University of Cambridge
Department of Physics

Computational Physics

Self-study guide 2

Programming in Fortran 95

Dr. Rachael Padman

Michaelmas 2007

Acknowledgements:

This handout was originally prepared by Dr. Paul Alexander, and has been updated and maintained by Dr Peter Haynes of the TCM group.

1. The Basics

In this section we will look at the basics of what a program is and how to make the program run or *execute*.

The non-trivial example programs can be found in the directory:

```
      $PHYTEACH/part_2/examples
```

with the name of the file the same as that of the program discussed in this guide.

Some sections are more advanced and are indicated clearly indicated by a thick black line to the right of the text. These can be skipped certainly on a first reading and indeed you will be able to tackle the problems without using the material they discuss.

1.1 A very simple program

A program is a set of instructions to the computer to perform a series of operations. Those operations will often be mathematical calculations, decisions based on equalities and inequalities, or special instructions to say write output to the screen. The program consists of "source code" which is "stored" in a text file. This code contains the instructions in a highly structured form. Each computer language has a different set of rules (or *syntax*) for specifying these operations. Here we will only consider the Fortran 90/95 (F95 for short) programming language and syntax.

- Using emacs enter the following text into a file called ex1.f90, the .f90 part of the file name is the extension indicating that this is program source code written in the Fortran 90/95 language

```
program ex1

   !
   ! My first program
   !
   write(*,*) 'Hello there'

end program ex1
```

This is a complete F95 program.

The first and last lines introduce the start of the program and show where it ends. Between the first and last lines are the program "statements". The lines beginning with an exclamation mark are special statements called comments. They are not instructions to the computer, but instead are there to enable us (the programmer) to improve the readability of the program and help explain what the program is doing.

The line beginning write is a statement giving a specific instruction to print to the screen.

Note that except within quotes:

⇒ Upper and lower case are NOT significant

(different from Unix commands and files)

⇒ Blank lines and spaces are not significant.

1.2 Running the program

Before we can run the program we must get the computer to convert this symbolic language (F95) into instructions it can understand directly. This process is called "*compilation*". At the same time the computer will check our program source for errors in the syntax, but not for errors in our logic! In general programs will be assembled from source in many files; bringing all of these instructions together is called "*linking*". We perform both of these tasks using the Unix command f95.

- Type the following, the -o is an option saying where to place the output which in this case is a program which is ready to run, we call this an *executable*. (The default executable name is a.out).

```
f95 -o ex1 ex1.f90
```

  - ➢ If you haven't made any typing errors there should be no output to the screen from this command, but the file ex1 should have been created. By convention executable programs under Unix do not normally have a file extension (i.e. no ".xxx" in the file name).

- To run the program type:

```
./ex1
```

  - ➢ Most Unix commands are files which are executed. The shell has a list of directories to search for such files, but for security reasons this list does not contain the current directory. The './' (dot slash) before ex1 tells the shell explicitly to look in the current directory for this file.
  - ➢ The output should be the words " Hello there".

- What happens if you make an error in the program? To see this let's make a deliberate error. Modify the line beginning write to read:

```
write(*,*) 'Hello there' 'OK'
```

  - ➢ Save the file, and compile again :

```
f95 -o ex1 ex1.f90
```

  - ➢ This time you get errors indicating that the syntax was wrong; i.e. you have not followed the rules of the F95 language! Correct the error by changing the source back to the original, recompile and make sure the program is working again.

1.3 Variables and expressions

The most important concept in a program is the concept of a variable. Variables in a program are much like variables in an algebraic expression, we can use them to hold values and write mathematical expressions using them. As we will see later F95 allows us to have variables of different types, but for now we will consider only variables of type real. Variables should be declared before they are used at the start of the program. Let us use another example to illustrate the use of variables.

- Enter the following program and save it to the file ex2.f90

```
program vertical
  !
  ! Vertical motion under gravity
  !
  real :: g          ! acceleration due to gravity
  real :: s          ! displacement
  real :: t          ! time
  real :: u          ! initial speed ( m / s)

  ! set values of variables
  g = 9.8
  t = 6.0
  u = 60

  ! calculate displacement
  s = u * t - g * (t**2) / 2

  ! output results
  write(*,*) 'Time = ',t,'   Displacement = ',s

end program vertical
```

- Compile and run the program and check the output is what you expect

```
f95 -o ex2 ex2.f90
./ex2
```

This program uses four variables and has many more statements than our first example. The variables are “declared” at the start of the program before any executable statements by the four lines:

```
real :: g          ! acceleration due to gravity
real :: s          ! displacement
real :: t          ! time
real :: u          ! initial speed ( m / s)
```

After the declarations come the executable statements. Each statement is acted upon sequentially by the computer. Note how values are assigned to three of the variables and then an expression is used to calculate a value for the fourth (s).

Unlike in an algebraic expression it would be an error if, when the statement calculating the displacement was reached, the variables g, t and u had not already been assigned values.

Some other things to note:

1. Comments are used after the declarations of the variables to explain what each variable represents.

2. The '*' represents multiplication

3. The '**' is the operator meaning "raise to the power of", it is called technically *exponentiation*.

4. In this program we have used single letters to represent variables. You may (and should if it helps you to understand the program) use longer names. The variable names should start with a character (A-Z) and may contain any character (A-Z), digit (0-9), or the underscore (_) character.

