text{avogadro} = 6.02e23 & 6.02 \times 10^{23} \\
\text{real, parameter} & : \text{millimeter}=1.0e-3 & 1.0 \times 10^{-3}
\end{align*}
\]

To extend the range of reals, FORTRAN has two more data type: double precision
and quadruple precision, the latter is latter is only optionally implemented
in some fortran compiler. The details of these new data types are shown in table
4.1. To indicate that a constant must have a specific data type the following format
must be used:

\[
\text{double precision :: asingle, adouble, avod}
\]
**4.5. KINDS**

It is often desirable in a computer code to change the precision on the fly. F90 has a nifty feature that allows it to codify the data type of variables and constants. If the programmer is disciplined this provides an easy way to make selection of machine precision portable.

Each intrinsic data type, except for characters, has a parameter called kind parameter associated with it. A kind parameter is intended to identify a machine representation for a particular data type. The kind parameters are integers and their value can vary from compiler to compiler. Kind parameters 1, 2, and 3 might indicate single, double and quad precision on one compiler; while 4, 8 and 16 might be used on another. One must check the compiler manual to find out the right values. There are at least two kinds of real and complex data types, and at least one kind of integer and logical types. Note that kind parameters are optional and one can always use the default data types if they are deemed adequate for the task at hand.

Code 4.2 is a simple example on how to use kinds to select the required data type. Lines 3 to 5 invoke the built-in intrinsic function `select_real_kind` to define the kind tag for a variable that can hold a minimum of 6 digits of precision (single on line 3), 13 digits (double on line 4), and 30 digits (quadrapule on line 5). Lines 6–7 define single, double, and quadrapule precision versions of the variable $\pi$ which are computed through lines 10-12. Notice that the constants 2 and 1 on each of these lines were tagged with the appropriate kind precision to force the compiler
program print_kinds
  implicit none
  integer, parameter :: s = selected_real_kind(6)
  integer, parameter :: d = selected_real_kind(13)
  integer, parameter :: q = selected_real_kind(30)
  real(s) :: spi
  real(d) :: dpi
  real(q) :: qpi
  spi = 2._s*asin(1._s)
  dpi = 2._d*asin(1._d)
  qpi = 2._q*asin(1._q)
  print *, 'single precision pi= ', spi
  print *, 'double precision pi= ', dpi
  print *, 'quadrapule precision pi= ', qpi
  stop
end program print_kinds

Figure 4.2: Example of kind-use to define compiler independent data types to use the appropriate form. The asine function return the inverse sine of its argument, and the returned value has the same data-type as the input value. This is an example of an overloaded function where the same name is used to refer to three-different functions: a single, double, and quadrapule versions. The compiler calls the appropriate one based on the type of input argument. The output of the program is:

$ kindtest
  single  precision pi= 3.141593
  double  precision pi= 3.14159265358979
  quadrapule precision pi= 3.14159265358979323846264338327950