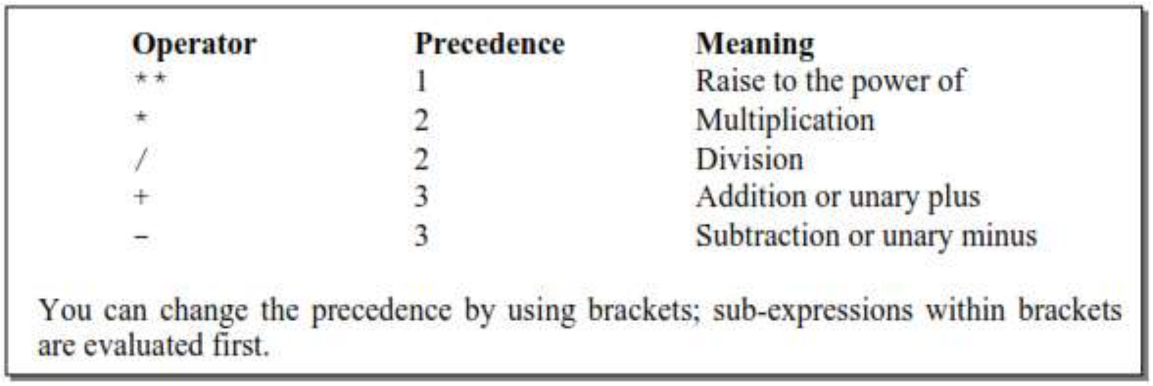
5. Upper and lower case are not distinguished. For example therefore the variables `T` and `t`, and the program names `vertical and Vertical` are identical.

The usefulness of variables is that we can change their value as the program runs.

All the standard operators are available in expressions. An important question is if we have the expression

```
g * t **2
```

what gets evaluated first? Is it g*t raised to the power of 2 or t raised to the power 2 then multiplied by g? This is resolved by assigning to each operator a precedence; the highest precedence operations are evaluated first and so on. A full table of numeric operators is (in decreasing precedence)



Operator	Precedence	Meaning
**	1	Raise to the power of
*	2	Multiplication
/	2	Division
+	3	Addition or unary plus
–	3	Subtraction or unary minus

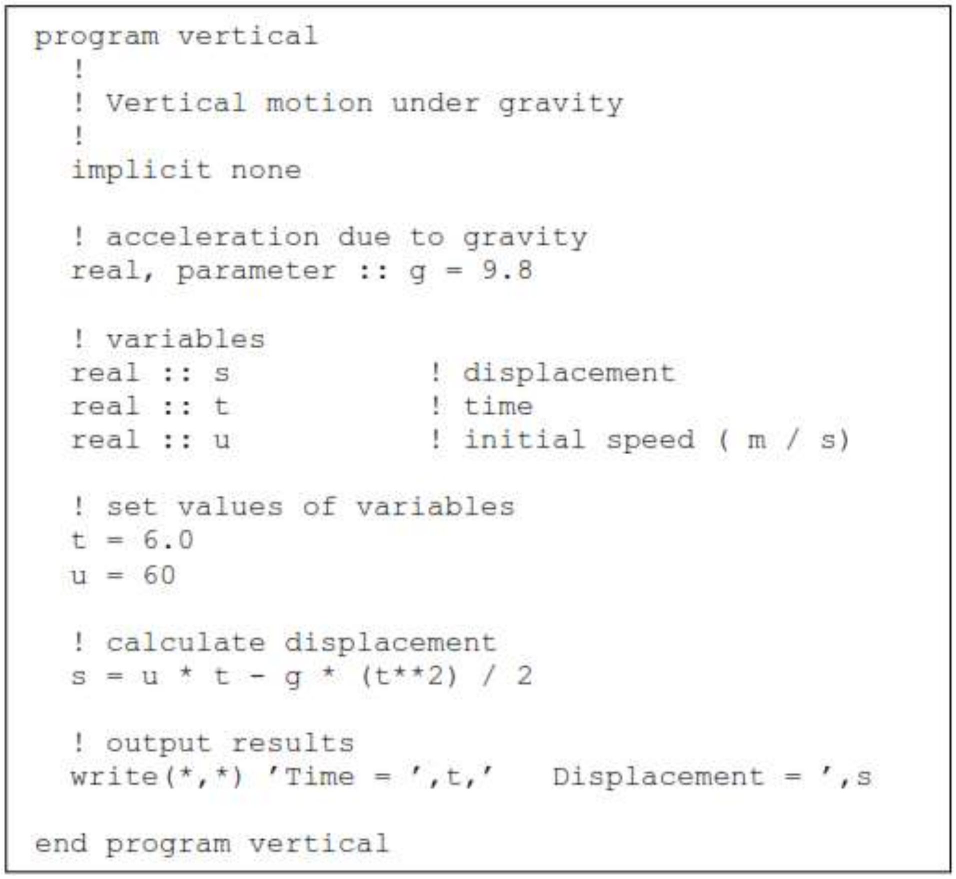
You can change the precedence by using brackets; sub-expressions within brackets are evaluated first.

Let's look at ways of improving this program. An important idea behind writing a good program is to do it in such a way so as to avoid errors that you may introduce yourself! Programming languages have ways of helping you not make mistakes. So let's identify some possible problems.

- The acceleration due to gravity is a constant, not a variable. We do not wish its value to change.

- We want to avoid using a variable which is not given a value; this could happen if we mistyped the name of a variable in one of the expressions.

Consider the following modified form of our program:



```
program vertical
  !
  ! Vertical motion under gravity
  !
  implicit none

  ! acceleration due to gravity
  real, parameter :: g = 9.8

  ! variables
  real :: s            ! displacement
  real :: t            ! time
  real :: u            ! initial speed ( m / s)

  ! set values of variables
  t = 6.0
  u = 60

  ! calculate displacement
  s = u * t - g * (t**2) / 2

  ! output results
  write(*,*) 'Time = ',t,'   Displacement = ',s

end program vertical
```

We have changed three lines and some of the comments. The line:

```
implicit none
```

is an important statement which says that all variables must be defined before use. You should always include this line in all programs. [1]

The second change is to the line:

```
real, parameter :: g = 9.8
```

This in fact defines g to be a *constant* equal to the value 9.8; an attempt to reassign g via a statement like the one in the original version (g = 9.8 on a line by itself) will now lead to an error. The syntax of this statement is as follows: After the definition of the variable type real we give a series of options separated by commas up until the '::' after which we give the variable name with an optional assignment.

1 It is an unfortunate legacy of older versions of Fortran that you could use variables without defining them, and in that case Fortran supplied rules to determine what the variable type was.

We will meet more options later.

Try out these new ideas:

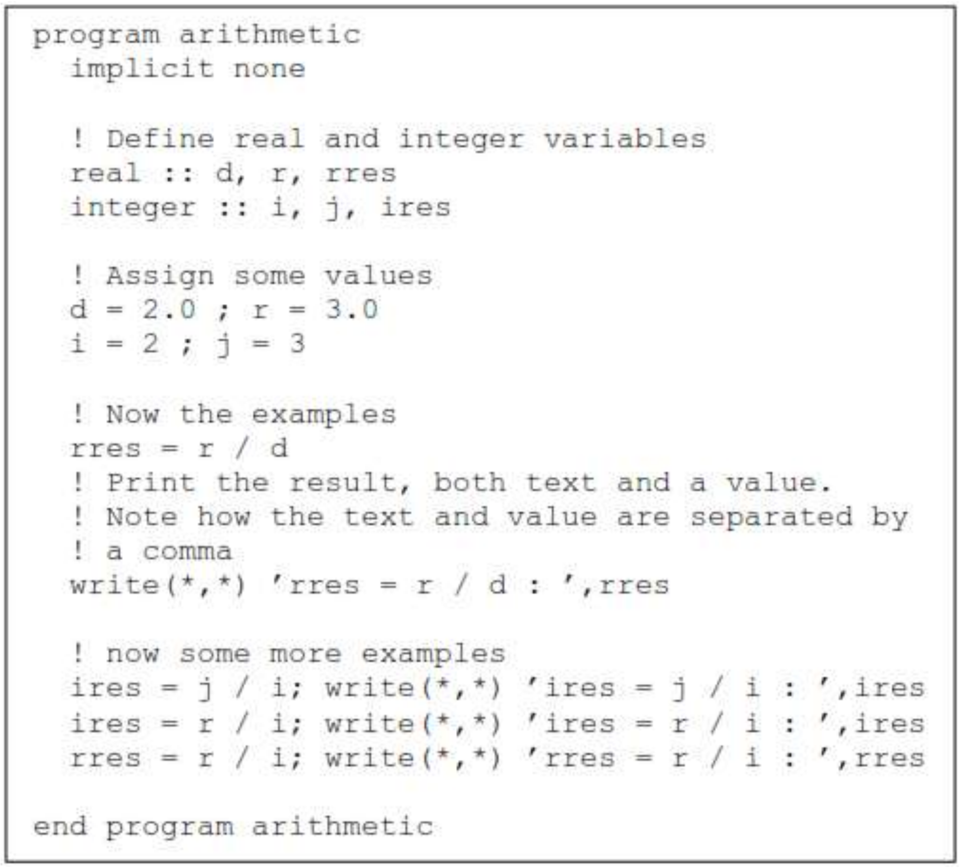
- Make these changes and make sure the program compiles.
- Now make some deliberate errors and see what happens. Firstly add back in the line g = 9.8 but retain the line containing the parameter statement.
- Compile and observe the error message.
- Now change one of the variables in the expression calculating s, say change u to v. Again try compiling.
- Fix the program.

1.4 Other variable types: integer, complex and character

As we have hinted at, there are other sorts of variables as well as real variables. Important other types are integer, complex and character.

Let’s first consider integer variables; such variables can only hold integer values. This is important (and very useful) when we perform calculations. It is also worth pointing out now that F95 also distinguishes the type of values you include in your program. For example a values of ‘3.0’ is a real value, whereas a value of ‘3’ without the ‘.0’ is an integer value. Some examples will illustrate this.

Enter the following program:



```
program arithmetic
  implicit none

  ! Define real and integer variables
  real :: d, r, rres
  integer :: i, j, ires

  ! Assign some values
  d = 2.0 ; r = 3.0
  i = 2 ; j = 3

  ! Now the examples
  rres = r / d
  ! Print the result, both text and a value.
  ! Note how the text and value are separated by
  ! a comma
  write(*,*) ’rres = r / d : ’,rres

  ! now some more examples
  ires = j / i; write(*,*) ’ires = j / i : ’,ires
  ires = r / i; write(*,*) ’ires = r / i : ’,ires
  rres = r / i; write(*,*) ’rres = r / i : ’,rres

end program arithmetic
```

First some things to note about the program:

1. We can declare more than one variable of the same type at a time by separating the variable names with commas:

```
real :: d, r, rres
```

2. We can place more than one statement on a line if we separate them with a semicolon:

```
d = 2.0 ; r = 3.0
```

- Compile and run the program. Note the different output. The rule is that for integer division the result is truncated towards zero. Note that the same rules apply to expressions containing a constant. Hence:

```
ires = 10.0 / 3      !  value of ires is 3
rres = 10 / 3        !  value of rres is 3.0
rres = 10.0 / 3.0    !  value of rres is 3.333333
```

- Make sure you are happy with these rules; alter the program and try other types of expression.

Some expressions look a little odd at first. Consider the following expression:

```
n = n + 1
```

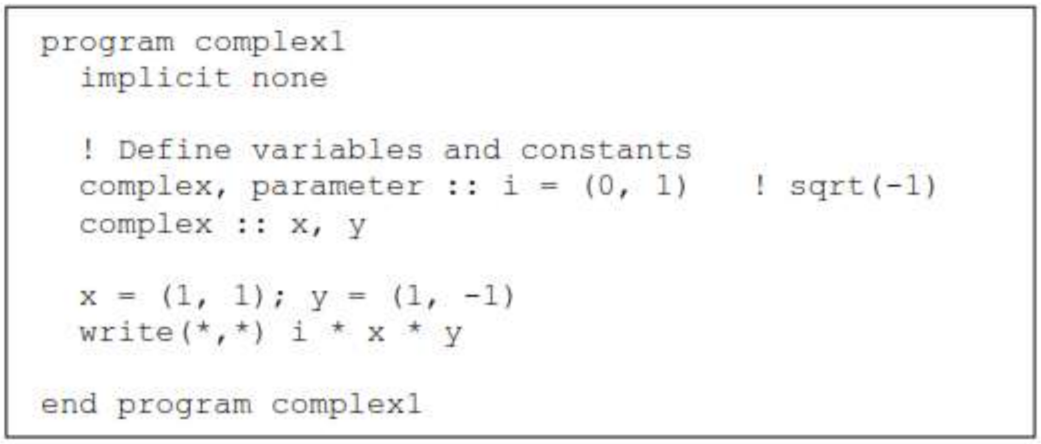
where n is an integer. The equivalent algebraic expression is meaningless, but in a program this is a perfectly sensible expression. We should interpret as:

"Evaluate the right hand side of the expression and set the variable on the left hand side to the value evaluated for the right hand side".

The effect of the above expression is therefore to increment the value of n by 1. Note the role played by the '=' sign here: it should be thought of not as an equality but instead as an "assignment".

The complex type represents complex numbers. You can do all the basic numerical expressions discussed above with complex numbers and mix complex and other data types in the same expression. The following program illustrates their use.

- Enter the program, compile and run it. Make sure you understand the output.

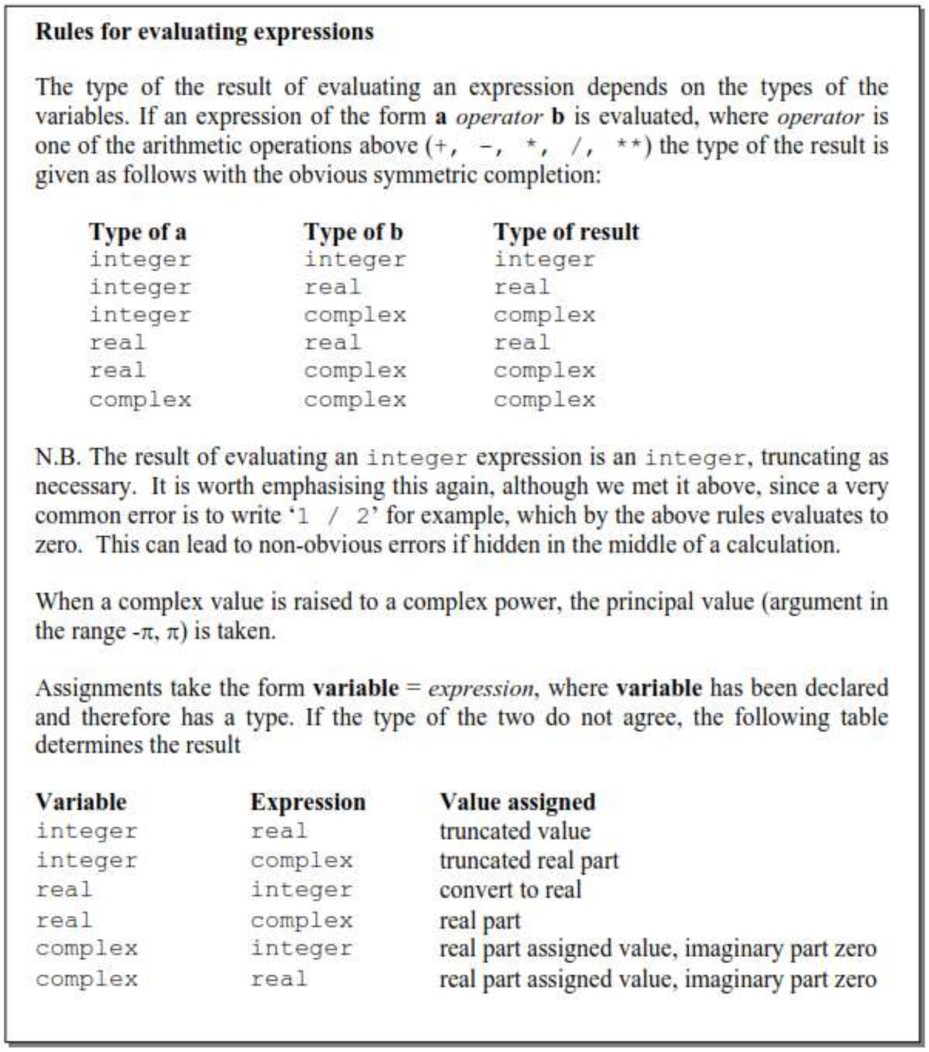


```fortran
program complex1
  implicit none

  ! Define variables and constants
  complex, parameter :: i = (0, 1)   ! sqrt(-1)
  complex :: x, y

  x = (1, 1); y = (1, -1)
  write(*,*) i * x * y

end program complex1
```

The character data type is used to store strings of characters. To hold a string of characters we need to know how many characters in the string. The form of the definition of characters is as follows:

```fortran
character (len = 10) :: word
! word can hold 10 characters
```

We will meet character variables again later.



Rules for evaluating expressions

The type of the result of evaluating an expression depends on the types of the variables. If an expression of the form **a** *operator* **b** is evaluated, where *operator* is one of the arithmetic operations above (`+, -, *, /, **`) the type of the result is given as follows with the obvious symmetric completion:

Type of a	Type of b	Type of result
`integer`	`integer`	`integer`
`integer`	`real`	`real`
`integer`	`complex`	`complex`
`real`	`real`	`real`
`real`	`complex`	`complex`
`complex`	`complex`	`complex`

N.B. The result of evaluating an `integer` expression is an `integer`, truncating as necessary. It is worth emphasising this again, although we met it above, since a very common error is to write '`1 / 2`' for example, which by the above rules evaluates to zero. This can lead to non-obvious errors if hidden in the middle of a calculation.

When a complex value is raised to a complex power, the principal value (argument in the range $-\pi$, π) is taken.

Assignments take the form **variable** = *expression*, where **variable** has been declared and therefore has a type. If the type of the two do not agree, the following table determines the result

Variable	Expression	Value assigned
`integer`	`real`	truncated value
`integer`	`complex`	truncated real part
`real`	`integer`	convert to real
`real`	`complex`	real part
`complex`	`integer`	real part assigned value, imaginary part zero
`complex`	`real`	real part assigned value, imaginary part zero

1.5 Intrinsic functions

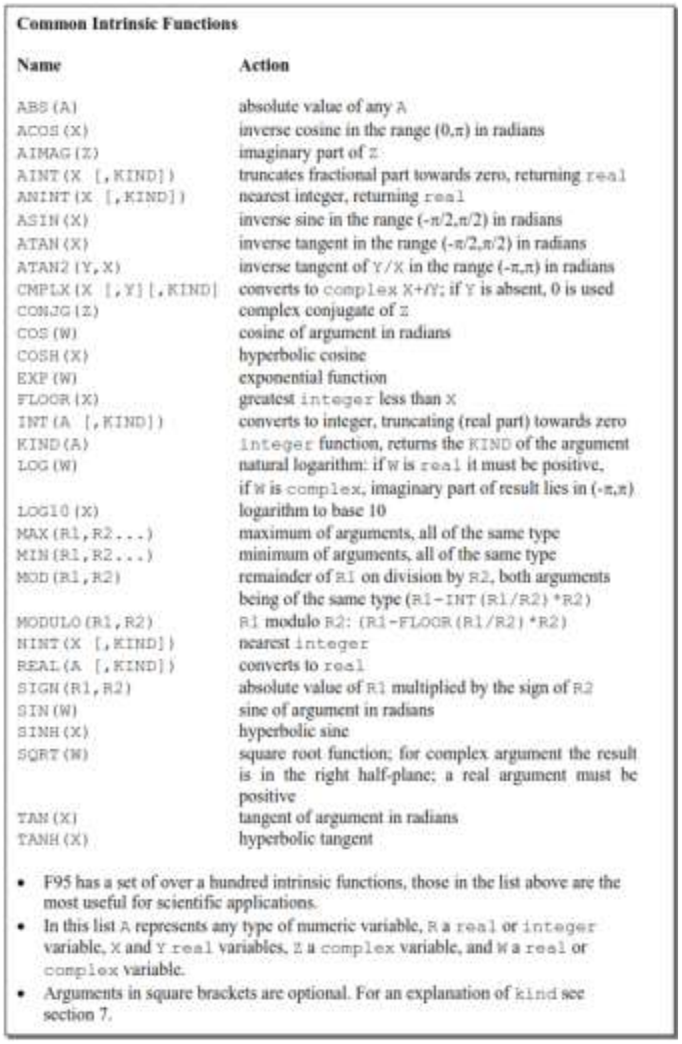
So far we have seen how to perform simple arithmetic expressions on variables. Real problems will involve more complicated mathematical expressions. As we shall see later, F95 enables you to define your own functions which return values. However, some functions are so common and important that they are provided for us as part of the language; these are called *intrinsic* functions.

Let us consider a program to compute projectile motion. The program computes the horizontal, *x*, and vertical, *y*, position of the projectile after a time, *t*: 2

$$x = u\,t\cos a \qquad\qquad y = u\,t\sin a - g\,t^2/2$$

```
program projectile
  implicit none

  ! define constants
  real, parameter :: g = 9.8
  real, parameter :: pi = 3.1415927

  real :: a, t, u, x, y
  real :: theta, v, vx, vy

  ! Read values for a, t, and u from terminal
  read(*,*) a, t, u

  ! convert angle to radians
  a = a * pi / 180.0

  x = u * cos(a) * t
  y = u * sin(a) * t - 0.5 * g * t * t

  vx = u * cos(a)
  vy = u * sin(a) - g * t
  v = sqrt(vx * vx + vy * vy)
  theta = atan(vy / vx) * 180.0 / pi

  write(*,*) 'x: ',x,'  y: ',y
  write(*,*) 'v: ',v,'  theta: ',theta

end program projectile
```

- Compile and run the program. It will wait. The statement "read(*,*)…" is requesting input from you. Enter three values for a, t and u. You should now get some output.
- Examine this program carefully and make sure you understand how it works.
- Note especially how we use the functions cos, sin, atan and sqrt much as you would use them in algebraic expressions. As always upper and lower case are equivalent.



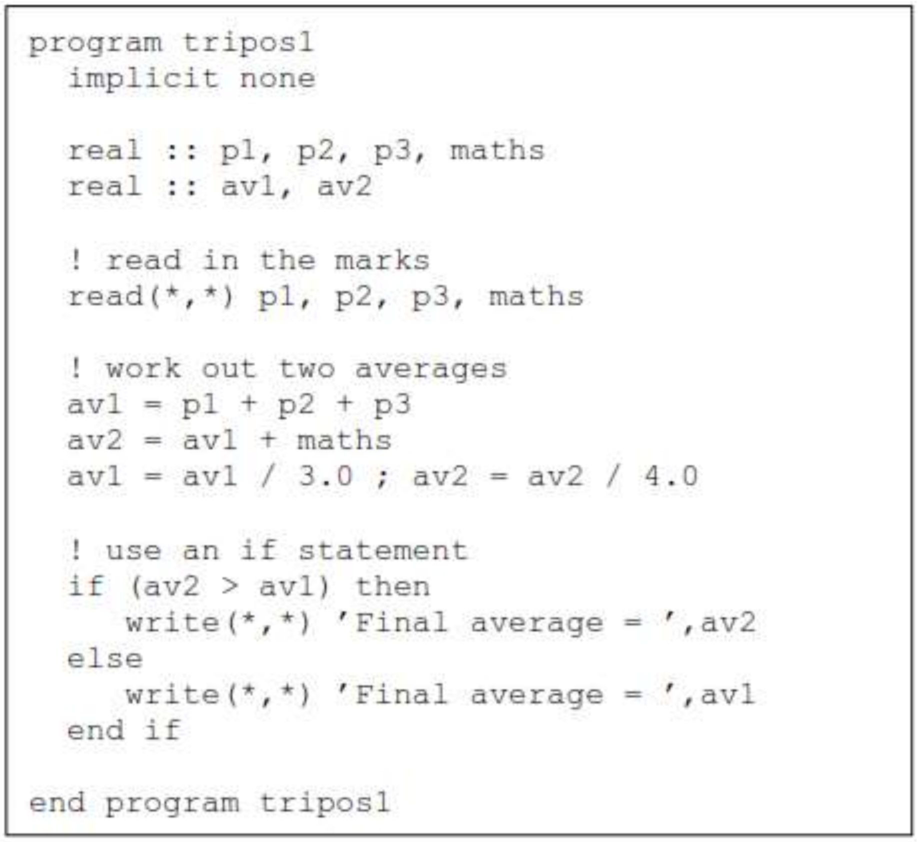
Common Intrinsic Functions

Name	Action
`ABS(A)`	absolute value of any `A`
`ACOS(X)`	inverse cosine in the range (0,π) in radians
`AIMAG(Z)`	imaginary part of `Z`
`AINT(X [,KIND])`	truncates fractional part towards zero, returning `real`
`ANINT(X [,KIND])`	nearest integer, returning `real`
`ASIN(X)`	inverse sine in the range (-π/2,π/2) in radians
`ATAN(X)`	inverse tangent in the range (-π/2,π/2) in radians
`ATAN2(Y,X)`	inverse tangent of `Y/X` in the range (-π,π) in radians
`CMPLX(X [,Y][,KIND]`	converts to `complex` `X`+*i*`Y`; if `Y` is absent, 0 is used
`CONJG(Z)`	complex conjugate of `Z`
`COS(W)`	cosine of argument in radians
`COSH(X)`	hyperbolic cosine
`EXP(W)`	exponential function
`FLOOR(X)`	greatest `integer` less than `X`
`INT(A [,KIND])`	converts to integer, truncating (real part) towards zero
`KIND(A)`	`integer` function, returns the `KIND` of the argument
`LOG(W)`	natural logarithm: if `W` is `real` it must be positive, if `W` is `complex`, imaginary part of result lies in (-π,π)
`LOG10(X)`	logarithm to base 10
`MAX(R1,R2...)`	maximum of arguments, all of the same type
`MIN(R1,R2...)`	minimum of arguments, all of the same type
`MOD(R1,R2)`	remainder of `R1` on division by `R2`, both arguments being of the same type (`R1-INT(R1/R2)*R2`)
`MODULO(R1,R2)`	`R1` modulo `R2`: (`R1-FLOOR(R1/R2)*R2`)
`NINT(X [,KIND])`	nearest `integer`
`REAL(A [,KIND])`	converts to `real`
`SIGN(R1,R2)`	absolute value of `R1` multiplied by the sign of `R2`
`SIN(W)`	sine of argument in radians
`SINH(X)`	hyperbolic sine
`SQRT(W)`	square root function; for complex argument the result is in the right half-plane; a real argument must be positive
`TAN(X)`	tangent of argument in radians
`TANH(X)`	hyperbolic tangent

- F95 has a set of over a hundred intrinsic functions, those in the list above are the most useful for scientific applications.
- In this list `A` represents any type of numeric variable, `R` a `real` or `integer` variable, `X` and `Y` `real` variables, `Z` a `complex` variable, and `W` a `real` or `complex` variable.
- Arguments in square brackets are optional. For an explanation of `kind` see section 7.

1.6 Logical controls

So far all the programming statements we have met will simply enable us to produce efficient calculators. That is useful, but there is a lot more to programming. In this and Section 1.8 we introduce two crucial ideas. The first is the idea of taking an action conditional upon a certain criteria being met. An example will help to introduce this idea. For many years it was the case in Part IA of the Tripos that your maths mark was only included if it improved your overall result. Let us write a program to perform that simple sum. We read in four marks and output a final average.



```
program tripos1
  implicit none

  real :: p1, p2, p3, maths
  real :: av1, av2

  ! read in the marks
  read(*,*) p1, p2, p3, maths

  ! work out two averages
  av1 = p1 + p2 + p3
  av2 = av1 + maths
  av1 = av1 / 3.0 ; av2 = av2 / 4.0

  ! use an if statement
  if (av2 > av1) then
     write(*,*) 'Final average = ',av2
  else
     write(*,*) 'Final average = ',av1
  end if

end program tripos1
```

- Compile and run this program and make sure you understand how it works.
- Note how the statements are indented. We use indenting to help show the logical structure of the program; indented statements are executed depending on the output of the test done by the if statement. The indenting is not essential, but it leads to a program which is much easier to follow. If you choose this style you can indent each level by any number of spaces as you wish.

The if statement is the simplest, but most important, of a number of ways of changing what happens in a program depending on what has gone before. It has the general form:

```
if ( logical expression) action
```

As another example we can use it to check for negative values:

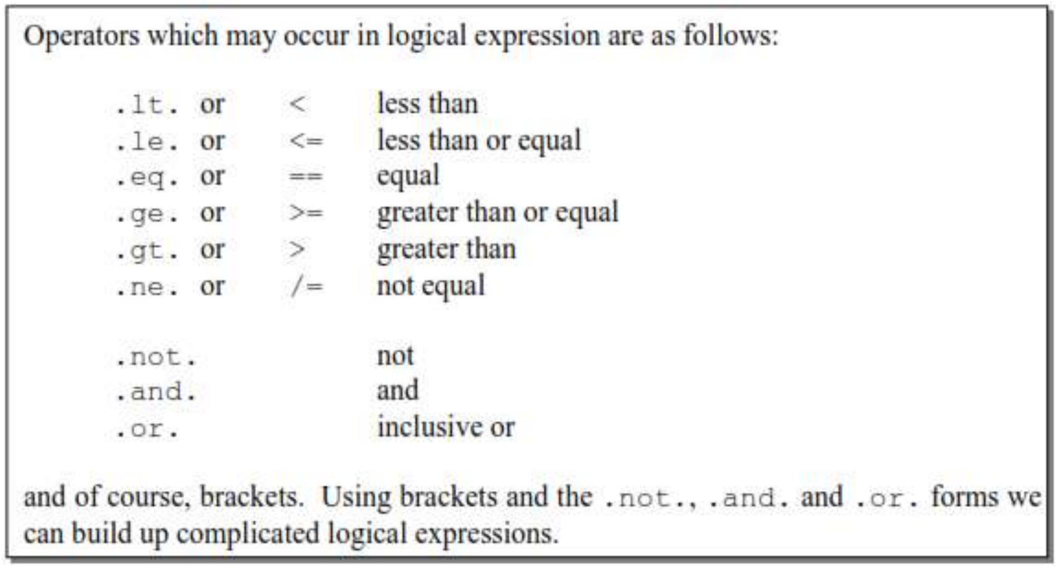
```
if (x < 0) x=0 ! replace negative x with zero
```

The if construct may also be used in more extended contexts (as above), such as:

```
if (logical expression) then
 xxx
 else
 xxx
end if
```

Here if the condition is false the statements following the else are executed. We can also include additional tests which are treated sequentially; the statements following the *first* logical test to be reached which is true are executed: if (*logical expression*) then

```
if (logical expression) then
 xxx
else if (logical expression) then
 xxx
else
  xxx
end if
```



Operators which may occur in logical expression are as follows:

`.lt.`	or	`<`	less than
`.le.`	or	`<=`	less than or equal
`.eq.`	or	`==`	equal
`.ge.`	or	`>=`	greater than or equal
`.gt.`	or	`>`	greater than
`.ne.`	or	`/=`	not equal
`.not.`			not
`.and.`			and
`.or.`			inclusive or

and of course, brackets. Using brackets and the `.not.`, `.and.` and `.or.` forms we can build up complicated logical expressions.

- As an exercise consider the following. Suppose the rules for Part IA of the Tripos were changed so that:

    1. The full maths course is always counted in the average
    2. Quantitative biology mark is only counted if it improves the average
    3. Elementary maths for biology is never counted.

- Modify the program tripos1 to compute the average mark. One further piece of information is required which is an integer code indicating the type of maths paper taken. This integer code can be assumed to take the values: Full

Full maths	1
Quantitative biology	2
Elementary maths	3

- One possible solution is available in the examples directory as tripos2.f90

if clauses may appear nested, that is one inside another. Suppose we wish to compute the expression $x = \left(b\sqrt{d}\right)/a$ which fails if $d < 0$ or a is zero. If these were entered by a user then they could (incorrectly) take on these values. A good program should check this. Here is some code to do this which illustrates nested if clauses

```
if (a /= 0.0) then
   if (d < 0.0) then
      write(*,*) 'Invalid input data d negative'
   else
      x = b * sqrt(d) / a
   end if
else
   write(*,*) 'Invalid input data a zero'
end if
```

1.7 Advanced use of if and logical comparisons

In a large program it is likely that if clauses will be nested, i.e. appear one within another. This causes us no problems, but might make it less clear which end if goes with which if. To overcome this we can name the if clauses. An example illustrates the syntax. Let's use the example we have just met:

```
outer: if (a /= 0.0) then
   inner: if (d < 0.0) then
      write(*,*) 'Invalid input data d negative'
   else inner
      x = b * sqrt(d) / a
   end if inner
else outer
   write(*,*) 'Invalid input data a zero'
end if outer
```

The names are outer and inner; note the syntax, especially the colon. Named if clauses are useful when you want to make your intention clear, but are not essential.

The logical expressions we have met in if clauses can be used more generally with a logical variable. Logical variables take on the value of .true. or .false.. Here is a simple example which illustrates their use.

```
logical :: l1, l2

l1 = x > 0.0
l2 = y /= 1.0
if (l1 .and. l2) then…
```

This program fragment could equally well have been written

```
if ((x > 0.0) .and. (y /= 1.0)) then
```

Using logical variables may make some things easier to understand.

1.8 Repeating ourselves with loops: do

Loops are the second very important concept needed in a program. If a set of instructions needs to be repeated, a loop can be used to do this repetition. As we shall see we have a lot of control over the loop and this makes them extremely powerful; this is especially true when combined with the if clauses we have just met.

The general form of the do loop is:

```
do var = start,  stop  [,step]
   xxx
end do
```

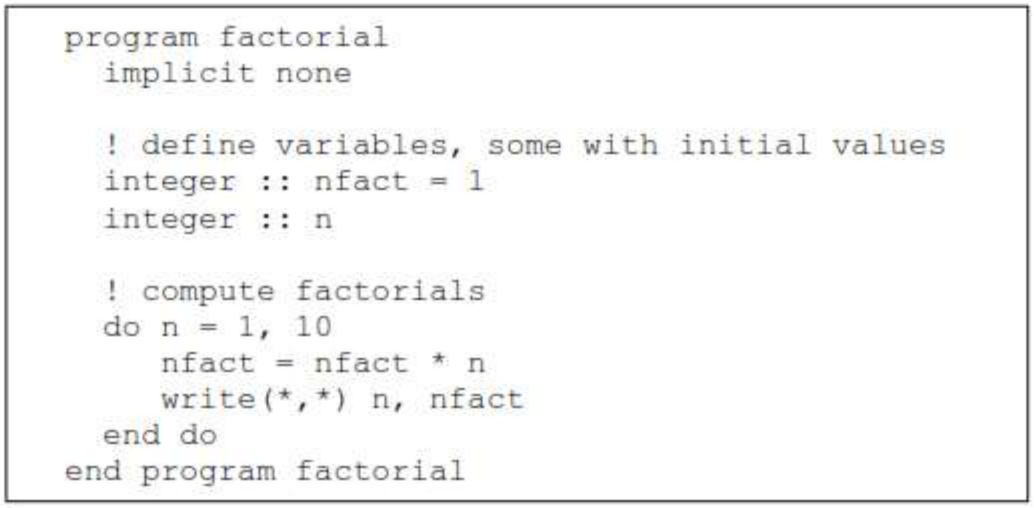
where as before the parts in square brackets are optional.

- *var* is an integer variable
- *start* is the initial value *var* is given
- *stop* is the final value
- *step* is the increment by which *var* is changed. If it is omitted, unity is assumed

The loop works by setting *var* to *start*. If *var* $\leq$ *stop* the statements up to the end do are executed. Then *var* is incremented by *step*. The process then repeats testing *var* against *stop* each time around the loop.

- It is possible for the included statements never to be executed, for instance if *start* > *stop* and *step* is 1.

This program is an example which computes factorials:



```
program factorial
  implicit none

  ! define variables, some with initial values
  integer :: nfact = 1
  integer :: n

  ! compute factorials
  do n = 1, 10
     nfact = nfact * n
     write(*,*) n, nfact
  end do
end program factorial
```

- Modify the factorial program as follows. Change 10 to 100 and insert the following line before the end do.

```
if (n > 10) exit
```

- What output do you get? Why? The exit command terminates the loop.